Linnea: A Compiler for Linear Algebra Operations

Henrik Barthels
AICES, RWTH Aachen
barthels@aices.rwth-aachen.de

Abstract

The evaluation of linear algebra expressions is a central part of both languages for scientific computing such as Julia and Matlab, and libraries such as Eigen, Blaze, and NumPy. However, the existing strategies are still rather primitive. At present, the only way to achieve high performance is by handcoding algorithms using libraries such as BLAS and LAPACK, a task that requires extensive knowledge in linear algebra, numerical linear algebra and high-performance computing. We present Linnea, the prototype of a compiler that automates the translation of linear algebra expressions to an efficient sequence of calls to BLAS and LAPACK kernels. Initial results indicate that the algorithms generated by Linnea significantly outperform existing high-level languages by a factor of up to six.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—compilers; G.4 [*Mathematical Software*]

Keywords linear algebra, compiler, code generation

1. Problem and Motivation

Linear algebra expressions appear in fields as diverse as computational biology, signal processing, communication technology, finite element methods, and control theory. While high-level tools are a convenient way to evaluate such expressions, they are usually suboptimal in terms of performance. On the other hand, finding the best algorithm to evaluate a linear algebra expression using low-level libraries is by no means a trivial task. Our goal is to develop a compiler that automatically solves this problem.

Even simple expressions such as $x := AB^{-1}c$ can be evaluated by multiple algorithms, which, while all equivalent in exact arithmetic, can differ greatly in terms of performance and numerical accuracy. For more complex expressions that occur in practice [6], as for example

$$x := \left(A^{-T}B^TBA^{-1} + R^T[\Lambda(Rz)]R\right)^{-1} A^{-T}B^TBA^{-1}y,$$

the number of possible algorithms grows exponentially in the size of the expressions. To obtain efficient algorithms, it is crucial to address the following challenges.

Mapping to Kernels Libraries such as BLAS [7] and LAPACK [4] offer highly optimized functions for basic linear algebra operations (*kernels*). Examples for some of the operations covered by BLAS are

$$y \leftarrow \alpha Ax + \beta y,$$
$$C \leftarrow \alpha AB + \beta C, \text{ and}$$
$$B \leftarrow \alpha L^{-1}B, \text{ where } L \text{ is triangular.}$$

LAPACK offers higher level operations such as matrix factorizations. In order to evaluate a linear algebra expression, it has to be broken down into a sequence of kernels. For all but the simplest expressions, many different sequences exist.

The inverse operator makes the mapping process especially challenging: The explicit inversion of a matrix ($Y := A^{-1}$) is slow and numerically unstable [11, Sec. 13.1.], and should consequently be avoided whenever possible. When the inverse appears in a product with other matrices or vectors, as for example in $x := A^{-1}y$, the explicit inversion can be avoided. To do so, one has to choose between several different options. Depending on the properties of A, one first has to apply a matrix factorization and apply direct solvers to the resulting factors, or using a direct solver for the original expressions. All those choices greatly influence the performance of the resulting algorithm.

While seemingly simple, matrix products pose another special case: Depending on the parenthesization, the cost of evaluating a product can differ by orders of magnitude. This is called the *matrix chain problem* [5].

Linear Algebra Knowledge In practice, matrices frequently have structure (for example symmetry). Libraries offer specialized functions that exploit those properties to speed up computations. Thus, for the mapping to kernels, it is vital to consider matrix properties. However, not only is it necessary to know the properties of the input operands, but also to determine the properties of intermediate operands. To do so, one needs to use knowledge about linear algebra to reason about how matrix properties interact with operations such as multiplication, transposition and inversion.

Furthermore, the laws of linear algebra can be used to rewrite expressions such that they can be computed more efficiently. One example is distributivity, which allows to write $AB + AC$ as $A(B + C)$. Similarly, those laws can

also be used to simplify expressions without actually performing any operations. For example, if A is symmetric, A^T can be simplified to A. Many transformations and simplifications depend on matrix properties. The challenges lies in deciding when and how to apply them, as they are frequently necessary to find optimal algorithms, but an indiscriminate application will increase the search space significantly.

Common Subexpression Elimination The elimination of common subexpressions is a well-known technique in compilers to reduce the number of operations. For linear algebra, the detection becomes more difficult because of identities such as $(AB)^T = B^T A^T$. As an example, only by using such identities, it is possible to detect that $X^T L^{-T} L^{-1} X$ contains $L^{-1}X$ twice. Furthermore, there exist cases where the elimination of a common subexpression does not lead to the optimal algorithm, so it is necessary to establish criteria to decide when the elimination is advantageous.

Our goal is to develop methods and tools that address all these challenges to make it possible to automatically generate algorithms that come close to what a human expert achieves.

2. State of the Art and Related Work

At the moment, when presented with the task of computing linear algebra expressions, one has a range of options. At one end, there are high-level programming languages and environments such as Matlab, Julia, R and Mathematica. Since these languages allow to almost directly describe the mathematical problem, working code can be written within minutes, with little or no knowledge of numerical linear algebra. However, the resulting code (which is possibly numerically unstable[1]) usually achieves suboptimal performance. One of the reasons is that, with the exception of a small number of functions, it is not possible to provide the tools with knowledge about properties. Some functions consider properties by inspecting matrix entries, which could be avoided with a more general method to annotate operands with properties. Furthermore, if the inverse operator is used, an explicit inversion takes place, even if the faster and numerically more stable solution would be to solve a linear system; it is up to the user to rewrite the inverse in terms of operators that solve linear systems, for example "/" or "\" in Matlab [3].

At the other end, there are libraries such as BLAS [7] and LAPACK [4], which offer highly optimized kernels for basic linear algebra operations. However, the translation of a linear algebra expression to an efficient sequence of kernel invocations is a lengthy, error-prone process that requires a deep understanding of both numerical linear algebra and high-performance computing.

In between, there are expression template libraries such as Eigen [10], Blaze [12], and Armadillo [15], which provide a domain-specific language integrated within C++. They offer a compromise between ease of use and performance. The main idea is to improve performance by eliminating temporary operands. While both high-level languages and libraries increase the accessibility, they do not consider common subexpressions, transformations, simplifications and the matrix chain problem. This frequently leads to slow code.

The Transfor program [9] is a first step towards the translation of mathematical descriptions of linear algebra problems (written in Maple) to BLAS kernels. However, it is only applicable to simple expressions as it does not cover the inverse operator. The search-based linear algebra compiler CLAK [8] exclusively uses pattern matching to identify which kernels can be applied. To reduce the size of the search space, kernels are selected according to static priorities, which does not guarantee efficient solutions. LGen solves the problem for small operand sizes, where BLAS and LAPACK do not perform very well, by directly generating C code [17]. The goal of BTO BLAS is to generate C code for bandwidth bound operations, such as matrix-vector products [16].

With Linnea, we aim to combine the advantages of existing approaches: The simplicity, and thus, productivity, of a high-level language, paired with performance that comes close to what a human expert achieves. First results indicate that Linnea is indeed able to substantially outperform existing high-level tools, despite requiring no linear algebra expertise from the user.

$$\begin{aligned} \text{assignments} &\rightarrow \text{assignment}^{+} \\ \text{assignment} &\rightarrow \textit{symbol} := \text{expr} \\ \text{expr} &\rightarrow \textit{symbol} \mid \text{expr} + \text{expr} \mid \text{expr} \cdot \text{expr} \mid \\ &\qquad \text{expr}^{-1} \mid \text{expr}^{T} \mid \text{expr}^{-T} \end{aligned}$$

Figure 1. Grammar describing the definition of expressions.

3. Linnea

Linnea operates in two phases: Symbolic generation of algorithms (Section 3.2), and code generation (Section 3.3). In the first phase, the input expressions (represented as an abstract syntax tree) are broken down into a sequence of kernels. In the second phase, such a sequence is translated to actual code, taking care of memory management and storage formats.

3.1 Input and Output

The input to Linnea is Python code. A simplified, but equally expressive, version of the input grammar is provided in Figs. 1 and 2. The first part (Fig. 1) allows for expressions similar to linear algebra expressions in high-level languages. Unlike those languages, however, it is also possible to annotate operands with properties (Fig. 2). For now, we only consider dense linear algebra.

[1] Some systems compute the condition number for certain operations and give a warning if results may be inaccurate.

$$
\begin{aligned}
\text{definitions} &\rightarrow \text{definition}^{+} \\
\text{definition} &\rightarrow \text{type } \textit{name} \text{ size } \langle \text{property}^{*} \rangle \\
\text{type} &\rightarrow \textbf{Matrix} \mid \textbf{Vector} \mid \textbf{Scalar} \\
\text{size} &\rightarrow (\textit{rows}, \textit{columns}) \\
\text{property} &\rightarrow \textbf{LowerTriangular} \mid \textbf{Orthogonal} \mid \ldots
\end{aligned}
$$

Figure 2. Grammar describing the definition of operands.

The instruction set of Linnea consists of kernels as provided by BLAS and LAPACK. Linnea is built in a way such that the set of kernels can easily be exchanged.

3.2 Symbolic Generation of Algorithms

As shown in Fig. 3, algorithms are generated by constructing a graph, similar to graph search approaches commonly used in the field of artificial intelligence [13]. The nodes of the graph contain the expressions that have yet to be computed. The source node contains the input expressions to Linnea. Edges are annotated with the operation(s) necessary to get from one expression to the next. If an edge is not annotated with any operations, it signifies a symbolic rewriting, without any actual computations happening. A node is *terminal* if it only contains assignments of the form *symbol* := *symbol*, that is, nothing is left to compute. During the derivation, Linnea maintains a set of *active nodes*. Progress is made by applying the different *derivation steps* to the active nodes and thus generating successors for those nodes. The derivation steps are designed in a way such that progress and termination is guaranteed. They are described in Section 3.2.2.

3.2.1 Linear Algebra Knowledge

At the core of the algorithm generation is an engine for the symbolic manipulation of expressions and inference of properties. This engine is built on top of MatchPy [1], which offers efficient associative-commutative many-to-one pattern matching, similar to the pattern matching in Mathematica [2]. Pattern matching is used to identify which kernels can be applied. Thanks to the linear algebra knowledge encoded in this engine, a number of optimizations are possible:

Properties Linnea determines how properties propagate with the application of kernels by making use of linear algebra knowledge. As a trivial example, irrespective of how it is computed, the product of two lower triangular matrices yields another lower triangular matrix. This knowledge is implemented as logical inference rules, for example:

$$
\begin{aligned}
\text{lowerTriangular}\,(A) &\rightarrow \text{upperTriangular}\left(A^{T}\right) \\
\text{Diagonal}\,(A) \wedge \text{Diagonal}\,(B) &\rightarrow \text{Diagonal}\,(AB) \\
A = A^{T} &\rightarrow \text{Symmetric}\,(A)
\end{aligned}
$$

A and B can be arbitrary matrix expressions.

Variants of Expressions Linear algebra expressions can be represented as a *sum of products* (e.g., $AB + AC$) or a *product of sums* (e.g., $A(B + C)$). As a default, we use sums of products, as this representation is unique. However, at each derivation step, Linnea transforms expressions between both representations to generate algorithms based on multiple variants of the same expression.

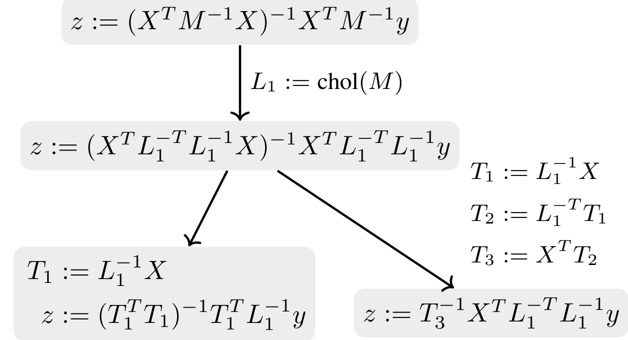

Figure 3. A small part of the derivation graph for $z := (X^T M^{-1} X)^{-1} X^T M^{-1} y$. The full graph has about 100 nodes.

Simplifications Expressions are simplified after each derivation step by applying rules such as:

$$
\begin{aligned}
(AB)^T &\rightarrow B^T A^T & & \\
A^{-1} A &\rightarrow I & & \\
A^T &\rightarrow A & & \text{if Symmetric}(A) \\
Q^T Q &\rightarrow I & & \text{if Orthogonal}(Q) \\
Q^{-1} &\rightarrow Q^T & & \text{if Orthogonal}(Q) \\
\alpha A + \beta A &\rightarrow (\alpha + \beta) A & & \alpha, \beta \text{ are scalars}
\end{aligned}
$$

Since simplifications frequently depend on matrix properties, rewrite rules are annotated with constraints which determine when they can be applied.

3.2.2 Derivation Steps

In the following, we describe the derivation steps currently implemented in Linnea.

Instruction Selection The instruction selection is a fairly complex derivation step. One of the most important objectives during this step is to avoid the explicit inversion of matrices. This is done by applying factorizations to matrices whenever they appear within an inverse. Matrices are only ever inverted explicitly if not avoidable, as for example for the assignment $X := A^{-1}$.

To reduce the size of the search space, Linnea adopts specialized algorithms that take advantage of the structure of (sub-)expressions. For example, the matrix chain algorithm [5] can be used to determine the optimal parenthesization of a product of matrices to minimize the number of floating point operations (FLOPs). We developed a generalized version of this algorithm that allows for matrices to be transposed and inverted, and which takes advantage of properties. A limitation of this approach is that it can only be used

with kernels of the form $f_1(A)f_2(B)$ where f_1, f_2 indicate any combination of transposition, inversion, and identity. A natural alternative, which does not impose any constraints on the kernels, consists in using pattern matching to find all possible sequences of kernels. However, this approach significantly increases the size of the search graph. We plan to investigate this approach in the future.

Common Subexpression Elimination Linnea incorporates an algorithm to detect common subexpressions of arbitrary length that takes into account identities such as $B^{-1}A^{-1} = (AB)^{-1}$ and $B^T A^T = (AB)^T$. As a result, even terms such as $A^{-1}B$ and $B^T A^{-T}$ are identified as a common subexpression. For products, the algorithm uses generalized suffix trees [18]. Since making use of a common subexpression does not necessarily lead to the optimal algorithm, Linnea also continues to work on the unmodified expressions.

Other Transformations Pattern matching enables Linnea to easily detect opportunities where non-trivial transformations can be applied. Those transformations are based on know identities. As an example, $X := A^T A + A^T B + B^T A$ can be rewritten as

$$Y := B + A/2 \qquad X := A^T Y + Y^T A,$$

which allows to compute X with fewer scalar operations.[2]

While such transformations are only applicable to specific cases, thanks to efficient many-to-one pattern matching, searching for a large number of different patterns does not significantly affect the performance of Linnea.

3.3 Code Generation

In the code generation phase, algorithms are translated from the intermediate representation to executable code. As a first target language, we consider Julia, which offers a low level API that exposes the full functionality of BLAS and LAPACK. Additionally, it provides built-in functions that simplify the generation of code with acceptable performance for functionality not part of those libraries, for example kernels operating on diagonal matrices. Since BLAS and LAPACK are available for many architectures and in different languages, Linnea is easily retargetable.

For the translation to code, the specifics of the BLAS and LAPACK interface have to be considered.

Overwriting By construction, algorithms are in static single assignment form. In practice, however, several kernels overwrite one of their input operands. As an example, the GEMM kernel that computes $C \leftarrow \alpha AB + \beta C$ stores the output in the memory location that originally contained C. We perform a liveness analysis to determine when operands can be overwritten. If an operand that will be overwritten is still needed for a different operation, it is copied to another memory location.

[2] For a collection of such identities, see [14].

Storage Formats BLAS and LAPACK use specialized storage formats for matrices with certain properties. As an example, for symmetric matrices, usually only the lower or upper half is stored. As a result, an input operand may be stored in a different format than the one required by the kernel. To deal with this problem, all kernels are annotated with the storage formats of their input and output operands. Linnea automatically inserts code snippets when storage format conversions are necessary.

3.4 Managing the Search Space

An important aspect of Linnea is the exploration of algorithmic alternatives. Due to the complexity of the problem, this can quickly lead to a large search space. Furthermore, it has to be guaranteed that the different derivation steps, transformations and simplifications do not lead to cycles.

Reducing the Size of the Search Space We prune the search graph, using heuristics that consider the complexity of expressions and cost estimations for the algorithms. How different pruning algorithms affect the performance of the compiler and the quality of the results has yet to be investigated. Furthermore, to reduce the size of the derivation graph (without reducing the size of the search space), branches are merged whenever possible. To make this possible, we make sure that different (sub-)expressions are always represented by the same intermediate operands, independently of how they were computed. For example, the results of $AB + AC$ and $A(B + C)$ are both represented by the same operand.

Progress and Termination Most knowledge about linear algebra can naturally be expressed in the form of identities, for example $(AB)^T = B^T A^T$. The difficulty in using such identities for the automatic generation of algorithms lies in the fact it is easy to create infinite loops. To avoid this, we carefully designed the different transformations and simplifications in a way such that infinite loops can not occur. As an example, when writing $(\alpha I + Q^T W Q)$ as $Q^T(\alpha I + W)Q$, the simplifications would undo this transformation. This is avoided by extracting $\alpha I + W$ into a separate assignment. This means that $X := (\alpha I + Q^T W Q)$ becomes

$$Y := \alpha I + W \qquad X := Q^T Y Q.$$

4. Results

We compare algorithms generated by Linnea to the default expression evaluation of Julia. The measurements were performed on an Intel Core i5 with 2,7 GHz. Algorithms were generated with Python 3.6 and executed in Julia 0.5. For each example problem, we use three different implementations: `naive`, `recommended`, and `compiler`. The `naive` implementation is the direct translation of the mathematical problem to Julia. This means that $A^{-1}B$ is implemented as `inv(A)*B`. Since the documentation discourages this use of `inv`, `recommended` uses the operator for the solution of linear systems (`A\B`) instead. `compiler` is the algorithm

#	Example	$s_{C/R}$	$s_{C/N}$	time
1	$b := (X^T X)^{-1} X^T y$	1.73	2.71	37
2	$b := (X^T M^{-1} X)^{-1} X^T M^{-1} y$	2.57	6.17	430
3	$W := A^{-1} B C D^{-T} E F$	2.58	5.15	9
4	$X := A B^{-1} C; Y := D B^{-1} A^T$	1.30	2.31	17
5	$x := W(A^T (A W A^T)^{-1} b - c)$	3.70	4.73	537

Table 1. Example problems with speedups over Julia and compilation time (in milliseconds) of Linnea.

generated by Linnea that performs the fewest FLOPs. The example problems are shown in Tab. 1. Example 1 is a least squares problem, example 2 a generalized least squares problem, both from the field of computational statistics. Examples 3 and 4 are synthetic examples of a matrix chain and common subexpression elimination, respectively. Example 5 is an optimization problem [19].[3,4] Fig. 4 shows the normalized execution times. In all cases, our algorithms outperform both Julia implementations. The speedups of `compiler` over `recommended` are between 1.3 and 3.7 ($s_{C/R}$ in Tab. 1), and between 2.3 and 6.2 over `naive` ($s_{C/N}$). The time it takes Linnea to find the algorithm is in most cases much smaller than half a second. Thus, Linnea could also be integrated in an interactive environment.

5. Contributions

This work introduces Linnea, the prototype of a compiler for linear algebra expressions. Initial results indicate that our approach significantly outperforms traditional evaluation strategies as employed by Julia, and at the same time relieves users from the burden to worry about implementation details. The key insight behind Linnea is that the extensive use of linear algebra knowledge is indispensable to find efficient algorithms, including inference rules for the propagation of matrix properties and algebraic identities to rewrite expressions. In the future, we plan to broaden the scope of Linnea, for example by adding matrix functions such as logarithm and determinant, and integrate Linnea into existing tools such as Julia.

References

[1] MatchPy module. `https://github.com/HPAC/matchpy`, .

[2] Wolfram Language. `http://reference.wolfram.com/`, .

[3] Matlab. `http://www.mathworks.com/help/matlab`, .

[4] E. Anderson, Z. Bai, et al. *LAPACK Users' guide*, volume 9. SIAM, 1999.

[3] Unfortunately, benchmarks for the problem of evaluating linear algebra expressions do not exist. We plan to assemble a list of suitable problems.

[4] Operands have the following sizes and properties: 1) $X \in \mathbb{R}^{1500\times1000}$, $y \in \mathbb{R}^{1500}$ 2) $X \in \mathbb{R}^{1500\times1000}$, $M \in \mathbb{R}^{1500\times1500}$, SPD, $y \in \mathbb{R}^{1500}$ 3) Sizes are (from left to right) 600, 600, 200, 1500, 1500, 800, 1000, A lower triangular, D, E upper triangular. 4) $A, B, C, D \in \mathbb{R}^{1000\times1000}$, B SPD 5) $A \in \mathbb{R}^{3000\times500}$, $W \in \mathbb{R}^{3000\times3000}$, diagonal and positive, $b \in \mathbb{R}^{500}$, $c \in \mathbb{R}^{3000}$

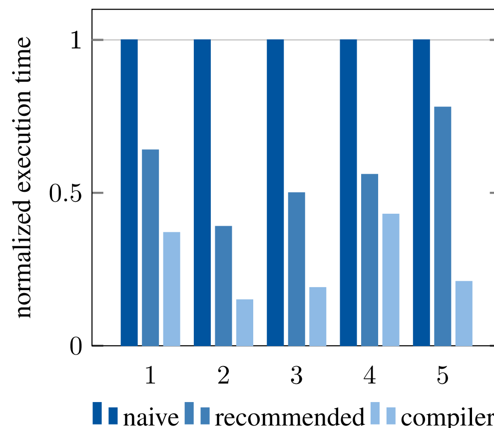

Figure 4. Normalized execution times of the examples.

[5] T. H. Cormen, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill, Inc., 1990.

[6] Y. Ding and I. W. Selesnick. Sparsity-Based Correction of Exponential Artifacts. *Signal Processing*, 120:236–248, 2016.

[7] J. J. Dongarra, J. Du Croz, et al. A set of Level 3 Basic Linear Algebra Subprograms. *ACM TOMS*, 16(1):1–17, 1990.

[8] D. Fabregat-Traver and P. Bientinesi. A Domain-Specific Compiler for Linear Algebra Operations. In *VECPAR 2010*, volume 7851 of *LNCS*, pages 346–361. Springer, 2013.

[9] C. Gomez and T. Scott. Maple Programs for Generating Efficient FORTRAN Code for Serial and Vectorised Machines. *Computer Physics Communications*, 115(2):548–562, 1998.

[10] G. Guennebaud, B. Jacob, et al. Eigen v3. `http://eigen.tuxfamily.org`, 2010.

[11] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 1996.

[12] K. Iglberger, G. Hager, et al. Expression Templates Revisited: A Performance Analysis of the Current ET Methodologies. *SIAM Journal on Scientific Computing*, 34, 2012.

[13] N. J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, 2014.

[14] K. B. Petersen, M. S. Pedersen, et al. The Matrix Cookbook. *Technical University of Denmark*, 7:15, 2008.

[15] C. Sanderson. Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments. 2010.

[16] J. G. Siek, I. Karlin, et al. Build to Order Linear Algebra Kernels. In *International Symposium on Parallel and Distributed Processing*. IEEE, 2008.

[17] D. G. Spampinato and M. Püschel. A Basic Linear Algebra Compiler for Structured Matrices. In *CGO*, pages 117–127, 2016.

[18] G. A. Stephen. *String Searching Algorithms*, volume 3. World Scientific, 1994.

[19] D. Straszak and N. K. Vishnoi. On a Natural Dynamics for Linear Programming. *arXiv preprint arXiv:1511.07020*, 2015.