Compiling Linear Algebra Expressions into Efficient Code

Julien Klaus, Mark Blacher, Joachim Giesen, Paul Gerhardt Rump, and Konstantin Wiedom

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

Abstract. In textbooks, linear algebra expressions often use indices to specify the elements of variables. This index form expressions cannot be directly translated into efficient code, since optimized linear algebra libraries and frameworks require expressions in index-free form. To address this problem, we developed *Lina*, a tool that automatically converts linear algebra expressions with indices into index-free linear algebra expressions that we map efficiently to NumPy and Eigen code.

Keywords: linear algebra · vectorization · domain specific languages · mathematics of computing.

1 Introduction

In textbooks, linear algebra expressions often use indices to access the entries of vectors and matrices, or to sum over certain dimensions. These expressions can be translated directly into loops over indices in the program code, which, however, is often not efficient [1,6]. It is more efficient to map expressions with indices to highly tuned parallel linear algebra libraries like NumPy [9] or Eigen [8]. These libraries expect their input in index-free form. Therefore, in order to use efficient linear algebra libraries, expressions in index form need to be transformed into index-free form. We present an implementation of this approach, that we call *Lina*. *Lina* comprises three parts:

1. A formal input language close to textbook form (Section 2),
2. the transformation from index form into index-free form (Section 3), and
3. mappings to linear algebra libraries and frameworks (Section 4).

The transformation from index form to index-free form is the most challenging part and requires a good understanding of fundamental linear algebra. Consider for example the classical ridge regression problem [13]. Given a feature matrix $X \in \mathbb{R}^{n\times m}$, a label vector $y \in \mathbb{R}^n$, and hyperparameters $\beta \in \mathbb{R}^m$, $\mu, \lambda \in \mathbb{R}$ the ridge regression problem in textbook form reads as

$$\min_{\beta} \quad \sum_{i=1}^{n} \Big(y_i - \mu - \sum_{j=1}^{m} X_{ij}\beta_j \Big)^2 + \lambda \sum_{j=1}^{m} \beta_j.$$

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_2

This expression includes three summation operations, each of which become loops in the implementation. Transforming the expression into index-free form results in

$$\min_{\beta} \quad (y - \mu \cdot \mathbb{1}^n - X^\top \beta)^\top (y - \mu \cdot \mathbb{1}^n - X^\top \beta) + \lambda \cdot \beta^\top \mathbb{1}^m,$$

where $\mathbb{1}^n = (1, 1, \ldots, 1)$ is the all-ones vector. Since the index-free expression does only contain compound linear algebra operations, we can map it directly to linear algebra libraries. However, developers usually do not take the time to formulate their problems in index-free form, although using a highly optimized linear algebra library would lead to a better performance. Here, our focus is on automatically transforming expressions into index-free form. We further optimize the resulting index-free expressions before we map them to Eigen and NumPy routines. An easy-to-use implementation of our approach can be found online at `https://lina.ti2.uni-jena.de`.

Related Work. Various approaches already exist for mapping expressions in index-free form to linear algebra libraries [7,14,15]. Often such methods make use of additional information about the expressions' variables and parameters, for example, symmetry of matrices [2,16]. Also, multiple attempts are known to generate efficient code for expressions in index form [3,4,12]. These approaches are not transforming expressions into index-free form, but directly optimize the loops, for instance by reordering.

2 A Language for Linear Algebra Expressions

In this section, we describe a formal language for extended linear algebra expressions in index form. The notation used in this language is close to MATLAB [10]. It is rich enough to cover most classical machine learning problems, even problems not contained in standard libraries like scikit-learn [5].

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

Fig. 1. EBNF grammar for linear algebra expressions in index form. In this grammar, *number* is a placeholder for an arbitrary floating point number and *alpha* for Latin characters.

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_2

The language supports binary operations as well as unary point-wise operations like log or exp, and of course, variables and numbers. A special operation is the summation operation *sum* that has a non-optional index. This index is used to address elements of vectors or matrices. The full grammar for the language is shown in Figure 1. In this language, the classical ridge regression example reads as

$$\text{sum}[i]((y[i] - \mu - \text{sum}[j](X[i,j] * \beta[j]))^2) + \lambda * \text{sum}[j](\beta[j]).$$

A point worth emphasizing is that the indices always select scalar entries of a vector or matrix. This makes every operation an operation between scalars, which is different in index-free notation, where operations are on compound structures.

Expressions that follow the above grammar are parsed into an expression tree. An expression tree $G = (V, E)$ is a binary tree, where each node $v \in V$ has a specific label. This label can be either an operation, a variable name, a number, or an index. Furthermore, we assign each node a scope, containing indices. For leaf nodes, describing vectors and matrices, the scope contains the associated indices, and for all other nodes, except for the special *sum* nodes, the scope is the union of the scopes of the child nodes. Since the *sum* operation removes an index, the scope of a *sum* node removes an index from the union of their children's scopes. An expression tree for the ridge regression example is shown in Figure 2.

3 Transformation from Index Form into Index-Free Form

The main part of *Lina* is the automatic transformation of expressions from index form into index-free form. In index form expressions, all operations are operations on scalars, whereas in index-free form expressions operations are on compound structures like vectors or matrices. For example, the multiplication $X_{ij} \cdot \beta_j$ multiplies the value of X at index (i, j) with the value of β at index j. We can collect these values into an $(m \times n)$-matrix $(X_{ij} \cdot \beta_j)_{i \in [n], j \in [m]}$. This matrix can be transformed into a point-wise product of two matrices, where X is the first matrix and the outer product $\mathbb{1}^n \beta^\top$ is the second matrix. Indeed, we compute

$$\begin{aligned}
\begin{pmatrix} X_{11} \cdot \beta_1 & \dots & X_{1m} \cdot \beta_m \\ \vdots & \ddots & \vdots \\ X_{n1} \cdot \beta_1 & \dots & X_{nm} \cdot \beta_m \end{pmatrix} &= \begin{pmatrix} X_{11} & \dots & X_{1m} \\ \vdots & \ddots & \vdots \\ X_{n1} & \dots & X_{nm} \end{pmatrix} \odot \begin{pmatrix} \beta_1 & \dots & \beta_m \\ \vdots & \ddots & \vdots \\ \beta_1 & \dots & \beta_m \end{pmatrix} \\
&= \begin{pmatrix} X_{11} & \dots & X_{1m} \\ \vdots & \ddots & \vdots \\ X_{n1} & \dots & X_{nm} \end{pmatrix} \odot \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \cdot \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_m \end{pmatrix}^\top .
\end{aligned}$$

The idea of increasing the dimension of a subexpression by an outer product to enable a point-wise operation works directly for vectors, but not for matrices. In

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_2

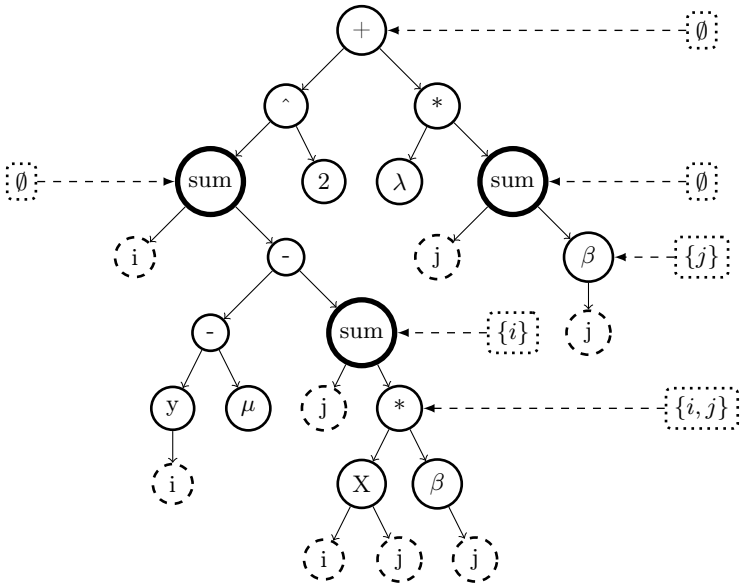

Fig. 2. Expression tree for the ridge regression problem with different node types. Bold nodes indicate sum operations, dashed nodes are indices, and all other nodes are either common operators or variables. For some nodes, we show the scope of the node in dotted rectangles.

these cases, we use the scope assigned to each node. To work with the scopes, we switch to an Einstein like notation. In this notation, each multiplication is represented by a tuple (m, l, r). The tuple (m, l, r) contains the dimension m of the result of the operation, the dimension l of the left, and the dimension r of the right operand. The following equations show how to represent the different multiplication types in Einstein-like notation as well as in linear algebra notation. Let $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^n$, then

$$x^\top \cdot y = x \cdot_{(\emptyset,n,n)} y, \qquad \text{(inner product)}$$
$$x \odot y = x \cdot_{(n,n,n)} y, \qquad \text{(point-wise multiplication)}$$
$$x \cdot y^\top = x \cdot_{(nn,n,n)} y. \qquad \text{(outer product)}$$

The scope of a node and the scopes of its left and right child, respectively, directly correspond to the entries of the tuple. This notation enables us to describe the multiplication type we need during the transformation. The transformation starts at the root and recursively adjust the left and right child of nodes to satisfy their index requirements. If we encounter a node, except a *sum* node, with a left or right child that does not have the same scope, we adjust the scope by adding a multiplication node to the respective subtrees. The new multiplication nodes multiply the old subtrees with an all-ones vectors to supply the missing index. In our example, we have multiplied an all-ones vector with β to supply the

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_2

dimension represented by index i. The *sum* node is special since it removes an index from the scope. This can be accomplished by an inner-product with an all-ones vector. We therefore relabel *sum* nodes as multiplication nodes, and change their left child nodes, which represent an index, into all-ones vectors with the corresponding indices.

In summary, we use outer products to transform binary operations over unequal dimensions into point wise operations over equal dimensions and inner products to reduce a dimension by *sum* operations. The transformed expression tree, for the ridge regression example, is shown in Figure 3. There, we highlight added and adjusted nodes in gray. Note, that *sum* nodes have turned into multiplication nodes. The semantics of product nodes can be decided from their scopes.

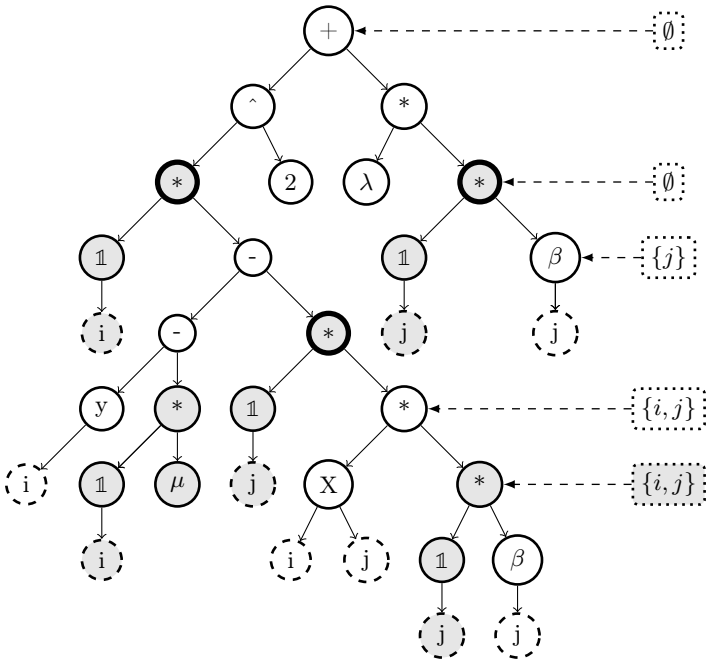

Fig. 3. Expression tree for the ridge regression problem after the transformation into index-free form. Nodes that have been transformed are highlighted in gray. For clarity, we still show the index nodes, although they are no longer needed.

4 Compilation into Multiple Backends

During the transformation from index to index-free form, the nodes of the expression tree are replaced by compound subexpressions, which can make the tree unnecessarily large. Listing 1 shows non-optimized index-free NumPy code.

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_2

Listing 1. Compiled NumPy code for the ridge regression problem given as $\sum_{i=1}^{n}(y_i - \mu - \sum_{j=1}^{m} X_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{m} \beta_j$ without any optimization.

```
np.add(np.sum(np.multiply(np.ones(y_shape[0]),np.power(np.
    subtract(np.subtract(y,np.multiply(m,np.ones(y_shape[0]))
    ),np.dot(np.multiply(X,b),np.ones(X_shape[1]))),2))),np.
    multiply(l,np.sum(np.multiply(np.ones(X_shape[1]),b))))
```

Therefore, before we compile the expression tree into the different backends, we perform various optimizations that reduce the size of the tree. For instance, we perform a default constant folding, and identify common subexpressions, extract them, and link nodes in the tree to the corresponding common subexpression. Furthermore, we exploit broadcasting operations of the individual backends like NumPy or Eigen. For example, the subexpression $y - \mathbb{1}^n \cdot \mu$ in index-free form can be written as $y - \mu$ in most backends. In this case, no extra all-ones vector has to be created, which speeds up the computation and reduces the memory footprint. Since broadcasting operations are different for different backends, we apply them during the compilation into the backends when possible.

Lina is currently compiling into NumPy and Eigen. The compilation traverses the index-free expression tree from the root to the leaves and replaces nodes by methods from the respective backends. At a leaf node, the name of the node is returned, and at a multiplication node the type is determined using the nodes' scope. During the transformation, higher-dimensional tensors may arise that cannot be expressed by matrix-vector operations. There are several approaches to map them efficiently to code [11,17]. For instance, NumPy directly allows us to calculate tensor subexpressions using the `einsum` method. To do so, we use the scope of the node and the scopes of its left and right child to compute an `einsum` string corresponding to the operation. The optimized compiled NumPy code for the ridge regression problem is shown in Listing 2.

Listing 2. Compiled NumPy code for the Ridge regression problem, after optimization.

```
np.sum((y-m-X.dot(b))**2)+l*np.sum(b)
```

5 Conclusion

We have presented *Lina*, a tool for automatically compiling extended linear algebra expressions with indices into efficient linear algebra routines. Our main contribution is the transformation of expressions with indices into index-free form. We map this index-free form to NumPy and Eigen, which exploit the parallelization and vectorization capabilities of the underlying hardware. *Lina*, is available at `https://lina.ti2.uni-jena.de`.

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 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_2

References

1. Ascher, D., Dubois, P.F., Hinsen, K., Hugunin, J., Oliphant, T., et al.: Numerical python (2001)
2. Barthels, H., Psarras, C., Bientinesi, P.: Linnea: Automatic generation of efficient linear algebra programs. ACM Trans. Math. Softw. **47**(3) (2021)
3. Baumgartner, G., Auer, A.A., Bernholdt, D.E., Bibireata, A., Choppella, V., Cociorva, D., Gao, X., Harrison, R.J., Hirata, S., Krishnamoorthy, S., Krishnan, S., Lam, C., Lu, Q., Nooijen, M., Pitzer, R.M., Ramanujam, J., Sadayappan, P., Sibiryakov, A.: Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. Proc. IEEE **93**(2) (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 25th Anniversary Volume. ACM (2014)
5. Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B., Varoquaux, G.: API design for machine learning software: experiences from the scikit-learn project. In: ECML PKDD Workshop (2013)
6. Cai, X., Langtangen, H.P., Moe, H.: On the performance of the python programming language for serial and parallel scientific computations. Sci. Program. **13**(1) (2005)
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 **106**(11) (2018)
8. Guennebaud, G., Jacob, B., et al.: Eigen v3. http://eigen.tuxfamily.org (2010)
9. Harris, C.R., Millman, K.J., van der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M.H., Brett, M., Haldane, A., del Río, J.F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., Oliphant, T.E.: Array programming with NumPy. Nature **585**(7825) (2020)
10. The Mathworks, Inc., Natick, Massachusetts: MATLAB version R2021a (2021)
11. Matthews, D.A.: High-performance tensor contraction without transposition. SIAM J. Sci. Comput. **40**(1) (2018)
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. IEEE Computer Society (2011)
13. Owen, A.B.: A robust hybrid of lasso and ridge regression. Contemporary Mathematics **443**(7) (2007)
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 **17**(4) (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. ACM (2018)
17. 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 **abs/1802.04730** (2018)

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_2