Fast Einsum Implementation Using Parallelization, Vectorization, and More

Algorithm Engineering 2025 Project Paper

Jonas Peters
Friedrich Schiller University Jena
Germany
jonas.peters@uni-jena.de

Abstract

(1) **Topic and background:** This paper concerns tensor operations, specifically those implemented in the Python library NumPy with its `numpy.einsum` method. Tensor operations are very common in todays data driven world, especially with the rise of *Deep Learning*. Hence, we should ensure that those important methods are implemented efficiently, as *time* and *space complexity* are a big concern when operating on increasingly bigger data.
(2) **Focus:** The einsum operation implements three fundamental tensor operations: *Transposition*, *Reduction* and *Contraction*. This paper tests all of these operations, as well as their combinations, and compares it to a hardcoded implementation written in C++.
(3) **Method:** To this end, I implemented a bare einsum implementation which implements these three operations in C++, where one external library for transposing tensors is employed. Then, both implementations are compared, and the findings are presented in this paper.
(4) **Key findings:** The library einsum implementation does a really good job at transposing tensors. However, when contraction and especially reduction take up much of the workload, NumPy's einsum implementation is remarkably inefficient, as my C++ implementation executes faster by magnitudes.
(5) **Conclusions or implications:** These findings emphasize that we cannot trust every implementation of a library to be implemented efficiently, even when speaking of popular libraries like NumPy. In this specific case of einsum, I advise the reader not to use it for big data operations which rely heavily on contraction or reduction.

Keywords

Tensor Operations, High-Performance Computing, NumPy, Transposition, Reduction, Contraction

1 Introduction

1.1 Background

The topic of this paper is tensor transformations, and how to perform them efficiently. Tensors consist of many dimensions. Take matrices as specialized tensors with two dimensions (rows and columns) for example. When multiplying two matrices, we calculate "row times column" for each position in the resulting matrix. In tensor language, this is equivalent to *contracting* the the second dimension of the first matrix with the first dimension of the second matrix. We can also just rearrange the dimensions in a new order, which is called *transposing*, similar to matrices. Lastly, we can *reduce* entire dimensions by summing over all entries, for example reducing the second dimension of a matrix M is equivalent to

$$M_{reduced} = M \cdot \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \quad .$$

In order to not always hardcode these operations, *einsum* can be employed by passing in the two operand tensors, together with a format string, which defines how every dimension will be treated. For example `ik,kj->ij` specifies a normal matrix-matrix-multiplication. Here is how these *column identifiers* of `lhs_string` in the einsum expression `<lhs_string>,<rhs_string>-><target_string>` are interpreted:

Table 1: Column Identifiers of Einsum Expression

Category	rhs_string	target_string
Batch	Yes	Yes
Kept left	No	Yes
Contracted	Yes	No
Summed left	No	No

You have to read the table in the following manner. Take a column identifier of `lhs_string`. Then you check whether it is present in `rhs_string` and `target_string`, and based on these results you assign it to one of the four specified categories, where "Summed left" describes a dimension to reduce.

A similar table can be produced for column identifiers in `rhs_string`. The column identifiers in `target_string` also specify the resulting shape of the operation. All of the necessary computations to achieve the desired transformation is performed by the einsum function.

1.2 Related Work

In order to implement einsum, it is necessary to transpose tensors. To this end, I used this external C++ library [1].

1.3 My Contributions

In order to have a complete einsum implementation, many steps are necessary, like parsing the input string, transposing and reshaping the operand tensors, reducing them, batch matrix multiply them, and reshape and transpose the resulting matrix again. The details

of all of these steps can be found in section 2. Implementing these individual operations, as well as orchestrating all of them, was part of this project.

1.4 Outline

Next, I present the implemented algorithm in more detail, and explain the design choices (section 2). After that, we use this implementation to do tensor operations on NumPy arrays, and compare the running times with NumPy's einsum implementation (section 3). To this end, we test different einsum calls, some with much emphasize on individual tensor operations like transposing, reducing or contracting, and other calls which require a combination of these operations. Finally, we analyze the findings and draw conclusions (section 4).

2 The Algorithm

2.1 Prerequisites

At this point, the reader should be familiar how einsum categorizes the dimensions of tensors (based on the format string), and how to treat the tensors accordingly. We only focus on two operand and one result tensor, and we only consider the three fundamental tensor operations described above. We store all our tensors in *row-major order*.

2.2 Parsing the Input

Firstly, we need to ensure the inputs are valid. Since we get a generic `PyObject*` as input, this includes checking if we have three objects: a string, and two NumPy arrays of the same data type. This basic parsing is done in `src/einspeed.cpp`.

Next, we have to ensure that the tuple `(std::string, Tensor<T>, Tensor<T>)` is a valid einsum expression. To this end, I implemented `is_valid_einsum_expression` function found in `include/compute_einsum.h`. It checks the following five criteria:

(1) Every character in target_string must be present in either lhs_string or rhs_string.
(2) There are no duplicate characters in lhs_string, rhs_string or target_string.
(3) The size of lhs_string must match the number of dimensions of lhs_tensor. The same for rhs_string and rhs_tensor.
(4) Common column identifiers in lhs_string and rhs_string must correspond to matching dimension sizes in lhs_tensor and rhs_tensor.

In case the tuple passed all these checks, the function won't throw any error, and hence we can now proceed with the function `compute_einsum` knowing we have a valid einsum expression.

2.3 Computing Einsum

2.3.1 Categorizing Dimensions. We now investigate the orchestrating function `compute_einsum` further. The first step is to categorize each dimension of the two input tensors into one of the four categories *batch, kept left, contracted* or *summed left* in case of the left hand side tensor, similarly for the right hand side. We do this by simply looping over all so called *column identifiers* specified by the format string (column identifiers are basically just the individual characters in the string) and checking if this identifier is also present in the `rhs_string` and `target_string`. Based on these checks we assign the categories, see table 1. We do similarly for the right hand side. We store the indices in appropriate arrays, for example, all the dimension indices which are batch dimensions in the `lhs_tensor` will be stored in `batch_dims_lhs`.

2.3.2 Transposing and Reshaping Operand Tensors. Now that we have categorized all the dimensions, we bring both tensors into the canonical form

$$\text{lhs\_tensor}[b, k_l, c, s_l] \ , \ \text{rhs\_tensor}[b, k_r, c, s_r] \quad .$$

One very important detail is that the order of the *batch* and *contracted* dimensions after transposing have the same ordering in the operand tensors. To illustrate this, say we have the einsum expression `einsum("ab,ba->", lhs_matrix, rhs_matrix)`. Naively, there is no need to transpose both matrices, as both of them are in the canonical form (every dimension is a contraction dimension). However, the orderings of these contracted dimensions do not match, hence we do need to transpose a matrix, either the left or the right hand side.

To this end, the algorithm reorders the indices-lists for the right hand side such that the resulting order of column identifiers matches the left hand side. For this purpose, we employ the function `column_identifiers_like`.

Furthermore, it is worth mentioning that there are different ways to transpose tensors. One way is the "lazy" way, where we just alter meta attributes of the tensor, in particular we alter the *strides*, which specify by "how much" we have to multiply an index of a given dimension in order to get the physical index to access the element in the underlying data array. This lazy way of doing it is very fast, as there is no need to move large amount of data. However, there is one big problem: If a tensor has non-canonical strides, we cannot reshape it. This is very bad, because we actually do want to reshape the tensors afterwards. Hence, we need to use the other way of transposing tensors: In he "eager" approach we basically move the data such that it is aligned with the new ordering of the dimensions. This is obviously much more expensive time and space complexity wise, but there is at least one benefit: The data is well aligned for the next operation (after reshaping): The following reduction operation will benefit from a stride access pattern of 1.

2.3.3 Reducing Operand Tensors. Now that we have both operand tensors in the desired shape, we have a fix plan on how to operate on these tensors, which means the following procedure is identical to all possible inputs. Only the reshaping and transposing of the result matrix at the very end needs specific treatment.

We start by reducing the fourth dimension of both operand tensors. This is fairly efficient as we have efficient access patterns as described earlier. After that, the two tensors will have the following form

$$\text{lhs\_tensor}[b, k_l, c] \ , \ \text{rhs\_tensor}[b, k_r, c] \quad .$$

It is worth mentioning that it might be possible to combine all of these operations so far (transposing, reshaping, reducing fourth dimension) into one big one. This will at least save some memory, as now we have three tensors for each operand tensor in memory: The original one which we have to leave untouched, the transposed one, as well as the reduced one. I still decided to implement it this way, as this approach is more modular. Furthermore, I initially implemented

a transpose method myself, but I realized that it was the bottleneck of my implementation. Hence, I decided to use an external library for transposing, such that the performance of my implementation is not limited by a poor implemented transposing function.

Of course we can now free both transposed tensors after the reduction operation, which we also do. However, I figured that it might be beneficial to actually keep some memory, preferably from the bigger tensor, in order to later reuse it for the result of the following batch matrix multiplication operation.

2.3.4 Batch Matrix Multiplication. Now we take care of contracting the appropriate dimensions by performing a batch matrix multiplication. Note, though, that the contraction dimensions are the last one in each operand tensor (the third dimension), and hence we actually perform a *batch matrix matmul transpose* operation, i.e. we treat the rhs operand batch matrices as being transposed. I chose this implementation, as this way we have a stride of one when multiplying individual matrices of a batch both in the lhs and rhs matrix, with respect to the inner most loop. Consequently, this ensures excellent spatial locality and hence we make perfect use of the cache. The result of this operation will be the three dimensional tensor `bmm_result`, with dimension sizes

$$\text{bmm\_result} = [b, k_l, k_r] \quad .$$

As hinted, we might store the result of this operation in previously allocated memory, all provided that the memory region is big enough. Otherwise we do have to allocate new memory. The question whether our available memory is sufficient for the new data depends on on the dimension sizes of the operand tensors and their categorized dimensions: We have

$$\begin{aligned} |\text{lhs\_tensor}| &= b \cdot k_l \cdot c \cdot s_l \\ |\text{rhs\_tensor}| &= b \cdot k_r \cdot c \cdot s_r \\ |\text{bmm\_result}| &= b \cdot k_l \cdot k_r \quad , \end{aligned}$$

and hence the memory will be sufficiently big enough iff

$$c \cdot s_l \geq k_r$$

in case that the lhs tensor is the bigger one of the two (similar if rhs tensor is bigger).

2.3.5 Reshaping and Transposing the Result. Now, there is only one final step: Bringing `bmm_result` in the correct shape specified by the format string. To this end, we have to firstly split the current three dimensions of our result tensor into the individual ones. Hence, we we look at our initial tensors and their column identifiers again to get the dimension sizes.

For example, to split the batch dimension, we check the the batch dimensions of `lhs_tensor` and store their specific dimension sizes in the same order as they are present in the tensor. We do this, as our `bmm_result` tensor has the same ordering of its data with respect to the batch dimension as `lhs_tensor`. Note that we specifically transposed `rhs_tensor` to match the batch and contraction order of `lhs_tensor`.

After doing similar for the remaining two dimensions, we reshape our tensor accordingly. Now we have the result tensor as desired, only with potential wrong ordering of the dimensions. Hence, we need a final transposition. But instead of creating a whole new tensor and doing an expensive transpose call, we are very lazy and just return the current tensor as is, only with accordingly altered strides metadata.

To illustrate this, say we have a matrix a 5x10 matrix M, and we want to transpose it. When accessing $M[i][j]$ we have to calculate the physical index of the row-major order stored matrix. Hence:

$$\text{physical\_index}_M(i, j) = 10 \cdot i + 1 \cdot j \quad .$$

When we now transpose this matrix, we can alter this translation function instead of actually moving the data. In this example:

$$\text{physical\_index}_{M^T}(i, j) = 1 \cdot i + 10 \cdot j \quad .$$

This means that instead of moving data, we transpose the coefficients of this function. These coefficients are stored as *strides* in the metadata of a tensor. NumPy does so as well, and hence we employ this by doing this final transposition the lazy manner by setting these NumPy metadata attributes.

This implementation choice is not "cheating", as NumPy's einsum implementation does the same lazy transposition. Hence, it is only fair to do so myself, as this way we get more meaningful comparisons. To prove my point, run the following lines of Python:

```
import numpy as np
matrix = np.array([[1,2],[3,4]])
print(matrix.strides)
transposed = np.einsum("ab,c->ba", matrix,
    np.array([1]))
print(transposed.strides)
```

You will likely see that the strides got altered simply.

2.4 Transposition Method

Let us now take a closer look at the smaller "puzzle pieces" making up the main `compute_einsum` function, starting with the tensor transposition method. As already discussed, I chose to use an external library for this purpose, and hence there is not much to analyze, as we just pass the control flow to the library. It is important to note though that we allocate memory for another tensor each call, as this is necessary for the library call, as well as ensuring that the original tensor stays constant. Hence the constant specifier in the method declaration.

2.5 Reshaping Method

The implementation of this method is fairly simple. As discussed, when calling this method, we ensure that the tensor has canonical strides, as in this case we only have to change the meta attributes `ndim` and `shapes`. There isn't else much to say, this is a simple and fast method which will run in $O(1)$.

2.6 Reduction Method

This method is not very generic, as it will only reduce the last dimension of the tensor. By doing so, I kept the implementation simple yet it is sufficient for our computations. At the beginning we have to allocate memory for the resulting tensor. This is the first method we look at which has an impact on performance, thus we should be cautious with implementation choices. Hence, we write the following compiler-optimization-friendly code:

```
#pragma omp parallel for simd aligned(data:
    32) aligned(target_data: 32)
for (size_t i = 0; i < target_size; i++) {
    T sum = 0;
    size_t start_idx = i * last_dimension;
    size_t end_idx = start_idx +
        last_dimension;

    for (size_t k = start_idx; k < end_idx;
        k++) {
        sum += data[k];
    }

    target_data[i] = sum;
}
```

Clearly, this code is very efficient. There are two minor problems though: First, there is a potential risk of overflow when summing up the values. Second, we parallelize the outer most loop, which is a wise choice, but in very rare cases when `target_size` equals one, like when reducing a one-dimensional vector, there won't be any parallelization.

To quickly address both issues, let me point out that, first, in the case of an overflow, there is not much that can be done other than throwing an error. However, implementing this would create overhead, and hence, it is ultimately the responsibility of the user to ensure correct computations. Second, we just accept that we are unlucky if such a case happens. There is also no quick fix, as both loops cannot get collapsed into one because of the line `T sum = 0;`, and also we would need to write template specializations for every possible type `T`, as else the compiler won't compile omp reduction clause. In the end, it comes down to keeping the implementation simple (*KISS*).

Despite that, I actually tried to optimize this method by using explicit vectorization in specialized template methods. Interestingly, they don't seem to have any performance benefits compared to the generic implementation, which is probably due to the fact that modern compilers will vectorize the generic implementation as well. The code is pretty much self explanatory, we do employ an efficient `load` instead of a `loadu` vector instruction, as the data will be aligned because of the previous transpose operation. Here is the code for tensors of type `float`:

```
#pragma omp parallel for
for (size_t i = 0; i < target_size; i++) {
    float sum = 0;
    size_t start_idx = i * last_dimension;

    // Use AVX SIMD for vectorized summation
    __m256 vec_sum = _mm256_setzero_ps(); //
        Initialize to zero

    for (size_t k = start_idx; k < start_idx
        + last_dimension; k += 8) {
        // Load 8 values at once
        __m256 vec_vals = _mm256_load_ps(&
            this->data[k]);
        vec_sum = _mm256_add_ps(vec_sum,
            vec_vals); // Add the values
    }

    // Sum the partial results
    sum += vec_sum[0] + vec_sum[1] + vec_sum
        [2] + vec_sum[3]
        +  vec_sum[4] + vec_sum[5] + vec_sum
            [6] + vec_sum[7];

    target_data[i] = sum;
}
```

2.7 Batch Matrix Multiplication

Let us now investigate the final sub-function `batch_matrix_matmul_transpose` found in `include/`. It is basically very straight forward, also note that we treat the rhs operand batch matrices as transposed, which allows for small strides in the inner most loop. We begin by allocating the memory needed for the output, but before allocating new memory, we check if the available memory we reserved is sufficiently large to hold the new data, and if so use it instead. Next, loop over the batch dimension, loop over the rows of lhs tensor, and loop over the rows of rhs tensor (as it is transposed). We collapse these three loops into one, and inside every iteration, which specifies one entry in the output tensor, we perform an inner product between the two last dimensions of the operand tensors at the specified index. In code it looks like this:

```
#pragma omp parallel for simd aligned(
    result_data: 32) aligned(lhs_data: 32)
    aligned(rhs_data: 32) collapse(3)
for (size_t b = 0; b < batch_size; ++b) {
    for (size_t i = 0; i < rows; ++i) {
        for (size_t j = 0; j < cols; ++j) {
            const T* lhs_row = lhs_data + b
                * rows * inner_dim + i *
                inner_dim;
            const T* rhs_col = rhs_data + b
                * cols * inner_dim + j *
                inner_dim;
            T* result_row = result_data + b
                * rows * cols + i * cols;
            T sum = T(); // Initialize sum
                as zero for type T

            #pragma omp simd
            for (size_t k = 0; k < inner_dim
                ; ++k) {
                sum += lhs_row[k] * rhs_col[
                    k];
            }
            result_row[j] = sum;
        }
    }
}
```

We face the **exact** problems we faced at the reduction method, and my responses are the same, we just keep the implementation simple for the sake of demonstration. This function can be manually vectorized as well, however the compiler should do a decent job at optimizing this function, and hence we shouldn't face a big performance penalty. For `float` and `double` vectorized code can be found in specialized template methods below.

2.8 Parameter Specification

As hinted multiple times, my einsum implementation is not very complete but rather specific. For example, NumPy's implementation also supports passing only one tensor as an argument, passing scalars as arguments, or calculating traces just to name a few features.

For the sake of simplicity my implementation only accepts two (multi-dimensional) NumPy arrays of the same data type, and only implements transposition, reduction, and contraction. It is able to return a scalar value though, and there is also a workaround to only passing in one NumPy array: Just pass in `np.array([1], dtype)` as a second operand, and specify the only dimension as a reduction one. This will have the same effect as only passing in the original tensor, and there is also little overhead, since in my implementation I check whether one operand tensor is a scalar with value `T(1)`, and if so skip the batch matrix multiplication.

Also, supported data types are limited by those supported by the external transposition library. Hence, integer value are actually not supported. The only supported data types are `np.float32`, `np.float64`, `np.complex64`, and `np.complex128`.

2.9 Possible Improvements

My implementation lacks some functionality to be used as a library itself. But the implementation could be a good starting point for building one, and because it is written fairly modular, it can be augmented without much effort.

There are several Improvements one could make, but the most obvious ones are:

- Augmenting supported data types. Therefore a new implementation of the transposition method is needed.
- More support for parameters, maybe also tensors of different data types, different operations, etc.
- More optimized main method, more heuristics like only transposing when needed, etc.
- More optimized sub methods, improving parallelization, vectorization by checking for aligned data, etc.
- Utilize GPU for parallelization.

3 Experiments

Now that everything is implemented, we test our einsum implementation and compare it to NumPy's. To this end, I have written `einspeed.py`, which compares both implementations on a variety of test cases focussing on different aspects of einsum calculations.

3.1 Correctness

We start by assessing the correctness. Our implementation seems to operate very precisely, as for small problem sizes both implementations yield the same output. For larger problem sizes, however, there is a growing discrepancy in the entries of the result tensor. This discrepancy can grow arbitrary large for increasing tensor sizes, which is why a relative measure is more meaningful. We calculate it like this:

```
maximum_relative_discrepancy = np.max(np.abs
    ((result - correct) / correct))
```

Note that per construction no entry of `correct` is zero. It turns out that this relative discrepancy stays below a value of approximately `1e-4`, which is very commendable. One might ask where this error arises, and I my answer to that will be that numerical operations of floating point numbers don't form a mathematical *field*. Consequently, the order of operation will have an effect on the final output, and I think that my implementation employs a different order than NumPy's.

Ultimately, however, neither implementation is exactly correct, and my implementation isn't necessarily worse. In fact, on some test cases it was closer to a hard coded answer, but this doesn't have to mean anything, as these computations will suffer from numerical errors as well. Just to give an example, one can test this by running:

```
lhs_vector = np.random.rand(100)
rhs_vector = np.random.rand(100)
print(np.einsum("i,i->", lhs_vector,
    rhs_vector))
print(einspeed.einsum("i,i->", lhs_vector,
    rhs_vector))
print(lhs_vector @ rhs_vector)
```

In conclusion, my implementation seems to operate correctly, and hence we now focus on performance.

3.2 Performance

When running the test cases, is is apparent that NumPy did a really good job at implementing einsum, as it outperforms my multi-threaded implementation on many test cases. I think this is mostly due to the fact that my memory management is not the most efficient as I use a lot of memory, and NumPy probably uses a bit more sophisticated algorithms for the computations as well. Notably, there seems to be only a constant factor that separates my implementation from NumPy's, and this factor is roughly 20.

These findings emphasize that the developers behind NumPy really know what they are doing. Hence, it is no surprise that it is very challenging to implement certain methods more efficient than they have. Still, it is very interesting to deeper understand the mechanism behind some implementations, and to test out the limits of computations myself.

Very interestingly, NumPy's einsum function seems to **really** struggle at big reduction operations, which is very surprising, as they are very simple to implement. This is one of the key findings of this paper. There seems to be a time complexity issue with its implementation, as with growing operand sizes the running time becomes horrendously bad. To give evidence, we execute this einsum expression

```
einsum("abc,def->", lhs_tensor, rhs_tensor)
```

at different operand scales, and compare both implementations. The results are shown in table 2.

4 Conclusions

In this paper, I implemented the einsum function from scratch, in order to evaluate the existing NumPy implementation. Despite the fact that I certainly could have done better, I was pretty satisfied with my work at the end. The fact that my implementation can "keep up" with NumPy's contributed to this. Yet, I couldn't outperform NumPy generally, only in very specific cases. To be honest, that was predictable. Therefore, these findings emphasize the quality of of libraries like NumPy, but they also show that we should be cautious about performance when using these functions for special purposes, like big reduction operations. Based on my results I advise the reader not to use NumPy's einsum implementation for big reduction operations, as this will result in a serious bottleneck. One should hardcode this operation instead, or use a different einsum implementation. For the latter case, one might use the results of this paper as a starting point to build a better einsum implementation.

Table 2: Comparison of einsum running times. Both NumPy's and my einspeed implementation are tested at varying problem sizes. The problem size specifies the number of entries in the two input tensors. Running times are measured in seconds, the speedup describes the fraction $\frac{numpy\_time}{einspeed\_time}$. Measurements were taken on a PC running Ubuntu 24.04 with 32 GB of RAM and an AMD Ryzen 5 5600G CPU (6 Cores, 12 Threads).

problem size	numpy time	einspeed time	speedup
12	$2 \cdot 10^{-5}$	$2.99 \cdot 10^{-4}$	$6.69 \cdot 10^{-2}$
96	$1.26 \cdot 10^{-5}$	$7.68 \cdot 10^{-5}$	0.16
768	$1.84 \cdot 10^{-5}$	$7.49 \cdot 10^{-5}$	0.25
6,144	$5.52 \cdot 10^{-4}$	$7.57 \cdot 10^{-4}$	0.73
49,152	$3.11 \cdot 10^{-2}$	$2.76 \cdot 10^{-3}$	11.25
$3.93 \cdot 10^{5}$	2.13	$1.37 \cdot 10^{-3}$	1,553.64
$3.15 \cdot 10^{6}$	494.44	$9.44 \cdot 10^{-3}$	52,352.81

References

[1] Paul Springer, Tong Su, and Paolo Bientinesi. 2017. HPTT: A High-Performance Tensor Transposition C++ Library. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (Barcelona, Spain) *(ARRAY 2017)*. ACM, New York, NY, USA, 56–62. https://doi.org/10.1145/3091966.3091968