The Generalized Matrix Chain Algorithm

Henrik Barthels
AICES
RWTH Aachen University
Schinkelstr. 2
Aachen, NRW 52062, Germany
barthels@aices.rwth-aachen.de

Marcin Copik
AICES
RWTH Aachen University
Schinkelstr. 2
Aachen, NRW 52062, Germany
mcopik@gmail.com

Paolo Bientinesi
AICES
RWTH Aachen University
Schinkelstr. 2
Aachen, NRW 52062, Germany
pauldj@aices.rwth-aachen.de

Abstract

In this paper, we present a generalized version of the matrix chain algorithm to generate efficient code for linear algebra problems, a task for which human experts often invest days or even weeks of works. The standard matrix chain problem consists in finding the parenthesization of a matrix product $M := A_1 A_2 \cdots A_n$ that minimizes the number of scalar operations. In practical applications, however, one frequently encounters more complicated expressions, involving transposition, inversion, and matrix properties. Indeed, the computation of such expressions relies on a set of computational kernels that offer functionality well beyond the simple matrix product. The challenge then shifts from finding an optimal parenthesization to finding an optimal mapping of the input expression to the available kernels. Furthermore, it is often the case that a solution based on the minimization of scalar operations does not result in the optimal solution in terms of execution time. In our experiments, the generated code outperforms other libraries and languages on average by a factor of about 9. The motivation for this work comes from the fact that—despite great advances in the development of compilers—the task of mapping linear algebra problems to optimized kernels is still to be done manually. In order to relieve the user from this complex task, new techniques for the compilation of linear algebra expressions have to be developed.

CCS Concepts • **Computing methodologies → Linear algebra algorithms**; • **Software and its engineering → Compilers**; **Domain specific languages**; • **Mathematics of computing** → *Mathematical software*;

Keywords matrix chain problem, linear algebra, compiler

1 Introduction

Although the evolution of languages and compilers in the last 60 years is nothing short of remarkable, when it comes to linear algebra computations, the efficiency levels achieved by experts are still unmatched. Popular systems such as Matlab [2] and Julia [6] allow to directly express matrix expression, which are however evaluated according to general and simple rules, usually resulting in slow execution. To achieve optimal performance, it is necessary to manually map the expressions to optimized routines for basic linear algebra operations, for example as provided by libraries such as BLAS [16, 17, 30] and LAPACK [4]. To automate this task, the linear algebra compiler Linnea is developed [5].

With Linnea, the objective is twofold: On the one hand, the goal is to generate programs that come close to the efficiency achieved by human experts; on the other hand, the goal is also to achive a generation time that is only a fraction of what a human would need, and without requiring the user to possess expertise in linear algebra or high-performance computing. As input, Linnea takes expressions as described by the grammar shown in Fig. 1, in combination with a description of the operands and their properties, as shown in Fig. 2. In this paper, we are concerned with expressions consisting of products of matrices and vectors; we also allow each of the operands to carry properties, and to be transposed and/or inverted. As examples, consider $X := AB^T C$ and $x := A^{-1} By$, where A, B, C, X are matrices, and x and y are vectors; to be computed efficiently, these expressions have to be mapped onto a set K of computational kernels[1] (e.g.: `C:=A*B`, `C:= A^-1*B`, `B:= A^-1`, ...). Furthermore, the mapping has to minimize a user-selected cost metric (such as number of flops or execution time). The output is then a sequence of kernel calls that computes the original expression.

We refer to this problem as the *Generalized Matrix Chain Problem* (GMCP); the classic Matrix Chain Problem (MCP) covers the specific instances of GMCP in which the input

---

[1]The terminology is explained in Sec. 1.1.

$$\text{assignments} \rightarrow \text{assignment}^+$$
$$\text{assignment} \rightarrow symbol := \text{expr}$$
$$\text{expr} \rightarrow symbol \mid \text{expr} + \text{expr} \mid \text{expr} \cdot \text{expr} \mid$$
$$\text{expr}^{-1} \mid \text{expr}^T \mid \text{expr}^{-T}$$

Figure 1. Linnea grammar describing the definition of expressions.

expression only consists of products (without additional operators such as transposition and inversion), K contains only the kernel C:=A*B, and none of the matrices have a special structure or properties [13]. Thus, MCP only consists in finding an optimal parenthesization. The number of floating point operations (FLOPs) is used as a cost metric.[2] The applicability of the original algorithm, while well suited to study dynamic programming algorithms, is fairly limited. Having analyzed dozens of linear algebra algorithms, we observed that long matrix chains occur only rarely. On the other hand, expressions involving the product of up to about ten matrices, where some of them are transposed and/or inverted, are much more common. These more complicated matrix chains occur as part of linear algebra problems in many different fields. For example, $L_{22}^{-1}L_{21}L_{11}^{-1}L_{10}$ is part of a blocked algorithm for the inversion of a triangular matrix [7]. The chains $X_i^b S_i (Y_i^b)^T R_i^{-1}$ is encountered in the ensemble Kalman filter [43]. In an algorithm to reduce matrices to tridiagonal form, $\tau_v \tau_v v v^T A u u^T$ appears [11]. Blocked algorithms for the simultaneous solution of two linear systems, $A := L^{-1} A L^{-H}$ contain multiple chains of length 3 [37]. Further examples include the fields of computer vision [9], optimization [49], information theory [3, 25], signal processing [15, 34, 42], regularization [36] and the simulation of power grids [44].

With GMCP, the challenge shifts from finding the optimal parenthesization to finding an optimal mapping to kernels. The standard algorithm operates on numbers that represent the chain. To solve GMCP, one has to work on symbolic expressions, rely on pattern matching and on the inference of properties. In general, the solution to GMCP depends on the available kernels. For the purpose of this paper, we assume that the kernels in K offer the functionality necessary to guarantee that every input chain is computable. In practice, the de-facto standard linear algebra libraries BLAS and LAPACK offer such functionality, with kernels for matrix-matrix products and solve linear systems, with optionally transposed operands.

In typical chains, matrices often have properties and structure. This information is relevant when trying to find the optimal way to compute an expression; specialized, usually more efficient kernels can be used. This is true for matrix products, but even more so for operations involving matrix inversion. In Sec. 3.2 we explain that in order to fully take

advantage of properties and structure, not only must the GMC algorithm select kernels based on this knowledge, it also needs to propagate and infer knowledge along with intermediate results.

The MC algorithm finds the parenthesization that minimizes the FLOP count. In practice, the number of FLOPs is not always an accurate metric to assess the performance of linear algebra operations. More accurate results can be achieved by taking the "efficiency" of the kernels into account.[3] For this reason, our GMC algorithm allows to specify an arbitrary cost metric, including vector measures, according to which the optimal solution is chosen.

In this paper, we describe GMCP, and present an algorithm—the Generalized Matrix Chain (GMC) algorithm—to solve it.

Organization of the paper   The remainder of this section contains an overview of the terminology related to GMCP, as well as a discussion of previous works. The standard dynamic programming matrix chain algorithm is summarized in Sec 2. Our GMC algorithm is presented in Sec. 3, and evaluated in Sec. 4. Finally, in Sec. 5 we discuss opportunities for future work.

1.1 Terminology

Chain and Expression A valid input to the GMCP—a *"matrix chain"* or simply *"chain"*—is a product $\mathcal{M} := f_0 \cdots f_{n-1}$ where f_i is a matrix or a vector that can be transposed and/or inverted. We use $\mathcal{M}_{[ij]}$ to denote the product $f_i \cdots f_j$, which we call *"sub-chain"* or simply *"expression"*. If $i = j$, the chain consists of one single matrix and is denoted by $\mathcal{M}_{[i]}$. Notice that the grammar in Fig. 1 does not imply the correctness of expression, i.e. it does not guarantee that the dimensions of all operands match.
In this paper, we require the input to be a well-formed matrix chain of length two or higher. Vectors are considered to be matrices of size $n \times 1$ or $1 \times n$. Since scalars commute with matrices, we do not further consider them.

Property In addition to the specification shown in Fig. 1, we allow each symbol (i.e. each matrix) to be annotated with one or more properties. The grammar for the definition of the operands, including properties, is shown in Fig. 2. Frequently encountered properties are: lower and upper triangular, symmetric, diagonal, and symmetric positive-definite (SPD).

Kernel A kernel is an optimized routine for computing the solution of a well defined linear algebra problem, e.g. C:=A*B, C:= A^{-1}*B, as provided by libraries such as BLAS. Throughout this paper, we assume that there

---

[2]The cost of a product AB, with $A \in \mathbb{R}^{n \times k}$ and $B \in \mathbb{R}^{k \times m}$, is $2mnk$ FLOPs.

[3]It is often believed that the minimum execution time of an algorithm is attained by minimizing the number of floating point operations performed. This is not true, as not all flops cost the same (they are not equally "efficient".

$$\text{definitions} \rightarrow \text{definition}^+$$
$$\text{definition} \rightarrow \textbf{Matrix } name \text{ size } \langle property^* \rangle$$
$$\text{size} \rightarrow (rows, columns)$$
$$\text{property} \rightarrow \textbf{LowerTriangular} \mid \textbf{Diagonal} \mid \ldots$$

Figure 2. Grammar describing the definition of operands.

exists a set of specialized kernels that can take advantage of matrix properties.[4]

Solution A solution for the GMCP consists of a parenthesization of the input chain, in conjunction with a mapping of expressions to kernels.

Cost function This is a function to quantify the quality of a solution to the GMCP. Commonly used metrics are the total number of FLOPs, and the number of FLOPs per second, called *performance*.[5] cost($\mathcal{M}$) is used to denote the cost of computing the chain $\mathcal{M}$. The solution that minimizes this cost function is the optimal solution.

1.2 Related Work

The matrix chain problem is subject to a lot of research. The classic algorithm to solve MCP uses dynamic programming and has $O(n^3)$ complexity, where n is the length of the chain [13]. The best known algorithm, by Hu and Shing, exploits the equivalence between MCP and the triangulation of polygons to achieve $O(n \log(n))$ complexity [26, 27]. A number of approaches take parallelism into account, some using multiple processors to reduce the time needed to find the solution (which will be evaluated on a sequential system) [8, 41, 50], while others find an ordering that is optimal when the matrix chain is evaluated on a parallel system [31]. Nishida et al. present a version for GPUs [35]. Additionally, both sequential [10] and parallel [14] algorithms exist that find approximate solutions. All the aforementioned algorithms deal with the basic problem of multiplying matrices that are neither transposed nor inverted.

High-level languages such as Matlab allow to directly express instances of GMCP, without explicit parenthesization, and link to highly optimized kernels. However, they put little to no effort into mapping the mathematical problem to said kernels in a way that results in a highly efficient evaluation. By constrast, expressions are typically evaluated according to simple rules. For example, if the inverse operator is used in Matlab, then an inverse is computed explicitly, even though the mathematically equivalent solution of solving a linear

system is faster and numerically more stable; indeed, it is up to the user to rewrite the inverse in terms of the slash (/) or backslash (\) operators, to "enable" the linear systems. Furthermore, in Matlab products are always evaluated from left to right [2]. Matrix properties are considered by inspecting matrix elements at runtime. Although not documented, Mathematica applies the same strategy to evaluate expressions,[6] including the choice between explicit inversion and solution of a linear system. Recently, the Julia project [6] set out to design a language that natively integrates tools for scientific computing, including linear algebra; while high-level expressions are accepted, they are evaluated just as in Matlab. Julia uses types to represent a small set of basic properties and uses multiple dispatch to select appropriate kernels.

An alternative approach consists in the use of (smart) expression templates in C++, as employed by libraries such as Blaze [28], Blitz++ [51], and Eigen [24]. The main idea is to improve performance by eliminating temporary operands and provide a domain-specific language integrated within C++. However, similar to high-level languages, expressions are evaluated according to very simple rules. To some extent, the C++ library Armadillo is an exception [45]: It uses a simple algorithm which is not guaranteed to find the best solution to the MCP (the algorithm is discussed in detail in Sec. 4). Moreover, the choice regarding the treatment of the inverse operator is again left to the user. Similar to Julia, matrix properties are represented by types.

Recognizing that BLAS is not optimal across the full spectrum of operations and problem sizes, some compilers such as Build to Order (BTO) [46] and LGen [47, 48] aim at generating directly code, without relying on standard building blocks. BTO specializes in bandwidth-bound operations (BLAS 1 and 2), while LGen focuses on small-scale problems for which BLAS usually performs poorly. Finally, one could approach the problem by systematically partitioning the input matrices, thus originating problems that fit exactly in a target cache level [18, 19].

In principle, GMCP (and thus also MCP) can be solved by means of a search-based approach, as the one adopted in the linear algebra compiler presented in [20, 21] (CLAK), which exclusively relies on pattern matching. This approach has two drawbacks: The type of pattern matching that CLAK uses is expensive, and due to the search-based nature, the number of explored solutions is exponential in the length of the chain, even for the standard matrix chain problem [13].

2 The Standard Matrix Chain Algorithm

MCP can be elegantly solved with a dynamic programming approach, both in a top-down and a bottom-up fashion [13].

---

[4]As an example, consider the BLAS kernels GEMM, TRMM and SYMM, which all compute the product of two matrices. GEMM computes a general matrix-matrix product, while TRMM and SYMM require one operand to be triangular and symmetric, respectively. Compared to GEMM, TRMM and SYMM perform half of the scalar operations.

[5]This use of *performance* is, admittedly, a somewhat unfortunate choice of terminology adopted in the HPC community.

[6]This can be easily tested by comparing the time necessary to evaluate $M_0 \cdots M_{k-1} x$ and $y M_0 \cdots M_{k-1}$, with $M_i \in \mathbb{R}^{n \times n}$, $x \in \mathbb{R}^{n \times 1}$ and $y \in \mathbb{R}^{1 \times n}$.

for $l \in \{1, \ldots, n-1\}$:
  for $i \in \{0, \ldots, n-l-1\}$:
    $j := i + l$
    for $k \in \{i, \ldots, j-1\}$:
      c := 2*sizes[i]*sizes[$k+1$]*sizes[$j+1$]
      cost := costs[i][k] + costs[$k+1$][j] + c
      if cost < costs[i][j]:
        costs[i][j] := cost
        solution[i][j] := k

Figure 3. The matrix chain algorithm.

Here, we briefly explain the bottom-up version, as it is the foundation for the algorithm presented in this paper.

Consider the chain $X := ABCDE$ as an example. The algorithm proceeds by finding the optimal parenthesization for parts of this chain of increasing length, using the optimal solutions for sub-chains. Let us assume the algorithm already computed all solutions for sub-chains of length up to three. The next step consists of computing solutions for sub-chains of length four. $\mathcal{M}_{[0,4]} = ABCDE$ has two such sub-chains, $\mathcal{M}_{[0,3]} = ABCD$ and $\mathcal{M}_{[1,4]} = BCDE$. Let us illustrate the step for $ABCD$: There are three different ways to write this chain as a product of two shorter chains, or, to put it differently, three ways to split $\mathcal{M}_{[0,3]}$ into $\mathcal{M}_{[0,k]}\mathcal{M}_{[k+1,3]}$, namely for $k \in \{0, 1, 2\}$: $A(BCD)$, $(AB)(CD)$ and $(ABC)D$. The algorithm assigns a cost to all those products, and stores the best solution together with its cost. The cost for $A(BCD)$ is the cost of computing $\mathcal{M}_{[0,0]} = A$, plus the cost of $\mathcal{M}_{[1,3]} = BCD$, plus the cost of the product of A and the result of BCD. The cost of $\mathcal{M}_{[0,0]}$ is known to be zero, and cost($\mathcal{M}_{[1,3]}$) was already computed in a previous step because the length of $\mathcal{M}_{[1,3]}$ is three. The same is done for $(AB)(CD)$, $(ABC)D$, as well as all possible ways to split $\mathcal{M}_{[1,4]}$. At this point, the algorithm uses all the results from the previous steps to find the best way to express $ABCDE$ as a product of two shorter parts.

The algorithm is shown in Fig. 3. The following arrays are used, where solution and costs have size $n \times n$, sizes is of size $n + 1$:

solution The entry solution[i][j] stores the integer k which specifies the optimal split for $\mathcal{M}_{[i,j]}$. This array has the exact same role as the s array in [13].

costs The value of costs[i][j] is the minimal cost for the computation of the sub-chain $\mathcal{M}_{[i,j]}$. The entries costs[i][i] are initialized to 0, while all other fields are initialized with ∞. This array has exactly the same role as the m array in [13].

sizes This array contains the operand sizes. sizes[0] contains the number of rows of $\mathcal{M}_{[0]}$. For $i > 0$, sizes[i] stores the number of columns of $\mathcal{M}_{[i-1]}$.

1 **for** $l \in \{1, \ldots, n-1\}$:
2   **for** $i \in \{0, \ldots, n-l-1\}$:
3      $j := i + l$
4   **for** $k \in \{i, \ldots, j-1\}$:
5   expr := tmps[i][k] $\times$ tmps[$k+1$][j]
6   kernel := match(expr)
7   cost := costs[i][k] + costs[$k+1$][j] + kernel.cost
8   **if** cost < costs[i][j]:
9   tmps[i][j] := create_tmp(expr)
10   tmps[i][j].properties := infer_properties(expr)

11   kernels[i][j] := kernel
12   costs[i][j] := cost
13   solution[i][j] := k

Figure 4. The GMC algorithm.

3 The GMC Algorithm

3.1 Unary Operators

In its standard version, the matrix chain algorithm only works with binary, non-commutative operators. To extend it to unary operators, we observe that compositions of binary and unary operators on two operands can still be seen as (an extended set of) binary operators. In fact, as long as it is possible to assign a cost to those compositions of operations, the dynamic programming approach remains applicable. The algorithm, however, becomes more complex because in addition to the parenthesisation, it also has to identify which kernels can be applied and when. To solve this problem, the GMC algorithm works on symbolic expressions, which are represented as expression trees. Operands have a name, a size and a set of properties (see Sec. 3.2). Instead of the one-dimensional array sizes, we now use the $n \times n$ array tmps, which is used to store store symbolic temporary variables representing sub-chains. In the following, consider the chain $\mathcal{M} = A^{-1}BC^T$ as an example. tmps[i][j] contains the temporary that represents $\mathcal{M}_{[i,j]}$. The entry tmps[i][i] is initialized with a symbolic representation of the matrix $\mathcal{M}_{[i]}$. For example tmps[0][0] is A^{-1}. When the algorithm terminates, tmps[1][2] contains a temporary T_{12} that represents BC^T. The symbols representing those operands are used to create the expressions that have to be computed. For $i = j = 0$, $k = 2$, expr is the expression tmps[0][0] $\times$ tmps[1][2] $= A^{-1}T_{12}$, which corresponds to the parenthesization $A^{-1}(BC^T)$. New temporaries are created by the function create_tmp (line 9), which creates an operand with a unique name and correct sizes.

To select a suitable kernel, our algorithm relies on pattern matching as offered by MatchPy [29], a Python library

Table 1. Examples of patterns for BLAS kernels.

Name	Pattern	Constraints	Cost
GEMM	XY	-	$2mnk$
TRMM	XY	is_lower_triangular(X)	m^2n
SYMM	XY	is_symmetric(X)	m^2n
TRSM	$X^{-1}Y$	is_lower_triangular(X)	m^2n
SYRK	X^TX	-	m^2k

that implements discrimination nets, data structures for efficient syntactic many-to-one pattern matching [12, 23, 33]. In many-to-one pattern matching, a set of patterns and one expression are given, and it is tested whether or not any of those patterns matches the expression. In our case, the set of patterns is the set of kernels K. Some examples are shown in Table 1. In the next section, we discuss the case in which more than one kernel matches the target expression. In the pseudocode in Fig. 4, pattern matching appears in line 6.

To store the solution, in addition to the `solutions` array—which contains the information on the parenthesization—it is necessary to also keep track of the kernel used for the operation. In the MC algorithm, this is not necessary because the kernel is always the same. For this purpose, we introduce the $n \times n$ `kernels` array, whose entry `kernels[i][j]` contains the kernel that is used to compute the temporary `tmps[i][j]`. In the end, the kernels are used to generate the code (see Sec. 3.5). To simplify the discussion, here we only consider direct solvers for linear systems, using matrix factorizations if necessary. While the GMC algorithm can also be applied to sparse linear algebra and iterative solvers, the kernel selection becomes even more challenging [32].

3.2 Properties

Many linear algebra operations can be sped up by taking advantage of the properties of the involved matrices. As a most basic example, the multiplication of a lower triangular matrices with a full matrix requires m^2n scalar operations, as opposed to $2m^2n$ operations for the multiplication of two full matrices of the same sizes; likewise, a linear system $A^{-1}B$ where A is symmetric positive definite can be solved faster than a system where A does not have any special properties.

Many properties are not mutually exclusive. As an example, a matrix can be banded and symmetric at the same time. Thus, it is possible that the same expression can be computed by multiple kernels. Whenever more than one kernel matches (`match` function, line 6), the algorithm selects the kernel that minimizes the cost function (cost functions are discussed in the next section).

To fully take advantage of properties, it is certainly important to select the best matching kernel, but it is even more critical to keep track of how structure and properties propagate throughout the intermediate results, as different kernels are applied. Take the product AB^T as an example. If it is possible to assert that B is upper triangular, B^T is known to be
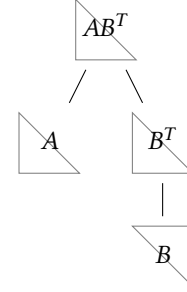
Figure 5. Example for the propagation of properties in AB^T where A is lower and B is upper triangular.

---

def is_lower_triangular(expr):
  if expr **is** Times:
    return $\forall$child $\in$ expr.children :
                              is_lower_triangular(child)
  if expr **is** Transpose:
    return is_upper_triangular(expr.child)
  if expr **is** Matrix:
    if LowerTriangular **in** expr.properties:
      return True
    else:
      return False
  # ...

Figure 6. Pseudocode implementation of the function is_lower_triangular.

lower triangular. Furthermore, the product of two lower triangular matrices is still lower triangular. Thus, if A is lower triangular, the entire expression AB^T has this property. Note that this property is independent of how AB^T is computed, and it can be inferred without actually computing the result, solely by inspecting the symbolic expression.

This knowledge about properties can naturally be represented by inference rules, as for example

$$\text{LoTri}(A) \wedge \text{LoTri}(B) \rightarrow \text{LoTri}(AB),$$
$$\text{LoTri}(A) \wedge \text{Diag}(B) \rightarrow \text{LoTri}(AB),$$
$$\text{LoTri}(A) \rightarrow \text{UppTri}\left(A^T\right).$$

In the GMC algorithm, the function `infer_properties` (line 10) is responsible for the inference of properties. Intuitively, matrix properties are propagated from the bottom to the top of the expression tree. An example is shown in Fig. 5. This is done by recursively traversing the symbolic expression tree `expr`. In practice, this is implemented as a set of functions, with a dedicated function for each property. A part of the function `is_lower_triangular` is shown in Fig. 6.

Some languages, for example Matlab, do not infer properties symbolically but test for them by inspecting all entries of a matrix. The symbolic inference of properties has additional advantages over this approach: The cost is independent of the matrix size. Furthermore, some properties might be masked by numerical inaccuracies, which can have a considerable impact on subsequent computations: A generalized eigenproblem $Ax = \lambda Bx$ is typically solved via a reduction to a standard eigenproblem $A'y = \lambda y$; this is done by computing the expression $A' := L^{-1}AL^{-T}$, where L is lower triangular and A is symmetric; in floating point arithmetic, if A' is computed by solving two linear systems, symmetry is lost; thus, when computing the eigenvalues of A', one can only use a non-symmetric eigensolver, which is about three times more expensive than a symmetric one [22], and worse yet, will deliver complex eigenvalues (with a very small imaginary part), doubling the amount of output data.

Let us consider the matrix chain $X := A^T AB$ as an example, where $A \in \mathbb{R}^{n \times n}$ and $B \in \mathbb{R}^{n \times m}$ are dense matrices with $n = 20$ and $m = 15$, using the FLOP count as a metric. The first possible solution is to compute X via two general matrix-matrix products:

$$W := AB,$$
$$X := A^T W.$$

In this case, W is of size $n \times m$ and has no special properties. Both products require $2n^2 k = 12000$ FLOPs; the overall cost of this solution is 24000 FLOPs. The alternative solution results in an intermediate $W \in \mathbb{R}^{n \times n}$ that is symmetric and positive definite:

$$W := A^T A,$$
$$X := WB.$$

If $A^T A$ is computed as a general matrix-matrix product, the cost is $2n^3 = 16000$ FLOPs. Disregarding that W is symmetric and computing WB as another general matrix-matrix product, we get $2n^2 k = 12000$ additional FLOPs. By taking advantage of the property instead, one computes this product with half the number of FLOPs ($n^2 k = 6000$). Hence, the solution obtained by using a specialized kernel has a cost of 22000 FLOPs, compared to 28000 FLOPs. This example shows that properties not only lead to better solutions, but also to solutions that might differ in the parenthesization. Note: instead of computing $A^T A$ as a general matrix-matrix product, it is also possible to use the specialized SYRK kernel, performing half the number of FLOPs.

3.3 Cost Functions

It is well understood that properties such as temporal and spacial locality impact the execution time of an algorithm as much as, or even more than the number of FLOPs [16]. In fact, two solutions that are identical in terms of FLOPs might have very different actual execution times; it might

even be the case that the faster algorithm performs significantly more FLOPs than the slower one. One such example is the reduction of a symmetric matrix to tridiagonal form [22]. As a second example, consider the matrix chain $ABCDE$ with matrix sizes (from left to right) 130, 700, 383, 1340, 193 and 900. The parenthesization that results in the smallest number of FLOPs is $(((AB)C)D)E$ with 3.16×10^8 FLOPs. The parenthesization that results in the shortest execution time is $((AB)(CD))E$, even though the number of FLOPs is slightly higher with 3.32×10^8. With an execution time of 7.6 milliseconds, the first parenthesization is about 10% slower than the second parenthesization, which takes 6.8 milliseconds.[7] That said, the FLOP count is still the most commonly used metric in practice; this is certainly because of its simplicity, but also because the performance modeling and prediction of linear algebra kernels remains a challenging, largely unsolved problem [38].

We want the GMC algorithm to use an arbitrary metric. To take into account the different operations, a cost function has to be provided that is defined for the different kernels. Once a kernel is identified, the function is used to determine its cost. This is done based on the sizes of the operands and their properties.

A more accurate metric than the FLOP count is the performance (in FLOPs/sec); if the execution time of the matrix chain algorithm is of no concern (Note: this is the time it takes to find the optimal mapping, and not the execution time of the resulting algorithm), real measurements could be used, for example using performance modeling tools such as ELAPS [40]. However, these approaches are still not able to accurately predict the performance of an entire chain. In fact, performance is not composable, which means that the combination of the performance of two kernels executed separately will not be the same as the performance of the same kernels executed back to back [39]. One of the reasons for this is the state of the cache. Despite this hurdle, performance is still a better approximation than the number of FLOPs.

A cost function can also take into account accuracy: The explicit inversion of matrices should be avoided if it is possible to solve a linear system instead, both for performance and stability reasons. Since explicit inversion is more expensive, using a cost metric based on performance automatically leads to a solution that favors the solution of linear systems.

3.4 Complexity and Completeness

The functions used in the GMC algorithm and their complexity are described below. For considerations regarding the time complexity, it is important to note that the size of the expression tree representing expr is limited. The most complex expressions have the form $f_1(A) \cdot f_2(B)$, with f being the transposition, inversion, or the combination of both,

---

[7]The presented times are the minimum out of 100 repetitions, using BLAS wrappers in Julia 0.6.1 on an Intel Core i5 with 2,7 GHz.

and operands A and B. Thus, those trees have at most five nodes and three levels. The same is true for the size of the patterns, as kernels that compute more complex expressions than $f_1(A) \cdot f_2(B)$ are not applicable.

match The complexity of syntactic pattern matching with discrimination nets does not depend on the number of patterns and is bounded by the size of the patterns, which in our case is constant. It follows that the complexity of pattern matching is $O(1)$.

create_tmp This function creates a symbolic temporary matrix that represents the result of computing $\mathcal{M}_{[i,j]}$. For example, for an outer product ab^T, with $a \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$, a temporary matrix $T \in \mathbb{R}^{n \times m}$ will be created. This function creates a symbolic object with a unique name and correct sizes. The size is determined by traversing the expression tree, that is bounded in its size by a constant, so this function is in $O(1)$.

infer_properties Since the size of the expression trees is limited by a small constant, this function has a complexity of $O(p)$, where p is the number of properties. Those functions are then also used for the constraints of the patterns that represent kernels (Table 1).

The loop body is executed $O(n^3)$ times, where n is the length of the matrix chain (see [13]). Thus, the complexity of the entire algorithm is $O(n^3 + n^3 p)$. This could be further reduced to $O(n^3 + n^2 p)$ by inferring properties outside of the k loop only for the temporary that might be used in the solution.

Completeness We stress that the GMC algorithm might deliver a solution even if one or more sub-chains are not computable because no suitable kernel is found. Let us assume we are given the matrix chain $X := A^{-1}B^{-1}C$, and there is no kernel that computes $X^{-1}Y^{-1}$, so $A^{-1}B^{-1}$ can not be computed. In this case, find_sequence would return ∞ as the cost of $A^{-1}B^{-1}$. However, this chain can still be computed by solving two linear systems:

$$T := B^{-1}C$$
$$X := A^{-1}T.$$

In general, the GMC algorithm will find a solution if there is at least one parenthesization such that all exposed binary operations can be computed.

3.5 Code Generation

Retrieving the sequence of kernels that was identified as the optimal solution is done by calling construct_solution($0, n - 1$), where n is the length of the chain. The function construct_solution is show in Fig. 2. The kernels are returned in an order that respects dependencies. However, in some cases, kernel calls can be reordered. This is for example the case for the chain $(AB)(CD)$, where AB and CD can be computed independently. Since performance is not composable, different orders likely result in different performance;

```
def construct_solution(i, j):
    if i ≠ j:
        yield from construct_solution(i, solution[i][j])
        yield from construct_solution(solution[i][j]+1, j)
        yield kernels[i][j]
```

Figure 7. Function to construct the solution. `yield` and `yield from` behave as the corresponding Python keywords.

the prediction of the best ordering is again a difficult task and it could be added as a final optimization.

4 Results

To evaluate the quality of the algorithms generated by the GMC algorithm, we compare against Julia[8], Matlab[9], Eigen[10], Blaze[11] and Armadillo[12]. We link against the Intel MKL implementation of BLAS and LAPACK (MKL 2017 update 3) [1], with the exception of Matlab, which instead uses LAPACK 3.5.0 and MKL 11.3.1. The GMC algorithm generates Julia code that uses BLAS and LAPACK wrappers. As a cost metric, FLOPs are used. When possible, we consider two different implementations for each library and language: `naive` and `recommended`. The `naive` implementation is the one that comes closest to the mathematical description of the problem. As an example, in Julia $A^{-1}B$ is implemented as `inv(A)*B`. However, since the documentations almost always discourage this use of the inverse operator, we also consider a so called `recommended` implementation, which uses dedicated functions to solve linear systems (`A\B`).

In the following, we describe the different implementations. As examples, in Table 2 we provide the implementations of $A^{-1}BC^T$ where A is symmetric positive definite and C is lower triangular.

Julia Properties are expressed via types. The `naive` implementation uses `inv()`, while the recommended one uses the `/` and `\` operators.

Matlab The `naive` implementation uses `inv()`, the recommended the `/` and `\` operators.

Eigen The recommended implementation uses the recommended linear systems solvers based on the matrix properties, as well as views to describe properties.

Armadillo In the `naive` implementation, specialized functions are used for the inversion of SPD and diagonal matrices. For `solve`, we use the `solve_opts::fast` option to disable an expensive refinement. In addition, `trimatu` and `trimatl` are used for triangular matrices.

Table 2. Implementations of $A^{-1}BC^T$.

Name	Implementation
GMC	`trmm!('R', 'L', 'T', 'N', 1.0, C, B)`
	`posv!('L', A, B)`
Jl n	`inv(A)*B*C'`
Jl r	`(A\B)*C'`
Arma n	`arma::inv_sympd(A)*B*(C).t()`
Arma r	`arma::solve(A, B)*C.t()`
Eig n	`A.inverse()*B*C.transpose()`
Eig r	`A.llt().solve(B)*C.transpose()`
Bl n	`blaze::inv(A)*B*blaze::trans(C)`
Mat n	`inv(A)*B*C'`
Mat r	`(A\B)*C'`
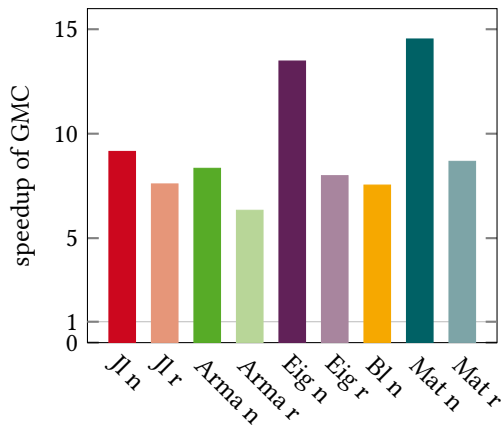
Figure 8. Average speedup of the GMC-generated code over other libraries and languages.

Blaze Since Blaze does not offer functions to solve linear systems, there is no `recommended` implementation. Properties are specified by adaptors.

The example problems are generated randomly, to include a mix of square and rectangular matrices as well as vectors. The length of the chains is chosen uniformly in the range [3, 10]. Matrices can be transposed and/or inverted, and may have one of the following properties: Diagonal, lower triangular, upper triangular, symmetric, symmetric positive definite. Matrix sizes are chosen uniformly from between 50 and 2000 in steps of 50. For the experiments, we use 100 chains. The measurements were taken on an Intel Xeon E5-2680 v3 with 2.5 GHz and 64 GB RAM. All experiments were repeated 20 times; and the average is reported.

The GMC algorithm takes on average 0.03s to generate solutions, and in all cases less than 0.07s. Thus, it would even be possible to use the GMC algorithm in an interactive environment. Notice that the generation time does not depend on matrix sizes.

The average speedup of the GMC-generated code over the other libraries and languages is between 6 and 15, as shown in Fig. 8. One can observe that the execution times of Julia, Armadillo and Blaze are comparable. The naive implementations in Eigen and Matlab are noticeably slower. As expected, the recommended implementations perform better than the corresponding naive implementations. In general, Armadillo emerges as the second fastest solution. This is likely because Armadillo, unlike all the other systems, uses a heuristic to finde better solutions for the matrix chain problem.

Matrix Chains in Armadillo As mention before, Armadillo is the only system that considers the matrix chain problem to some extent, using a simplified algorithm to solve it. For a chain $ABCD$, $(ABC)D$ is chosen if ABC is smaller in size than BCD. Otherwise, $A(BCD)$ is used. Similarly, for a chain ABC, either $(AB)C$ or $A(BC)$ is chosen, depending on the sizes of AB and BC. Chains with more than four matrices are broken down into chains of length $n \leq 4$. This happens in a deterministic way that depends on how expression templates are constructed in Armadillo. Using this method, not all parenthesizations can be found; $(AB)(CD)$ is not possible. However, parenthesizations found by this algorithm have the advantage that they have good caching behavior: Every binary product uses the result of the previous one. As an example, consider $A((BC)D)$, which results in the following sequence of kernels:

$$T_1 := BC$$
$$T_2 := T_1 D$$
$$T_3 := AT_2$$

The execution times of all experiments are shown in Fig. 9. The average multiplication time of the GMC solutions is 0.13s, ranging from 0.0002s to 0.9s. For 86% of the test cases, the code generated by the GMC algorithm is the fastest. In those cases when the GMC implementation is not the fastest, it is never more than a factor of 1.66 worse than the best solution. In only 4% of the test cases, other solutions are more than 1.1 times faster than the generated code. In at least 10% of the test cases (from a minimum of 10% for Armadillo `recommended` to 25% for Eigen `naive`), the other implementations are more than 10 times slower than the GMC solutions. In the worst case, the naive Eigen and Matlab solutions are about 200 times slower than the best solution. The maximal speedup of GMC implementations over other solutions depends both on the length of the chains and matrix sizes. Since it is possible for the GMC algorithm to find solutions with lower asymptotic complexity, the speedup of GMC implementations can potentially become arbitrarily large.

Inspecting the cases where the GMC-generated code is not the fastest reveals some patterns. There are multiple cases with chains of the form $M_1 \cdots M_{n-1} M_n v_1 v_2^T$, where v_1 and v_2 are vectors. The best algorithm (in terms of FLOPs) for this type of chain is usually to first compute a sequence of
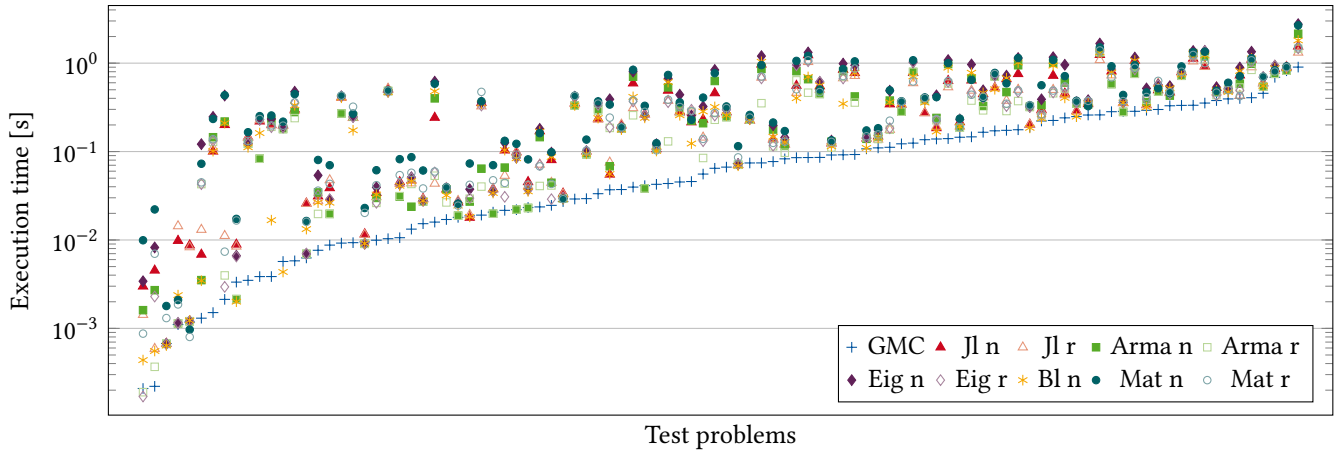
Figure 9. Execution times of all test problems (sorted by the execution time of the GMC-generated code).

matrix-vector products

$$t_1 := M_n v_1$$
$$t_2 := M_{n-1} t_1$$
$$\ldots$$
$$t_n := M_1 t_{n-1}$$

and then compute the outer product $t_n v_2^T$. This is the solution that the GMC algorithms finds, but also the one used by Armadillo, Blaze and Eigen. In Armadillo, this sequence is found by the heuristic for matrix chains. While Blaze does not solve the matrix chain problem, products of the form ABv, where v is a vector, are computed as $A(Bv)$ [28], resulting in the same sequence of kernels. For a chain $M_1 M_2^{-1} v_1 v_2^T$, the recommended Eigen implementation performs well because of the inverse operator: Using the `.solve()` method in Eigen to solve the linear system result in the parenthesization $(M_1(M_2^{-1} v_1))v_2^T$. Measuring the execution time of the individual kernels reveals that Armadillo, Blaze and Eigen have an implementation of an outer product that is significantly faster than the BLAS implementation used in the GMC implementation.

For all remaining test cases where other solutions are more than 5% faster than the GMC-generated code, the evaluation from left to right happens to be optimal (or almost optimal) in terms of FLOPs. As a result, all implementations use the same (or comparable) parenthesizations. Thus, in can be concluded that other implementations outperform the GMC-generated code because the respective languages and libraries use faster implementations of some kernels, or because the GMC-generated code contains some overhead. This also means that for the presented test cases, the cost function is sufficiently accurate to find good solutions.

5 Conclusion and Future Work

In this paper, we introduce a number of extensions to the standard matrix chain algorithm to generate efficient algorithms and code for classes of problems as they commonly occur in actual applications. The extensions include additional operations like transposition and solution of linear systems, the use of matrix properties, which are necessary to take advantage of specialized kernels and a flexible cost function. Our experimental results provide evidence that the algorithms generated by the GMC algorithm substantially outperform existing libraries and languages.

In summary, this paper makes the following contributions.

- In the MCP, expressions only consist of products. We demonstrate how the original algorithm can be extended to deal with more complex expressions, which also involve unary operators, by treating compositions of unary and binary operators as an extended set of binary operators.
- We discuss the use of properties and the importance of propagating them to automatically map linear algebra expressions to sequences of specialized kernels that can take advantage of said properties.
- We provide evidence that the GMC algorithm is a useful tool for the generation of efficient code for practical linear algebra problems. In addition to being part of a compiler for linear algebra, it could even be used in interactive environments such as Julia.

For the linear algebra compiler Linnea [5], the GMC algorithm allows to find good solutions while at the same time keeping the search space relatively small.

The GMC algorithm can be extended even further in multiple ways. For the purpose of this paper, we assumed that a kernel for $X := A^{-1} B^{-1}$ is provided. In practice, such kernels do not exists. Instead of manually constructing them from existing BLAS and LAPACK kernels, it would also be possible to again use Linnea to generate them when necessary,

as this is the exact type of problem that Linnea solves. Of course, in that case one can easily expand the set of operations even further, adding unary operators like complex conjugation, matrix logarithm or exponentiation and other non-commutative binary operators.

In general, the metric does not have to be a measure of the execution time; it can be a measure of numerical accuracy, memory consumption, number of bytes moved, or a combination of multiple objectives. It is also possible to use a vector for the metric, as long as addition and a total ordering is defined on the vector space.

Acknowledgments

References

[1] 2017. Intel®Math Kernel Library documentation. https://software. intel.com/en-us/mkl-reference-manual-for-c. (2017).

[2] 2017. Matlab documentation. http://www.mathworks.com/help/ matlab. (2017).

[3] Zaid Albataineh and Fathi M. Salem. 2014. A Blind Adaptive CDMA Receiver Based on State Space Structures. *CoRR* abs/1408.0196 (2014).

[4] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, A McKenney, and D Sorensen. 1999. *LAPACK Users' guide*. Vol. 9. SIAM.

[5] Henrik Barthels. 2016. A Compiler for Linear Algebra Operations. In *SPLASH '16 Companion*. ACM. http://doi.org/10.1145/2984043.2998539 Student Research Competition.

[6] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. 2012. Julia: A Fast Dynamic Language for Technical Computing. (September 2012). arXiv:1209.5145

[7] 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

[8] Phillip G. Bradford, Gregory J.E. Rawlins, and Gregory E. Shannon. 1998. Efficient Matrix Chain Ordering in Polylog Time. *SIAM J. Comput.* 27, 2 (1998), 466–490.

[9] Alexander M. Bronstein, Yoni Choukroun, Ron Kimmel, and Matan Sela. 2016. Consistent Discretization and Minimization of the L1 Norm on Manifolds. *CoRR* abs/1609.05434 (2016). arXiv:1609.05434

[10] Francis Y. Chin. 1978. An O(N) Algorithm for Determining a Near-optimal Computation Order of Matrix Chain Products. *Commun. ACM* 21, 7 (July 1978), 544–549. https://doi.org/10.1145/359545.359556

[11] Jaeyoung Choi, Jack J Dongarra, and David W Walker. 1995. The Design of a Parallel Dense Linear Algebra Software Library: Reduction to Hessenberg, Tridiagonal, and Bidiagonal Form. *Numerical Algorithms* 10, 2 (1995), 379–399.

[12] Jim Christian. 1993. Flatterms, Discrimination Nets, and Fast Term Rewriting. *Journal of automated reasoning* 10, 1 (1993), 95–113.

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

[14] Artur Czumaj. 1996. Very Fast Approximation of the Matrix Chain Product Problem. *Journal of Algorithms* 21, 1 (1996), 71–79.

[15] Yin Ding and Ivan W. Selesnick. 2016. Sparsity-Based Correction of Exponential Artifacts. *Signal Processing* 120 (2016), 236–248.

[16] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. 1990. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software (TOMS)* 16, 1 (1990), 1–17.

[17] Jack J. Dongarra, Sven Hammarling, Richard J. Hanson, and Jeremy Croz. 1988. An Extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software (TOMS)* 14, 4 (1988), 399.

[18] Diego Fabregat-Traver and Paolo Bientinesi. 2011. Automatic Generation of Loop-Invariants for Matrix Operations. In *Computational Science and its Applications, International Conference*. IEEE Computer Society, Los Alamitos, CA, USA, 82–92. http://hpac.rwth-aachen.de/ aices/preprint/documents/AICES-2011-02-01.pdf

[19] Diego Fabregat-Traver and Paolo Bientinesi. 2011. Knowledge-Based Automatic Generation of Partitioned Matrix Expressions. *CASC* 6885, 4 (2011), 144–157.

[20] Diego Fabregat-Traver and Paolo Bientinesi. 2013. A Domain-Specific Compiler for Linear Algebra Operations. In *High Performance Computing for Computational Science – VECPAR 2010 (Lecture Notes in Computer Science)*, O. Marques M. Dayde and K. Nakajima (Eds.), Vol. 7851. Springer, Heidelberg, 346–361.

[21] Diego Fabregat-Traver and Paolo Bientinesi. 2013. Application-tailored Linear Algebra Algorithms: A search-sased Approach. *International Journal of High Performance Computing Applications (IJHPCA)* 27, 4 (Nov. 2013), 425 – 438.

[22] Gene H. Golub and Charles F. Van Loan. 2013. *Matrix Computations*. Vol. 4. Johns Hopkins.

[23] Albert Gräf. 1991. Left-to-Right Tree Pattern Matching. In *International Conference on Rewriting Techniques and Applications*. Springer, 323–334.

[24] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen. tuxfamily.org. (2010).

[25] M. Hejazi, S. M. Azimi-Abarghouyi, B. Makki, M. Nasiri-Kenari, and T. Svensson. 2016. Robust Successive Compute-and-Forward Over Multiuser Multirelay Networks. *IEEE Transactions on Vehicular Technology* 65, 10 (Oct 2016), 8112–8129. https://doi.org/10.1109/TVT.2015.2506981

[26] T.C. Hu and M.T. Shing. 1982. Computation of Matrix Chain Products. Part I. *SIAM J. Comput.* 11, 2 (1982), 362–373.

[27] T.C. Hu and M.T. Shing. 1984. Computation of Matrix Chain Products. Part II. *SIAM J. Comput.* 13, 2 (1984), 228–251.

[28] Klaus Iglberger, Georg Hager, Jan Treibig, and Ulrich Rüde. 2012. Expression Templates Revisited: A Performance Analysis of the Current ET Methodologies. *SIAM Journal on Scientific Computing* 34, 2 (2012), C42–C69.

[29] Manuel Krebber. 2017. Non-linear Associative-Commutative Many-to-One Pattern Matching with Sequence Variables. *CoRR* abs/1705.00907 (2017). http://arxiv.org/abs/1705.00907

[30] Chuck L. Lawson, Richard J. Hanson, David R. Kincaid, and Fred T. Krogh. 1979. Basic Linear Algebra Subprograms for FORTRAN Usage. *ACM Transactions on Mathematical Software (TOMS)* 5, 3 (1979), 308–323.

[31] Heejo Lee, Jong Kim, Sung Je Hong, and Sunggu Lee. 2003. Processor Allocation and Task Scheduling of Matrix Chain Products on Parallel Systems. *Parallel and Distributed Systems, IEEE Transactions on* 14, 4 (2003), 394–407.

[32] Noël M Nachtigal, Satish C Reddy, and Lloyd N Trefethen. 1992. How Fast are Nonsymmetric Matrix Iterations? *SIAM J. Matrix Anal. Appl.* 13, 3 (1992), 778–795.

[33] Nadia Nedjah, Colin Walter, and Stephen Eldridge. 1997. Optimal Left-to-Right Pattern-Matching Automata. In *Algebraic and Logic Programming*. Springer, 273–286.

[34] Elias D. Niño, Adrian Sandu, and Xinwei Deng. 2016. A Parallel Implementation of the Ensemble Kalman Filter Based on Modified Cholesky Decomposition. *CoRR* abs/1606.00807 (2016). http://arxiv.

org/abs/1606.00807

[35] Kazufumi Nishida, Yasuaki Ito, and Koji Nakano. 2011. Accelerating the Dynamic Programming for the Matrix Chain Product on the GPU. In *Networking and Computing (ICNC), 2011 Second International Conference on*. IEEE, 320–326.

[36] Silvia Noschese and Lothar Reichel. 2016. Some Matrix Nearness Problems Suggested by Tikhonov Regularization. *Linear Algebra Appl.* 502 (2016), 366–386.

[37] Devangi N Parikh, Maggie E Myers, and Robert A van de Geijn. 2017. Deriving Correct High-Performance Algorithms. *arXiv preprint arXiv:1710.04286* (2017). arXiv:1710.04286

[38] Elmar Peise and Paolo Bientinesi. 2012. Performance Modeling for Dense Linear Algebra. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis (PMBS12) (SCC '12)*. IEEE Computer Society, Washington, DC, USA, 406–416.

[39] Elmar Peise and Paolo Bientinesi. 2015. A Study on the Influence of Caching: Sequences of Dense Linear Algebra Kernels. In *High Performance Computing for Computational Science – VECPAR 2014 (Lecture Notes in Computer Science)*, Michel Daydé, Osni Marques, and Kengo Nakajima (Eds.), Vol. 8969. Springer International Publishing, 245–258. arXiv:1402.5897v1

[40] Elmar Peise and Paolo Bientinesi. 2015. The ELAPS Framework: Experimental Linear Algebra Performance Studies. *CoRR* abs/1504.08035 (2015). http://arxiv.org/abs/1504.08035

[41] Prakash Ramanan. 1996. An Efficient Parallel Algorithm for the Matrix-Chain-Product Problem. *SIAM J. Comput.* 25, 4 (1996), 874–893.

[42] V. Rao, A. Sandu, M. Ng, and E. Nino-Ruiz. 2015. Robust Data Assimilation Using L_1 and Huber Norms. *ArXiv e-prints* (Nov. 2015). arXiv:math.NA/1511.01593

[43] Vishwas Rao, Adrian Sandu, Michael Ng, and Elias D. Nino-Ruiz. 2017. Robust Data Assimilation Using L_1 and Huber Norms. *SIAM Journal on Scientific Computing* 39, 3 (2017), B548–B570. https://doi.org/10.1137/15M1045910

[44] Henrik Ronellenfitsch, Marc Timme, and Dirk Witthaut. 2015. A Dual Method for Computing Power Transfer Distribution Factors. *CoRR* abs/1510.04645 (2015). http://arxiv.org/abs/1510.04645

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

[46] Jeremy G. Siek, Ian Karlin, and Elizabeth R. Jessup. 2008. Build to Order Linear Algebra Kernels. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 1–8.

[47] Daniele G. Spampinato and Markus Püschel. 2014. A Basic Linear Algebra Compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 23.

[48] Daniele G Spampinato and Markus Püschel. 2016. A Basic Linear Algebra Compiler for Structured Matrices. In *International Symposium on Code Generation and Optimization (CGO)*. 117–127.

[49] Damian Straszak and Nisheeth K Vishnoi. 2015. On a Natural Dynamics for Linear Programming. (2015). arXiv:1511.07020

[50] Steve Strate, Roger L. Wainwright, et al. 1990. Parallelization of the Dynamic Programming Algorithm for the Matrix Chain Product on a Hypercube. In *Applied Computing, 1990., Proceedings of the 1990 Symposium on*. IEEE, 78–84.

[51] Todd L. Veldhuizen. 1998. Arrays in Blitz++. In *International Symposium on Computing in Object-Oriented Parallel Environments*. Springer, 223–230.