Linnea: Compiling Linear Algebra Expressions to High-Performance Code

Henrik Barthels
Paolo Bientinesi
barthels@aices.rwth-aachen.de
pauldj@aices.rwth-aachen.de

ABSTRACT

Linear algebra expressions appear in fields as diverse as computational biology, signal processing, communication technology, finite element methods, and control theory. Libraries such as BLAS and LAPACK provide highly optimized building blocks for just about any linear algebra computation; thus, a linear algebra expression can be evaluated efficiently by breaking it down into those building blocks. However, this is a challenging problem, requiring knowledge in high-performance computing, compilers, and numerical linear algebra. In this paper we give an overview of existing solutions, and introduce Linnea, a compiler that solves this problem. As shown through a set of test cases, Linnea's results are comparable with those obtained by human experts.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; **Domain specific languages**; *Very high level languages*; *Source code generation*; • **Computing methodologies** → Linear algebra algorithms;

KEYWORDS

linear algebra, compiler, code generation

ACM Reference format:
Henrik Barthels and Paolo Bientinesi. 2017. Linnea: Compiling Linear Algebra Expressions to High-Performance Code. In *Proceedings of PASCO, Kaiserslautern, Germany., July 2017 (PASCO'17),* 3 pages.
https://doi.org/10.475/123_4

1 PROBLEM AND MOTIVATION

Even simple linear algebra expressions such as $x := AB^{-1}c$ can be evaluated in multiple ways, which we refer to as algorithms. Those algorithms, while all equivalent in exact arithmetic, can differ greatly in terms of performance and numerical accuracy. For more complex expressions that occur in practice [5], as for example

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

the number of possible algorithms grows exponentially in the size of the expression.

When presented with the task of evaluating linear algebra expressions, a range of options exist. At one end of the spectrum, high-level programming languages such as Matlab, Julia, R, and Mathematica, allow users to code almost with the same notation as the mathematical problem, thus producing working code in matter of minutes, with little or no knowledge of numerical linear algebra. At the other end, libraries such as BLAS [6] and LAPACK [3] offer highly optimized kernels for basic linear algebra operations; it is however the user's responsibility to map a linear algebra expression to an efficient sequence of kernel invocations. This task 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 [8], Blaze [9], and Armadillo [11], which provide a domain-specific language integrated within C++, offering increased productivity combined with some performance optimizations. While both high-level languages and libraries increase the accessibility, they do not aggressively optimize for performance.

We present Linnea, a prototype compiler that translates linear algebra expressions to a sequence of BLAS and LAPACK kernel invocations. 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. Initial results indicate that the algorithms generated by Linnea outperform existing high-level languages by a factor of up to six.

Related Work. Three compilers specifically target matrix expressions: CLAK [7], LGEN [13], and BTO [12]. CLAK maps expressions on kernels by relying on pattern matching and prescribed priorities to limit the search space; LGen focuses on operations with small operands, a regime in which most libraries do not achieve high-performance, and directly generates C code; BTO instead targets bandwidth bound operations, such as matrix-vector products, and also generates C code.

2 LINNEA

Linnea takes as input linear algebra expressions, annotated with properties, and maps them onto sequences of kernels as offered by libraries. To this end, Linnea utilizes modules that encode knowledge from linear algebra, numerical linear algebra, as well as high-performance computing; thanks to these modules, Linnea rewrites and simplifies expressions in different ways, reasons about matrix properties, considers the matrix chain problem, and takes advantage of common subexpressions.

As shown in Fig. 1, algorithms are generated by constructing a graph, similar to graph search approaches commonly used in the field of artificial intelligence [10]. 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 transition from one expression to the next.

PASCO'17, July 2017, Kaiserslautern, Germany.
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.475/123_4

$$\begin{tikzcd}
 & z := (X^TM^{-1}X)^{-1}X^TM^{-1}y \arrow[d, "L_1 := \mathrm{chol}(M)"] & \\
 & z := (X^TL_1^{-T}L_1^{-1}X)^{-1}X^TL_1^{-T}L_1^{-1}y \arrow[dl] \arrow[dr, "\substack{T_1 := L_1^{-1}X \\ T_2 := L_1^{-T}T_1 \\ T_3 := X^TT_2}"] & \\
\substack{T_1 := L_1^{-1}X \\ z := (T_1^TT_1)^{-1}T_1^TL_1^{-1}y} & & z := T_3^{-1}X^TL_1^{-T}L_1^{-1}y
\end{tikzcd}$$

Figure 1: Computation of $z := (X^TM^{-1}X)^{-1}X^TM^{-1}y$: a subset of the derivation graph. The full graph has about 100 nodes.

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], a Python module which offers efficient associative-commutative many-to-one pattern matching, similar to the pattern matching in Mathematica [2]. Pattern matching is used to identify the kernels that can be applied. Thanks to the linear algebra knowledge encoded in this engine, a number of optimizations are possible: As a trivial example, irrespective of how it is computed, the product of two lower triangular matrices yields another lower triangular matrix. This makes it possible to select kernels that take advantage of matrix properties. Furthermore, expressions are transformed between different representations, for example a *sum of products* (e.g., $AB+AC$) and a *product of sums* (e.g., $A(B+C)$). Expressions are also symbolically simplified. For example, Q^TQ is replaced with the identity matrix if Q is orthogonal, thus saving unnecessary computations.

The instruction selection is a central part of the derivation. 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 [4] can be used to determine the optimal parenthesization of a product of matrices to minimize the number of floating point operations (FLOPs). Linnea also incorporates an algorithm to detect common subexpressions of arbitrary length that takes into account identities such as $B^{-1}A^{-1} = (AB)^{-1}$ and $B^TA^T = (AB)^T$. As a result, even terms such as $A^{-1}B$ and B^TA^{-T} are identified as a common subexpression. Finally, pattern matching enables Linnea to easily detect opportunities where non-trivial transformations can be applied. As an example, $X := A^TA + A^TB + B^TA$ can be rewritten as $Y := B + A/2;\ X := A^TY + Y^TA$, a simplification that reduces the number of scalar operations.

3 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 generated by Linnea that performs the fewest FLOPs. The example problems shown in Tab. 1 are a combination of real application problems and synthetic problems.[1] Fig. 2 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, it is our future goal to integrate it in an interactive environment.

#	Example	$s_{C/R}$	$s_{C/N}$	time
1	$b := (X^TX)^{-1}X^Ty$	1.73	2.71	37
2	$b := (X^TM^{-1}X)^{-1}X^TM^{-1}y$	2.57	6.17	430
3	$W := A^{-1}BCD^{-T}EF$	2.58	5.15	9
4	$X := AB^{-1}C;\ Y := DB^{-1}A^T$	1.30	2.31	17
5	$x := W(A^T(AWA^T)^{-1}b - c)$	3.70	4.73	537

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

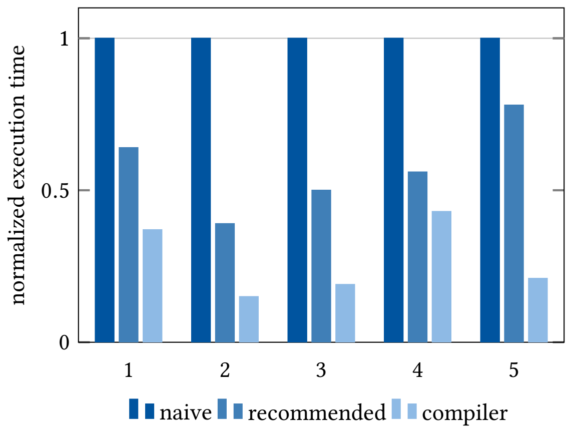

Figure 2: Normalized execution times of the examples.

REFERENCES

[1] MatchPy module. https://github.com/HPAC/matchpy. (????).
[2] Wolfram Language. http://reference.wolfram.com/. (????).
[3] Edward Anderson, Zhaojun Bai, et al. 1999. *LAPACK Users' guide.* Vol. 9. SIAM.
[4] Thomas H. Cormen, Ronald L. Rivest, and Charles E. Leiserson. 1990. *Introduction to Algorithms.* McGraw-Hill, Inc.
[5] Yin Ding and Ivan W. Selesnick. 2016. Sparsity-Based Correction of Exponential Artifacts. *Signal Processing* 120 (2016), 236–248.
[6] Jack J. Dongarra, Jeremy Du Croz, et al. 1990. A set of Level 3 Basic Linear Algebra Subprograms. *ACM TOMS* 16, 1 (1990), 1–17.
[7] Diego Fabregat-Traver and Paolo Bientinesi. 2013. A Domain-Specific Compiler for Linear Algebra Operations. In *VECPAR 2010 (LNCS)*, Vol. 7851. Springer, 346–361.

[1]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, $b \in \mathbb{R}^{500}$, $c \in \mathbb{R}^{3000}$

[8] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org. (2010).
[9] Klaus Iglberger, Georg Hager, et al. 2012. Expression Templates Revisited: A Performance Analysis of the Current ET Methodologies. *SIAM Journal on Scientific Computing* 34 (2012).
[10] Nils J Nilsson. 2014. *Principles of Artificial Intelligence.* Morgan Kaufmann.
[11] Conrad Sanderson. 2010. Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments. (2010).
[12] Jeremy G. Siek, Ian Karlin, et al. 2008. Build to Order Linear Algebra Kernels. In *International Symposium on Parallel and Distributed Processing.* IEEE.
[13] Daniele G Spampinato and Markus Püschel. 2016. A Basic Linear Algebra Compiler for Structured Matrices. In *CGO.* 117–127.