Code Generation in Linnea

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

Paolo Bientinesi
Umeå Universitet
Umeå, Sweden
pauldj@cs.umu.se

Abstract

Linnea is a code generator for the translation of high-level linear algebra problems to efficient code. Unlike other languages and libraries for linear algebra, Linnea heavily relies on domain-specific knowledge to rewrite expressions and infer matrix properties.

Here we focus on two aspects related to code generation and matrix properties: 1) The automatic generation of code consisting of explicit calls to BLAS and LAPACK kernels, and the corresponding challenge with specialized storage formats. 2) A general notion of banded matrices can be used to simplify the inference of many matrix properties. While it is crucial to make use of matrix properties to achieve high performance, inferring those properties is challenging. We show how matrix bandwidth can be used as a unifying language to reason about many common matrix properties.

ACM Reference Format:
Henrik Barthels and Paolo Bientinesi. 2019. Code Generation in Linnea. In *6th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, June 22, 2019, Phoenix, AZ*. ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Linnea is a code generator which translates the mathematical description of a linear algebra problem to an efficient sequence of BLAS [5] and LAPACK [2] calls [3, 6]. While high-level languages and libraries such as Matlab [1], Julia [4], Eigen [9], and Armadillo [10] offer a convenient interface to describe linear algebra problems, the resulting code is frequently suboptimal compared to code written in C or Fortran by a human expert. The goal of Linnea is to offer a tool that allows domain specialists to "code" at the same level at which they reason about application problems, and still achieve close to optimal performance. Linnea is written in Python and targets mid-to-large scale linear algebra expressions, where problems are typically compute bound.

The input to Linnea is a mathematical description of a linear algebra problem, similar to the notation in Matlab or Julia. In addition, Linnea also allows to specify properties of the input matrices, for example symmetric, triangular or diagonal. Examples of possible inputs are problems such as

$$H^{\dagger} := H^T(HH^T)^{-1}$$
$$y_k := H^{\dagger}y + (I_n - H^{\dagger}H)x_k,$$

which appears in an image restoration application [11], or $X_{k+1} := X_k + WA^TS(S^TAWA^TS)^{-1}S^T(I_n - AX_k)$, an expression that is part of a randomized matrix inversion algorithm [8].

As output, we decided to generate Julia code because it offers a good tradeoff between simplicity and performance: Low-level wrappers expose the full functionality of BLAS and LAPACK, while additional routines can be implemented easily without compromising performance.

ARRAY '19, June 22, 2019, Phoenix, AZ
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 Storage Formats

Internally, the algorithms generated by Linnea are represented as a symbolic sequence of kernels calls; they still have to be translated to actual code. Most importantly, all operands are represented symbolically, with no notion of where and how they are stored in memory. During the code generation, operands are assigned to memory locations, and it is decided in which storage format they are stored. The BLAS and LAPACK interface introduces some challenges for this phase because kernels commonly overwrite one of their input operands, and because some operands are stored in specialized formats.

Overwriting As an example, the `gemm` kernel $\alpha AB + \beta C$ writes the result into the buffer containing C. Linnea performs a basic liveness analysis to identify if an operand can be overwritten. Only if this is not the case, the operand is copied. At present, Linnea does not reorder kernel calls to avoid unnecessary copies.

Storage Formats Some kernels use specialized storage formats for matrices with properties. As an example, factorizations that output triangular matrices only store the non-zero upper or lower part. Those storage formats have to be considered when generating code: While specialized kernels for triangular matrices only access the non-zero entries, a more general kernel would read from the entire buffer. Thus, it has to be ensured that operands are always in the correct storage format, if necessary by converting the storage format. During the code generation, operands are converted to different storage formats when necessary. To avoid unnecessary conversions, in Linnea we implemented mechanisms to reason about those storage formats.

We define a set of storage formats used by BLAS and LAPACK kernels. As an example, `full` describes operands where all values are explicitly stored in memory, and the `lower_triangular` format is used for lower triangular matrices where only the non-zero part is stored explicitly. `ipiv` is the format used by the LU factorization for the pivoting information, representing a permutation matrix. In the `diagonal_vector` format, only the non-zero entries of diagonal matrices are stored in a one-dimensional vector.

For those storage formats, we define a partial ordering that represents whether one format is compatible with the other. As an example, consider the `trsv` kernel that accesses only the lower (or upper) triangular half of the input matrix A. In this case, we say that `trsv` requires A to be in the `lower_triangular` format. If a matrix that is stored as `full` is passed to `trsv`, no storage format conversion is necessary because `lower_triangular` is compatible with `full`. The reverse does not hold: If a kernel requires a `full` matrix, it is not possible to use one that is stored as `lower_triangular`. However, not all storage formats for matrices with certain properties are compatible at all: One such example are diagonal matrices,

which can be stored as `full` and `diagonal_vector`. As a results, storage format conversions are necessary in both directions.

Finally, we created a collection of functions for storage format conversions. We distinguish between two different types of conversions: In-place and out-of-place. Whenever possible, storage format conversions are done in-place. This may not be possible if 1) the amount of memory used increases, for example when converting from `diagonal_vector` to `full`, or 2) when the conversion would overwrite another operand that is still needed. The latter can happen after factorizations such as LU, when both L and U are stored in the same memory location.

Due to the transitivity of the compatibility relation, it is not necessary to provide conversions for all pairs of formats; if no conversion to the required format is available, a conversion to an even more general format can be used instead.

3 An Algebra of Banded Matrices

Oftentimes, matrices arising in application problems exhibit properties that can be exploited to achieve good performance. However, not only is it important to identify and exploit the properties of the input matrices, it is equally important to keep track of how such properties propagate with the application of kernels. To this end, in Linnea we implemented an inference engine for matrix properties.

The automatic reasoning about matrix properties and how they propagate is challenging for two reasons: First, the relationships between properties are complex. For example, a diagonal matrix can be seen as a special case of both a lower triangular and an upper bidiagonal matrix. Second, for operators with more than one operand, for example addition and multiplication, there is a large number of ways in which the properties of the result matrix are determined by the properties of the input matrices.

The inference can be simplified by representing matrix properties with a unifying formalism: Several properties, including diagonal, triangular and Hessenberg, can fall under the class of banded matrices [7]: A matrix $A \in \mathbb{R}^{m \times n}$ has lower bandwidth l_A if $a_{ij} = 0$ whenever $i > j + l_A$, and upper bandwidth u_A if $j > i + u_A$ implies $a_{ij} = 0$. Intuitively, l_A and u_A specify the number of non-zero off-diagonals below and above the main diagonal, respectively. In the following, we write these bandwidths as a pair (l_A, u_A). In this notation, a diagonal matrix is a banded matrix with bandwidth $(0, 0)$, a lower triangular matrix has bandwidth $(m - 1, 0)$, and an upper Hessenberg matrix is described as $(1, n - 1)$.

We observed that the notion of bandwidth can also be used to infer how matrix properties propagate. For the sum of two matrices $A+B$, the bandwidth can be computed as $(\max(l_A, l_B), \max(u_A, u_B))$. For the product AB with $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$, the bandwidth is $(min(l_A + l_B, m - 1), min(u_A + u_B, n - 1))$. Formula can be derived for operations such as inversion and transposition. This formalism can even be extended to allow for negative bandwidth, covering the case where the main diagonal is zero. With this extension, one can also easily compute the bandwidth of the blocks of a partitioned matrix.

The bandwidth-based notation significantly simplifies the inference of properties because it removes redundancy, and it is eliminates the need to explicitly specify the property resulting from the sum or product of other known properties. Unfortunately, many common operations with banded matrix are not supported by the standard linear algebra libraries.

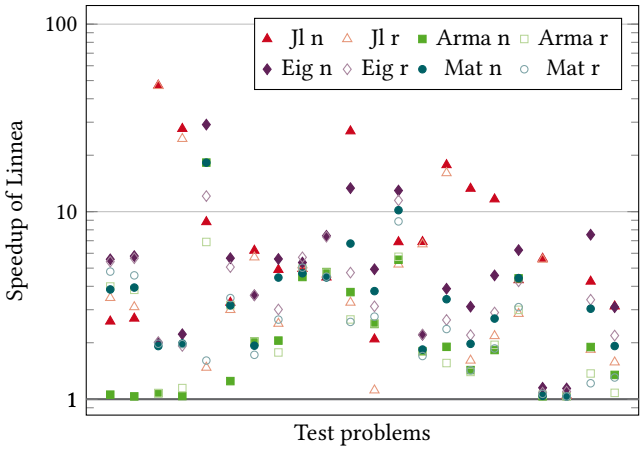

Figure 1. Speedup of Linnea over other languages and libraries for 22 application problems. The problems are sorted by computational intensity (increasing from left to right).

4 Experiments

In Fig. 1, we present the speedup of the code generated by Linnea over Julia, Matlab, Eigen, and Armadillo, for 22 application test cases. For each library and language, two different implementations are used: *naive* and *recommended*. The naive implementation is the one that comes closest to the mathematical description of the problem. It is also closer to input to Linnea. As an example, the naive implementation for $A^{-1}B$ in Julia is `inv(A)*B`. However, since documentations almost always discourage this use of the inverse operator, we also consider a so called *recommended* implementation, which uses dedicated functions to solve linear systems (`A\B`). In all cases, Linnea generates the fastest algorithm.

Acknowledgments

Financial support from the Deutsche Forschungsgemeinschaft (German Research Foundation) through grants GSC 111 is gratefully acknowledged.

References

[1] 2017. Matlab. http://www.mathworks.com/help/matlab.
[2] Edward Anderson, Zhaojun Bai, et al. 1999. *LAPACK Users' guide*. Vol. 9. SIAM.
[3] Henrik Barthels and Paolo Bientinesi. 2017. Linnea: Compiling Linear Algebra Expressions to High-Performance Code. In *Proceedings of the 8th International Workshop on Parallel Symbolic Computation*. ACM, Kaiserslautern, Germany, Article 1, 3 pages. https://doi.org/10.1145/3115936.3115937
[4] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. 2012. Julia: A Fast Dynamic Language for Technical Computing. (September 2012). arXiv:cs.PL/1209.5145
[5] 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.
[6] Diego Fabregat-Traver and Paolo Bientinesi. 2013. A Domain-Specific Compiler for Linear Algebra Operations. In *High Performance Computing for Computational Science – VECPAR 2010 (Lecture Notes in Computer Science)*, O. Marques M. Dayde and K. Nakajima (Eds.), Vol. 7851. Springer, Heidelberg, 346–361.
[7] Gene H. Golub and Charles F. Van Loan. 2013. *Matrix Computations*. Vol. 4. Johns Hopkins.
[8] Robert M. Gower and Peter Richtárik. 2017. Randomized Quasi-Newton Updates Are Linearly Convergent Matrix Inversion Algorithms. *SIAM J. Matrix Analysis Applications* 38, 4 (2017), 1380–1409.
[9] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.
[10] Conrad Sanderson. 2010. Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments. (2010).
[11] Tom Tirer and Raja Giryes. 2017. Image Restoration by Iterative Denoising and Backward Projections. *arXiv.org* (Oct. 2017), 138–142. arXiv:cs.CV/1710.06647v1