The Linear Algebra Mapping Problem. Current State of Linear Algebra Languages and Libraries

CHRISTOS PSARRAS and HENRIK BARTHELS, RWTH Aachen University, Germany
PAOLO BIENTINESI, Umeå Universität, Sweden

We observe a disconnect between developers and end-users of linear algebra libraries. On the one hand, developers invest significant effort in creating sophisticated numerical kernels. On the other hand, end-users are progressively less likely to go through the time consuming process of directly using said kernels; instead, languages and libraries, which offer a higher level of abstraction, are becoming increasingly popular. These languages offer mechanisms that internally map the input program to lower level kernels. Unfortunately, our experience suggests that, in terms of performance, this translation is typically suboptimal.

In this paper, we define the problem of mapping a linear algebra expression to a set of available building blocks as the “Linear Algebra Mapping Problem” (LAMP); we discuss its NP-complete nature, and investigate how effectively a benchmark of test problems is solved by popular high-level programming languages and libraries. Specifically, we consider Matlab, Octave, Julia, R, Armadillo (C++), Eigen (C++), and NumPy (Python); the benchmark is meant to test both compiler optimizations, as well as linear algebra specific optimizations, such as the optimal parenthesization of matrix products. The aim of this study is to facilitate the development of languages and libraries that support linear algebra computations.

CCS Concepts: • Mathematics of computing → Mathematical software performance; • Theory of computation → Complexity classes; • Software and its engineering → Domain specific languages;

Additional Key Words and Phrases: LAMP, linear algebra mapping problem, linear algebra, domain specific languages, compilers

ACM Reference format:
Christos Psarras, Henrik Barthels, and Paolo Bientinesi. 2022. The Linear Algebra Mapping Problem. Current State of Linear Algebra Languages and Libraries. ACM Trans. Math. Softw. 48, 3, Article 26 (September 2022), 30 pages.
https://doi.org/10.1145/3549935

1 INTRODUCTION

Linear algebra expressions are at the heart of countless applications and algorithms in science and engineering, such as linear programming [73], signal processing [21], direct and randomized matrix inversion [14, 41], the Kalman and the ensemble Kalman filter [53, 59], image restoration [74], stochastic Newton method [19], Tikhonov regularization [37], and minimum mean square error filtering [52], just to name a few. The efficient computation of such expressions is a task that...
Table 1. Exemplary Target Linear Algebra Expressions

| Application                  | Expression | Properties |
|------------------------------|------------|------------|
| Standard Least Squares       | $b := (X^T X)^{-1} X^T y$ |            |
| Rand. Matrix Inversion       | $X_{k+1} := X_k + W A^T S (S^T A W A^T S)^{-1} S^T (I_n - AX_k)$ | $W$: SPD |
| Kalman Filter                | $K_k := P_{k-1} H_k^T (H_k P_{k-1} H_k^T + R_k)^{-1}$ | $P$: SPD |
|                              | $P_k := (I - K_k H_k) P_{k-1}$ |            |
|                              | $x_k := x_{k-1} + K_k (z_k - H_k x_{k-1})$ | $R$: SPD |
|                              |            |            |
| Signal Processing            | $x := \left( A^{-1} B^T B A^{-1} + R^T L R \right)^{-1} A^{-T} B^T B A^{-1} y$ | $L$: DI, R: UT |

The properties are as follows. SP(S)D: Symmetric Positive (Semi-)Definite, DI: Diagonal, UT: Upper Triangular.

Table 2. Exemplary Linear Algebra Building Blocks

| Name  | Expression | Description          |
|-------|------------|----------------------|
| DOT   | $\alpha := x^T y$ | inner product        |
| GER   | $A := a x y^T + A$ | outer product        |
| TRSV  | $L x = b$ | triangular linear system |
| GEMM  | $C := \alpha AB + \beta C$ | matrix-matrix product |
| POTRF | $L L^T = A$ | Cholesky factorization |
| SYEVR | $Q^T T Q = A$ | eigendecomposition |

requires a thorough understanding of both numerical methods and computing architectures. To address these requirements, the numerical linear algebra community puts a significant effort into the identification and development of a relatively small set of kernels to act as building blocks towards the evaluation of said expressions. Such kernels are tailored for many different targets, such as computing platforms, matrix properties, and data types, and are often packaged into highly sophisticated and portable libraries, such as OpenBLAS and LAPACK. However, many of the application problems encountered in practice are more complex than the operations supported by those kernels, making it necessary to break the target problem down into a sequence of kernel invocations. The problem we consider in this article is that of computing target linear algebra expressions, such as the ones presented in Table 1, from a set of available building blocks, such as the kernels offered by the BLAS/LAPACK libraries (see Table 2). We refer to this problem as the Linear Algebra Mapping Problem (LAMP).

Solutions to LAMPs\(^1\) range from entirely manual to fully automatic. The manual approach consists in writing a program in a low-level language such as C or Fortran, and explicitly invoking library kernels. While this process might lead to high-performing implementations, it is both time consuming and error prone: It requires users to make decisions about which properties to exploit, which kernels to use and in which order, and all of this while adhering to rather complex APIs. Automated solutions are provided by high-level languages and libraries such as Matlab, Julia, and Armadillo, which allow users to write programs that closely mirror the target linear algebra expressions. It is then a compiler/interpreter that automatically identifies how to map the input program onto the available kernels. The quality of the mapping, and by extension the performance of the resulting code, depends on the specifics of the language\(^2\) of choice, but in general, it will likely be significantly lower than that of a program hand-written by an expert. However, such automatic approaches make it possible even for non-experts to quickly obtain a working program, thus boosting productivity and enabling experimentation. Furthermore, high-level languages give users the

---

1LAMPs stands for the plural of LAMP: Linear Algebra Mapping Problems.
2From now on, we use the term “language” loosely, without distinguishing between programming languages, libraries and frameworks.

ACM Transactions on Mathematical Software, Vol. 48, No. 3, Article 26. Publication date: September 2022.
The Linear Algebra Mapping Problem

opportunity to partially influence how expressions are evaluated, for example by using parenthe-
sization. Some languages even allow for a hybrid approach, offering not only a high-level interface,
but also more or less direct access to the underlying BLAS/LAPACK kernels. Ultimately, the per-
formance of the generated code is closely coupled with the quality of the mapping of the target
expression to the available building blocks, i.e., the solution of the LAMP.

Contributions. With this article we make the following contributions:

1. We introduce LAMP, an umbrella definition that unifies a number of problems related to the
efficient computation of linear algebra expressions; we also prove that the LAMP is at least
NP-complete and provide an extensive overview of the different LAMP solvers.
2. We present a benchmark, consisting of a set of minimal tests, each exposing one single
optimization with regard to solving different LAMP instances.
3. We use our benchmark to assess the capabilities of current state-of-the-art languages that
solve instances of LAMPs. Our intention is not to compare languages with one another, but
to help users and developers understand the capabilities of each individual language.
4. We provide recommendations on how to prioritize optimizations in the context of a LAMP
solver. These recommendations are targeted towards both language developers and end
users who solve LAMPs either using a specialized language or manually (e.g., using direct
calls to BLAS/LAPACK).

The definition of LAMP is not tied to any particular sub-class of linear algebra operations (e.g.,
dense vs. sparse, small scale vs. large scale, linear solvers vs. ...) or a specific underlying hard-
ware configuration (e.g., single-threaded, multi-threaded, distributed, accelerators). However, due
to the complexity of the different aspects of the LAMP, in this article we limit our experiments to
single-threaded, dense linear algebra. In Section 5 we then discuss the associated challenges and
limitations of expanding this study to sparse computations and parallelism.

The organization of the article follows. In Section 2, we define LAMP and discuss its computa-
tional complexity. In Section 3 we survey the landscape of languages, libraries, frameworks, and
tools that solve different instances of LAMPs. In Section 4, we introduce a benchmark of linear
algebra expressions, and use it to evaluate the extent to which high-level programming languages
incorporate optimizations that play a significant role in the solution of LAMPs. In Section 5, we out-
line the challenges associated with extending the current study to multi-threaded environments
and to sparse linear algebra. In Section 6 we offer recommendations to help language developers
and end-users prioritize optimizations when solving a LAMP. Finally, in Section 7 we summarize
our contributions and discuss ways of expanding this study.

2 THE LINEAR ALGEBRA MAPPING PROBLEM

In its most general form, the LAMP is defined as follows. Given a linear algebra expression \( L \), a
set of instructions \( I \), and a cost function \( C \), the LAMP consists of constructing a program \( P \), using
the instructions from \( I \) that computes \( L \) and minimizes the cost \( C(\mathcal{P}) \). Depending on the specific
choice of \( L, I, \) and \( C \), one will recognize that many different, seemingly unrelated, problems are
all instances of the LAMP. A few examples follow.

- When \( L \) is the matrix-matrix product \( C := AB + C \) with variable operand sizes, \( I \) is the set of
machine instructions, and \( C \) is the execution time, the problem reduces to the development
of the high-performance \texttt{GEMM} kernel. This problem is central to many high-performance
linear algebra libraries [22], and significant effort is put both into manual solutions such as
GotoBLAS [39], OpenBLAS [82], and BLASFEO [33], as well as with auto-tuned libraries
such as ATLAS [79].
• When $L$ consists of a matrix product $X := M_1 M_2 \cdots M_k$, the only available instruction in $I$ is the matrix product $C := A B$, and the cost function counts the number of floating point operations, the LAMP reduces to the matrix chain problem [20]. Several variants of this problem have been studied, including finding solutions for parallel systems [56] and GPUs [60].

• When $L$ contains small-scale, memory bound problems, and $I$ consists of scalar and vectorized instructions, the LAMP covers the domain of code generators such as BTO BLAS [69], which aims to minimize the number of memory accesses, as well as LGen [72] and SLinGen [71], which instead minimize execution time.

• When $L$ consists of BLAS-like operations, such as matrix inversion, least-squares problems, and the derivative of matrix factorizations [35, 70], $I$ contains BLAS/LAPACK kernels, and $C$ is a performance metric, the LAMP captures problems solved by the FLAME methodology [14, 29].

• When $L$ is made up of matrix expressions as those shown in Table 1, $I$ contains kernels as those shown in Table 2, and the cost is execution time, the LAMP describes the problem that languages such as Matlab aim to solve. This class of LAMP instances is the main focus of this article.

While execution time is the most commonly used performance metric, all practical solutions to the LAMP also have to fulfill requirements regarding numerical stability. This means that in practice the cost function is a multi-level metric, e.g., a tuple in which the first entry is a measure of numerical stability, and the following ones are performance metrics such as execution time and data movement.

2.1 Complexity of the LAMP

As evinced by the large number of languages and libraries that solve LAMPs, finding a potentially suboptimal solution is not especially challenging. However, as we show in this section, finding the optimal solution is difficult for many variants of LAMPs. Specifically, any variant of the LAMP that makes it possible to have common subexpressions is at least NP-complete. Our proof hinges on the NP-completeness of the Optimal Common Subexpression Elimination problem (OCSE), since the optimal solution of the LAMP requires the solution of OCSE.

**Definition 2.1 (Optimal Common Subexpression Elimination).** Let $D$ be a set, $\bullet : D \times D \to D$ be an associative-commutative operator, and $A$ a finite set of variables over $D$. Consider (i) a collection of equations $x_k = a_1 \bullet \ldots \bullet a_l$, with $a_1, \ldots, a_l \in A$, and $k = 1, \ldots, n$, where each variable appears at most once per equation, and (ii) a positive integer $\Omega$. Is it possible to find a sequence of assignments $u_i = s_i \bullet t_i$, with $i = 1, \ldots, \omega$ and $\omega \leq \Omega$, where $s_i$ and $t_i$ are either an element of $A$ or $u_j$ with $j < i$, such that for all $k$ there exists a $u_i$ which equals $x_k$?

Intuitively, given a set of assignments that contain common subexpressions, the problem consists in computing the assignments with as few operations as possible. An instance of OCSE (left) and its solution (right) are given below:

\[
\begin{align*}
A &= \{a_1, a_2, a_3, a_4\} & u_1 &= a_1 \bullet a_2 = x_1 \\
 x_1 &= a_1 \bullet a_2 & u_2 &= a_2 \bullet a_3 \\
x_2 &= a_1 \bullet a_2 \bullet a_3 & u_3 &= a_1 \bullet u_2 = x_2 \\
x_3 &= a_2 \bullet a_3 \bullet a_4 & u_4 &= a_4 \bullet u_2 = x_3 \\
\Omega &= 4
\end{align*}
\]
This example contains two common subexpressions: \(a_1 \cdot a_2\) (which appear in \(x_1\) and \(x_2\)), and \(a_2 \cdot a_3\) (which appear in \(x_2\) and \(x_3\)). Since in \(x_2\) they overlap, it is not possible to make use of them both. In this case, using either one leads to a solution, but in general the difficulty of OCSE lies in deciding which common subexpressions to use to minimize the number of assignments \(u_i\). Since the definition of OCSE only requires one associative-commutative binary operator, the problem arises in many areas: The set \(D\) can be the set of integers, real or complex numbers, but also vectors or matrices. The operator can either be addition or multiplication, with the exception of matrix multiplication, as it is not commutative.

We prove that OCSE is NP-complete by reduction from Ensemble Computation (EC) [34], which is known to be NP-complete. By showing that for every instance of EC there is an equivalent instance of OCSE, we show that OCSE is at least as difficult as EC. The definition of EC is provided below.

**Definition 2.2 (Ensemble Computation).** Consider (i) a collection \(C = \{C_k \subseteq A \mid k = 1, \ldots, n\}\) of subsets of a finite set \(A\), and (ii) a positive integer \(\Omega\). Is there a sequence \(u_i = s_i \cup t_i\) for \(i = 1, \ldots, \omega, \omega \leq \Omega\), where \(s_i\) and \(t_i\) are either \(\{a\}\) for some \(a \in A\), or \(u_j\) for some \(j < i\) and \(s_i \cap t_i = \emptyset\), such that for all \(C_k \in C\) there is a \(u_i = C_k\)?

The idea of EC is to construct a collection of subsets \(C_k\) of a set \(A\) with as few binary unions as possible. For those unions, one either has to use singleton sets \(\{a\}\) with \(a \in A\), or intermediate results from previous unions. The similarity to OCSE lies in the challenge to optimally make use of subsets that the different \(C_k\) have in common.

The NP-completeness of OCSE is demonstrated in two steps: First, we show that OCSE is in NP by showing that its solutions can be verified in polynomial time. Then, we show that it is possible to reduce EC to OCSE in polynomial time.

**Proof.** **Verification:** A solution to OCSE can be verified in polynomial time by traversing the sequence \(u_i = s_i \cup t_i, i = 1, \ldots, \omega\), collecting the sets of all variables that contribute to each \(u_i\), and comparing those sets with the right-hand sides of the \(n\) input equations \(x_k = \ldots\) with \(k = 1, \ldots, n\).

**Reduction.** For each instance of EC, an equivalent instance of OCSE is obtained as follows. For each \(C_k\), an input equation is constructed as \(x_k = a_1 \cdot a_2 \cdots a_t\) with all \(a_1, \ldots, a_t \in C_k\). In the solution, the sets \(\{a_i\}\) are substituted with the corresponding variables \(a_i\), and the unions \(u_i = s_i \cup t_i\) with operations \(u_i = s_i \cdot t_i\).

We conclude with the EC instance (left) and its solution (right) that correspond to the OCSE instance shown above:

\[
A = \{a_1, a_2, a_3, a_4\} \quad u_1 = \{a_1\} \cup \{a_2\} = C_1
\]
\[
C = \{\{a_1, a_2\}, \{a_1, a_2, a_3\}, \{a_2, a_3, a_4\}\} \quad u_2 = \{a_2\} \cup \{a_3\}
\]
\[
\Omega = 4 \quad u_3 = \{a_1\} \cup u_2 = C_2
\]
\[
\omega = 4 \quad u_4 = \{a_3\} \cup u_2 = C_3
\]

Common subexpressions are not the only reason why variants of the LAMP can be at least NP-complete. Since the LAMP bears similarities to code generation for scalar code, results carry over. For example, in practice the amount of available memory is limited. Thus, it could be important to identify if a given sequence of kernels can be computed with a certain amount of memory, under the assumption that kernels can be reordered as long as the data-flow dependencies are satisfied. It is possible to show that this problem is NP-complete by reduction from Register Sufficiency [34, App. A11.1].
Register Sufficiency is the problem of identifying whether a given program which is described in terms of a dependency graph can be computed with at most \( k \) registers. The input to the Register Sufficiency problem is a directed acyclic graph that represents the data-flow dependencies between the instructions of the program. The problem then consists of finding an ordering of the nodes (the instructions) that satisfies the dependencies which can be computed with at most \( k \) registers.

Similarly, one could also reduce the Register Allocation problem [18] to the LAMP. Given a fixed sequence of instructions, Register Allocation is the problem of finding an optimal assignment of variables to registers that minimizes the cost of loads and stores. The role of the registers in the Register Allocation problem is played by the cache in the case of the LAMP.

3 RELATED WORK

A considerable number of languages, libraries, frameworks, and tools are available for the solution of different instances of LAMPs. In this section, we highlight those that support a high-level notation for linear algebra expressions and provide some degree of automation in the construction of efficient solutions. Furthermore, we survey a number of kernel libraries, which offer the necessary building blocks for higher-level LAMP “solvers”.

3.1 Languages

Several languages and development environments have been created for scientific computations. Matlab [5] is a popular language with extensions (toolboxes) for many scientific domains. GNU Octave [24] is open source software which supports similar functionality and syntax to Matlab. Julia [13] is a rapidly emerging language; it features just in time compilation, and uses a hierarchical type system paired with multiple dispatch. While the main focus of the R language [67] is on statistics, it also supports linear algebra computations. Further examples of computer algebra systems that natively support linear algebra are Mathematica [81] and Maple [4]. All these languages provide mechanisms that help solve certain instances of LAMPs.

3.2 Libraries

For virtually every established high-level programming language, libraries for linear algebra computations exist. The idea is usually to offer a domain-specific language for linear algebra within the host language, usually by adding classes for matrices and vectors, by overloading operators and in the case of C++, by expression templates. Expression template libraries for C++ include: Eigen [43], Blaze [50], Armadillo [68], HASEM [2], MTL4 [40], uBLAS [78], and blitz++ [77]. They offer a compromise between ease of use and performance. Similar libraries exist for many other languages; examples include NumPy [45] for Python and the Apache Commons Mathematics Library [1] and ND4j [25] for Java. By virtue of these libraries, users of general purpose programming languages are exposed to some of the LAMP solving functionality that is available in linear algebra targeted languages.

3.3 Tools and Algorithms

The Transform program [38] is likely the first translator of linear algebra expressions (written in Maple) into BLAS kernels. More recently, several other solutions to different variants of LAMPs have been developed. CLAK [28] and its successor, Linnea [11], are tools that receive a linear algebra expression as input and produce as output a sequence of calls to BLAS and LAPACK that compute the input expression. The Formal Linear Algebra Methods Environment

---

\[3\] Notice that whether or not Register Allocation is NP-complete depends on the exact definition of the problem; the details are discussed in [17].
(FLAME) [15, 44] is a methodology for the derivation of algorithmic variants for BLAS-like operations and for equations such as triangular Sylvester and Lyapunov; Cl1ck [26, 27] is an automated implementation of the FLAME methodology. The goal of BTO BLAS [69] is to generate C code for bandwidth bound operations, such as fused matrix-vector operations. DxTer [57] uses domain knowledge to optimize programs represented as dataflow graphs. LGen [72] targets basic linear algebra operations for small operand sizes, a regime in which BLAS and LAPACK do not perform very well, by directly generating vectorized C code. SLinGen [71] combines Cl1ck and LGen to generate code for more complex small-scale problems. The generalized matrix chain algorithm [10] is an extension of the standard matrix chain algorithm [20]: it finds the optimal solution (in terms of FLOPs) for matrix chains with operands that can be transposed or inverted, and considers matrix properties. LINVIEW [58] introduces techniques for incremental view maintenance of linear algebra.

3.4 Kernel Libraries

Kernels are highly optimized routines that perform relatively simple operations, and that allow more complex algorithms to be structured in a layered fashion. In numerical linear algebra, the BLAS specification was introduced to standardize vector [55], matrix-vector [23], and matrix-matrix [22] operations, and to assist the development of highly optimized libraries. Several libraries offer optimized BLAS implementations, including GotoBLAS [39], OpenBLAS [82], BLIS [76], BLASFEO [33], cBLAST [61], and LIBXSMM [46].

Built on top of BLAS, kernels for more complex operations (e.g., solvers for linear systems, least-squares problems, and eigenproblems) are offered in LAPACK [8], libflame [75], and RELAPACK [64]. Proprietary kernel libraries that implement a superset of BLAS and LAPACK include Intel MKL [3], Nvidia cuBLAS [62], IBM ESSL [51], and the Apple Accelerate Framework [9].

Similar libraries exist for sparse computations, including PSBLAS [30], clSparse [42], HSL (formerly the Harwell Subroutine Library) [48], and PETSc [6].

4 A BENCHMARK TO EVALUATE PROGRAMMING LANGUAGES

Several programming languages make it possible for users to input linear algebra expressions almost as if they were writing them on a blackboard. For instance, in Matlab/Octave, Armadillo, and Julia, the assignment $C := AB^T + BA^T$ can be written as $C = A*B’ + B*A’$, $C = A*trans(B) + B*trans(A)$, and $C = A*transpose(B) + B*transpose(A)$, respectively. When using such a level of abstraction, users relinquish control on the actual evaluation of the expressions, effectively relying on the internal mechanisms of the language to solve the LAMP.

In this section, we consider seven such languages—Armadillo, Eigen, Julia, Matlab, NumPy (Python), GNU Octave, and R⁴—and introduce a benchmark to assess how efficiently they solve a number of test expressions. These expressions were designed to be as simple as possible, while capturing, in isolation, scenarios that occur frequently in practice and for which one specific optimization is applicable. The results, in terms of execution time, are compared to an “expert” implementation, written either in the same language or in C. We use timings to determine whether or not a language implements a specific optimization; timings are not intended to be used to rank the different languages in terms of (absolute) speed. Ultimately, this section is meant to evaluate the quality of the solutions provided by the languages that solve LAMPS, thus inspiring and guiding their development.

⁴This is by no means an exhaustive list of languages that offer a high-level API for linear algebra; others exist (e.g., Mathematica and Maple). In our experience, the languages considered are among the most commonly used for numerical computations and data analysis applications.
4.1 Setup

Our benchmark consists of 13 experiments, each one of them containing one or more test expressions, to be used as input to the languages. In all cases, the input programs (expressions) resemble the mathematical representation as closely as possible. Consequently, whenever an operation is supported by both a function and an operator, the latter is preferred (e.g., for matrix multiplication, NumPy supports both the function `matmul` and the operator `@`). Furthermore, the input expressions are as compact as possible, that is, not broken into multiple assignments and without explicit parenthesization. In addition, all matrices (input and output) are preallocated and initialized before any timing.

Each experiment is run using fixed size operands (see Appendix B.1). We chose the size of the operands to be large enough so that: a) the problem is compute bound, b) the individual timings are less susceptible to noise and fluctuations, and c) the operand sizes resemble the ones found in the actual applications listed in Appendix A. We postulate that in principle languages could make different choices for different problem sizes, especially when dealing with very small (bandwidth bound) problems as opposed to compute bound ones. The observed execution time for all experiments is listed in Appendix B, unless already presented in Section 4.

For each experiment, we report the minimum execution time over 20 repetitions, flushing all cache memories in between each repetition. Special measures are taken to avoid dead code elimination in both the experiments and cache flushing. For those languages that have a garbage collector (Julia, Python, and R), we explicitly invoke it after cleaning the cache to reduce the chances of interference with our timings. Furthermore, we do not concern ourselves with how much time it takes for languages to make decisions; this decision making process for some languages could be part of compilation, interpretation, Just In Time (JIT) compilation, or any other mechanism supported by the specific language. Instead, we evaluate the quality of those decisions and assess whether or not a specific optimization is implemented.

The experiments are performed on a single core of a Linux machine with an Intel Xeon E5-2680V3 processor, with Turbo Boost disabled. All languages are linked to the Intel Math Kernel Library (MKL) 19.0, which implements a super-set of BLAS and LAPACK, and compiled with gcc.\(^5\) The versions of the languages used are the latest stable releases as of December 2020: Armadillo 10.1.x, Eigen 3.3.8, Julia 1.5.2, Matlab 2020a, GNU Octave 5.2.0, NumPy 1.19.4, and R 4.0.3. The source code for the experiments is available online.\(^6\)

4.2 Mapping to Kernels

BLAS and LAPACK offer a set of kernels that are the de-facto standard building blocks for linear algebra computations. Offered by a variety of developers and vendors (both open source and proprietary, see Section 3.4), these kernels are considered to provide the best available implementations for the operations they support, in terms of performance. All the aforementioned languages have access to optimized kernels via MKL. Here we investigate the capabilities of modern linear algebra languages in mapping fundamental operations to BLAS kernel calls.

4.2.1 Experiment #1: GEMM.

Input. In this first experiment, we initialize the random matrices \(A \in \mathbb{R}^{m \times k}, \ B \in \mathbb{R}^{k \times n}\), and \(C \in \mathbb{R}^{m \times n}\), and we input the expression \(C := AB\) in each language by using the available matrix representations (objects) and the operator for matrix multiplication. The goal is to determine whether languages compute this expression by invoking the optimized BLAS kernel `GEMM`, or via another (inferior) implementation. The `GEMM` kernel included in the optimized BLAS libraries is an

---

\(^5\)Version 8.2.0 with optimization flag `-O3`.
\(^6\)https://github.com/HPAC/LAMP_benchmark.
extremely sophisticated piece of code [39]; consequently, the difference in performance between a call to GEMM and to any other (suboptimal) implementation is going to be significant and easily distinguishable by comparing the execution time with that of an explicit call to GEMM implemented in C, henceforth referred to as “reference”.

**Results.** Table 3 shows the execution time for each language to perform the matrix product. The expectation is that if the timings are “close enough” to the reference, then it can be inferred that the languages do rely on the GEMM kernel, modulo some overhead. The timings indicate that all languages are within 15% of the execution time of the reference, thus providing strong evidence that they all invoke the optimized GEMM.

4.2.2 **Experiment #2: SYRK.**

**Input.** Since all languages successfully map to GEMM, in this second experiment, we initialize the random matrices $A \in \mathbb{R}^{n \times k}$, $C \in \mathbb{R}^{n \times n}$, and input the expression $C := AA^T$, which is a special instance of GEMM in which matrix $B$ is substituted with $A^T$. Similarly to Experiment #1, we make use of the high-level abstractions offered by each language. Although the output matrix $C$ could be computed with a call to GEMM, performing $2n^2k$ FLOPs (“Floating Point Operations”), BLAS offers a specialized routine, SYRK (“SYmmetric Rank-K update”), which only performs $n^2k$ FLOPs. One expects SYRK to complete in approximately half the execution time of GEMM. As a reference implementation, we also performed a call to SYRK in C.

**Results.** In Table 3, by comparing the timings for SYRK to those of GEMM, one can tell which languages take advantage of the specialized routine, and which do not. Specifically, most languages have a computation time that is significantly less than that of a GEMM (almost half), strongly suggesting that they make the right decision. However, this is not the case for Eigen and R, whose computation time is equal to that of a GEMM.

4.2.3 **Experiment #3: SYR2K.**

**Input.** Since several languages are able to tell apart SYRK and GEMM, we now initialize the random matrices $A, B \in \mathbb{R}^{n \times k}$, $C \in \mathbb{R}^{n \times n}$, and test the slightly more complex expression $C := AB^T + BA^T$. This assignment could be computed by two successive calls to GEMM; it is however supported by the SYR2K kernel (“SYmmetric Rank-2K update”), which—similarly to SYRK—takes advantage of the fact that the matrix $C$ is symmetric (cost: $2n^2k$ FLOPs). Therefore, its execution time is expected to be approximately equal to that of a GEMM. As a reference implementation, we performed a call to SYR2K in C.

**Results.** In Table 3, by comparing the timings for SYR2K to those for GEMM and to the reference, one observes that in all cases, SYR2K requires double the time of a GEMM, thus indicating that no language selects the specialized BLAS kernel for SYR2K.

4.2.4 **Experiment #4: Update of C.** As specified in the BLAS interface [22], the kernels GEMM, SYRK and SYR2K offer the option of updating the matrix $C$. The full definition of GEMM is $C := \alpha AB + \beta C$, where $\alpha$, $\beta$ are scalars and $A$, $B$ and $C$ are matrices. Therefore, expressions such as $C := AB + C$ can be computed using one single call to GEMM, without the need for intermediate storage for $AB$. This functionality, which is also supported by SYRK and SYR2K, increases the overall performance and reduces the size of temporary storage; however, the computational cost for the addition of two matrices of size $\mathbb{R}^{n \times n}$ is $O(n^3)$, and for mid- and large-sized matrices this will be dwarfed by the $O(n^3)$ cost for the multiplication. On the contrary, the smaller the problem size, the more significant the contribution of the addition to the overall computation time. For completeness, we investigate whether or not the languages require a separate matrix addition when given such expressions as input.
Table 3. Experiments #1–3

| Name | Expression | C | Armadillo | Eigen | Julia | Matlab | NumPy | Octave | R |
|------|------------|---|----------|-------|-------|--------|-------|--------|---|
| GEMM | $C := AB$  | 1.43 | 1.43 | 1.46 | 1.44 | 1.44 | 1.48 | 1.48 | 1.47 |
| SYRK | $C := AA^T$| 0.73 | 0.74 | 1.57 | 0.76 | 0.76 | 0.78 | 0.78 | 1.51 |
| SYR2K| $C := AB^T + BA^T$ | 1.47 | 2.91 | 2.89 | 2.92 | 2.91 | 2.96 | 2.96 | 3.04 |

Timings are in seconds. By comparing the execution time of each language with that of hand-written C code, one can deduce whether or not a language makes use of the most appropriate BLAS kernel for the evaluation of each expression.

Table 4. Experiment #4: Update of C

| Expression | Armadillo | Eigen | Julia | Matlab | NumPy | Octave | R |
|------------|----------|-------|-------|--------|-------|--------|---|
| $C := AB$  | ✓✓✓✓✓✓✓✓✓ | ✓✓✓✓✓✓✓✓✓ | ✓✓✓✓✓✓✓✓✓ | ✓✓✓✓✓✓✓✓✓ | ✓✓✓✓✓✓✓✓✓ | ✓✓✓✓✓✓✓✓✓ | ✓✓✓✓✓✓✓✓✓ |
| $C := AB + C$ | – | – | – | – | – | – | – |
| $C + = AB$ | ✓ | – | – | n.a. | – | – | n.a. |
| $C := AA^T$ | ✓✓✓✓✓✓✓✓✓ | ✓✓✓✓✓✓✓✓✓ | ✓✓✓✓✓✓✓✓✓ | ✓✓✓✓✓✓✓✓✓ | ✓✓✓✓✓✓✓✓✓ | ✓✓✓✓✓✓✓✓✓ | ✓✓✓✓✓✓✓✓✓ |
| $C := AA^T + C$ | – | – | – | – | – | – | – |
| $C + = AA^T$ | – | – | n.a. | – | – | – | – |

With the exception of the $+=.$ operator overload in Armadillo for GEMM, no language maps to one single kernel call which includes the update to C.

**Input.** We used the expressions in Table 4 as input, where the matrices have the same sizes as in the three experiments above. To test if the languages require an extra addition, we also measured the time it takes for a similarly sized matrix addition in each language.

**Results.** Timings (see Table B.1) suggest that in all cases the expression is computed as two steps, a matrix multiplication followed by a matrix addition. The only exception is the “$+=.$” operator overload in Armadillo for GEMM.

### 4.3 Linear Systems

Although matrix inversion is an extremely common operator in linear algebra expressions, only selected applications actually require the explicit inversion of a matrix. In the vast majority of cases, the inversion can (and should) be avoided by solving a linear system, gaining both in speed and numerical stability [47, p. 260]. However, we observed that it is extremely common for inexperienced users to blindly translate the mathematical representation into code, resulting in the expressions such as $(AB + C)^{-1}Y$ being coded in Matlab as `inv(AB + C)*Y`, instead of the recommended $(AB+C)^{-1}Y$.

As shown in Table 5, most languages provide a special function (or operator) for solving linear systems of the form $Ax = B$ (or $xA = B$), where $A \in \mathbb{R}^{n \times n}$ is a matrix and $x$ and $B$ are either a vector of size $\mathbb{R}^{n\times1}$, or a matrix (multiple right-hand sides) of size $\mathbb{R}^{n \times m}$. These functions are usually quite sophisticated and try to determine certain properties of $A$, so that the most suitable (in terms of data structure, speed, and accuracy) factorization can be used. The extent to which languages can determine those properties will be further investigated in Section 4.5.

#### 4.3.1 Experiment #5: Explicit Inversion.

**Input.** We examine how languages handle the inverse operator; specifically, we aim to determine whether or not languages avoid (if possible) the explicit computation of a matrix inverse. The input
Table 5. Functions and Operands for Solving Linear Systems

| Name     | Solve linear system |
|----------|---------------------|
| Armadillo| solve(A, B)         |
| Eigen    | n.a.               |
| Julia    | A\B                |
| Matlab/Octave | A\B         |
| NumPy    | np.linalg.solve(A, B) |
| R        | solve(A, B)        |

Table 6. Experiment #5: Explicit Inversion

| Operation  | Armadillo | Eigen | Julia | Matlab | NumPy | Octave | R |
|------------|-----------|-------|-------|--------|-------|--------|---|
| inv(A)*b   | 0.68      | 2.26  | 1.74  | 1.79   | 2.26  | 1.87   | 2.25 |
| A\b        | 0.68      | 0.69  | 0.68  | 0.74   | 0.71  | 0.77   | 0.73 |

Armadillo is the only language that replaces the explicit inversion of a matrix with the solution of a linear system.

We compared the execution time to that of the expression A\b or solve(A, b).

Results. The timings in Table 6 indicate that Armadillo is the only language that substitutes the inv function with a solve (or \" operator). It should be noted, however, that most languages provide warnings either during runtime or in their documentation that using explicit inversion should be avoided whenever possible, in favor of their solve functions. The automatic replacement of the inv function with a solve is a rather bold decision that alters the semantics of the input expression. For this reason, it is questionable whether or not this optimization is reasonable. In light of the extremely common misuse of inversion in application codes, we feel that the replacement is at least partly justified. Indeed, our recommendation is that languages automatically map calls to the inv operator to a linear system (whenever possible), and that the actual matrix inversion is offered by less convenient functions such as explicit_inverse.

4.4 Matrix Chains

Because of associativity, a chain of matrix products (a “matrix chain”), can be computed in many different ways, each identified by a specific parenthesization. Depending on the size of the matrices in the chain, different parenthesizations lead to vastly different execution times and temporary storage requirements. The problem of determining the best parenthesization, in terms of number of floating point operations, is commonly referred to as the Matrix Chain Problem (MCP) [10, 36, 49]. In practice, different parenthesizations may also lead to different results because floating point arithmetic is not associative. However, since no convention for evaluating a product of matrices exists, languages can evaluate a chain in any order.

4.4.1 Experiment #6: Optimal Parenthesization.

Input. This experiment consists of three different matrix chains. Each of these is given as input to the languages as a single statement, and without any parenthesization. As Figure 1 shows,

---

7Eigen does not provide a general case solve function. One must explicitly factorize and then solve a linear system with specific methods.

8The number of different parenthesizations for a chain of length $n$ is given by the Catalan number $C_{n-1} = \frac{(2n)!}{(n+1)!n!}$. 

ACM Transactions on Mathematical Software, Vol. 48, No. 3, Article 26. Publication date: September 2022.
Fig. 1. Visual representation of the input expressions for Experiment #6: Optimal parenthesization.

Table 7. Experiment #6: Optimal Parenthesization

| Evaluation Sequence | Armadillo | Eigen | Julia | Matlab | NumPy | Octave | R |
|---------------------|-----------|-------|-------|--------|-------|--------|---|
| LtR no parenthesis   | 0.59      | 0.59  | 0.58  | 0.59   | 0.60  | 0.59   | 0.62 |
| LtR parenthesis      | 0.59      | 0.59  | 0.58  | 0.59   | 0.60  | 0.59   | 0.62 |
| Right-to-Left       | ✓         | ✓     | ✓     | ✓      | ✓     | ✓      | ✓  |
| RtL no parenthesis   | 0.60      | 1.75  | 1.74  | 1.74   | 1.78  | 1.78   | 1.77 |
| RtL parenthesis      | 0.60      | 0.60  | 0.59  | 0.59   | 0.59  | 0.60   | 0.63 |
| Mixed no parenthesis | 2.03      | 2.07  | 2.05  | 2.05   | 2.08  | 2.10   | 2.07 |
| Mixed parenthesis    | 0.89      | 0.92  | 0.90  | 0.90   | 0.91  | 0.92   | 0.92 |
| Mixed               | –         | –     | –     | –      | –     | –      | –  |

Armadillo is the only language that incorporates a (partial) solution to the matrix chain problem.

Results. Table 7 indicates that most languages evaluate the chain from left to right, without considering the MCP. Armadillo is the only language that partially solves the problem, by checking whether to evaluate from left or right; however, it does not properly handle the mixed case. It should be noted that NumPy offers a function called `multi_dot`, which solves the MCP using dynamic programming, although the user has to explicitly invoke it. Furthermore, several third-party developed packages that consider the MCP exist in Eigen, Julia, Matlab, and R.

4.5 Properties

BLAS & LAPACK offer specialized kernels for specific types of operands (e.g., SPD, Symmetric, Triangular, Banded matrices...). We test the matrix multiplication and the solution of a linear system, and investigate if high-level languages make use of those kernels without the explicit help of the user. To this end, we purposely do not use annotations about properties either in the matrix construction or in the computation. One could—correctly—argue that in this experiment languages are not used to their best potential. The rationale for not specifying properties is threefold: First, we aim to capture the scenario in which non-proficient users are not aware of properties, or do not know how to exploit them. Second, matrices can have many different origins, e.g., the

Since v1.7.0, Julia accurately solves Experiment #6: “Optimal Parenthesization”; that was not the case for v1.5.2.
Table 8. Experiments #7–8

| Operation       | Property | Armadillo | Eigen | Julia | Matlab | NumPy | Octave | R |
|-----------------|----------|-----------|-------|-------|--------|-------|--------|---|
| Multiplication  | Triangular| –         | –     | –     | –      | –     | –      | – |
|                 | Diagonal | –         | –     | –     | –      | –     | –      | – |
| Linear System   | Symmetric| –         | n.a.  | –     | –      | –     | –      | – |
| SPD             | ✓        | n.a.      | –     | ✓     | –      | ✓     | –      | – |
| Triangular      | ✓        | n.a.      | ✓     | ✓     | –      | ✓     | –      | – |
| Diagonal        | †        | n.a.      | ✓     | ✓     | †      | –     | †      | – |

The † indicates that the solver for triangular linear systems is used instead of a more efficient algorithm for diagonal systems. The ‡ indicates that a band matrix solver is used.

explicit construction with a specialized function, or the evaluation of an expression, and it is not guaranteed that the resulting matrices are always correctly annotated. Finally, there are cases where properties are only known at runtime.

4.5.1 Experiment #7: Multiplication.

Input. For the multiplication of matrices with properties, we examine two cases: Triangular and Diagonal. In the Triangular case, we input the expression $B := AB$, where $A$ and $B$ are a Lower Triangular and a Full matrix, respectively. We compare with a C program that explicitly invokes the BLAS kernel TRMM, which performs half of the FLOPs of a GEMM ($n^3$ vs. $2n^3$). For the Diagonal case, we input the expression $C := AB$, where $A$ and $B$ are a Diagonal and a Full matrix, respectively. Since BLAS offers no kernel for this operation, as a C language reference, one could use a loop to scale each row of $B$ individually, via the kernel SCAL.

Results. None of the languages examined use specialized kernels or methods to perform a multiplication between a Triangular/Diagonal and a Full matrix. It should be noted that Julia enables the user to easily annotate matrices with types that encode certain matrix properties such as Lower/Upper Triangular, Symmetric, Diagonal, and more. Specifically, Julia uses multiple dispatch [13], a technique with which it can separately define the multiplication operation for the pairs Triangular-Full and Diagonal-Full, and achieve high performance for these operations by mapping to the most appropriate kernels. Similarly, if a matrix is created with the diag function, Octave stores the information that the matrix is Diagonal, and then uses the annotation to select an efficient multiplication strategy. However, in both cases, the effectiveness of those mechanisms depends heavily on the method used to create or initialize the matrices, as well as the ability of those languages to propagate those properties across intermediate computations.

4.5.2 Experiment #8: Properties in Linear Systems.

Input. We created general matrices to satisfy the properties shown in the “Property” column of Table 8, and used the expressions in Table 5. We measured the execution time and compared it to the C implementation. The results are displayed in Table 8, while the execution time results are in Table B.2.

Results. Armadillo performs the optimal Cholesky factorization for the SPD case and forward substitution for the Triangular case; for the Diagonal case, a band matrix solver is used. Eigen does not participate in this experiment, as it requires the user to explicitly specify the type of factorization to perform on the input operand before solving the linear system. Julia recognizes the Triangular and Diagonal properties and performs forward substitution and vector scaling respectively for those cases. However, it does not make use of the Cholesky factorization for SPD.
matrices nor of the Bunch–Kaufman decomposition for symmetric matrices. Matlab is known\(^\text{10}\) to have an elaborate decision tree when solving a linear system to select the most suitable factorization based on the properties of the operands. Those properties are detected during runtime either by examining the contents of the matrices or by trial-and-error. Indeed, both Matlab and Octave take advantage of the SPD and the Triangular case; the Diagonal case is treated like Triangular. Finally, timings in Table B.2 suggest that Python and R use the general purpose LU factorization for all cases.

4.6 Common Subexpression Elimination

A common feature of modern compilers, at least when it comes to scalar computations, is **Common Subexpression Elimination (CSE)**. Compilers perform data flow analysis to detect subexpressions that evaluate to the same value and assess whether or not it is beneficial to compute them only once and substitute them with a temporary value in all subsequent instances. In Section 2, we proved that the optimal selection of common subexpressions is an NP-complete problem.

In light of the increased computational cost of matrix operations compared to scalar operations, it is mostly beneficial to detect and eliminate common subexpressions within linear algebra expressions. Consider for example the expression which occurs in the Stochastic Newton equations\(^\text{19}\): \(B_1 := \alpha(I_n - A^T W_1(A_1 I_l + W_1 A A^T W_1)^{-1} W_1^T A),\) where \(A \in \mathbb{R}^{m \times n}\) and \(W_1 \in \mathbb{R}^{m \times l}\) are general matrices. The term \(A^T W_1\) appears a total of four times in its original and transposed form, and can be factored out and computed only once, saving \(6 n m l\) FLOPs. However, when dealing with matrix expressions, the elimination of common subexpressions might be counter-productive, as the following example illustrates: Consider the expression \(A^T B A^{-1} y\), where \(A, B \in \mathbb{R}^{n \times n}\) and \(y \in \mathbb{R}^{n \times 1}\), which appears in signal processing\(^\text{21}\). First, the expression has to be changed to the form \((B A^{-1})^T B A^{-1}\) for the common subexpression to appear. Then, one might be tempted to factor out \(K := B A^{-1}\), solve it and then proceed to compute \(K^T K y\). The cost of this strategy is \(8 n^3 / 3 + 7 n^2\). By contrast, the optimal solution is to evaluate the initial expression from right to left, for a cost of \(4 n^3 / 3 + 7 n^2\). Furthermore, in addition to the computational cost, the decision to eliminate a common subexpression has to take into account the memory overhead of temporary matrices, which might represent a hard constraint, especially for architectures with limited memory.

4.6.1 Experiment #9: Common Subexpressions.

**Input.** To identify if any of the languages performs CSE, we create the random matrices \(A, B \in \mathbb{R}^{n \times n}\) and use the two expressions in Table 9(a) as input. In the “naive” column, the product \(A B\) appears twice; in the “recommended” column, the product is factored out with the help of a temporary variable.

**Results.** By comparing the execution time for the two experiments in Table B.3 in Appendix B, we conclude that no language eliminates the redundant operation. Since this experiment is particularly simple in terms of analysis and substitution, there is no reason to explore more advanced and frequently occurring scenarios, such as the stochastic Newton method mentioned above, or the Kalman filter and signal processing shown in Table 1.

4.7 Loop-Invariant Code Motion

Another common scalar optimization is Loop-Invariant Code Motion. For this, the compiler first looks for expressions that occur within a loop but yield the same result regardless of how many times the loop is executed\(^\text{7, p. 592}\), and then moves them out of the loop body. Again, when

\(^{10}\)https://uk.mathworks.com/help/matlab/ref/mldivide.html.
Table 9. Input Expressions for Experiments #9: Common Subexpression Elimination, and #10: Loop-Invariant Code Motion

| (a) Common Subexpression Elimination. | (b) Loop-Invariant Code Motion. |
|-------------------------------------|---------------------------------|
| Naive                               | Recommended                     |
| $X := ABAB$                         | $M := AB$                       |
| $X := MM$                           | $M := AB$                       |
| for $i$ in $1:n$ $M := AB$          | for $i$ in $1:n$ $X[i] := M[i, i]$ |
|                                     | $X[i] := M[i, i]$                |

dealing with matrix computations this optimization is particularly important due to their high computational cost. However, in the case of matrices, memory limitations might occur more frequently, compared to scalars, if intermediate storage matrices are large. The two code snippets shown in Table 9(b) extract the diagonal of the matrix product $AB$. In the “naive” column, $AB$ is recomputed in every iteration of the loop, for a total cost of $O(n^4)$ floating point operations. In the “recommended” column, the product is computed only once, outside the loop body, for a total cost of $O(n^3)$ operations.

4.7.1 Experiment #10: Loop-Invariant Code Motion.

Input. To identify whether or not any of the languages performs loop-invariant code motion, we measured the execution time of the snippets in Table 9(b).

Results. No language eliminates the redundant operations.

4.8 Partitioned/Blocked Operands

In many applications such as finite element methods [16] and signal processing [63], matrices exhibit blocked structures (e.g., block diagonal, block tridiagonal, block Toeplitz). In these cases, a blocked matrix representation is often extremely convenient to write concise equations; see Equation (1) for an example. However, an evaluation of such expressions that does not explicitly consider the blocked structure is likely to lead to suboptimal performance. The ability to handle each block individually can improve performance by reducing the overall amount of computation and/or by using specialized functions on blocks with certain properties.

\[
\begin{pmatrix} A_1 & 0 \\ 0 & A_2 \end{pmatrix}^{-1} B = \begin{pmatrix} A_1 & 0 \\ 0 & A_2 \end{pmatrix}^{-1} \begin{bmatrix} B_T \\ B_B \end{bmatrix} = \begin{bmatrix} A_1^{-1} B_T \\ A_2^{-1} B_B \end{bmatrix}
\]  

(1)

4.8.1 Experiment #11: Blocked Matrices.

Input. Equation 1 shows the experiment, where a block diagonal matrix consisting of matrices $A_1, A_2 \in \mathbb{R}^{n \times n}$ is used to solve a linear system. Since all the languages considered offer mechanisms to construct matrices out of blocks—the “[ ]” brackets in Matlab/Octave and Julia; explicit functions in Armadillo, Eigen, NumPy, and R—we examine if the structure of a blocked matrix (built with these mechanisms) is considered in subsequent operations. To give languages the best chance, we create the blocked matrix and immediately (without temporary storage) use it to solve a linear system. For example, the input expression for Matlab is “C = [A1 zeros(n, n); zeros(n, n) A2]\B;”

Results. By comparing the time to solve a linear system with a blocked matrix with the time it takes to solve two small linear systems, we conclude that no language makes use of the block diagonal structure of the matrix.
4.9 Partial Operand Access

It is often the case that only parts of the output operands are needed. Potentially, this means that not all operations need to be performed, but only those that contribute to the result. For instance, in audio segmentation [31], the self-similarity matrix of a signal is convoluted with a kernel, but only the elements of the diagonal are needed for further computations.

4.9.1 Experiment #12: Partial Operand Access.

Input. We perform six experiments on each language to determine the extent to which this optimization is applied. Specifically, we choose two operations, matrix addition and matrix multiplication, and request one single element, one column, or the diagonal of the result. The exact expressions used (for Octave) are shown in Table 10.

Results. As Table 10 indicates, Eigen is the only language that fully supports this optimization for the case of matrix addition. In the case of matrix multiplication, Armadillo and Eigen are the only languages to support the extraction of the diagonal of a product of two matrices, without performing a matrix-matrix multiplication. For the other two operations, no language is able to simplify computations and avoid performing a GEMM before extracting the user-requested part of the result. While Armadillo performs the optimization for the experiments \( \text{diag}(A+B) \) and \( \text{diag}(A*B) \), it does not support the syntax necessary for the other four experiments. Specifically, the expressions \((A+B)(c, c)\) and \((A+B).\text{col}(c)\), for an arbitrary constant \(c\), do not compile. Similarly, Matlab does not support the indexing of the result of operations in parenthesis.

4.10 Operation Merging

As demonstrated in Section 4.2.4 “Experiment # 4: Update of C”, BLAS offers in some cases the ability to fuse operations, i.e., perform two or more linear algebra operations using a single kernel invocation. In the spirit of operation merging, we investigate the case where the same matrix is multiplied with a series of vectors, and the result is stored in a different set of vectors; both the input and output vectors are stored contiguously in memory. The multiple matrix-vector operations can be fused together into a single matrix-matrix operation, yielding significantly higher performance. Similar situations could occur, for example, when implementing tensor contractions using a series of BLAS kernels [65].

4.10.1 Experiment #13: Merging Matrix-vector Products.

Input. To further investigate and highlight the importance of such optimizations, for each language we compare the execution time of the naive and recommended code snippets in Table 11, for

---

Table 10. Experiment #12: Partial Operand Access

| Experiment       | Armadillo | Eigen | Julia | Matlab | NumPy | Octave | R |
|------------------|-----------|-------|-------|--------|-------|--------|---|
| \((A+B)(c, c)\)  | n.a.      | ✓     | –     | n.a.   | –     | –      | – |
| \((A+B)(:, c)\)  | n.a.      | ✓     | –     | n.a.   | –     | –      | – |
| \(\text{diag}(A+B)\) | ✓         | ✓     | –     | –      | –     | –      | – |
| \((A*B)(c, c)\)  | n.a.      | –     | –     | n.a.   | –     | –      | – |
| \((A*B)(:, c)\)  | n.a.      | –     | –     | n.a.   | –     | –      | – |
| \(\text{diag}(A*B)\) | ✓\(^1\)   | ✓     | –     | –      | –     | –      | – |

Armadillo and Eigen are the only languages that avoid unnecessary computations when a user requests the diagonal of a matrix product.

---

\(^1\)Feature added in latest version (since 9.850-RC1). Not present in version 9.800.x.
A, B, C ∈ \( \mathbb{R}^{n \times n} \). The naive implementation already places the input and output vectors together, in contiguous positions in memory, by creating matrices B and C and using their columns as vectors; this gives languages the best chance of identifying that the operation can be performed using a single matrix-matrix operation.

Results. By comparing the execution time of the two implementations we conclude that no language performs this optimization.

5 FURTHER EXPERIMENTS

The experiments presented in Section 4 are meant to expose an initial set of optimizations that we deemed essential for programming languages to generate solutions of LAMPs that are competitive with those created by human experts. In particular, we only considered LAMPs with dense operands, and single core execution. However, given the prevalence of parallelism in modern computing, it would be remiss of us not to address the challenges associated with extending the study in that direction. Furthermore, as sparse operands arise in a plethora of applications, in this section we also highlight the challenges of performing a similar study for sparse linear algebra.

5.1 Multi-threaded Parallelism

We determined that all the seven languages support multi-threading: For all of them it is sufficient to link to a multi-threaded version of MKL BLAS and LAPACK; several (e.g., Armadillo, Eigen, R) also recommend compiling them with OpenMP support. Then, at runtime, all languages are aware of the 24 cores (2 socket machine) that are available to them without any further action necessary by the user. A natural extension of the LAMP to multi-threading experiments starts with the question “Do the languages automatically use multi-threading in solving instances of LAMPs?”

As shown in Table 12, an initial experiment on GEMM (random matrices of size \( A, B, C \in \mathbb{R}^{n \times n}, \ n = 6000 \)) clearly indicates that—for this operation—all languages make use of multiple cores.

While encouraging, this observation is not enough to claim that the languages solve LAMPs efficiently in multi-threaded environments; on the contrary, we argue that such a claim requires an in-depth, separate analysis which goes well beyond the set of experiments previously discussed.

- In a multi-threaded environment, the efficient solution of a LAMP requires choosing among different algorithms and implementations based on their scalability. For example, given the matrices \( A \in \mathbb{R}^{6000 \times 6000} \) and \( B \in \mathbb{R}^{6000 \times 200} \), with \( A \) symmetric, the solution of the linear system \( Y := A \ \setminus \ B \) with a parallel implementation of the \( LDL^T \) factorization (method of choice for symmetric systems) is slower (0.75 seconds) than by using a parallel \( LU \) factorization.
Table 13. Time (in sec.) to Solve the Linear System $AX = B$, where $A$ is a Sparse Matrix of Size $n \times n$ for $n = 25000$ and has a Density of $d = 0.0002$ and $B$ is a Random Dense Matrix of Size $n \times 10$

| Operation     | Property | Arma | Eigen | Julia | Matlab | Octave | SciPy | R  |
|---------------|----------|------|-------|-------|--------|--------|-------|----|
| Linear System | General  | n.a. | n.a.  | 34.0  | 33.3   | 36.2   | 1621  | 52.7|
| Symmetric     | n.a.     | n.a. | 103.4 | 18.2  | 42.8   | 1148   | 38    |
| SPD           | n.a.     | n.a. | 6     | 5.9   | 5.8    | 1110   | 6.3   |
| Triangular    | n.a.     | n.a. | 0.007 | 0.005 | 0.005  | 807    | 0.005 |

(0.29 seconds). This simple example shows the need for scalability analyses and performance models to make choices among different algorithmic options.

- An extra level of complexity comes from the possibility of distributing the available computational resources over independent parts of the target computation. For example, the least squares problem

$$B := (X^T X)^{-1} X^T Y$$

includes two operations that can be performed simultaneously: a SYRK for $X^T X$ and a GEMM for $X^T Y$. Whether or not it is beneficial to split resources over these two operations, and in which proportion, is a question that depends on problem sizes and efficiency of the kernels. In a nutshell, the problem goes beyond the “simple” mapping of operations to kernels, as it takes the form of resource allocation.

5.2 Sparse Linear Algebra

From a first investigation, we determined that most of the languages considered support sparse matrix representation(s) natively (i.e., without installing extra packages or libraries). It should be noted, however, that even though all languages support sparse data structures (or sparse matrices), the assessment of how well they automatically deal with sparsity is not as straightforward as their dense counterpart.

A natural extension of our investigation would be to determine if the solution to LAMP instances is affected by the presence of sparse operands, for instance, by posing and answering questions such as “If $A$ is sparse, is $X := A \backslash B$ mapped to sparse solver?”. In this spirit, we repeated “Experiment #8: Properties in Linear Systems” with sparse $A$. In all cases, matrix $A$ is of size $n \times n$ for $n = 25000$ and has a density of $d = 0.0002$, and $B$ is a random dense matrix of size $n \times 10$.

The results presented in Table 13 indicate a clear difference in how these languages handle sparse operands with different properties. Armadillo requires the installation of external packages (SuperLU) to support sparse solvers; Eigen (similarly to dense) requires the explicit statement by the user for what kind of method to use to solve a linear system. Julia, Matlab, Octave, and R use an optimized sparse solver for the general case, considering that the execution time of LU in C for a same size dense matrix is $\approx 180$ seconds (vs. 34sec, 33sec, 36sec, and 52sec). Similarly, they exhibit a significant drop in execution time from the general case to the SPD, and even more so to the triangular case, indicating that they identify and make use of the corresponding properties to reduce execution time. SciPy mentions in its documentation\(^\text{12}\) that if $B$ is dense, its method is expected to be highly inefficient. It is difficult to draw conclusions for the symmetric case. Matlab and R seem to be the only languages that take advantage of symmetry in a way that improves performance (relative to their individual performance in the general case). For Julia and Octave, it

\(^{12}\)https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.spsolve.html.
cannot be determined whether the increased time required compared to the general case is due to 
the time needed to determine the appropriate factorization to use, or to the use of an inefficient 
algorithm for the particular sparsity pattern.

This rather simple experiment already highlights a few key challenges associated to sparse 
computations:

- There exist many sparse storage formats [54] that are tailored for specific applications and/or 
operations. Each of the languages supports a subset of those formats. A comprehensive study 
of LAMPs in the sparse domain would have to first determine if and to what extent the 
languages convert operands to different storage formats depending on the computations to 
be performed.
- Due to the enormous differences in density and sparsity patterns observed in matrices from 
different application domains, it is unclear what to expect from languages when they lack 
annotations from the users. Not only is the performance (or even the success) of a solver 
hardly predictable, but different users might also favor different computation aspects (e.g., 
accuracy vs. execution time vs. space). The situation further worsens when considering also 
re-ordering of variables, iterative solvers, and of course preconditioning.

6 GUIDELINES FOR OPTIMIZING LINEAR ALGEBRA CODE

In Section 4, we presented results for 12 different optimizations; fully aware that not all optimiza-
tions have the same potential/impact on performance, we offer here a set of recommendations on 
how to prioritize optimizations in the context of a LAMP solver. These recommendations shall be 
useful to both language developers—who aim to provide a LAMP solver—and to end users—who 
solve LAMPs either using a specialized language or manually (e.g., using direct calls to BLAS/LA-
PACK).

We identified three categories of optimizations; respectively, they aim at improving asymptotic 
complexity, reducing the complexity by a constant factor, and increasing arithmetic intensity. De-
dpending on the characteristics of the problem, the importance of the different categories varies. 
For example, for problems that are compute bound—i.e., their execution time is mostly dictated 
by the computational speed of the processor—it is clearly beneficial to apply optimizations that 
reduce the total amount of floating point operations; in this case, the categories should be consid-
ered in the same order as they are listed. By contrast, problems that are bandwidth bound—i.e., 
their execution time is mostly determined by the memory bandwidth—would benefit more from 
optimizations that reduce the total accesses to global memory [80]; the categories should then 
be considered starting from the last: “3. Increasing arithmetic intensity” (followed by 1. and 2.). 
We premise this discussion by pointing out that, given the complex ways in which different optimi-
izations interact with one another, it is infeasible to provide a definitive list of steps for how to 
optimize arbitrary linear algebra problems. Nonetheless, in many cases significant improvements 
are already achieved by exploring a small number optimizations, particularly those that improve 
asymptotic complexity.

The three optimization categories follow.

(1) Improving asymptotic complexity
   (a) Matrix chain problem (Section 4.4)
   (b) Exploit properties such as diagonal, banded (Section 4.5), or sparsity.
   (c) Loop-invariant code motion (Section 4.7)
   (d) Exploit block structure (Section 4.8)
   (e) Partial computation (Section 4.9)
(2) Reducing complexity by a constant factor
   (a) Exploit properties such as SPD, symmetric, or triangular (Section 4.5).
   (b) Common subexpression elimination (Section 4.6).
   (c) Distributivity and other algebraic identities \[66\].

(3) Increasing arithmetic intensity
   (a) Operation fusion (Section 4.10).

Notes on Specific Optimizations

(2.b) It should be noted that if an expression of the form $X^{-1}$ appears multiple times, one should not precompute this expression (see Section 4.3.1). Instead, one should compute a factorization of $X$ once, and use (and re-use) the resulting factors to solve the linear systems. In addition, note that when dealing with matrix expressions, the precomputation of a subexpression can sometimes lead to suboptimal solutions (in terms of execution time). For example, consider the expression $b := ABABy$, for $A, B \in \mathbb{R}^{n \times n}$ and $y \in \mathbb{R}^{n \times 1}$. In this example, by eliminating the common subexpression $AB$ (i.e., computing it only once), one would perform $O(n^3)$ FLOPs, leading to a suboptimal solution, compared to performing four consecutive matrix-vector multiplications (each of $O(n^2)$ FLOPs) to compute the target expression.

(2.c) As an example, the direct translation to code of Example 10 in Appendix A results in $O(n^3)$ FLOPs, due to the matrix product $H^\dagger H$. However, by applying the law of distributivity the expression is transformed to $y_k := H^\dagger (y - Hx_k) + x_k$, which only contains matrix-vector products, each with cost of $O(n^2)$ FLOPs \[12\].

(3.a) While optimizations that reduce the computational cost of an operation are—in general—an effective way of improving performance, the effects of these optimizations are not as noticeable for bandwidth bound problems. The reason is that the computation time for these problems is dominated by the time spent on fetching data from memory. In this case, increasing the arithmetic intensity—i.e., the ratio of arithmetic operations per memory access—is the most effective way to reduce the execution time.

General Remarks

- When applying the optimizations discussed in this section (particularly 2.b and 1.a), it is beneficial to look at the target linear algebra problem as one large expression, instead of a sequence of smaller operations. For instance, Example 12 in Appendix A is sometimes represented as two expressions, namely: $\Lambda := S(S^T A^T W A S)^{-1} S^T$ and $X_{k+1} := X_k + (I_n - X_k A^T) A \Lambda A^T W$. This representation, which clearly improves its readability and ease of handling, might however make certain optimization opportunities more difficult to identify. For instance, the full expression $X_{k+1} := X_k + (I_n - X_k A^T) S(S^T A^T W A S)^{-1} S^T A^T W$ contains the subexpression $AS$ three times; in contrast, the same subexpression exists only twice in $\Lambda$. Similarly, the full expression reveals a larger matrix chain to be optimized.

- Instead of using kernels or operators that directly solve linear systems (e.g., LAPACK’s GESV kernel or MATLAB’s “\" operator), it is often beneficial to apply matrix factorizations. The reason is that matrix factorizations may enable other optimizations that are not possible when using black-box solvers. For example, consider Example 2 of Appendix A. By applying the Cholesky factorization to matrix $M$, one gets $b := (X^T L^{-1} L^{-T} X)^{-1} X^T L^{-1} L^{-T} y$. In this resulting expression, it is now possible to identify $X^T L^{-1}$ as a common subexpression that appears three times, and to compute it only once \[12\].

- In general, the fastest way of evaluating an expression may not be the most numerically stable. For example, consider Example 1 of Appendix A. Assuming $X$ to be full rank, the fastest
solution would be to first compute \( T := X^T X \), followed by the Cholesky factorization of \( T \). However, by doing so, the condition number of \( X \) would be squared; in certain applications, this would lead to \( T \) being close to singular. If that is the case (i.e., \( X \) has a high condition number), a more stable—albeit slower—solution consist in computing the QR factorization of \( X \) [32].

7 CONCLUSIONS

We consider LAMP, the problem of mapping target linear algebra expressions onto a set of available instructions while minimizing a cost function. We provide a definition to the LAMP that unifies diverse and seemingly distant research directions in numerical linear algebra and high-performance computing; from this, we prove that in general, a LAMP is at least NP-complete. We then focus on matrix expressions that arise in practical applications, select popular programming languages that offer a high-level interface to linear algebra, and set out to investigate how efficiently they solve LAMPS. To this end, we create a benchmark consisting of simple tests, and exposing individual optimizations that are necessary to achieve good performance; these include both standard compiler optimizations such as common subexpression elimination and loop-invariant code motion, as well as linear algebra specific optimizations such as the matrix chain problem, and matrix properties. We discuss the details of each optimization and demonstrate its effect on performance. This investigation aims not only to showcase the capabilities and limitations of high-level languages for matrix computations, but also to serve as a guide for the future development of such languages. Finally, we offer a set of recommendations, targeted towards both language developers and end users, on how to prioritize optimizations when solving a LAMP.

Future Work. As explained in detail in Section 5, follow-up studies are possible (albeit challenging) to extend this analysis to multi-threaded environments and/or to sparse computations. On the contrary, any extension to distributed computing or accelerators is conditional to further language development or to the use of external libraries, since support from the existing list of languages is mostly lacking.

One further investigation, motivated by applications that require mixed precision computations, such as machine learning, would repeat the experiments using different data types (single precision, bfloat, etc.), while monitoring whether or not performance scales accordingly. Finally, in all our experiments, we concerned ourselves only with performance; in practice, numerical stability and the proper handling of ill-conditioned matrices are critically important aspects of matrix computations. Further experiments should be designed to assess how high-level languages deal with such issues.

APPENDICES

A EXAMPLE PROBLEMS

(1) Standard Least Squares

\[
b := (X^T X)^{-1} X^T y
\]

\( X \in \mathbb{R}^{n \times m}; y \in \mathbb{R}^{n \times 1}; n > m \)

(2) Generalized Least Squares

\[
b := (X^T M^{-1} X)^{-1} X^T M^{-1} y
\]

\( M \in \mathbb{R}^{n \times n}, \text{SPD}; X \in \mathbb{R}^{n \times m}; y \in \mathbb{R}^{n \times 1}; n > m \)
(3) Optimization [73]

\[ x := W(A^T(AWAT)^{-1}b - c) \]

\[ A \in \mathbb{R}^{m \times n}; \quad W \in \mathbb{R}^{n \times n}, \text{ DI, SPD}; \quad b \in \mathbb{R}^{m \times 1}; \quad c \in \mathbb{R}^{n \times 1}; \quad n > m \]

(4) Optimization [73]

\[ x_f := WAT(AWAT)^{-1}(b - Ax) \]
\[ x_o := W(A^T(AWAT)^{-1}Ax - c) \]

\[ A \in \mathbb{R}^{m \times n}; \quad W \in \mathbb{R}^{n \times n}, \text{ DI, SPD}; \quad b \in \mathbb{R}^{m \times 1}; \quad c \in \mathbb{R}^{n \times 1}; \quad n > m \]

(5) Signal Processing [21]

\[ x := (A^{-T}BTBA^{-1} + R^T L R)^{-1}A^{-T}BTBA^{-1}y \]

\[ A, B \in \mathbb{R}^{n \times n}, \text{ Band matrices (Toeplitz)}; \quad R \in \mathbb{R}^{n-1 \times n}, \text{ Upper Bidiagonal} ; \quad L \in \mathbb{R}^{n \times n-1}, \text{ DI}; \quad y \in \mathbb{R}^{n \times 1} \]

(6) Triangular Matrix Inversion [14]

\[ X_{10} := L_{10} L_{00}^{-1} \]
\[ X_{20} := L_{20} + L_{22} L_{21} L_{11}^{-1} L_{10} \]
\[ X_{11} := L_{11}^{-1} \]
\[ X_{21} := -L_{22} L_{21} \]

\[ L_{00} \in \mathbb{R}^{n \times n}, \text{ LT}; \quad L_{11} \in \mathbb{R}^{m \times m}, \text{ LT}; \quad L_{22} \in \mathbb{R}^{k \times k}, \text{ LT}; \quad L_{10} \in \mathbb{R}^{m \times n}; \quad L_{20} \in \mathbb{R}^{k \times n}; \quad L_{21} \in \mathbb{R}^{k \times n} \]

(7) Ensemble Kalman Filter [59]

\[ X^a := X^b + (B^{-1} + H^T R^{-1} H)^{-1}(Y - HX^b) \]

\[ B \in \mathbb{R}^{N \times N}, \text{ SPSD}; \quad H \in \mathbb{R}^{m \times N}; \quad R \in \mathbb{R}^{m \times m}, \text{ SPSD}; \quad Y \in \mathbb{R}^{m \times N}; \quad X^b \in \mathbb{R}^{n \times N} \]

(8) Ensemble Kalman Filter [59]

\[ \delta X := (B^{-1} + H^T R^{-1} H)^{-1}H^T R^{-1}(Y - HX^b) \]

see 7

(9) Ensemble Kalman Filter [59]

\[ \delta X := XV^T (R + HX(HX)^T)^{-1}(Y - HX^b) \]

\[ X \in \mathbb{R}^{m \times N}; \text{ see 7} \]

(10) Image Restoration [74]

\[ x_k := (H^T H + \lambda \sigma^2 I_n)^{-1}(H^T y + \lambda \sigma^2 (v_{k-1} - u_{k-1})) \]

\[ H \in \mathbb{R}^{m \times n}; \quad y \in \mathbb{R}^{m \times 1}; \quad v_{k-1} \in \mathbb{R}^{n \times 1}; \quad u_{k-1} \in \mathbb{R}^{n \times 1}; \quad \lambda > 0; \quad \sigma > 0; \quad n > m \]
(11) Image Restoration \cite{74}

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

\[ H^\dagger \in \mathbb{R}^{nxm}; \text{ see } 10 \]

(12) Randomized Matrix Inversion \cite{41}

\[ X_{k+1} := X_k + WA^T S(S^T AWA^T S)^{-1}S^T (I_n - AX_k) \]
\[ W \in \mathbb{R}^{nxn}, \text{ SPD}; S \in \mathbb{R}^{nxq}; A \in \mathbb{R}^{nxn}, X_k \in \mathbb{R}^{nxn}, q \ll n \]

(13) Randomized Matrix Inversion \cite{41}

\[ \Lambda := S(S^T AWA^T S)^{-1}S^T \]
\[ X_{k+1} := X_k + (I_n - X_k A^T)\Lambda A^T W \]

see 12

(14) Randomized Matrix Inversion \cite{41}

\[ \Lambda := S(S^T AWA^T S)^{-1}S^T \]
\[ \Theta := \Lambda AW \]
\[ M_k := X_k A - I \]
\[ X_{k+1} := X_k - M_k \Theta - (M_k \Theta)^T + \Theta^T (AX_k A - A)\Theta \]

\[ A \in \mathbb{R}^{nxn}, \text{ SYM}; X_k \in \mathbb{R}^{nxn}, \text{ SYM}; \Lambda \in \mathbb{R}^{nxn}, \text{ SYM}; \Theta \in \mathbb{R}^{nxn}, M_k \in \mathbb{R}^{nxn}; \text{ see } 12 \]

(15) Randomized Matrix Inversion \cite{41}

\[ X_{k+1} := S(S^T AS)^{-1}S^T + (I_n - S(S^T AS)^{-1}S^T A)X_k(I_n - AS(S^T AS)^{-1}S^T) \]
\[ A \in \mathbb{R}^{nxn}, \text{ SPD}; W \in \mathbb{R}^{nxn}, \text{ SPD}; S \in \mathbb{R}^{nxq}; X_k \in \mathbb{R}^{nxn}, q \ll n \]

(16) Stochastic Newton \cite{19}

\[ B_k := \frac{k}{k-1} B_{k-1} (I_n - A^T W_k ((k-1)I_l + W_k^T A B_{k-1} A^T W_k)^{-1} W_k^T A B_{k-1}) \]
\[ W_k \in \mathbb{R}^{m \times l}; A \in \mathbb{R}^{m \times n}, B_k \in \mathbb{R}^{n \times n}, \text{ SPD}; l < n \ll m \]

(17) Stochastic Newton \cite{19}

\[ B_1 := \frac{1}{\lambda_1} (I_n - A^T W_1 (\lambda_1 I_l + W_1^T A A^T W_1)^{-1} W_1^T A) \]

see 16

(18) Tikhonov regularization \cite{37}

\[ x := (A^T A + \Gamma^T \Gamma)^{-1} A^T b \]
\[ A \in \mathbb{R}^{nxm}, \Gamma \in \mathbb{R}^{nxm}, b \in \mathbb{R}^{nx1} \]
(19) Tikhonov regularization [37]
\[ x := (A^T A + \alpha^2 I)^{-1} A^T b \]
\( \alpha > 0; \) see 18

(20) Generalized Tikhonov regularization
\[ x := (A^T P A + Q)^{-1} (A^T P b + Q x_0) \]
P \( \in \mathbb{R}^{m \times m}, \) SPDS; Q \( \in \mathbb{R}^{m \times m}, \) SPDS; x₀ \( \in \mathbb{R}^{m \times 1}, \) A \( \in \mathbb{R}^{n \times m}, \) Γ \( \in \mathbb{R}^{m \times m}; \) b \( \in \mathbb{R}^{n \times 1} \)

(21) Generalized Tikhonov regularization
\[ x := x_0 + (A^T P A + Q)^{-1} (A^T P (b - A x_0)) \]
see 20

(22) LMMSE estimator [52]
\[ x_{\text{out}} = C_X A^T (A C_X A^T + C_Z)^{-1} (y - A x) + x \]
A \( \in \mathbb{R}^{m \times n}, \) C_X \( \in \mathbb{R}^{n \times n}, \) SPDS; C_Z \( \in \mathbb{R}^{m \times m}, \) SPDS; x \( \in \mathbb{R}^{n \times 1}, \) y \( \in \mathbb{R}^{m \times 1} \)

(23) LMMSE estimator [52]
\[ x_{\text{out}} := (A^T C_Z^{-1} A + C_X^{-1})^{-1} A^T C_Z^{-1} (y - A x) + x \]
see 22

(24) LMMSE estimator [52]
\[ K_{t+1} := C_t A^T (A C_t A^T + C_Z)^{-1} \]
x_{t+1} := x_t + K_{t+1} (y - A x_t)
C_{t+1} := (I - K_{t+1} A) C_t
A \( \in \mathbb{R}^{m \times n}, \) K_{t+1} \( \in \mathbb{R}^{m \times m}, \) C_t \( \in \mathbb{R}^{n \times n}, \) SPDS; C_Z \( \in \mathbb{R}^{m \times m}, \) SPDS; x_{t} \( \in \mathbb{R}^{n \times 1}, \) y \( \in \mathbb{R}^{m \times 1} \)

(25) Kalman Filter [53]
\[ K_k := P_{k-1} H_k^T (H_k P_{k-1} H_k^T + R_k)^{-1} \]
P_k := (I - K_k H_k) P_{k-1}
x_k := x_{k-1} + K_k (z_k - H_k x_{k-1})
K_k \( \in \mathbb{R}^{n \times m}, \) P_k \( \in \mathbb{R}^{n \times n}, \) SPD; H_k \( \in \mathbb{R}^{m \times n}, \) SPD; R_k \( \in \mathbb{R}^{m \times m}, \) SPDS; x_{k} \( \in \mathbb{R}^{n \times 1}, \) z_k \( \in \mathbb{R}^{m \times 1} \)
## TIMINGS AND OPERAND SIZES

### Table B.1. Experiment #4: Update of C

| Expression | Armadillo | Eigen | Julia | Matlab | NumPy | Octave | R   |
|------------|-----------|-------|-------|--------|-------|--------|-----|
| $C_2 := C_1 + C_2$ | 0.01 | 0.01 | 0.03 | 0.03 | 0.03 | 0.03 | 0.03 |
| $C := AB$ | 1.43 | 1.43 | 1.46 | 1.44 | 1.44 | 1.48 | 1.48 | 1.47 |
| $C := AB + C$ | 1.43 | 1.46 | 1.47 | 1.45 | 1.49 | 1.51 | 1.5  |
| $C += AB$ | n.a. | 1.43 | 1.46 | 1.47 | n.a. | 1.49 | 1.51 | n.a. |
| $C := AA^T$ | 0.73 | 0.74 | 0.76 | 0.76 | 0.78 | 0.78 |
| $C := AA^T + C$ | 0.73 | 0.77 | 0.80 | 0.77 | 0.79 | 0.81 |
| $C += AA^T$ | n.a. | 0.77 | 0.81 | n.a. | 0.79 | 0.81 |

### Table B.2. Experiments #7–8 Properties in Multiplication and Linear Systems

| Operation | Property | Armadillo | Eigen | Julia | Matlab | NumPy | Octave | R   |
|-----------|----------|-----------|-------|-------|--------|-------|--------|-----|
| Multiplication | General | 1.46 | 1.43 | 1.46 | 1.44 | 1.44 | 1.45 | 1.48 | 1.47 |
| Triangular | 0.73 | 1.44 | 1.46 | 1.44 | 1.44 | 1.48 | 1.48 | 1.47 |
| Diagonal | 0.06 | 1.43 | 1.46 | 1.44 | 1.44 | 1.48 | 1.48 | 1.47 |
| Linear System | General | 0.65 | 0.68 | 0.68 | 0.74 | 0.71 | 0.77 | 0.73 |
| Symmetric | 0.52 | 0.68 | 0.68 | 0.76 | 0.71 | 0.77 | 0.73 |
| SPD | 0.36 | 0.42 | 0.65 | 0.46 | 0.68 | 0.47 | 0.70 |
| Triangular | 0.05 | 0.09 | 0.06 | 0.06 | 0.71 | 0.07 | 0.70 |
| Diagonal | 0.002 | 0.04 | 0.02 | 0.06 | 0.68 | 0.08 | 0.69 |

### Table B.3. Experiment #9: Common Subexpression Elimination

| Experiment | Armadillo | Eigen | Julia | Matlab | NumPy | Octave | R   |
|------------|-----------|-------|-------|--------|-------|--------|-----|
| naive | 2.87 | 2.90 | 2.89 | 2.91 | 2.90 | 2.97 | 2.99 | 2.96 |
| recommended | 1.44 | 1.44 | 1.47 | 1.47 | 1.45 | 1.50 | 1.50 | 1.50 |

### Table B.4. Experiment #10: Loop Invariant Code Motion

| Experiment | Armadillo | Eigen | Julia | Matlab | NumPy | Octave | R   |
|------------|-----------|-------|-------|--------|-------|--------|-----|
| naive | 0.457 | 0.489 | 0.461 | 0.458 | 0.473 | 0.545 |
| recommended | 0.002 | 0.002 | 0.002 | 0.002 | 0.002 | 0.005 | 0.002 |

### Table B.5. Experiment #11: Partitioned Operands

| Experiment | Armadillo | Eigen | Julia | Matlab | NumPy | Octave | R   |
|------------|-----------|-------|-------|--------|-------|--------|-----|
| compact | 2.06 | 2.13 | 2.08 | 2.18 | 2.23 | 2.23 | 2.17 |
| blocked (manually) | 0.97 | 0.98 | 0.97 | 1.01 | 1.06 | 1.06 | 1.02 |
Table B.6. Experiment #12: Partial Operand Access

| Experiment          | Armadillo | Eigen  | Julia | Matlab | NumPy  | Octave | R      |
|---------------------|-----------|--------|-------|--------|--------|--------|--------|
| \((A+B)(c,c)\)      | n.a.      | 0.000000 | 0.029364 | n.a.  | 0.030491 | 0.030074 | 0.017350 |
| \(A(c,c)+B(c,c)\)   | 0.000000  | 0.000000 | 0.000022 | 0.000005 | 0.000043 | 0.000012 |
| \((A+B)(:,c)\)      | n.a.      | 0.000007 | 0.029320 | n.a.  | 0.029229 | 0.030080 | 0.017368 |
| \(A(:,c)+B(:,c)\)   | 0.000007  | 0.000022 | 0.000092 | 0.000043 |
| diag(A+B)           | 0.0001    | 0.0001  | 0.0294  | 0.0291  | 0.0305  | 0.0298  | 0.0174  |
| diag(A)+diag(B)     | 0.0001    | 0.0001  | 0.0001  | 0.0001  | 0.0006  | 0.0001  | 0.0002  |

| Experiment          | Armadillo | Eigen  | Julia | Matlab | NumPy  | Octave | R      |
|---------------------|-----------|--------|-------|--------|--------|--------|--------|
| \(A*B\)            | 1.43      | 1.46   | 1.44  | 1.44   | 1.47   | 1.47   | 1.47   |
| \((A*B)(c,c)\)     | n.a.      | 1.45   | 1.44  | n.a.   | 1.48   | 1.48   | 1.46   |
| \((A*B)(:,c)\)     | 0.033     | 0.026  | 1.44  | 1.44   | 1.48   | 1.48   | 1.46   |

Table B.7. Experiment #13: Merging Matrix-vector Products

| Experiment          | Armadillo | Eigen  | Julia | Matlab | NumPy  | Octave | R      |
|---------------------|-----------|--------|-------|--------|--------|--------|--------|
| naive               | 16.23     | 16.23  | 16.29 | 16.31  | 16.66  | 16.51  | 16.55  |
| recommended         | 1.43      | 1.46   | 1.44  | 1.44   | 1.47   | 1.47   | 1.47   |

B.1 Operand Sizes

For all experiments \(n = 3000\).

Experiment #1: GEMM: \(A, B, C \in \mathbb{R}^{n \times n}\)

Experiment #2: SYRK: \(A, C \in \mathbb{R}^{n \times n}\)

Experiment #3: SYR2K: \(A, B, C \in \mathbb{R}^{n \times n}\)

Experiment #4: Update of \(C\): \(A, B, C \in \mathbb{R}^{n \times n}\)

Experiment #5: Explicit Inversion: \(A \in \mathbb{R}^{n \times n}; B, C \in \mathbb{R}^{n \times 200}\)

Experiment #6: Optimal Parenthesization:

- LtR: \(M_1 \in \mathbb{R}^{(n/5) \times n}; M_2, M_3 \in \mathbb{R}^{n \times n}\)
- RtL: \(M_1, M_2 \in \mathbb{R}^{n \times n}; M_3 \in \mathbb{R}^{n \times (n/5)}\)
- Mixed: \(M_1, M_4 \in \mathbb{R}^{n \times n}; M_2 \in \mathbb{R}^{n \times (n/5)}; M_3 \in \mathbb{R}^{(n/5) \times n}\)

Experiment #7: Properties in Multiplication: \(A, B, C \in \mathbb{R}^{n \times n}\)

Experiment #8: Properties in Linear Systems: \(A \in \mathbb{R}^{n \times n}, B, C \in \mathbb{R}^{n \times 200}\)

Experiment #9: Common Subexpressions: \(A, B, X, M \in \mathbb{R}^{n \times n}\)

Experiment #10: Loop Invariant Code Motion: \(A, B, M \in \mathbb{R}^{n \times n}\)

Experiment #11: Blocked Matrices: \(A_1, A_2 \in \mathbb{R}^{(n/4) \times (n/4)}; B, C \in \mathbb{R}^{(n/2) \times (n/2)}\)

Experiment #12: Partial Operand Access: \(A, B \in \mathbb{R}^{n \times n}\)

Experiment #13: Merging Matrix-vector Products: \(A, B, C \in \mathbb{R}^{n \times n}\)

REFERENCES

[1] [n.d.]. Commons Math: The Apache Commons Mathematics Library. https://commons.apache.org/math/.
[2] [n.d.]. HASEM. https://sourceforge.net/p/hasem/wiki/HASEM/.
[3] 2017. Intel®Math Kernel Library documentation. https://software.intel.com/en-us/mkl-reference-manual-for-c.
[4] 2018. Maple, Maplesoft. https://www.maplesoft.com.
[5] 2018. Matlab, The MathWorks, Inc. https://www.mathworks.com/.
[6] Shrirang Abhyankar, Jed Brown, Emil M. Constantinescu, Debojyoti Ghosh, Barry F. Smith, and Hong Zhang. 2018. PETSc/TS: A Modern Scalable ODE/DAE Solver Library. https://doi.org/10.48550/ARXIV.1806.01437
The Linear Algebra Mapping Problem

26:27

[7] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., USA.

[8] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. 1990. LAPACK: A portable linear algebra library for high-performance computers. In Supercomputing ’90: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing. 2–11. https://doi.org/10.1109/SUPER.1990.129995

[9] Apple Inc. [n.d.]. Accelerate Framework. https://developer.apple.com/documentation/accelerate

[10] Henrik Barthels, Marcin Copik, and Paolo Bientinesi. 2018. The generalized matrix chain algorithm. In Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018). Association for Computing Machinery, New York, NY, USA, 138–148. https://doi.org/10.1145/3168804

[11] Henrik Barthels, Christos Psarras, and Paolo Bientinesi. 2020. Automatic generation of efficient linear algebra programs. In Proceedings of the Platform for Advanced Scientific Computing Conference (PASC’20). Association for Computing Machinery, New York, NY, USA, Article 1, 11 pages. https://doi.org/10.1145/3394277.3401836

[12] Henrik Barthels, Christos Psarras, and Paolo Bientinesi. 2021. Linnea: Automatic generation of efficient linear algebra programs. ACM Trans. Math. Softw. 47, 3, Article 22 (June 2021), 26 pages. https://doi.org/10.1145/3446632

[13] J. Bezanson, A. Edelman, S. Karpinski, and V. Shah. 2017. Julia: A fresh approach to numerical computing. SIAM Rev. 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671

[14] Paolo Bientinesi, Brian Gunter, and Robert A. van de Geijn. 2008. Families of algorithms related to the inversion of a symmetric positive definite matrix. ACM Trans. Math. Softw. 35, 1, Article 3 (July 2008), 22 pages. https://doi.org/10.1145/1377603.1377606

[15] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. 2005. Representing linear algebra algorithms in code: The FLAME application program interfaces. ACM Trans. Math. Softw. 31, 1 (Mar. 2005), 27–59. https://doi.org/10.1145/1055531.1055533

[16] Markus Blatt. 2010. A Parallel Algebraic Multigrid Method for Elliptic Problems with Highly Discontinuous Coefficients. Ph.D. Dissertation. Ruprecht-Karls-Universität Heidelberg. https://doi.org/10.11858/heidok.00010856

[17] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. 2007. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? Or revisiting register allocation: Why and how. In Languages and Compilers for Parallel Computing, George Almási, Càlin Caşcaval, and Peng Wu (Eds.). Springer Berlin, Berlin, 283–298. https://doi.org/10.1007/978-3-540-72521-3_21

[18] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register allocation via coloring. Computer Languages 6, 1 (1981), 47–57. https://doi.org/10.1016/0096-0551(81)90048-5

[19] Julianne Chung, Matthias Chung, J. Tanner Slagel, and Luis Tenorio. 2017. Stochastic Newton and Quasi-Newton Methods for Large Linear Least-squares Problems. https://doi.org/10.48550/ARXIV.1702.07367

[20] Thomas H. Cormen, Ronald L. Rivest, and Charles E. Leiserson. 1990. Introduction to Algorithms. McGraw-Hill, Inc.

[21] Yin Ding and Ivan W. Selesnick. 2016. Sparsity-based correction of exponential artifacts. A C MT r a n s .M a t h .S o f t w . 57, 9 (Mar. 2016), 1–17. https://doi.org/10.1145/2894667.2894676

[22] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. 1990. A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw. 16, 1 (Mar. 1990), 1–17. https://doi.org/10.1145/77626.79170

[23] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. 1988. An extended set of FORTRAN basic linear algebra subprograms. ACM Trans. Math. Softw. 14, 1 (Mar. 1988), 1–17. https://doi.org/10.1145/42288.42291

[24] John W. Eaton, David Bateman, Søren Hauberg, and Rik Wehbring. 2019. GNU Octave Version 5.1.0 Manual: A High-level Interactive Language for Numerical Computation. https://www.gnu.org/software/octave/doc/v5.1.0/.

[25] Eclipse Deeplearning4j Development Team. 2016. ND4J: Fast, scientific and numerical computing for the JVM. https://github.com/eclipse/deeplearning4j

[26] Diego Fabregat-Traver and Paolo Bientinesi. 2011. Automatic generation of loop-invariants for matrix operations. In 2011 International Conference on Computational Science and Its Applications. 82–92. https://doi.org/10.1145/1109637.1109651

[27] Diego Fabregat-Traver and Paolo Bientinesi. 2011. Knowledge-based automatic generation of partitioned matrix expressions. In Computer Algebra in Scientific Computing, Vladimir P. Gerdt, Wolfram Koepf, Ernst W. Mayr, and Evgenii V. Vorozhtsov (Eds.). Springer Berlin, Berlin, 144–157. https://doi.org/10.1007/978-3-642-23568-9_12

[28] Diego Fabregat-Traver and Paolo Bientinesi. 2013. A domain-specific compiler for linear algebra operations. In High Performance Computing for Computational Science - VECPAR 2012, Michel Daydé, Osni Marques, and Kengo Nakajima (Eds.). Springer Berlin, Berlin, 346–361. https://doi.org/10.1007/978-3-642-38718-0_33

[29] Diego Fabregat-Traver and Paolo Bientinesi. 2014. Computing petaflops over terabytes of data: The case of genome-wide association studies. ACM Trans. Math. Softw. 40, 4, Article 27 (July 2014), 22 pages. https://doi.org/10.1145/2560421

[30] Salvatore Filippone and Michele Colajanni. 2000. PSBLAS: A library for parallel linear algebra computation on sparse matrices. ACM Trans. Math. Softw. 26, 4 (Dec. 2000), 527–550. https://doi.org/10.1145/365723.365732

ACM Transactions on Mathematical Software, Vol. 48, No. 3, Article 26. Publication date: September 2022.
[31] J. Foote. 2000. Automatic audio segmentation using a measure of audio novelty. In 2000 IEEE International Conference on Multimedia and Expo. ICME2000. Proceedings. Latest Advances in the Fast Changing World of Multimedia (Cat. No.00TH8532), Vol. 1. 452–455. https://doi.org/10.1109/ICME.2000.869637

[32] Alvaro Frank, Diego Fabregat-Traver, and Paolo Bientinesi. 2016. Large-scale linear regression: Development of high-performance routines. Appl. Math. Comput. 275 (2016), 411–421. https://doi.org/10.1016/j.amc.2015.11.078

[33] Gianluca Frison, Dimitris Kouzoupiis, Tommaso Sarto, Andrea Zanelli, and Moritz Diehl. 2018. BLASPEO: Basic linear algebra subroutines for embedded optimization. ACM Trans. Math. Softw. 44, 4, Article 42 (July 2018), 30 pages. https://doi.org/10.1145/3210754

[34] Michael R. Garey and David S. Johnson. 1990. Computers and Intractability: A Guide to the Theory of NP-Completeness. S. W. Freeman & Co., USA.

[35] Mike B. Giles. 2008. Collected matrix derivative results for forward and reverse mode algorithmic differentiation. In Advances in Automatic Differentiation, Christian H. Bischof, H. Martin Bucker, Paul Hovland, Uwe Naumann, and Jean Utke (Eds.). Springer Berlin, Berlin, 35–44. https://doi.org/10.1007/978-3-540-68942-3_4

[36] Sadashiva S. Godbole. 1973. On efficient computation of matrix chain products. IEEE Trans. Comput. 22, 9 (Sep. 1973), 864–866. https://doi.org/10.1109/TC.1973.5009181

[37] Gene H. Golub, Per Christian Hansen, and Dianne P. O’Leary. 1999. Tikhonov regularization and total least squares. SIAM J. Matrix Anal. Appl. 21, 1 (1999), 185–194. https://doi.org/10.1137/S0895479897326432

[38] Claude Gomez and Tony Scott. 1998. Maple programs for generating efficient FORTRAN code for serial and vectorised machines. Computer Physics Communications 115, 2 (1998), 548–562. https://doi.org/10.1016/S0010-4655(98)00114-3

[39] Robert M. Gower and Peter Richtárik. 2017. Randomized Quasi-Newton updates are linearly convergent matrix inversion algorithms. SIAM J. Matrix Anal. Appl. 38, 4 (2017), 1380–1409. https://doi.org/10.1137/16M1062053

[40] Peter Gottschling, David S. Wise, and Michael D. Adams. 2007. Representation-transparent matrix algorithms with scalable performance. In Proceedings of the 21st Annual International Conference on Supercomputing (ICS’07). ACM, New York, NY, USA, 116–125. https://doi.org/10.1145/1274971.1274989

[41] Robert M. Gower and Peter Richtárik. 2017. Randomized Quasi-Newton updates are linearly convergent matrix inversion algorithms. SIAM J. Matrix Anal. Appl. 38, 4 (2017), 1380–1409. https://doi.org/10.1137/16M1062053

[42] Joseph L. Greathouse, Kent Knox, Jakub Pola, Kiran Varaganti, and Mayank Daga. 2016. CLSPARSE: A vendor-optimized open-source sparse BLAS library. In Proceedings of the 4th International Workshop on OpenCL (IWOCL ’16). Association for Computing Machinery, New York, NY, USA, Article 7, 4 pages. https://doi.org/10.1145/2909437.2909442

[43] Gaël Guennebaut, Benoit Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

[44] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. 2001. FLAME: Formal linear algebra methods environment. ACM Trans. Math. Softw. 27, 4, Article 12 (May 2008), 25 pages. https://doi.org/10.1145/1356052.1356053

[45] Alvaro Frank, Diego Fabregat-Traver, and Paolo Bientinesi. 2016. Large-scale linear regression: Development of high-performance routines. Appl. Math. Comput. 275 (2016), 411–421. https://doi.org/10.1016/j.amc.2015.11.078

[46] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: Accelerating small matrix multiplications by runtime code generation. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC’16). IEEE Press, Article 84, 11 pages.

[47] Nicholas J. Higham. 2002. Accuracy and Stability of Numerical Algorithms (2nd ed.). Society for Industrial and Applied Mathematics, USA. https://doi.org/10.1137/030902333

[48] Klaus Iglberger, Georg Hager, Jan Treibig, and Ulrich Rüde. 2012. Expression templates revisited: A performance analysis of current methodologies. SIAM Journal on Scientific Computing 34, 2 (2012), C42–C69. https://doi.org/10.1137/110830125

[49] International Business Machines Corporation (IBM). [n.d.]. Engineering and Scientific Subroutine Library (ESSL). https://www.ibm.com/docs/en/essl.

[50] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Martin H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. Nature 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[51] Alexander Heinecke, Greg Henry, Maxwell T. Hutchinson, and Hans Pabst. 2016. LIBXSMM: Accelerating small matrix multiplications by runtime code generation. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC’16). IEEE Press, Article 84, 11 pages.

[52] Nicholas J. Higham. 2002. Accuracy and Stability of Numerical Algorithms (2nd ed.). Society for Industrial and Applied Mathematics, USA. https://doi.org/10.1137/030902333

[53] Peter Kabal. 2011. Minimum mean-square error filtering: Autocorrelation/covariance, general delays, and multirate systems. (2011).

[54] Rudolf Emil Kalman. 1960. A new approach to linear filtering and prediction problems. Journal of Basic Engineering 82, 1 (03 1960), 35–45. https://doi.org/10.1115/1.3662552

ACM Transactions on Mathematical Software, Vol. 48, No. 3, Article 26. Publication date: September 2022.
[54] Daniel Langr and Pavel Tvrdík. 2016. Evaluation criteria for sparse matrix storage formats. *IEEE Transactions on Parallel and Distributed Systems* 27, 2 (2016), 428–440. https://doi.org/10.1109/TPDS.2015.2401575

[55] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.* 5, 3 (Sep. 1979), 308–323. https://doi.org/10.1145/355841.355847

[56] Daniel Langr and Pavel Tvrdík. 2016. A basic linear algebra compiler for structured matrices. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO’16)*. Association for Computing Machinery, New York, NY, USA, 327–339. https://doi.org/10.1145/3061664.3061701

[57] Bryan Marker, Don Batory, and Robert van de Geijn. 2013. A case study in mechanically deriving dense linear algebra code. *The International Journal of High Performance Computing Applications* 27, 4 (2013), 440–453. https://doi.org/10.1177/1094342013492178

[58] Milos Nikolic, Mohammed ElSeidy, and Christoph Koch. 2014. LINVIEW: Incremental view maintenance for complex analytical queries. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD’14)*. ACM, New York, NY, USA, 253–264. https://doi.org/10.1145/2588555.2610519

[59] Elias D. Nino-Ruiz, Adrian Sandu, and Xinwei Deng. 2015. A Parallel Ensemble Kalman filter implementation based on modified Cholesky decomposition. In *Proceedings of the 6th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA’15)*. Association for Computing Machinery, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/2832080.2832084

[60] Kazufumi Nishida, Yasuaki Ito, and Koji Nakano. 2011. Accelerating the dynamic programming for the matrix chain product on the GPU. In *2011 Second International Conference on Networking and Computing*. 320–326. https://doi.org/10.1016/j.icnc.2011.62

[61] Cedric Nugteren. 2018. CLBlast: A tuned OpenCL BLAS library. In *Proceedings of the International Workshop on OpenCL (IWOC’18)*. ACM, New York, NY, USA, 5:1–5:10. https://doi.org/10.1145/3204919.3204924

[62] NVIDIA Corporation. [n.d.]. cuBLAS. https://developer.nvidia.com/cublas.

[63] Jae Young Park, Han Lun Yap, Christopher J. Rozell, and Michael B. Wakin. 2011. Concentration of measure for block diagonal matrices with applications to compressive signal processing. *IEEE Transactions on Signal Processing* 59, 12 (2011), 5859–5875. https://doi.org/10.1109/TSP.2011.2166546

[64] Elmar Peise and Paolo Bientinesi. 2017. Algorithm 979: Recursive algorithms for dense linear algebra: The ReLAPACK collection. *ACM Trans. Math. Softw.* 44, 2, Article 16 (Sep. 2017), 19 pages. https://doi.org/10.1145/3061664

[65] Elmar Peise, Diego Fabregat-Traver, and Paolo Bientinesi. 2015. On the performance prediction of BLAS-based tensor contractions. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond (Eds.). Springer International Publishing, Cham, 193–212. https://doi.org/10.1007/978-3-319-17248-4_10

[66] Kaare Brandt Petersen and Michael Syskind Pedersen. 2006. *The Matrix Cookbook*. Technical University of Denmark. Version 20051003.

[67] R Development Core Team. 2008. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org.

[68] Conrad Sanderson and Ryan Curtin. 2016. Armadillo: A template-based C++ library for linear algebra. *Journal of Open Source Software* 1, 2 (2016), 26. https://doi.org/10.21105/joss.00026

[69] Jeremy G. Siek, Ian Karlin, and E. R. Jessup. 2008. Build to order linear algebra kernels. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–8. https://doi.org/10.1109/IPDPS.2008.4536183

[70] Stephen P. Smith. 1995. Differentiation of the Cholesky algorithm. *Journal of Computational and Graphical Statistics* 4, 2 (1995), 134–147.

[71] Daniele G. Spampinato, Diego Fabregat-Traver, Paolo Bientinesi, and Markus Püschel. 2018. Program generation for small-scale linear algebra applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 327–339. https://doi.org/10.1145/3168812

[72] Daniele G. Spampinato and Markus Püschel. 2016. A basic linear algebra compiler for structured matrices. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO’16)*. Association for Computing Machinery, New York, NY, USA, 117–127. https://doi.org/10.1145/2854038.2854060

[73] Damian Straszak and Nisheeth K. Vishnoi. 2015. On a natural dynamics for linear programming. *CoRR* abs/1511.07020 (2015). arXiv:1511.07020 http://arxiv.org/abs/1511.07020

[74] Tom Tirer and Raja Giryes. 2019. Image restoration by iterative denoising and backward projections. *IEEE Transactions on Image Processing* 28, 3 (2019), 1220–1234. https://doi.org/10.1109/TIP.2018.2875569

[75] Field G. Van Zee, Ernie Chan, and Robert A. van de Geijn. 2011. *libflame*. Springer US, Boston, MA, 1010–1014. https://doi.org/10.1007/978-0-387-09766-4_91

[76] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Softw.* 41, 3, Article 14 (June 2015), 33 pages. https://doi.org/10.1145/2764454

ACM Transactions on Mathematical Software, Vol. 48, No. 3, Article 26. Publication date: September 2022.
[77] Todd L. Veldhuizen. 1998. Arrays in Blitz++. In Computing in Object-Oriented Parallel Environments, Denis Caromel, Rodney R. Oldehoeft, and Marydell Tholburn (Eds.). Springer Berlin, Berlin, 223–230. https://doi.org/10.1007/3-540-49372-7_24

[78] Joerg Walter, Mathias Koch, et al. 2006. uBLAS. Boost C++ software library available from http://www.boost.org/doc/libs (2006).

[79] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically tuned linear algebra software. In Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC’98). IEEE Computer Society, USA, 1–27.

[80] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. Commun. ACM 52, 4 (apr 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[81] Wolfram Research, Inc. [n.d.]. Mathematica, Version 12.0. https://www.wolfram.com/mathematica. Champaign, IL, 2019.

[82] Zhang Xianyi, Wang Qian, and Zhang Yunquan. 2012. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In 2012 IEEE 18th International Conference on Parallel and Distributed Systems. 684–691. https://doi.org/10.1109/ICPADS.2012.97

Received September 2021; revised April 2022; accepted May 2022