A Compiler for Linear Algebra Operations

Henrik Barthels

AICES, RWTH Aachen, Germany

barthels@aices.rwth-aachen.de

Abstract

In this paper we present a compiler that translates arithmetic expressions containing matrices to efficient sequences of calls to basic linear algebra kernels.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

Keywords Linear Algebra Compiler

1. The Problem

Linear algebra problems appear in fields as diverse as computational biology, finite element methods and control theory. Finding the best way to evaluate a linear algebra expression is by no means a trivial task. Our goal is to develop a compiler that automatically solves this problem. As a first example, consider the assignment $x := AB^{-1}c$, where A and B are matrices, and c and x are vectors. Even for such a simple expression, x can be computed in many different ways, which, while all equivalent in exact arithmetic, can differ greatly in terms of performance and numerical accuracy. We refer to such alternatives as algorithms, and discuss some aspects of how to obtain them in the following.

The first challenge one faces is the inverse operator. In general, the explicit inversion of a matrix ($Y := A^{-1}$) should be avoided, because it is slower and numerically less stable than solving a linear system (see [8, Sec. 13.1.]). A major difference between linear algebra and scalars is the concept of properties of the operands. While they are rarely relevant for scalar operations, they are of particular importance for solving linear systems. For instance, if B is triangular, the linear system can be directly solved. Otherwise, a matrix factorization has to be used. The choice of the most suitable factorization again depends on the properties of B. If B is symmetric and positive definite, the Cholesky factorization can be used. In that case, B is decomposed into the product LL^T, where L is lower triangular. Substituting B in $x := AB^{-1}c$ and, with the help of linear algebra knowledge, symbolically distributing the inverse, $x := AL^{-T}L^{-1}c$ is obtained. No actual computations are performed during this step. The value of x can now be computed by solving two linear systems. Then, another question concerns the parenthesization of the resulting expression. Since the multiplication is associative, for example $(AL^{-T})(L^{-1}c)$ and $A(L^{-T}(L^{-1}c))$ are possible. When working with scalars, the influence of different parenthesizations is often negligible. With matrices, the story is entirely different: In this example, the two parenthesizations require $\mathcal{O}(n^3)$ and $\mathcal{O}(n^2)$ scalar operations, respectively.

It should already be apparent that even for seemingly simple expressions, there are many different algorithms, and some are significantly better than others. For more complex expressions that occur in practice, as for example $x := \left(A^{-T}B^TBA^{-1} + R^T[\Lambda(Rz)]R\right)^{-1} A^{-T}B^TBA^{-1}y$, [4], where matrices have several different properties, the number of possible algorithms grows exponentially.

At the moment, when presented with the task of computing such expressions, one has two contrasting options: (1) High-level programming languages and environments such as Matlab and Julia. Since these languages allow to almost directly describe the mathematical problem, working code can be generated within minutes, with little or no knowledge of numerical linear algebra. However, the resulting code (which is possibly numerically unstable) usually achieves suboptimal performance. (2) Low-level programming languages such as C or Fortran. In that case, it is advisable to use libraries like BLAS [5] and LAPACK [2], which offer highly optimized kernels to compute 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.

We are developing a compiler that aims to offer both: The simplicity, and thus, productivity, of a language like Matlab, paired with performance that comes close to what a human expert achieves.

2. Related Work

High-level languages such as Matlab allow to directly describe linear algebra problems. However, this does not in-

SPLASH Companion'16, October 30 – November 4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4437-1/16/10...
http://dx.doi.org/10.1145/2984043.2998539

clude matrix properties. Only a small number of functions consider properties, in some cases even by inspecting matrix elements.Furthermore, in Matlab, if the inverse operator is used, the inverse is computed explicitly, even if the faster and numerically more stable solution is to solve a linear system; it is up to the user to rewrite the inverse in terms of the slash (/) or backslash (\) operators that solve linear systems [1].

An alternative approach consists in the use of (smart) expression templates in C++, as employed by libraries such as Eigen [7]. The main idea is to improve performance by eliminating temporary operands and provide a domain-specific language integrated within C++. Both in Matlab and Eigen, expressions are evaluated according to fairly simple rules, frequently leading to suboptimal code.

The search-based linear algebra compiler CLAK [6] 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; this approach does not guarantee efficient solutions.

3. The Compiler

Our compiler takes as input linear algebra expressions, annotated with properties, and maps them onto sequences of kernels as offered by libraries. To this end, the compiler utilizes knowledge from linear algebra (for example about properties), numerical linear algebra (e.g. stability), as well as high-performance computing (e.g. performance). Here, we discuss four aspects of the problem.

Properties Many linear algebra operations can be sped up by taking advantage of the properties of the involved matrices. For example, the multiplication of two lower triangular matrices requires $n^3/3$ scalar operations, as opposed to $2n^3$ operations for two full matrices. The most important insight is that one has to track not only the properties of the input operands, but especially the properties of the outputs of the intermediate operations. For this, we developed an engine which uses an algebra of properties to symbolically infer the properties of the outputs. Furthermore, those properties are used to symbolically rewrite and simplify expressions.

Matrix chain problem The problem of finding the parenthesization of a product of matrices that minimizes the number of operations is called *matrix chain problem*, and efficient algorithms exist [9]. To deal with more practical problems, e.g. $X := AB^TC^{-1}D$, we generalized the $\mathcal{O}(n^3)$ dynamic programming matrix chain algorithm [3], such that it takes into account additional operators, properties and arbitrary cost functions.

Common subexpression elimination Modern compilers handle the elimination of scalar common subexpressions very well. Unfortunately, the known approaches do not apply to linear algebra, due to the properties of operators such as transposition and inversion: The terms $A^{-1}B$ and B^TA^{-T} can—and should—be considered as one common subexpression, since $B^TA^{-T} = (A^{-1}B)^T$. Our compiler incorporates an algorithm to detect such common subexpressions of arbitrary length.

Inversion and factorizations In our input language, users express the problem in a way that directly matches the mathematical description. Most important, this includes the inverse operator: Users are not expected to identify linear systems. Instead, the compiler automatically computes expressions containing the inverse by solving linear systems and applying suitable matrix factorizations if necessary.

4. Results and Future Work

We have implemented a prototype of the compiler in Python. As an example for how the compiler proceeds, consider $b := (X^TX)^{-1}X^Ty$. To deal with the inverse operator, in addition to computing X^TX directly, different factorizations are applied to X. One of those is the QR factorization, that is, X is replaced by QR, where Q is orthogonal and R is triangular. After symbolically distributing the transpose operator, $b := (R^TQ^TQR)^{-1}R^TQ^Ty$ is obtained. Then, linear algebra knowledge is used to simplify the expression: Since Q is orthogonal, it holds that $Q^TQ = I$. Further simplifications eventually result in $b := R^{-1}Q^Ty$, for which the matrix chain algorithm finds the best parenthesization. This derivation takes about 0.1 seconds on a modern laptop. For a more complicated generalized least squares problem ($b := (X^TM^{-1}X)^{-1}X^TM^{-1}y$), dozens of (pseudocode) algorithms are generated in about 0.8 seconds.

As next steps, we will implement a code generator for C and extend the set of allowed operations, as well as the set of transformations that can be used.

References

[1] Matlab doc. `http://www.mathworks.com/help/matlab`.

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

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

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

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

[6] 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.

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

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

[9] T. Hu and M. Shing. Computation of Matrix Chain Products. Part II. *SIAM Journal on Computing*, 13(2):228–251, 1984.