Compiling Tensor Expressions into Einsum

Julien Klaus, Mark Blacher, and Joachim Giesen

Friedrich Schiller University Jena, Germany
{julien.klaus,mark.blacher,joachim.giesen}@uni-jena.de

Abstract. Tensors are a widely used representations of multidimensional data in scientific and engineering applications. However, efficiently evaluating tensor expressions is still a challenging problem, as it requires a deep understanding of the underlying mathematical operations. While many linear algebra libraries provide an Einsum function for tensor computations, it is rarely used, because Einsum is not yet common knowledge. Furthermore, tensor expressions in textbooks and scientific articles are often given in a form that can be implemented directly by using nested for-loops. As a result, many tensor expressions are evaluated using inefficient implementations. For making the direct evaluation of tensor expressions multiple orders of magnitude faster, we present a tool that automatically maps tensor expressions to highly tuned linear algebra libraries by leveraging the power of Einsum. Our tool is designed to simplify the process of implementing efficient tensor expressions, and thus making it easier to work with complex multidimensional data.

Keywords: tensor expressions · einsum · domain specific languages · mathematics of computing.

1 Introduction

Tensors are higher-dimensional generalizations of vectors and matrices. We can think of tensors as multidimensional arrays. Tensors are used in various applications. For example, an RGB-image can be represented by a tensor A_{ijk} with three dimensions, where the RGB-values of the first pixel can be accessed through A_{00k}. In general, computations over such tensors are mostly written in a form that uses summation symbols and access the tensors by indices. Such computations are typically implemented by nesting for-loops. This, however, can be quite inefficient, when compared to highly-tuned libraries for working on tensors, like NumPy [8], PyTorch [13] and TensorFlow [1]. These libraries use a function called *Einsum*. Although Einsum is quite powerful, it requires some knowledge to use it efficiently and correctly. Therefore, we present a transformation of tensor expressions into Einsum expressions, that can be mapped directly to multiple Einsum libraries or backends. For the transformation, tensor expressions are specified in a simple formal language that is close to the language used in textbooks and scientific articles.

In this article, all steps are described on the example of the Tucker decomposition [17]. This decomposition is used to decompose a high-dimensional tensor into a tensor with fewer entries than the original tensor, and a set of matrices. Although this decomposition works with tensors of arbitrary order, we

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36021-3_10

will focus, for simplicity, on tensors of order three, that is decomposing a tensor $A \in \mathbb{R}^{I \times J \times K}$ into a tensor $Z \in \mathbb{R}^{L \times M \times N}$ and matrices $B \in \mathbb{R}^{L \times I}, C \in \mathbb{R}^{M \times J}, D \in \mathbb{R}^{N \times K}$, where $L \ll I, M \ll J$, and $N \ll L$. To obtain such a decomposition, we have to solve the following optimization problem:

$$\min_{Z,B,C,D} \sqrt{\sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{k=1}^{K}\left(A_{ijk} - \sum_{l=1}^{L}\sum_{m=1}^{M}\sum_{n=1}^{N} Z_{lmn}B_{li}C_{mj}D_{nk}\right)^2}.$$

Evaluating the objective function of this problem entails six nested loops. By using highly-tuned backends, the objective function can be computed orders of magnitude more efficiently than the naive implementation.

Related Work. There are already approaches that map tensor expressions into various backends [7,14,15]. But most of them are either only usable for linear algebra expressions [2,11], need additional information about the expressions' variables and parameters [2,16], do not map into backends, but optimize the loops directly [3,4,12], or do not allow unary operations [9,18]. We present a solution that can handle tensors of arbitrary orders, support unary and binary operations, and does not need any additional information about variables and parameters, like symmetries.

2 Understanding Einsum Notation

Einsum notation is a generalization of the Einstein summation convention, introduced by Einstein in 1916 [6]. The Einstein summation convention simply means summing over shared indices. For example, a matrix-vector product $Ab = \sum_j A_{ij}b_j$ is written as $A_{ij}b_j$, when using the Einstein summation convention. Implicitly, the summation is performed over the shared index, in this case j. In contrast to the Einstein summation convention, which sums over shared indices, the output of the operation is explicitly defined in Einsum notation. Since the tensor names are not relevant for the description of the operation, only the indices of the expression are retained. Therefore, in Einsum notation our matrix-vector product can be written simply as `ij,j->i`. It is important to note, that this makes Einsum notation more general than the Einstein summation convention. In Einsum notation we could also write `ij,j->ij`, which describes the operation of elementwise multiplying the vector represented by the second operand with each column of the first operand, resulting in a matrix. As we can use as many indices as we like, describing tensors of arbitrary dimension is possible, which makes Einsum notation a powerful tool. A slightly more complex example, is the partial expression $\sum_l Z_{nml}D_{lk}$ from the Tucker decomposition. This expression calculates the tensor $E \in \mathbb{R}^{N \times M \times K}$ as

$$E_{nmk} = \sum_{l=1}^{L} Z_{nml}D_{lk}.$$

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36021-3_10

In Einsum notation, this operation is written as `nml,lk->nmk`, where `nml` are the indices of Z (left operand), `lk` are the indices of D (right operand), and `nmk` are the indices of E (result).

Users are not accustomed to directly write Einsum expressions, but are used to a language typically used in textbooks. In textbooks, tensor expressions are almost exclusively written in a form that makes sums and multiplications explicit by using indices. Some examples of explicit expressions and their translation into Einsum are shown in Table 1.

Table 1. Example tensor expressions in their Einsum Notation and our Tensor Expression language.

Operation	Explicit expression	Einsum notation
Scalar times vector	`s*a[j]`	`,j->j`
Vector times vector	`a[i]*b[i]`	`i,i->i`
Vector outer product	`a[i]*b[j]`	`i,j->ij`
Matrix times vector	`A[i,j]*b[j]`	`ij,j->i`
Inner product	`sum[i](a[i]*b[i])`	`i,i->`
Batch matrix multiplication	`sum[k](A[b,i,k]*B[b,k,j])`	`bik,bkj->bij`
Marginalization (sum over axes)	`sum[i,l,n,o](A[i,l,m,n,o])`	`ilmno->m`
Mahalanobis distance	`sum[i,j](a[i]*A[i,j]*b[j])`	`i,ij,j->`

Although Einsum is quite flexible, it lacks frequently used element-wise functions like exp and log, and binary operations such as $+, -$ between Einsum expressions. For instance, our example of the Tucker decomposition, also uses the square-root function and the difference of two terms. To overcome these issues, we need a language to describe tensor expressions with the additional operations.

3 A Language for Tensor Expressions

In this section, we describe a simple formal language for explicit tensor expressions, which is close to standard textbook form. We extend a language for linear algebra expressions [11] to tensors, by allowing multiple indices for variables and sums. Thereby, the language becomes powerful enough to cover arbitrary tensor expressions and most classical machine learning problems, even problems not contained in standard libraries like scikit-learn [5]. An EBNF grammar for the language is shown in Figure 1.

The formal language supports the combination of arbitrary tensors with unary and binary operators, as well as numbers. A special operation is the summation operation *sum*, which includes a list of non-optional indices. The list of indices describes the dimensions over which of the underlying tensors are contracted.

In the formal language, the Tucker decomposition reads as

$$\text{sqrt}(\text{sum}[i,j,k]((A[i,j,k] - \text{sum}[n,m,l](Z[n,m,l] * B[n,i] * C[m,j] * D[k,l]))\hat{}2))$$

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36021-3_10

⟨*expr*⟩ ::= ⟨*term*⟩ {('+' | '-') ⟨*term*⟩}
⟨*term*⟩ ::= ['-'] ⟨*factor*⟩ {('*' | '/') ['-'] ⟨*factor*⟩}
⟨*factor*⟩ ::= ⟨*atom*⟩ ['^' ⟨*factor*⟩]
⟨*atom*⟩ ::= *number* | ⟨*function*⟩ '(' ⟨*expr*⟩ ')' | ⟨*variable*⟩
⟨*function*⟩ ::= 'sin' | 'cos' | 'exp' | 'log' | 'sign' | 'sqrt' | 'abs' | 'sum' '[' ⟨*indices*⟩ ']'
⟨*variable*⟩ ::= *alpha*+ ['[' ⟨*indices*⟩ ']']
⟨*indices*⟩ ::= ⟨*index*⟩ {',' ⟨*index*⟩}
⟨*index*⟩ ::= *alpha*

Fig. 1. EBNF grammar for tensor expressions. In this grammar, *number* is a placeholder for an arbitrary floating point number and *alpha* for Latin characters.

A point worth emphasizing is that indices always select scalar entries of tensors, which makes every operation an operation between scalars. Expressions that conform to the grammar from Figure 1 are parsed into an expression tree. An expression tree $G = (V, E)$ is a directed tree, where every node $v \in V$ has a specific label, which can be either an operation, a tensor name, a number, or a list of indices.

Furthermore, we assign to each node its dimension, that is, the order of the tensor, after evaluating the node, described by its indices. For example, a node of dimension i, j, k has the order three. The dimension *dim* is computed recursively for each node $n \in V$ as

$$\dim(n) = \begin{cases} \text{indices of } n & \text{, if n is a variable} \\ \emptyset & \text{, if n is a number} \\ \bigcup_{\text{children of } n} \dim(c) & \text{, if n is an operation node} \\ \bigcup_{\text{children of } n} \dim(c) \setminus \{\text{indices of the sum}\} & \text{, if n is a } \textit{sum} \text{ node} \end{cases}$$

For leaf nodes, the dimension is the index list, for all other nodes, except for the special *sum* nodes, the dimension is the union of the dimension of their children.

Since the *sum* operation removes indices, the dimension of a *sum* node is the union of their childrens' dimension, minus the indices of the *sum* node. For example, the inner summation of the Tucker decomposition `sum[n,m,l](Z[n,m,l] * B[n,i] * C[m,j] * D[k,l])`, has the dimension

$$\underbrace{\{i,j,k,l,m,n\}}_{\text{union of its children dimensions}} \quad \setminus \quad \overbrace{\{l,m,n\}}^{\text{summation indices}} \quad = \{i,j,k\}.$$

Additionally, for each index, we determine its size (number of entries in this dimension), through the tensors, that refer to the index. For example, the tensor Z_{lmn}, has the indices lmn, and so we know that l has the size of Z's first dimension, and analogously for m and n.

An expression tree for the Tucker decomposition is shown in Figure 2. Our task, and the contribution of this paper, becomes to compile expressions that

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36021-3_10

conform to the grammar in Figure 1 into Einsum expressions that can be evaluated efficiently.

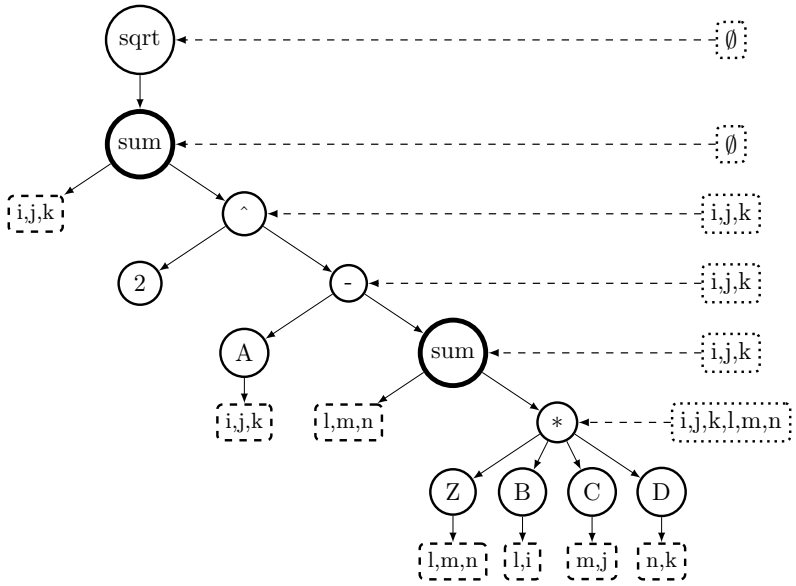

Fig. 2. Expression tree for the Tucker decomposition problem with different node types. Bold nodes indicate sum operations, dashed rectangle nodes denote indices, and all other nodes are either common operations or tensors. For operations and sum nodes, we show the indices (dimension) after performing the operation in dotted rectangles on the right side.

4 Transformation and Compilation into Einsum

The information that we need for the compilation is available in the expression tree. The compilation is performed in a recursive manner, mapping the label of a node to the corresponding library function and then continuing with the children of the node. There are, however, two special cases in the compilation process that cannot be dealt with by simple mappings.

First, at *sum* nodes, we check if the child node is a multiplication node. If this is the case, we combine the summation and multiplication in one operation. For example, during the compilation of our Tucker decomposition when we arrive at the node `sum[l,m,n](Z[l,m,n]*B[l,i]*C[m,j]*D[n,k])`, we can combine the summation and multiplication into

```
einsum('lmn,li,mj,nk->ijk', Z, B, C, D),
```

which is much shorter and easier to read than compiling the sum and multiplication individually

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36021-3_10

```
einsum('ijklmn->ijk',
  einsum('lmn,li,mj,nk->ijklmn', Z, B, C, D)).
```

Second, at binary operation nodes, the dimensions of the two operand tensors have to be equal, because as already mentioned, each binary operation is a pointwise operation, that is only defined on equally shaped tensors. For example, $A_{ijk} - E_{ijklmn}$ is not a valid expression, since A is missing the dimensions lmn. During compilation, we can verify the matching dimensions condition by checking the dimensions of the child nodes. If the dimensions are different, we extend the missing dimensions for each child. In our example, we add the dimensions lmn to node A. Fortunately, this is possible in Einsum notation by using an additional all-ones tensor. The shape of the all-ones tensor is determined, because each of its indices is associated with the shape of some tensor. Thus, we can extend the dimensions of A as follows

```
einsum('ijk,lmn->ijklmn', A, ones(L, M, N)),
```

where `ones(L,M,N)` is an all-ones tensor. The transformation does not change the value of the expression [10]. Compiling the expression of our Tucker decomposition example as shown in Figure 3 gives the following Python code (here with the NumPy backend):

```
np.sqrt(np.einsum('ijk->',(A -
  np.einsum('lmn,li,mj,nk->ijk', Z, B, C, D))**2)).
```

We support the compilation into multiple backends, namely, NumPy, PyTorch and TensorFlow, but more backends can be easily added. For comparing the different backends, we have evaluated the objective function of our Tucker decomposition example on random tensors $A \in \mathbb{R}^{s,s,s}, Z \in \mathbb{R}^{10,10,10}, B \in \mathbb{R}^{10,s}, C \in \mathbb{R}^{10,s}, D \in \mathbb{R}^{10,s}$, with $s \in \{25, 50, 100, 150, 200\}$.

Table 2. Relative speed-up of evaluating the objective function of the Tucker decomposition. The speed-up is computed as $\mathrm{S}_s^{\mathrm{method}} = \mathrm{T}_s^{\mathrm{baseline}} / \mathrm{T}_s^{\mathrm{method}}$, where *method* can be NumPy, TensorFlow or PyTorch and T is the runtime of the *method*.

Backend	$s = 25$	$s = 50$	$s = 100$	$s = 150$	$s = 200$
NumPy	221	248	283	268	278
TensorFlow	31	255	2 083	6 601	16 005
PyTorch	5 777	70 941	191 796	370 973	371 042

For the measurements, we use a machine with an Intel i9-10980XE 18-core processor (36 hyperthreads) running Ubuntu 20.04.5 LTS with 128 GB of RAM. Each core has a base frequency of 3.0 GHz and a max turbo frequency of 4.6 GHz, and supports the AVX-512 vector instruction set. Table 2 shows the relative speed-up of our compiled code using NumPy, TensorFlow and PyTorch, with respect to a baseline implementation with simple nested for-loops. This shows that, our approach can speed up the evaluation of tensor expressions up to four orders of magnitude.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36021-3_10

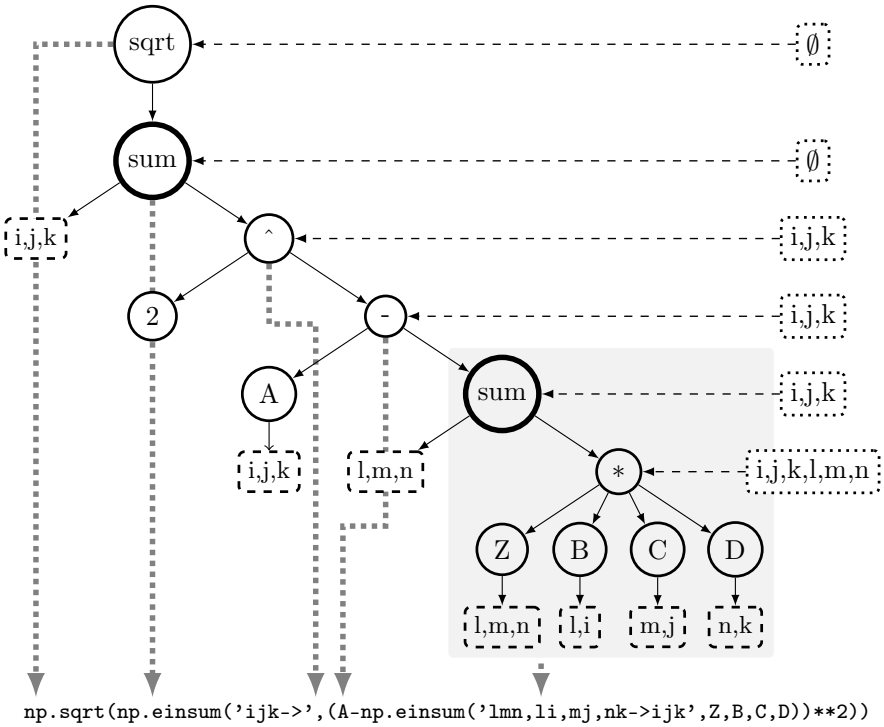

Fig. 3. Expression tree with compiled NumPy code for the Tucker decomposition problem. For each operation node, a dotted gray arrow points to the corresponding part in the code. Note, that the *sum* and child multiplication nodes are merged into one Einsum operation. The Einsum notation for such operations can be read-off from the tree.

5 Conclusion

We have presented a recursive algorithm for compiling tensor expressions into multiple Einsum backends. The compiled expressions evaluate orders of magnitude faster than their straightforward Python implementations using for-loops. To make our approach and its implementation easily accessible, the implementation and the code for the experiments are available at `http://github.com/julien-klaus/tec`.

6 Acknowledgements

This work was supported by the Carl Zeiss Foundation within the project *Interactive Inference* and from the Ministry for Economics, Sciences and Digital Society of Thuringia (TMWWDG), under the framework of the Landesprogramm ProDigital (DigLeben-5575/10-9).

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36021-3_10

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., et al.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015)
2. Barthels, H., Psarras, C., Bientinesi, P.: Linnea: Automatic generation of efficient linear algebra programs. ACM Trans. Math. Softw. (2021)
3. Baumgartner, G., Auer, A.A., Bernholdt, D.E., Bibireata, A., Choppella, V., et al.: Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. Proc. IEEE (2005)
4. Bilmes, J.A., Asanovic, K., Chin, C., Demmel, J.: Author retrospective for optimizing matrix multiply using phipac: a portable high-performance ANSI C coding methodology. In: ACM International Conference on Supercomputing (2014)
5. Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., et al.: API design for machine learning software: experiences from the scikit-learn project. In: ECML PKDD Workshop: Languages for Data Mining and Machine Learning (2013)
6. Einstein, A.: Die Grundlage der allgemeinen Relativitätstheorie. Annalen der Physik (1916)
7. Franchetti, F., Low, T.M., Popovici, D., Veras, R.M., Spampinato, D.G., Johnson, J.R., Püschel, M., Hoe, J.C., Moura, J.M.F.: SPIRAL: extreme performance portability. Proc. IEEE (2018)
8. Harris, C.R., Millman, K.J., van der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., et al.: Array programming with NumPy. Nature (2020)
9. Kjolstad, F., Kamil, S., Chou, S., Lugato, D., Amarasinghe, S.: The tensor algebra compiler. Proc. ACM Program. Lang. **1**(OOPSLA) (2017)
10. Klaus, J.: Visualizing, Analyzing and Transforming Tensor Expressions. Ph.D. thesis, Friedrich-Schiller-Universität Jena (2022)
11. Klaus, J., Blacher, M., Giesen, J., Rump, P.G., Wiedom, K.: Compiling linear algebra expressions into efficient code. In: Computational Science - ICCS 2022 - 22nd International Conference (2022)
12. Nuzman, D., Dyshel, S., Rohou, E., Rosen, I., Williams, K., Yuste, D., Cohen, A., Zaks, A.: Vapor SIMD: auto-vectorize once, run everywhere. In: Proceedings of the CGO 2011 (2011)
13. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., et al.: Pytorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems 32, pp. 8024–8035 (2019)
14. Psarras, C., Barthels, H., Bientinesi, P.: The linear algebra mapping problem. arXiv preprint arXiv:1911.09421 (2019)
15. Sethi, R., Ullman, J.D.: The generation of optimal code for arithmetic expressions. J. ACM (1970)
16. Spampinato, D.G., Fabregat-Traver, D., Bientinesi, P., Püschel, M.: Program generation for small-scale linear algebra applications. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization (2018)
17. Tucker, L.R.: Some mathematical notes on three-mode factor analysis. Psychometrika (1966)
18. Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W.S., Verdoolaege, S., Adams, A., Cohen, A.: Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. CoRR (2018)

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36021-3_10