Linnea:
A Compiler for Mapping Linear Algebra Problems onto High-Performance Kernel Libraries

Der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University vorgelegte Dissertation zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften

von

Henrik Barthels, Master of Science

aus

Viersen-Dülken, Deutschland.

Für Kerstin.

ABSTRACT

The translation of linear algebra computations to efficient code consisting of kernels as provided by libraries such as BLAS and LAPACK is a frequently encountered problem in different areas of science and engineering. Since the manual translation is a tedious, error-prone and time consuming process that requires a lot of expertise, several high-level languages and libraries have been developed that solve this problem automatically. Example include Matlab, Julia, Eigen, and Armadillo. These languages and libraries allow users to describe linear algebra computations in code which closely resembles the mathematical description of the problem; this high-level code is then internally translated to sequences of kernel calls. Unfortunately, while those languages and libraries increase the productivity of the user, it has been shown that this automatic translation frequently results in suboptimal code.

In this thesis, we present Linnea, a domain-specific compiler that automatically translates the high-level description of linear algebra problems to efficient sequences of kernel calls. Linnea aims to combine the ease-of-use of high-level languages with the performance of low-level code written by an expert. Unlike other languages and libraries, Linnea extensively makes use of knowledge about linear algebra: Algebraic identities such as associativity, commutativity, and distributivity are used to rewrite the input problem. In addition, Linnea is able to infer matrix properties and detect common subexpressions. To explore the large space of candidate solutions, Linnea uses a custom best-first search algorithm that quickly finds a first solution, and increasingly better solutions when given more time.

This thesis concludes with an extensive experimental evaluation of Linnea on 25 application and 100 synthetic test problems, both with sequential and parallel execution. The results show that Linnea almost always outperforms Matlab, Julia, Eigen and Armadillo, with speedups up to and exceeding $10\times$. For all test problems, Linnea finds a first solution in less than one second; finding the optimal solution rarely takes longer than a few minutes.

ZUSAMMENFASSUNG

Die Übersetzung von Berechnungen der linearen Algebra in effizienten Code, der aus Funktionen (sogenannte *Kernel*) besteht, wie sie von Bibliotheken wie BLAS und LAPACK bereitgestellt werden, ist ein häufig auftretendes Problem in verschiedenen Bereichen der Natur- und Ingenieurwissenschaften. Da die manuelle Übersetzung ein mühsamer, fehleranfälliger und zeitaufwendiger Prozess ist, der viel Fachwissen erfordert, wurden mehrere höhere Programmiersprachen und Bibliotheken entwickelt, die dieses Problem automatisch lösen. Beispiele hierfür sind Matlab, Julia, Eigen und Armadillo. Diese Sprachen und Bibliotheken ermöglichen es dem Benutzer, Berechnungen der linearen Algebra in Code zu beschreiben, der der mathematischen Beschreibung des Problems sehr ähnlich ist; dieser Code wird dann intern in Sequenzen von Kernel-Aufrufen übersetzt. Diese Sprachen und Bibliotheken führen zwar zu einer höheren Produktivität des Anwenders, es hat sich jedoch gezeigt, dass die automatische Übersetzung häufig in suboptimalem Code resultiert.

In dieser Arbeit stellen wir Linnea vor, einen domänenspezifischen Compiler, der die mathematische Beschreibung von Problemen der linearen Algebra automatisch in effiziente Sequenzen von Kernel-Aufrufen übersetzt. Ziel von Linnea ist es, die Benutzerfreundlichkeit von höheren Programmiersprachen mit der Geschwindigkeit von Code zu kombinieren, der von einem Experten in einer niedrigeren Programmiersprache geschrieben wurde. Im Gegensatz zu anderen Sprachen und Bibliotheken macht Linnea ausgiebig Gebrauch von Wissen über lineare Algebra: Algebraische Identitäten wie Assoziativität, Kommutativität und Distributivität werden verwendet, um das Problem umzuschreiben. Darüber hinaus ist Linnea in der Lage, Eigenschaften von Matrizen abzuleiten und redundante Teilausdrücke zu erkennen. Um die große Anzahl von möglichen Lösungen zu durchsuchen verwendet Linnea einen modifizierten Best-First-Suchalgorithmus, der schnell eine erste Lösung und mit etwas mehr Zeit zunehmend bessere Lösungen findet.

Diese Arbeit schließt mit einer umfangreichen experimentellen Evaluation von Linnea anhand von 25 Anwendungs- und 100 synthetischen Problemen, sowohl mit sequentieller als auch paralleler Ausführung. Die Ergebnisse zeigen, dass der generierte Code fast immer schneller ist als Matlab, Julia, Eigen und Armadillo, zum Teil um einen Faktor von mehr als 10. Für alle Testprobleme findet Linnea eine erste Lösung in weniger als einer Sekunde; die optimale Lösung zu finden dauert selten länger als ein paar Minuten.

PUBLICATIONS

This thesis includes material that has been presented in the following publications:

Henrik Barthels, Christos Psarras, and Paolo Bientinesi. "Linnea: Automatic Generation of Efficient Linear Algebra Programs." In: *ACM Trans. Math. Softw.* 47.3 (June 2021). ISSN: 0098-3500. DOI: 10.1145/3446632.

Henrik Barthels, Christos Psarras, and Paolo Bientinesi. "Automatic Generation of Efficient Linear Algebra Programs." In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. PASC '20. Geneva, Switzerland: Association for Computing Machinery, 2020. ISBN: 9781450379939. DOI: 10.1145/3394277.3401836.

Henrik Barthels and Paolo Bientinesi. *Code Generation in Linnea*. Extended abstract. 6th International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY). Phoenix, Arizona, June 2019.

Henrik Barthels, Marcin Copik, and Paolo Bientinesi. "The Generalized Matrix Chain Algorithm." In: *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization*. Vienna, Austria, Feb. 2018, pp. 138–148. DOI: 10.1145/3168804.

Henrik Barthels. *Linnea: A Compiler for Linear Algebra Operations*. ACM Student Research Competition Grand Finals Candidates, 2016 - 2017. May 2017.

Henrik Barthels and Paolo Bientinesi. "Linnea: Compiling Linear Algebra Expressions to High-Performance Code." In: *Proceedings of the 8th International Workshop on Parallel Symbolic Computation*. New York, NY, USA: ACM, July 2017, 1:1–1:3. ISBN: 978-1-4503-5288-8. DOI: 10.1145/3115936.3115937.

Henrik Barthels. "A Compiler for Linear Algebra Operations." In: *SPLASH '16 Companion*. Student Research Competition. Amsterdam, Netherlands: ACM, Oct. 2016. DOI: 10.1145/2984043.2998539.

Henrik Barthels and Paolo Bientinesi. *The Matrix Chain Algorithm to Compile Linear Algebra Expressions*. Two-page abstract. 4th Workshop on Domain Specific Language Design and Implementation (DSLDI). Amsterdam, Netherlands, Oct. 2016. arXiv: 1611.05660.

While working on this thesis, I co-authored the following publications:

Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. "Fireiron: A Data-Movement-Aware Scheduling Language for GPUs." In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. PACT '20. Virtual Event, GA, USA: Association for Computing Machinery, 2020, pp. 71–82. ISBN: 9781450380751. DOI: 10.1145/3410463.3414632.

Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. "Swizzle Inventor: Data Movement Synthesis for GPU Kernels." In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: ACM, 2019, pp. 65–78. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304059.

Christos Psarras, Henrik Barthels, and Paolo Bientinesi. "The Linear Algebra Mapping Problem." In: *CoRR* abs/1911.09421 (2019). arXiv: 1911.09421 [cs.MS].

Manuel Krebber and Henrik Barthels. "MatchPy: Pattern Matching in Python." In: *Journal of Open Source Software* 3.26 (June 2018), p. 2. DOI: 10.21105/joss.00670.

Manuel Krebber, Henrik Barthels, and Paolo Bientinesi. "Efficient Pattern Matching in Python." In: *Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing*. In conjunction with SC17: The International Conference for High Performance Computing, Networking, Storage and Analysis. Denver, Colorado, Nov. 2017. DOI: 10.1145/3149869.3149871.

Manuel Krebber, Henrik Barthels, and Paolo Bientinesi. "MatchPy: A Pattern Matching Library." In: *Proceedings of the 15th Python in Science Conference*. Austin, Texas, July 2017. arXiv: 1710.06915.

ACKNOWLEDGMENTS

This thesis would have not come into existence without the support from many people; too numerous to mention all of them here. First and foremost, I have to thank my supervisor Prof. Paolo Bientinesi for his great support and advice, and for giving me the freedom to pursue my research according to my own ideas. In addition, I would like to thank Prof. Uwe Naumann and Prof. Markus Püschel for agreeing to review this thesis.

Thanks to all of my colleagues in the High-Performance and Automatic computing group; Sebastian Achilles, Diego Fabregat-Traver, Caterina Fenu, Markus Höhnerbach, Francisco López, William McDoniel, Elmar Peise, Christos Psarras, Aravind Sankaran, Paul Springer, Jan Winkelmann, and Mickaël Zehren, for extensive discussions about work, life, and coffee. Special thanks to Francisco López and Christos Psarras for proofreading this thesis. In addition, thanks to everyone at AICES, both staff and students.

I would like to extend my thanks to NVIDIA for giving me the opportunity to do an internship with them. Thanks to everyone I worked with during this exciting time, especially Prof. Rastislav Bodik and Vinod Grover.

Finally, I wish to express my gratitude to my family and friends for always supporting me, and for being there when I needed them.

CONTENTS

1 INTRODUCTION

In the domain of numerical linear algebra, a significant effort is invested into optimized implementations of a relatively small number of common matrix operations. Over time, this effort has lead to a number of libraries, such as BLAS and LAPACK, that offer highly-tuned code for those operations. The routines provided by those libraries are used as building blocks in countless scientific codes, as well as in languages and libraries that support linear algebra.

Unfortunately, the near-optimal performance of those computational building blocks does not necessarily carry over to the high-level application problems that domain experts solve in their day-to-day work. The reason is that making optimal use of those building blocks requires expertise in both high-performance computing and numerical linear algebra that domain experts rarely have. As a result, high-level languages and libraries such as Matlab, Julia, Eigen, and Armadillo become increasingly popular. They allow users to describe a linear algebra problem with code which closely resembles the mathematical description of the problem; this high-level code is then internally translated to routines as provided by BLAS and LAPACK. Those languages and libraries have the advantage that they free the application experts from the tedious, error-prone and time consuming process of using such libraries directly by writing their code in C or Fortran. Unfortunately, while the productivity of the user is increased, it has been shown that this automatic translation frequently results in suboptimal code [108].

1.1 PROBLEM DEFINITION

The goal of this thesis is to combine the advantages of existing approaches in the domain of linear algebra: The simplicity, and thus, productivity, of a high-level language, paired with performance that comes close to what a human expert can achieve by manually writing low-level code. Specifically, given a high-level, mathematical description of a linear algebra problem and a set of kernels as provided by libraries such as BLAS and LAPACK, the goal is to automatically find a sequence of kernel calls that efficiently computes the input problem. This problem is a case of the Linear Algebra Mapping Problem (LAMP) as introduced in [108], which is NP-complete.

The following examples illustrate some of the challenges that arise in the mapping from high-level expressions to low-level kernels. A straightforward translation of the assignment $y_k := H^\dagger y + (I_n -$

$H^\dagger H)x_k$, which appears in an image restoration application [126], will result in code containing one $\mathcal{O}(n^3)$ matrix-matrix multiplication to compute $H^\dagger H$. In contrast, by means of distributivity, this assignment can be rewritten as $y_k := H^\dagger (y - Hx_k) + x_k$, and computed with only $\mathcal{O}(n^2)$ matrix-vector multiplications. The computation of the expression

$$B_k := \frac{k}{k-1} B_{k-1} \Big(I_n - A^T W_k \left((k-1) I_l + W_k^T A B_{k-1} A^T W_k \right)^{-1} W_k^T A B_{k-1} \Big),$$

which is part of a stochastic method for the solution of least squares problems [23], can be sped up by identifying that the subexpression $W_k^T A$ or its transpose $(A^T W_k)^T$ appear four times.

Often times, application experts possess domain knowledge that leads to imporved implementations: In $x := (A^T A + \alpha^2 I)^{-1} A^T b$ [50], since $\alpha > 0$, it can be inferred that $A^T A + \alpha^2 I$ is symmetric positive definite (SPD); consequently, the linear system can always be solved by using the Cholesky factorization, which is less costly than LU or LDL. Most languages and libraries either do not offer the means to specify such additional knowledge, or do not automatically infer or exploit it.

To give an idea of the difference that an efficient sequence of kernels can make in practice, we use the expression

$$B_1 := \frac{1}{\lambda_1} \left(I_n - A^T W_1 \left(\lambda_1 I_l + W_1^T A A^T W_1 \right)^{-1} W_1^T A \right)$$

as an example, which is also part of a stochastic method for the solution of least squares problems [23]. All matrices have full rank, with $W_1 \in \mathbb{R}^{m \times l}$, $A \in \mathbb{R}^{m \times n}$, $l = 625$, $n = 1000$, $m = 5000$, and $\lambda_1 > 0$. I_n and I_l are respectively identity matrices of size n and l. Fig. 1.1 shows two different Julia implementations that compute this expression; a basic one that is the result of an almost direct translation of the expression to code (Fig. 1.1a), and an optimized one that makes explicit use of BLAS and LAPACK kernels (Fig. 1.1b). With one thread, the optimized implementation is 5.6× faster than the basic one; with 24 threads, it is 4.6× faster.[1] In this thesis, we aim to achieve the high performance of the optimized implementation with input as simple as the basic one.

1.2 TERMINOLOGY AND NOTATION

In this section, we give a brief overview of the most important terminology and notation used throughout this thesis.

1 For this experiment, the same setup as described at the beginning of Ch. 10 was used; it was performed on the Haswell processor.

```
In = Array{Float64}(I, 1000, 1000)
Il = Array{Float64}(I, 625, 625)
B = 1/lambda*(In-transpose(A)*W*((lambda*Il+transpose(W)*A*
    transpose(A)*W)\transpose(W))*A)
```

(a) Basic translation of the expression to code.

```
1  tmp1 = Array{Float64}(undef, 1000, 625)
2  gemm!('T', 'N', 1.0, A, W, 0.0, tmp1)
3  tmp2 = Array{Float64}(I, 625, 625)
4  syrk!('L', 'T', 1.0, tmp1, lambda, tmp2)
5  potrf!('L', tmp2)
6  trsm!('R', 'L', 'T', 'N', 1.0, tmp2, tmp1)
7  tmp3 = -1.0 / lambda
8  B = Array{Float64}(I, 1000, 1000)
9  B ./= lambda
10 syrk!('L', 'N', tmp3, tmp1, 1.0, B)
11 for i = 1:1000-1;
12    view(B, i, i+1:1000)[:] = view(B, i+1:1000, i)
13 end;
```

(b) Optimized implementation that uses BLAS and LAPACK kernels. Since the SYRK kernel only computes the lower triangular half of the symmetric matrix B, in lines 11–13 the full matrix is reconstructed.

Figure 1.1: Two Julia implementations that compute the expression $B_1 := \frac{1}{\lambda_1}(I_n - A^T W_1(\lambda_1 I_l + W_1^T A A^T W_1)^{-1} W_1^T A$.

EXPRESSION Symbolic expressions are tree-like data structures that represent mathematical expressions such as $A^{-1}B + C$. As a convention, for matrices we use uppercase letters such as A, B, or C. Vectors are denoted with lowercase letters, for example v, x, or y. For scalars, we use lowercase greek letters, for instance α, β, or γ. A special case of expressions are *sequences of assignments* such as

$$x_f := WA^T \left(AWA^T\right)^{-1} (b - Ax)$$
$$x_o := W \left(A^T \left(AWA^T\right)^{-1} Ax - c\right).$$

In this thesis, those sequences of assignments describe *what* needs to be computed, but not *how* something is computed. Expressions are formally defined in Sec. 3.2.

PROPERTY The operands that appear in expressions frequently have *properties*. Examples of matrix properties are lower triangular, diagonal, and SPD. All properties used in this thesis are shown in Tab. 6.1.

KERNEL A *kernel* is an optimized routine that computes a specific linear algebra operation, such as $A^TB + C$. In this thesis, we only consider kernels provided by BLAS and LAPACK. Many kernels do not just compute a single operation, but an entire family of operations. For instance, the TRSM kernel solves linear systems $\alpha\,\mathrm{op}(A^{-1})B$ and $\alpha B\,\mathrm{op}(A^{-1})$, where op is either the identity function $\mathrm{op}(A) = A$ or the transposition $\mathrm{op}(A) = A^T$, and A is either upper or lower triangular. Which operation is computed by a given call to TRSM is determined by a number of arguments. Throughout this thesis, we are frequently only interested in one specific operation that can be computed with a kernel. In case of TRSM such an operation could be for instance $A^{-T}B$, where A is lower triangular and $\alpha = 1$. When we refer to such an operation, it is assumed that all arguments are set accordingly. Unless otherwise noted, kernels also include matrix factorizations, as for example the Cholesky factorization, or the symmetric eigenvalue decomposition.

SEQUENCE OF KERNELS While a sequence of assignments describes what needs to be computed, a *sequence of kernels* describes *how* something is computed. In order to emphasize the distinction between those two concepts, we use $\leftarrow$ for the assignment operator in sequences of kernels. As an example, the assignment

$X := S^{-1}A$, where S is SPD, can be computed with the sequence of kernels

$$\begin{aligned} L &\leftarrow \text{chol}(S) && \text{POTRF} \\ M &\leftarrow L^{-1}A && \text{TRSM} \\ X &\leftarrow L^{-T}M && \text{TRSM} \end{aligned}$$

Unless they are relevant for the discussion, we usually do not annotate sequences of kernels with the names of the kernels. It should be noted that a sequence of kernels only considers the mathematical operations computed by the kernels; it does not describe any implementation details such as where and how the operands are stored in memory.

SOLUTION In this thesis, a *solution* is a sequence of kernels that computes the input expression.

COST FUNCTION The *cost function* is used to quantify the quality of a solution. The solution that minimizes this cost function is the *optimal solution*. The cost function used in this thesis counts the number of floating-point operations (FLOPs) required by a sequence of kernels; it is discussed in more detail in Sec. 4.5.

As a convention both for sequences of assignments and sequences of kernels, it is assumed that the value of all operands that do not appear on the left-hand side of an assignment (either $:=$ or $\leftarrow$) are known input operands. As an example, in the expression

$$\begin{aligned} X &:= ABC \\ Y &:= X + D, \end{aligned}$$

A, B, C, and D are input operands, while X and Y are unknown output operands. However, once the value of an output operand is determined by an assignment, it can be used on the right-hand side in subsequent assignments/kernel calls.

1.3 LINNEA: AN OVERVIEW

In this thesis, we present Linnea, a compiler that makes use of domain knowledge to translate the mathematical description of linear algebra problems to efficient sequences of kernel calls.[2] Linnea is written in Python and targets mid-to-large scale linear algebra expressions, where problems are typically compute-bound. It currently supports real-valued computations, and parallelism via multi-threaded kernels. As input, Linnea accepts a sequence of assignments, where the left-hand side is a single operand, and the right-hand side is an expression

A problem is compute-bound *if the speed at which the problem can be solved is limited by the speed of the processor. In contrast, a problem is* memory-bound *if the speed at which the problem can be solved is limited by the memory bandwidth.*

2 Linnea is available at https://github.com/HPAC/linnea. An online demo of Linnea can be found at http://linnea.cs.umu.se/.

```
m = 1000
n = 2000

ColumnVector b(m) <>
ColumnVector c(n) <>
ColumnVector x(n) <>
Matrix A(m, n) <FullRank>
Matrix W(n, n) <SPD, Diagonal>

x = W*(trans(A)*inv(A*W*trans(A))*b-c)
```

Figure 1.2: An example of the input to Linnea for the expression $x := W(A^T(AWA^T)^{-1}b - c)$.

built from addition, multiplication, subtraction, transposition, and inversion. The input language is described in more detail in Sec. 3.1. As building blocks, Linnea uses BLAS and LAPACK kernels, as well as a small number of code snippets for simple operations not supported by those libraries. However, other kernel libraries such as the ones described in Sec. 2.1 could be used instead. As output, we decided to generate Julia code because it offers a good tradeoff between simplicity and performance: Low-level wrappers expose the full functionality of BLAS and LAPACK, while additional routines can be implemented easily without compromising performance [12]. The input and output of Linnea for the expression $x := W(A^T(AWA^T)^{-1}b - c)$, which appears in an optimization problem [123], are shown in Figs. 1.2 and 1.3, respectively.[3]

While Linnea was built having in mind users that are not experts in numerical linear algebra or high-performance computing, it is nonetheless useful for experts too: It saves implementation time by quickly exploring the most important optimizations, it makes it possible to easily investigate alternative implementations, and it serves as a starting point for further optimizations. Since Linnea generates code, it is—unlike other languages and libraries—transparent in the sense that users can verify how solutions are computed. To aid this verification, in the generated code, each kernel call is annotated with the mathematical operation that it computes. In addition, Linnea can also generate a description of how the input expression was rewritten to generate a specific algorithm, together with the costs of the individual kernels.

The structure of Linnea is illustrated in Fig. 1.4. Similar to other compilers, Linnea consists of three main components:

FRONT-END The front-end translates the description of the input expression written in the Linnea language (Sec. 3.1) into a symbolic

3 The code shown in Fig. 1.1b was also generated by Linnea.

```
1  function algorithm0(ml0::Array{Float64,2}, ml1::Array{Float64,1},
       ml2::Array{Float64,2}, ml3::Array{Float64,1})
2      # cost: 4.34e+09 FLOPs
3      ml4 = diag(ml0)
4      ml5 = Array{Float64}(undef, 1000, 2000)
5      blascopy!(1000*2000, ml2, 1, ml5, 1)
6      # tmp2 = (A W)
7      for i = 1:size(ml2, 2);
8         view(ml2, :, i)[:] .*= ml4[i];
9      end;
10
11     ml6 = Array{Float64}(undef, 1000, 1000)
12     # tmp3 = (tmp2 A^T)
13     gemm!('N', 'T', 1.0, ml2, ml5, 0.0, ml6)
14
15     # (L4 L4^T) = tmp3
16     potrf!('L', ml6)
17
18     # tmp10 = (L4^-1 b)
19     trsv!('L', 'N', 'N', ml6, ml3)
20
21     # tmp65 = (W c)
22     ml1 .*= ml4
23
24     # tmp12 = (L4^-T tmp10)
25     trsv!('L', 'T', 'N', ml6, ml3)
26
27     # tmp15 = ((-1.0 tmp65) + (tmp2^T tmp12))
28     gemv!('T', 1.0, ml2, ml3, -1.0, ml1)
29
30     # x = tmp15
31     return (ml1)
32 end
```

Figure 1.3: The generated code for $x := W(A^T(AWA^T)^{-1}b - c)$. The documentation of the function as well as some additional comments were removed in the interest of space. Lines 7–9 is one of the code snippets for operations not supported by BLAS and LAPACK; the multiplication of a full and a diagonal matrix.

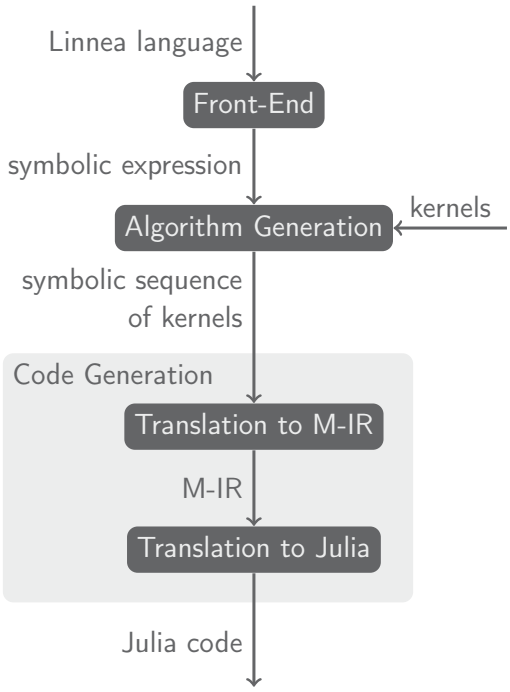


Figure 1.4: The structure of Linnea.

expression (Sec. 3.2). In addition, the correctness of the input is checked.

ALGORITHM GENERATION The algorithm generation (Ch. 4) is the core of Linnea. This component takes as input the symbolic input expression, as well as a set of kernels, and translates this input expression to a sequence of kernels (Ch. 5). In order to optimize the sequence, during this translation Linnea uses knowledge about linear algebra to rewrite the expression (Ch. 7) and infer matrix properties (Ch. 6). In addition, common subexpressions are eliminated (Ch. 8).

CODE GENERATION The code generation (Ch. 9) consists of two steps: In the first step, the symbolic sequence of kernels is translated to an augmented representation called Memory-IR (M-IR) that also considers the location and storage formats of operands in memory. In the second step, the M-IR is then directly translated to the actual Julia code.

1.4 CONTRIBUTIONS

The main contribution of this thesis is Linnea. Beyond the implementation itself, the contributions include the techniques and algorithms that are used in Linnea to achieve a low algorithm generation time and

to solve different subproblems that appear as part of LAMP. Those contributions are:

1. A customized anytime best-first search algorithm which quickly finds a first solution, and increasingly better solutions when given more time (Ch. 4). This search algorithm makes it possible to combine algorithms that constructively find solutions for specific subproblems with an exhaustive exploration of the search space. To achieve a low generation time, the algorithm makes use of the algebraic nature of the domain to significantly reduce the size of the search graph without reducing the size of the search space.

2. Two so called *constructive algorithms* that translate certain types of subexpressions to efficient sequences of kernels in polynomial time (Sec. 5.6): The Generalized Matrix Chain Algorithm [7], and an algorithm for matrix sums.

3. A method for the application of factorizations to solve linear systems and invert matrices that ensures termination (Secs. 4.4 and 5.7). In contrast to kernels that solve linear systems, the direct use of factorizations enables optimizations that are not possible otherwise.

4. Techniques to make use of knowledge about linear algebra. This includes an algorithm for the inference of matrix properties (Ch. 6), and a method for the systematic exploration of different representations of expressions (Ch. 7).

5. An algorithm for the detection and elimination of common subexpressions in symbolic expressions (Ch. 8).

6. An algorithm for the translation of symbolic sequences of kernels to code that is able to take advantage of specialized storage formats for matrices and the ability of kernels to overwrite their input operands (Ch. 9).

Since all steps of the generation are performed symbolically, using rewrite rules and term replacement, the generated algorithms are correct by construction.

The methodology for the automatic generation of algorithms that is implemented in Linnea can be applied to every domain that forms an algebra (see Sec. 4.6). Other possible domains include tensor algebra, the simplification and efficient implementation of automatically generated index expressions, for example as they appear in Lift [107, Sec. 5.3], Optimal Jacobian Accumulation [98], query optimization for relational databases, and graph algorithms when formulated as linear algebra problems [27, 79]. In addition, the methodology could also be applied to real or complex scalars, boolean algebra, quaternions or polynomials.

EXPERIMENTAL EVALUATION This thesis includes an extensive experimental evaluation of Linnea, including:

1. An evaluation of the generated code on application and synthetic test problems, both sequential and in parallel (Sec. 10.2). The generated code is compared to Matlab, Julia, Eigen, and Armadillo.
2. An evaluation of the algorithm generation time of Linnea, including the effectiveness of using the algebraic nature of the domain to reduce the size of the search graph and speed up the generation time (Sec. 10.3).
3. An evaluation of the accuracy of FLOPs as a cost function (Sec. 10.4).
4. An evaluation of the influence of the hardware in the experiments above (Sec. 10.5).

The experiments indicate that the code generated by Linnea usually outperforms Matlab, Julia, Eigen and Armadillo. At the same time, the code generation time is mostly in the order of a few seconds, that is, significantly faster than human experts.

1.5 ORGANIZATION OF THIS THESIS

The organization of this thesis mostly follows the structure of Linnea as shown in Fig. 1.4. Related work is discussed in Ch. 2. In Ch. 3, we provide a description of the input language of Linnea, a formal definition of expressions, and a definition of pattern matching. The algorithm generation is presented in Ch. 4, including a detailed description of the search algorithm. The subsequent chapters cover the different parts of the algorithm generation; the application of kernels in Ch. 5, the inference of properties in Ch. 6, the rewriting of expressions in Ch. 7, and the elimination of common subexpressions in Ch. 8. The code generation is discussed in Ch. 9. Ch. 10 contains the experimental evaluation, followed by the conclusion and possible directions for future work in Ch. 11. The appendix contains a list of the application problems used in the experiments and as examples throughout this thesis (App. a), a description of the specification language for kernels (App. b), and a formalism for the inference of properties based on the bandwidth of matrices (App. c).

READING GUIDE For readers who are only interested in certain aspects of Linnea, below we provide some guidelines for how to read this thesis.

OVERVIEW For an overview of the main ideas of the algorithm generation, without going into the details of the optimizations that

are applied, we recommend reading Sec. 3.2, Sec. 3.3, and Ch. 4. For a complete, but significantly shorter overview of Linnea, we recommend reading [9].

GENERALITY For an overview of the theoretical foundations of the algorithm generation that are independent of the domain of linear algebra, it is sufficient to read Sec. 3.2, Sec. 3.3, and Sec. 4.6.

OPTIMIZATIONS The chapters that cover the different optimizations used in Linnea, that is the inference of properties (Ch. 6), rewriting expressions (Ch. 7), and common subexpression elimination (Ch. 8), can also be read independently from the rest of this thesis. The only prerequisites are Sec. 3.2 and Sec. 3.3. For the code generation (Ch. 9), it is sufficient to know what a sequence of kernels is (see Sec. 1.2).

USERS For readers who are only interested in using Linnea, we recommend reading Sec. 3.1 and Ch. 10.

LIBRARY DEVELOPERS Developers of kernel libraries who are interested in making their library available to Linnea should read Ch. 9 and App. b.

2 RELATED WORK

Parts of this chapter have been published in [8] and [9].

Linnea relates to three different areas of research: Software for linear algebra, code generation as it is done in compilers, and the automatic generation of programs. In this chapter, we provide an overview over the existing work in those areas.

2.1 LINEAR ALGEBRA

Due to its importance in science and engineering, there is a large number of languages, libraries, and other tools that support linear algebra.

LANGUAGES FOR SCIENTIFIC COMPUTING The support for linear algebra is a core feature of high-level languages for scientific computing such as Matlab, Octave, Julia, and R, but also computer algebra systems such as Mathematica and Maple. In those languages, working code can be written within minutes, with little or no knowledge of numerical linear algebra. However, the resulting code (which is possibly numerically unstable[1]) usually achieves suboptimal performance [108]. One of the reasons is that, with the exception of Julia, which supports matrix properties in its type system, these languages rarely make it possible to express properties. A few Matlab functions exploit properties by inspecting matrix entries, a step which could be avoided with a more general method to annotate operands with properties. Furthermore, if the inverse operator is used, an explicit inversion takes place, even if the faster and numerically more stable solution would be to solve a linear system instead [65, Sec. 13.1]; it is up to the user to rewrite the inverse in terms of operators, such as "/" and "\" [95], which solve linear systems.

LIBRARIES Libraries that make it possible to describe linear algebra problems at a high level of abstraction exist in many different programming languages. For instance, expression template libraries such as Eigen [56], Blaze [72], Armadillo [112], and the Matrix Template Library (MTL) [54] provide a domain-specific language embedded into C++. The main idea of those libraries is to improve performance by eliminating temporary operands. NumPy [62] is an example of a library for linear algebra in Python. Similar to high-level languages,

1 Some systems compute the condition number for certain operations and give a warning if results may be inaccurate.

those libraries are missing many important optimization. As a result, they frequently deliver suboptimal code [108].

KERNELS Internally, most high-level languages and libraries rely on the highly optimized computational kernels for basic linear algebra operations provided by BLAS [33, 34, 87] and LAPACK [4]. In addition, there are several other libraries that implement the same or similar functionality with a compatible interface, such as Intel MKL [73], OpenBLAS [131], GotoBLAS [53], libflame [128], BLIS [129], ReLAPACK [104], and BLASFEO [45]. While those libraries offer very good performance, the translation of a mathematical problem into an efficient sequence of kernel invocations is a lengthy and error-prone process that requires deep understanding of both numerical linear algebra and high-performance computing.

DOMAIN-SPECIFIC APPROACHES Transfor [52] is likely the first translator of linear algebra problems (written in Maple) into BLAS kernels; since the inverse operator was not supported, the system was only applicable to relatively simple expressions. More recently, several other approaches have been developed that address different problems in the area of numerical linear algebra: The Formal Linear Algebra Methods Environment (FLAME) [16, 59] is a methodology for the derivation of algorithmic variants for linear algebra operations such as factorizations and the solution of linear systems; Cl1ck [36, 37] is an automated implementation of the FLAME methodology. The goal of BTO BLAS [99] is to generate C code for memory-bound operations, such as fused matrix-vector operations. DxTer uses domain knowledge to optimize programs represented as dataflow graphs [92, 93]. LGen targets 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 [121]. SLinGen [120] combines Cl1ck and LGen to generate code for more complex small-scale problems, but still requires that the input is described as a sequence of basic operations. The functional programming language NumLin [90] incorporates a type system that enforces the correct usage of BLAS and LAPACK kernels. Pilatus [116] is a polymorphic domain-specific language for linear algebra embedded into Scala.

SPORES [130] is an optimization approach for the computation of linear algebra expressions that appear in machine learning application. The idea behind SPORES is to convert the linear algebra expressions into relational algebra expressions in order to find optimization that might be difficult to detect by rewriting expressions according to linear algebra identities. Linnea and SPORES do not support the same set of operations: While SPORES also supports elementwise multiplication, the summation of matrix elements, and sparse matrices, it does not support linear systems and inversion. However, SPORES

could potentially be useful to augment the rewriting of expressions in Linnea.

The goal of the LAGO framework [35, 115] is the automatic generation of programs that compute an incremental update of linear algebra expressions. While the focus of LAGO is different from that of Linnea, both have in common that they make use of algebraic identities to find efficient programs. In contrast to Linnea, LAGO is not able to make use of matrix factorizations.

While the modification or replacement of unoptimized code is an optimization method that is not specific to the domain of linear algebra, there are some cases in which it has been applied to this domain: Using knowledge provided by the user, the Broadway compiler [60] optimizes C code containing library calls by replacing those calls with other code. In [60] this compiler is used to optimize parallel linear algebra code. With Multi-Level Tactics [20], it is possible to detect and replace a naive implementation of a matrix-matrix product consisting of three loops with a call to the GEMM kernel. In addition, Multi-Level Tactics can be used to apply high-level, domain specific optimizations such as solving the matrix chain problem (see also Sec. 5.6.2). LiLAC [48] replaces low-level implementations of sparse linear algebra code with kernel calls.

In contrast to the linear algebra compiler CLAK [38], which inspired the work presented in this thesis, Linnea includes several advances: An improved search algorithm that can incorporate constructive algorithms, the use of the algebraic nature of the domain to remove redundancy in the search graph, an extended inference of properties, a more systematic method for rewriting expressions, as well as improved algorithms for the application of factorizations, common subexpression elimination, and code generation.

RELATED DOMAINS Several tools and compilers exists for domains closely related to linear algebra, such as tensor algebra or linear transforms: The Tensor Contraction Engine (TCE) [10] solves a generalization of the matrix chain problem for dense tensors. The Tensor Algebra Compiler (TACO) [80] generates code for operations over combinations of dense and sparse tensors; the focus is on the support for a large number of different storage formats. The Tensor Contraction Code Generator (TCCG) [122] generates code for binary, GEMM-like contractions of dense tensors. Spiral [43] and FFTW [44] generate optimized code for operations such as the Fourier transform.

2.2 CODE GENERATION

The translation from the intermediate representation of a program in the form of an expression tree to machine instructions is a problem closely related to that discussed in this thesis. However, existing

approaches using pattern matching and dynamic programming [1, 2], as well as bottom-up rewrite systems (BURS) [105] solely focus on expressions containing basic operations directly supported by machine instructions. The two main objectives of code generation are to minimize the number of instructions and the optimal use of registers. While there are approaches that generate optimal code for arithmetic expressions, considering associativity and commutativity [114], more complex properties of the underlying algebraic domain, for example distributivity, are usually not considered.

Instead of applying optimization passes sequentially, Equality Saturation [125] is a compilation technique that constructs many equivalent programs simultaneously, stored in a single intermediate representation. Domain specific knowledge can be provided in the form of axioms. Since it allows for control flow, Equality Saturation is more general in its scope than Linnea. In general, Equality Saturation could also be applied to the domain of linear algebra. However, it is not clear how matrix factorizations can be incorporated.

2.3 PROGRAM SYNTHESIS

The automatic generation of programs, also called *program synthesis*, covers a wide range of different approaches and applications. In this section, we provide a brief overview.

The goal of superoptimizers is to find the best possible implementation of a given functionality [94]. They are usually used to generate short pieces of straight-line code that consists of machine instructions. Those implementations can be generated exhaustively [94] or by a probabilistic search [113]. In both cases, the input is a suboptimal implementation of the desired functionality. The Denali superoptimizer [76] instead generates code based on a formal specification and relies on a SAT solver. The optimizations applied by Denali are described in terms of axioms. In GreenThumb [106], equivalence classes are used to reduce the size of the search space. This idea is similar to the idea of merging branches in the search graph of Linnea (see Sec. 4.2). The difference is that the use of algebraic expressions together with a normal form allows for a rather simple equivalence test. The idea behind program sketches is to fill placeholders in an incomplete program [118, 119]. Another synthesis approach is to generate programs from pairs of input/output examples, for example for string [57] or table processing [40, 63], or more general tasks [3]. To this end, some approaches use machine learning techniques [89]. Component-based synthesis consists in composing components provided by libraries to a program with the desired functionality. Some approaches rely on SAT or SMT solvers for the program generation. The input may be described by a formal specification [58], an input/output oracle [74] or by examples [40]. Others use specialized data structures to represent

the relations and interactions between components, for example based on petri nets [41], or a so called *signature graph* [91].

Using terminology from the domain of program synthesis, Linnea can be described as a superoptimizer for linear algebra expressions. The input is described in terms of a mathematical specification; the output program is constructed from components. To find an optimal program, Linnea relies on a graph search.

3 PRELIMINARIES

In this chapter, both the input language and the internal representation of linear algebra expressions are described. Since the optimizations applied by Linnea mostly consist in the manipulation of expressions according to mathematical laws, the internal representation of expressions is much closer to the representation of symbolic expressions in a computer algebra system such as Mathematica than the internal representations used in traditional compilers. In addition, we provide an overview of pattern matching, an important tool to manipulate expressions that is primarily used for the application of kernels.

3.1 INPUT LANGUAGE

The input to Linnea is a domain-specific language for the description of linear algebra problems. In this language, a linear algebra problem consists of a sequence of assignments, where the left-hand side is a single operand, and the right-hand side is an expression built from addition, multiplication, subtraction, transposition, and inversion. As operands, matrices, vectors, and scalars can be used. Operands can be annotated with the properties shown in Tab. 6.1. It is possible for operands to have more than one property, as long as they do not contradict one another.[1] For instance, a matrix can be diagonal and SPD, which implies that all diagonal elements are positive. An example of the input to Linnea is shown in Fig. 3.1. The input consists of three parts: The definition of operand sizes, the definition of the operands and their properties, and the assignments for which code shall be generated. The grammar of the input language is shown in Fig. 3.2; additional details of this language are described in the language manual.[2]

In Linnea, operands are considered as unique, mathematical objects. For this reason, the value of an operand cannot change, and it is not possible to make an assignment to an operand more than once. As a result, the input assignments have to be in static single assignment (SSA) form. Which operands are input and which ones are output is determined from the sequence of assignments: All operands that appear on the left-hand side of an assignment are considered output, all other operands are input. For instance, in the expressions

1 In order to detect contradicting properties, for each property there is a set of incompatible properties. If a property is added to an operand, it is checked that this operand does not have any properties that are in this set.

2 The language manual is available at https://github.com/HPAC/linnea/blob/master/documentation/language_manual.pdf.

```
m = 1000
n = 5000

Matrix H(m, n) <FullRank>
Matrix Hd(n, m) <FullRank>
IdentityMatrix I_n(n, n)
ColumnVector y(m) <>
ColumnVector y_k(n) <>
ColumnVector x_k(n) <>

Hd = trans(H)*inv(H*trans(H))
y_k = Hd*y + (I_n-Hd*H)*x_k
```

Figure 3.1: An example of the input to Linnea.

$$
\begin{aligned}
\textit{model} &= \{\textit{size\_decl}\}, \{\textit{op\_decl}\}, \{\textit{assignment}\};\\
\textit{size\_decl} &= \textit{id}, \texttt{"="}, \textit{int};\\
\textit{op\_decl} &= \texttt{"Matrix"}, \textit{id}, \textit{dim\_matrix}, \textit{properties}\\
&\mid \texttt{"RowVector"}, \textit{id}, \textit{dim\_vector}, \textit{properties}\\
&\mid \texttt{"ColumnVector"}, \textit{id}, \textit{dim\_vector}, \textit{properties}\\
&\mid \texttt{"Scalar"}, \textit{id}, \textit{properties}\\
&\mid \texttt{"IdentityMatrix"}, \textit{id}, \textit{dim\_matrix}\\
&\mid \texttt{"ZeroMatrix"}, \textit{id}, \textit{dim\_matrix};\\
\textit{dim\_vector} &= \texttt{"("}, \textit{id}, \texttt{")"};\\
\textit{dim\_matrix} &= \texttt{"("}, \textit{id}, \texttt{","}, \textit{id}, \texttt{")"};\\
\textit{properties} &= \texttt{"<"}, [\textit{property}, \{\texttt{","}, \textit{property}\}], \texttt{">"};\\
\textit{assignment} &= \textit{id}, \texttt{"="}, \textit{expr};\\
\textit{expr} &= \textit{term}, \{(\texttt{"+"} \mid \texttt{"-"}), \textit{term}\};\\
\textit{term} &= \textit{factor}, \{\texttt{"*"}, \textit{factor}\};\\
\textit{factor} &= \texttt{"("}, \textit{expr}, \texttt{")"}\\
&\mid \texttt{"trans("}, \textit{expr}, \texttt{")"}\\
&\mid \texttt{"inv("}, \textit{expr}, \texttt{")"}\\
&\mid \texttt{"-"}, \textit{factor}\\
&\mid \textit{number}\\
&\mid \textit{id};
\end{aligned}
$$

Figure 3.2: Grammar of the Linnea language in extended Backus-Naur form. In the interest of brevity, the rules for the nonterminals *property*, *number*, *id* (identifiers), and *int* (integers) are not included.

```
X = A*B
Y = C+D
```

`A`, `B`, `C`, and `D` are input, `X`, and `Y` are output. Once a value has been assigned to an output operand, this operand can be used on the right-hand side in subsequent assignments:

```
X = A*B
Y = C+X
```

If an output operand is used in a subsequent assignment, its properties are inferred from the right-hand side of its initial assignment. As an example, consider the following assignments:

```
X = trans(A)*A
Y = X*B
```

It is not necessary to specify that `X` is symmetric. Instead, this property is inferred from `trans(A)*A`.

An important difference between Linnea and most other languages and libraries that support linear algebra computations is that in Linnea, there is no distinction between the explicit inversion of a matrix and the solution of a linear system; there is only one inversion operation. Linnea automatically detects if it is possible to compute this operation by solving a linear system, or if it is necessary to explicitly invert a matrix instead. Matrices are explicitly inverted only if this is unavoidable, for example in expressions such as $A^{-1} + B$.

ADDITIONAL ARGUMENTS In addition to the description of the input expression, Linnea takes as input several arguments that influence the algorithm generation. Among other things, those arguments allow to set the floating point precision (either single or double precision), a time limit for the generation, and whether or not to produce additional information about the algorithm generation, such as a visual representation of the search graph, and a description of how the input expression was rewritten to find a specific algorithm.

3.2 REPRESENTATION OF EXPRESSIONS

Internally, the input, the operations that are computed by kernels, and the sequences of kernels are represented as symbolic expressions. For expressions and pattern matching (see Sec. 3.3), we mostly use the same formalism and notation as in [5]. The main difference is that for the sake of consistency, we exclusively use *expression* for what is called *term* in [5].

Symbolic expressions are tree-like algebraic data structures constructed from function symbols and variables. As an example, given a binary function f, a unary function g, a constant a, and two variables

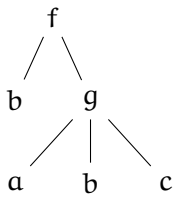


Figure 3.3: Representation of the expression $f(b, g(a, b, c))$ as a tree. f is a binary function, g is a ternary function, while a, b, and c are constants.

x and y, expressions such as $f(x, g(y))$ and $g(a)$ can be constructed. An example of the representation of an expression as a tree is shown in Fig. 3.3. The set of available function symbols is defined as follows:

Definition 3.1 (Signature)**.** A *signature* $\Sigma = \bigcup_{n \geqslant 0} \Sigma^{(n)}$ is a set of function symbols, where $\Sigma^{(n)}$ contains function symbols with arity n. The function symbols in $\Sigma^{(0)}$ are also called *constant symbols*. ■

In this thesis, constant symbols or simply *constants* usually represent matrices, vectors, and scalars. While formally, constants are constant functions, that is, functions without any arguments, they are treated as constants in the usual sense; they are written as a instead of $a()$.[3] In contrast to constants, which represent a fixed value, variables can be thought of as placeholders for arbitrary expressions. More formally, the main difference between constants and variables is that variables can be replaced with other expressions by a substitution (see Sec. 3.3).

In Mathematica, a variable is written as `x_`*, while* `x` *denotes a constant.*

Definition 3.2 (Expressions)**.** Let Σ be a signature and X be a set of variables with $\Sigma \cap X = \varnothing$. The set of all expressions $T(\Sigma, X)$ is defined as the smallest set such that

1. $X \subseteq T(\Sigma, X)$ and
2. for all $n \geqslant 0$, all $f \in \Sigma^{(n)}$ and all $t_1, \ldots, t_n \in T(\Sigma, X)$ we have $f(t_1, \ldots, t_n) \in T(\Sigma, X)$. ■

The set of variables occurring in an expression t is denoted by $\mathrm{Var}(t)$. An expressions that does not contain any variables, that is $\mathrm{Var}(t) = \varnothing$, is called *ground expressions*. The set of all ground expressions is denoted with $T(\Sigma, \varnothing)$.

Example 3.1. Let $f \in \Sigma^{(2)}$ and $g \in \Sigma^{(1)}$ be function symbols, $a, b \in \Sigma^{(0)}$ be constants, and $x, y \in X$ be variables. Then a, $f(a, b)$ and $f(g(a), x)$ are expressions. a and $f(a, b)$ are also ground expressions, as they do not contain variables, while $t = f(g(a), x)$ is not a ground expression because of $\mathrm{Var}(t) = \{x\}$. ■

3 The reason for using constant functions is that they simplify definitions involving expressions.

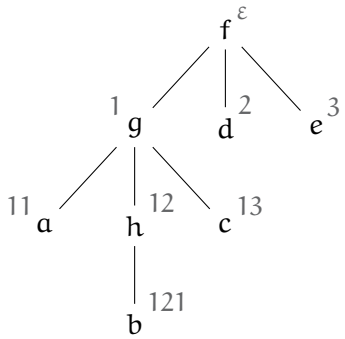


Figure 3.4: Expression tree for $f(g(a, h(b), c), d, e)$. Nodes are labelled with their paths.

In order to refer to and manipulate arbitrary subexpressions in an expression, we define paths on expressions.[4]

Definition 3.3 (Paths)**.** Let $s, t \in T(\Sigma, X)$.

1. A *path* is a sequence of integers $p \in \mathbb{N}^*$. The set of paths of t, $\mathrm{Paths}(t)$, is defined as follows:
   a) If $t \in X \cup \Sigma^{(0)}$, then $\mathrm{Paths}(t) := \{\varepsilon\}$, where ε denotes the empty sequence.
   b) If $t = f(t_1, \ldots, t_n)$, then
   $$\mathrm{Paths}(t) := \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in \mathrm{Paths}(t_i).\}$$
2. Given $p \in \mathrm{Paths}(t)$ with $p = iq$, the *subexpression of* t *at* p, denoted by $t|_p$, is defined as
   $$t|_\varepsilon := t$$
   $$f(t_1, \ldots, t_n)|_{iq} := t_i|_q.$$
3. Given $p \in \mathrm{Paths}(s)$ with $p = iq$, the *replacement of the subexpression at* p *in* s *with* t, denoted by $s[t]_p$, is defined as
   $$s[t]_\varepsilon := t$$
   $$f(s_1, \ldots, s_n)[t]_{iq} := f(s_1, \ldots, s_i[t]_q, \ldots, s_n).$$ ■

Given a set S, S^* is the set of all sequences over S, including the empty sequence ε. For instance, $ab, aaa \in \{a, b\}^*$. The concatenation of two sequences $s_1, s_2 \in S^*$ is written as $s_1 s_2$. For $s \in S^*$, it holds that $s\varepsilon = \varepsilon s = s$.

Example 3.2. Let $t = f(g(a, h(b), c), d, e)$ be the expression shown in Fig. 3.4. The set of all paths is

$$\mathrm{Paths}(t) = \{\varepsilon, 1, 11, 12, 121, 13, 2, 3\}.$$

The subexpressions at 1 and 121 are $t|_1 = g(a, h(b), c)$ and $t|_{121} = b$, respectively. The replacement of the subexpression at 12 with $i(b)$ results in $t[i(b)]_{12} = f(g(a, i(b), c), d, e)$. ■

4 The definition mostly follows that of positions in [5, pp. 36–37]. We use *path* for what is called *position* in [5] because in Ch. 8 we define positions as a generalization of paths.

Throughout this thesis, common functions are written in their usual notation and parenthesis are omitted if they are not necessary. For instance, an expression that represents the addition of two matrices is written as $A + B$, the multiplication of two matrices is written as AB, and the transposition of a matrix is written as A^T.

Function symbols can be associative and/or commutative. A binary function symbol f is associative if $f(x, f(y, z)) = f(f(x, y), z)$ for all x, y and z. Nested associative functions are always flattened to variadic functions, that is, both $f(x, f(y, z))$ and $f(f(x, y), z)$ are flattened to $f(x, y, z)$. A binary function symbol f is commutative if $f(x, y) = f(y, x)$ for all x and y. Arguments in commutative functions are always sorted according to an arbitrary total order defined on all expressions.[5]

IMPLICATIONS OF FLATTENING ASSOCIATIVE FUNCTIONS Flattening associative functions has both advantages and disadvantages. The main advantage is that it reduces the number of alternative representations for algebraically equivalent expressions. This simplifies pattern matching, and in algorithms that manipulate expressions there is no need to cover the case of nested addition or multiplication. The disadvantage is that checking the correctness of the input expressions and determining the number of rows and columns of expressions becomes more difficult. This difficulty is related to the fact that in Linnea there are no dedicated functions for the multiplication of a matrix with a scalar, as well as for inner products. Furthermore, even though the grammar allows for parentheses in associative functions, flattening those associative functions prevents that parentheses are preserved in the symbolic expressions. Thus, for a vector v and a matrix A, the input expression $(v^T v)A$ becomes a single product with three arguments; v^T, v and A. For a matrix product to be valid, it is usually required that the number of columns of an operand is equal to the number of rows of the operand to its left; for the subexpression vA in $v^T vA$, this rule is not applicable. Thus, both to check correctness and determine the size of this expression, it is necessary to identify scalars and inner products in matrix products. How the size of expressions and especially products is computed is discussed in Sec. 6.4.

SEQUENCES OF ASSIGNMENTS Formally, both assignments and sequences of assignments can be described as an expression as defined in Def. 3.2 by introducing a binary assignment operator $:= \in \Sigma^{(2)}$, and an associative binary sequence operator $s \in \Sigma^{(2)}$. As a result, most definitions and algorithms that involve expressions also apply to

5 In Linnea, functions and variables are sorted by their names. Expressions that have the same functions at the root are sorted by the number of arguments. If the number of arguments is the same, functions are sorted according to a lexicographic order on the arguments (see also [82, Def. 2.7]).

sequences of assignments.[6] In order to keep the presentation simple, in those cases we do not distinguish between expressions and sequences of assignments and use the term *expression* for both.

3.3 PATTERN MATCHING

Pattern matching is a convenient and powerful tool to work with and manipulate symbolic expressions. In Linnea, it is primarily used to identify where kernels can be applied, but also to rewrite expressions. In order to define pattern matching, it is necessary to first introduce substitutions. Substitutions are functions that can be applied to expressions and replace variables with other expressions. As an example, given an expression $f(x, y)$ and a substitution that replaces x with a and y with $g(b)$, the application of this substitution to $f(x, y)$ results in $f(a, g(b))$.

Definition 3.4 (Substitution)**.** Let Σ be a signature and X be a set of variables. A *substitution* is a function $\sigma : T(\Sigma, X) \to T(\Sigma, X)$ with

$$\sigma(f(t_1, \ldots, t_n)) := f(\sigma(t_1), \ldots, \sigma(t_n))$$

for all $f \in \Sigma$ and $t_1, \ldots, t_n \in T(\Sigma, X)$. Variables can be mapped to arbitrary expressions in $T(\Sigma, X)$, including themselves. The set of variables that σ does not map to themselves is called the domain of σ; it is defined as $\mathrm{Dom}(\sigma) = \{x \in X \mid \sigma(x) \neq x\}$. The set of all substitutions is denoted with $\mathcal{S}$. ■

Since the variables that are not mapped to themselves are sufficient to fully describe a substitution, we usually write a substitution σ with $\mathrm{Dom}(\sigma) = \{x_1, \ldots, x_n\}$ as $\sigma = \{x_1 \mapsto \sigma(x_1), \ldots, x_n \mapsto \sigma(x_n)\}$.

Example 3.3. Let $s = f(x, b)$ and $t = f(x, f(x, y))$ be expressions, and $\sigma = \{x \mapsto a, y \mapsto g(b)\}$ be a substitution. The application of σ to s results in $\sigma(s) = f(a, b)$, while $\sigma(t) = f(a, f(a, g(b)))$. ■

With substitutions, pattern matching is defined as follows.

Definition 3.5 (Pattern Matching)**.** A pattern $t \in T(\Sigma, X)$ matches a subject $s \in T(\Sigma, \varnothing)$ if and only if there exists a substitution σ such that $\sigma(t) = s$. We also refer to such a substitution as a *match*. ■

Formally, there is no difference between expressions and patterns. We call an expression a pattern *if it is used as a pattern for pattern matching.*

As an example, the pattern $t = f(x, y)$ matches the subject $s = f(a, b)$ with the substitution $\sigma = \{x \mapsto a, y \mapsto b\}$ since $\sigma(t) = s$. Throughout this thesis, associativity and commutativity are taken into account for pattern matching. That is, given a pattern $t \in T(\Sigma, X)$ and a subject $s \in T(\Sigma, \varnothing)$, we say that t matches s with σ if it is possible to rewrite s to s' by making use of associativity and commutativity such that $\sigma(t) = s'$. Pattern matching without associative and commutative functions is called *syntactic* pattern matching. Paths can be used to

6 Sequences of assignments require a separate treatment whenever the dataflow between consecutive assignments is relevant.

describe that a pattern matches a subexpression of an expression: A pattern t matches a subexpression in s if there is a path p and a substitution σ such that $\sigma(t) = s|_p$.

Example 3.4. Let $f \in \Sigma^{(2)}$, $g \in \Sigma^{(1)}$, $a, b, c \in \Sigma^{(0)}$, and $x \in X$.

1. Given the pattern $t_1 = f(a, x)$ and the subject $s_1 = f(a, b)$, the substitution $\sigma_1 = \{x \mapsto b\}$ is a match.
2. Assuming that f is commutative, the subject $s_2 = f(c, a)$ can be rewritten to $s_2' = f(a, c)$ such that t_1 matches s_2 with $\sigma_2 = \{x \mapsto c\}$.
3. Assuming that f is also associative, the subject $s_3 = f(b, a, c)$ can be rewritten to $s_3' = f(a, f(b, c))$ such that t_1 matches s_3 with $\sigma_3 = \{x \mapsto f(b, c)\}$.
4. Given a pattern $t_2 = g(x)$ and a subject $s_4 = f(g(a), b)$, t_2 matches the subexpression $g(a)$ in s_4 at position 1 with $\sigma_4 = \{x \mapsto a\}$ since $\sigma_4(t_2) = s_4|_1$.
5. There can be more than one match for a given pattern. For instance, if f is associative but not commutative, there are two matches for the pattern $f(x, y)$ and the subject $f(a, b, c)$:

   $$\sigma_1 = \{x \mapsto a, y \mapsto f(b, c)\}$$
   $$\sigma_2 = \{x \mapsto f(a, b), y \mapsto c\}$$

Variables in patterns can have constraints, that is, conditions that have to be satisfied in order for the pattern to match a given subject. In Linnea, those constraints usually concern matrix properties, or the type of an expression, that is, whether it is a matrix, vector, or scalar.

Example 3.5. Let $x^{-1}y$ be the pattern for the TRSM kernel. In this case, x has the constraints that it can only match triangular, square matrices that have full rank, and y can only match matrices.

In addition to standard variables that can match exactly one expression, in Linnea we also use *sequence variables* that match a sequence of expressions. There are two different types of sequence variables: Sequence variables x^* that match zero or more expressions, and sequence variables x^+ that match at least one expression.

In Mathematica, a variable x^+ is written as x__, while x^* is written as x___.

Example 3.6. Let $f \in \Sigma^{(2)}$, $g \in \Sigma^{(1)}$, $a, b \in \Sigma^{(0)}$, and $x^+, x^* \in X$. In the following, as a subject we use $f(a, g(b))$.

1. For the pattern $f(x^*)$, the substitution $\sigma_1 = \{x^* \mapsto (a, g(b))\}$ is a match.
2. For the pattern $f(a, g(b), x^*)$, the substitution $\sigma_2 = \{x^* \mapsto ()\}$ is a match, where $()$ is the empty sequence.
3. For the pattern $f(a, g(b), x^+)$, there is no match because x^+ has to match at least one expression.

Pattern matching is frequently used for the application of rewrite rules. A rewrite rule describes how subexpressions of an expression can be replaced with other expressions. As an example, consider the rewrite rule $h(x) \to f(x)$; it describes that any subexpression that matches the pattern $h(x)$ with a substitution σ can be replaced with $\sigma(f(x))$. Rewrite rules are defined as follows:

Definition 3.6 (Rewrite Rule)**.** A *rewrite rule* is a tuple $(l, r) \in T(\Sigma, X) \times T(\Sigma, X)$ with $\mathrm{Var}(l) \supseteq \mathrm{Var}(r)$. Rewrite rules will be written as $l \to r$.

Given a rewrite rule $l \to r$, an expression $s \in T(\Sigma, X)$, a position $p \in \mathrm{Paths}(s)$, and a substitution σ with $s|_p = \sigma(l)$, the application of $l \to r$ at position p in s results in $t = s[\sigma(r)]_p$.

Example 3.7. Let $f \in \Sigma^{(2)}$, $g, h \in \Sigma^{(1)}$, $a, b, c, d \in \Sigma^{(0)}$ and $t = f(a, g(b))$. The rewrite rule $g(b) \to f(c, d)$ can be applied at position 2 in t with $\sigma = \varnothing$, resulting in $f(a, f(c, d))$. The application of the rule $f(x, y) \to h(x)$ at position ε in t with $\sigma = \{x \mapsto a, y \mapsto g(b)\}$ results in $h(a)$.

IMPLEMENTATION For pattern matching as well as for the representation of expressions, Linnea relies on the Python module MatchPy [83, 85]. Specifically, expressions in Linnea are subclasses of MatchPy expressions. In general, pattern matching with associative and/or commutative functions is NP-complete [11]. Despite this theoretical limitation, when many patterns need to be matched against a single subject, it is still possible to significantly speed up matching by making use of the similarities among patterns. The idea of this approach, which is known as *many-to-one matching*, is to combine all patterns into a data structure similar to a decision tree. Given a subject, all possible matches for this subject are found by traversing this tree. MatchPy implements efficient algorithms for many-to-one matching, both for syntactic and associative-commutative matching [84].

4 ALGORITHM GENERATION

Most of the material presented in this chapter has been published in [9]. In addition, some of the material has been published in [8].

The core idea behind Linnea is to rewrite the input problem while successively identifying parts that are computable by a sequence of one or more of the available kernels. In general, for a given input problem and cost function, Linnea generates many different sequences, which all compute the problem, but differ in terms of cost. In order to efficiently store all generated sequences, we use a graph in which nodes represent the input problem at different stages of the computation, and edges are annotated with the kernels used to transition from one stage (node) to another.

This process starts with a single root node containing a symbolic expression that represents the input problem. The generation process consists of two steps, which are repeated until termination. 1) In the first step, the input expression is rewritten in different ways, for example by making use of distributivity. The different representations of a given expression are not stored explicitly; instead, a node only contains one canonical representation, and it is rewritten when necessary. 2) In the second step, on each representation of the expression, different algorithms are used to identify subexpressions that can be computed with one or more of the available kernels. Whenever such an expression is found, a new successor of the parent node is constructed. The edge from the parent to the new child node is annotated with the sequence of kernels, and the child contains the expression that is left to be computed.

The two steps are then repeated on the new nodes, until at least one node with nothing left to compute is found, or until the time limit is exceeded. In practice, this process corresponds to the construction and traversal of a graph. An example of such a graph is shown in Fig. 4.1.

Upon termination, the concatenation of all kernels along a path in the graph from the root to a leaf is a program that computes the input problem. In Sec. 4.4, we discuss how termination is guaranteed. Given a function that assigns a cost to each kernel, the optimal program is found by searching for the shortest path in the graph from the root node to a leaf.

4.1 THE SEARCH ALGORITHM

In Linnea, the construction and traversal of the search graph is done with a best-first search algorithm. The rationale is to find a good, although potentially suboptimal solution as quickly as possible, to then use the cost of that solution to prune branches that cannot lead

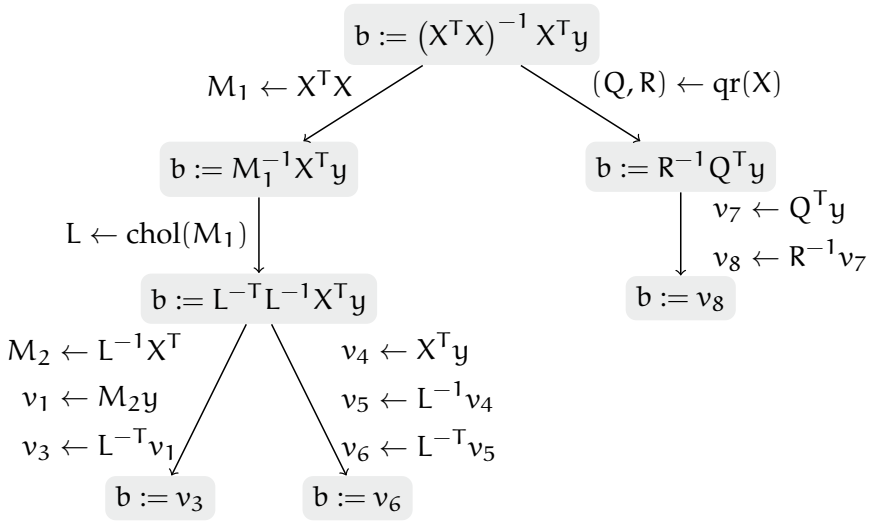


Figure 4.1: An example of a search graph for the input $b := (X^TX)^{-1}X^Ty$. This graph represents only a small part of the search space that Linnea actually explores.

to a better solution. Over time, progressively better solutions are found. To guide the search towards good solutions, we use priorities to indicate which node to explore next. Priorities are non-negative integers, where smaller numbers indicate higher priority. In order to break ties and ensure that a first solution is found quickly, nodes are stored in a priority stack. This stack can be seen as a collection of stacks, one for each priority. In a priority stack, the push operation corresponds to putting an element onto the internal stack of the corresponding priority. The pop operation instead takes an element from the top of the highest priority, non-empty stack.

The property that high priorities (i.e., small numbers) correspond to nodes that are likely to have a promising next successors is due to the following two facts: 1) The priority of a node is equal to the number of successors that have already been generated for this node, and 2) the next_successor function, which returns a new successor of a node every time it is called, is designed to return the most promising successors first (this function is described in more detail in Sec. 4.3). By using the number of current successors of a node as its priority, the algorithm effectively balances the number of successors of all nodes, that is, it does not explore the $n+1$th successor of any node before having explored the first n successors of all nodes. As a result, in contrast to depth-first search which explores the current branch first, the algorithm quickly goes back to the root node. The underlying idea is that the importance of a node does not depend on its position in the graph. While a node deep in the graph is closer to a solution, the decision that has the largest impact on the quality of this solution could have taken place already at the top of the graph. As it is usually not necessary, and frequently also not practical to explore the full

```
1  G := ({v_input}, ∅) = (V, E)                          # graph initialization
2  best_solution := c_∞                                  # cost initialization
3  stack := PriorityStack()                              # stack initialization
4  stack.push(0, v_input)                 # the root node is added to the stack
5  while ¬stack.empty() and elapsed_time < t_max:
6      (p, v) := stack.pop()
7      if cost(v) > best_solution:                       # node is pruned
8          continue                                      # jump to line 5
9      v_new := next_successor(v)                        # successor creation
10     V := V ∪ {v_new}                                  # update graph
11     E := E ∪ {(v, v_new)}
12     if ¬is_terminal(v_new):
13         stack.push(0, v_new)
14     else:
15         if cost(v_new) < best_solution:
16             best_solution := cost(v_new)   # update cost of best solution
17     stack.push(p + 1, v)   # the current node is added back to the stack
```

Figure 4.2: Pseudocode of the search algorithm.

graph, we allow to specify an upper limit for the time spent on this search.

The algorithm is shown in Fig. 4.2. The search graph is initialized with v_{input} as the root node in line 1; the variable best_solution will hold the cost of the current best solution, and is initialized with infinity in line 2; the priority stack initially contains v_{input} with priority 0 (line 4). At every iteration of the while loop, a new successor is generated. To this end, in line 6 the node with the highest priority is taken from the stack. This operation returns both the node v, as well as its priority p. If the cost of v (the cost of the path from the root node to v), is higher than the cost of the current best solution, then node v is pruned (it cannot lead to a better solution), and the rest of the loop body is skipped (lines 7–8). If v is not pruned, then its next successor, v_{new}, is generated in line 9; $\text{cost}(v_{\text{new}})$ is set to the sum of $\text{cost}(v)$ and the cost of the kernel(s) along the edge from v to v_{new}. Although not shown in the code, if v_{new} does not exist because all successors were already explored, the rest of the loop body is skipped too. If v_{new} is a terminal node, that is, there is nothing left to compute, best_solution may have to be updated with $\text{cost}(v_{\text{new}})$ (lines 15–16); if v_{new} is not terminal, in line 13 it is added to the stack with priority 0. Finally, in line 17, the node v is put back on the stack with priority $p + 1$. The loop terminates either when the stack is empty, or when the time limit is reached.

Example 4.1. Fig. 4.3 shows the order in which the first 15 nodes are visited in a ternary tree. The successors of each node are sorted from left to right, the leaves are terminal nodes. At the point shown in

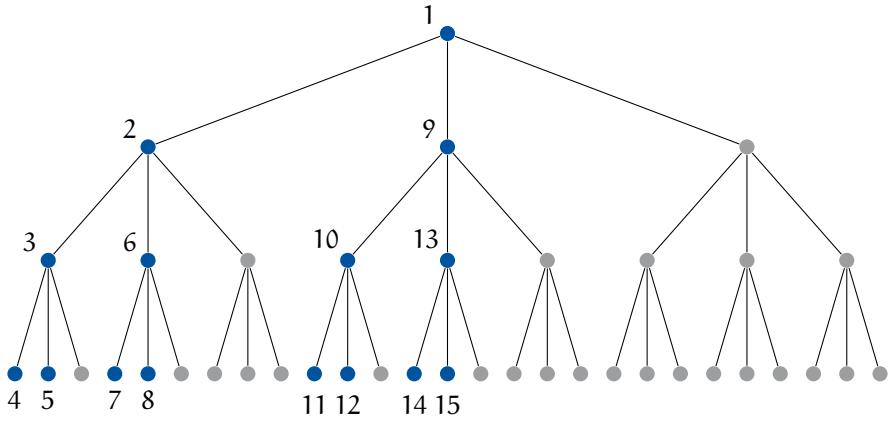


Figure 4.3: Example of the visitation order.

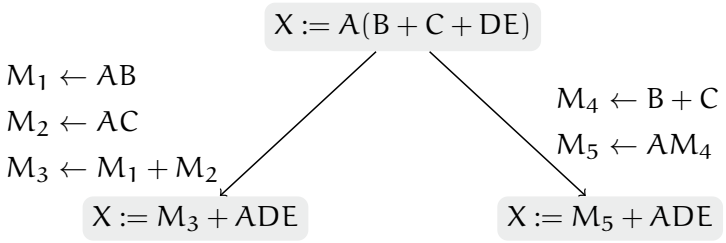


Figure 4.4: Search graph with redundancy.

Fig. 4.3, the algorithm has explored the first and second successor of all non-terminal nodes that have been visited so far. Thus, in the next step, the third successor of the node labelled with 13 will be visited, because it is the last node that was put on the priority stack. Thereafter, the third successor of the node labelled with 9 will be visited. ■

4.2 REDUNDANCY IN THE SEARCH GRAPH

With large input expressions, it frequently happens that there is a lot of redundancy in the search graph. As an example, to compute the subexpression $A(B+C)$ of $A(B+C+DE)$, the two different programs shown in Fig. 4.4 were constructed. As the generation process unfolds, both leaf nodes will be expanded, deriving the same programs for ADE twice. This phenomenon can be alleviated by taking advantage of the algebraic nature of the domain: In Fig. 4.4, it is clear that M_3 and M_5 represent the same quantity because $AB + AC = A(B+C)$.[1] Thus, it is possible to merge the two branches and only do the generation for ADE once.

Our approach for detecting equivalent nodes and for merging branches in the search graph consists of two parts: First, we define a normal form for expressions, that is, a unique representation for algebraically equivalent terms. Then, we make sure that irrespec-

1 Ignoring differences due to floating-point arithmetic.

Table 4.1: The table of intermediate operands after deriving two programs that compute the subexpression $A(B+C)$ in $X := A(B+C+D)$.

intermediate	expression
M_1	AB
M_2	AC
M_3	$AB+AC$
M_4	$B+C$

tive of how a subexpression was computed, its result is represented by the same, unique intermediate operand. In case of the graph in Fig. 4.4, this would mean that the same intermediate is used for $AB+AC = A(B+C)$ in both leaves. When rewritten to their normal form, the equivalence of two expressions can simply be checked by a syntactic comparison. The normal form is discussed in more detail in Sec. 7.2.1.

It should be noted that this normal form is not a true normal form in the sense that there is no guarantee that all algebraically equivalent expressions have the same normal form. This aspect is explained in more detail in the following section, as well as in Sec. 4.6.4 and Sec. 7.3. However, since the normal form is only necessary for merging branches, which is a performance optimization, it is not required that all algebraically equivalent expressions can be identified as equivalent. If two equivalent expressions are not identified as equivalent, this simply has the effect that an opportunity for merging is not identified, so the optimization is less effective.

4.2.1 *Unique Intermediate Operands*

To ensure that the same intermediate operand is used for equivalent expressions, we make use of the normal form of expressions. The idea is to maintain a table of intermediate operands and the expressions they represent in the normal form. Whenever a kernel is used to compute part of an expression, we reconstruct the full expression that is computed by recursively replacing all intermediate operands. The resulting expression is then transformed to its normal form, and it is checked if there already is an intermediate operand for this expression in the table of intermediate operands.

Example 4.2. Let us assume we are given the input $X := A(B+C+D)$. Initially, the table of intermediate operands, which is shown in

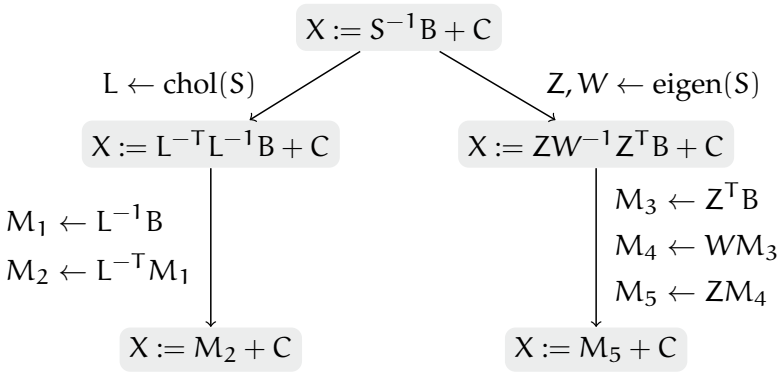


Figure 4.5: Search graph with redundancy due to factorizations.

Tab. 4.1, is empty. The first partial program is found by rewriting this assignment as $X := AB + AC + AD$ and computing

$$M_1 \leftarrow AB$$
$$M_2 \leftarrow AC$$
$$M_3 \leftarrow M_1 + M_2,$$

resulting in $X := M_3 + AD$. For the first two kernels, we simply add the intermediates M_1 and M_2, and the corresponding expressions AB and AC to the table. For $M_1 + M_2$, we first use the table to replace the intermediate operands M_1 and M_2 with the expressions they represent, resulting in $AB + AC$. As this expression is already in normal form, we can simply check if there already is an entry for it in the table. Since at this point, there is no entry for $AB + AC$ yet, $AB + AC$ is added to the table, and a new operand M_3 is created. Alternatively, the same part of $X := A(B + C + D)$ can be computed as

$$M_4 \leftarrow B + C$$
$$M_3 \leftarrow AM_4.$$

For the kernel invocation AM_4, the intermediate operand is created by replacing M_4 by $B + C$, and then converting the resulting expression $A(B + C)$ to normal form, which in this case is $AB + AC$. Then, from the table, M_3 is retrieved. Tab. 4.1 shows the state of the table after deriving those two programs. ■

4.2.1.1 *Factorizations*

To solve linear systems and, if necessary, to explicitly invert matrices, Linnea directly uses factorizations. Unfortunately, the approach for the construction of unique intermediate operands as presented above fails if factorizations are applied in an expression. The problem can be illustrated well with the expression $X := S^{-1}B + C$, where S is SPD. A search graph for this expression is shown in Fig. 4.5. In this graph, both the Cholesky factorization and the symmetric eigenvalue decomposition are applied to S, respectively resulting in $X := L^{-T}L^{-1}B + C$

Table 4.2: The table of intermediate operands after generating two programs that compute the subexpression $S^{-1}B$ in $X := S^{-1}B + C$.

intermediate	expression
M_1	$L^{-1}B$
M_2	$S^{-1}B$
M_3	Z^TB
M_4	WZ^TB

and $X := ZW^{-1}Z^TB + C$ after the conversion to normal form. Even though the subexpressions $L^{-T}L^{-1}B$ and $ZW^{-1}Z^TB$ are mathematically equivalent, their normal forms are different; as a result, with the approach described above, they would not be identified as equivalent. The problem is that with factorizations, it is much more difficult to go back to the original expression: With kernels that produce a single output operand, that operand can simply be replaced with the operation that was computed. Factorizations instead produce an output expression; in this case LL^T and Z^TWZ. While it would be possible to replace those output expressions with the operand S that was factored to go back to the original input expression $S^{-1}B$, this is only possible if the output expression still appears in its original form. Since in the normal form the inverse is pushed down, this is usually not the case.

In order to solve this problem, whenever factorizations are applied, rewrite rules are generated that take the conversion to normal form into account. In case of the example above, when the Cholesky factorization is applied to S, the rewrite rule $L^{-T}L^{-1} \to S^{-1}$ is generated. The left-hand side of this rule is constructed from the output expression LL^T of the Cholesky factorization, which is first inverted and then converted to normal form. Whenever an intermediate operand is constructed for an expression that contains $L^{-T}L^{-1}$, this rewrite rule is applied to obtain the original expression.

Example 4.3. As an example, we use again the expression $X := S^{-1}B + C$, where S is SPD. During the construction of the graph in Fig. 4.5, the rewrite rules $L^{-T}L^{-1} \to S^{-1}$ and $ZW^{-1}Z^T \to S^{-1}$ are constructed. As before, when the intermediate operand M_2 is generated, M_1 is replaced with $L^{-1}B$, resulting in $L^{-T}L^{-1}B$. In this expression, the rewrite rule for the Cholesky factorization of S matches; it is applied to obtain $S^{-1}B$, the equivalent expression of M_2.

When the intermediate operand for the operation ZM_4 is constructed, the intermediate M_4 is replaced to obtain $ZW^{-1}Z^TB$. At this point, the rewrite rule for the symmetric eigenvalue decomposition is used to retrieve the original expression $S^{-1}B$. Since this expression is already in the table of intermediate operands, instead of creating M_5 as shown in Fig. 4.5, the existing M_2 is reused for this operation.

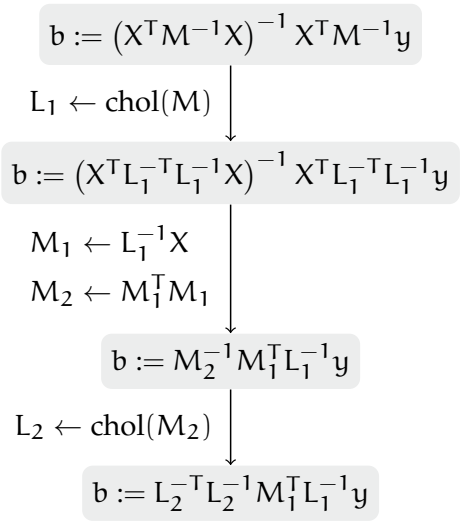


Figure 4.6: Example of a graph where two rounds of applying rewrite rules are necessary to obtain the input expression.

As a result, the two leaves in this graph can be merged. The table of intermediate operands after the construction of the graph is shown in Tab. 4.2. ■

If the operand which is factored is an intermediate operand, in order to retrieve the input expression, it might be necessary to apply multiple rounds of replacing intermediate operands, converting the expression to normal form, and applying rewrite rules. To avoid this, instead of the operand being factored, its equivalent expression is used in the right-hand side of the rule. With such rules, one application of each of the three steps is sufficient.

Example 4.4. An example of a case where rewrite rules have to be applied multiple times is shown in Fig. 4.6. In order to go back from the expression

$$b := L_2^{-T} L_2^{-1} M_1^T L_1^{-1} y$$

to the input

$$b := (X^T M^{-1} X)^{-1} X^T M^{-1} y,$$

it is necessary to

1. apply rule $L_2^{-T} L_2^{-1} \to M_2^{-1}$,
2. replace M_2 with its equivalent expression $X^T L_1^{-T} L_1^{-1} X$,
3. replace M_1 with its equivalent expression $L_1^{-1} X$,
4. convert the resulting expression to normal form,
5. and apply rule $L_1^{-T} L_1^{-1} \to M^{-1}$.[2]

2 Other orders are also possible. The dependencies are 1. → 2. → 5. and 3. → 4. → 5.

This process can be simplified by replacing the intermediate operand M_2 with its equivalent expression $X^T M^{-1} X$ in the right-hand side of the first rewrite rule, resulting in the rule $L_2^{-T} L_2^{-1} \to (X^T M^{-1} X)^{-1}$. With this rule, it is sufficient to

1. replace M_1 with its equivalent expression $L^{-1}X$,
2. convert the resulting expression to normal form,
3. and apply rules $L_2^{-T} L_2^{-1} \to (X^T M^{-1} X)^{-1}$ and $L_1^{-T} L_1^{-1} \to M^{-1}$.

Since factorizations are also applied to matrices that are both inverted and transposed, as well as to matrices that are not inverted directly, but instead appear within an inverted subexpression, for example S in $(A^T S A)^{-1}$ (see Sec. 5.7), up to four rewrite rules are generated when a factorization is applied: Rules for inversion, transposition, the combination of inversion and transposition, as well as a rule that replaces the unmodified output expression. As an example, when the LU factorization is applied to a matrix A, the following rules are generated:

$$
\begin{aligned}
U^{-1} L^{-1} P &\to A^{-1} \\
U^T L^T P &\to A^T \\
P^T L^{-T} U^{-T} &\to A^{-T} \\
P^T L U &\to A
\end{aligned}
$$

Rewrite rules involving the inversion are only generated if the factored operand is square; rules involving the transposition are only generated if the output expression is not symmetric.

LIMITATIONS There are cases where the conversion to normal form modifies the expression in a way such that the generated rewrite rules are not applicable anymore. This is for example the case when factors cancel out: When the QR factorization is applied to the subexpression $S(S^T A S)^{-1} S^T A$ that appears in example problem a.15, this subexpression is simplified to $Q(Q^T A Q)^{-1} Q^T A$. Going back to the original expressions with rules such as $QR \to S$ is not possible. Instead, it would be necessary to detect that $Q(Q^T A Q)^{-1} Q^T$ is equivalent to $S(S^T A S)^{-1} S^T$ and replace the former with the latter. While it is relatively simple for a human expert to identify that those two expressions are equivalent, this problem is difficult to solve algorithmically. The development of an algorithm that solves this problem is planned for the future.

4.2.2 *Merging Branches*

When merging branches, we implicitly assume that nodes do not have any state information such as the state of the registers, caches, or

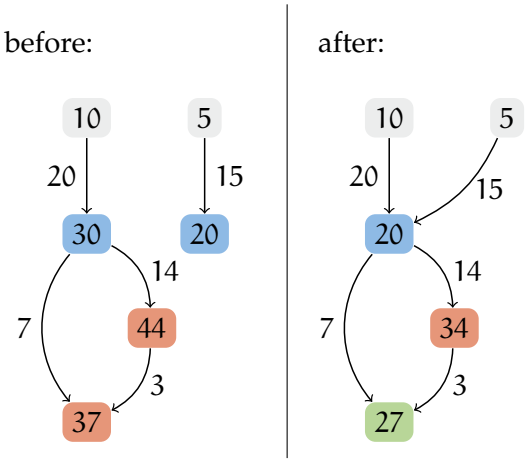


Figure 4.7: An example of how through merging, pruned nodes need to be reactivated. Let us assume that the cost of the current best solution is 32, and that the blue nodes can be merged. Initially, the two red nodes are pruned. When the blue nodes are merged, the cost of one of the pruned nodes decreases from 37 to 27. As a consequence, this node may now lead to a new solution with a cost below 32.

memory. This is a simplification that does not hold true in reality. However, without this assumption, it would not be possible to merge branches in the search graph. This optimization can drastically reduce the size of the search graph without reducing the size of the search space, thus making it possible to generate programs for larger input expressions.

4.2.3 *Updated Search Algorithm*

Merging branches is an optimization that imposes some changes in the search algorithm; when a new node is generated, it has to be checked whether or not it is equivalent to an already existing one. If there is an equivalent node, the new node is merged into the existing one. In addition, through merging, it is possible that the cost of a pruned node decreases, such that the node can again lead to a new, better solution. As a result, pruned nodes need to be reactivated. An example of how merging affects pruned nodes is shown in Fig. 4.7.

The pseudocode of the updated algorithm is shown in Fig. 4.8. Compared to the original algorithm in Fig. 4.2, this one contains the following changes: In line 5, V_{pruned} is initialized with the empty set; during the execution of the algorithm, it will hold pairs containing pruned nodes and their priorities. Those pairs are added to the set in line 9. After a new node is generated, in line 12 it is tested whether an equivalent node with the same expression already exists. If an equivalent node v' exists, v_{new} is not added to the graph. Instead, an edge is added from v, the successor of v_{new}, to v' (line 13). If the cost

```
1  G := ({v_input}, ∅) = (V, E)                          # graph initialization
2  best_solution := c_∞                                  # cost initialization
3  stack := PriorityStack()                              # stack initialization
4  stack.push(0, v_input)                 # the root node is added to the stack
5  V_pruned := ∅
6  while ¬stack.empty() and elapsed_time < t_max:
7      (p, v) := stack.pop()
8      if cost(v) > best_solution:                       # node is pruned
9          V_pruned := V_pruned ∪ {(p, v)}
10         continue                                      # jump to line 6
11     v_new := next_successor(v)                        # successor creation
12     if ∃v' ∈ V with v_new ≡ v':                       # equivalent node exists
13         E := E ∪ {(v, v')}
14         update_cost(v', cost(v_new))
15         for p_p, v_p in V_pruned:                     # reactivate pruned nodes
16             if cost(v_p) < best_solution:
17                 stack.push(p_p, v_p)
18     else:
19         V := V ∪ {v_new}
20         E := E ∪ {(v, v_new)}
21         if ¬is_terminal(v_new):
22             stack.push(0, v_new)
23         else:
24             if cost(v_new) < best_solution:
25                 best_solution := cost(v_new)
26     stack.push(p + 1, v)    # the current node is added back to the stack
```

Figure 4.8: Pseudocode of the updated search algorithm with merging.

of v_{new} is lower than that of v', the cost of v' and its successors is updated (line 14). This is implemented by visiting all successors in topological sort order and updating their cost if necessary. Pruned nodes are reactivated by adding them back to the stack (lines 15–17).

4.2.4 *Diamonds*

Merging branches in the search graph frequently leads to so called diamonds. One such diamond is shown in Fig. 4.9. Intuitively, a diamond simply means that subexpressions or assignments can be computed independently of one another and in any order. In the worst case, for a sequence of n kernel calls that can be reordered in any way,

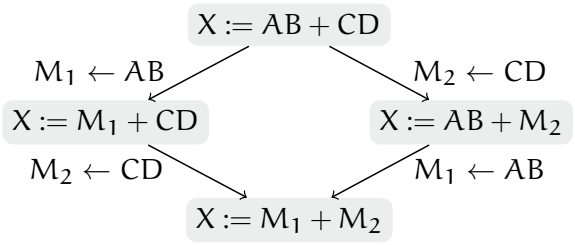


Figure 4.9: An example of a diamond in the search graph.

the search graph will contain $n!$ paths, one for each order (and 2^n nodes when all branches are merged).[3]

In isolation, diamonds can be considered redundant: The different sequences of kernels that are represented by a diamond can be obtained from any single path in the diamond by reordering the kernel calls. Thus, in order to reduce the size of the search graph, a diamond can be represented by a single edge that is annotated with a sequence of kernels in an arbitrary order. However, diamonds are not necessarily redundant in the context of a larger search graph: Specifically, intermediate nodes (for instance $X := M_1 + CD$ in Fig. 4.9) that lead to a solution that cannot be reached from the bottom node of the diamond are not redundant.

As discussed in Sec. 5.2, to some extent redundant diamonds can be avoided by designing the generation steps such that independent computations are avoided. However, since it is difficult to predict the interaction between different generation steps, it is difficult to avoid redundant diamonds entirely.

4.3 SUCCESSOR GENERATION

A crucial part of Linnea's search algorithm is the design of the `next_successor` function. In that function, the optimizations that Linnea uses to generate sequences of kernels are applied to the different representations of an expression. Those optimizations are encapsulated in *generation steps*. A generation step is a function that takes an expression as input and generates one or more output expressions, each together with a possibly empty sequence of kernels. The following generation steps are implemented in Linnea:

1. The exhaustive application of kernels through pattern matching (Ch. 5),

2. the application of kernels with constructive algorithms (Sec. 5.6), that is, an algorithm for matrix sums (Sec. 5.6.1), as well as the generalized matrix chain algorithm (Sec. 5.6.2),

3 For a sequence of n kernel calls that can be reordered in any way, diamonds generalize to n-dimensional hypercubes. The number of vertices of such hypercubes is 2^n, and the number of paths between two opposing vertices is $n!$.

3. the application of factorizations (Sec. 5.7),
4. common subexpression elimination (Ch. 8), and
5. the application of so called tricks (Sec. 7.5).

The design of Linnea allows for a high degree of freedom in the implementation of the individual generation steps, ranging from very simple steps that only use pattern matching to arbitrarily complex algorithms. Simple generation steps usually only change a small part of the expression and make relatively little progress towards a solution, while complex generation steps change larger parts of an expression and make more progress. However, since in the search graph, generation steps are applied repeatedly to the same expressions, the repeated application of a simple step can lead to the same result as a single application of a much more complex generation step. The generalized matrix chain algorithm is an example of a complex generation step that efficiently finds the optimal parenthesization of a matrix product. The disadvantage of this algorithm is that it can only make use of a limited set of kernels. In contrast, the exhaustive application of kernels through pattern matching inefficiently enumerates all possible parenthesizations, but pattern matching allows to make use of all available kernels. One of the strengths of Linnea is that it allows to combine both simple and complex generation steps for the same type of subproblem: The matrix chain algorithm is used to quickly find the optimal parenthesization of matrix products, while the exhaustive application of kernels may lead to an improved sequence of kernels by making use of additional kernels. More complex generation steps can also be used to implement heuristics to consider only the most promising alternatives. This is done for the application of factorizations and common subexpression elimination. Especially in the application of factorizations, the number of different combinations of factorizations that can be applied is usually much larger than the number of combinations that are likely to lead to a good solution. While the repeated application of a simple generation step that only applies one factorization at a time eventually generates all combinations, a more complex algorithm that analyses the expression helps to reduce the number of suboptimal solutions.

As mentioned in Sec. 4.1, for a given node, `next_successor` has to return the most promising successors first. There are several design decisions that determine the behavior of this function:

1. The order in which different representations of an expression are explored.
2. The order in which the generation steps are applied to an expression.
3. The combination of representations and generation steps.

4. Given a generation step that is applied to a representation of an expression, the order in which the generated output expressions are explored.

Most of these decisions are based on heuristics that encode the expertise of linear algebra library developers. Those decisions and heuristics are discussed in the following.

4.3.1 *Order of Representations*

For a given expression, Linnea uses up to four different representations. The normal form, which is a sum of products, two product of sums representations, and a sum of products representation where all inversion operators are pushed up as far as possible. The details of those representations as well as how they are obtained is described in Ch. 7. For simple expressions, some or all of those representations might be the same. If they are distinct, the different representations are used in the following order:

1. Product of sums (left first),
2. product of sums (right first),
3. sum of products with inversion pushed up, and
4. normal form.

The reason for beginning with the product of sums is that it reduces the number of expensive multiplications: While $AB + AC$ requires two matrix-matrix multiplications, $A(B + C)$ requires only one. Pushing up the inversion operator allows to decrease the number of matrix factorizations in favor of matrix-matrix multiplications: While $A^{-1}B^{-1}C$ requires two factorizations, $(BA)^{-1}C$ requires only one.

4.3.2 *Order of Generation Steps*

The following order of generation steps is used for all nodes except the root:

1. Application of kernels with constructive algorithms,
2. common subexpression elimination,
3. application of factorizations,
4. exhaustive application of kernels through pattern matching, and
5. application of tricks.

The idea is to begin with those steps that are most important to quickly find a relatively good solution. While factorizations are in many cases necessary to find a solution at all, there are also cases where the application of factorizations is possible, but not necessary to find a solution. Because of the latter, the application of factorizations is only the third step. If the application of factorizations is necessary to make progress, then eventually both the constructive algorithms and common subexpression elimination will not be able to generate new expressions. In that case, the application of factorizations effectively becomes the first step. The exhaustive application of kernels is only the fourth step because the number of applicable kernels is usually very large, while the benefit of the exhaustive application compared to the constructive algorithms tends to be low. Tricks come last because at present, only a small number of highly problem-specific tricks are implemented. As a result, there is a low chance that they are applicable. Should the number of tricks be increased, it might be beneficial to give them higher priority. The application of factorizations and tricks are not used again if they were used to generate the current node. The reason is that is is unlikely that the repeated application of those steps without another step in between is beneficial.

The order for the root node differs in that common subexpression elimination is the first step. This has the effect that, if there are any, common subexpressions will be replaced at least once along the path that leads to the first solution.

4.3.3 *Combination of Representations and Generation Steps*

For a given expression, every generation step is applied to every representation. Since it is very challenging to predict which generation step is the most promising for a given representation, we favor variety over depth: for instance, instead of first replacing all common subexpressions and then proceeding to factorizations, we replace one common subexpression, followed by one factorization, and continue in a round-robin fashion. The combination of all representations and generation steps can be thought of as a nested round-robin order. The idea is illustrated in Fig. 4.10. The outer round-robin cycles between all representations; for each representation, the inner round-robin cycles between the generation steps, every time returning one new expression per step. If for any combination of representation and generation step, no (further) expressions can be generated, this combination is skipped.

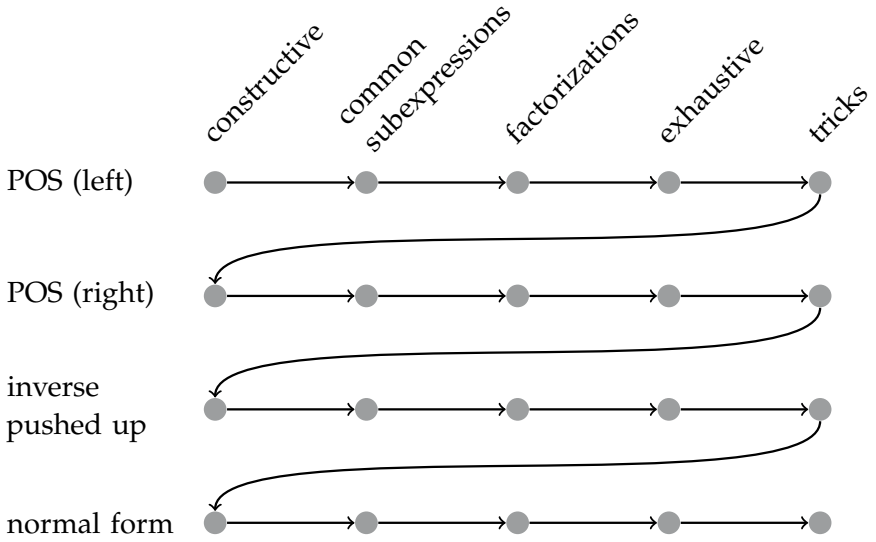


Figure 4.10: The order in which new expressions are generated from combinations of representations and generation steps (excluding the root node).

4.3.4 *Order of Generated Expressions*

The order of the expressions that result from the application of a generation step to a given representation is determined by the implementation of the respective generation step.

1. The constructive algorithms return only one new expression.
2. Those common subexpressions that allow to replace the largest number of operands at once are replaced first. The details are discussed on page 128.
3. The application of factorizations starts with those factorizations that are necessary to make progress, before also applying factorizations that may not be necessary. In addition, the cheapest factorizations are used first. The details are described on page 85.
4. For the exhaustive application of kernels, the order of the generated expressions is arbitrary. Is it determined by the order in which MatchPy finds matches for the applicable kernels.
5. The order of the application of tricks is again arbitrary and determined by the order in which MatchPy finds matches for the applicable tricks.

4.3.5 *Implementation*

Generators are functions that return a sequence of outputs one at a time. Since outputs can be returned anywhere in the function, the outputs can be returned as they are constructed. The execution of a generator is suspended after an output is returned, and it can be resumed later.

In the interest of performance, the implementation of the next_successor function heavily relies on generators. Generators allow to easily implement lazy evaluation. All generation steps as well as the functions that rewrite expressions into different representations are implemented as generators; the amount of computation that is performed to construct the next expression is reduced to the necessary minimum. Similarly, the implementation of the round-robin order is a generator that does not require the eager evaluation of the generation steps it iterates over. The use of generators allows to save computational effort if a node has a large number of possible successors, but only a few are actually explored. In addition, it decreases the time to a first solution.

4.4 EXISTENCE OF A SOLUTION AND TERMINATION

For Linnea to be useful in practice, it has to be guaranteed that 1) a solution exists for every valid input problem, and 2) that Linnea always terminates.

Trivially, to guarantee that a solution exists, it is sufficient to have one general kernel for every supported operation. In practice, Linnea uses a much larger set, including multiple kernels for the same operations that make use of different properties, as well as kernels that combine multiple operations. However, Linnea does not use kernels that compute the inverse of a full matrix, either explicitly or as part of a linear system. Instead, those operations are computed with matrix factorizations (Sec. 5.7). Since both the LU factorization and the singular value decomposition can be applied to full matrices, the inverse of full matrices can always be computed. As a result, with the set of kernels used by Linnea, a solution exists for every input problem.

There are two aspects that are relevant for termination; the application of kernels to a fixed expression, that is, an expression that is not rewritten, and the rewriting of expressions.

Excluding factorizations, the application of kernels to a fixed expression is guaranteed to terminate because every application decreases the size of the expression, that is, the number of nodes in the expression tree. With factorizations, the situation is more complicated: Repeatedly applying a matrix factorization and then undoing it by a matrix product can easily lead to infinite loops. In Linnea, such loops are avoided by labeling operands as factors and by requiring that for any given kernel call, there must be at least one operand that is not a factor. For instance, in the expression $S^{-1}B$, the Cholesky factorization is applied to S, resulting in $(L^TL)^{-1}B$. To compute the resulting expression, first the inverse has to be distributed over L^TL, yielding $L^{-1}L^{-T}B$. Then, the linear system $M = L^{-T}B$ is solved, which is allowed because B is not a factor, and the remaining linear system

$L^{-1}M$ can be solved too because M is not a factor either. An exception from this rule has to be made for explicit inversion. As an example, consider the assignment $X := S^{-1}$, where S is SPD. Since S cannot be inverted directly, the Cholesky factorization is applied to it, resulting in $X := L^{-1}L^{-T}$. In order to compute the right-hand side of this assignment, it is necessary to compute operations where all operands originate from the same factorization, which is usually not allowed. To ensure that such expression can be computed, during the application of kernels, Linnea searches for subexpressions that form an explicit inversion; those subexpressions are products where all operands are factors that originate from the same factorization. If such a subexpression is found, the requirement that the arguments of a kernel cannot originate from the same factorization is lifted and the generalized matrix-chain algorithm (Sec. 5.6.2) is used to compute the subexpression.

Rewriting can affect termination because it can lead to arbitrarily large expressions. This can happen for instance with the replacement of a single operand X by $(X^T)^T$, which can be repeated an arbitrary number of times. If after every application of a kernel, the size of the expressions grows due to rewriting, the algorithm generation may not terminate. In order to prevent such cases of non-termination, those rewritings that could potentially cause non-termination are implemented in a way such that they cannot be applied arbitrarily often (see Ch. 7). For instance, the replacement of X with $(X^T)^T$ is only used indirectly as part of the application of transposed kernels (Sec. 5.4); there is no general rule $X \to (X^T)^T$ that is applied unconditionally.

4.5 COST FUNCTION

For most inputs, Linnea generates many alternative programs, all mathematically equivalent, but with different performance signatures and numerical properties. To discriminate programs and to choose one that satisfies constraints such as memory usage, a cost function is necessary. This can either be an exact cost or an estimate. Such a function could take into account the number and the cost of kernel invocations (e.g., the number of floating-point operations performed, the number of bytes moved), and even the numerical stability of the program.

A cost function has to fulfill two requirements: 1) It has to be defined on any sequence of one or more kernels, and 2) its codomain has to be a totally ordered set. For some simple functions, such as the number of FLOPs, both conditions are satisfied. For many others, the first condition poses a challenge. For example, while the efficiency of individual kernels can be (tediously) modeled [71, 102], the efficiency of an arbitrary sequence of kernels is expensive to obtain via measurements and cannot be accurately derived by simply combining that of the

individual kernels [103]. Similarly, incorporating numerical stability into a cost function is a challenging task: It is not necessarily clear how to represent an error analysis by means of one or few numbers, it is still difficult to derive stability analyses even for individual kernels, and the analysis for a sequence of kernels is not a direct composition of the analyses of the kernels [17, 65].

In addition to the two requirements discussed above, it is useful (but not necessary) for the cost function to have the property that the cost of a sequence of kernels is the sum of the costs of the individual kernels. Formally, given two sequences of kernels p_1 and p_2, this means that $\text{cost}(p_1 p_2) = \text{cost}(p_1) + \text{cost}(p_2)$, where $p_1 p_2$ denotes the concatenation of the sequences p_1 and p_2. The reason is that this property allows to use standard graph search algorithms to find the shortest path in the graph. Without this property, it is necessary to enumerate all paths. Since the number of paths is exponential in the size of the graph, the enumeration of all paths is in many cases infeasible. If the cost function does not have this property, a simplified cost function that does have this property could be used to preselect a number of paths in the graph.

As a cost function, Linnea presently uses the number of FLOPs. This function has the advantage that it is easy to determine, and for the targeted regime of mid-to-large scale operands, it is usually a good proxy for the execution time. An evaluation of the effectiveness of the number of FLOPs as a cost function is carried out in Sec. 10.4.

For each kernel, Linnea has a formula that computes the number of FLOPs performed from the sizes of the matched operands. As an example, for the GEMM kernel—which computes $AB + C$ with $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$—the formula is $2mnk$. Those formulas were either taken from [66, pp. 336–337], or inferred by hand. To find the path in the search graph with the lowest cost, we use a K shortest paths algorithm [75]. In case of ties, an arbitrary path is selected. The K shortest paths algorithm is used to allow the user to inspect more than just the best solution. In addition, it is used to generate the best 100 algorithms used for the evaluation of the cost function in Sec. 10.4.

COMBINED OPERATIONS Several BLAS and LAPACK kernels combine the computation of multiple mathematical operations. For instance, the GEMV kernel, which computes $Ax + y$, combines a matrix-vector product with the sum of two vectors. In most of those cases, the same operations can also be computed with multiple calls to kernels that compute simpler operations. The operation $Ax + y$ can also be computed with a call to GEMV to compute the matrix-vector product Ax, followed by a call to AXPY to compute the sum of two vectors:

$$v_1 \leftarrow Ax$$
$$v_2 \leftarrow v_1 + x$$

Especially due to the constructive algorithms (see Sec. 5.6), it is likely that Linnea's search graph contains both solutions: The single GEMV call, as well as the sequences of two kernel calls. While mathematically both sequences are equivalent, in practice most likely the single kernel call is better in terms of performance because it avoids unnecessary data movement.

Care has to be taken when comparing such alternative solutions in terms of FLOPs; since both solutions carry out the same computations, they require the same number of FLOPs. In Linnea, those two solutions can still be distinguished because the cost function only considers the highest order term for each kernel. That is, the formula for the number of FLOPs of $Ax + y$ is $2mn$, where m and n are respectively the number of rows and columns of A, as opposed to the formula $2mn + n$ which also considers the cost of adding a vector. As a result, the cost of the solution that only consists of one call to GEMV is $2mn$, while the cost of the sequence of two kernels is $2mn + n$.

While explicit transposition does not require any FLOPs, to ensure that solutions that use explicit transposition are more expensive, in Linnea this kernel has a cost of 1 FLOP.

4.6 FORMAL DESCRIPTION

While many implementations details of Linnea are specific to the domain of linear algebra, the underlying idea of the algorithm generation is applicable to many other domains. In this section, we provide a formal description of the algorithm generation. This section serves two purposes:

1. The basic ideas of the algorithm generation in Linnea are presented from a high-level, theoretical point of view that is independent of the domain of linear algebra. This high-level description demonstrates the generality of the approach.

2. The computational complexity of the algorithm generation problem is discussed. Specifically, we show that already relatively simple cases of this problem are NP-complete.

The formal description builds upon elements from universal algebra and term rewriting. Thus, we make use of terminology, results, and notation from [5], which provides an in-depth overview of those subjects. This section is written to be accessible for readers unfamiliar with universal algebra and term rewriting. For this reason, in the presentation we favor simplicity and intuitive explanations based on examples over technical details and rigorous definitions. Nonetheless, the definitions in this section are entirely compatible with the ones in [5], and the definitions therein can be used instead.

Independent of the domain of linear algebra, the problem that Linnea solves can informally be stated as follows: Given

1. an input expression from an algebra,
2. a set of algebraic identities,
3. a set of functions that compute expressions from this algebra, and
4. a cost function that assigns a cost to every sequence of function calls,

find a sequence of function calls that 1) computes the input expression and 2) is optimal according to the cost function. In principle, the approach for the algorithm generation that is implemented in Linnea can be applied to every other problem that fits this description. Possible domains include tensor algebra, the simplification and efficient implementation of automatically generated index expressions, for example as they appear in Lift [107, Sec. 5.3], Optimal Jacobian Accumulation [98], query optimization for relational databases, and graph algorithms when formulated as linear algebra problems [27, 79]. In addition, the approach could also be applied to real or complex scalars, boolean algebra, quaternions or polynomials.

In order to formally define the algorithm generation problem, in the following we provide definitions of all the concepts used above. In Sec. 4.6.1, the algebra and algebraic identities are defined, followed by the computational functions, the cost function, and the problem itself in Sec. 4.6.2. The complexity of the problem is discussed in Sec. 4.6.3. In Sec. 4.6.4, the search graph that is used in Linnea is defined in terms of the formalism introduced in this section. A number of possible extensions to the formalism are outlined in Sec. 4.6.5.

4.6.1 Σ-*algebras*

Throughout the remainder of this thesis, we do not distinguish between the symbolic expressions themselves and their interpretation as mathematical objects in linear algebra, such as matrices or vectors. However, for the purpose of this section, this distinction is necessary. To this end, it is important to note that in Def. 3.2, expressions are defined as purely syntactic objects without any particular meaning. As an example, let $T(\Sigma, X)$ be a set of expressions with symbolic constants $1, 2, 3 \in \Sigma^{(0)}$ and a binary function $+ \in \Sigma^{(2)}$. According to the definition, the expressions $1+2, 2+1, 3 \in T(\Sigma, X)$ are all distinct, that is

$$1+2 \neq 2+1 \qquad 1+2 \neq 3 \qquad 2+1 \neq 3.$$

The intuitive notion that those three expressions all have the same value and are thus 'equal' was not defined yet. To introduce such a notion, an interpretation is necessary that assigns some meaning to the symbols 1, 2, 3, and $+$. An obvious interpretation is to consider

1, 2, and 3 as natural numbers and + as the addition of two natural number. Formally, such an interpretation is provided by a Σ-algebra:

Definition 4.1 (Σ-Algebra [5, Def. 3.2.1])**.** Let Σ be a signature. A Σ-*algebra* $\mathcal{A}$ consists of a carrier set (domain) A, and a mapping that associates with each function $f \in \Sigma^{(n)}$ a function $f^{\mathcal{A}} : A^n \to A$ for all $n \geqslant 0$. ■

The Σ-algebra provides an interpretation of the function symbols in Σ in a domain A. In the example above, one can choose A to be the set of natural numbers $\mathbb{N}$, and $+^{\mathcal{A}} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ to be the addition of two natural numbers. However, other interpretations are possible too; for instance, A could also be the set of integers $\mathbb{Z}$ or reals $\mathbb{R}$.

In order to formalize the connection between expressions in $T(\Sigma, X)$ and the corresponding elements in the Σ-algebra, we define an evaluation function that maps expressions to elements of the Σ-algebra.

Definition 4.2 (Evaluation Function)**.** Let Σ be a signature and $\mathcal{A}$ be a Σ-algebra. An *evaluation function* is a function $\varepsilon : T(\Sigma, X) \to A$ with $\varepsilon(f(t_1, \ldots, t_n)) := f^{\mathcal{A}}(\varepsilon(t_1), \ldots, \varepsilon(t_n))$ for all $n \geqslant 0$, $f \in \Sigma^{(n)}$, $t_1, \ldots, t_n \in T(\Sigma, X)$. ■

In the following example, in order to distinguish between the symbols $1, 2, 3 \in \Sigma^{(0)}$ and natural number in $\mathbb{N}$, we use a bold font for the numbers; they are written as $\mathbf{1}, \mathbf{2}, \mathbf{3} \in \mathbb{N}$. Let ε be an evaluation function with

$$\varepsilon(1) = \mathbf{1} \qquad \varepsilon(2) = \mathbf{2} \qquad \varepsilon(3) = \mathbf{3}.$$

With this function, the expression $1 + 2$ evaluates to

$$\varepsilon(1 + 2) = \varepsilon(1) +^{\mathcal{A}} \varepsilon(2) = \mathbf{1} +^{\mathcal{A}} \mathbf{2} = \mathbf{3}.$$

Thus, it is now possible to express the fact that the expressions $1 + 2$, $2 + 1$, and 3 have the same value and represent the same object in the Σ-algebra (the number **3**) as

$$\varepsilon(1 + 2) = \varepsilon(2 + 1) = \varepsilon(3) = \mathbf{3}.$$

It should be noted that there are many different evaluation functions for a given Σ-algebra. Let $a, b, c \in \Sigma^{(0)}$, and ε, ε' be evaluation functions with

$$\begin{aligned} \varepsilon(a) &= \mathbf{1} & \varepsilon(b) &= \mathbf{2} & \varepsilon(c) &= \mathbf{3} \\ \varepsilon'(a) &= \mathbf{1} & \varepsilon'(b) &= \mathbf{1} & \varepsilon'(c) &= \mathbf{3}. \end{aligned}$$

Whether or not the expressions $a + b$ and c are 'equal' now depends on the evaluation function. While $\varepsilon(a + b) = \varepsilon(c)$ holds, $\varepsilon'(a + b) = \varepsilon'(c)$ does not.

In many cases, functions in a Σ-algebra have some properties that can be described in terms of expressions. As an example, the addition of two natural number is commutative, that is, the expressions $x + y$ and $y + x$ have the same value for all values of x and y. In universal algebra, this notion is expressed with Σ-identities:

Definition 4.3 (Σ-identities [5, Def. 3.1.7])**.** Let Σ be a signature. A Σ-*identity* is a pair $(s,t) \in T(\Sigma,X) \times T(\Sigma,X)$. Identities will be written as $s \approx t$. In the follow, we use E to denote a set of Σ-identities. ∎

Σ-identities can be seen as axioms of a Σ-algebra $\mathcal{A}$. However, by themselves, those identities are again purely syntactic objects with no connection to $\mathcal{A}$; we need to require that $\mathcal{A}$ is 'compatible' with E. Intuitively, the functions $f^{\mathcal{A}}$ need to actually have the properties that are described by the identities in E. If there is for example an identity $f(x,y) \approx f(y,x)$ that describes that f is commutative, then $f^{\mathcal{A}}$ has to be commutative too. More formally, we require that $\mathcal{A}$ is a model of E [5, Def. 3.5.1 and Def. 3.5.2], that is, for all identities $s \approx t \in E$ and all possible evaluation functions ε, $\varepsilon(s) = \varepsilon(t)$ holds.[4]

If we think of Σ-identities as axioms of a Σ-algebra $\mathcal{A}$, it is clear that those identities do not just provide information about functions over $\mathcal{A}$; they can also be used to rewrite expressions without changing their value. This notion naturally leads to a relation on expressions; while $1+2$ and $2+1$ are distinct expressions, they are related through the fact that one can be rewritten into the other by using the identity $x+y \approx y+x$ as a rewrite rule $x+y \to y+x$. This relation is formalized as follows:

Definition 4.4 (Equational Theory[5])**.** Let E be a set of Σ-identities. The relation $\approx_E \subseteq T(\Sigma,X) \times T(\Sigma,X)$ is called the *equational theory* induced by E. Given two expressions $s,t \in T(\Sigma,X)$, $s \approx_E t$ holds if $s = t$ or if it is possible to rewrite s into t (and vice versa) with the identities in E. ∎

Thus, with $x+y \approx y+x \in E$, it holds that $1+2 \approx_E 2+1$. In this case, only one application of one identity in E is necessary to rewrite $1+2$ to $2+1$. In general, the number of rewrite steps can be arbitrarily large, and identities can be used multiple times and in both directions.

The equational theory $\approx_E$ forms an equivalence relation on the set of expressions $T(\Sigma,X)$. In the following, we use $[t]_{\approx_E}$ to denote the equivalence class of the expression t, which is defined as $[t]_{\approx_E} := \{t' \in T(\Sigma,X) \mid t \approx_E t'\}$. Intuitively, it is clear that two expressions that can be rewritten into one another by means of the identities in E have the same value. Formally, this means that given a Σ-algebra $\mathcal{A}$ that is a model of E and an evaluation function ε, $s \approx_E t$ implies $\varepsilon(s) = \varepsilon(t)$. The other direction does not need to hold, though. While $\varepsilon(1+2) = \varepsilon(3)$, with $E = \{x+y \approx y+x\}$ the equivalence $1+2 \approx_E 3$ does not hold.

4 Technically, $\mathcal{A}$ is a model of E if for all homomorphisms $\varphi : T(\Sigma,X) \to A$, $\varphi(s) = \varphi(t)$ holds. An evaluation function as defined above is such a homomorphism.

5 This definition is closer to the definition of the rewrite relation $\overset{*}{\leftrightarrow}_E$ in [5, p. 40] than the definition of equational theories [5, Def. 3.5.3]. However, since $\overset{*}{\leftrightarrow}_E$ and $\approx_E$ coincide according to Birkhoff's Theorem [5, Th. 3.5.14], they can be used interchangeably. We chose this definition because we find it more intuitive.

If E is empty, then expressions cannot be rewritten and $\approx_E$ reduces to the syntactic equality $=$, that is, $s \approx_E t$ if and only if $s = t$.

4.6.2 *Optimal Program Generation*

By making use of definitions from the previous sections, we can work towards defining the problem itself. We begin with the so called *kernel functions* that are used to compute parts of the input expression. Each function computes one operation that can be described in terms of an expression. As examples, for the TRSM kernel, this expression is $X^{-1}Y$; a fused multiply-add instruction computes $xy + z$. The corresponding kernel functions can be thought of as functions $f(X, Y) = X^{-1}Y$ and $g(x, y, z) = xy + z$, respectively. Kernel functions are defined as follows:

Definition 4.5 (Kernel Function)**.** Let Σ be a signature, ε be an evaluation function, and $t_f \in T(\Sigma, X)$ be an expression with $\mathrm{Var}(t_f) = \{x_1, \ldots, x_n\}$. A *kernel function* is a function $f : (\Sigma^{(0)})^n \to \Sigma^{(0)}$ with exactly one argument for each variable in $\mathrm{Var}(t_f)$.[6]

1. Given a substitution $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ with $t_1, \ldots, t_n \in \Sigma^{(0)}$, we define the application of f to $t_1, \ldots, t_n$ as

   $$\sigma(f(x_1, \ldots, x_n)) := f(\sigma(x_1), \ldots, \sigma(x_n)) = f(t_1, \ldots, t_n).$$

   As an abbreviation, we also write $\sigma(f(x_1, \ldots, x_n))$ as $\sigma(f)$.

2. For all substitutions σ with $\mathrm{Dom}(\sigma) = \mathrm{Var}(t_f)$, it holds that $\varepsilon(\sigma(f(x_1, \ldots, x_n))) = \varepsilon(\sigma(t_f))$.

In the following, we use $\mathcal{F}$ to denote a set of kernel functions. ■

In 1., the application of a function to a set of arguments is defined in terms of a substitution because the substitution creates a straightforward connection to t_f: Given a kernel function f with $t_f = xy + z$, the expression $t = 2a + b$ can be computed with f because t_f matches t with the substitution $\sigma = \{x \mapsto 2, y \mapsto a, z \mapsto b\}$, that is, $\sigma(t_f) = t$. The same substitution σ can also be applied to f, resulting in $\sigma(f) = f(2, a, b)$. With 2., it is ensured that the constant that is returned by the application of f has the same value as the expression that is computed. In the example, this means that $\varepsilon(f(2, a, b)) = \varepsilon(2a + b)$. f is defined on constants $\Sigma^{(0)}$ to ensure that expressions can only be used as the input to a kernel function once they have been fully computed. As an example, let $g, h \in \mathcal{F}$ be functions with $t_g = xy$ and $t_h = x + y$. While $g(2, a)$ can be used to compute $2a$, it is not possible to compute $2a + b$ with $h(2a, b)$ since $2a \notin \Sigma^{(0)}$, which means that $2a$ has not been computed yet.

6 Kernel functions that return more than one output operand are discussed in Sec. 4.6.5.

It should be noted that since kernel functions are defined over symbolic constants $\Sigma^{(0)}$ instead of elements of A, they are not mathematical functions in the usual sense. While such functions are not necessary for the discussion in this section, one could require that for every kernel function $f \in \mathcal{F}$, there is a function $f^{\mathcal{A}}$ with

$$\varepsilon(\sigma(f(x_1, \ldots, x_n))) = f^{\mathcal{A}}(\varepsilon(\sigma(x_1)), \ldots, \varepsilon(\sigma(x_n))).$$

In practice, kernel functions can also have constraints regarding the operands. For instance, a function f with $t_f = \frac{x}{y}$ could have the constraint $\varepsilon(y) \neq 0$. In that case, f can only be applied to operands that satisfy the constraints.

Next, the application of a kernel function to an expression is defined. Given a kernel function f and an expression s, a subexpression of s can be computed with f is there is a match σ of t_f in s. The matching subexpression $\sigma(t_f)$ is then replaced with the output of $\sigma(f)$. As an example, we use again the expression $2a + b$ and a function f with $t_f = xy$. There is a match for t_f in $2a + b$ with substitution $\sigma = \{x \mapsto 2, y \mapsto a\}$. The subexpression $\sigma(t_f) = 2a$ can thus be computed as $\sigma(f) = f(2, a) = c$, resulting in the expression $c + b$.

Definition 4.6 (Application of Kernel Functions)**.** Let $s \in T(\Sigma, \varnothing)$ be an expression[7], and $f \in \mathcal{F}$ be a kernel function that computes t_f. Given a position p and a substitution σ with $s|_p = \sigma(t_f)$ and $t = s[\sigma(f)]_p$, the application of f to s is defined as $s \xrightarrow{\sigma(f)} t$. ∎

If it is convenient, we spell out $\sigma(f)$ in a kernel function application. For instance, in the example above we also write the application of f with substitution $\sigma = \{x \mapsto 2, y \mapsto a\}$ as

$$2a + b \xrightarrow{c=f(2,a)} c + b.$$

Following from $\varepsilon(\sigma(t_f)) = \varepsilon(\sigma(f))$ in Def. 4.5, for $s \xrightarrow{\sigma(f)} t$ it holds that $\varepsilon(s) = \varepsilon(t)$, that is, the value of an expression does not change with the application of a kernel function.

Given an expression s and a kernel function f, there can be multiple matches of t_f in s. This is for example the case for $ab + cd$ and $t_f = xy$; t_f matches both ab and cd. If there are multiple matches, it is possible to obtain different results by the application of f. That is, if there are two matches σ_1, σ_2 of f in s, then there are two different applications $s \xrightarrow{\sigma_1(f)} t_1$ and $s \xrightarrow{\sigma_2(f)} t_2$ with $t_1 \neq t_2$. In addition, it is also possible that there are multiple identical matches σ at different positions p_i, which means that there is a common subexpression in s. Since all matching subexpressions $\sigma(t_f)$ can be computed with the same function application $\sigma(f)$, as a form of common subexpression elimination it is possible to replace $\sigma(t_f)$ with $\sigma(f)$ at all positions p_i.

7 This definition only applies to single expressions, not sequences of assignments. The extension to sequences of assignments is discussed in Sec. 4.6.5.

The kernel functions are used to construct programs. Specifically, the idea is to repeatedly apply kernel functions to an input expression until the expression reduces to a single constant. The sequence of function applications then forms a program that computes the input expression. Since the use of Σ-identities can improve the quality of programs, we allow that expressions can be rewritten before and after function applications.

Definition 4.7 (Partial Program)**.** Let $t_0, t_k \in T(\Sigma, \varnothing)$ be two expressions with $t_0 \neq t_k$, E be a set of Σ-identities, and $\mathcal{F}$ be a set of kernel functions. A *partial program* is a sequence of applications of kernel functions

$$\begin{aligned} t'_0 &\xrightarrow{\sigma_1(f_1)} t_1 \\ t'_1 &\xrightarrow{\sigma_2(f_2)} t_2 \\ &\vdots \\ t'_{k-1} &\xrightarrow{\sigma_k(f_k)} t'_k \end{aligned}$$

with $t_i \approx_E t'_i$ for all $0 \leqslant i \leqslant k$. In the following, we also denote such a program as $t_0 \xrightarrow{p}_E t_k$ with $p = ((\sigma_1, f_1), \ldots, (\sigma_k, f_k))$. Since the output operands are uniquely determined by $\sigma_i(f_i)$, they are omitted from p. We use $\mathcal{P}(\mathcal{F})$ for the set of all programs that can be constructed from functions in $\mathcal{F}$. ∎

A program $t_0 \xrightarrow{p}_E t_k$ with $p = ((\sigma_1, f_1), \ldots, (\sigma_k, f_k))$ can also be understood as a sequence of function applications

$$\begin{aligned} c_1 &= \sigma_1(f_1) \\ c_2 &= \sigma_2(f_2) \\ &\vdots \\ c_k &= \sigma_k(f_k) \end{aligned}$$

with input expression t_0 and output expression t_k.

Example 4.5. Let E be a set of Σ-identities that contains

$$\begin{aligned} x + y &\approx y + x \\ xy &\approx yx \\ x(y + z) &\approx xy + xz, \end{aligned}$$

let $f, g, h \in \mathcal{F}$ be kernel functions with

$$t_f = xy + z \qquad t_g = xy \qquad t_h = x + y$$

For the input expressions $ab + 3a$, one possible program is

$$\begin{aligned} ab + 3a &\xrightarrow{c_1 = g(a,b)} c_1 + 3a \\ 3a + c_1 &\xrightarrow{c_2 = f(3,a,c_1)} c_2 \end{aligned}$$

with $c_1 + 3a \approx_E 3a + c_1$. Alternatively, it is also possible to rewrite the input expression to $a(b+3)$ and construct the program

$$a(b+3) \xrightarrow{c_1 = h(b,3)} ac_1$$
$$ac_1 \xrightarrow{c_2 = g(a,c_1)} c_2$$

It should be noted that it is possible for programs to compute the input expressions only partially. That is, it is not required that t_k is a constant in $\Sigma^{(0)}$. As an example,

$$ab + 3a \xrightarrow{c_1 = g(a,b)} c_1 + 3a$$
$$c_1 + 3a \xrightarrow{c_2 = g(3,a)} c_1 + c_2$$

is a valid partial program. ■

Partial programs have several useful properties:

1. By construction, the function applications $\sigma_i(f_i)$ are ordered such that p can be executed in the usual sense, that is, operands are only used as input to a function after they have computed. This can be seen as follows: Let $t_{i-1} \xrightarrow{\sigma_i(f_i)} t_i$ be a function application in a partial program. σ_i is a match of t_{f_i} in t_{i-1}, and according to Def. 4.5, σ_i can only map to constants in $\Sigma^{(0)}$. The constants in t_{i-1} are either constants that already appeared in t_0, that is, they are part of the input of the program, or they are the result of a previous function application $\sigma_j(f_j)$ with $j < i$. In addition, since ε provides a fixed mapping of operands to their values, p is in static single assignment (SSA) form.

2. For all programs $t_0 \xrightarrow{p}_E t_k$, it holds that $\varepsilon(t_0) = \varepsilon(t_k)$, that is, the input and output expressions have the same value. The reason is that for all function applications $s \xrightarrow{\sigma(f)} t$, $\varepsilon(s) = \varepsilon(t)$ holds, and for all expressions s and t, $s \approx_E t$ implies $\varepsilon(s) = \varepsilon(t)$. Thus, if the output expression t_k is a constant ($t_k \in \Sigma^{(0)}$), then p fully computes the input expression t_0, that is $\varepsilon(c_k) = \varepsilon(t_0)$.

3. Since the input and output of a partial program $t_0 \xrightarrow{p}_E t_k$ are fully specified by t_0 and t_k, partial programs can be concatenated as follows: Let $t_0 \xrightarrow{p_1}_E t_l$ and $t'_l \xrightarrow{p_2}_E t_k$ be partial programs with $t_l \approx_E t'_l$. Their concatenation results in a partial program $t_0 \xrightarrow{p_1 p_2}_E t_k$, where $p_1 p_2$ denotes the concatenation of the sequences p_1 and p_2.

As a consequence of 1. and 2., programs are correct by construction.

Unfortunately, programs can have an infinite length. As an example, let $g \in \Sigma^{(1)}$ be a unary function, $E = \{x \approx g(g(x))\}$, and $f \in \mathcal{F}$ be a function with $t_f = g(x)$. In this case, there is a partial program

$$g(a) \xrightarrow{b = g(a)} b \approx_E g(g(b)) \xrightarrow{a = g(b)} g(a)$$

which can be repeated infinitely often.

Since the goal is to find a program that is in some way optimal, we define a cost function on programs.

Definition 4.8 (Cost Function)**.** Let C be a totally ordered set. A *cost function* is a function $\text{cost} : \mathcal{P}(\mathcal{F}) \to C$. ∎

The requirement that C is a totally ordered set is necessary to ensure that the cost of all programs is comparable.

Finally, we can define the problem of finding an optimal program that computes a given input expression.

Definition 4.9 (Optimal Program Generation (OPG))**.** Given an expression $t_0 \in T(\Sigma, \varnothing)$, a cost function, a set of Σ-identities E, and a set of kernel functions $\mathcal{F}$, *Optimal Program Generation* is the problem of finding a program $t_0 \xrightarrow{p}_E t_k \in \mathcal{P}(\mathcal{F})$ with $t_k \in \Sigma^{(0)}$ that minimizes $\text{cost}(t_0 \xrightarrow{p}_E t_k)$. ∎

Instead of an optimization problem, this problem can also be stated as a decision problem by choosing a fixed cost c and asking whether a program exists with $\text{cost}(t_0 \xrightarrow{p}_E t_k) \leqslant c$. A solution trivially exists if for every function $f \in \Sigma^{(n)}$ there is one function $f' \in \mathcal{F}$ with $t_{f'} = f(x_1, \ldots, x_n)$. If there are no other functions in $\mathcal{F}$ and E is empty, then the solutions only differ in the order of function applications, and in whether or not common subexpressions are computed more than once. Multiple solutions may exists either if there are additional kernel functions in $\mathcal{F}$, and/or if E is not empty.

It should be noted that the Def. 4.9 is a very basic definition of the problem. In Sec. 4.6.5, we present a number of extensions that are used in Linnea.

4.6.3 *Complexity*

Due to its generality, many different types of problems can be formulated as an instance of OPG. In this section, we show that OPG is NP-complete already with relatively simple algebras. Specifically, we show that an algebra with two binary functions, out of which one is associative and commutative, is sufficient for NP-completeness. In the following, we refer to this variant of OPG as OPG_{AC}.

We prove that OPG_{AC} is NP-complete by reduction from Ensemble Computation (EC) [46, Problem PO9], which is known to be NP-complete. By showing that for every instance of EC there is an equivalent instance of OPG_{AC}, we show that OPG_{AC} is at least as difficult as EC. The definition of EC is provided below.

Definition 4.10 (Ensemble Computation)**.** Let $C = \{C_k \subseteq A \mid k = 1, \ldots, n\}$ be a collection of subsets of a finite set A, and Ω be a positive integer. Is there a sequence $u_i = s_i \cup t_i$ for $i = 1, \ldots, \omega$, $\omega \leqslant \Omega$, where s_i and t_i are either $\{a\}$ for some $a \in A$, or u_j for some $j < i$ and $s_i \cap t_i = \varnothing$, 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. An instance of EC (left) and its solution (right) is shown below:[8]

$$
\begin{aligned}
A &= \{a_1, a_2, a_3, a_4\} & u_1 &= \{a_1\} \cup \{a_2\} = C_1 \\
C &= \{\{a_1, a_2\}, \{a_1, a_2, a_3\}, \{a_2, a_3, a_4\}\} & u_2 &= \{a_2\} \cup \{a_3\} \\
\Omega &= 4 & u_3 &= \{a_1\} \cup u_2 = C_2 \\
& & u_4 &= \{a_4\} \cup u_2 = C_3
\end{aligned}
$$

For the proof of NP-completeness, we use a Σ-algebra that allows for a straightforward translation of instances of EC to instances of $\mathrm{OPG}_{\mathrm{AC}}$. Specifically, we use a Σ-algebra $\mathcal{B}$ with two binary operations $\Sigma^{(2)} = \{\cup, c\}$, where $\cup$ is the union of two sets, and c is an unspecified function that is used to represent the enclosing set C in the input. E contains the three Σ-identities

$$
\begin{aligned}
x \cup y &\approx y \cup x \\
x \cup (y \cup z) &\approx (x \cup y) \cup z \\
c(c(x, y), z) &\approx c(x, c(y, z)).
\end{aligned}
$$

It should be noted that associativity of c is not required for the proof; c is only defined to be associative here because it simplifies the construction of the input expression t_0. The set of kernel functions $\mathcal{F}$ contains two functions: 1) $f(x, y)$ with $t_f = x \cup y$ and the constraint that $x \cap y = \varnothing$, and 2) $g(x, y)$ with $t_g = c(x, y)$. The cost of a program is given by the number of applications of f. Since c is only used to represent C, g does not contribute to the cost. Thus, formally the cost of a program $t_0 \xrightarrow{p}_E t$ is defined as

$$
\mathrm{cost}\left(t_0 \xrightarrow{p}_E t\right) := \sum_{\sigma(h) \in p} \mathrm{cost}(\sigma(h))
$$

with

$$
\mathrm{cost}(\sigma(h)) := \begin{cases} 1 & \text{if } h = f \\ 0 & \text{if } h = g \end{cases}.
$$

Finally, $\mathrm{OPG}_{\mathrm{AC}}$ is defined as a decision problem, that is, the goal is to determine whether or not a program exists with $\mathrm{cost}(t_0 \xrightarrow{p}_E t) \leqslant \Gamma$.

Theorem 4.1. $\mathrm{OPG}_{\mathrm{AC}}$ is NP-complete.

Proof: For each instance of EC, an equivalent instance of $\mathrm{OPG}_{\mathrm{AC}}$ is obtained as follows. For all $a_i \in A$, let there be a $\{a_i\} \in \Sigma^{(0)}$. The input expression t_0 is constructed as $t_0 = c(t_1, \ldots, t_n)$ with $t_k =$

8 This example was taken from [98].

$\{a_1\} \cup \ldots \cup \{a_j\}$ for $a_1, \ldots, a_l \in C_k$. Let $\Gamma = \Omega$. A solution of OPG_{AC} can be verified in polynomial time by executing the output program $t_0 \xrightarrow{p}_E t$ and checking that all sets C_k are computed, as well as checking that $\text{cost}(t_0 \xrightarrow{p}_E t) \leqslant \Gamma$. ∎

With this reduction, the instance of OPG_{AC} that corresponds to the instance of EC that is shown above is:

$$\{\{a_1\},\{a_2\},\{a_3\},\{a_4\}\} \subset \Sigma^{(0)}$$
$$t_0 = c(\{a_1\} \cup \{a_2\}, \{a_1\} \cup \{a_2\} \cup \{a_3\}, \{a_2\} \cup \{a_3\} \cup \{a_4\})$$
$$\Gamma = 4.$$

In the solution $t_0 \xrightarrow{p}_E t$, the sequence p is

$$u_1 = f(\{a_1\}, \{a_2\})$$
$$u_2 = f(\{a_2\}, \{a_3\})$$
$$u_3 = f(\{a_1\}, u_2)$$
$$u_4 = f(\{a_4\}, u_2)$$
$$u_5 = g(u_1, u_3)$$
$$t = g(u_5, u_4).$$

with $\varepsilon(u_1) = C_1$, $\varepsilon(u_3) = C_2$, and $\varepsilon(u_4) = C_3$.

REMARKS Instead of using c to represent the collection of sets C, for the input expression t_0 one could also use a sequence of assignments with one assignment for every C_k, or a tuple of expressions with one expression for every C_k. Here, c is used to show that a sequence of assignments is not necessary for OPG_{AC} to be NP-complete; a single input expression is sufficient. In addition, instead of $\cup$, many other binary associative-commutative operations could be used, for example scalar multiplication or addition, or the addition of matrices or vectors.

4.6.4 *Program Generation via Graph Search*

In Linnea, OPG is solved with a graph search. In the following, the search graph is defined in terms of the formalism introduced in this section: The nodes of the graph are equivalence classes of expressions, and there is an edge between two classes $[s]_{\approx_E}$ and $[t]_{\approx_E}$ if there is a $f \in \mathcal{F}$ with $s \xrightarrow{\sigma(f)} t$. Finding a program that is optimal according to a given cost function is equivalent to finding the shortest path in this graph from the root node to a node with $t' \in [t]_{\approx_E}$ and $t' \in \Sigma^{(0)}$.

Definition 4.11 (Program Graph)**.** Given an expression $t_0 \in T(\Sigma, \varnothing)$, a set of Σ-identities E, and a set of kernel functions $\mathcal{F}$, a *program graph* $G(t_0, \mathcal{F})$ is a tuple (V, E) with $V \subseteq \{[t]_{\approx_E} \mid t \in T(\Sigma, \varnothing)\}$ and $E \subset V \times \mathcal{S} \times \mathcal{F} \times V$ such that

1. $V = \{[t_0]_{\approx_E}\} \cup \{[t]_{\approx_E} \mid \text{there is a program } t_0 \xrightarrow{p}_E t \in \mathcal{P}(\mathcal{F})\}$, and

2. $E = \{([s]_{\approx_E}, \sigma, f, [t]_{\approx_E}) \mid \text{there is a program } s \xrightarrow{\sigma(f)} t \in \mathcal{P}(\mathcal{F})\}$. ∎

Given an expression t_0 and a set of kernel functions $\mathcal{F}$, $G(t_0, \mathcal{F})$ contains all possible programs consisting of functions in $\mathcal{F}$ that compute t_0. Since there can be programs $t \xrightarrow{p}_E t$, the graph can contain loops.

If $\mathcal{F}$ and E are large, for some input expression t_0 the graph $G(t_0, \mathcal{F})$ might be so large that it is infeasible to fully construct it. In those cases, as done in Linnea, the only feasible approach might be to construct only parts of the graph during the search either until a sufficiently good solution is found, or until a time or memory limit is reached.

In order to be able to use standard graph search algorithms, it is useful to require that the cost function satisfies the property that for any two programs $t_0 \xrightarrow{p_1}_E t_l$ and $t'_l \xrightarrow{p_2}_E t_k$ with $t_l \approx_E t'_l$ it holds that

$$\text{cost}\left(t_0 \xrightarrow{p_1}_E t_l\right) + \text{cost}\left(t'_l \xrightarrow{p_2}_E t_k\right) = \text{cost}\left(t_0 \xrightarrow{p_1 p_2}_E t_k\right).$$

Without this property, it might be necessary to enumerate all paths in a graph to find the optimal solution. Again, in many cases the exhaustive enumeration might be infeasible because the number of paths is exponential in the size of the graph.

MERGING The use of equivalence classes as nodes allows for a compact representation of a large number of programs. However, it also requires to identify whether or not two expressions $s, t \in T(\Sigma, \varnothing)$ belong to the same equivalence class, that is, it is necessary to decide if $s \approx_E t$ holds. This problem is known as the ground word problem, and it is in general undecidable [5, p. 59]. As described in Sec. 4.2, in Linnea a normal form is used to decide if two expressions are equivalent: Given two expressions s and t together with their normal forms s' and t', $s \approx_E t$ can be decided with the syntactic comparison $s' = t'$. However, as mentioned in Sec. 4.2, in Linnea it is possible that this procedure fails because in some cases two equivalent expressions have different normal forms $s' \neq t'$. Fortunately, if two expressions cannot be correctly identified as equivalent, this only has the effect that an opportunity for merging nodes is missed and the graph becomes larger than necessary; the correctness of the generated programs is not affected.

An important aspect in the identification of equivalent expressions is the use of unique intermediate operands as described in Sec. 4.2.1. That is, given an input expression t_0 with a subexpression $t = t_0|_p$, and two function applications $\sigma_1(f_1)$ and $\sigma_2(f_2)$ with $\varepsilon(t) = \varepsilon(\sigma_1(f_1)) = \varepsilon(\sigma_2(f_2))$, the goal is to ensure that $\sigma_1(f_1) = \sigma_2(f_2)$. While this property is not required by Def. 4.5, it is useful to increase the number of expressions that can be identified as equivalent and consequently reduce the size of the search graph.

4.6.5 *Extensions*

The formalism presented in this section can be extended in several ways. In the following, some extensions which are used in Linnea are discussed.

FACTORIZATIONS Kernel functions as defined in Def. 4.5 cannot be used to represent matrix factorizations. The reason is that instead of a single output operand, factorizations produce an output expression that may contain more than one output operand. For instance, the symmetric eigenvalue decomposition produces an output expression ZWZ^T that contains two output operands W and Z. Factorizations can be incorporated into the formalism presented in this section by defining a second type of functions that have an output expression and produce multiple output operands. In case of the symmetric eigenvalue decomposition, this output expression is YXY^T, and the factorization function produces as output the tuple (W, Z). A substitution $\tau = \{X \mapsto W, Y \mapsto Z\}$ is used to construct the output expression ZWZ^T.

Definition 4.12 (Factorization Function)**.** Let Σ be a signature, ε be an evaluation function ε, and $t_f \in T(\Sigma, X)$ be an expression with $\mathrm{Var}(t_f) = \{x_1, \ldots, x_n\}$. A factorization function is a function $f : \Sigma^{(0)} \to (\Sigma^{(0)})^n$ with one input argument x.

Given a substitution σ with $\mathrm{Dom}(\sigma) = \{x\}$, an output tuple $\sigma(f(x)) = (t_1, \ldots, t_n)$, and a substitution $\tau = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, it holds that $\varepsilon(\sigma(f(x))) = \varepsilon(\tau(t_f))$. ■

When a factorization function is applied to an expression s at position p with $s|_p = \sigma(x)$, the resulting expression is constructed as $t = s[\tau(t_f)]_p$.

Instead of defining factorization functions separately from kernel functions, it would also be possible to define more general kernel functions that also cover factorizations by allowing for an input expression t_f^{in}, an input expression t_f^{out}, and an arbitrary number of both input and output operands.

SEQUENCES OF ASSIGNMENTS Instead of using an expression as input, it is also possible to generate programs that compute a sequence of assignments. In such a sequence, there can be dataflow between assignments, that is, the left-hand side of an assignment can be appear in the right-hand side of a subsequent assignment. All operands that do not appear on the left-hand side of an assignment are input operands whose value is known. Since constants in $\Sigma^{(0)}$ represent unique mathematical objects, their value cannot change. As a result, repeated assignments to the same constant are not possible. Sequences of assignments can be defined as follows:

Definition 4.13 (Sequence of Assignments)**.** A *sequence of assignments* is a tuple of the form $(l_1 := r_1, \ldots, l_n := r_n)$, with $l_i \in \Sigma^{(0)}$, $r_i \in T(\Sigma, \varnothing)$, and $\varepsilon(l_i) = \varepsilon(r_i)$ for all i. In addition, we require that

1. for all i, j with $i \neq j$, $l_i \neq l_j$ holds, and
2. for all i, l_i does not appear in any r_j with $j \leqslant i$. ∎

Requirement 1. ensures that there is at most one assignment to a given constant, while 2. ensures that the left-hand side of an assignment is not used before the assignment. In addition, in a partial program, a constant $c = l_i$ that appears inside r_j with $j > i$ can only be used as input to a function $f \in \mathcal{F}$ if $r_i \in \Sigma^{(0)}$, that is, once the right-hand side of the assignment to l_i has been fully computed. A program $t_0 \xrightarrow{p}_E t_k$ fully computes the input expression t_0 if in t_k, $r_i \in \Sigma^{(0)}$ holds for all i.

If dataflow between different assignments is not required, as a simpler alternative to a sequence of assignments, it would also be possible to use a tuple of expressions.

COMMON SUBEXPRESSIONS There are different ways to incorporate common subexpression elimination into the presented approach:

1. As mention above, if for a function f and an expression s there are multiple positions p_i with $s|_{p_i} = \sigma(t_f)$, it is possible to replace all occurrences of $\sigma(t_f)$ with $\sigma(f)$ during a single function application.
2. In a sequence of assignments, it is possible to extract common subexpressions into separate assignments. This approach is implemented in Linnea (Ch. 8).
3. In a program with $\sigma_i(f_i) = \sigma_j(f_j)$ where $j > i$, it is possible to remove the second, redundant function application $\sigma_j(f_j)$.

REWRITING AS GENERATION STEPS In Linnea, instead of increasing the number of different representations that expressions are rewritten to, it proved to be useful to implement the exploration of some types of representations as generation steps. Recall that given a node $[t]_{\approx_E}$ in a graph $G(t_0, \mathcal{F})$, different representations are considered during the application of kernel functions by first rewriting t into another expression s with $s \approx_E t$. Instead, it is also possible to add a new node $[s]_{\approx_E}$ to the graph with an edge from $[t]_{\approx_E}$ to $[s]_{\approx_E}$. If expressions are always correctly identified as equivalent, this approach leads to nodes with self-loops because $t \approx_E s$ and consequently $[t]_{\approx_E} = [s]_{\approx_E}$. However, if the algorithm that is used to decide $t \approx_E s$ fails for t and s, this approach allows the graph search algorithm to directly influence how different representations of expressions are explored. In Linnea, this approach is used for common subexpression elimination (Ch. 8)

and the application of those tricks that do not apply any kernels (Sec. 7.5); technically, both are cases of rewriting expressions. However, by extracting subexpressions into separate assignments, expressions are rewritten in a way such that t and s are not identified as equivalent. As a result, common subexpression elimination and the application of tricks can be implemented as generation steps. Since the extraction of subexpressions into separate assignments is not undone by the conversion to normal form, the changes that are applied by those two steps are preserved in the subgraph that originates at the node $[s]_{\approx_E}$.

4.7 CONCLUSION

While on a high level, the structure of Linnea is similar to that of traditional compilers, the application of optimizations is quite different. In traditional compilers, the optimizations are applied on an intermediate representation; both the input and the output of a given optimization is in the intermediate representation. The translation of the abstract syntax tree to the intermediate representation happens in a separate step. In Linnea, the majority of the optimizations take place in the generation steps, that is, during the translation from a symbolic expression, which can be seen as the abstract syntax tree, to a sequence of kernels (the intermediate representation).

Even though the application of optimizations works differently, just like other compilers, Linnea is affected by the phase ordering problem; the problem of ordering the optimizations in a way that results in the best code [127]. In Linnea, we address this problem with a search; the graph search explores different orderings of the generation steps (the optimizations). The cost function is used to guide this search towards good solutions. This approach is influenced by search-based approaches used in the field of artificial intelligence [111, Ch. 3].

5 APPLICATION OF KERNELS

The application of kernels is a central part of the algorithm generation in Linnea. The problem statement is rather simple: Given a sequence of assignments and a set of patterns that describe operations that can be computed with kernels, the task is to find subexpressions in the assignments that can be computed with the available kernels. As an example, consider application problem a.6:

$$
\begin{aligned}
X_{10} &:= L_{10}L_{00}^{-1} \\
X_{20} &:= L_{20} + L_{22}^{-1}L_{21}L_{11}^{-1}L_{10} \\
X_{11} &:= L_{11}^{-1} \\
X_{21} &:= -L_{22}^{-1}L_{21}
\end{aligned}
$$

For the purpose of this example, we only look at one operation which can be computed with the TRSM kernel: BA^{-1}, where A is lower triangular. Pattern matching as provided by MatchPy [83] is used to find all subexpressions that can be computed by this operation. In this case, there are two matches; $L_{10}L_{00}^{-1}$ and $L_{21}L_{11}^{-1}$. For each match, a kernel call is generated; for $L_{10}L_{00}^{-1}$, the kernel call is $M_1 \leftarrow L_{10}L_{00}$, where M_1 is the intermediate operand that represents the result of this operation. In addition, in the original sequence of assignments, the subexpression $L_{10}L_{00}^{-1}$ is replaced with M_1, originating a new node in the search graph. This new node containing the assignments

$$
\begin{aligned}
X_{10} &:= M_1 \\
X_{20} &:= L_{20} + L_{22}^{-1}L_{21}L_{11}^{-1}L_{10} \\
X_{11} &:= L_{11}^{-1} \\
X_{21} &:= -L_{22}^{-1}L_{21}.
\end{aligned}
$$

The edge from the node containing the original assignments to this new node is annotated with $M_1 \leftarrow L_{10}L_{00}$. The same is done for the subexpression $L_{21}L_{11}^{-1}$. At this point, the graph contains three nodes and two edges.

With the exception of Sec. 5.1, in the remainder of this chapter, a number of aspects are discussed that are relevant for the application of kernels either to decrease the generation time or to increase the quality of the generated algorithms. The application of factorizations is discussed in Sec. 5.7. Since factorizations are in several ways different from other kernels and thus require a separate treatment, for the remainder of this chapter, when we refer to *kernels*, this explicitly excludes factorizations.

5.1 REPRESENTATION OF KERNELS AS PATTERNS

Most BLAS and LAPACK kernels can compute relatively complex operations. For instance, the GEMM kernel can compute the operation $\alpha\,\mathrm{op}(A)\,\mathrm{op}(B) + \beta C$, where op is either the identity function or transposition. Intuitively, it is clear that such complex kernels can also be used to compute simpler operations. For example, by setting $\alpha = 1$ and $\beta = 0$, and choosing the identity function for both occurrences of op, this kernel computes the much simpler operation AB. However, in MatchPy, the different operations that can be computed with the GEMM kernel cannot be described in a single pattern: MatchPy does not support optional operations as described with op, and it is not able to identify that a product $M_1 M_2$ can be computed with the operation $\alpha AB + \beta C$ if $\alpha = 1$ and $\beta = 0$.[1] In order to solve this problem, from the description of kernels (see App. b), one pattern is generated for every operation that can be computed with a given kernel.[2] As an example, for the GEMM kernel, 24 different patterns are generated, for all possible combinations of 1) the identity function or transposition in place of op, 2) optionally setting α to 1, and 3) optionally setting β to 0 or 1. For kernels such as TRSM which have input operands that can have different properties, separate patterns are generated for each property. Since MatchPy uses many-to-one matching algorithms, the large number of patterns does not significantly affect the cost of pattern matching.

The patterns that represent kernels have the constraint that variables can only match a single operand. For the pattern XY of the GEMM kernel, $\sigma = \{X \mapsto A, Y \mapsto B\}$ is for example a valid match, while $\sigma = \{X \mapsto A, Y \mapsto B^T + C\}$ is not. This constraint ensures that an expression, in this case $B^T + C$, is not used as input to a kernel before it has been computed. As a side effect of this constraint, a pattern such as XY is not sufficient to find all subexpressions that match this operation. For instance, there is no match for XY in the product ABC, even thought both AB and BC can be computed with this operation. The reason is that since X and Y only match single operands, the remaining operands C and A, respectively, are not matched by any variable. As a result, a pattern that should also match AB and BC in ABC needs additional variables. To solve this problem, instead of XY one can use the pattern $t = c_1^* XY c_2^*$, where c_1^* and c_2^* are sequence variables that match an arbitrary number of operands (including zero).

1 Pattern matching in Mathematica supports default values, as well patterns involving alternatives which could be used to describe operations containing op [110]. Those features are not supported yet by MatchPy.

2 Those patterns are the patterns t_f of the kernel functions defined in Def. 4.5.

We refer to those variables as *context variables*. There are two matches of t in ABC:

$$\sigma_1 = \{c_1^* \mapsto (), X \mapsto A, Y \mapsto B, c_2^* \mapsto (C)\}$$
$$\sigma_2 = \{c_1^* \mapsto (A), X \mapsto B, Y \mapsto C, c_2^* \mapsto ()\}$$

In commutative operations such as addition, one context variable is sufficient to match the remaining operands. As an example, consider the AXPY kernel that computes $\alpha X + Y$. In the expression $A + 2B + C + D$, there are three matches for the pattern $\alpha X + Y + c^*$ with the sequence variable c^*:

$$\sigma_1 = \{\alpha \mapsto 2, X \mapsto B, Y \mapsto A, c^* \mapsto (C, D)\}$$
$$\sigma_2 = \{\alpha \mapsto 2, X \mapsto B, Y \mapsto C, c^* \mapsto (A, D)\}$$
$$\sigma_3 = \{\alpha \mapsto 2, X \mapsto B, Y \mapsto D, c^* \mapsto (A, C)\}$$

Since in some cases, patterns without context variables are sufficient to find all subexpressions that can be computed with a kernel,[3] for every kernel one pattern with context variables and one without is generated.

5.2 AVOIDING DIAMONDS

Except for very simple linear algebra problems, the number of applicable kernels is typically very large and contributes significantly to the size of the search graph. In addition, in larger problems, there are usually many kernels that can be applied independently of one another. For instance, in the assignment $X_{20} := L_{20} + L_{22}^{-1} L_{21} L_{11}^{-1} L_{10}$, since the subexpressions $L_{22}^{-1} L_{21}$ and $L_{11}^{-1} L_{10}$ do not overlap, kernels that compute those subexpressions can be applied independently of one another and in any order. This phenomenon is the primary cause of diamonds in the search graph (see Sec. 4.2.4). Consequently, the number of diamonds can be reduced by limiting the number of kernels that can be applied independently of one another. This is achieved by applying kernels only to the right-hand side of one assignment at a time. Specifically, only the first non-terminal assignment is used. The underlying idea is that kernels which are applied in different assignments are always independent of one another, which means that they cause diamonds. In addition, this approach guarantees that output operands are never used in subsequent computations before they are computed. As an example, in the expression

$$X := AB$$
$$Y := X + C,$$

3 Context variables are not necessary for the generalized matrix chain algorithm (Sec. 5.6.2). The reason is that in this algorithm, all patterns and subjects are products of two operands.

if a kernel is applied to $X + C$ before AB, the resulting sequence of kernels

$$Y \leftarrow X + C$$
$$X \leftarrow AB$$

does not correctly compute the input expression. Such incorrect sequences are avoided by limiting the application of kernels to the first non-terminal assignment.

5.3 EXPLICIT TRANSPOSITION AND INVERSION

In order to ensure that Linnea can generate code for every possible input expression, including ones such as $X := A^T$ or $X := L^{-1}$, kernels for the explicit transposition and inversion of a matrix are necessary.[4] However, if it is avoidable, those kernels should not be used: Instead of explicitly computing the transpose of a matrix, if possible this transposition should be performed with a kernel that allows for transposed input arguments, such as GEMM or TRSM. The reason is that with the combined transposition, an intermediate operand is avoided and the transposition is performed by changing how the operand is accessed. Explicit inversion should be avoided not only because it is slow, but also because it is numerically less stable than solving a linear system [65, Sec. 13.1].

While there are few cases where explicit transposition or inversion is necessary to find a solution, the respective kernels can be applied whenever an expression contains the transposition or inversion operator. As a result, without leading to better solutions, those kernels have the potential to significantly increase the size of the search space. For this reason, kernels for the explicit transposition and inversion are only applied if no other kernels can be applied to the right-hand side of a given assignment. This approach ensures that those kernels are only applied if they are necessary to find a solution.

5.4 TRANSPOSED KERNELS

Many BLAS and LAPACK kernels allow some input arguments to be transposed; however, this is usually not the case for all arguments. For instance, the TRSM kernel can compute the operations $\alpha\,\mathrm{op}(A^{-1})B$ and $\alpha B\,\mathrm{op}(A^{-1})$, where A is triangular and the function op is either the identity function or transposition. Thus, while A can optionally be transposed, this is not possible for B. As a result, by itself this kernel is not sufficient to compute the expression $L^{-1}M_1^T$, where L is lower

4 Since the explicit inversion of most matrices is computed by means of a factorization, kernels for the explicit inversion of diagonal and triangular matrices are sufficient for computing every input expression (see also Sec. 5.7).

triangular. There are two alternatives: The first one is to compute $X_1 \leftarrow M_1^T$ and then apply the TRSM kernel to $L^{-1}X_1$:

$$\begin{aligned} X_1 &\leftarrow M_1^T \\ X &\leftarrow L^{-1}X_1 \end{aligned}$$

The second alternative is to use the TRSM kernel to compute $X_2 \leftarrow M_1 L^{-T}$, which is the transpose of $L^{-1}M_1^T$. In the following, this is called the application of a *transposed kernel*. To obtain the result of $L^{-1}M_1^T$, it is still necessary to transpose X_2:

$$\begin{aligned} X_2 &\leftarrow M_1 L^{-T} \\ X &\leftarrow X_2^T \end{aligned}$$

There are however situations where the application of a transposed kernel allows to avoid an explicit transposition, because the transposition can be combined with another operation. One such case is the expression $L^{-1}M_1^T M_2$, where M_2 has more columns than rows. Because of the shape of M_2, the optimal parenthesization for this expression is $(L^{-1}M_1^T)M_2$. As in the previous example, with this parenthesization there are two alternatives: The first alternative is to explicitly transpose M_1^T, followed by a one call to TRSM and GEMM, respectively. The sequence of kernels is the following:

$$\begin{aligned} Y_1 &\leftarrow M_1^T \\ Y_2 &\leftarrow L^{-1}Y_1 \\ Y &\leftarrow Y_2 M_2 \end{aligned}$$

In the second alternative, the explicit transposition is avoided by applying the transposed TRSM kernel, and making use of the fact that GEMM supports transposed input arguments:

$$\begin{aligned} Y_3 &\leftarrow M_1 L^{-T} \\ Y &\leftarrow Y_3^T M_2 \end{aligned}$$

In Linnea, the application of transposed kernels is supported as follows: In the description of kernels, it is possible to specify that for a given kernel, in addition to the original, unmodified patterns, transposed patterns have to be generated (see App. b.1). As an example, the original and transposed patterns for the TRSM kernels are shown in Tab. 5.1. During the application of kernels, the transposed patterns are used in the same way as the original ones, with the exception that whenever a match is found, the subexpression that is computed is replaced with a transposed intermediate operand. For instance, in $L^{-1}M_1^T M_2$, when $L^{-1}M_1^T$ is computed with the kernel call $Y_3 \leftarrow M_1 L^{-T}$, the subexpression $L^{-1}M_1^T$ is replaced with Y_3^T, resulting in $Y_3^T M_2$.

Table 5.1: Original and transposed patterns for the TRSM kernel (ignoring α).

Pattern		Parameters	
Original	Transposed	`side`	`transA`
$A^{-1}B$	B^TA^{-T}	`L`	`N`
$A^{-T}B$	B^TA^{-1}	`L`	`T`
BA^{-1}	$A^{-T}B^T$	`R`	`N`
BA^{-T}	$A^{-1}B^T$	`R`	`T`

Alternatively, the same effect as with the application of transposed kernels could also be achieved by pushing up the transposition operator, similar to how the inversion operator is pushed up when rewriting expressions (see Sec. 7.1.4). For instance, in the example above, the expression $L^{-1}M_1^TM_2$ is effectively computed as rewritten to $(M_1L^{-T})^TM_2$. The former expression can be rewritten into the latter by first introducing the transposition for L, and then pushing up the transposition of L and M_1:

$$L^{-1}M_1^TM_2 = \left(L^{-T}\right)^T M_1^TM_2 = \left(M_1L^{-T}\right)^T M_2$$

The disadvantage of this approach is that it requires to introduce the transposition operator for operands that were previously not transposed. Doing this systematically would significantly increase the number of possible alternative representations of expressions. For this reason, in Linnea the application of transposed kernels is achieved through the generation of transposed patterns as described above, which is more targeted than pushing up the transposition.

5.5 SELECTION OF OPTIMAL KERNELS BASED ON PROPERTIES

Since BLAS and LAPACK offer specialized kernels for matrices with certain properties, many expressions can be computed with several different kernels. In Linnea, this is exacerbated by the additional code snippets for operations not supported by BLAS and LAPACK. The problem is illustrated well by the product DL, where both matrices are square, D is diagonal, and L is lower triangular. This product can be computed by the following kernels:

1. GEMM; D and L are treated as full matrices.
2. TRMM with `side` = `R` and `uplo` = `L`; D is treated as a full matrix.
3. TRMM with `side` = `L` and `uplo` = `L`; L is treated as a full matrix, D as a lower triangular matrix.

4. TRMM with `side = L` and `uplo = U`; L is treated as a full matrix, D as an upper triangular matrix.

5. SYMM with `side = L`; L is treated as a full matrix, D as a symmetric matrix.

6. The code snippet for the product of a diagonal and a full matrix; L is treated as a full matrix.

Intuitively, it is clear that such an expression should not be computed with the overly general GEMM kernel, but instead with a kernel that is more specific to the properties of D and L. This intuition is based on the observation that kernels which make use of more specific properties usually perform fewer FLOPs. The TRMM kernel for instance performs half the number of FLOPs of a GEMM. As a result, specialized kernels are usually faster than more general ones.

In Linnea, we can make use of this intuition to pre-select kernels purely based on properties, without the evaluation of the cost function. Instead of generating a new successor for every kernel that computes a given subexpression, we only generates successors for those kernels that make use of the most specific properties. This approach has several advantages: It reduces the number of successors per node, which in turn decreases the size of the search graph and speeds up the algorithm generation. In addition, the number of suboptimal algorithms is reduced. When the current cost function is replaced with one that is more expensive to evaluate, the pre-selection based on properties allows to reduce the number of expensive evaluations of the cost function. Finally, once Linnea will be extended to support variable operand sizes, this approach will make it possible to compare different kernels that might be difficult to compare otherwise.

The idea is to make use of the partial order on properties, which is shown in Fig. 6.1, that follows from the special-case relationship of properties: For each operation, a tuple of sets of properties, in the following called *property tuple*, is constructed. Each set in a property tuple corresponds to the constraints of one argument.[5] As an example, for the TRSM kernel that computes $A^{-1}B$, where A is lower triangular, the property tuple is $(\{\text{lower triangular}, \text{square}\}, \varnothing)$. For the GEMM kernel that computes $\alpha A^T B$, the property tuple consists of three empty sets: $(\varnothing, \varnothing, \varnothing)$.[6] In order to determine which property tuple represents the

5 For the property tuple, only those arguments of a kernel are considered that appear in the operation that is computed. As an example, consider two operations that can be computed with the GEMM kernel: The property tuple for the operation $\alpha AB + \beta C$ has five elements, while the tuple for the operation AB (`alpha = 1` and `beta = 0`) has only two.

6 Constraints regarding the type of an argument, that is, whether it is a matrix, vector, or scalar, are not included in the property sets because they are already used as a special kind of constraint on the variables. Specifically, those constraints do not check properties as discussed in Ch. 6, but instead whether a matched expression object is an instance of the `Matrix`, `Vector`, or `Scalar` class.

most specific kernel, the partial order on properties is first extended to sets of properties, and then to tuples of sets of properties. The kernels that make use of the most specific properties can then be identified by the tuples that are minimal according to the partial order.

Given a subset S of a partially ordered set, a minimal element *$s \in S$ is an element that is not greater than any other element in S. A subset S can have multiple minimal elements.*

During the application of kernels, the property tuples are used to pre-select kernels as follows: The many-to-one matching algorithms in MatchPy allow to return matched patterns in groups; a group consists of all structurally identical patterns that only differ in the constraints. For instance, one group contains all matches for patterns of the form AB, regardless of whether they belong to GEMM, TRMM, SYMM, or a code snippet for the multiplication of a diagonal and a full matrix. Among all matches in a group, considered are only the ones where the property tuples are minimal elements of the set of property tuples for those matches. Intuitively, a kernel is only used if there is no other kernel that is more specific. If there are multiple minimal property tuples, the cost function is used to select the cheapest kernel.

5.5.1 *Extension of Partial Order to Property Tuples*

For the extensions of the partial order on properties to sets of properties, it is first necessary to define how sets of properties are represented. Due to the relationships between different properties, most properties and combinations of properties can be represented by several different sets. As an example, since diagonal is a special case of both upper triangular and lower triangular, and a matrix that is both upper and lower triangular has to be diagonal, the following sets all represent the same mathematical property:

{diagonal, lower triangular, upper triangular}

{lower triangular, upper triangular}

{diagonal, upper triangular}

{diagonal, lower triangular}

{diagonal}

In order to allow for a concise definition of the partial order on sets of properties, it is useful to require that sets of properties are represented in a canonical form. There are two natural choices for such a canonical representation: A minimal representation that does not contain any redundant properties, and a maximal representation that contains all redundant properties. In the example above, the first set is maximal, and the last set is minimal. Since the property sets are derived from the property constraints in the description of kernels, and the minimal representation is usually more natural for humans, we decided for the minimal representation.

A property set S_1 is considered more specific than a property set S_2 if S_1 contains more specific properties than S_2 and/or if S_1 contains additional properties that are not contained in S_2.

Definition 5.1 (Property Set Order)**.** Let S_1, S_2 be property sets in minimal canonical form. S_1 is more specific than or equal to S_2 ($S_1 \leqslant S_2$) if for every $p_2 \in S_2$, there is a $p_1 \in S_1$ with $p_1 \leqslant p_2$.

Example 5.1. Let

$$S_1 = \{\text{symmetric}, \text{diagonal}\}$$
$$S_2 = \{\text{square}, \text{lower triangular}\}$$
$$S_3 = \{\text{SPD}, \text{diagonal}\}$$
$$S_4 = \{\text{SPD}\}$$

It holds that $S_1 \leqslant S_2$ because symmetric is more specific than square, and diagonal is more specific than lower triangular. $S_3 \leqslant S_4$ since the only property in S_4, SPD, is also in S_3. S_1 and S_4 are not comparable: $S_1 \leqslant S_4$ does not hold because neither symmetric nor diagonal is more specific than SPD, nor does $S_4 \leqslant S_1$ because SPD is not more specific than diagonal. This is consistent with the observation that there are symmetric, diagonal matrices that are not SPD, and SPD matrices that are not diagonal.

The partial order on sets of properties is extended to property tuples with a product order, that is, a tuple T_1 is more specific than a tuple T_2 if every element in T_1 is more specific than or equal to the element at the same position in T_2.

Definition 5.2 (Property Tuple Order)**.** Let $T_1 = (S_1, \ldots, S_n)$ and $T_2 = (R_1, \ldots, R_n)$ be tuples of property sets. $T_1 \leqslant T_2$ if for all $i = 1, \ldots, n$, $S_i \leqslant R_i$.

Example 5.2. Let DL be the expressions from the initial example, where both matrices are square, D is diagonal, and L is lower triangular. For this expression, the property tuples of all matching kernels are the following:

1. GEMM: $(\varnothing, \varnothing)$.
2. TRMM: $(\varnothing, \{\text{lower triangular}, \text{square}\})$.
3. TRMM: $(\{\text{lower triangular}, \text{square}\}, \varnothing)$.
4. TRMM: $(\{\text{upper triangular}, \text{square}\}, \varnothing)$.
5. SYMM: $(\{\text{symmetric}\}, \varnothing)$.
6. Code snippet: $(\{\text{diagonal}, \text{symmetric}\}, \varnothing)$.[7]

The partial order on those tuples is shown in Fig. 5.1. The minimal elements are $\{\{\text{diagonal}, \text{symmetric}\}, \varnothing\}$ and $\{\varnothing, \{\text{lower triangular}, \text{square}\}\}$,

7 This is one of the cases where the minimal representation of property sets is somewhat unintuitive: The code snippet requires that the diagonal matrix has to be square. However, since a square, diagonal matrix is also symmetric, and the property symmetric is a special case of square, the minimal representation contains symmetric instead of square.

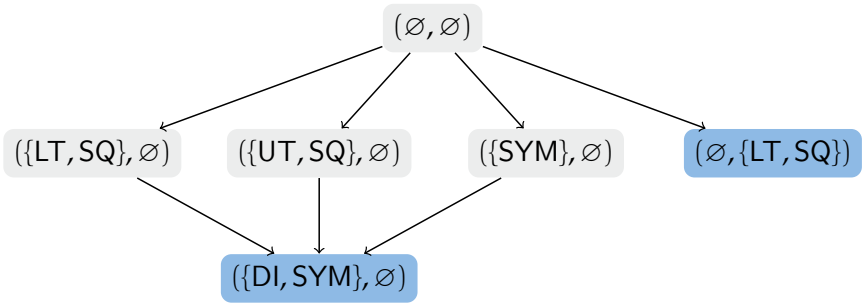


Figure 5.1: Partial order of the property tuples of all kernels matching the expression DL. Tuples towards the top are more general, tuples towards the bottom more specific. Minimal elements are highlighted in blue.

which respectively belong the code snippet for the multiplication of a diagonal and a full matrix, and the TRMM kernel with `side` = `R` and `uplo` = `L`.

5.6 CONSTRUCTIVE ALGORITHMS

While the application of kernels with pattern matching as described so far is sufficient to find good solutions, it has the disadvantage of exploring the potentially very large search space almost exhaustively. For specific types of subexpressions, however, relatively good solutions can be found without an exhaustive search. As an example, in expressions with high computational intensity, different parenthesizations in sums of matrices do not significantly affect performance. For products of multiple matrices, on the other hand, different parenthesizations can make a large difference, but there is no need to exhaustively generate all of them. Instead, efficient algorithms exist that find the optimal solution in terms of FLOPs to this so called matrix chain problem in polynomial [49] and log-linear time [68, 69].

For those subexpressions, to find a first solution quickly, and to increase the chances that this solution is relatively good, Linnea uses specialized algorithms. In order to distinguish those algorithms from the exhaustive application of kernels, we refer to them as *constructive algorithms*. For sums, we developed a simple greedy algorithm. For products, we developed a generalized version of the matrix chain algorithm [7], which finds the optimal parenthesization for matrix chains containing transposed and inverted matrices and considers matrix properties.

5.6.1 *Constructive Algorithm for Sums of Matrices*

For this algorithm, as input we allow sums where all arguments are matrices or vectors that can optionally be transposed and/or

multiplied with a single scalar. For instance, the sum $A + \alpha B + C^T + \beta D^T$ is a valid input for this algorithm. In general, kernels for the following three operations would be sufficient to compute those sums: $A + B$, A^T, and αA. However, since those sums are memory bound, is it beneficial to also use kernels that combine those operations, such as $A + \alpha B$ and $A + B^T$.

For this algorithm, three sets of kernels are used: *Addition kernels* compute a sum of two terms. Those terms can optionally be transposed and/or multiplied with a scalar. *Transposition kernels* compute operations of the form A^T. *Scaling kernels* compute operations of the form αA. All operands can have constraints regarding properties; for this reason, there are multiple transposition and scaling kernels.

The algorithm performs three steps:

1. Pattern matching is used to to apply the first matching addition kernel.
2. If in the previous step no matches were found, pattern matching is used to apply the first matching transposition kernel.
3. If in the previous step no matches were found, pattern matching is used to apply the first matching scaling kernel.

Those three steps are repeated until a sequence of kernels is found for the entire sum.

Since addition kernels are applied first, and all other kernels are only used if there are no matching addition kernels, kernels that combine multiple operations are prioritized. Conversely, transposition and scaling kernels are only used if they are necessary: There is for example no kernel that computes $\alpha A + \beta B$. As a result, for such a sum, it is necessary to first use a scaling kernel to either compute αA or βB. However, only one term needs to be computed with a scaling kernel; the remainder is computed with the AXPY kernel that allows for one scaled term.

COMPLEXITY In order to determine the complexity of this algorithm, it is necessary to first determine the complexity of the three steps in which the different sets of kernels are applied:

1. In MatchPy, pattern matching in commutative operations is done in two steps [82]. In the first step, for every term in the input operation, syntactic many-to-one pattern matching is used to find matching terms of the patterns. The complexity of syntactic many-to-one matching does not depend on the number of patterns, but only on the size of the patterns [82], that is, the number of nodes in the expression trees of the patterns, which is fixed. As a result, the cost of the first step is $\mathcal{O}(n)$, where n is the number of terms in the input sum. In the second step, a

bipartite graph is constructed. The nodes of this graph are the terms of the input sum, as well as the matching terms of the patterns. The Hopcroft-Karp algorithm is then used to construct a bipartite graph matching. The complexity of finding a first graph matching is $\mathcal{O}(|V|^{2.5})$, where $|V|$ is the number of nodes [67]. As mentioned above, the number of nodes depends both on the number of terms in the input sum, as well as on the number of patterns. For the purpose of this algorithm, the latter is constant. Thus, the complexity is $\mathcal{O}(n^{2.5})$.

2. Syntactic many-to-one pattern matching is used to find matches of the transposition kernels. Again, the cost of pattern matching is constant. In the worst case, it is performed $\mathcal{O}(n)$ times.

3. Scaling kernels are applied in the same way as transposition kernels. Thus, the complexity of this step is again $\mathcal{O}(n)$.

In order to compute a sum with n terms, $n-1$ binary addition kernels are necessary; consequently, the three steps described above are repeated $n-1$ times. As a result, the overall complexity of the constructive algorithm for sums of matrices is $\mathcal{O}(n^{3.5})$.

5.6.2 *Generalized Matrix Chain Algorithm*

The content of this section has been published in [7]. It was edited to avoid repetitions and ensure consistent terminology and notation.

Given a product of matrices $M := A_1 A_2 \cdots A_n$ where all matrices are full, also called a *matrix chain*, the optimal parenthesization can be computed efficiently with different algorithms (see Sec. 5.6.2.1). However, in practice such matrix chains are rare. Instead, in most matrix products, some matrices are transposed and or inverted, and they frequently have properties. For such chains, finding an optimal parenthesization is not sufficient to generate a sequence of kernels; it is necessary to also select kernels to compute the operations that appear. The selection of kernels is especially relevant because, unlike in the original matrix chain problem, kernels that make use of different properties have different costs, even if the operand sizes are the same. We refer to the problem of finding an optimal sequence of kernels for matrix chains that contain transposed and inverted operands as well as properties as the *Generalized Matrix Chain Problem* (GMCP). In order to quickly find good sequences of kernels for such products, we developed the Generalized Matrix Chain (GMC) algorithm. This algorithm builds on the original matrix chain algorithm, but extends it to support transposed and inverted operands, as well as to make use of matrix properties.

As input, the GMC algorithms accepts matrix chains of the form $\mathcal{M} = t_0 \cdots t_{n-1}$, where t_i is a matrix or a vector that can be transposed and/or inverted. We use $\mathcal{M}_{[ij]}$ to denote the product $t_i \cdots t_j$, which we also call *sub-chain*. If $i = j$, the chain consists of one single matrix

and is denoted by $\mathcal{M}_{[i]}$. Since Linnea does not use kernels that solve general linear systems[8] (see Sec. 5.7), the GMC algorithm cannot be applied to matrix chains that contain inverted matrices unless they are triangular or diagonal. This requirement is specific to Linnea and not a limitation of the GMC algorithm: It is not necessary if solvers for general linear systems are used; this case is discussed in [7].

5.6.2.1 *Related Work*

The matrix chain problem is subject to a lot of research. The classic algorithm to solve the matrix chain problem uses dynamic programming and has $\mathcal{O}(n^3)$ complexity, where n is the length of the chain [49]. The best known algorithm, by Hu and Shing, exploits the equivalence between the matrix chain problem and the triangulation of polygons to achieve $\mathcal{O}(n \log(n))$ complexity [68, 69]. 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) [18, 109, 124], while others find an ordering that is optimal when the matrix chain is evaluated on a parallel system [88]. Nishida et al. present a version for GPUs [101]. Additionally, both sequential [21] and parallel [26] 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 and Julia, as well as libraries such as Eigen and Armadillo allow to directly express instances of GMCP, without explicit parenthesization. However, with the exception of Armadillo, such product are always evaluated from left to right [108].[9] Armadillo instead uses a simplified algorithm to solve the matrix chain problem: For a chain $ABCD$, the parenthesization $(ABC)D$ 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 $n \leqslant 4$. 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

8 Since Linnea directly applies factorizations to inverted matrices with other properties, this is not a limitation.

9 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 $y \in \mathbb{R}^{1 \times n}$.

```
1  for l ∈ {1, ..., n − 1}:
2      for i ∈ {0, ..., n − l − 1}:
3          j := i + l
4          for k ∈ {i, ..., j − 1}:
5              c := 2 · sizes[i] · sizes[k + 1] · sizes[j + 1]
6              cost := costs[i][k] + costs[k + 1][j] + c
7              if cost < costs[i][j]:
8                  costs[i][j] := cost
9                  solution[i][j] := k
```

Figure 5.2: Pseudocode of the matrix chain algorithm.

of the previous one. As an example, consider $A((BC)D)$, which results in the following sequence of kernels:

$$M_1 := BC$$
$$M_2 := M_1 D$$
$$M_3 := A M_2$$

5.6.2.2 *The Standard Matrix Chain Algorithm*

The matrix chain problem can be elegantly solved with a dynamic programming approach, both in a top-down and a bottom-up fashion [25]. Here, we briefly explain the bottom-up version, as it is the foundation for the algorithm presented in this section.

We use 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 $\text{cost}(\mathcal{M}_{[1,3]})$ 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. 5.2. The following arrays are used, where solution and costs have size $n \times n$, sizes is of size $n + 1$:

```
for l ∈ {1, ..., n − 1}:
    for i ∈ {0, ..., n − l − 1}:
        j := i + l
        for k ∈ {i, ..., j − 1}:
            expr := tmps[i][k] · tmps[k + 1][j]
            kernel := match(expr)
            cost := costs[i][k] + costs[k + 1][j] + kernel.cost
            if cost < costs[i][j]:
                tmps[i][j] := create_tmp(expr)
                tmps[i][j].properties := infer_properties(expr)
                kernels[i][j] := kernel
                costs[i][j] := cost
                solution[i][j] := k
```

Figure 5.3: Pseudocode of the GMC algorithm.

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 [25].

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 [25].

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 $\mathcal{M}_{[i-1]}$.

5.6.2.3 *Extension to 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 parenthesization, it also has to identify which kernels can be applied and when. To solve this problem, the GMC algorithm works on symbolic expressions. The pseudocode of the algorithm is shown in Fig. 5.3. Instead of the one-dimensional array sizes, we now use the $n \times n$ array tmps, which is used to store intermediate operands representing sub-chains. In the following, consider the chain $\mathcal{M} = A^{-1}BC^T$ as an example. tmps[i][j] contains the intermediate that represents $\mathcal{M}_{[i,j]}$. The entry tmps[i][i] is initialized with the matrix $\mathcal{M}_{[i]}$. For example tmps[0][0] is A^{-1}. When the algorithm terminates,

tmps[1][2] contains an intermediate M_{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] · tmps[1][2] $= A^{-1}M_{12}$, which corresponds to the parenthesization $A^{-1}(BC^T)$. New intermediates 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 again on pattern matching as offered by MatchPy (line 6). The subset of kernels that is used in the GMC algorithm is constructed automatically from the set of all kernels available in Linnea by selecting those kernels that compute operations of the form $f_1(A) \cdot f_2(B)$, with f being the transposition, inversion, or the combination of both, and operands A and B. If more than one kernel matches the target expression, the algorithm described in Sec. 5.5 is used to select the best one.

Since the selection of kernels in the GMC algorithm uses the same techniques as the remainder of Linnea, namely symbolic expressions and pattern matching, the algorithm inherits some of Linnea's features. Specifically, the algorithm makes use of matrix properties (line 10; see also Ch. 6) and supports all cost functions that satisfy the requirements described in Sec. 4.5.

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 standard matrix chain 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 intermediate tmps[i][j]. In the end, the kernels are combined to the solution sequence (see Sec. 5.6.2.5).

COMPLEXITY 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 $f_1(A) \cdot f_2(B)$. 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 $\mathcal{O}(1)$.

create_tmp This function creates a symbolic intermediate 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 $b \in \mathbb{R}^m$, an intermediate

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 $\mathcal{O}(1)$.

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

The loop body is executed $\mathcal{O}(n^3)$ times, where n is the length of the matrix chain (see [25]). Thus, the complexity of the entire algorithm is $\mathcal{O}(n^3 + n^3 p)$. This could be further reduced to $\mathcal{O}(n^3 + n^2 p)$ by inferring properties outside of the k loop only for the intermediate 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 either because no suitable kernel is found, or because the operation is not allowed (see Sec. 4.4). Let us assume we are given the matrix chain $X := A^{-1}B^{-1}C$, and no kernel is available that computes $X^{-1}Y^{-1}$, so $A^{-1}B^{-1}$ can not be computed. In this case, the cost of computing $A^{-1}B^{-1}$ is considered to be ∞. 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.

5.6.2.4 *Additional Extensions*

In Linnea, the GMC algorithm is extended with some additional features which are described in the following. As explained there, two of those extensions may prevent the algorithm from finding the optimal solution. In Linnea, we use those extensions because their benefits outweigh their shortcomings. In addition, since the GMC algorithm is only used to find a first solution quickly, optimality is not crucial.

DECOMPOSITION OF EXPRESSIONS The underlying idea of the GMC algorithm that compositions of binary and unary operators on two operands can be seen as an extended set of binary operators can be taken even further: If there is no single kernel to compute a given operation $\mathcal{M}_{[0,k]}\mathcal{M}_{[k+1,n-1]}$, we use a much simplified version of the search graph used in Linnea to decompose those expressions into a

sequence of up to two unary operations and one binary operation. With this extension, the kernels array may contain sequences of kernels. While this extension does make the algorithm more expensive, it does not affect its asymptotic complexity: The graphs have at most five nodes, and pattern matching, which as discussed above has a constant cost, is performed only once per node.

SCALARS While matrix multiplications is not commutative, scalars in matrix products do commute. This includes subexpressions that form scalars, for example inner products such as $x^T y$ and $x^T A y$. It is easy to see that the placement even of a single scalar in a matrix chain can affect the optimal solution: Consider the chain $\alpha M D$, with $M, D \in \mathbb{R}^{n \times n}$, where D is diagonal. Both parenthesizations $\alpha(MD)$ and $(\alpha M)D$ have the same cost of $2n^2$ FLOPs; n^2 FLOPs for the product of a full and a diagonal matrix, and another n^2 FLOPs for the multiplication of a full matrix with a scalar. By making use of the commutativity of α and rewriting this chain to $M \alpha D$, it can be computed with $n^2 + n$ FLOPs: n FLOPs for the multiplication of a diagonal matrix with a scalar, and again n^2 FLOPs for the product of a full and a diagonal matrix. In practice, whenever possible, operations with scalars should be performed as part of compute bound operations. However, since the resulting operations are ternary operations, as for example GEMM that computes αAB, or TRSM that computes $\alpha A^{-1} B$, they are not supported by the GMC algorithm. Even though the solution might be suboptimal, the GMC algorithm can be applied to matrix chains that contain scalars as long as there are binary kernels for operations with scalars, for example αA.

TRANSPOSED KERNELS It is possible to use transposed kernels (see Sec. 5.4) in the GMC algorithm. If a transposed kernel is used, a transposed intermediate operand is stored in the tmps array. Theoretically, the use of transposed kernels can cause the algorithm to produce a suboptimal solution. The reason is that the algorithm does not consider the cost of computing the full chain if the last kernel (the kernel stored in kernels[0][n]) is a transposed kernel; the cost of the remaining transposition is not counted. Depending on the cost function, the cost of computing the full chain, that is, including the final transposition, might be higher than the cost of a different sequence that does not end with a transposed kernel. Accounting for the cost of the final transposition in the algorithm is not possible because it is not know if this transposition will be computed as a single operation, or instead as part of another kernel. If the transposition is computed as part of another kernel, it usually does not negatively affect the solution for the full expression.

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

Figure 5.4: Function to construct the solution. yield and yield from behave as the corresponding Python keywords.

5.6.2.5 *Construction of the Solution*

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 shown in Fig. 5.4. The complexity of this function is $\mathcal{O}(n)$. 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.

5.6.3 *Limitations*

As a tradeoff for finding solutions quickly, without the exploration of a large search space, the constructive algorithms come with some limitations. The first one is that both algorithms can only use a limited set of kernels; neither algorithm can make use of kernels that combine addition and matrix multiplication, such as the GEMM kernel that computes $AB + C$, or SYR2K. This is not a limitation for Linnea since those kernels are used during the exhaustive application of kernels through pattern matching. The second limitation is that those algorithms do not make use of common subexpressions. However, during the construction of the search graph, the constructive algorithms are applied both to expressions that still contain common subexpressions, as well as to expressions where common subexpressions were eliminated.

5.7 FACTORIZATIONS

Some parts of this section have been published in [8] and [9].

In contrast to other languages and libraries, in the input, Linnea does not distinguish between the explicit inversion of a matrix and the solution of a linear system. Whenever possible, inversion is avoided in favor of a linear system; matrices are explicitly inverted only if this is unavoidable, for example in expressions such as $A^{-1} + B$. Even though LAPACK offers kernels that encapsulate a factorization followed by a linear system solve (e.g., GESV), Linnea ignores those kernels and applies factorizations directly. For instance, the expression $X := A^{-1}B$ is not computed with a single operation

$X \leftarrow A^{-1}B$. Instead, the LU factorization is applied to A, resulting in $X := (P^TLU)^{-1}B$. This expression is then simplified to $X := U^{-1}L^{-1}PB$ and computed from right to left. The motivation for the application of factorizations is that it might enable other optimizations which are not possible when using a 'black box' kernel such as GESV. As an example, consider the generalized least squares problem $b := (X^TM^{-1}X)^{-1}X^TM^{-1}y$ (example problem a.2). This problem can be computed efficiently by applying the Cholesky factorization to M, resulting in $b := (X^TL^{-1}L^{-T}X)^{-1}X^TL^{-1}L^{-T}y$. In this expression, the subexpression X^TL^{-1} or its transpose $L^{-T}X$ appears three times and only needs to be computed once. If either X^TM^{-1} or $M^{-1}X$ were computed with a single kernel, this redundancy would not be exposed and exploited. Furthermore, the use of the Cholesky factorization allows to maintain the symmetry of $X^TM^{-1}X$.

Linnea uses the following factorizations: Cholesky, LU, QR, symmetric eigenvalue decomposition and singular value decomposition. LDL^T is currently not supported, because with the current LAPACK interface, it is not possible to separately access L and D; they can only be used in kernels to directly solve linear systems or invert matrices.

5.7.1 *Application*

The application of factorizations introduce several challenges that do not exist with other kernels. Kernels which compute more or less complicated operations can only be applied to matching subexpressions. Since the factorizations used in Linnea only take one matrix as input, they can be applied to every matrix in an expression that has the required properties. As a result, the number of factorizations that can be applied to a given expression is usually much larger than the number of factorizations that can be expected to lead to a good solution. As an example, consider the expression $X := S^{-1}A$, where S is SPD, and A is square and has full rank. Since S is SPD, all five factorizations used in Linnea can be applied to it; three factorizations can be applied to A. Considering that A does not have to be factored to find a solution for this expression, there are $5 \cdot 4 = 20$ different combinations of factorizations that can be applied. Clearly, most of those combinations do not lead to a good solution for this expression. There is for example no reason to factor A at all, and there is no reason to apply the more expensive LU factorization to S if the Cholesky factorization can be used instead.

If an operand is factored, it is often advisable to apply factorizations to all occurrences of that operand. For instance, in case of the least squares problem $b := (X^TX)^{-1}X^Ty$, the numerical most stable solution is obtained by applying the QR factorization to all three occurrences of X. There are exceptions, however: In the expression $A + A^{-1}$, where A is a full matrix, it is necessary to factor the matrix A on the right, but

detrimental to factor it on the left. This leads to the question of which occurrences of an operand should be factored if it appears multiple times.

In order to address those challenges, we define a number of rules for the application of factorizations that aim to capture the intuition of a human expert. The goal is to find a good balance between the exhaustive application of factorizations, and the targeted application in response to the structure of an expression. While the targeted application significantly reduces the size of the search space, there is the danger that it prevents finding unexpected solutions that were not considered in the design of the rules. For this reason, the rules are relatively general; rather than enforcing a behavior that leads to known, good solutions, they mostly prevent cases that are likely to lead to suboptimal solutions.

Those rules are:

1. Matrices are only factored if they are not triangular, diagonal or orthogonal. The reason is that linear systems and explicit inversions with triangular and diagonal matrices can be computed directly. For orthogonal matrices, the inversion is replaced with transposition (see Sec. 7.1.1), thus there is no need to solve linear systems with or compute the explicit inversion of orthogonal matrices.

2. Matrices that are defined by a previous assignment (instead of being a known input operand) can only be factored once a sequence of kernels has been generated for the right-hand side of this assignment, that is, once this right-hand side consists of a single symbol. An example of such a matrix is X in

   $$X := AB$$
   $$Y := X^{-1}C.$$

   The reason is that otherwise, in the sequence of kernels those matrices are factored before they have been computed.

3. If there are multiple occurrences of a matrix within an expression and this matrix is factored, the same factorization is applied to all occurrences that are factored. As an example, $b := (X^T X)^{-1} X^T y$, it is not allowed that the QR factorization is applied to one occurrence of X, while the singular value decomposition is applied to another. The reason is that multiple factorizations make an algorithm more expensive, so it is less likely to be a good algorithm. In addition, if the same factorization is used for all occurrences, some of the factors may cancel out.

4. Matrices that appear directly inside an inverse (A in A^{-1}) are always factored. Matrices that appear inside inverted subexpression (A and B in $(AB)^{-1}$) are factored, but the case where they

are not factored is also considered. The intuition is that in the first case, a factorization is necessary to find a solution, while in the second case, it is not.

5. If a matrix is factored, all of its occurrences are factored, including those that are not located within an inverse. In order to account for cases such as $A + A^{-1}$, an exception is made for occurrences that are not located within an inverse and are unlikely to interact with other occurrences that are located within inverses. Specifically, this includes terms in a sum where the term does not contain any occurrences of the matrix within an inverse, and assignments where the right-hand side does not contain any occurrences of the matrix within an inverse.

6. Some factorizations rule out others: If the Cholesky factorization can be applied to a matrix, the LU factorization is not applied. If the symmetric eigenvalue decomposition can be applied, the singular value decomposition is not applied.

Factorizations are applied in all combinations that satisfy the rules above.

5.7.1.1 *Implementation*

The implementation of the rules consists of three parts: 1) The selection of sets of matrices that are factored at the same time, 2) the selection of the occurrences of those matrices in the expression to which a factorization is applied, and 3) the actual application of factorizations.

SELECTION OF MATRICES In a first step, with the exception of matrices that have not been computed yet (rule 2), all those matrices are collected that 1) appear within an inverse operator, and 2) have properties that make a factorization necessary (rule 1). In the following, the set that contains those matrices is called O. For instance, in problem a.8,

$$\delta X := \left(B^{-1} + H^T R^{-1} H\right)^{-1} H^T R^{-1} \left(Y - H X^b\right)$$

O is $\{B, H, R\}$. The set O is then divided into two disjoint subsets, F and F_{opt}: The set F contains all matrices that appear at least once directly inside an inverse; those matrices have to be factored. The set F_{opt} contains the remaining matrices $O \setminus F$ which may optionally be factored (rule 4). In case of example problem a.8, those sets are $F = \{B, R\}$ and $F_{opt} = \{H\}$. Factorizations are applied to all sets of matrices $F \cup F'_{opt}$ with $F'_{opt} \subseteq F_{opt}$.

SELECTION OF OCCURRENCES In addition, the occurrences of all matrices in O are divided into groups of occurrences that are unlikely to interact with one another. Specifically, occurrences are placed in

different groups if they are located in different assignments, and if they are located in the same assignment, but in different terms of a sum (rule 5). As an example, consider the expression

$$X := A_1^T B A_2 + A_3^{-1}$$
$$Y := A_4^{-1} B.$$

The subscripts are used to distinguish the occurrences of A. Those occurrences are divided into three groups: $\{1, 2\}$, $\{3\}$, and $\{4\}$. Occurrence 4 is in its own group because it is located in a different assignment than all other occurrences. The remaining occurrences are divided into the groups $\{1, 2\}$ and $\{3\}$ because they appear in different terms of a sum. All groups that do not contain any occurrence within an inverse are discarded and not considered for the application of factorizations. In case of the expression above, the group $\{1, 2\}$ is discarded; factorizations are only applied to the occurrences 3 and 4.

APPLICATION If a matrix is factored, the same factorization is applied to all remaining occurrences (rule 3). If multiple matrices are factored at the same time, new expressions are generated for all possible combinations of factorizations that can be applied to the different matrices. Per matrix that is factored, only one invocation of the factorization kernel is generated, even if multiple occurrences of the matrix are factored. This can be seen as a form of common subexpression elimination.

SUCCESSOR GENERATION ORDER The order in which factorizations are applied is determined as follows: The algorithms starts with the application of factorizations only to those matrices that need to be factored. If multiple factorizations can be applied to a given matrix, they are applied in the order of increasing cost. Thus, the order is:

1. Cholesky or LU factorization.
2. QR factorization.
3. Symmetric eigenvalue or singular value decomposition.

6 MATRIX PROPERTIES

Linnea's input language makes it possible to annotate matrices with properties. However, for the generation of algorithms, not only is it important to know the properties of the input matrices, it is at least equally important to know the properties of expressions that contain those operands, as well as the properties of intermediate operands as the computation unfolds. Properties are important both to select the best kernel for a given operation, as well as to rewrite expressions. Consider for instance the generalized least squares problem $b := (X^T M^{-1} X)^{-1} X^T M^{-1} y$. Since X has full rank and more rows than columns, and M is symmetric positive definite, it can be inferred that $X^T M^{-1} X$ is symmetric positive definite, irrespective of how it is computed. This knowledge then allows one to solve the linear system $(\dots)^{-1} X^T M^{-1} y$ with a Cholesky factorization, as opposed to the more expensive LU factorization.

In order to be able to make use of properties, Linnea needs to be able to infer the properties of an expression from the properties provided by the user. This inference is implemented with sets of inference rules that describe how properties of an operand imply other properties, and how properties propagate through expressions. In contrast to most languages and libraries for linear algebra, the inference is a purely symbolic process that does not rely on inspecting the operands at run-time. In this chapter, we describe the inference of properties in Linnea in detail.

DEFINITIONS In the following, we formally describe properties in terms of logical predicates that are either true or false for a given operand or expression. For instance, if the expression AB is lower triangular, the predicate LowerTriangular(AB) is true. The definitions of all properties currently supported as input by Linnea are shown in Tab. 6.1. In addition, there are some properties that are only used internally; they are shown in Fig. 6.2. They are not part of the language because they can always be inferred from the input, either from other properties, or the operand sizes. How the size of an expression is computed is described in Sec. 6.4. Expressions can have multiple properties as long as their mathematical definitions do not contradict one another. A square matrix can be for example both SPD and diagonal.

Table 6.1: Definitions of properties supported by Linnea. Let $A \in \mathbb{R}^{m \times n}$ be a real matrix. The elements of this matrix are denoted as a_{ij} with $i \in \{0, \ldots, m-1\}$ and $j \in \{0, \ldots, n-1\}$. For scalars, α is used.

Property	Definition
diagonal	$a_{ij} = 0$ if $i \neq j$
lower triangular	$a_{ij} = 0$ if $i < j$
upper triangular	$a_{ij} = 0$ if $i > j$
unit diagonal	$a_{ij} = 1$ if $i = j$ (requires upper or lower triangular)
symmetric	$m = n$ and $a_{ij} = a_{ji}$
SPD	$m = n$ and $x^T Ax > 0$ for all $x \in \mathbb{R}^n$ with $x \neq 0$
SPSD	$m = n$ and $x^T Ax \geqslant 0$ for all $x \in \mathbb{R}^n$ with $x \neq 0$
orthogonal	$m = n$ and $A^T A = AA^T = I$
orthogonal columns	$m \geqslant n$ and $A^T A = I$
orthogonal rows	$n \geqslant m$ and $AA^T = I$
permutation	$m = n$ and there is exactly one 1 in each row and column, all other entries are 0
identity	$a_{ij} = 1$ for $i = j$ and $a_{ij} = 0$ for $i \neq j$
zero	$a_{ij} = 0$ for all i, j
positive	$\alpha > 0$ (only for scalars)
full rank	A has full rank
non-singular	$m = n$ and A has full rank

Table 6.2: Definitions of properties that are only used internally. Let m and n be the number of rows and columns of an expression, respectively.

Property	Definition
square	$m = n \neq 1$
column panel	$m > n$ and $n \neq 1$
row panel	$m < n$ and $m \neq 1$
matrix	$m > 1$ and $n > 1$
vector	$m = 1$ and $n > 1$, or $m > 1$ and $n = 1$
scalar	$m = n = 1$
triangular	matrix is upper or lower triangular
constant	operand is constant; including the identity and zero matrix, as well as constant scalars

6.1 INFERENCE

In the simplest case, we are concerned with the following problem: Given a linear algebra expression where the properties of the operands that appear in this expression are known, infer some or all of the properties of the entire expression.

As an example, one instance of this problem is the subexpression $\lambda_1 I_l + W_1^T A A^T W_1$ in example problem a.17, where it is known that the matrices A and W_K both have more rows than columns and full rank, I_l is the identity matrix, and the scalar λ is positive. The question now is if the expression is for example SPD.

To solve this problem, we use inference rules that describe the relationships between different properties. In this chapter, we describe those inference rules as first-order logic formulas. Inference rules are universally quantified over all variables; the domain of discourse is the set of all expressions $T(\Sigma, X)$. In order to simplify the presentation, we omit the quantifiers. The rule $\forall A(\text{SPD}(A) \rightarrow \text{Symmetric}(A))$ is for instance written as $\text{SPD}(A) \rightarrow \text{Symmetric}(A)$.

The inference problem stated above can be broken down into two subproblems; the inference of the properties of a single operand, and the inference of the properties of an expression.

6.1.1 *Properties of Operands*

Operands that are annotated with properties usually posses additional properties that can be inferred from the known ones. As an example, since a permutation matrix is a special case of an orthogonal matrix, all permutation matrices are also orthogonal. As a second example, a matrix that is both diagonal and square is also symmetric. From the perspective of the user, if they already specified that a matrix is a permutation matrix, the information that it is also orthogonal is redundant. For this reason, we do not expect the user to annotate operands with such redundant properties. However, those properties are still needed internally, for example when expressions are simplified: When Linnea attempts to simplify Q^{-1} to Q^T, it is only tested if Q is orthogonal, not if it is a permutation matrix. From a mathematical perspective, testing that Q is orthogonal is sufficient; it is assumed that Q is correctly identified as orthogonal even if the user did not explicitly specify this property.[1] As a result, Linnea needs to be able to infer properties of an operand that follow from other known properties.

We distinguish between two types of inference rules that are relevant for this problem:

1. Inference rules of the form $P(A) \rightarrow Q(A)$, where both $P(A)$ and $Q(A)$ are predicates over the same operand, as for example

1 This is a design decision in Linnea. The intention is to make the use of properties throughout Linnea as simple and intuitive as possible.

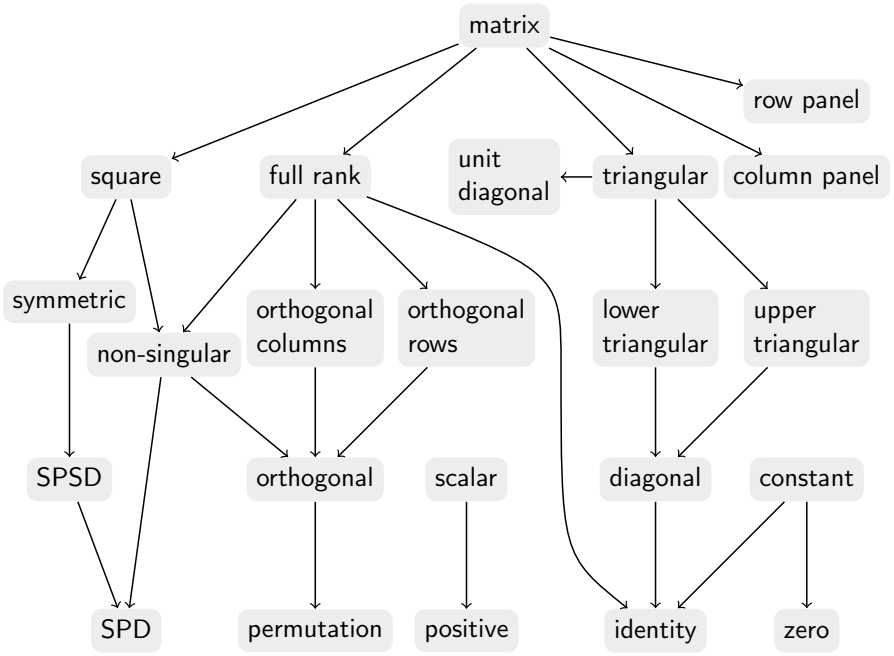


Figure 6.1: Partial order of matrix properties. More general properties are placed towards the top, more specific properties towards the bottom.

$\mathrm{SPD}(A) \rightarrow \mathrm{Symmetric}(A)$. Those rules follow from the definitions of the properties and encode that a property is a special case of another. Those rules can also be represented by a partial order on all properties. A graphical representation of this partial order is shown in Fig. 6.1.

2. Inference rules of the form $\bigwedge_i \mathrm{P}_i(A) \rightarrow \mathrm{Q}(A)$, where $\bigwedge_i \mathrm{P}_i(A)$ is a conjunction of two or more predicates. An example of such a rule is $\mathrm{Diagonal}(A) \wedge \mathrm{Square}(A) \rightarrow \mathrm{Symmetric}(A)$. Again, those rules follow from the definitions of properties, but in this case they describe how combinations of two or more properties imply another property.

In this thesis, properties are used both as an adjective ('M is square') and as a mathematical object, which is used as a noun ('M has the property square', or 'square ∈ P'). In order to distinguish the adjective from the mathematical object, a sans-serif font is used for the mathematical object: 'M has the property square'.

REPRESENTATION Before we can discuss how the properties of operands are inferred, we first need to describe how properties are represented. In Linnea, properties are represented by labels, and each operand has a set of such labels: If the label for a property is in the set, it means that the operand has the respective property. The reverse does not need to hold, though: An operand may has a property that is not in its property set. This can be the case if there exist properties that can be inferred from other properties in the set. As an example, let D be a matrix with the property set $\{\mathsf{diagonal}, \mathsf{square}\}$. While D is also symmetric, this property is not contained in the set.

6.1.1.1 *Implications* $P(A) \to Q(A)$

Implications of the form $P(A) \to Q(A)$ are incorporated by making use of the transitivity of the partial order defined by those rules. For each property, we precompute all properties that are implied by it. For instance, from the implications

$$\begin{aligned} \text{Identity}(A) &\to \text{Diagonal}(A) \\ \text{Diagonal}(A) &\to \text{LowerTriangular}(A) \\ \text{Diagonal}(A) &\to \text{UpperTriangular}(A), \end{aligned}$$

the rule

$$\begin{aligned} \text{Identity}(A) \to \text{Diagonal}(A) &\wedge \text{LowerTriangular}(A) \\ &\wedge \text{UpperTriangular}(A) \end{aligned}$$

is derived.[2] Whenever the property identity is added to the property set of an operand, for example because the user specifies it, all properties on the right-hand side of this rule are added to the set too. As a result, in the majority of cases, it is sufficient to test if an operand has a property by testing if the property is in its property set.

6.1.1.2 *Implications* $\bigwedge_i P_i(A) \to Q(A)$

Implications of the form $\bigwedge_i P_i(A) \to Q(A)$ are used in the backward direction: To determine if $Q(A)$ is true, it is checked if all $P_i(A)$ are true.

As an example, let us assume that there is an operand D which has the property set {diagonal, square}. To identify if D is symmetric, it is first checked if the property is contained in the set; this is not the case here. Thus, as a second step, we look at all rules with the right-hand side $\text{Symmetric}(A)$; one such rule is $\text{Diagonal}(A) \wedge \text{Square}(A) \to \text{Symmetric}(A)$. Now, it is checked if the left-hand side of this rule is true for D. Since D is both diagonal and square, it is inferred that D is symmetric. To cache the result of this inference, the property symmetric is then added to the property set of D. If necessary, additional rules of the form $\bigwedge_i P_i(A) \to Q(A)$ are evaluated recursively to check if the left-hand side of the initial rule is true.

6.1.2 *Properties of Expressions*

The inference of properties of expressions is not as structured as that of operands; properties and mathematical operations can interact in many different ways. Some examples follow:

1. The inverse of a lower triangular matrix is lower triangular.

2 In the interest of brevity, $\text{Triangular}(A)$ and $\text{Matrix}(A)$ were excluded from this example.

2. The transpose of a lower triangular matrix is upper triangular.
3. The product of two orthogonal matrices is orthogonal.
4. An expression A is symmetric if it is equal to its transpose, that is, if $A = A^T$ holds.
5. An expression A^TSA is SPD if A has full rank and at least as many rows as column, and S is SPD.

As a result, inference rules have the form $\psi \to P(t)$, where ψ is an arbitrary formula, and $P(t)$ is a predicate over an expression t. For the examples above, the rules are:

$$\begin{aligned}
\text{LowerTriangular}(A) &\to \text{LowerTriangular}\left(A^{-1}\right)\\
\text{LowerTriangular}(A) &\to \text{UpperTriangular}\left(A^T\right)\\
\text{Orthogonal}(A) \wedge \text{Orthogonal}(B) &\to \text{Orthogonal}(AB)\\
A = A^T &\to \text{Symmetric}(A)\\
\text{rows}(A) \geqslant \text{cols}(A) \wedge \text{FullRank}(A)& \\
\wedge \text{SPD}(S) &\to \text{SPD}\left(A^TSA\right)
\end{aligned}$$

$\text{rows}(A)$ and $\text{cols}(A)$ are functions that return the number of rows and columns of A, respectively. Those rules are again used in the backward direction, to traverse the expression tree from the root to the leaves. As an example, we use the expression $Q_1^TQ_2$, where both Q_1 and Q_2 are orthogonal, and we try to determine if $Q_1^TQ_2$ is orthogonal. The inference rules for this property are:

$$\text{Orthogonal}(A) \to \text{Orthogonal}\left(A^T\right) \tag{6.1}$$
$$\text{Orthogonal}(A) \to \text{Orthogonal}\left(A^{-1}\right) \tag{6.2}$$
$$\text{Orthogonal}(A) \to \text{Orthogonal}\left(A^{-T}\right) \tag{6.3}$$
$$\text{Orthogonal}(A) \wedge \text{Orthogonal}(B) \to \text{Orthogonal}(AB) \tag{6.4}$$

Since the root node of the expression tree of $Q_1^TQ_2$ is a product, rule (6.4) has to be applied with $A = Q_1^T$ and $B = Q_2$. Thus, it needs to be checked if $\text{Orthogonal}\left(Q_1^T\right)$ and $\text{Orthogonal}(Q_2)$ are true. $\text{Orthogonal}(Q_2)$ is true because Q_2 is known to be orthogonal; for $\text{Orthogonal}\left(Q_1^T\right)$, rule (6.1) has to be applied with $A = Q_1$. Since Q_1 is orthogonal, $\text{Orthogonal}(Q_1)$ is true as well. As a result, by determining that $\text{Orthogonal}\left(Q_1^TQ_2\right)$ is true, we have inferred that $Q_1^TQ_2$ is orthogonal.

In Linnea, the rules for the inference of properties of expressions are implemented as (in some cases mutually) recursive functions, one for each property. Similar to the example above, those functions traverse the expression tree from the root to the leaves.

For rules where the left-hand side ψ only consists of a conjunction of predicates, the implementation is straightforward. As an example,

```
1 def is_orthogonal(expr):
2     if isinstance(expr, Symbol):
3         return infer_property_symbol(expr, ORTHOGONAL)
4     if isinstance(expr, Times):
5         return all(map(is_orthogonal, expr.operands))
6     if isinstance(expr, (Transpose, Inverse,
          InverseTranspose)):
7         return is_orthogonal(expr.operand)
8     return False
```

Figure 6.2: Implementation of the function `is_orthogonal`.

the code of the function that infers the property orthogonal is shown in Fig. 6.2. If `expr` is a symbol, that is, a matrix, vector, or scalar, a function is called that implements the inference of properties of operands as described in Sec. 6.1.1 (lines 2 and 3). In lines 4 and 5, a rule for products with an arbitrary number of arguments is implemented; it is a generalization of rule (6.4) which follows from the associativity of matrix multiplication. The rules (6.1), (6.2), and (6.3) are implemented in lines 6 and 7. If `expr` is an addition, none of the rules are applicable and the function returns `False`.

For those rules where the left-hand side ψ is more complex, we implemented some auxiliary functions, for example one to test if an expression is the transpose of another, and one to determine if a product is SPD. The code of the latter is shown in Fig. 6.3. It implements the following rules:

$$\text{Positive}(\alpha) \wedge \text{SPD}(A) \rightarrow \text{SPD}(\alpha A) \tag{6.5}$$

$$\text{rows}(A) \geqslant \text{cols}(A) \wedge \text{FullRank}(A) \rightarrow \text{SPD}\left(A^{\mathsf{T}} A\right) \tag{6.6}$$

$$\text{rows}(A) \geqslant \text{cols}(A) \wedge \text{FullRank}(A) \wedge \text{SPD}(S) \rightarrow \text{SPD}\left(A^{\mathsf{T}} S A\right) \tag{6.7}$$

The function begins with separating scalars from matrices and vectors (line 2). The reason is that scalars commute in matrix products, and treating scalars separately simplifies the implementation of the remainder of this function. In line 3, it is checked if the product of all scalars is positive; if this is not the case, rule (6.5) cannot be satisfied and `False` is returned. The remainder of the function consists of a case distinction based on the number of remaining arguments in the product:

1. If there are no arguments left, `expr` only consists of scalars and hence cannot be SPD (lines 6 and 7).

2. If there is one argument, it is checked if this argument is SPD (lines 8 and 9).

3. If the number of arguments is divisible by two, the arguments are split into two parts of equal length. It is checked if the left

```
1  def is_SPD_product(expr):
2     scalars, non_scalars = expr.split_operands()
3     if scalars and not is_positive(Times(*scalars)):
4        return False
5     length = len(non_scalars)
6     if length == 0:
7        return False
8     elif length == 1:
9        return is_SPD(non_scalars[0])
10    elif length % 2 == 0:
11       left = Times(*non_scalars[:length//2])
12       right = Times(*non_scalars[length//2:])
13       return left.columns >= left.rows and is_full_rank(
             left) and left.transpose_of(right)
14    else:
15       left = Times(*non_scalars[:length//2])
16       right = Times(*non_scalars[length//2+1:])
17       middle = non_scalars[length//2]
18       return left.columns >= left.rows and is_full_rank(
             left) and is_SPD(middle) and left.transpose_of(
             right)
```

Figure 6.3: Implementation of the function `is_SPD_product`.

part has at least as many columns as rows, if it has full rank, and if it is equal to the transpose of the right part (lines 10–13). This corresponds to rule 6.6.

4. If the number of arguments is not divisible by two, the arguments are split into three parts: a left and right part of equal length, and a remaining argument in the middle. It is checked if the left part has at least as many columns as rows, if it has full rank, and if it is equal to the transpose of the right part. For the argument in the middle, it is checked if it is SPD (lines 14–18). This corresponds to rule (6.7).

6.2 INTERMEDIATE OPERANDS

Since expressions can contain intermediate operands, the actual inference of properties in Linnea is more complicated than the problem discussed so far.

The problem with intermediate operands can be illustrated well with the expressions ADA^T, where A is a full and D a diagonal matrix. Let us assume that a kernel is applied that computes $M_1 \leftarrow DA^T$, such that the expression becomes AM_1. Intuitively, it is clear

that AM_1 is symmetric, since ADA^T is symmetric. It is however not possible to infer this from AM_1: Mathematically, the properties of the intermediate M_1 are determined by the expression that was computed to obtain it; in this case DA^T. However, by itself, DA^T and consequently M_1 do not have any properties that allow to infer that AM_1 is symmetric.

For the application of kernels, this problem can be solved by making use of the table of intermediate operands (Sec. 4.2.1). Due to the constraint for the application of kernels that input arguments can only be single operands, not arbitrary expressions (see Def. 4.5), arguments of a kernel are either input operands to the program, or intermediate operands. For input operands of the program, the properties are known; for intermediate operands, the properties can be inferred from the equivalent expression stored in the table of intermediate operands. In case of ADA^T, this means that AM_1 as a whole can only be an argument of a kernel if it was already computed by another kernel; in that case, an intermediate operand was generated for it. Let M_2 be this intermediate operand. The equivalent expression of M_2 is ADA^T, from which it can be inferred that M_2 is symmetric.

In conclusion, for the application of kernels, the inference of properties of expressions that contain intermediate operands is avoided by inferring the properties of the resulting intermediate operand from its equivalent expression instead of the expression that is computed.

6.3 USER-SPECIFIED PROPERTIES OF EXPRESSIONS

The table of intermediate operands can also be used to annotate entire subexpressions (instead of individual operands) with properties that cannot be inferred from the subexpression.[3] As an example, consider application problem a.16,

$$B_k := \frac{k}{k-1} B_{k-1} \Big(I_n - A^T W_k \left((k-1) I_l + W_k^T A B_{k-1} A^T W_k \right)^{-1} W_k^T A B_{k-1} \Big).$$

Since B_{k-1} is SPD and k is larger than 1, the subexpression $(k-1)I_l + W_k^T A B_{k-1} A^T W_k$ is SPD as well. However, it is currently only possible to specify that k is positive, not that it is larger than 1. Thus, Linnea is not able to infer that the subexpression is SPD. For the application of kernels, this problem can be solved with the table of intermediate operands as follows: When the user specifies that $(k-1)I_l + W_k^T A B_{k-1} A^T W_k$ is SPD, before the algorithm generation starts, an entry for this expression is added to the table of intermediate operands. The intermediate operand that is created for this expression is annotated with the property SPD. Following again from Def. 4.5,

3 This feature is not supported in the input language yet.

the subexpression $(k-1)I_l + W_k^T A B_{k-1} A^T W_k$ can only be one of the input operands of a kernel once it has been fully computed; if this is the case, the subexpression has been replaced with its intermediate operand. Thus, if the subexpression is the input operand of a kernel, the property SPD is read off the intermediate, even though it cannot be inferred from the expression.

This approach has some limitations, though. In theory, it should be sufficient to specify that $k-1$ is positive, since this is the missing piece of information that prevents Linnea from inferring the property SPD. In practice, this does not solve the problem due to how the table of intermediate operands works: While the table would contain an entry for $k-1$ that is annotated with the property positive, this entry would not be considered when determining the properties of $(k-1)I_l + W_k^T A B_{k-1} A^T W_k$. To consider this entry, it would be necessary to infer the properties of the expression $sI_l + W_k^T A B_{k-1} A^T W_k$, where s is the intermediate operand for $k-1$ that is annotated with the property positive. However, this does not happen because the expressions stored in the table do not contain any intermediates; the intermediates are replaced with the expressions that they represent when the entry is created.

6.4 SIZE OF EXPRESSIONS

Some of the properties used in Linnea are based on the size of an expression, that is, the number of rows and/or columns. To infer those properties, Linnea contains functions to compute the size of expressions. Similar to the inference of properties, the size of an expression is computed by traversing the expression tree. With the exception of products, the rules are trivial:

1. The number of rows (columns) of a sum $M_1 + \ldots + M_i$ is the number of rows (columns) of M_1.
2. The number of rows (columns) of M^T is the number of columns (rows) of M.
3. The number of rows and columns of M^{-1} and M^{-T} is the number or rows of M.

For a product of matrices $M_1 \cdots M_i$, the number of rows of the entire expression is the number of rows of M_1, and the number of columns is the number of columns of M_i. However, care has to be taken for products that contain scalars or inner products. For example, in case of αAB, the number of rows is not determined by the left-most operand, which is a scalar, but instead by the matrix A. For the expression $v_1^T A v_2 B$, where v_1 and v_2 are vectors, the number of rows is equal to the number of rows of B.

```
1  @property
2  def rows(self):
3    n = 0
4    for op in self.operands:
5      if op.rows == 1:
6        if op.columns != 1:
7          # row vector
8          n += 1
9      else:
10       if n == 0:
11         # column vector or matrix
12         return op.rows
13       if op.columns == 1:
14         # column vector
15         n -= 1
16   return 1
```

Figure 6.4: Python code for the computation of the number of rows of a product, implemented as a method of the `Times` class.

The number of rows of a product that may contain scalars or inner products is computed with the algorithm in Fig. 6.4. The idea is to traverse the operands from left to right and keep track of whether the current operand is part of an inner product. The beginning of an inner product is marked by a row vector, the end by a column vector. The variable `n` is used to keep track of inner products; it is incremented whenever a row vector is encountered (line 8), and decremented when a column vector is encountered (line 15). Since instead of a boolean variable, an integer is used, the algorithm works even for arbitrarily nested inner products. If `n` is zero and the current operand is a column vector or matrix, the number of rows of that operand is the number of rows of the product (line 12). Since the loop body does not contain a case for scalars (that is, the number of rows and columns are one), they are simply ignored and do not affect `n`. If the entire product consists of scalars, line 16 is reached and one is returned. If the product starts with a row vector that is not part of an inner product and thus the number of rows of the product is one, `n` never becomes zero and one is returned in the last line. The number of columns is computed analogously by traversing the operands in reversed order and switching rows and columns.

6.5 CONCLUSION AND FUTURE WORK

While the inference of properties in Linnea is very advanced compared to other languages and libraries, it is still rudimentary compared to

the analysis a mathematician would carry out. In order to improve the inference of properties in Linnea, it is necessary both to introduce a more expressive language for the description of properties, as well as to extend the algorithms for the inference to be able to deal with a richer set of properties and inference rules.

At present, properties are limited to predicates over operands and expressions. In a more expressive language, properties could provide details that go beyond simple predicates, for example the possible range of values of a scalar instead of the property positive, or the range of eigenvalues instead of the property SPD. For matrices with block structure, a hierarchical description could be used to specify the properties of different parts of a matrix.

A disadvantage of the rule-based approach for the inference of properties is that the addition of new properties can require a large number of additional rules. As an example, consider the properties upper and lower bidiagonal, and tridiagonal. The product of two matrices is tridiagonal either if 1) one matrix is diagonal and one matrix is tridiagonal, or if 2) one matrix is upper and one matrix is lower bidiagonal. Since it does not matter which matrix is the left and which one is the right operand in the product, there are four cases that have to be checked. For products of more than two matrices, there are even more cases. At least for some properties, including bidiagonal and tridiagonal, the inference of properties can be implemented with an alternative approach that does not rely on rules for many different combinations of properties. This approach is based on the observation that a relatively large number of common properties can be described in terms of a generalized notion of bandwidth. In App. c, we discuss how this notion of bandwidth can be used for the inference of properties.

However, additional properties are only useful if there are kernels that allow to make use of them.

7

REWRITING EXPRESSIONS

Several of those identities are axioms of the algebra that is used in Linnea.

The ability to use algebraic identities to rewrite expressions is one of the main features that distinguish Linnea from other languages and libraries for linear algebra. Those identities are used for different purposes, and in different parts of the algorithm generation. The challenge with using algebraic identities to rewrite expressions is related to the fact that they can and in some cases need to be applied in both directions. For instance, when the Cholesky factorization is applied to the SPD linear system $S^{-1}A$, in the resulting expression $(L^TL)^{-1}B$, the inverse has to be distributed over L^TL before the expression can be computed (see also Sec. 4.4). This amounts to the application of the identity $(XY)^{-1} = Y^{-1}X^{-1}$, where X and Y have to be square, from left to right. On the other hand, for the expression $A^{-1}B^{-1}C$, where all matrices are square and full, the optimal sequence of kernels is obtained by making use of that same equality, but from right to left, to rewrite the expression to $(BA)^{-1}C$. In example problem a.11, the optimal solution for the expression $y_k := H^\dagger y + (I_n - H^\dagger H)x_k$ is found by applying the law of distributivity, in this case both $YX + ZX = (Y + Z)X$ and $XY + XZ = X(Y + Z)$, to rewrite the expression to $y_k := H^\dagger(y - Hx_k) + x_k$ (this example is discussed in detail on page 147).

Clearly, a brute-force application of all identities in both directions is impractical: Using both directions leads to infinite loops, and identities such as $X = (X^T)^T$ and $X = XI$, where I is the identity matrix, lead to arbitrarily large expressions that are not expected to result in good algorithms. To prevent loops and arbitrarily large expressions, in Linnea the use of those identities is structured into a set of functions that rewrite expressions in a targeted fashion. While the same identities can be used in multiple functions and in both directions, per function only one direction is used. Those functions are used to generate a number of representations of expressions that are explored systematically during the algorithm generation.

In this chapter, we first describe the different functions that are used to rewrite expressions in Sec. 7.1, followed by the representations that are generated with those functions in Sec. 7.2. In Sec. 7.3, we discuss why we do not use term rewriting systems to rewrite expressions. Two special cases of rewriting expressions are presented thereafter: Rewriting sequences of assignments (Sec. 7.4), and highly problem-specific rewritings we refer to as *tricks* (Sec. 7.5). Sec. 7.6 concludes this chapter. The elimination of common subexpressions, which can be seen as a special case of rewriting expressions, is discussed in Ch. 8.

7.1 REWRITE FUNCTIONS

The identities that are used in Linnea are organized into four functions: One to simplify expression (`simplify`, Sec. 7.1.1), two to apply the law of distributivity in different directions (`to_SOP`, Sec. 7.1.2, and `to_POS`, Sec. 7.1.3), and one that pushes up the inversion operator (`push_up_inv`, Sec. 7.1.4). Those functions are discussed in the following.

7.1.1 *Simplification*

Simplifications cover all algebraic identities that, in the intuitive sense of the word, make expressions simpler, that is, identities that can be used to obtain a more compact, concise representation of an expression. In many cases, simplifications help to avoid unnecessary computations. The simplifications are described below:

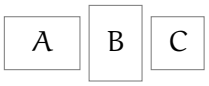
CONSTANT FOLDING Subexpressions consisting of constant scalars are evaluated and replaced with their value. Matrix products containing the zero matrix or the scalar 0 become the zero matrix, and scalar products containing 0 become 0. The commutativity of scalars in matrix products is considered, that is, the expression $2A3B$ is simplified to $6AB$. Neutral elements in all operations and types of expressions are removed. That includes the square identity matrix and the scalar 1 in matrix products, the zero matrix in matrix sums, 1 in scalar products, and 0 in scalar sums.

SYMBOLIC ADDITION OF TERMS Terms that appear multiple times in sums are added up, including the case when one or both of the terms are multiplied with a scalar. As examples, for an arbitrary expression A, $A + A$ becomes $2A$, $-1.5A + A$ becomes $-0.5A$, and $\alpha A + \beta A$ becomes $(\alpha + \beta)A$, where α and β are arbitrary scalar expressions.

CANCELING OUT INVERTED EXPRESSIONS In products, for arbitrary expressions A, subexpressions of the form $A^{-1}A$ and AA^{-1} are removed. If the product does not contain any other arguments, an identity matrix is inserted. In addition, A^TA is removed if A has orthogonal columns, and AA^T is removed if A has orthogonal rows.

DISTRIBUTING INVERSION AND TRANSPOSITION Both the inversion and transposition operators are pushed down as far as possible, that is, transposition is distributed over products and sums, and inversion is distributed over products whenever possible. The inversion operator can only be distributed over square matrices, or products

of non-square matrices that from a square matrix. As an example, consider the product



with $A \in \mathbb{R}^{n \times m}$, $B \in \mathbb{R}^{m \times n}$, $C \in \mathbb{R}^{n \times n}$, and $m > n$. While both A and B are not square, their product is, so distributing the inversion over this product results in $C^{-1}(AB)^{-1}$.

Nested transposition and inversion is removed. In addition, inversion and transposition is combined to a single inversion-transposition operator, that is, both $(A^{-1})^T$ and $(A^T)^{-1}$ become A^{-T}. Transposition is removed from symmetric matrices, and inversion is replaced with transposition in case of orthogonal matrices.

GROUPING SCALARS Scalars in matrix products are moved to the left. This is done because MatchPy does not support making use of the commutativity of scalars in matrix products. For instance, MatchPy does not find a match for a kernel that computes the scalar product $\gamma\delta$ in the product $\alpha A \beta$, but it does find a match if the product is rewritten to $\alpha\beta A$.

7.1.2 *Conversion to Sum of Products*

The conversion to a *sum of products* consists in in the application of the law of distributivity, that is, the identities $YX + ZX = (Y + Z)X$ and $XY + XZ = X(Y + Z)$, from right to left. Intuitively, all products that contain sums are multiplied out. For instance, $A(B + C^T)D$ becomes $ABD + AC^TD$. The only exception are scalar sums, that is, expressions such as $(\alpha + \beta)A$, where α and β are scalars and A is a matrix; such expressions are not rewritten to $\alpha A + \beta A$.

7.1.3 *Conversion to Product of Sums*

The conversion to a *product of sums* is the opposite of the conversion to a sum of products; the identities $YX + ZX = (Y + Z)X$ and $XY + XZ = X(Y + Z)$ are used from left to right. Intuitively, the conversion consists in factoring out common factors of terms in a sum. As a result, $ABD + AC^TD$ is rewritten to $A(B + C^T)D$. In contrast to the conversion to a sum of products, different results can be obtained depending on the order in which those identities are used, or equivalently depending on the order in which operands are factored out on the left- and right-hand side of a product. This is illustrated well by the expression $AC + AD + BC$. When operands are first factored out on the left-hand side, $A(C + D) + BC$ is obtained; starting on the right-hand side yields $(A + B)C + AD$. In both resulting expressions, no further operands can be factored out. In addition, in expressions with more than two terms

that have a factor in common, the resulting expression can depend on whether the common factor is factored out in all terms, or only in a subset of the terms.

Example 7.1. As an example, we use the sum[1]

$$AA + AB + AC + BA + BB.$$

Factoring out both A and B to the left in all terms results in

$$A(A + B + C) + B(A + B). \tag{7.1}$$

However, it is also possible to factor out A only in the terms AA and AB, resulting in

$$AC + A(A + B) + B(A + B).$$

At this point, $(A + B)$ can be factored out on the right, which yields

$$AC + (A + B)(A + B). \tag{7.2}$$

Both in (7.1) and (7.2), no further factors can be factored out. ∎

As a tradeoff between the exhaustive exploration of all possible product of sums and a reduction of the size of the search space, in Linnea we only consider two different product of sums representations: One is obtained by first factoring out all operands to the right side, and then to the left, the other one by first factoring out to the left, and then to the right.

If a term only consists of a single matrix that is factored out, that matrix is replaced with an identity matrix. As a result, the product of sums representation of $A + AB$ is $A(I + B)$. In the conversion to a product of sums, no inverted operands are introduced; while it would be possible to also factor out B in $A(I + B)$, resulting in $AB(B^{-1} + I)$, it seems unlikely that such an expression leads to good algorithms.

7.1.4 *Pushing up the Inversion*

As demonstrated at the beginning of this chapter, it is necessary to use the identity $(XY)^{-1} = Y^{-1}X^{-1}$ in both direction. The direction from left to right is already part of `simplify`. In `push_up_inv`, this identity is used from right to left to push up the inversion operator in products. It is applied whenever there are two or more adjacent arguments in a product that are inverted.

An exception is made when inverted operands are the result of a factorization. As an example, consider again the SPD linear system $S^{-1}B$. After the application of the Cholesky factorization, this expression is simplified to $L^{-T}L^{-1}B$. If the inversion was pushed up, one would again obtain $(LL^T)^{-1}B$, which cannot be computed (see Sec. 4.4). As a consequence, for such expressions there is no reason to push up the inversion in the first place.

1 Recall that matrix powers are not supported by Linnea yet.

```
1  def to_SOP(expr):
2    if isinstance(expr, Times) and any(isinstance(op, Plus)
         and not is_scalar(op) for op in expr.operands):
3      factors = []
4      for op in expr.operands:
5        if isinstance(op, Plus) and not is_scalar(op):
6          factors.append(op.operands)
7        else:
8          factors.append([op])
9      return Plus(*(to_SOP(Times(*product)) for product in
           itertools.product(*factors)))
10   elif isinstance(expr, Operator):
11     return type(expr)(*map(to_SOP, expr.operands))
12   else:
13     return expr
```

Figure 7.1: Python code of the function `to_SOP`.

7.1.5 *Implementation*

The basic idea for the implementation of all functions that rewrite expressions is the same: The expression tree is traversed recursively, either in the top-down or bottom-up direction; at every node, the necessary steps are carried out to rewrite the expression in the intended way.

As an example, the code for `to_SOP` is shown in Fig. 7.1. The expression tree is traversed recursively from the top to the bottom; only if the current node is a product and contains at least one argument that is a sum (line 2), the actual conversion to the sum of products takes place. In a first step, the arguments are collected in a nested list: for arguments that are sums, all summands are stored in one sublist, while all other arguments are put in their own lists. For instance, for the product $A(B + C^T)D$, the list $[[A], [B, C^T], [D]]$ is constructed (lines 3–8). Then, the `product` function from the Python standard library module `itertools` is used to construct the cartesian product of all sublists; for $[[A], [B, C^T], [D]]$, it is $[[A, B, D], [A, C^T, D]]$. This new list then directly corresponds to the product of sums representation of the input expression; the inner lists correspond to the products, the outer list corresponds to the sum. Thus, for $A(B + C^T)D$, the output is $ABC + AC^TD$. In line 9, this expression is constructed from the nested list. In addition, to continue the traversal of the expression tree, `to_SOP` is immediately applied to the new products. If the current node is any other operator, for example addition or transposition, `to_SOP` is applied to all of its arguments (lines 10–11). Since expressions in Linnea are immutable, a new expression has to be constructed. If the current

node is a matrix, vector or scalar, nothing needs to be done (lines 12–13). The condition `not is_scalar(op)` in lines 2 and 5 ensures that scalar sums are not considered for the conversion to a sum of products, that is, $(\alpha + \beta)A$ is not rewritten to $\alpha A + \beta A$.

The `simplify` function on the other hand traverses the expression tree from the bottom to the top. Different simplifications are carried out depending on the type of the current node: For instance, if the current node is a product, scalars are collected and grouped, while the matrices are scanned for subexpressions such as $A^{-1}A$. In order to push down transposition and inversion, auxiliary functions exist that transpose and/or invert an expression; if a subexpression such as $(A + B^T)^T$ is encountered, the `transpose` function is applied to $A + B^T$, resulting in $A^T + B$.

7.2 REPRESENTATIONS

The rewrite functions are used to obtain four different representations of expressions.

1. The normal form, which consists of fully simplified expressions represented as a sum of products. It is obtained by applying `simplify`, `to_SOP`, and again `simplify` to an arbitrary expression. The first application of `simplify` is necessary for example for the expression $(A + B)^T C$; in order to convert this expression to a sum of products, it is necessary to first distribute the transposition over the sum, resulting in $(A^T + B^T)C = A^T C + B^T C$. The second application is necessary for cases such as $AB + (A + C)B$; after the conversion to a sum of products, in the resulting expression $AB + AB + CB$, the two occurrences of AB can be added up to $2AB$. The normal form is discussed in more detail in Sec. 7.2.1.

2. An expression in the sum of products representation with inversion pushed up. It is obtained by applying `push_up_inv` to expressions in normal form.

3. Two different product of sums representations. They are obtained by applying `to_POS` to expressions in normal form.

A graph of how the rewrite functions are used to rewrite expressions into different representations is shown in Fig. 7.2. Clearly, those four representations are only a subset of all possible representations that can be obtained with the available rewrite functions. As an example, it would be possible to apply `push_up_inv` to both product of sums representations. The four representations were chosen as a tradeoff between the exhaustive exploration of a large number of representations and a faster algorithm generation. It should be noted that the different representations do not need to be distinct. For instance, in case of the expression $AB^T + C$, which is in normal form, neither the

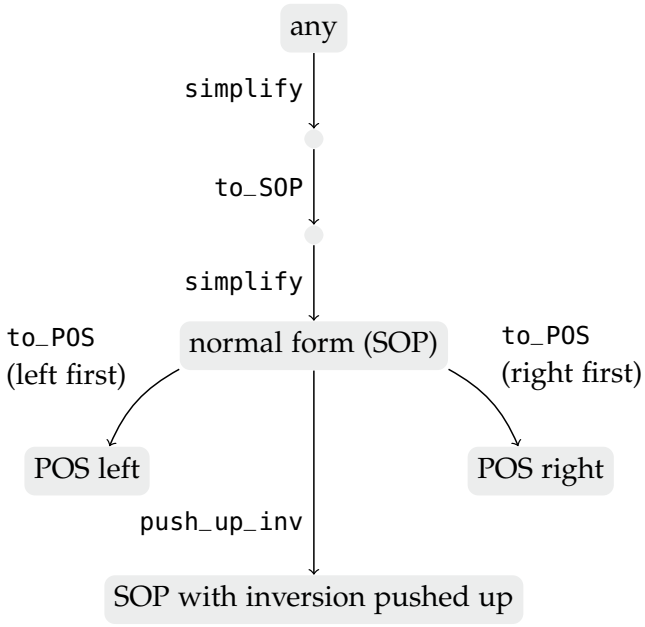


Figure 7.2: A visualization of how the the rewrite functions are used to rewrite expressions into different representations.

conversion to a product of sums, nor pushing up the inverse changes the expression. As a result, in this case all four representations are the same.

During the algorithm generation, the representations are explored systematically: By default, to ensure that equivalent nodes can be merged, expressions are stored in the normal form. Before the application of a generation step, the other representations are constructed on demand. Given sufficient time, every generation step is applied to every representation. The four representations are used in the following order:

1. Product of sums (left first),
2. product of sums (right first),
3. sum of products with inversion pushed up, and
4. normal form.

A representation is skipped if it is identical to one that was already used. The reason for beginning with the product of sums is that it reduces the number of expensive matrix-matrix multiplications: While $AB + AC$ requires two matrix-matrix multiplications, $A(B + C)$ requires only one. Pushing up the inversion operator allows to decrease the number of matrix factorizations, in favor of matrix-matrix multiplications: While $A^{-1}B^{-1}C$ requires two factorizations, $(BA)^{-1}C$ requires only one. After the application of a generation step, expressions are converted back to the normal form: `simplify` pushes the

inverse back down, and `to_SOP` converts the product of sums representation back to a sum of products.

7.2.1 *Normal Form*

As a normal form for linear algebra expression, the sum of products was chosen because it can easily be made unique by sorting arguments. Arguments in sums are sorted automatically in MatchPy expressions, and scalars in products are sorted by the `simplify` function. In contrast, as discussed in Sec. 7.1.3, many expressions have multiple product of sums representations. Since expressions are converted between different representations during the algorithm generation, the choice of the normal form does not influence the quality of the generated code.

Recall that as mentioned in Sec. 4.2, this normal form is not a true normal form in the sense that there is no guarantee that all algebraically equivalent expressions have the same normal form. Fortunately, this only has the effect that merging branches, which is a performance optimization, is less effective in those cases.

7.3 A COMMENT ON TERM REWRITING SYSTEMS

The application of algebraic identities to rewrite an expressions can naturally be described and implemented as a term rewriting system, that is, a set of rewrite rules (Def. 3.6). In Linnea, relevant rewrite rules could be $XI \to X$ or $(XY)^T \to Y^T X^T$, where X and Y match matrices and I is the identity matrix. Based on those examples, it should be clear that rewrite rules can naturally be constructed from algebraic identities. In this case, the rules originate from the identities $XI = X$ and $(XY)^T = Y^T X^T$. However, it is important to note that the identities by themself do not prescribe the direction of the rewrite rules; the rules $X \to XI$ and $Y^T X^T \to (XY)^T$ are equally valid. In addition, term rewriting systems do not prescribe in which order the rules should be applied.

There are both theoretical and practical limitations that make the use of term rewriting systems difficult. Those limitations are discussed in this section.

THEORETICAL LIMITATIONS Term rewriting systems are especially useful if they have two properties: *Termination* and *confluence*. A term rewriting system is said to be terminating if no infinite sequence of rewrite steps can exist, and it is confluent if, for a given initial expression, the same result is obtained no matter in which order the rules are applied. A term rewriting system with those two properties defines a normal form for expressions [5, Sec. 2.1]: All expressions that are algebraically equivalent are rewritten to the same normal form. As a

result, such a term rewriting system solves the *word problem*; the problem of identifying whether or not two expressions are algebraically equivalent. For many algebras, this problem is undecidable [5, Sec. 4.1]. While we do not have a proof that the word problem is undecidable for the algebra used in Linnea, given the complexity of the algebra, it seems likely that this is the case. If the word problem is undecidable for the algebra used in Linnea, there cannot exist a confluent and terminating term rewriting system that converts expressions to a normal form.

Another indication that such a term rewriting system may not exist is given by the Knuth-Bendix completion algorithm. Given a set of algebraic identities[2], the Knuth-Bendix completion algorithm can be used to construct a confluent and terminating term rewriting system [81].[3] Internally, this algorithm relies on a process called unification (a generalization of pattern matching, see [5, Sec. 10.1]). For operations that are associative but not commutative, such as matrix multiplication, unification is *infinitary*, which means that is has an infinite number of solutions [64]. As a result, the Knuth-Bendix completion algorithm constructs an infinite number of rewrite rules and does not terminate.

As mentioned before, in Linnea the normal form is only used as a performance optimization; it is not required for the correctness of the generated algorithms. Thus, one could attempt to manually construct a term rewriting system that may not be confluent: To benefit from the performance optimization, it would be sufficient if most (but not all) equivalent expressions are converted to the same normal form, as long as the term rewriting system is terminating. However, even constructing a term rewriting system that is only terminating can still be a problem; while there are several tools that attempt to determine if a given term rewriting system is terminating (for an overview, consider the participants of the Termination Competition [47]), this problem is again undecidable [5, Sec. 5.1.1].

PRACTICAL LIMITATIONS Irrespective of properties such as confluence and termination, in our experience it is difficult to manually construct a term rewriting system that reliably converts an expression to a desired representation. Due to the complexity of the algebra, there are many special cases that have to be considered. While the same special cases have to be considered in the rewrite functions, the advantage of a function that traverses the expression tree is that it is possible to make assumptions about the parts of the expression tree that were already traversed. Those assumptions allow to simplify the implementation. Rewrite rules that can be applied anywhere in an

2 In addition to the set of identities, completion requires a so called *reduction order*, that is, a total order on terms that prescribes the direction in which expressions are rewritten. Several different types of reduction orders exist [28]. Finding a suitable reduction order can be difficult.

3 Several extensions of this algorithm exist. For an overview, consider [30].

expression need to work irrespective of the state of the surrounding expression. As a result, a larger number of rules can be necessary to account for all cases. Finally, we observed that in comparison to dedicated functions, term rewriting systems are slow.

7.4 REWRITING SEQUENCES OF ASSIGNMENTS

In addition to rewriting expressions, there are cases where it is advantageous to rewrite sequences of assignments.

Example 7.2. Let us assume that in example problem a.2,

$$b := \left(X^T M^{-1} X\right)^{-1} X^T M^{-1} y,$$

the Cholesky factorization is applied to M. In the resulting expression, the common subexpression $X^T L^{-1}$ is detected and extracted into a separate assignment, which yields

$$\begin{aligned} M_1 &:= X^T L^{-1} \\ b &:= \left(M_1 M_1^T\right)^{-1} M_1 L^{-T} y. \end{aligned}$$

When $X^T L^{-1}$ is computed with a transposed kernel (Sec. 5.4), the first assignment becomes $M_1 := M_2^T$. By itself, this assignment introduces an explicit transposition. However, this explicit transposition can be avoided by replacing all occurrences of M_1 in the second assignment with M_2^T and removing $M_1 := M_2^T$. This step results in

$$b := \left(M_2^T M_2\right)^{-1} M_2^T L^{-T} y. \qquad \blacksquare$$

In addition to assignments of the form $X := Y^T$, it is also beneficial to remove assignments $X := X$ and $X := Y$, where X and Y are intermediate operands. In case of $X := Y$, this is again done by replacing all occurrences of X in the subsequent assignments with Y.[4] While those assignments do not introduce unnecessary operations, they can prevent expressions that are otherwise equivalent from being merged. All three types of assignments discussed here appear after a subexpression is placed in a separate assignment, either as part of common subexpression elimination (Ch. 8) or the application of tricks (Sec. 7.5). Assignments of the form $X := X$ appear once the expression that was placed in a separate assignment is fully computed. For instance, in the example above, $M_1 := M_2^T$ becomes $M_1 := M_1$ if the explicit transposition $M_1 \leftarrow M_2^T$ is computed. Assignments of the form $X := Y$ can arise in cases where the right-hand side does not have a unique

4 Assignments of the form $X := Y$ effectively rename an operand. If there is such an assignment, this means that there also is a kernel call $Y \leftarrow \ldots$ on the path leading to the current node. However, in the subsequent assignments, the same operand is denoted with X. While it would be possible to introduce $X := Y$ as a renaming of a variable in the generated code, we chose to substitute X in the subsequent assignments.

normal form and Linnea is unable to detect that X and Y represent the same expression.[5] In all cases, those assignments are immediately removed when they appear, which is usually after the application of kernels.

At present, the substitution and removal of assignments is only done when the left-hand side is an intermediate operand and limited to those three cases discussed here. The substitution of arbitrary assignments is discussed as part of future work (Sec. 7.6.1).

7.5 TRICKS

For some problems, a good solution can sometimes be found by rewriting an expression in a way that is not possible with the mechanisms described in this chapter. Those rewritings may require human intuition, or insights into linear algebra too deep to be easily found automatically. In Linnea, we refer to such rewritings as *tricks*. In addition, there are some rewritings that simply do not fit into any of the functions that rewrite expressions, or they are too specific to warrant their own function or representation.

Example 7.3. An example of how such a trick can be used to find a better algorithm is the computation the expression $X := A^T A + A^T B + B^T A$. A more complicated version of this expression appears in a material science application [29]. In its original form, two $\mathcal{O}(n^3)$ operations are required to compute the expression: SYRK and SYR2K. However, the expression can be rewritten such that the call to SYRK becomes unnecessary:

$$\begin{aligned} X &:= A^T A + A^T B + B^T A \\ &= \frac{1}{2}A^T A + A^T B + \frac{1}{2}A^T A + B^T A \\ &= A^T \left(\frac{1}{2}A + B\right) + \left(\frac{1}{2}A^T + B^T\right) A \\ &= A^T \left(\frac{1}{2}A + B\right) + \left(\frac{1}{2}A + B\right)^T A. \end{aligned}$$

At this point, the common subexpression $Y := \frac{1}{2}A + B$, which is an $\mathcal{O}(n^2)$ operation, only needs to be computed once. The remaining expression $X := A^T Y + Y^T A$ can be computed with a single call to the SYR2K kernel. ∎

In order to support tricks, as a proof of concept Linnea provides a framework that allows to implement almost arbitrary rewritings. This framework is designed to be as general and flexible as possible. It is independent of the functions that rewrite expressions and the different

5 Since assignments of the form $X := Y$ imply $X = Y$, those assignments could potentially be used to improve merging branches in the search graph in cases where Linnea is not able to detect that X and Y represent the same expression.

representations: Instead, it is used as a generation step that is applied to every representation. Within this framework, the implementation of a trick consists of two parts: 1) A pattern to detect when the trick can be applied, and 2) a callback function that takes the expression, the position where the match was found, and the substitution as input. The callback function then returns the modified expression and optionally a sequence of kernels. While tricks are usually only applicable in special cases, thanks to efficient many-to-one pattern matching, Linnea can identify such cases with only minimal impact on the overall performance. The order in which tricks are applied is arbitrary and depends on the order in which MatchPy finds matches for the different patterns.

There is only one requirement for tricks: In order to prevent loops and ensure that the tricks actually have an effect on the generated algorithms, they have to modify the expression such that the conversion to normal form does not undo the trick. This can usually be achieved either by the immediate application of kernels, or by extracting part of the expression into a new assignment. For instance, in case of $X := A^T A + A^T B + B^T A$, it is not sufficient to simply rewrite the expression as $X := A^T(\frac{1}{2}A + B) + (\frac{1}{2}A + B)^T A$; the conversion to normal form rewrites this expression back into its original representation. Instead, either one immediately generates a sequence of kernels that computes $\frac{1}{2}A + B$, or $\frac{1}{2}A + B$ is extracted into a new assignment, resulting in

$$
\begin{aligned}
Y &:= \frac{1}{2}A + B \\
X &:= A^T Y + Y^T A.
\end{aligned}
$$

Other possible applications of tricks include the introduction of ZZ^T in the Genome-Wide Association Studies [39, Sec. 3.1], factoring out A in order to reduce the bandwidth of the linear system in example problem a.5 [31, Sec. 4.2], and the application of Sherman-Morrison-Woodbury formulas [61].

As an additional advantage, the framework for tricks can be used to add new features, that would otherwise require to make significant changes to the structure of Linnea with relatively little effort. As a result, this framework can be used to conveniently test features, such as algorithms for the application of kernels or additional rewritings; if they prove to be beneficial, they can still be properly integrated into the structure of Linnea.

7.6 CONCLUSION AND FUTURE WORK

Through different means, Linnea is able to make use of the algebraic identities that are most relevant for the quality of the generated algorithms in both directions: Associativity and commutativity are covered

by 1) associative-commutative pattern matching (Sec. 3.3), which is used for the exhaustive application of kernels, and 2) the constructive algorithms for the application of kernels (Sec. 5.6). Distributivity is used when rewriting expressions between the sum of products and product of sums representations. The transposition and inversion operator is pushed down by `simplify`, and inversion is pushed up by `push_up_inv`. While transposition is not pushed up, the same effect is achieved in a more targeted fashion through the application of transposed kernels (see Sec. 5.4). Even though many identities are used in both directions, the organization of those identities into rewrite functions prevents loops and the generation of arbitrarily large expressions. Additional ways of rewriting expressions can be implemented as tricks.

It is important to note that the use of algebraic identities affects the numerical stability of the generated algorithms. Ideally, in order to consider the stability for the selection of the optimal solution, it should be part of the cost function. Unfortunately, as discussed in Sec. 4.5, incorporating the numerical stability into the cost function is a difficult problem.

7.6.1 *Future Work*

There are several ways in which Linnea's ability to rewrite expressions can be further improved in order to generate better algorithms. Two examples follow.

At present, Linnea does not maintain the original representation of the input problem as provided by the user in case it is different from the four representations that are used internally. One such case is application problem a.15:

$$X_{k+1} := S\left(S^T A S\right)^{-1} S^T + \left(I_n - S\left(S^T A S\right)^{-1} S^T A\right) X_k \left(I_n - A S\left(S^T A S\right)^{-1} S^T\right)$$

The problem is that this particular product of sums representation is different from the two product of sums representations that are considered by Linnea. However, this representation is the one that leads to the best solution. While it would be possible to increase the number of product of sums representations that are explored, a better approach with a smaller impact on the size of the search space might be to make sure that the original representation is used in the algorithm generation. The challenge with maintaining the original representation of the input expression is that it is not clear how to enable merging branches in this case, because merging requires to convert expression to the normal form.

Maintaining the original representation allows the user to influence which representations are used during the algorithm generation. At

present, this can only be achieved by placing subexpressions into separate assignments. In case of example problem a.15, a better solution is found by placing the common subexpression $(I_n - AS(S^TAS)^{-1}S^T)$ in a separate assignment; in the resulting expression

$$\begin{aligned} M &:= \left(I_n - AS(S^TAS)^{-1}S^T\right) \\ X_{k+1} &:= S\left(S^TAS\right)^{-1}S^T + M^TX_kM, \end{aligned}$$

the conversion to a product of sums has no effect.

In this example, the extraction of a subexpression into a separate assignment allows to overcome a current limitation of Linnea because it affects the algorithm generation. In general, the fact that the extraction and conversely the substitution of assignments in the input does affect the algorithm generation can be seen as a limitation in itself. This is demonstrated with the second example; it consists of problem a.12. In the article where this problem is described [55], it appears in two different variants: The first variant consists of the two assignments:

$$\begin{aligned} \Lambda &:= S\left(S^TAWA^TS\right)^{-1}S^T \\ X_{k+1} &:= X_k + WA^T\Lambda(I_n - AX_k). \end{aligned} \tag{7.3}$$

The second variant only consists of one assignment; compared to the first variant, Λ in the second assignment is substituted with the right-hand side of the first assignment, and the first assignment is removed:

$$X_{k+1} := X_k + WA^TS\left(S^TAWA^TS\right)^{-1}S^T(I_n - AX_k). \tag{7.4}$$

Linnea is able to find a better solution for this second variant because 1) an additional occurrence of the common subexpression A^TS can be eliminated, 2) the common subexpression WA^TS can be eliminated, and 3) a better parenthesization is possible. If Λ is an output operand that needs to be computed explicitly, the assignment to Λ cannot be removed. However, even in that case it is still beneficial to substitute Λ in the second assignment and compute this expression as

$$\begin{aligned} \Lambda &:= S\left(S^TAWA^TS\right)^{-1}S^T \\ X_{k+1} &:= X_k + WA^TS\left(S^TAWA^TS\right)^{-1}S^T(I_n - AX_k). \end{aligned} \tag{7.5}$$

Again due to common subexpression elimination and a better parenthesization in the second assignment, this third variant (7.5) requires only about 1.7 times as many FLOPs as the second variant (7.4); in comparison, the first variant (7.3) is about 10 times as expensive as the second one (7.4).[6]

Instead of requiring the user to manually change the input expression, Linnea could automatically explore whether it is beneficial to

6 Based on the best solutions that are found within 30 minutes.

substitute assignments. The challenge is that the substitution of assignments can conflict with other optimizations; substituting assignments might undo a trick, and if there are multiple occurrences of the variable that is substituted, it is the opposite of common subexpression elimination. As a result, care has to be taken to avoid loops. In addition, the substitution of assignments can significantly increase the size of an expression. Finally, in order to know whether or not a substituted assignment can be removed, it needs to be possible for the user to specify whether this assignment is only used for convenience, to simplify the description of the input expression, or because the left-hand side of the assignment is an output operand that explicitly needs to be computed. This could be implemented by adding a special property to the input language for the operand on the left-hand side of those assignments that can be removed.

In order to increase the flexibility of Linnea, it would be possible to expose the number of different representations that are explored as a parameter to the user. With such a parameter, the user could chose between a faster generation time and the exploration of a larger search space.

8 COMMON SUBEXPRESSION ELIMINATION

Linear algebra expressions that appear in application problems frequently contain common subexpressions, that is, a subexpression that appears multiple times. In those cases, the algorithms for the application of kernels as described in Ch. 5 generate a sequence of kernels that computes this subexpression more than once. As an example, consider the expression

$$X := ABC$$
$$Y := ABD,$$

where all matrices are square. This expression contains the common subexpression AB. One possible sequence of kernels for this expression is

$$\begin{aligned} M_1 &\leftarrow AB \\ X &\leftarrow M_1 C \\ M_1 &\leftarrow AB \\ Y &\leftarrow M_1 D. \end{aligned}$$

Clearly, the second kernel call $M_1 \leftarrow AB$ is redundant and can be removed.

Several algorithms exists for the detection and elimination of redundant computations in a sequence of operations. This includes standard common subexpression elimination, partial redundancy elimination, and global value numbering [96, Chap. 13]. For Linnea, those algorithms have the disadvantage that their ability to detect a common subexpression depends on the sequence of kernels: Another sequence that computes the expression shown above with the same cost as the first one is

$$\begin{aligned} M_1 &\leftarrow BC \\ X &\leftarrow AM_1 \\ M_2 &\leftarrow BD \\ Y &\leftarrow AM_2. \end{aligned}$$

In this sequence, AB is effectively still computed twice, but existing algorithms are not able to detect this redundancy. By itself, this dependency on the sequence of kernels is not a fundamental limitation for Linnea because the exhaustive application of kernels eventually generates every sequence, including those where the redundancy can easily be detected and removed. Thus, after the construction of the

search graph, existing common subexpression elimination algorithms could simply be applied as a post-processing step on the sequences of kernels. In practice however, due to pruning only a subset of all sequences is explored. Recall that pruning is designed to only remove sequences that are known to be suboptimal. If the elimination of common subexpressions is performed on the sequences of kernels, pruning can prevent finding the optimal solution when a suboptimal sequence only becomes optimal through the elimination of common subexpressions. In addition, this approach has the disadvantage that it is not possible for the search algorithm to influence how the elimination of common subexpressions is combined with other generation steps.

To avoid the adverse effect of pruning on the elimination of common subexpressions, we developed an approach that does not operate on the sequence of kernels: Instead, the idea is to rewrite the input expression.[1] For instance,

$$X := ABC$$
$$Y := ABD$$

can be rewritten to

$$M_1 := AB$$
$$X := M_1 C$$
$$Y := M_1 D.$$

In the sequence of kernels generated for this expression, AB is computed only once. In the following, we refer to such a rewriting as the *replacement* of a common subexpression.

This approach has several advantages: It allows to reason about common subexpressions and their elimination in a way that is not possible otherwise. This reasoning can be used to integrate the search for a good replacement, that is, a replacement that leads to a good sequence of kernels, into the graph search of Linnea. In addition, it is possible to ensure that common subexpression elimination takes place already for the solutions found with the constructive algorithms. This is especially advantageous because the replacement of common subexpressions reduces the complexity of the input expression, such that a first solution is found more quickly.

In the domain of linear algebra, the detection of common subexpressions is more difficult compared to scalar expressions because

1 Alternatively, it would also be possible to eliminate common subexpressions with pattern matching: Whenever a match for a given kernel is found, all other occurrences of the operation computed by this kernel are replaced too. This approach is mention in Sec. 4.6, and it is used in CLAK [38]. It has the drawback that it depends on the (arbitrary) order in which matches for kernels are found. Thus, similar to the algorithms that operate on the sequences of kernels, with this approach the search algorithm cannot influence the elimination of common subexpressions.

Table 8.1: All common subexpressions and their number of occurrences in the expression $X_{k+1} := X_k + WA^TS(S^TAWA^TS)^{-1}S^T(I_n - AX_k)$. Individual operands are not considered.

subexpression	occurrences
A^T	2
S^T	2
A^TS	3
WA^T	3
WA^TS	3
AWA^TS	2
$(S^TAWA^TS)^{-1}S^T$	2

of transposition and inversion.[2] The reason is that expressions such as AB^T and BA^T can and should be considered as two occurrences of the same common subexpressions because one is the transpose of the other, that is $AB^T = (BA^T)^T$. Similar, AS and SA^T are a common subexpression if S is symmetric.

Compared to existing algorithms for common subexpression elimination that operate on sequences of operations, the replacement of common subexpressions in expressions introduces several new challenges:

MULTIPLE SUBEXPRESSIONS Especially when considering transposition and inversion, the number of common subexpressions in a given expression can be relatively large. A good example is application problem a.12 when written as a single assignment,

$$X_{k+1} := X_k + WA^TS\left(S^TAWA^TS\right)^{-1}S^T(I_n - AX_k),$$

where the matrix W is SPD. All common subexpressions (excluding individual operands) are shown in Tab. 8.1. For those with more than two occurrences, such as A^TS, it is possible that the best sequence of kernels is found by replacing only a subset of occurrences. The application of factorizations further increases the number of common subexpressions.

OVERLAPPING SUBEXPRESSIONS With large numbers of common subexpressions, it can happen that some occurrences overlap. This can happen both between different common subexpressions, such as

2 This is a general problem that also applies to the detection of common subexpressions in sequences of kernels.

WA^T and A^TS in example problem a.12, but also for occurrences of the same one: One such example is WA^TS in

$$X_{k+1} := X_k + \underbrace{WA^TS}_{1}\big(\overbrace{S^TAWA^TS}^{2}\big)^{-1}S^T(I_n - AX_k).$$

The occurrence labelled with 3 overlaps with the transposed occurrence 2, S^TAW. If some occurrences overlap either with occurrences of the same or a different common subexpression, it might still be possible to replace a subset of occurrences. Finally, there are cases where some common subexpressions are fully subsumed by others: All occurrences of both WA^T and A^TS are part of the occurrences of WA^TS. In those cases, it may not be necessary to replace WA^T or A^TS if WA^TS is replaced.

SELECTION OF REPLACEMENTS Not all common subexpressions should be replaced, either because the replacement may not be beneficial in terms of performance or numerical stability. Consider the assignments

$$x := ABv$$
$$y := ABw$$

as an example. In order to compute the common subexpression AB, an $\mathcal{O}(n^3)$ matrix-matrix product is necessary. However, both assignments can be computed with $\mathcal{O}(n^2)$ FLOPs by evaluating the products from right to left, only using matrix-vector products. An example of a common subexpression that should not be replaced in the interest of numerical stability is $(AWA^T)^{-1}$ in example problem a.4. In order to compute the resulting expression

$$M := \left(AWA^T\right)^{-1}$$
$$x_f := WA^TM(b - Ax)$$
$$x_o := W\left(A^TMAx - c\right),$$

it is necessary to explicitly invert AWA^T; the original expression can be computed by solving a linear system instead. Alternatively, it would also be possible to only replace AWA^T instead of $(AWA^T)^{-1}$. Finally, simply replacing the largest common subexpression is not guaranteed to lead to the best solution. For instance in case of example problem a.4, the optimal solutions is found by first replacing WA^T; for problem a.12 as shown above, replacing A^TS leads to the best sequence of kernels.

As can be seen from the proof in Sec. 4.6.3, the problem of finding an optimal sequence of kernels in the presence of common subexpressions

is NP-complete. For this reason, in Linnea we do not exhaustively explore all possible replacements of common subexpressions. Instead, heuristics are used to select only those that are most likely to lead to a good sequence of kernels. For the selection, we focus on the relationships between different common subexpressions and their occurrences.

In this chapter, we introduce a formalism that makes it possible to describe and reason about overlapping common subexpressions (Sec. 8.1). This formalism is used as a heuristic to select which common subexpressions are replaced. The actual algorithm for the detection, selection, and replacement of common subexpressions is presented in Sec. 8.2.

8.1 PRELIMINARIES

In this section, we introduce the formalism that is used in Linnea to reason about common subexpressions. As part of this formalism, we define some properties of common subexpressions that are used as heuristics to decide when to replace them.

We begin with the description of the common subexpressions in an expression. For this description, paths as defined in Def. 3.3 are not sufficient to identify every subexpression in the presence of associative operators. As an example, $a + b$ is a subexpression of $a + b + c$, but when represented as a variadic function, there is no single node in the expression tree that represents this subexpression. As a result, there is also no single path that identifies this subexpression. While it is possible to rewrite the variadic addition $a + b + c$ as two binary additions $(a + b) + c$ in order to introduce such a node, this rewriting introduces a dependency between the path to a subexpression and the representation of the expression. In addition, it becomes impossible to simultaneously also refer to another subexpression such as $b + c$. To solve this problem, we define positions on expressions, which consist of sets of paths. In the simplest case, a position p contains a single path π. In this case, the position p behaves exactly the same as the path π. If a position p contains more than one path, it identifies a subset of the operands of an associative function. As an example, for $t = AB^TC + D + E$ in Fig. 8.1, the position of the subexpression AB^T is $\{11, 12\}$. For functions $f(t_1, \ldots, t_n)$ that are only associative, all paths in a position are adjacent, in order to reflect the fact that only sequences of adjacent subexpressions t_i form valid subexpressions. While $\{11, 12\}$ describes a valid subexpression of $AB^TC + D + E$, $\{11, 13\}$ does not. For functions that are associative and commutative, the paths do not have to be adjacent; $\{1, 3\}$ is a valid position that identifies the subexpression $AB^TC + E$.

Definition 8.1 (Position)**.** Given an expression $t \in T(\Sigma, X)$, a *position* is a set of paths $p \subset \text{Paths}(t)$. The set of positions of t, $\text{Pos}(t)$, is defined as:

1. If $t \in X \cup \Sigma^{(0)}$, then $\text{Pos}(t) := \{\{\varepsilon\}\}$, where ε denotes the empty sequence.
2. If $t = f(t_1, \ldots, t_n)$, and
   a) f is not associative, then
   $$\text{Pos}(t) := \bigcup_{i=1}^{n} \{\{i\pi_1, \ldots, i\pi_k\} \mid \{\pi_1, \ldots, \pi_k\} \in \text{Pos}(t_i)\} \cup \{\{\varepsilon\}\}$$
   b) f is associative but not commutative, then
   $$\text{Pos}(t) := \bigcup_{i=1}^{n} \{\{i\pi_1, \ldots, i\pi_k\} \mid \{\pi_1, \ldots, \pi_k\} \in \text{Pos}(t_i)\} \cup \{\{\varepsilon\}\} \cup \text{Pos}_A$$
   where Pos_A is the set of all sets that contain sequences of adjacent integers from $\{1, \ldots, n\}$, that is
   $$\text{Pos}_A := \{\{i, i+1, \ldots, i+l-1\} \mid \text{for all } i \in \{1, \ldots, n-l+1\}, l \in \{2, \ldots, n-1\}\}$$
   c) f is associative and commutative, then
   $$\text{Pos}(t) := \bigcup_{i=1}^{n} \{\{i\pi_1, \ldots, i\pi_k\} \mid \{\pi_1, \ldots, \pi_k\} \in \text{Pos}(t_i)\} \cup \{\{\varepsilon\}\} \cup \text{Pos}_{AC}$$
   where Pos_{AC} is the set of all subsets of $\{1, \ldots, n\}$ with at least two and at most $n-1$ elements, that is
   $$\text{Pos}_{AC} := \{P \mid \text{for all } P \subset \{1, \ldots, n\} \text{ with } 2 \leqslant |P| \leqslant n-1\}$$

The subexpression at $p \in \text{Pos}(t)$, denoted by $t|_p$, is defined as:

1. If $p = \{\pi\}$, then $t|_p := t|_\pi$.
2. If $p = \{\tau i_1, \ldots, \tau i_n\}$ with $i_1, \ldots, i_n \in \mathbb{N}$, $i_1 < \ldots < i_n$ and $t|_\tau = f(t_1, \ldots, t_k)$, then $t|_p := f(t_{i_1}, \ldots, t_{i_n})$. ∎

As an example, the set of all position of $AB^TC + D + E$ in Fig. 8.1 is

$$\text{Pos}(t) = \big\{\{\varepsilon\}, \{1\}, \{11\}, \{12\}, \{121\}, \{13\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{11,12\}, \{12,13\}\big\}.$$

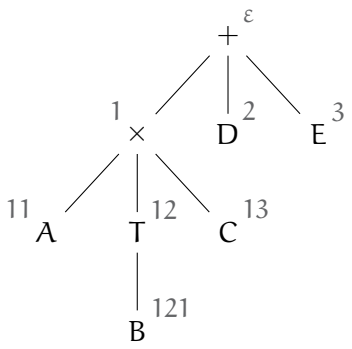


Figure 8.1: Expression tree for $AB^TC + D + E$. Nodes are labelled with their paths.

In the definition of positions, for $t = f(t_1, \ldots, t_n)$ where f is associative (and optionally commutative), we exclude the position $\{1, \ldots, n\}$ from $\mathrm{Pos}(t)$. This ensures that positions are unique. If we included this position, for an expression t' with $\pi \in \mathrm{Paths}(t')$ and $t'|_\pi = t$, it would be possible to refer to t in two different ways: With the position $\{\pi\}$ containing a single path, or with a set of paths $\{\pi 1, \ldots, \pi n\}$, one path for every t_i in f. By excluding $\{1, \ldots n\}$, the latter is not possible.

Definition 8.2 (Common Subexpression)**.** Given an expression $t \in T(\Sigma, X)$, a *common subexpression of* t is a set of positions $P \subset \mathrm{Pos}(t)$ with $|P| > 1$, such that for all $p_1, p_2 \in P$ with $p_1 \neq p_2$, $t|_{p_1} = t|_{p_2}$ holds. The set of all common subexpressions of an expression t is denoted by $\mathrm{CSE}(t)$. ■

For an expression t and a common subexpression $P \in \mathrm{CSE}(t)$, we refer to a position $p \in P$ as an occurrence of the common subexpression in t. In the following, if it is clear from the context, we usually refer to a common subexpression $P \in \mathrm{CSE}(t)$ by one of its occurrences $t|_p$, $p \in P$, instead of the set of positions P.

Transposed (and similarly, inverted) common subexpressions can be incorporated with an equivalence relation on expressions. As an example, let $t_1 \sim t_2$ be the relation $t_1 = t_2 \vee t_1 = t_2^T$. The definition of common subexpressions can be extended to take into account such an equivalence relation as follows:

Definition 8.3 (Common Subexpression under Equivalence)**.** Given an expression $t \in T(\Sigma, X)$, a *common subexpression of* t *under an equivalence* $\sim$ is a set of positions $P \subset \mathrm{Pos}(t)$, such that for all $p_1, p_2 \in P$, $t|_{p_1} \sim t|_{p_2}$. The set of all common subexpressions that are equivalent under $\sim$ of an expression t is denoted by $\mathrm{CSE}_\sim(t)$. ■

As shown earlier, it is possible for different occurrences of one or more common subexpressions to overlap with one another. Whether two occurrences overlap can be tested using positions.

Definition 8.4 (Overlapping Positions)**.** Let $t \in T(\Sigma, X)$ be an expression, $p_1, p_2 \in \mathrm{Pos}(t)$ be two positions, and l_1 and l_2 be the lengths of the paths in p_1 and p_2, respectively. p_1 and p_2 *overlap* if and only if

1. $l_1 = l_2$ and $p_1 \cap p_2 \neq \varnothing$, or
2. $l_1 > l_2$ and the common prefix π of length l_2 of the paths in p_1 is contained in p_2, or
3. $l_1 < l_2$ and the common prefix π of length l_1 of the paths in p_2 is contained in p_1. ■

Example 8.1. Let $t = AB^TC + D + E$, $p_1 = \{12, 13\}$ and $p_2 = \{121\}$, such that $t|_{p_1} = B^TC$ and $t|_{p_2} = B$. Then, $l_1 = 2$ and $l_2 = 3$. The common prefix of length $l_1 = 2$ of the paths in p_2 is 12. Since $12 \in p_1$, p_1 and p_2 overlap. ■

A common subexpression $P \in \mathrm{CSE}_{\sim}(t)$ is said to be replaceable if for all $p_1, p_2 \in P$ with $p_1 \neq p_2$, p_1 and p_2 do not overlap. If a common subexpression is not replaceable, it might be possible to make it replaceable by only considering a subset $P' \subset P$ of its occurrences.

Similar to the overlap relation, we define a relation among subexpressions to test whether one is contained within the other.

Definition 8.5 (Subexpression Relation on Positions)**.** Let t be an expression, $p_1, p_2 \in \mathrm{Pos}(t)$ be two positions, and l_1 and l_2 be the lengths of the paths in p_1 and p_2, respectively. The occurrence at position p_1 is a *subexpression* of the occurrence at p_2, denoted by $p_1 \sqsubset p_2$, if

1. $l_1 = l_2$ and $p_1 \subset p_2$, or
2. $l_1 > l_2$ and the common prefix π of length l_2 of the paths in p_1 is contained in p_2. ■

Example 8.2. Let $t = AB^TC + D$, $p_1 = \{11, 12\}$ and $p_2 = \{1\}$, such that $t|_{p_1} = AB^T$ and $t|_{p_2} = AB^TC$. Then, $l_1 = 2$ and $l_2 = 1$. The common prefix of length $l_2 = 1$ of the paths in p_1 is 1. Since $1 \in p_1$, p_1 is a subexpression of p_2. ■

Using this subexpression relation on positions, we can formalize what it means for a common subexpression to be subsumed by another. The subsume relation forms a partial order on all common subexpressions of an expression. It can be used as a heuristic to decide which common subexpressions to replace as follows: A common subexpression is only used if it is maximal according to the partial order, that is, it is only used if it is not subsumed by any other common subexpression.

Definition 8.6 (Subsume Relation)**.** Given an expression $t \in T(\Sigma, X)$ and two common subexpressions $P_1, P_2 \in \mathrm{CSE}_{\sim}(t)$, P_1 *subsumes* P_2 if for every $p_2 \in P_2$, there is a $p_1 \in P_1$ such that $p_2 \sqsubset p_1$. In the following, we also write $P_2 \sqsubset P_1$ if P_1 subsumes P_2. ■

Example 8.3. We use problem a.5,

$$x := (A^{-T}B^{T}BA^{-1} + R^{T}LR)^{-1}A^{-T}B^{T}BA^{-1}y, \tag{8.1}$$

as an example. To reduce the length of paths throughout this example, we consider the right-hand side of (8.1) as ε. We look at three common subexpression: $A^{-T}B^{T}BA^{-1}$, $B^{T}B$ and $A^{-T}B^{T}$. Their positions are shown, from left to right, in (8.2), (8.3), and (8.4), respectively.

$$p_1 = \{111\} \qquad p_2 = \{2,3,4,5\} \tag{8.2}$$

$$q_1 = \{1112,1113\} \quad q_2 = \{3,4\} \tag{8.3}$$

$$r_1 = \{1111,1112\} \quad r_2 = \{1113,1114\} \quad r_3 = \{2,3\} \quad r_4 = \{4,5\} \tag{8.4}$$

1. $A^{-T}B^{T}BA^{-1}$ subsumes $B^{T}B$ because $q_1 \sqsubset p_1$ and $q_2 \sqsubset p_2$.
2. $A^{-T}B^{T}BA^{-1}$ subsumes $A^{-T}B^{T}$ too because $r_1 \sqsubset p_1$, $r_2 \sqsubset p_1$, $r_3 \sqsubset p_2$ and $r_4 \sqsubset p_2$.
3. $B^{T}B$ and $A^{-T}B^{T}$ are not comparable according to the subsume relation because none of their positions are comparable. ∎

In practice, using the subsume relation as a heuristic is insufficient because it does not consider how many occurrences of a common subexpression can actually be replaced. For instance, in case of example problem a.12, written as a single assignment,

$$X_{k+1} := X_k + WA^{T}S\left(S^{T}AWA^{T}S\right)^{-1}S^{T}(I_n - AX_k),$$

the common subexpression $A^{T}S$ would not be used because it is subsumed by $WA^{T}S$. However, while both appear three times, only for the former, all occurrences can be replaced at the same time; two occurrences of $WA^{T}S$ overlap with one another. Thus, in addition to the subsume relation, we also take into account the number of replaceable occurrences, that is, the maximal number of occurrences that do not overlap with one another.

Definition 8.7 (Maximal-Replaceable CSE)**.** Given an expression $t \in T(\Sigma, X)$ and a set of positions $P \subset \mathrm{Pos}(t)$, let $|P|_R \in \mathbb{N}$ be the the size of the largest subset $P' \subseteq P$ such that for all $p_1, p_2 \in P'$, p_1 and p_2 do not overlap. A common subexpression $Q \in \mathrm{CSE}_{\sim}(t)$ is *maximal-replaceable* if for all common subexpressions $Q' \in \mathrm{CSE}_{\sim}(t)$ with $Q \sqsubset Q'$, $|Q|_R > |Q'|_R$ holds. ∎

Intuitively, $|P|_R$ is the maximal number of occurrences that can be replaced at the same time. A common subexpression is maximal-replaceable if increasing the size of the subexpression always reduces the number of replaceable occurrences. Conversely, a common subexpression is not maximal-replaceable if it is possible to increase its size and keep the number of replaceable occurrences constant. If $|P|_R = 1$, there are no two occurrences that can be replaced at the same time;

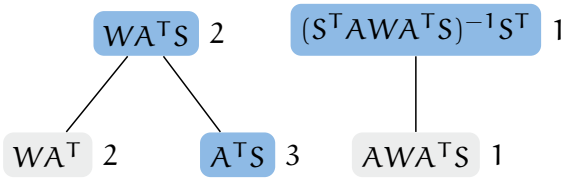


Figure 8.2: The subsume relation of the common subexpressions in example problem a.12. The common subexpressions at the bottom are subsumed by the ones at the top. Nodes are labelled with their $|P|_R$; blue nodes are maximal-replaceable. As explained in Sec. 8.2.1, single transposed operands are not considered for replacement; for this reason, A^T and S^T are not included in this graph.

such common subexpressions are not considered for replacement. As an example, consider again

$$X_{k+1} := X_k + WA^TS\left(S^TAWA^TS\right)^{-1}S^T(I_n - AX_k).$$

A visualization of the subsume relation of the common subexpressions in this expression is shown in Fig. 8.2. The common subexpression WA^T is not maximal-replaceable because it can be replaced two times, as often as the common subexpression WA^TS that subsumes it. A^TS on the other hand is maximal-replaceable; while it is also subsumed by WA^TS, all of its three occurrences can be replaced at the same time. WA^TS is not subsumed by AWA^TS because the leftmost occurrence of the former is not contained in any occurrence of the later. Thus, since WA^TS is not subsumed by any other common subexpression, it is maximal-replaceable. Even though according to the definition, $(S^TAWA^TS)^{-1}S^T$ is maximal-replaceable too, neither $(S^TAWA^TS)^{-1}S^T$ nor AWA^TS are considered for replacement because both can only be replaced once.

8.2 THE ALGORITHM

The common subexpression elimination algorithm in Linnea consists of three parts; the detection of common subexpressions, the selection of those common subexpressions that are replaced, and the actual replacement. As the equivalence relation $t_1 \sim t_2$ on expressions, we use $t_1 = t_2 \vee t_1 = t_2^T \vee t_1 = t_2^{-1} \vee t_1 = t_2^{-T}$ to account both for transposed and inverted occurrences.

8.2.1 *Detection*

The detection of common subexpressions can be thought of as grouping the positions of equivalent subexpressions. This is implemented as the construction of a mapping from expressions to lists of positions. In this mapping, equivalent subexpressions are mapped to the same list. As an example, the mapping for the expression $AB^T + CBA^T$ is shown

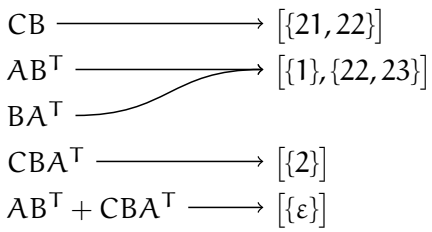


Figure 8.3: The mapping for the detection of common subexpression in $AB^T + CBA^T$. Entries for single operands are not included.

```
1  cses := dict()                                  # dictionary initialization
2  for P in Pos(s):
3    t := s|_P
4    subexpr := None
5    for t' in [t, t^T, t^-1, t^-T]:
6      if t' in cses:
7        subexpr := t'
8        break
9    if subexpr is None:
10     cses[t] := [P]                              # add entry t ↦ [P]
11   else:
12     cses[t] := cses[subexpr]    # add reference from t to list of positions
13     cses[t].append(P)                  # append P to list of positions
```

Figure 8.4: Pseudocode of the algorithm for the detection of common subexpressions.

in Fig. 8.3. The pseudocode of the algorithm that constructs the mapping is shown in Fig. 8.4. The algorithm proceeds as follows: Given an expression s, all subexpressions $t = s|_P$, $P \in \mathrm{Pos}(s)$ are constructed (lines 2–3). For every t, it is tested whether any equivalent expression of t, that is, t^T, t^{-1}, or t^{-T}, is in the mapping (lines 5–8). If there is no entry yet, an entry added that maps t to a list containing its position P (lines 9–10). If there is an entry $t' \mapsto l$, where t' is equivalent to t, then an entry is created that maps t to l (line 12), and P is added to the existing list l (line 13). All lists that contain at least two positions that do not overlap form a common subexpression.

The complexity of this algorithm is determined by the number of positions $|\mathrm{Pos}(s)|$. In the worst case, with functions that are associative and commutative, the number of positions can be exponential in the number of nodes in the expression tree of s (see Def. 8.1).

Subexpressions that should not be considered for the elimination of common subexpressions are already excluded from this mapping. Those subexpressions are

1. single operands, because they do not require any computation,

2. transposed operands, because many kernels support transposed input operands, and replacing transposed operands introduces explicit transposition,

3. inverted expressions that appear in products, because they introduce explicit inversion when a linear system could be solved instead, and

4. expressions that only consist of operands resulting from the application of a factorization, because kernels are not applied to such expressions (see Sec. 4.4).

Equivalence relations can lead to the detection of spurious common subexpressions that are not replaceable. This problem can be illustrated with the expression $t = (AB)^{-1}C$. Since $t|_{\{1\}} = (AB)^{-1}$ is the inverse of $t|_{\{11\}} = AB$, according to Def. 8.3 this expression contains the common subexpression $\{\{1\},\{11\}\}$. To avoid the detection of such common subexpression, one could exclude subexpressions from the mapping that have the inversion (or transposition) operator at their root. As a result, in the previous example, $t|_{\{1\}} = (AB)^{-1}$ would not be added to the mapping. While the exclusion of such subexpressions prevents that $(AB)^{-1}$ is detected as a common subexpression, it does not prevent that AB is detected as part of another common subexpression as long as $t|_{\{11\}} = AB$ is added to the mapping. In Linnea, excluding such subexpressions is not necessary because the transposition is always distributed, such that transposition only appears on single operands, and both transposed operands as well as inverted expressions are already excluded from the mapping.

8.2.2 *Selection*

The selection of common subexpressions consists of two phases: In the first phase, we select all the maximal-replaceable ones. In the second phase, for the remaining common subexpressions, subsets of replaceable occurrences are selected.

In order to identify maximal-replaceable common subexpressions, $|P|_R$ needs to be determined. This is be done as follows: In a first step, a graph is constructed where each position in P is represented by a node, and two nodes are connected with an edge if the positions do not overlap. $|P|_R$ is then given by the size of a maximum clique of the graph.While the problem of finding cliques in a graph is NP-complete [46], in practice, those graphs usually have less than ten nodes. To find cliques, the algorithm by Bron and Kerbosch is used [19]. An example

A maximum *clique is a clique with the largest possible number of nodes in a given graph. A clique is* maximal *if its size cannot be increased. While all maximum cliques are also maximal, the reverse does not hold in general.*

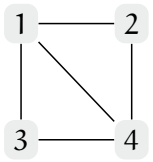


Figure 8.5: Compatibility graph of the common subexpression $B_{k-1}A^TW_k$.

of the graph for the common subexpression $B_{k-1}A^TW_k$ in example problem a.16,

$$B_k := \frac{k}{k-1}\Big((B_{k-1} - \underbrace{B_{k-1}A^TW_k}_{1}\big((k-1)I_l + \overbrace{W_k^TA\underbrace{B_{k-1}}_{3}A^TW_k}^{2}\big)^{-1}\overbrace{W_k^TAB_{k-1}}^{4}\Big),$$

here with B_{k-1} multiplied in, is shown in Fig. 8.5. This graph contains two maximum cliques; $\{1,2,4\}$ and $\{1,3,4\}$. It follows that $|P|_R$ of $B_{k-1}A^TW_k$ is 3.

For the selection of the subsets of replaceable occurrences, we use again the graph of positions: A subsets of replaceable occurrences corresponds to a clique in the graph of positions. In Linnea, we only consider maximal cliques. In the example, the two maximum cliques $\{1,2,4\}$ and $\{1,3,4\}$ are also the only maximal cliques.

8.2.3 *Replacement*

For each common subexpression that is replaced, a new intermediate operand is generated that represents the replaced expressions. The intermediate operand is added to the table of intermediate operands, such that the properties of the intermediate operand can be inferred from the common subexpression (see Sec. 6.2). In sequences of multiple assignments, the new assignment is placed directly above the first assignment that contains an occurrences.

With transposed and inverted common subexpressions, there is the choice of which occurrences are considered as the original, and which ones are considered to be transposed and/or inverted. For instance, in $X := AB^T + CBA^T$, one could either consider AB^T as the original occurrence, and BA^T as the transposed one, or vice versa. If AB^T is considered as the original, the replacement yields

$$M := AB^T$$
$$X := M + CM^T.$$

Alternatively, one obtains

$$M := BA^T$$
$$X := M^T + CM.$$

In Linnea, one alternative is chosen arbitrarily.

SUCCESSOR GENERATION ORDER To determine the order in which the different replacements are returned, we use two criteria:

1. The overall number of operands that is replaced, that is, the number of occurrences times the number of operands per occurrence.

2. The number of occurrences that is replaced.

All common subexpression that are considered for replacement are sorted by the overall number of operands in decreasing order. To break ties, the number of occurrences is used; common subexpressions with a larger number of occurrences are given precedence.

While the cost of computing a common subexpression might be an obvious criterion, it is difficult to use in practice. The problem is that this cost is not known at the time of the replacement; it is necessary to find a sequence of kernels that computes the common subexpression first. In addition, if the replaced common subexpression contains others that might be replaced later, the cost depends on the replacement of other common subexpressions.

8.3 CONCLUSION

The presented algorithm for the elimination of common subexpressions, especially the selection process, is designed to be flexible and modular. The implementation in Linnea is only one of many alternatives; it was chosen because in combination with Linnea's search algorithm, it produces good results for the example problems used in the experiments.

The algorithm is designed to replace one common subexpression at a time; to replace multiple common subexpressions in one expression, we rely on the fact that in the graph search, common subexpression elimination is applied repeatedly. Alternatively, one could also replace multiple common subexpressions at once. This can be done by constructing graphs of positions for multiple common subexpressions; for example pairs of two common subexpressions, or even all common subexpressions in an expression. In that case, care has to be taken because it is possible that cliques contain only one occurrence of a common subexpression. The replacement of multiple common subexpressions at once might be useful when the common subexpression elimination algorithm is applied to an expression only once.

In situations where the exploration of a search space is not possible, the selection can be adapted to only select one common subexpression. In that case, as a heuristic one could rank common subexpressions by their cost of computation, or the memory required for the result.

9 CODE GENERATION

Some of the material presented in this chapter has been published in [6], [8], and [9].

A path in the search graph is only a symbolic representation of an algorithm; it still has to be translated to actual Julia code. Most importantly, all operands are represented symbolically, with no notion of where and how they are stored in memory. During the code generation, operands are assigned to memory, and it is decided in which storage format they are stored.

Many BLAS and LAPACK kernels overwrite one of their input operands. As an example, the GEMM kernel $\alpha AB + \beta C$ writes the result into the array containing C. While overwriting input operands can be used to reduce the amount of memory used, as a consequence the code cannot be in SSA form; a conversion out of SSA form is necessary. In order to make use of the kernels' ability to overwrite input operands, a naive conversion of the individual calls is not sufficient. A call such as $M \leftarrow \alpha AB + \beta C$ can be translated to code by allocating memory for the output operand M, copying C to M, and using M instead of C as an argument. However, if C is not used again later, neither the allocation nor the copy is necessary; they can both be avoided by passing the original C and allowing M to overwrite it. A conversion out of SSA form that is able to avoid those unnecessary allocations and copies requires a liveness analysis to determine when operands can be overwritten.

Some kernels use specialized storage formats for matrices with properties. For instance, the GETRF kernel that computes the LU factorization only stores the upper, non-zero part of the upper triangular matrix U. Those storage formats have to be considered when generating code: While specialized kernels for triangular matrices only access the non-zero entries, a more general kernel would read from the entire array. Thus, it has to be ensured that operands are always in the correct storage format, if necessary through a conversion.

In addition, specialized storage formats make it possible that an array contains multiple operands. Some kernels make use of this option for their output. GETRF for example stores both L and U in the array that contained its input operand.[1] If multiple operands are stored in the same array, care has to be taken to ensure that an operand is not accidentally overwritten. Most of the challenges related to storage formats can be avoided by always converting operands to full matrices, where all entries are represented explicitly. This approach is

1 Some kernels also require that multiple input operands are stored in the same array. One such example is the GETRS kernel that uses the result of a LU factorization to solve a linear system. As mention in Sec. 5.7, such kernels are not used in Linnea, so it is not necessary to support this case.

implemented by the majority of other languages and libraries for linear algebra. However, it has the disadvantage that is requires additional copies and memory. For instance, in case of the LU factorization, the conversion of all operands to full matrices almost triples the amount of required memory: After the call to GETRF, L and U are stored in a single array of size $n \times n$; the permutation matrix P is stored as a one-dimensional vector of size n. As full matrices, the amount of memory required by those operands increases from $n^2 + n$ to $3n^2$.

The goal of the code generation in Linnea is to perform the conversion out of SSA form while avoiding copies and storage format conversions as much as possible. It should be noted that while Linnea currently only generates Julia code, the approach described in this chapter is not specific to any particular language.

MEMORY MODEL Linnea operates on a continuous amount of memory which is populated with *memory locations* that are allocated and freed[2] as necessary. A memory location is a consecutive piece of memory that contains one or more operands. In case of Julia, memory locations are typically arrays. Their physical location during the execution is determined by the Julia runtime environment. It is assumed that all operands are available in memory and do not have to be read from or written to files.

ORGANIZATION OF THIS CHAPTER This chapter is structured as follows: In Sec. 9.1, the algorithm for the code generation is outlined. Storage formats and storage format conversions are discussed in more detail in Sec. 9.2. This chapter concludes with some possible directions for future work in Sec. 9.3.

9.1 TRANSLATION TO MEMORY-IR

In order to generate the code, the symbolic representation of an algorithm is translated to an augmented representation that considers the location of operands in memory, their storage formats, as well as memory operations such as copies and storage format conversions. In the following, we refer to this representation as the Memory-IR (M-IR). The grammar of the M-IR is shown in Fig. 9.1. In addition to the kernel calls, the M-IR includes the following memory operations:

(1) The allocation of an uninitialized memory location. As an example, for a matrix of size 100×100, this operation translates to `Array{Float64}(undef, 100, 100)` in Julia.

(2) The allocation of a memory location that is initialized with a constant operand. In the case of a square identity matrix

2 Since Julia uses garbage collection, memory locations are freed automatically.

```
code    = { kernel | mem_op };
mem_op  = ml "= alloc(" int "," int ")";                          (1)
        | ml "=" const;                                           (2)
        | "copy(" ml "," ml ")";                                  (3)
        | "convert(" ml "," format "," format ")";                (4)
        | ml "= convert(" ml "," format "," format ")";           (5)
const   = "Identity(" int "," int "," format ")";
        | "Zero(" int "," int "," format ")";
        | number;
```

Figure 9.1: Grammar of the M-IR in extended Backus–Naur form. The non-terminal *kernel* represents a kernel call, *ml* is a memory location, *int* is a positive integer, *format* is a storage format as shown in Tab. 9.1, and *number* is a floating-point number.

of size 100 which is stored as a full matrix, this operation is translated to `Array{Float64}(I, 100, 100)`.

(3) The creation of a copy of an operand. Since this operation does not allocate a new memory location for the copied operand, it needs to be preceded by an allocation. This operation is implemented with the BLAS COPY kernel.

(4), (5) The conversion of the storage format of an operand, either in-place (4) or out-of-place (5). `convert` takes as arguments both the input and the output storage format. Out-of-place conversions allocate memory for the output operand; an additional allocation operation is not necessary.

The idea for the conversion to M-IR is to traverse the sequence of kernels, assign symbolic operands to memory locations, and add memory operations as necessary. During the translation, a mapping $\mu : \Sigma^{(0)} \to \mathcal{ML}$ is used to assign all live symbolic operands ($\Sigma^{(0)}$) to their memory locations ($\mathcal{ML}$). A second mapping $\rho : \Sigma^{(0)} \to \mathcal{SF}$ assigns symbolic operands to their storage formats ($\mathcal{SF}$). Both mappings are updated during the translation to reflect the current state of the memory during the execution of the code. μ is used to assign memory locations to the arguments of kernels, while ρ is used to determine if storage format conversions are necessary. In order to determine if the modification of an operand in a given memory location may affect another operand in the same memory location, for each memory location there is a list of the operands it currently contains. The translation algorithm performs the following steps:

1. In order to determine if and when operands can be overwritten, a liveness analysis is performed on the symbolic representation of the algorithm. Since this representation is in SSA form and does

not contain any control flow statements[3], the liveness analysis is trivial: The kernels are traversed in reverse order. If an operand shows up as the input of a kernel for the first time, that is its last use. If an operand shows up as the output, that is its definition.

2. μ and ρ are initialized with the input operands of the algorithm. Since constant operands are not considered part of the input, but instead created when necessary, they are not added to μ and ρ in this step. At present, it is assumed that all input matrices are stored as full matrices.

3. The sequence of kernels is traversed. In order to generate memory operations and update μ and ρ, the algorithm inspects the input and output operands of each kernel.

   For input operands, there are two cases to consider:

   a) If the storage format of an input operand of a kernel is not compatible with the required storage format, a storage format conversion is generated. The storage format in ρ is updated. If the conversion has to be done out-of-place, μ is updated with the new location. Storage formats as well as storage format conversions are discussed in more detail in Sec. 9.2.

   b) For input operands that are constant (for example the identity matrix), one of the following two actions is performed: If the operand is overwritten by the kernel, a new memory location is allocated that is initialized with the operand, and the new operand is added to μ. Alternatively, if the operand is not overwritten, the constant operand is passed directly. Constant operands are directly generated in the required storage format.

   For output operands, there are three cases:

   a) For output operands that do not overwrite an input operand, a new memory location is allocated and the operand is added to μ.

   b) For output operands that would overwrite an input operand that is still needed (its liveness extends past the current kernel), the memory location containing the operand that would be overwritten is copied. After the generation of the copy operation, μ is updated with the new location of the copied operand or operands. The existing memory location is used for the output operand.

3 While the code snippets for operations not supported by BLAS and LAPACK may contain loops, those snippets are implemented so that they can be treated as a single kernel.

c) For output operands that overwrite an input operand that is not needed anymore (its liveness ends at the current kernel), no memory operations need to be generated. The output operand is added to μ.

In all three cases, the storage formats of the output operands are added to ρ. After all input and output operands of a kernels are inspected, μ is used to replace the symbolic operands in the kernel call with their memory locations, and all input operands whose liveness ends at the current kernel are removed from μ and ρ.

Example 9.1. As an example for the translation to M-IR, we use the artificial input problem

$$X := \alpha AB + 3I$$
$$Y := CDE + LX.$$

All matrices have size 1000×1000. L is lower triangular, I is the identity matrix; all other matrices are full. Let

$$X \leftarrow \alpha AB + 3I$$
$$M_1 \leftarrow DE$$
$$M_2 \leftarrow LX$$
$$Y \leftarrow CM_1 + M_2$$

be a sequence of kernels generated for this problem. The code for this algorithm in M-IR is shown in Fig. 9.2. The comments in lines 1, 6, 11, 17, and 21 show the state of μ, the remaining comments show the operations computed by the kernels. Since all operands are stored as full matrices throughout the algorithm, storage format conversions are not necessary.

ll. 1–4 The constants 3 and I are not considered to be part of the input to the algorithm, and they are initially not contained in μ. Instead, they are only instantiated and added to μ when they are needed. Since the first GEMM call overwrites the memory location that contains I on input with its output X, I has to be stored in a memory location; 3 can instead be passed directly.

ll. 6–9 The second GEMM call only computes the product of D and E; since `beta` is set to zero, nothing is overwritten. In this case, an empty memory location is allocated for the output M_1.

ll. 11–15 The TRMM call that computes LX overwrites the memory location that contains X, `ml7`, with its output M_2. However, since X is an output of the algorithm, its liveness does not end at this kernel. Thus, the memory location `ml9` is allocated, and X is copied to it.

```
1  # {α ↦ ml0, A ↦ ml1, B ↦ ml2, L ↦ ml3, C ↦ ml4, D ↦ ml5, E ↦ ml6}
2  ml7 = Identity(1000, 1000, full)
3  # X ← αAB + 3I
4  gemm!('N', 'N', ml0, ml1, ml2, 3.0, ml7)
5
6  # {L ↦ ml3, C ↦ ml4, D ↦ ml5, E ↦ ml6, X ↦ ml7}
7  ml8 = alloc(1000, 1000)
8  # M1 ← DE
9  gemm!('N', 'N', 1.0, ml5, ml6, 0.0, ml8)
10
11 # {L ↦ ml3, C ↦ ml4, X ↦ ml7, M1 ↦ ml8}
12 ml9 = alloc(1000, 1000)
13 copy(ml7, ml9)
14 # M2 ← LX
15 trmm!('L', 'L', 'N', 'N', 1.0, ml3, ml7)
16
17 # {C ↦ ml4, X ↦ ml9, M1 ↦ ml8, M2 ↦ ml7}
18 # Y ← CM1 + M2
19 gemm!('N', 'N', 1.0, ml4, ml8, 1.0, ml7)
20
21 # {X ↦ ml9, Y ↦ ml7}
```

Figure 9.2: Example of the M-IR.

ll. 17–19 Since the liveness of M_2 ends at the final GEMM call, it can safely be overwritten with the output Y.

l. 21 At the end of the algorithm, X is stored in memory location `ml9`, and Y is stored in `ml7`. ■

CODE GENERATION FOR KERNELS CALLS Most BLAS and LAPACK kernels require arguments that go beyond the mathematical input operands. For instance, a call to the TRSM kernel that solves the upper triangular linear system $2U^{-T}B$ has only three mathematical input operands; 2, U, and B. However, as Julia code, the same call is written as `trsm!('L', 'U', 'T', 'N', 2.0, ml0, ml1)`, where `ml0` and `ml1` are memory locations that contain the input operands U and B. The first four arguments determine the specific operation that this call to TRSM computes. For example, `'U'` specifies that an upper triangular systems is solved. Those arguments can be ignored for the purpose of this chapter; they were set during the generation of the patterns that represent the operations computed by a kernel from the description of kernels through partial function application (see App. b, especially App. b.1.8). Only a few of the BLAS and LAPACK wrappers in Julia expose arguments for operand sizes and strides. One such case

Table 9.1: Description of the storage formats supported by Linnea. Matrices are of size $m \times n$.

storage format	description
`full`	All entries are represented explicitly.
`upper_triangular`/ `lower_triangular`	Upper/lower triangular matrix, stored in a two-dimensional array of size $m \times n$. Only the non-zero upper/lower triangular half is represented explicitly.
`upper_triangular_ud`/ `lower_triangular_ud`	As above, except that the elements on the diagonal are 1, and the diagonal is not represented explicitly.
`symmetric_upper`/ `symmetric_lower`	Symmetric matrix stored in a two-dimensional array of size $m \times n$. Only the upper/lower triangular half is represented explicitly.
`diag_vec`	Diagonal matrix, stored as a one-dimensional array of size $\min(m, n)$ that only contains the diagonal elements.
`perm_vec`	Permutation matrix, stored as a one-dimensional array that contains a permutation of the integers $\{1, \ldots, n\}$.
`ipiv`	Permutation matrix, stored as a one-dimensional array as produced by the LU factorization kernel GETRF.

is the DOT kernels with the signature `dot(n, X, incx, Y, incy);` `n` is the length of the vectors `X` and `Y`, and `incx` and `incy` are their strides. The kernel description allows to specify how to set such arguments based on the mathematical input operands, in this case `X` and `Y`. Those arguments are then set automatically during the translation to M-IR when μ is used to replace the symbolic operands.

9.2 STORAGE FORMATS

The storage formats supported by Linnea are shown in Tab. 9.1. During the translation to M-IR, storage format conversions are added when necessary. To avoid unnecessary conversions, Linnea implements a mechanism to reason about them: A compatibility relation on the storage formats is defined that forms a partial order. Given two storage formats x and y, x is considered to be compatible with y if 1) all values that are explicitly represented in x are also explicitly represented in y, and 2) those values are stored in the same positions (relative to

```
output = zeros(Float64, m, n)
for i = 1:min(m, n);
   output[i, i] = input[i];
end;
```

Figure 9.3: Code snippet for the storage format conversion from `diag_vec` to `full`.

the first element of the memory location) in both formats.[4] It should be noted that the values in a given storage format do not need to be actual elements of the matrix. As an example, the `perm_vec` format is a permutation of the integers $\{1, \ldots, n\}$, even though the permutation matrix that is represented only has elements that are either 0 or 1. The input operands of all kernels are annotated with the required storage formats (see App. b.1). If the format of the input operand is compatible with the required one, no conversion is necessary; only if it is not, a conversion is selected returns a compatible format. To this end, Linnea contains a collection of storage format conversions. Due to the transitivity of the compatibility relation, it is not necessary to provide conversions for all pairs of formats; if no conversion to the required format is available, a conversion to a different compatible format can be used instead.

As an example for how the compatibility relation is used, consider the TRSV kernel that accesses only the lower triangular half of the input matrix A. In this case, we say that TRSV requires A to be in the `lower_triangular` format. If a matrix that is stored as `full` is passed to TRSV, no storage format conversion is necessary because `lower_triangular` is compatible with `full`. The reverse does not hold: If a kernel requires a `full` matrix, it is not possible to use one that is stored as `lower_triangular`; it is necessary to set the upper half to zero first. There are also cases where two storage formats for a given property are not compatible in either direction: One such example are diagonal matrices, which can be stored as `full` and `diag_vec`. As a result, storage format conversions are necessary in both directions.

Storage format conversions are implemented as code snippets in Julia. In some cases, the conversion can be done with an existing function. For instance, the conversion from `full` to `diag_vec` is implemented with the `diag` function. The conversion in the other direction is implemented with the code shown in Fig. 9.3.

We distinguish between two different types of conversions: In-place and out-of-place. Whenever possible, storage format conversions are done in-place. This may not be possible if 1) the amount of memory used changes, for example when converting between `diag_vec` and

4 In an object-oriented language, a storage format could also be represented by a class. Different storage formats are compatible if the classes implement compatible interfaces.

`full`, or 2) when the conversion would overwrite another operand that is still needed. The latter can happen after factorizations such as LU, when both L and U are stored in the same memory location. To save implementation effort, storage format conversion that can be done in-place are only implemented as in-place conversions; if it is necessary to perform the conversion out-of-place, the operand is copied first.

During the translation to M-IR, storage format conversions are selected as follows: Given the format of the input operand and the required format, all conversions with the input format as their source format are checked. The first conversion that has a target format that is compatible with the required format is used. Once a conversion is selected, the necessary memory operations are generated: Since the allocation of the new location is part of out-of-place conversions, no allocation operation is necessary. For in-place conversions, there are two cases to consider: 1) If no other operand is stored in the memory location, the conversion is done in place; no additional memory operation are necessary. 2) If another operand is stored in the same location, the conversion is done out-of-place. In this case, a new memory location is allocated, the operand that needs to be converted is copied to the new location, and the conversion is performed on the new location. In all cases, ρ is updated with the new storage format. If applicable, μ is updated with the new memory location of the converted operand.

Example 9.2. We demonstrate how storage formats are considered in the translation to M-IR with the expression $X := A^T A + BD$. All matrices have size 1000×1000. With the exception of D, which is diagonal, all matrices are full. As a sequence of kernels for this expression, we use

$$M_1 \leftarrow A^T A$$
$$M_2 \leftarrow BD$$
$$X \leftarrow M_1 + M_2.$$

The code in M-IR is shown in Fig. 9.4.

l.5 The product $A^T A$ is computed with the SYRK kernel; the result M_1 is stored in the `symmetric_lower` format.

ll.9–11 The DIAGMMR kernel is a code snippet that computes the product XY, where Y is diagonal. Y has to be stored in the `diag_vec` format, that is, as a one-dimensional vector that only contains the elements on the diagonal. Since D is initially stored as a full matrix, a storage format conversion is introduced. This conversion has to be done out-of-place, so D is moved from `ml1` to the new location `ml4`.

ll.15–17 The AXPY kernel that is used to compute the sum $M_1 + M_2$ requires both operands to be stored as full matrices.

```
1  # μ = {A ↦ ml2, B ↦ ml0, D ↦ ml1}
2  # ρ = {A ↦ full, B ↦ full, D ↦ full}
3  ml3 = alloc(1000, 1000)
4  # M1 ← A^T A
5  syrk!('L', 'T', 1.0, ml2, 0.0, ml3)
6
7  # μ = {B ↦ ml0, D ↦ ml1, M1 ↦ ml3}
8  # ρ = {B ↦ full, D ↦ full, M1 ↦ symmetric_lower}
9  ml4 = convert(ml1, full, diag_vec)
10 # M2 ← BD
11 diagmmr(ml0, ml4) # overwrites ml0
12
13 # μ = {M1 ↦ ml3, M2 ↦ ml0}
14 # ρ = {M1 ↦ symmetric_lower, M2 ↦ full}
15 convert(ml3, symmetric_lower, full)
16 # X ← M1 + M2
17 axpy!(1.0, ml0, ml3)
18
19 # μ = {X ↦ ml3}
20 # ρ = {X ↦ full}
```

Figure 9.4: Example for an algorithm in M-IR with storage format conversions.

Thus, the symmetric matrix M_1 is converted to the `full` storage format. This conversion can be done in place, so M_1 remains in memory location `ml3`. ■

9.2.1 *Auxiliary Storage Formats*

In addition to the ones in Tab. 9.1, Linnea uses a small number of auxiliary storage formats that do not describe actual formats; instead they describe some special cases where a single fixed storage format is not sufficient. As an example, `as_overwritten` describes that an output operand has the same storage format as the input operand that was originally stored in the same memory location. It is used for instance for the output of the TRTRI kernel that computes the inverse of a triangular matrix. The `symmetric_triangular` format is used for symmetric input matrices that can either be stored as `symmetric_upper` or `symmetric_lower`. It is used for kernels such as SYMM and SYRK. As a final example, the `explicit_diagonal` format is used for the input operand A in a kernel that computes $A + D$, where A is a general and D a diagonal matrix. Since this kernel only modifies the diagonal of A, the only requirement for the storage format of this matrix is that the diagonal is represented explicitly. This is the case for

`full`, `upper_triangular`, `lower_triangular`, `symmetric_upper`, and `symmetric_lower`.

The auxiliary formats are integrated into the partial order in a way such that in most cases, no special treatment is necessary in the algorithm for the selection of storage format conversions. As an example, `symmetric_upper` and `symmetric_lower` are compatible with `symmetric_triangular`.

9.3 FUTURE WORK

At present, the translation to M-IR is rather basic; the sequence of kernels is considered to be fixed, and memory operations are generated by a greedy algorithm. As a result, there are several opportunities to improve the quality of the generated code:

1. In some cases, it might be possible to avoid copying operands by reordering kernel calls. In addition, with a suitable cost function, reordering kernels calls and memory operations could be used to improve the caching behavior of the code [103].

2. It is not possible for the user to specify the storage formats of the input and output operands, nor which operands can or cannot be overwritten. Instead, it is assumed that all input and output operands are stored as `full`, and that input operands can always be overwritten. Depending on the use case, those assumptions can lead to unnecessary memory operations. Those operations could be avoided by allowing the user to specify the storage formats of input and output operands, whether operands can be overwritten, and which output operands should be overwritten by which input operands.

3. The algorithm for the translation to M-IR is currently not able to take advantage of kernels that compute the same mathematical operation but use different storage formats. Especially if the set of supported storage formats is extended, considering storage formats for the selection of kernels could help to avoid some storage format conversions. While it might be difficult to incorporate storage formats into the algorithm generation, the replacement of mathematically equivalent kernels that use different storage formats could potentially be implemented as an optimization on the M-IR.

4. Only a subset of the storage formats used by BLAS and LAPACK is supported. Additional storage formats include a packed format for triangular and symmetric matrices that store a $n \times n$ matrix in an array of size $n(n+1)/2$, and storage formats for tridiagonal and banded matrices that only store the non-zero diagonals.

5. There are a few kernels whose interface is not compatible with the current memory model; the most notable examples are the GEQRF kernel that computes the QR factorization, and the ORMQR kernel that computes a product with the matrix Q resulting from GEQRF. The problem is twofold: 1) Parts of Q are stored in two memory locations. However, since μ is a surjective mapping of operands to memory locations, it is currently not possible that parts of one operand are stored in different memory locations. 2) There is no storage format conversion in LAPACK to convert an orthogonal matrix that is stored as `full` to the storage format of Q. This is a problem because storage formats are not considered for the selection of kernels. If the ORMQR kernel was used in Linnea, it would always be selected for the multiplication with an orthogonal matrix because it is the cheapest kernel for this operation, even if that matrix was not the result of a QR factorization and consequently not stored in the required storage format. In that case, however, it would not be possible to convert Q to the required format. In order to use GEQRF and ORMQR, and extended memory model is necessary, and storage formats need to be considered for the selection of kernels. At present, to circumvent this problem, the `qr!` function is used which provides a higher-level interface to the QR factorization.

As mention earlier, the code generation described in this chapter is not specific to Julia. However, the Julia language has some properties that simplify minor aspects of the code generation. In order to support lower-level languages such as C, some extension are necessary:

1. In Julia, all function arguments can be passed directly,[5] whereas the C interface of BLAS and LAPACK usually requires pointers to the actual arguments. As a result, auxiliary variables need to be created for all arguments, including those that determine the functionality of kernels.

2. Some LAPACK kernels require additional memory as a workspace, which needs to be passed to the kernel in the form of an array. The optimal size of this array can be computed with a call to the same kernel with special arguments. While in Julia, this additional kernel call takes place in the LAPACK wrappers, in C it needs to be added to the generated code.

3. Since there is no garbage collection in C, memory locations need to be freed whenever the liveness of an operand ends.

Finally, in Julia, most storage format conversions and code snippets for operations not supported by BLAS and LAPACK can be implemented

5 Julia implements pass-by-sharing.

efficiently with relatively little effort. In many cases, existing Julia function can be reused, and simple operations as well as loops are vectorized almost automatically. Implementing the same functionality with comparable performance in C requires more effort.

Many challenges discussed in this chapter are caused by the fact that the storage formats required by kernels are mostly fixed. Thus, the code generation could be simplified either with kernels that support multiple formats, or with the automatic generation of kernels for a given combination of formats. The latter is implemented by TACO for dense and sparse tensor operations [80]. Likewise, the selection of storage format conversions could be simplified by their automatic generation, similar to the approach presented in [22] for sparse tensors.

10 EXPERIMENTS

Most of the material presented in this chapter has been published in [9]. In addition, some of the material has been published in [8].

To evaluate Linnea, we perform four different types of experiments.[1]

1. We assess the quality of the code generated by Linnea[2] by comparing against Julia[3], Matlab[4], Eigen[5], and Armadillo[6] (Sec. 10.2).
2. We then investigate the generation time with and without merging branches, as well as the fraction of the search space that is explored (Sec. 10.3).
3. We evaluate the quality of FLOPs as a cost function (Sec. 10.4).
4. We conclude by investigating the influence of the hardware in the three experiments above (Sec. 10.5).

For all but Matlab, we linked against the Intel MKL implementation of BLAS and LAPACK (MKL 2020.1.217); Matlab instead uses MKL 2019.0.3. For the execution of generated code, all reported timings refer to the minimum of 20 repetitions, each on cold data, to avoid any caching effects. The generation time was obtained from one single repetition. Unless noted otherwise, the generation time was limited to 30 minutes.

The measurements for experiments 1. to 3. were taken on a dual socket system, featuring two Intel Xeon E5-2680 v3 (*Haswell* architecture) with 12 cores each, a clock speed of 2.5 GHz and 64 GB of RAM. Hyper-Threading and Turbo Boost were disabled on this machine, and we had exclusive access to the machine when the experiments were conducted. The parallel experiments were run with 24 threads. The measurements for experiment 4. were taken on a dual socket Intel Xeon Platinum 8160 (*Skylake* architecture) with 24 cores each, a base clock speed of 2.1 GHz and 192 GB of RAM. Hyper-Threading was disabled, while Turbo Boost was enabled.[7] In addition, we did not have exclusive access to the machine. The parallel experiments were run with 12 threads on a single NUMA region. All experiments were performed using double precision.

1 The code for the experiments is available at https://github.com/HPAC/linnea/tree/master/experiments.
2 Executed with Python version 3.9.1.
3 Version 1.5.3.
4 Version 2020a.
5 Version 3.3.9.
6 Version 10.2.x.
7 The maximal Turbo Boost clock speed depends on the number of cores that are used: If one core is active, it is 3.7 GHz; if all 24 cores are active, it is 2.8 GHz.

TEST PROBLEMS We use two different sets of test problems, one consisting of expressions coming from applications, and a synthetic one. The first set consists of a collection of 25 problems from real applications, from domains such as image and signal processing, statistics, and regularization. Those problems are shown in App. a; in those problems, the operand sizes are selected to reflect realistic use cases. The second set consists of 100 randomly generated linear algebra expressions, each consisting of a single assignment. The number of operands in each expression is chosen uniformly between 4 and 7. Operand dimensions are chosen uniformly between 50 and 2000 in steps of 50. We set square operands to have a 75% probability to have one of the following properties: diagonal, lower triangular, upper triangular, symmetric, or symmetric positive definite. To introduce realistic common subexpressions, some expressions contain patterns of the form XX^T and XMX^T, where X is a subexpression with up to two matrices, and M is a symmetric matrix.

Both sets of tests problems have somewhat different characteristics. For the application problems, we intentionally selected the largest and most complex problems we could find; in our experience, average problems are simpler. In comparison, the random problems usually consist of smaller expressions and exhibit less structure.

In some of the figures in this chapter, the test problems are sorted by their computational intensity. The intensity is computed as the number of FLOPs performed by the optimal sequence of kernels divided by the amount of data that this sequence of kernels operates on. The amount of data is computed as the number of double precision elements of unique operands that appear in the problem. For the number of elements, properties are considered. As an example, the number of elements of a diagonal matrix of size $n \times n$ is counted as n.

10.1 LIBRARIES AND LANGUAGES

For each library and language, two different implementations are used: *naive* and *recommended*. The naive implementation is the one that comes closest to the mathematical description of the problem. It is also closest to the input to Linnea. As examples, in Tab. 10.1 we provide the implementations of $A^{-1}BC^T$, where A is symmetric positive definite and C is lower triangular. Since documentations almost always discourage the use of the inverse operator to solve linear systems, we instead use dedicated functions, for example `A\B`, in the recommended implementations. The different implementations are described below.

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

Table 10.1: Input representations for the expression $A^{-1}BC^T$, where A is SPD and C is lower triangular. The letters "n" and "r" denote the naive and recommended implementation, respectively.

Name	Implementation
Julia n	`inv(A)*B*transpose(C)`
Julia r	`(A\B)*transpose(C)`
Armadillo n	`arma::inv_sympd(A)*B*(C).t()`
Armadillo r	`arma::solve(A, B)*C.t()`
Eigen n	`A.inverse()*B*C.transpose()`
Eigen r	`A.llt().solve(B)*C.transpose()`
Matlab n	`inv(A)*B*transpose(C)`
Matlab r	`(A\B)*transpose(C)`

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

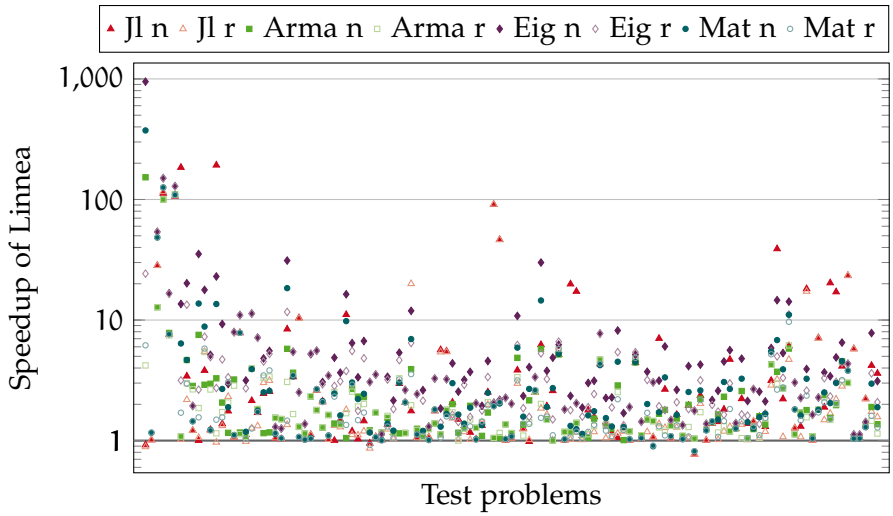
EIGEN In the recommended implementation, matrix properties are described with views. For linear systems, we select solvers based on properties.

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.

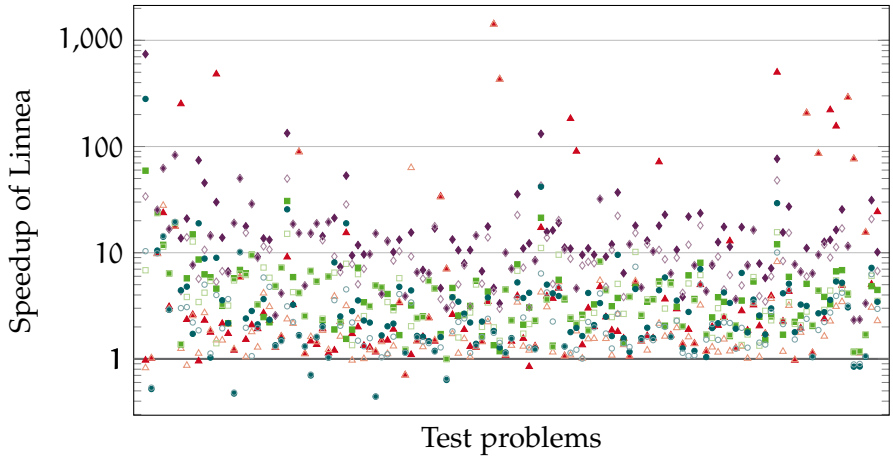
10.2 QUALITY OF THE GENERATED CODE

In Fig. 10.1, we present the speedups of the code generated by Linnea over other languages and libraries for both the random and application test cases. For one and 24 threads, the code generated by Linnea is the fastest in 94% and 89% of the cases, respectively. If not the fastest, the code is at most 1.3× and 2.3× slower than the other languages and libraries. Fig. 10.2 summarizes the results in performance profiles [32].

There is no fundamental difference between the speedups for the random and the application problems. The most notable difference is that for the random problems there are more cases where the speedups are close to 1; the speedups for the application problems tend to be higher. Specifically, in the single-threaded case the median speedup of the code generated by Linnea over all other implementations is 1.6× for the random problems, compared to 3.2× for the application problems. In the multi-threaded case, the medians are respectively 3.1× and 4.7×. In addition, with one thread, the speedups for the

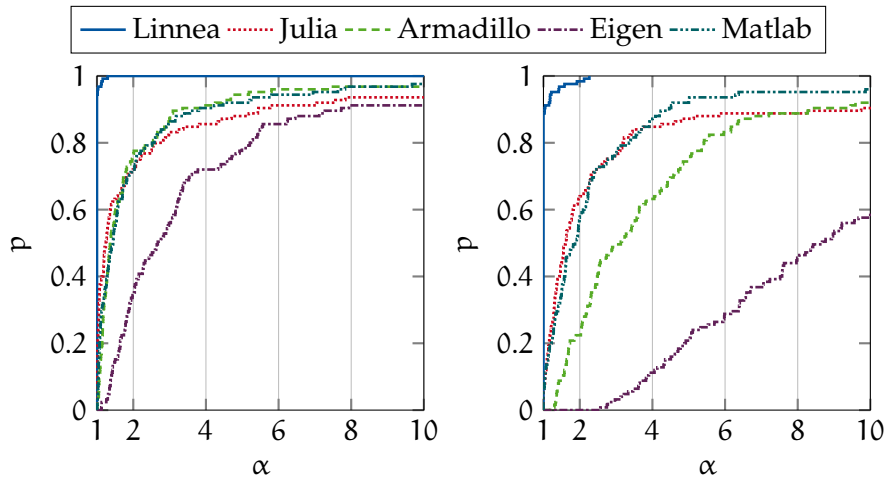


(a) 1 thread.



(b) 24 threads.

Figure 10.1: Speedup of Linnea over four reference languages and libraries for 125 test problems. The test problems are sorted by computational intensity, increasing from left to right.



(a) 1 thread.

(b) 24 threads.

Figure 10.2: Performance profiles: For a given implementation $\mathcal{I}$, and a given point α on the x-axis, the corresponding value p on the y-axis indicates the relative number of test problems for which $\mathcal{I}$ is at most a factor of α slower than the fastest of all implementations. For the other languages and libraries, the execution times of the naive and recommended implementations were merged; per example problem, the lower execution time was selected.

random problems have a larger spread (in both directions); with 24 threads, the spread is similar.

To understand where the speedups for the code generated by Linnea come from, we discuss the details of few exemplary test problems.

DISTRIBUTIVITY The application problem a.11,

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

which is part of an image restoration application [126], illustrates well how distributivity might affect performance. Due to the matrix-matrix product $H^\dagger H$, the computation of y_k based on the original formulation of the problem leads to $\mathcal{O}(n^3)$ FLOPs. Instead, for y_k Linnea finds the solution

$$v := -Hx_k + y$$
$$y_k := H^\dagger v + x_k,$$

which only uses matrix-vector products (GEMV), and requires $\mathcal{O}(n^2)$ FLOPs. This solution is obtained in two steps: First, $H^\dagger y + (I_n - H^\dagger H)x_k$ is converted to Linnea's normal form, returning $H^\dagger y + x_k - H^\dagger H x_k$; then, by factoring out $H^\dagger$, the expression is written back to a product of sums, resulting in $H^\dagger(y - Hx_k) + x_k$, which can be computed with two calls to GEMV. For this problem, this optimization

yields speedups between 4.2× (Matlab naive) and 7.7× (Eigen naive) with respect to the other languages and libraries for one thread, and speedups between 3.4× (Matlab naive) and 32× (Eigen naive) for 24 threads.

ASSOCIATIVITY With the exception of Armadillo, none of the languages and libraries we compare with consider the matrix chain problem (see Sec. 5.6.2). Instead, products are always computed from left to right. The synthetic test case $X := M_1 M_1^T (M_2 + M_3) M_4 v_5 v_6^T$ is a good example to illustrate the importance of making use of associativity in products. The operands have the following dimensions: $M_1 \in \mathbb{R}^{150\times 450}$, $M_2, M_3 \in \mathbb{R}^{150\times 900}$, $M_4 \in \mathbb{R}^{900\times 100}$, $v_5 \in \mathbb{R}^{100}$, and $v_6 \in \mathbb{R}^{150}$. All matrices are full. Not only does Linnea successfully avoid any matrix-matrix products in the evaluation of this problem, surprisingly Linnea even finds a solution that avoids the sum $M_2 + M_3$. As a first step, the matrix-vector product $z_1 := M_4 v_5$ is computed. Then, Linnea rewrites the resulting $X := M_1 M_1^T (M_2 + M_3) z_1 v_6^T$ as $X := M_1 M_1^T M_2 z_1 v_6^T + M_1 M_1^T M_3 z_1 v_6^T$, where a second matrix-vector product $z_2 := M_3 z_1$ is computed. The resulting expression is rewritten again to $X := M_1 M_1^T (M_2 z_1 + z_2) v_6^T$, which is now computed as a sequence of three more matrix-vector products and one outer product:

$$
\begin{aligned}
z_3 &:= M_2 z_1 + z_2 \\
z_4 &:= M_1^T z_3 \\
z_5 &:= M_1 z_4 \\
X &:= z_5 v_6^T
\end{aligned}
$$

Despite the rather small operand sizes, the speedups for this test case are between 7.5× and 17× with one thread, and between 2.9× and 17× with 24 threads.

COMMON SUBEXPRESSIONS Expressions arising in applications frequently exhibit common subexpressions; one such example is given by problem a.17,

$$B_1 := \frac{1}{\lambda_1}\left(I_n - A^T W_1 \left(\lambda_1 I_l + W_1^T A A^T W_1\right)^{-1} W_1^T A\right),$$

which is used in the solution of large least-squares problems [23]. Linnea successfully identifies that the term $W_1^T A$ (or its transposed form $(A^T W_1)^T$) appears four times, and computes it only once. In this example, these savings lead to speedups between 5.2× and 6.6× with one thread, and between 3.9× and 20× with 24 threads.

PROPERTIES Many matrix operations can be sped up by taking advantage of matrix properties. As an example, here we discuss the evaluation of application problem a.19, $x := (A^T A + \alpha^2 I)^{-1} A^T b$, a least-squares problem with Tikhonov regularization [50], where matrix

A is of size 3000×200 and has full rank. Since A has more rows than columns and is full rank, Linnea is able to infer that A^TA is not only symmetric, but also positive definite (SPD). Similarly, Linnea infers that $\alpha^2 I$ is SPD because 1) the identity matrix is SPD, 2) α^2 is positive and 3) a SPD matrix scaled by a positive factor is still SPD. Since the sum of two SPD matrices is SPD, $A^TA + \alpha^2 I$ is identified as SPD. As a result, the Cholesky factorization is used to solve the linear system. If $A^TA + \alpha^2 I$ had not been identified as SPD, a more expensive factorization such as LU had to be used. Finally, since Linnea infers properties based on the annotations of the input matrices, no property checks have to be performed at runtime; if the input matrices have the specified properties, all inferred properties hold. Altogether, the code generated for this assignment is between $1.2\times$ and $5.5\times$ faster than the other languages and libraries with one thread, and $2.2\times$ and $13\times$ faster with 24 threads.

Epilog

In general, the speedups of Linnea depend both on the potential for optimization in a given problem, as well as on the similarity of the default evaluation strategy of each language and library to the optimal one.

In case of problem a.19 for example, with one thread, the code generated by Linnea is $3.4\times$ faster than the recommended Armadillo implementation, but only $1.2\times$ faster than the naive implementation. The reason is that for this problem, the parenthesization has the largest influence on the execution time. While Armadillo does solve a simplified version of the matrix chain problem, the `solve` function used in the recommended implementation (see Tab. 10.1) effectively introduces a fixed parenthesization. Due to the explicit inversion in the naive implementation, there is no such fixed parenthesization, so Armadillo is able to find a solution which is very similar to that generated by Linnea.

For problem a.6, which is the loop body of a blocked algorithm for the inversion of a triangular matrix, there is a relatively large spread between the speedups: The recommended Julia, Matlab, and Armadillo solutions are respectively around $1.4\times$, $1.5\times$, and $3.1\times$ slower than Linnea, while the recommended Eigen implementation is $12\times$ slower (one thread). In this case, the spread is mostly caused by a combination of the interface that the different systems offer, and how they utilize properties. Neither Armadillo nor Eigen have functions to solve linear systems of the form AB^{-1}, with the inverted matrix on the right-hand side. Thus, even in the recommended solution, for $X_{10} := L_{10}L_{00}^{-1}$, explicit inversion is used instead. In addition, Eigen is not able to identify that L_{00} is lower triangular and instead uses an algorithm for the inversion of a general matrix, leading to a significant

Table 10.2: Statistics regarding the size of the generated code. Empty lines and comments are not counted towards the number of lines of code.

	Average	Median	Min	Max
Lines of code	19.4	19	7	46
Kernel calls	7.3	7	2	21

loss in performance, while Julia, Matlab, and Armadillo correctly make use of this property .

For expression a.14, all solutions have very similar execution times; the speedups of Linnea are between 1.3× and 2.2× with one thread. The cost of computing this problem is dominated by the cost of computing the value of X_{k+1}, for which the solution found by all other languages and libraries is almost identical to the solution found by Linnea. While Linnea is able to save some FLOPs in the computation of Λ, those savings are negligible for the evaluation of the entire problem. With 24 threads, there is a larger spread, with speedups ranging from 1.0× to 16×. This spread is likely caused by the differences in how well the operations not supported by BLAS and LAPACK are parallelized.

As can be seen in Fig. 10.1, with 24 threads there are a few cases where the naive and recommended Matlab implementations are up to 2.3× faster than the code generated by Linnea. In all those cases, the generated code either contains code snippets for operations not supported by BLAS and LAPACK or storage format conversions. Most likely, the suboptimal performance of the generated code is caused by the fact that those code snippets and storage format conversions are not sufficiently optimized for the multi-threaded execution.

It is important to note that by changing the operand sizes, in several cases one can increase or reduce the speedups almost arbitrarily. The goal of our experiments is to show that there are many cases where the code generated by Linnea achieves speedups (much) larger than one, indicating that it finds better algorithms than the other languages and libraries.

SIZE OF THE GENERATED CODE Tab. 10.2 contains some statistics about the size of the code generated by Linnea. The difference between the number of kernel calls and the lines of code is caused by the code for memory allocations, copying operands, storage format conversions, and the code snippets for operations not supported by BLAS and LAPACK; while we treat those snippets as one kernel call, they may consists of multiple lines of code. Considering the relatively small number of kernels, the manual translation of the generated code to other languages should be relatively straightforward for an

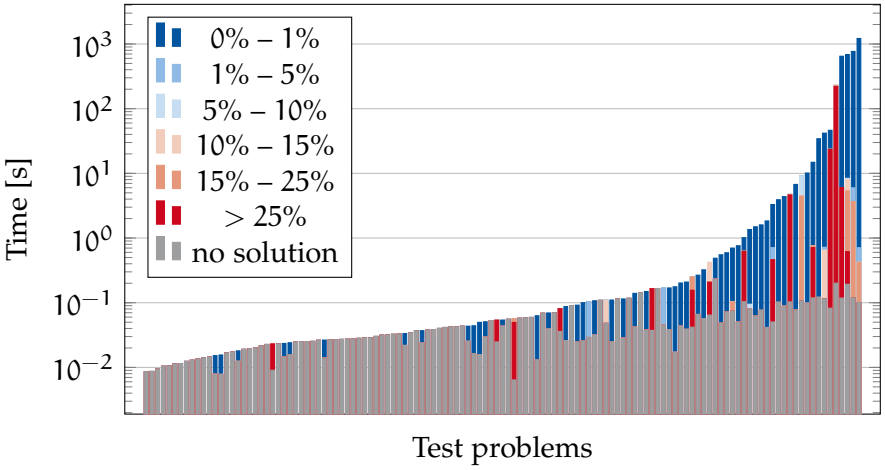


Figure 10.3: Code generation time in Linnea. The height of the bars indicates the time to find $\mathcal{S}_{30\text{min}}$. The colors indicate the cost of the current best solution at any given time, relative to the cost of $\mathcal{S}_{30\text{min}}$. The relative cost is given as the overhead over the best solution, in percent. Gray indicates that no solution has been found yet. Bars that are only gray indicate that $\mathcal{S}_{1\text{st}}$ and $\mathcal{S}_{30\text{min}}$ coincide.

expert, not least because in the code, each kernel is annotated with the mathematical operation that it performs.

10.3 GENERATION TIME AND MERGING

The search algorithm gives Linnea flexibility: A potentially suboptimal solution can be found quickly, and better ones can be found if more time is invested. In the following, we distinguish between 1) the time needed for the construction of the graph, and 2) the time needed to retrieve the best solution from the graph and to translate this solution into code. In this section, we use $\mathcal{S}_{1\text{st}}$ to refer to the solution (that is, the sequence of kernels) that is found first. As a reference, since in general the truly optimal solution is not known, we use the best solution that is found by Linnea within 30 minutes as a proxy. In the following, we call this solution $\mathcal{S}_{30\text{min}}$. Note that $\mathcal{S}_{30\text{min}}$ may be found in much less than 30 minutes, and it can coincide with $\mathcal{S}_{1\text{st}}$.

Fig. 10.3 reports for all 125 test problems the graph construction time and the quality of the different solutions found over time. For all problems, $\mathcal{S}_{1\text{st}}$ is found in less than one second; for 83% of the problems, also $\mathcal{S}_{30\text{min}}$ is found in less than one second. In only five cases, $\mathcal{S}_{30\text{min}}$ is found after more than two minutes. In terms of FLOPs, $\mathcal{S}_{1\text{st}}$ is within 25% of $\mathcal{S}_{30\text{min}}$ for 88% of the test problems, and within 1% in 81% of the cases.

Since the application problems usually consists of larger and more complex expressions than the random problems, the generation time tends to be larger for the application problems: For the random prob-

lems, finding $\mathcal{S}_{30\text{min}}$ takes longer than one second only for 6% of the cases, compared to 60% for the application problems. Out of the five cases where finding $\mathcal{S}_{30\text{min}}$ takes more than two minutes, two are random problems, while three are application problems.

The average time to retrieve the best sequence of kernels ($\mathcal{S}_{30\text{min}}$) from the graph is 0.06 seconds (maximum 0.6 seconds); the average time to generate the code is 0.03 seconds (maximum 0.2 seconds).

10.3.1 *Impact of Merging Branches*

As discussed in Sec. 4.2, in order to reduce the size of the search graph and thus speed up the algorithm generation, redundant branches in the derivation graph are merged. To evaluate the impact of this optimization, we performed the algorithm generation with this optimization enabled and disabled. Since merging branches only reduces redundancy without eliminating any solutions, given sufficient time, the same solutions will be found. As the search graph initially contains very little redundancy, the time to find $\mathcal{S}_{1\text{st}}$ is mostly unaffected by the merging. There are, however, notable differences in the time to find $\mathcal{S}_{30\text{min}}$, especially for those problems where the best solution is not found within a few seconds. Without merging, there are 12 test problems for which the best solution found with merging is not found within 30 minutes. In 17 cases, it takes more than twice as long to find $\mathcal{S}_{30\text{min}}$, including 7 cases where it takes at least 10 times longer.

10.3.2 *Explored Percentage of the Search Space*

The fraction of the search space that is actually explored is reduced both by pruning, and by limiting the time spent on the search. In order to give an idea of the effect of pruning and limiting the algorithm generation time, we performed two different experiments. For those experiments, the application problems were divided into two sets: 1) Problems where the search terminates within 30 minutes, and 2) problems where the search does not terminate within 30 minutes. As a proxy for the size of the search space, the number of nodes of the search graph is used; the number of sequences of kernels can be larger than the number of nodes.

The first set is used to assess the effect of pruning on the size of the search space. For this experiment, pruning was disabled for the algorithm generation, and the time limit was set to 48 hours. The results of this experiment are shown in Tab. 10.3. While there are exceptions, the percentage of the search graph that is pruned tends to increase with the size of the graph. For large graphs, pruning decreases the number of nodes by more than 99%. As mentioned at the beginning of Sec. 4.1, pruning only removes parts of the graph that are guaranteed to lead to suboptimal solutions.

Table 10.3: Percentage of the search graph that is explored with pruning enabled relative to the size of the search graph without pruning. The † indicates that Linnea did not terminate within 48 hours, the ‡ indicates that Linnea ran out of memory before reaching the time limit.

	Nodes		
Problem	Pruning	No Pruning	Explored (%)
a.6	21	21	100
a.18	123	131	94
a.19	39	47	83
a.25	15,446	22,885	67
a.11	53	85	62
a.1	17	36	47
a.24	1,976	6,878	29
a.3	224	5,264	4.3
a.2	576	15,301	3.8
a.21	30,701	$> 1,584,319$	$< 1.9^{\dagger}$
a.9	14,187	$> 1,130,459$	$< 1.3^{\ddagger}$
a.17	1,068	90,521	1.1
a.22	9,841	$> 1,300,063$	$< 0.76^{\dagger}$
a.20	6,239	844,740	0.74

Table 10.4: Percentage of the search graph that is explored in 30 minutes relative to the size of the search graph after 48 hours. The † indicates that Linnea did not terminate within 48 hours, the ‡ indicates that Linnea ran out of memory before reaching the time limit.

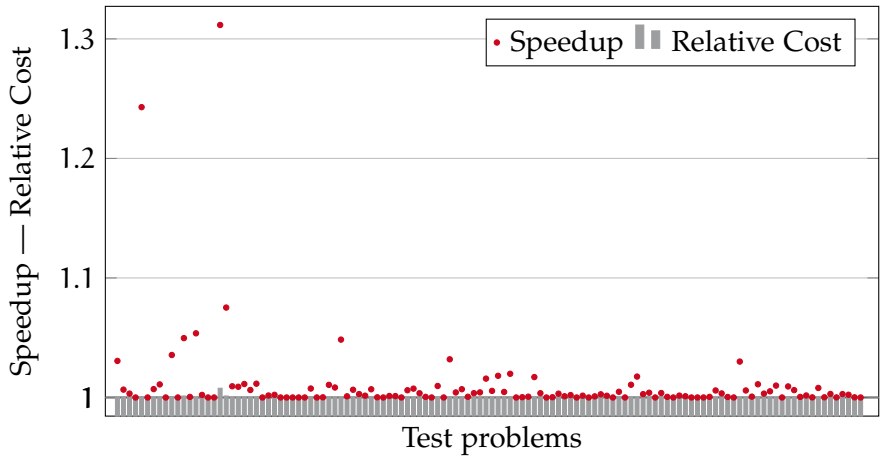
	Nodes		
Problem	30 minutes	> 48 hours	Explored (%)
a.5	57,689	184,793	31
a.4	54,087	199,209	27
a.7	58,923	235,276	25
a.13	47,521	462,442	10
a.16	64,768	> 688,892	< 9.4†
a.8	89,434	> 1,051,288	< 8.5†
a.10	45,509	> 584,790	< 7.8†
a.23	70,149	> 1,045,802	< 6.7†
a.15	39,023	> 609,476	< 6.4‡
a.12	46,246	> 765,551	< 6.0†
a.14	44,800	> 928,947	< 4.8†

The second set of example problems is used to investigate the effect of limiting the graph search to 30 minutes. For this experiment, the time limit was again set to 48 hours, but pruning was enabled. The results are shown in Tab. 10.4. In several cases, the full search space was not explored even after 48 hours. In case of the most difficult problem, less than 5% of the search space is explored in 30 minutes. Nonetheless, there are only two cases in which the additional generation time leads to an improved solution. For problem a.14, the improvement of the cost is about 0.001%; for problem a.15, the cost is reduced by 29%.

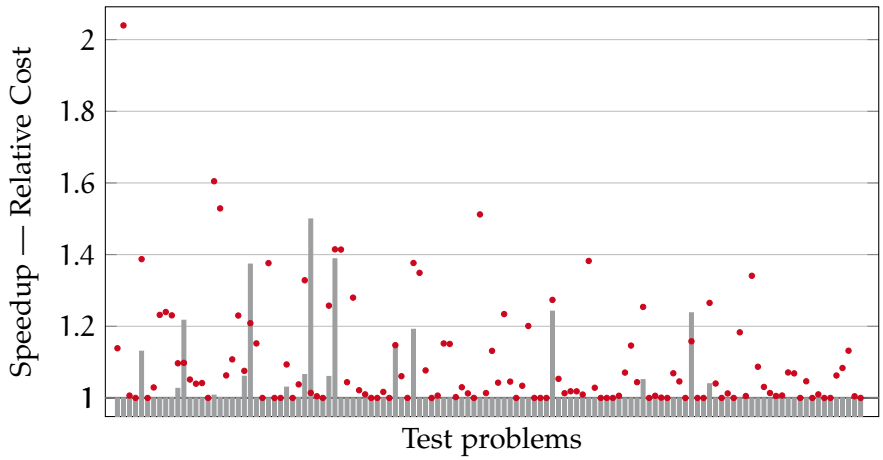
It should be noted that in addition, the size of the search space is intentionally reduced by decisions such as avoiding the application of overly general kernels (Sec. 5.5 and Sec. 5.7). The effect of those decisions is difficult to quantify; we estimate that they decrease the size of the search space by up to one order of magnitude.

10.4 QUALITY OF THE COST FUNCTION

As a cost function, Linnea uses the number of FLOPs. To assess the accuracy of this function, we modified the pruning (at line 7 in the algorithm in Fig. 4.2) such that all algorithms with a cost of up to 1.5× of the best solution are generated and run the best 100 of those algorithms per test problem. In the following, we use s_{FLOPs} to denote



(a) 1 thread.



(b) 24 threads.

Figure 10.4: Comparison between $\mathcal{S}_{\text{time}}$ and $\mathcal{S}_{\text{FLOPs}}$ in terms of execution time (dots) and FLOP count (bars) for 1 and 24 threads. The test problems are sorted by computational intensity, increasing from left to right.

the algorithm that minimizes the cost function,[8] and $\mathcal{S}_{\text{time}}$ for the algorithm that is actually the fastest among all candidates. For each test problem, we compare the number of FLOPs and the execution time of those two algorithms. Example problem a.14 was not used because the generated code ran out of memory. The results are shown in Fig. 10.4. In the single-threaded case, except for two problems the speedup of $\mathcal{S}_{\text{time}}$ over $\mathcal{S}_{\text{FLOPs}}$ is below 1.1×, and in all cases the relative cost of $\mathcal{S}_{\text{time}}$ compared to $\mathcal{S}_{\text{FLOPs}}$ is below 1.01×. With 24 threads, in 108 cases, $\mathcal{S}_{\text{time}}$ performs at most 1% more FLOPs than $\mathcal{S}_{\text{FLOPs}}$; there are only few cases where more FLOPs lead to a significantly lower execution time. The speedup of $\mathcal{S}_{\text{time}}$ over $\mathcal{S}_{\text{FLOPs}}$ is below 1.5× in

8 If multiple algorithms have the same minimal cost, one of those algorithms is selected as $\mathcal{S}_{\text{FLOPs}}$ at random.

120 cases. It can be concluded that for the kind of problems that Linnea solves, the number of FLOPs is often a good indicator for the execution time and never entirely unreliable. This is especially true when the code is executed with one thread. In addition, it should be noted that compared to the speedups that Linnea achieves, the loss of performance due to the inaccuracy of FLOPs as a cost function is in most cases small. Most of the cases where the cost function is inaccurate are a result of not considering the efficiency of the kernels. Two examples where the cost function is particularly off follow.

As a first example, we consider the randomly generated test problem

$$X := M_1 \left(M_2^T M_3 M_4\right)^{-1} M_5,$$

with $M_1 \in \mathbb{R}^{650\times1250}$, M_2 and $M_3 \in \mathbb{R}^{1700\times1250}$, $M_4 \in \mathbb{R}^{1250\times1250}$, and $M_5 \in \mathbb{R}^{1250\times1550}$; all matrices are full. As a first step, both $\mathcal{S}_{\text{time}}$ and $\mathcal{S}_{\text{FLOPs}}$ compute $Z_1 := M_2^T M_3$, yielding $X := M_1(Z_1 M_4)^{-1} M_5$. At this point, in $\mathcal{S}_{\text{FLOPs}}$ the LU factorization is applied to both Z_1 and M_4. After distributing the inverse, the problem becomes the matrix product $X := M_1 U_2^{-1} L_2^{-1} P_2 U_1^{-1} L_1^{-1} P_1 M_5$. The computation of this product, which is done from left to right, involves four calls to the TRSM kernel for the solution of triangular linear systems.[9] In contrast, $\mathcal{S}_{\text{time}}$ only uses one LU factorization and two triangular solves, but performs one additional matrix-matrix product: Instead of factoring Z_1 and M_4, those two matrices are multiplied together. After applying the LU factorization to the result of this product and distributing the inverse, $X := M_1 U_3^{-1} L_3^{-1} P_3 M_5$ is obtained. This product is again computed from left to right. This algorithm requires about 4% more FLOPs, but when executed with 24 threads is 27% faster than the algorithm with the minimum number of FLOPs. While both the product $Z_1 M_4$, with $Z_1, M_4 \in \mathbb{R}^{1250\times1250}$, as well as one LU factorization (1250×1250) and two triangular solves (with operands of size 650×1250 and 1250×1250) require almost the same number of FLOPs, the matrix-matrix product achieves higher efficiency and is thus faster.

As a second example, we look at the randomly generated problem

$$X := M_1 M_2^T + M_3 M_3^T + M_4^T + M_5^T,$$

with M_1 and $M_2 \in \mathbb{R}^{1100\times1800}$ $M_3 \in \mathbb{R}^{1100\times1150}$, and M_4 as well as $M_5 \in \mathbb{R}^{1100\times1100}$. Matrices M_4 and M_5 are upper triangular, all others are full. In $\mathcal{S}_{\text{FLOPs}}$, the product $M_3 M_3^T$ is computed with the SYRK kernel that makes use of symmetry. Since only half of the output matrix is stored, a storage format conversion is necessary to use this matrix in the following computations. In $\mathcal{S}_{\text{time}}$, the same product is computed with a call to GEMM. While this choice requires more FLOPs, it makes the storage format conversion unnecessary. The

9 The actual order of the kernel calls in $\mathcal{S}_{\text{FLOPs}}$ is slightly different; we are describing a different order here because it is easier to follow. The solution that uses the order described here has almost the same execution time as $\mathcal{S}_{\text{FLOPs}}$.

resulting algorithm performs 24% more FLOPs, but is around 27% faster with 24 threads. Again, this difference is caused by the higher parallel efficiency of GEMM compared to SYRK for matrices of the same size, but also by the storage format conversion.

The two examples discussed above are exceptions; for most of our test cases, the number of FLOPs is quite an accurate cost function. In general, while FLOPs as a cost function are most likely not good enough to compete with a human expert who is given sufficient time, they are good enough to outperform other languages and libraries, and to find a solution that is close to the optimum in a fraction of the time a human expert would require. In this regard, Linnea is similar to other compilers, for example for languages such as C: While the assembly code generated by those compilers is inferior to assembly code written by human experts, the increased productivity of the user usually outweighs the comparably small loss of performance. In cases where performance is critical, similar to how it is done in the manual development of code, a user can benchmark different algorithms generated by Linnea to find the optimal one. In this case, Linnea still makes it possible to save a significant implementation effort.

10.5 INFLUENCE OF THE HARDWARE

To investigate the influence of the hardware on the experiments discussed above, we repeated them on the more recent Skylake processor. The most important difference for the experiments is that on the Skylake processor, Turbo Boost was enabled and the experiments were run with non-exclusive access to the machine, leading to more noise in the measurements.

QUALITY OF THE GENERATED CODE With one thread, across all implementations the Skylake processor yields a median speedup of 1.4× over the Haswell processor. In the multi-threaded case, the median speedup of the execution with 12 threads on Skylake relative to the execution with 24 threads on Haswell is 1.2×. There are no substantial differences in the speedups of Linnea over the other languages and libraries.

QUALITY OF THE COST FUNCTION On the Skylake processor, the quality of the cost function decreases a bit with one thread. Specifically, the speedups of $\mathcal{S}_{\text{time}}$ over $\mathcal{S}_{\text{FLOPs}}$ are larger compared to Haswell. The number of cases where the speedup is 1×, that is, $\mathcal{S}_{\text{time}}$ and $\mathcal{S}_{\text{FLOPs}}$ coincide, is almost unchanged with 31 cases on Haswell and 30 cases on Skylake. There are 21 cases on Haswell in which the speedup is above 1.01×, compared to 83 cases on Skylake. In addition, on Haswell there are only two cases in which the speedup is above 1.1×;

on Skylake, there are 28 cases. The increased speedups with one thread are caused by the larger noise on the Skylake processor. In contrast, in the multi-threaded case there are no noticeable differences because the use of a single NUMA region on Skylake reduces the interference caused by other threads.

Despite the difference in the speedups, the relative costs are very similar on both processors. The reason is that the differences in the execution time due to the noise are still small compared to the differences in the execution time caused by the different costs of the algorithms.

GENERATION TIME Both for the time to the best solution, as well as for the time to the first solution, the Skylake processor yields a median speedup of 1.3×. However, the faster generation time does not lead to improved solutions within the time limit of 30 minutes.

11 CONCLUSION

In this thesis, we presented Linnea, a compiler that translates high-level linear algebra problems to efficient sequences of high-performance kernels. With its mathematical input language, Linnea successfully combines the ease-of-use of high-level languages with a performance that comes close to what a human expert can achieve. This performance is achieved by applying several optimizations that are not used in other languages and libraries for linear algebra:

1. Algebraic identities such as associativity, commutativity, distributivity are used to rewrite expressions into different representations.

2. Linnea is able to make use of a large number of matrix properties to select the most suitable kernels. If necessary, the properties of intermediate operands are automatically inferred from the properties of the input operands as provided by the user.

3. Common subexpressions are eliminated, including transposed and inverted occurrences.

4. Instead of using black-box solvers for linear systems, Linnea directly applies factorization, enabling optimizations that are not possible otherwise.

5. Linnea uses the full functionality of most kernels. This includes both the mathematical operations they can compute, as well as their interface; unnecessary copies are avoided for kernels that overwrite their input operands, and specialized storage formats for operands with properties are taken advantage of.

Our experiments on randomly generated and application problems indicate that Linnea almost always outperforms Matlab, Julia, Eigen, and Armadillo, both with sequential and parallel execution. With its custom search algorithm and the use of constructive algorithms, Linnea is able to find a first solutions for all problems used in our experiments in less than one second, that is, much faster than a human expert. Given more time, Linnea finds increasingly better solutions. While the cost function currently used by Linnea, the number of FLOPs performed by an algorithm, is relatively simple, our experiments demonstrate that it is accurate enough to reliably find good solutions for mid- to large-scale linear algebra problems.

11.1 FUTURE WORK

In this section, we give an overview of possible directions for future work. As part of this overview, we point out the challenges with those extensions and outline some of the steps necessary to implement them. Two possible directions for future work are discussed in more detail: The support for variable operand sizes, and a reduction of the generation time.

11.1.1 *Support for Variable Operand Sizes*

The fact that Linnea requires the sizes of all input operands to be fixed is one of the major obstacles for the integration of Linnea into existing languages such as Julia. The reason is that in most languages, the size of a matrix is usually not known at compile-time. Support for variable operand sizes can be added to Linnea as follows: The current cost function that counts the number of FLOPs is replaced with a symbolic cost function that describes the cost of a sequence of kernels as a function of the operand sizes. With the set of kernels that is currently supported, the number of FLOPs performed by a sequence can be described as a multivariate polynomial.

The main challenge with such a symbolic cost function is to identify the optimal solution. Quite likely there is no single best solution for all operand sizes. Instead, the optimal sequence of kernels depends on the operand sizes. Thus, it is necessary to solve inequalities over multivariate polynomials to determine for which operand sizes which sequence is optimal. Then, case distinctions can be used in the generated code to select the optimal sequence based on the operand sizes at runtime.

While the cost of solving such inequalities is exponential in the number of variables in general [24], in case of Linnea there are several constraints that may simplify the problem:

1. In application problems, the number of distinct operand sizes is usually small. Out of the 25 application problems in App. a, one problem has only one operand size, 18 problems have two distinct operand sizes, and six problems have three sizes. As a result, the number of variables in the polynomials is in practice small.

2. Since kernels have at most cubic complexity, the degree of the polynomials is at most three.

3. Since many kernels have similar costs, the number of distinct monomials that can appear in the polynomials is relatively small.

4. Since the variables represent operand sizes, their values are limited to integers larger or equal to one.

5. It is not uncommon that there are additional inequalities over variables that either follow from the application, or from the expression itself. As an example, in the least squares problem $b := (X^T X)^{-1} X^T y$ where X is of size $n \times m$, the linear system cannot be solved if $n < m$ because $X^T X$ is singular in that case.

If solving the polynomial inequalities is still too expensive, or the number of sequences of kernels is too large, heuristics can be used to remove sequences that are likely suboptimal. Such heuristics could for example only consider the asymptotic complexity, or compute the values of the polynomials at some points.

How a symbolic cost function affects the generation time is an open question. If there is a large number of solutions that are optimal at least for some operand sizes, the fraction of the search space that can be pruned decreases, resulting in a longer generation time. The generation time will also increase if it is not possible to extend the constructive algorithms to work with a symbolic cost function.

11.1.2 *Reduced Generation Time*

While there are already many cases in which Linnea finds the optimal solution in less than one seconds, there is still potential to decrease the generation time. To this end, it is important to note that the overall generation time is closely related to the quality of the solutions that are found early. The reason is that due to pruning, improving the early solutions reduces the size of the remaining search space and thus accelerates the discovery of further improved solutions.

The idea for improving the quality of the early solutions is to first apply those optimizations that lead to the largest reduction of the cost; first and foremost, this is the matrix chain algorithm in combination with the rewriting of expressions to different representations. While the current successor generation order already gives priority to the matrix chain algorithm, this prioritization is only applied per representation. If an expression can be rewritten to different representations, multiple optimizations are applied to the first representation before the matrix chain algorithm is applied to the last one. Thus, if the last representation leads to the optimal solution, this successor generation order results in an unnecessarily long generation time. In addition, rewriting the input expressions after every generation step is not necessary for quickly finding a first solution.

These observations can be used to improve the early solutions as follows: A simplified, deterministic generation algorithm is used that only uses constructive algorithms and applies the cheapest necessary factorizations, without rewriting the expressions between generation steps. Initially, this simplified algorithm is applied to different representations of the input expression to generate a first set of solutions. In a second phase, common subexpression elimination is performed

on the different representations of the input expression before the application of the simplified algorithm. After those steps, quite likely a solution has been found that has a cost relatively close to that of the optimal solution. Only then, the remaining optimizations are applied, and expressions are rewritten between the application of generation steps.

While the constructive algorithms make it possible to quickly find good solutions, they have the drawback that they cannot make use of kernels such as $A^T B + C$. This limitation can be alleviated without an exhaustive exploration of the full search space by also applying optimizations on the generated sequences of kernels. Specifically, peephole optimizations can be used to replace sequences of kernels that can be computed with a single kernel. As an example, if X is not used again later, the two kernel calls

$$X \leftarrow A^T B$$
$$Y \leftarrow X + C$$

can be replaced with a single GEMM call that computes $A^T B + C$. One of the challenges with such peephole optimizations is that due to the complexity of BLAS and LAPACK kernels, the number of sequences of kernels that can be replaced is large. Manually defining all those sequences is impractical. Fortunately, Linnea can be used to generate those sequences as follows: As input, all operations that can be computed with a single kernel are used, for example $AB + C$ or $\alpha AB^T + \beta C$. Linnea then generates all possible sequences of two or more kernels that compute those operations. For $\alpha AB^T + \beta C$, two possible sequences are shown below:

$$X \leftarrow B^T \qquad\qquad X \leftarrow AB^T$$
$$Y \leftarrow \alpha AX \qquad\qquad Y \leftarrow \alpha X$$
$$Z \leftarrow Y + \beta C \qquad\qquad Z \leftarrow Y + \beta C$$

Every generated sequence is a pattern that can be replaced with the operation that was used as input.

Especially for patterns that consist of more than two kernels, the detection in a sequence of kernels can become difficult because the kernels in the patterns can appear in different orders and interleaved with other kernels. For this reason, it might be beneficial to convert the sequence to a dependency graph and replace the patterns in the graph, similar to the optimizations performed by DxTer [93].

11.1.3 *Other Future Work*

There are several other possible directions for future work to extend the scope and features of Linnea:

LOOPS It is not uncommon that linear algebra problems involve loops. They can appear both explicitly, as for example in an iterative algorithm that repeatedly performs the same operation, or implicitly, for instance as a sum over a matrix expression such as $x := \sum_i Av_i$. With loops, new optimizations such as loop-invariant code motion become possible. In addition, there can be cases where operations that appear in a loop can be combined to a single operation. Once such case is $x_i := Av_i$, where both x_i and v_i change with every iteration. If those vectors are respectively combined to matrices X and V, where each column is a vector x_i or v_i, the loop can be transformed to a single matrix-matrix product $X := AV$.

COMPLEX LINEAR ALGEBRA At present, Linnea only supports real-valued linear algebra. Adding support for complex linear algebra is relatively simple; while both the inference of properties as well as the rewriting of expressions needs to be extended, the necessary extensions are very similar to existing code and algorithms.

SPARSE LINEAR ALGEBRA Considering that many application problems involve sparse matrices, support for sparse linear algebra would be a useful addition to Linnea. The main challenge with sparse linear algebra is that estimating the cost of the kernels becomes more difficult. If iterative solvers are used, another challenge consists in the optimal selection of a preconditioner and iterative solver for a given linear system, which is known to be a difficult problem [13, 97].

MATRIX FUNCTIONS The set of operations currently supported by Linnea is relatively limited. In order to extend the set of problems that can be solved with Linnea, it would be useful to add matrix functions such as the logarithm, exponentiation, trace, and determinant. With matrix functions, new ways of rewriting expressions become possible. In order to take advantage of those rewritings for the generation of algorithms, additional rewrite functions, representation, and/or tricks are necessary.

OPERANDS WITH BLOCK STRUCTURE In several application problems, operands exhibit block structure. Especially when some blocks are zero, this structure can be exploited to save computations. In order to take advantage of the block structure, a mechanism is necessary to decompose expressions that contain operands with block structure into sets of expressions that operate on the blocks, similar to how expressions are decomposed in the FLAME methodology [14, 59]. In addition, the code generation needs to be updated to support strided arrays.

PARALLELISM Linnea is currently limited to a single parallelization scheme; the sequential execution of multi-threaded kernels. For the future, we plan to investigate different methods of parallelization, such as algorithms by blocks, the concurrent execution of kernels, and offloading to accelerators. These schemes can also be combined, for instance by allowing the concurrent execution of multi-threaded kernels. In that case, the cost function could be used to find a good schedule.

IMPROVED COST FUNCTION There are several ways in which the cost function can be improved. It could be made more accurate by integrating the expected efficiency and scalability of kernels, or by considering caching effects, for example with approaches such as [102] and [70]. The cost function could also be extended to include for example the amount of memory used by an algorithm, or some measure of the numerical stability. Unfortunately, as discussed in Sec. 4.5, both performance modeling and prediction as well as automatic stability analysis are difficult problems.

INTERFACE TO CODE GENERATORS For problems that are bandwidth bound, including small-scale problems, the approach used by Linnea to break down the input expression into a sequence of kernels is frequently suboptimal. Instead, fusing operations and avoiding intermediate operands is one of the most important optimizations. Those optimizations are implemented by code generators such as BTO BLAS [117] and LGen [121] that target bandwidth bound and small-scale operations. However, they do not make use of algebraic identities to rewrite the input expression and explore different algorithms.

The advantages of both approaches can be combined by incorporating code generators for bandwidth bound operations into Linnea as follows: Similar to how the constructive algorithms are used for specific types of subexpressions that fulfill certain criteria, Linnea could apply code generators to subexpressions that are suitable as input for those generators, for example sums of an arbitrary number of matrices.

A second use-case of code generators (that is not limited to bandwidth bound operations) is the on-demand generation of kernels, for example for combinations of properties that are not supported by BLAS and LAPACK. This can be implemented by defining patterns for operations such as $A^T B + C$ or $A^{-1} B$, without any constraints regarding the properties. Whenever a match for one of those operations is found, the code generator is used to generate a kernel that is able to make use of the properties of the input operands.

INTEGRATION INTO EXISTING SOFTWARE In order to make Linnea easily accessible to a broad audience, it would be useful to integrate it

into existing languages. How this can be achieved highly depends on the language in question. In case of Julia, Linnea could be integrated in the form of a package that relies on Julia's powerful metaprogramming features. For C++, syntax plugins as described in [42] could be used. To integrate the optimizations that Linnea applies into a more general compiler infrastructure, the MLIR framework [86] could be used. Depending on the target language, it might be beneficial or even necessary to reimplement Linnea in the corresponding language.

ADDITIONAL OUTPUT LANGUAGES AND LIBRARIES In addition to generating Julia code that uses BLAS and LAPACK kernels, it would be useful to support both additional languages, for example C, and alternative libraries such as BLIS. The overall structure of the code generation remains the same for C. However, the implementation of efficient storage format conversions and code snippets for operations not supported by libraries can be laborious in lower-level languages.

APPENDIX

A

EXAMPLE PROBLEMS

In this chapter, we list the 25 application problems used in the experiments. The following properties are used: diagonal (DI), lower/upper triangular (LT/UT), symmetric positive definite (SPD), symmetric positive semi-definite (SPSD), symmetric (SYM). The identity matrix of size $n \times n$ is denoted by I_n. Unless otherwise noted, matrices are assumed to have full rank.

A.1 STANDARD LEAST SQUARES

$$b := \left(X^T X\right)^{-1} X^T y$$

$X \in \mathbb{R}^{n \times m}$; $y \in \mathbb{R}^n$; $b \in \mathbb{R}^m$;
$n = 2500$; $m = 500$

A.2 GENERALIZED LEAST SQUARES

$$b := \left(X^T M^{-1} X\right)^{-1} X^T M^{-1} y$$

$M \in \mathbb{R}^{n \times n}$, SPD; $X \in \mathbb{R}^{n \times m}$; $y \in \mathbb{R}^n$; $b \in \mathbb{R}^m$;
$n = 2500$; $m = 500$

A.3 OPTIMIZATION [123]

$$x := W\left(A^T \left(AWA^T\right)^{-1} b - c\right)$$

$A \in \mathbb{R}^{m \times n}$; $W \in \mathbb{R}^{n \times n}$, DI, SPD; $b \in \mathbb{R}^m$; $c \in \mathbb{R}^n$; $x \in \mathbb{R}^n$;
$n = 2000$; $m = 1000$

A.4 OPTIMIZATION [123]

$$x_f := WA^T \left(AWA^T\right)^{-1} (b - Ax)$$
$$x_o := W\left(A^T \left(AWA^T\right)^{-1} Ax - c\right)$$

$A \in \mathbb{R}^{m \times n}$; $W \in \mathbb{R}^{n \times n}$, DI, SPD; $b \in \mathbb{R}^m$; $c \in \mathbb{R}^n$; $x_f \in \mathbb{R}^n$; $x_o \in \mathbb{R}^n$;
$n = 2000$; $m = 1000$

A.5 SIGNAL PROCESSING [31]

$$x := \left(A^{-T} B^T B A^{-1} + R^T L R\right)^{-1} A^{-T} B^T B A^{-1} y$$

$A \in \mathbb{R}^{n \times n}$; $B \in \mathbb{R}^{n \times n}$; $R \in \mathbb{R}^{(n-1) \times n}$, UT; $L \in \mathbb{R}^{(n-1) \times (n-1)}$, DI;
$y \in \mathbb{R}^n$; $x \in \mathbb{R}^n$;
$n = 2000$

A.6 TRIANGULAR MATRIX INVERSION [15]

$$
\begin{aligned}
X_{10} &:= L_{10} L_{00}^{-1} \\
X_{20} &:= L_{20} + L_{22}^{-1} L_{21} L_{11}^{-1} L_{10} \\
X_{11} &:= L_{11}^{-1} \\
X_{21} &:= -L_{22}^{-1} L_{21}
\end{aligned}
$$

$L_{00} \in \mathbb{R}^{n \times n}$, LT; $L_{11} \in \mathbb{R}^{m \times m}$, LT; $L_{22} \in \mathbb{R}^{k \times k}$, LT; $L_{10} \in \mathbb{R}^{m \times n}$;
$L_{20} \in \mathbb{R}^{k \times n}$; $L_{21} \in \mathbb{R}^{k \times m}$; $X_{10} \in \mathbb{R}^{m \times n}$; $X_{20} \in \mathbb{R}^{k \times n}$; $X_{11} \in \mathbb{R}^{m \times m}$;
$X_{21} \in \mathbb{R}^{k \times m}$;
$n = 2000$; $m = 200$; $k = 2000$

A.7 ENSEMBLE KALMAN FILTER [100]

$$X^a := X^b + \left(B^{-1} + H^T R^{-1} H\right)^{-1} \left(Y - H X^b\right)$$

$B \in \mathbb{R}^{n \times n}$ SPSD; $H \in \mathbb{R}^{m \times n}$; $R \in \mathbb{R}^{m \times m}$ SPSD; $Y \in \mathbb{R}^{m \times N}$; $X^b \in \mathbb{R}^{n \times N}$; $X^a \in \mathbb{R}^{n \times N}$;
$N = 200$; $n = 2000$; $m = 2000$

A.8 ENSEMBLE KALMAN FILTER [100]

$$\delta X := \left(B^{-1} + H^T R^{-1} H\right)^{-1} H^T R^{-1} \left(Y - H X^b\right)$$

$B \in \mathbb{R}^{n \times n}$ SPSD; $H \in \mathbb{R}^{m \times n}$; $R \in \mathbb{R}^{m \times m}$ SPSD; $Y \in \mathbb{R}^{m \times N}$; $X^b \in \mathbb{R}^{n \times N}$; $\delta X \in \mathbb{R}^{n \times N}$;
$N = 200$; $n = 2000$; $m = 2000$

A.9 ENSEMBLE KALMAN FILTER [100]

$$\delta X := X(HX)^T \left(R + HX(HX)^T\right)^{-1} \left(Y - H X^b\right)$$

$H \in \mathbb{R}^{m \times n}$; $R \in \mathbb{R}^{m \times m}$ SPSD; $Y \in \mathbb{R}^{m \times N}$; $X \in \mathbb{R}^{n \times n}$ LT; $X^b \in \mathbb{R}^{n \times N}$;
$\delta X \in \mathbb{R}^{n \times N}$;
$N = 200$; $n = 5000$; $m = 1000$

A.10 IMAGE RESTORATION [126]

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

$H \in \mathbb{R}^{m \times n}$; $y \in \mathbb{R}^m$; $v_{k-1} \in \mathbb{R}^n$; $u_{k-1} \in \mathbb{R}^n$; $x_k \in \mathbb{R}^n$; $\lambda > 0$; $\sigma > 0$;
$n = 5000$; $m = 1000$

A.11 IMAGE RESTORATION [126]

$$
\begin{aligned}
H^{\dagger} &:= H^{T}\left(HH^{T}\right)^{-1} \\
y_k &:= H^{\dagger}y + \left(I_n - H^{\dagger}H\right)x_k
\end{aligned}
$$

$H \in \mathbb{R}^{m\times n}$; $H^{\dagger} \in \mathbb{R}^{n\times m}$; $y \in \mathbb{R}^{m}$; $x_k \in \mathbb{R}^{n}$; $y_k \in \mathbb{R}^{n}$;
$n = 5000$; $m = 1000$

A.12 RANDOMIZED MATRIX INVERSION [55]

$$
\begin{aligned}
\Lambda &:= S\left(S^{T}AWA^{T}S\right)^{-1}S^{T} \\
X_{k+1} &:= X_k + WA^{T}\Lambda\left(I_n - AX_k\right)
\end{aligned}
$$

$W \in \mathbb{R}^{n\times n}$, SPD; $S \in \mathbb{R}^{n\times q}$; $A \in \mathbb{R}^{n\times n}$; $X_k \in \mathbb{R}^{n\times n}$; $\Lambda \in \mathbb{R}^{n\times n}$;
$X_{k+1} \in \mathbb{R}^{n\times n}$;
$n = 5000$; $q = 500$

A.13 RANDOMIZED MATRIX INVERSION [55]

$$
\begin{aligned}
\Lambda &:= S\left(S^{T}A^{T}WAS\right)^{-1}S^{T} \\
X_{k+1} &:= X_k + \left(I_n - X_kA^{T}\right)\Lambda A^{T}W
\end{aligned}
$$

$W \in \mathbb{R}^{n\times n}$, SPD; $S \in \mathbb{R}^{n\times q}$; $A \in \mathbb{R}^{n\times n}$; $X_k \in \mathbb{R}^{n\times n}$; $\Lambda \in \mathbb{R}^{n\times n}$;
$X_{k+1} \in \mathbb{R}^{n\times n}$;
$n = 5000$; $q = 500$

A.14 RANDOMIZED MATRIX INVERSION [55]

$$
\begin{aligned}
\Lambda &:= S\left(S^{T}AWAS\right)^{-1}S^{T} \\
\Theta &:= \Lambda AW \\
M_k &:= X_kA - I_n \\
X_{k+1} &:= X_k - M_k\Theta - (M_k\Theta)^{T} + \Theta^{T}(AX_kA - A)\Theta
\end{aligned}
$$

$W \in \mathbb{R}^{n\times n}$, SPD; $S \in \mathbb{R}^{n\times q}$; $A \in \mathbb{R}^{n\times n}$, SYM; $\Lambda \in \mathbb{R}^{n\times n}$, SYM;
$\Theta \in \mathbb{R}^{n\times n}$; $X_k \in \mathbb{R}^{n\times n}$, SYM; $M_k \in \mathbb{R}^{n\times n}$; $X_{k+1} \in \mathbb{R}^{n\times n}$;
$n = 5000$; $q = 500$

A.15 RANDOMIZED MATRIX INVERSION [55]

$$
\begin{aligned}
X_{k+1} &:= S\left(S^{T}AS\right)^{-1}S^{T} \\
&\quad + \left(I_n - S\left(S^{T}AS\right)^{-1}S^{T}A\right)X_k\left(I_n - AS\left(S^{T}AS\right)^{-1}S^{T}\right)
\end{aligned}
$$

$A \in \mathbb{R}^{n\times n}$, SPD; $S \in \mathbb{R}^{n\times q}$; $X_k \in \mathbb{R}^{n\times n}$; $X_{k+1} \in \mathbb{R}^{n\times n}$;
$n = 5000$; $q = 500$

A.16 STOCHASTIC NEWTON [23]

$$B_k := \frac{k}{k-1} B_{k-1} \Big(I_n - A^T W_k \left((k-1) I_l + W_k^T A B_{k-1} A^T W_k \right)^{-1} W_k^T A B_{k-1} \Big)$$

$W_k \in \mathbb{R}^{m \times l}$; $A \in \mathbb{R}^{m \times n}$; $B_{k-1} \in \mathbb{R}^{n \times n}$, SPD; $B_k \in \mathbb{R}^{n \times n}$; $k > 0$;
$l = 625$; $n = 1000$; $m = 5000$

A.17 STOCHASTIC NEWTON [23]

$$B_1 := \frac{1}{\lambda_1} \left(I_n - A^T W_1 \left(\lambda_1 I_l + W_1^T A A^T W_1 \right)^{-1} W_1^T A \right)$$

$W_1 \in \mathbb{R}^{m \times l}$; $A \in \mathbb{R}^{m \times n}$; $B_1 \in \mathbb{R}^{n \times n}$; $\lambda_1 > 0$;
$l = 625$; $n = 1000$; $m = 5000$

A.18 TIKHONOV REGULARIZATION [50]

$$x := \left(A^T A + \Gamma^T \Gamma \right)^{-1} A^T b$$

$A \in \mathbb{R}^{n \times m}$; $\Gamma \in \mathbb{R}^{m \times m}$; $b \in \mathbb{R}^n$; $x \in \mathbb{R}^m$;
$n = 3000$; $m = 200$

A.19 TIKHONOV REGULARIZATION [50]

$$x := \left(A^T A + \alpha^2 I \right)^{-1} A^T b$$

$A \in \mathbb{R}^{n \times m}$; $b \in \mathbb{R}^n$; $x \in \mathbb{R}^m$; $\alpha > 0$;
$n = 3000$; $m = 200$

A.20 GENERALIZED TIKHONOV REGULARIZATION

$$x := \left(A^T P A + Q \right)^{-1} \left(A^T P b + Q x_0 \right)$$

$P \in \mathbb{R}^{n \times n}$, SPSD; $Q \in \mathbb{R}^{m \times m}$, SPSD; $x_0 \in \mathbb{R}^m$; $A \in \mathbb{R}^{n \times m}$; $b \in \mathbb{R}^n$;
$x \in \mathbb{R}^m$;
$n = 3000$; $m = 200$

A.21 GENERALIZED TIKHONOV REGULARIZATION

$$x := x_0 + \left(A^T P A + Q \right)^{-1} A^T P (b - A x_0)$$

$P \in \mathbb{R}^{n \times n}$, SPSD; $Q \in \mathbb{R}^{m \times m}$, SPSD; $x_0 \in \mathbb{R}^m$; $A \in \mathbb{R}^{n \times m}$; $b \in \mathbb{R}^n$;
$x \in \mathbb{R}^m$;
$n = 3000$; $m = 200$

A.22 LMMSE ESTIMATOR [77]

$$x_{out} := C_X A^T \left(A C_X A^T + C_Z\right)^{-1} (y - Ax) + x$$

$A \in \mathbb{R}^{m \times n}$; $C_X \in \mathbb{R}^{n \times n}$, SPSD; $C_Z \in \mathbb{R}^{m \times m}$, SPSD; $x \in \mathbb{R}^n$; $y \in \mathbb{R}^m$;
$x_{out} \in \mathbb{R}^n$;
$n = 2000$; $m = 1500$

A.23 LMMSE ESTIMATOR [77]

$$x_{out} := \left(A^T C_Z^{-1} A + C_X^{-1}\right)^{-1} A^T C_Z^{-1} (y - Ax) + x$$

$A \in \mathbb{R}^{m \times n}$; $C_X \in \mathbb{R}^{n \times n}$, SPSD; $C_Z \in \mathbb{R}^{m \times m}$, SPSD; $x \in \mathbb{R}^n$; $y \in \mathbb{R}^m$;
$x_{out} \in \mathbb{R}^n$;
$n = 2000$; $m = 1500$

A.24 LMMSE ESTIMATOR [77]

$$\begin{aligned} K_{t+1} &:= C_t A^T \left(A C_t A^T + C_z\right)^{-1} \\ x_{t+1} &:= x_t + K_{t+1}(y - A x_t) \\ C_{t+1} &:= (I - K_{t+1} A) C_t \end{aligned}$$

$A \in \mathbb{R}^{m \times n}$; $C_t \in \mathbb{R}^{n \times n}$, SPSD; $C_Z \in \mathbb{R}^{m \times m}$, SPSD; $x_t \in \mathbb{R}^n$; $y \in \mathbb{R}^m$;
$K_{t+1} \in \mathbb{R}^{n \times m}$; $x_{t+1} \in \mathbb{R}^n$; $C_{t+1} \in \mathbb{R}^{n \times n}$;
$n = 400$; $m = 400$

A.25 KALMAN FILTER [78]

$$\begin{aligned} K_k &:= P_{k-1} H_k^T \left(H_k P_{k-1} H_k^T + R_k\right)^{-1} \\ P_k &:= \left(I - K_k H_k\right) P_{k-1} \\ x_k &:= x_{k-1} + K_k(z_k - H_k x_{k-1}) \end{aligned}$$

$P_{k-1} \in \mathbb{R}^{n \times n}$, SPD; $H_k \in \mathbb{R}^{m \times n}$; $R_k \in \mathbb{R}^{m \times m}$, SPSD; $x_{k-1} \in \mathbb{R}^n$;
$z_k \in \mathbb{R}^m$; $K_k \in \mathbb{R}^{n \times m}$; $P_k \in \mathbb{R}^{n \times n}$; $x_k \in \mathbb{R}^n$;
$n = 400$; $m = 400$

b

DESCRIPTION OF KERNELS

Linnea is build in a way such that the set of available kernels can easily be extended or changed, both to add kernels for additional operations, and to generate code with kernels from different libraries. The challenge with such an extensibility is that the interface of BLAS and LAPACK kernels is relatively complex, and that many kernels allow to compute families of related operations. As an example, the TRSM kernel solves triangular linear systems $A^{-1}B$ or BA^{-1}, where A can either be upper or lower triangular. As mentioned in Sec. 5.1, in Linnea each of those operations is represented by a different pattern.

In order to simplify the process of making kernels available to Linnea, Linnea offers a specification language that allows to describe the full functionality of a kernel such as TRSM in a concise manner, without the exhaustive enumeration of all distinct operations that can be computed with a single kernel. Instead, from this specification the patterns that represent the different operations that can be computed with a given kernel are generated automatically. This language is embedded into Python and implemented through Python objects. The same language can also be used for code snippets that implement operations not supported by libraries. While this language was developed for BLAS and LAPACK kernels, it generalizes to a much larger set of BLAS-like kernels that have a similar interface.

Since kernels for matrix factorizations are quite different from other kernels, there is a separate description language for factorizations.

The description language for kernels (excluding factorizations) is described in App. b.1, followed by the language for factorizations in App. b.2.

b.1 KERNELS

In this section, we give an overview of the specification language for kernels. A kernel is described by an instance of the `KernelDescription` class. The constructor of this class takes several objects as input that describe different aspects of the kernel. Those objects are described below.

b.1.1 *Signature*

The signature is represented by a Python template string. The argument names are identified by a prefixed `$` symbol. As an example, the signature of the Julia wrapper for the GEMM kernel is

```
gemm!($transA, $transB, $alpha, $A, $B, $beta, $C).
```

The meaning the arguments as well as their possible values are specified with other objects that are discussed below. For operations not supported by libraries, instead of the signature of a function, it is possible to directly use a code snippet. For instance, the following code snippet implements the product of a full and a diagonal matrix where the diagonal matrix is represented as a vector of the diagonal elements:

```
for i = 1:size($B, 2);
  view($B, :, i)[:] .*= $A;
end;
```

B.1.2 *Operation*

The mathematical operation that is computed by a kernel is described by the `Operation` class that takes as input a symbolic MatchPy expression. The operation that the AXPY kernel computes is for example specified as

```
Operation(Plus(Times(alpha, x), y)).
```

The operand names have to be the same names as the corresponding arguments in the signature. For kernels that allow operands to optionally be transposed, there are placeholder operators. With those operators, the operation that the GEMM kernel computes is described as

```
Operation(
  Plus(Times(alpha, Op1(A), Op2(B)), Times(beta, C))).
```

The placeholder operators `Op1` and `Op2` are either replaced with the transposition or identity operator. In case of the latter, this means that the placeholder is removed. For kernels such as TRSM that allow to compute different operations, it is possible to use a `OperationKV` object that takes as input the name of an argument together with a dictionary. The dictionary maps argument values to different expressions:

```
OperationKV("side",
 {"L": Times(alpha, Op1(A), B),
  "R": Times(alpha, B, Op1(A))}
)
```

In this case, if the `side` argument has the value `'L'`, the operation represented by the expression `Times(alpha, Op1(A), B)` is computed; if `side` is set to `'R'`, the operation is `Times(alpha, B, Op1(A)`. The operands that appear in the expressions are symbolic operands that have to be defined before the definition of the `KernelDescription` object. During the generation of the patterns, those operands are

replaced with variables. Properties that are fixed for all operations computed by a given kernel are specified by assigning those properties to the respective operands. For instance, the property `symmetric` for the operand `A` of the SYMM kernel is set with

```
A.set_property(symmetric).
```

B.1.3 *Variants*

The `KernelDescription` class takes a list of an arbitrary number of `KernelVariant` objects. Those objects further describe the family of operations that can be computed by the kernel. There are three different subclasses of the `KernelVariant` class:

PLACEHOLDER OPERATORS The `OperatorKV` class specifies the possible values of placeholder operators. The constructor of the this class takes three arguments: The name of the corresponding argument, a dictionary that maps argument values to operators, and the placeholder operator. For instance,

```
OperatorKV("transA", {"N": Identity, "T": Transpose}, Op1)
```

describes that if the value of the argument `transA` is `N`, then the placeholder operator `Op1` is replaced with the identity function and subsequently removed. Alternatively, if `transA` is `T`, then `Op1` is the transposition.

DEFAULT VALUES As mentioned in Sec. 5.1, the pattern matching implemented in MatchPy is not able to automatically insert the neutral element of multiplication or addition for a variable that cannot match anything else. For this reason, the DefaultValueKV class allows to specify default values for operands. The specification of default values consists of two parts: The operand, and a list of the default values. An example for the argument `beta` of the GEMM kernel is shown below:

```
DefaultValueKV(beta,
  [ConstantScalar(0.0), ConstantScalar(1.0)]).
```

When the different patterns are generated from a `KernelDescription` object, the operand is replaced with a default value, and the resulting pattern is simplified (see Sec. 7.1.1) For instance, if `beta` is `0.0`, the pattern for the GEMM kernel is simplified to

```
Times(alpha, Op1(A), Op2(B))).
```

For each operand that has default values, one pattern is generated where the operand is not replaced with a default value.

PROPERTIES The `PropertyKV` class specifies a property of an input operand in case the property is determined by an argument. In the pattern generated for the kernel, this property is used a constraint for the respective operand. The constructor of the `PropertyKV` class takes three arguments: The name of the corresponding argument, a dictionary that maps argument values to properties, and the operand. As an example, in case of the TRSM kernel,

```
PropertyKV("uplo",
  {"U": upper_triangular, "L": lower_triangular}, A)
```

describes that if the value of the argument `uplo` is `L`, then `A` needs to have the property `lower_triangular`. Alternatively, if `uplo` is `U`, then `A` needs to have the property `upper_triangular`.

B.1.4 *Input and Output Operands*

The input operands are described by a list of `InputOperand` objects. The constructor of the `InputOperand` class takes the operand together with its required storage format as input. The storage formats are described in Sec. 9.2. The list of input operands of the SYMM kernel is for instance

```
InputOperand(alpha, full)
InputOperand(A, symmetric_triangular)
InputOperand(B, full)
InputOperand(beta, full)
InputOperand(C, full).
```

Similarly, the output operand is described by the `OutputOperand` class that takes the operand as well as its storage format as input. In case of SYMM, the output operand is described as

```
OutputOperand(C, full).
```

If the output operand is also an input operand, this means that the output of this kernel overwrites the memory location that initially contains the respective input operand.

B.1.5 *Additional Arguments*

Argument objects describe additional arguments that are not covered the objects described so far. There are two different types of arguments.

SIZE The `SizeArgument` class is used to specify the value of arguments that require the size of an operand. In addition, this class also defines the values that can be used in the cost function, which is described below. The `SizeArgument` class takes three inputs: The name of the argument, the operand, and an identifier that is either `"rows"`

or `"columns"`. Depending on the identifier, respectively the number of rows or columns of this operand is used as a value for the argument. As an example, the size arguments of the AXPY kernel are

```
SizeArgument("n", A, "columns")
SizeArgument("m", A, "rows").
```

In order to take into account the effect of placeholder operators, it is possible to use placeholder operators in `SizeArgument` objects. For instance, for the GEMM kernel, the following objects are used:

```
SizeArgument("m", Op1(A), "rows")
SizeArgument("n", Op2(B), "columns")
SizeArgument("k", Op1(A), "columns").
```

STORAGE FORMAT The `StorageFormatArgument` class is used to specify the value of arguments that describe the storage format of an operand. This class requires three inputs: The name of the argument, the name of the operand, and a dictionary that maps storage formats to arguments values. For the SYMM kernel, it is used as shown below:

```
StorageFormatArgument("uplo", A,
  {symmetric_lower: "L", symmetric_upper: "U"}).
```

B.1.6 *Cost Function*

The cost function is described by a Python function that computes the cost of a kernel call from the sizes of the input operands. For the operand sizes, the names can be used that have been defined with `SizeArgument` objects. Since the cost functions are relatively simple, they are usually implemented as lambda functions. The cost function of the GEMM kernel is for example

```
lambda m, n, k: 2*m*n*k.
```

B.1.7 *Options*

In addition to the specifications described so far, the `KernelDescription` class takes a set of options that influence the generation of the different patterns. At present, there are two such options:

TRANSPOSE The `transpose` option specifies that this kernel should also be used as a transposed kernel (see Sec. 5.4). With this option, transposed patterns are generated.

NO SIMPLIFICATIONS The `no_simplifications` option specifies that the patterns for this kernel should not be simplified. Usually, patterns are simplified to remove the identity function introduced

by placeholder operators, to push down the transpose in case of transposed kernels, and to remove default values. In some cases, those simplifications have undesired side effects. For instance, for the kernel that computes the product of a matrix with a scalar, both patterns αA and $A\alpha$ have to be generated. However, as a convention, scalars in matrix products are always sorted and moved to the left (see Sec. 7.1.1). Thus, if $A\alpha$ is simplified, it is converted to αA. The `no_simplifications` option exists to avoid that simplifications are applied in those cases.

Example b.1. As a complete example, the description of the GEMM kernel is shown in Fig. b.1. The description of a code snipped that computes the product of a full and a diagonal matrix is shown in Fig. b.2. ■

B.1.8 *Generation of Patterns*

From the `KernelDescription` objects, one pattern is generated for each distinct operation that can be computed with a given kernel. For the most part, this generation consists of the enumeration of all combinations of different alternatives specified by `KernelVariant` objects. In addition, as a preparation for the code generation, values are assigned to arguments that determine the operation that is computed. This process can be thought of as partial function application. As an example, consider the TRSM with the signature

```
trsm!($side, $uplo, $transA, $diag, $alpha, $A, $B).
```

For the operation $\alpha A^{-1}B$, where A is lower triangular, values are assigned to the first four arguments, resulting in

```
trsm!('L', 'L', 'N', 'N', $alpha, $A, $B).
```

While `A` and `B` are always set during the code generation, for those operations where a default value is used for `alpha`, this argument is already set during the generation of patterns. That is, for the operation $A^{-1}B$, the signature becomes

```
trsm!('L', 'L', 'N', 'N', 1.0, $A, $B).
```

Since operand sizes and storage formats depend on the actual input operands to the kernel, the corresponding arguments are always set during the code generation.

B.2 FACTORIZATIONS

Due to the differences between factorizations and all other kernels, a separate specification language exists for matrix factorizations. In this language, factorizations are described by the `FactorizationKernel` class.

```
A = Matrix("A")
B = Matrix("B")
C = Matrix("C")
alpha = Scalar("alpha")
beta = Scalar("beta")

gemm = KernelDescription(
 OperationKV(
  Plus(Times(alpha, Op1(A), Op2(B)), Times(beta, C))),
 [OperatorKV("transA",
   {"N": Identity, "T": Transpose}, Op1),
  OperatorKV("transB",
   {"N": Identity, "T": Transpose}, Op2),
  DefaultValueKV(alpha, [ConstantScalar(1.0)]),
  DefaultValueKV(beta,
   [ConstantScalar(0.0), ConstantScalar(1.0)])
 ],
 [InputOperand(alpha, full),
  InputOperand(A, full),
  InputOperand(B, full),
  InputOperand(beta, full),
  InputOperand(C, full),
 ],
 OutputOperand(C, full),
 lambda m, n, k: 2*m*n*k,
 "gemm!($transA, $transB, $alpha, $A, $B, $beta, $C)",
 [SizeArgument("m", Op1(A), "rows"),
  SizeArgument("n", Op2(B), "columns"),
  SizeArgument("k", Op1(A), "columns")
 ]
)
```

Figure b.1: Description of the GEMM kernel.

```
A = Matrix("A")
A.set_property(DIAGONAL)
A.set_property(SYMMETRIC)
B = Matrix("B")

diagmmr = KernelDescription(
 OperationKV(Times(B, A)),
 [],
 [InputOperand(A, diag_vec),
  InputOperand(B, full),
 ],
 OutputOperand(B, full),
 lambda m, n: m*n,
 """
 for i = 1:size($B, 2);
    view($B, :, i)[:] .*= $A[i];
 end;
 """,
 [SizeArgument("m", B, "rows"),
  SizeArgument("n", B, "columns")
 ],
 options={transpose}
)
```

Figure b.2: Description of a code snippet that computes the product of a full and a diagonal matrix.

In terms of functionality, the main differences between factorizations and other kernels are that factorizations have only one input operand, but multiple output operands, and some factorizations store multiple output operands in the same memory location. The interface of factorizations is less uniform and to some extent also more complicated: While most factorizations do have some arguments that affect their functionality, those arguments are very specific to the different factorizations. Thus, the semantics of those arguments cannot easily be described with a small number of relatively general `KernelVariant` objects the way it is done for other kernels. As a result, the specification language for factorizations has a lower level of abstraction, and only one operation can be described with a `FactorizationKernel` object. However, if a factorization kernel should be used in different ways, it is possible to use the same factorization with different arguments in multiple objects. This approach can also be used if the properties of output operands depend on the properties of input operands. An example of the use of multiple objects for one factorization is provided at the end of this section.

The input of the constructor of the `FactorizationKernel` class is described below. Since the specification of the signature, the input operands, the arguments, and the cost function is the same as for the `KernelDescription` class, the corresponding objects are omitted in this section.

B.2.1 *Pattern*

The operand that can be used as input for a factorization is described by a MatchPy `Pattern` object. This pattern optionally includes the constraints regarding the properties of this operand. The pattern used for the LU factorization is for instance

```
Pattern(A, PropertyConstraint("A", {NON_SINGULAR})),
```

where `A` is a variable that only matches matrices. The `PropertyConstraint` object denotes that `A` can only match operands that are non-singular.

B.2.2 *Output Expressions*

The output of a factorization is described by a symbolic expression. For the LU factorization, this expression is

```
Times(Transpose(P), L, U),
```

where `P`, `L`, and `U` are variables.

B.2.3 *Output Operands*

The output operands, that is, the factors produced by the factorization, are specified by a dedicated `OutputOperand` class for factorizations. This class takes five inputs:

1. The output operand.
2. The input operand that is overwritten. If no input operand is overwritten by a given output operand, this can be `None`.
3. The size of the output operand, as a tuple of argument names specified by `SizeArgument` objects.
4. A list of the properties of the operand.
5. The storage format of the output operand.

Since some factorizations store multiple output operands in the same memory location, it is allowed that multiple `OutputOperand` objects use the same input operand for the operand that is overwritten. The output operands of the LU factorizations are specified as

```
OutputOperand(L, A, ("n", "n"),
  [LOWER_TRIANGULAR, UNIT_DIAGONAL, NON_SINGULAR],
  lower_triangular_ud)
OutputOperand(U, A, ("n", "n"),
  [UPPER_TRIANGULAR, NON_SINGULAR],
  upper_triangular),
OutputOperand(P, None, ("n", "n"),
  [PERMUTATION],
  ipiv).
```

Both `L` and `U` overwrite `A`. This is possible because with the storage formats `lower_triangular_ud` and `upper_triangular`, both operands occupy disjoint parts of the same memory location.

Example b.2. The description of the GETRF kernel that computes the LU factorization is shown in Fig. b.3.

The GESVD kernel that computes the singular value decomposition $(U, S, V) \leftarrow A$ with $A = USV$ has two arguments that allow to specify which output operands are computed and where they are stored: Either U or V can overwrite the input operand A. Depending on the dimensions of A, either U or V is smaller than A, while the other one has the same size as A. Thus, it is beneficial to overwrite A with the operand that has the same size as A. To specify this, multiple `FactorizationKernel` objects are necessary, where different values are used for the arguments `jobu` and `jobvt` that determine which output operand overwrites A. In Linnea, separate objects are used for $m < n$, $m = n$, and $m > n$, where m is the number of rows and n the number of columns of A. As an example, for $m < n$, V overwrites A, so the output operands are

```
A = Wildcard.symbol("A", symbol_type=Matrix)
P = Wildcard.symbol("P")
L = Wildcard.symbol("L")
U = Wildcard.symbol("U")

getrf = FactorizationKernel(
   Pattern(A, PropertyConstraint("A", {NON_SINGULAR})),
   InputOperand(A, full),
   Times(Transpose(P), L, U),
   [OutputOperand(L, A, ("n", "n"),
     [LOWER_TRIANGULAR, UNIT_DIAGONAL, NON_SINGULAR],
     lower_triangular_ud),
    OutputOperand(U, A, ("n", "n"),
     [UPPER_TRIANGULAR, NON_SINGULAR],
     upper_triangular),
    OutputOperand(P, None, ("n", "n"),
     [PERMUTATION],
     ipiv)
   ],
   lambda n: 2/3*n**3,
   "($A, $P, info) = getrf!($A)",
   [SizeArgument("n", A, "rows")],
   )
```

Figure B.3: Description of the GETRF kernel.

```
OutputOperand(U, None, ("m", "m"),
  [ORTHOGONAL],
  full)
OutputOperand(S, None, ("m", "m"),
  [DIAGONAL],
  diag_vec)
OutputOperand(V, A, ("m", "n"),
  [ORTHOGONAL_ROWS],
  full)
```

together with the signature

```
($U, $S, _) = gesvd!('S', 'O', $A).
```

For $m > n$ on the other hand, U overwrites A, so the output operands are

```
OutputOperand(U, A, ("m", "n"),
  [ORTHOGONAL_COLUMNS],
  full)
OutputOperand(S, None, ("n", "n"),
  [DIAGONAL],
  diag_vec)
OutputOperand(V, None, ("n", "n"),
  [ORTHOGONAL],
  full)
```

with the signature

```
(_, $S, $V) = gesvd!('O', 'S', $A).
```

C

PROPERTIES BASED ON BANDWIDTH

A relatively large number of common matrix properties can be described in terms of a generalized notion of bandwidth. This includes for instance triangular, tridiagonal, and upper Hessenberg. The idea is to separately define the upper and lower bandwidth of a matrix as the number of non-zero off-diagonals above and below the main diagonal. With this notion of bandwidth, the inference of properties that can be described in terms of bandwidth can be done as follows: The properties of the input matrices as provided by the user are translated to the bandwidth. From the bandwidth of the operands, the bandwidth of the full expression is computed. Then, the properties of the expression are determined from its bandwidth. In contrast to a rule-based approach, the inference of properties based on bandwidth makes it possible to support a relatively large number of properties with a single, simple inference method.

In this section, we present this generalized notion of bandwidth and show how the bandwidth of expressions can be computed. We begin with the definition of the bandwidth:

Definition c.1 (Bandwidth [51, Section 1.2.1])**.** Let $A \in \mathbb{R}^{m \times n}$ be a matrix. The *bandwidth of* A is a tuple $b = (l, u)$ with

$$-n \leqslant l \leqslant m - 1 \tag{c.1}$$

$$-m \leqslant u \leqslant n - 1. \tag{c.2}$$

A has a *lower bandwidth* l if $i > j + l \Rightarrow a_{ij} = 0$ and an *upper bandwidth* u if $j > i + u \Rightarrow a_{ij} = 0$. In the following, the bandwidth of a matrix A is denoted by $\mathcal{B}(A)$. ∎

Intuitively, l and u specify the number of non-zero off-diagonals below and above the main diagonal, respectively. For instance, the matrix

$$\begin{pmatrix} \times & \times & \times & 0 \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \end{pmatrix}$$

has bandwidth $(0, 2)$; $\times$ denotes entries with arbitrary values. Tab. c.1 shows a list of properties that can be defined in terms of bandwidth.

The definition of bandwidth intentionally does not require a matrix to be square. The reason is that this notion of bandwidth naturally extends to non-square matrices. As a result, it provides an unambiguous generalization of all properties that can be defined in terms of bandwidth to non-square matrices. In addition, the bandwidth of a

Table c.1: Definition of properties in terms of bandwidth. Let $A \in \mathbb{R}^{m \times n}$ be a matrix with $\mathcal{B}(A) = (l, u)$.

Property	Definition
lower triangular	$u \leqslant 0$
upper triangular	$l \leqslant 0$
diagonal	$l = 0$ and $u = 0$
zero	$l + u + 1 \leqslant 0$
lower Hessenberg	$u \leqslant 1$
upper Hessenberg	$l \leqslant 1$
lower bidiagonal	$l \leqslant 1$ and $u \leqslant 0$
upper bidiagonal	$l \leqslant 0$ and $u \leqslant 1$
tridiagonal	$l \leqslant 1$ and $u \leqslant 1$
pentadiagonal	$l \leqslant 2$ and $u \leqslant 2$

matrix can be negative. Intuitively, a negative bandwidth means that the matrix is zero on the main diagonal; it allows to describe matrices that have a non-zero band anywhere, not just around the main diagonal. An example of a matrix with negative bandwidth, in this case $(-1, 2)$, is given below.

$$\begin{pmatrix} 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{pmatrix}$$

c.1 COMPUTATION OF THE BANDWIDTH

With the introduction of bandwidth, all properties that can be defined in terms of bandwidth can be determined by computing the bandwidth of an expression once and then checking the conditions in Tab. c.1. In the following, we describe how the bandwidth is computed for expressions built from the operations supported by Linnea. Let $A \in \mathbb{R}^{m \times n}$ be a matrix with $\mathcal{B}(A) = b = (l, u)$, and $A_i \in \mathbb{R}^{m_i \times n_i}$ be matrices with $\mathcal{B}(A_i) = b_i = (l_i, u_i)$.

ADDITION For sums of matrices, the upper (or lower) bandwidth is the maximum of the upper (or lower) bandwidth of all summands:

$$\mathcal{B}(A_1 + \ldots + A_i) = \big(\max(l_1, \ldots, l_i), \max(u_1, \ldots, u_i)\big).$$

It should be noted that this function can return an overapproximation of the bandwidth if entries cancel out.

MATRIX MULTIPLICATION For the product of two matrices, the upper (or lower) bandwidth of the product is the sum of the upper (or lower) bandwidths of the factors. In order to account for the fact that the bandwidth is limited by the size of the matrix, we define an auxiliary function that limits the value of a variable v:

$$\text{limit}(v, l, u) := \begin{cases} u & \text{if } u < v \\ v & \text{if } l \leqslant v \leqslant u \\ l & \text{if } v < l. \end{cases}$$

With this function, the bandwidth of a product of two matrices can be defined as:

$$\begin{aligned} \mathcal{B}(A_1 A_2) = \big(&\text{limit}(l_1 + l_2, -n_2, m_1 - 1), \\ &\text{limit}(u_1 + u_2, -m_1, n_2 - 1)\big). \end{aligned} \tag{c.3}$$

Proof: Let $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$, and $C \in \mathbb{R}^{m\times n}$. Without loss of generality, we assume that the absolute values of all bandwidths are much smaller than the matrix sizes. Under this assumption, the bandwidth of $C = AB$ is

$$\mathcal{B}(C) = (l_A + l_B, u_A + u_B).$$

We show that for all i and j, $i > j + l_A \Rightarrow a_{ij} = 0$ and $i > j + l_B \Rightarrow b_{ij} = 0$ implies $i > j + l_C \Rightarrow c_{ij} = 0$ with $l_C = l_A + l_B$. The element c_{ij} of C is computed as $c_{ij} = \sum_{\lambda=1}^{k} a_{i\lambda} b_{\lambda j}$. Thus, $c_{ij} = 0$ if for all λ, either $a_{i\lambda} = 0$ or $b_{\lambda j} = 0$. Following from $i > j + l_A \Rightarrow a_{ij} = 0$, $a_{i\lambda} = 0$ if $i > \lambda + l_A$, or equivalently $i - l_A > \lambda$. Following from $i > j + l_B \Rightarrow b_{ij} = 0$, $b_{\lambda j} = 0$ if $\lambda > j + l_B$. Thus, $c_{ij} = 0$ if $i - l_A > j + l_B$, which is equivalent to $i > j + l_A + l_B$. Analogously, one can show that $u_C = u_A + u_B$. ∎

Unfortunately, this formula can produce an overapproximation of the bandwidth if it is applied to products of more than two matrices. Consider as an example the product $A = A_1 A_2 A_3$ with

$$\begin{aligned} (m_1, n_1) &= (8, 4) & b_1 &= (0, 0) \\ (m_2, n_2) &= (4, 4) & b_2 &= (2, 0) \\ (m_3, n_3) &= (4, 4) & b_3 &= (2, 0) \\ (m, n) &= (8, 4). \end{aligned}$$

Since formula (c.3) is only defined for products of two matrices, it is necessary to introduce parenthesis and compute the bandwidth in two steps. Due to associativity, there are two alternatives:

1. $(A_1 A_2) A_3$.

   Let $A' = (A_1 A_2)$. Then $(m', n') = (8, 4)$ and $\mathcal{B}(A') = (2, 0)$. According to formula (c.3), the bandwidth of $A = A' A_3$ is $\mathcal{B}(A) = (4, 0)$.

2. $A_1(A_2A_3)$.

   Let $A' = (A_2A_3)$. Then $(m', n') = (4, 4)$ and $\mathcal{B}(A') = (3, 0)$ since the upper limit for $l_{A'}$ is $m' - 1 = 3$. Intuitively, the entire lower half of A' is already full with $l_{A'} = 3$. With this parenthesization, the bandwidth of $A = A_1A'$ is $\mathcal{B}(A) = (3, 0)$.

While in this case, the parenthesization $A_1(A_2A_3)$ produces the most accurate result, this is not a general rule. With the bandwidths

$$b_1 = (7, 3) \qquad b_2 = (3, 3) \qquad b_3 = (-3, 3),$$

the parenthesization $A_1(A_2A_3)$ leads to the bandwidth $\mathcal{B}(A) = (7, 3)$, while the more accurate result $\mathcal{B}(A) = (4, 3)$ is obtained with the parenthesization $(A_1A_2)A_3$. Formula (c.3) already returns inaccurate results for products of two matrices if one of them is zero: For a product of two matrices of size 4×4 with bandwidths $(2, 2)$ and $(3, -4)$, that is, the second matrix is zero, the bandwidth is computed as $(3, -2)$, which does not indicate that the resulting matrix is zero too. However, in this case, it can easily be detected that the resulting matrix must be zero based on the two input matrices. In products of more than two matrices, the situation is more complicated because the zero matrix can appear as the result of some intermediate product.

It is an open question whether or not a method can be developed to accurately compute the bandwidth of products of an arbitrary number of matrices. However, it is also not clear if the cases described above actually occur in practice.

SCALAR MULTIPLICATION The multiplication of a matrix with a scalar $\alpha \neq 0$ does not change the bandwidth:

$$\mathcal{B}(\alpha A) = (l, u).$$

TRANSPOSITION When a matrix is transposed, upper and lower bandwidth are switched:

$$\mathcal{B}\left(A^T\right) = (u, l).$$

INVERSION The upper (or lower) bandwidth of an inverted matrix is zero if the upper (or lower) bandwidth of the original matrix is zero. Non-zero elements above (or below) the main diagonal can lead to a fill-in of the entire upper (or lower) half.[1] Thus, for a matrix $A \in \mathbb{R}^{n \times n}$ where $\mathcal{B}(A^{-1}) = (l', u')$, the lower bandwidth l' is

$$l' = \begin{cases} n - 1 & \text{if } l > 0 \\ 0 & \text{if } l = 0. \end{cases}$$

1 In practice, the fill-in depends on the location of the non-zero elements in the original matrix. If they are close to the main diagonal, in the inverse the magnitude of the fill-in decreases with increasing distance from the main diagonal.

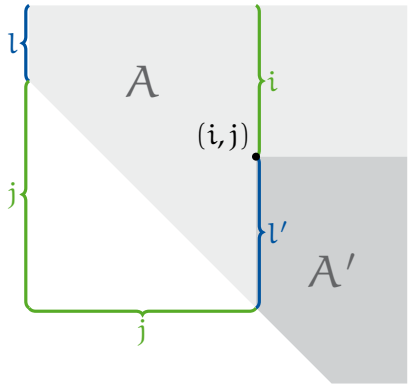


Figure c.1: An illustration of the computation of the lower bandwidth l' of the submatrix A' at position (i, j) from the lower bandwidth l of the matrix A. As shown above, it holds that $l + j = l' + i$, which can be rewritten to $l' = l + j - i$. The formula for the upper bandwidth can be derived analogously.

The upper bandwidth u' is computed analogously. A matrix with negative bandwidth does not have full rank and cannot be inverted.

C.2 BANDWIDTH OF SUBMATRICES

Even though Linnea is not able yet to partition matrices into submatrices, we would like to point out that the bandwidth of a submatrix can be computed from the bandwidth of the full matrix.

Let $A \in \mathbb{R}^{m \times n}$ be a matrix with bandwidth (l, u), and $A' \in \mathbb{R}^{m' \times n'}$ be a submatrix of A. The position of A' in A is described by the tuple (i, j) where i and j are respectively the row and column offset of the top-left corner of A' relative to the top-left corner of A. With indices starting at zero, this offset is equal to the indices of the element a_{ij} of A that is in the top-left corner of A'. For instance, given a matrix

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \end{pmatrix},$$

the submatrix of A at position $(1, 2)$ of size 2×2 is

$$A' = \begin{pmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{pmatrix}$$

The bandwidth (l', u') of A' with size $m' \times n'$ is computed as

$$\begin{aligned} \mathcal{B}(A') = \big(&\text{limit}(l + j - i, -n', m' - 1), \\ &\text{limit}(u + i - j, -m', n' - 1)\big). \end{aligned}$$

Fig. c.1 illustrates how this formula can be derived. Again, this formula also works with negative bandwidth. Indeed, allowing for the

bandwidth to be negative is especially useful when computing the bandwidth of submatrices because it makes it possible to accurately compute the bandwidth of arbitrary submatrices, including the case where the submatrix is zero. As an example, let A be the lower triangular matrix

$$\begin{pmatrix} a_{00} & 0 & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 \\ a_{20} & a_{21} & a_{22} & 0 \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}$$

which has the bandwidth $\mathcal{B}(A) = (3, 0)$. The submatrix A' at position $(0, 1)$ of size 3×2 is

$$\begin{pmatrix} 0 & 0 \\ a_{11} & 0 \\ a_{21} & a_{22} \end{pmatrix}.$$

The bandwidth of A' is $\mathcal{B}(A') = (2, -1)$. The submatrix A'' at position $(0, 2)$ of size 2×2 is zero. With the formula shown above, its bandwidth is computed as $(1, -2)$, which indeed implies that A'' is zero (see Tab. c.1).

BIBLIOGRAPHY

[1] Alfred V. Aho, Mahadevan Ganapathi, and Steven W.K. Tjiang. "Code Generation Using Tree Matching and Dynamic Programming." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 11.4 (1989), pp. 491–516.

[2] Alfred V. Aho and Stephen C. Johnson. "Optimal Code Generation for Expression Trees." In: *Journal of the ACM (JACM)* 23.3 (July 1, 1976), pp. 488–501. DOI: 10.1145/321958.321970.

[3] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. "Recursive Program Synthesis." In: *Lecture Notes in Computer Science*. Vol. 8044. Chapter 67 vols. Berlin, Heidelberg: Springer, Berlin, Heidelberg, July 13, 2013, pp. 934–950. ISBN: 978-3-642-39798-1. DOI: 10.1007/978-3-642-39799-8_67.

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

[5] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999. 301 pp. ISBN: 978-0-521-77920-3.

[6] Henrik Barthels and Paolo Bientinesi. *Code Generation in Linnea*. Extended abstract. 6th International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY). Phoenix, Arizona, June 2019.

[7] Henrik Barthels, Marcin Copik, and Paolo Bientinesi. "The Generalized Matrix Chain Algorithm." In: *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization*. Vienna, Austria, Feb. 2018, pp. 138–148. DOI: 10.1145/3168804.

[8] Henrik Barthels, Christos Psarras, and Paolo Bientinesi. "Automatic Generation of Efficient Linear Algebra Programs." In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. PASC '20. Geneva, Switzerland: Association for Computing Machinery, 2020. ISBN: 9781450379939. DOI: 10.1145/3394277.3401836.

[9] Henrik Barthels, Christos Psarras, and Paolo Bientinesi. "Linnea: Automatic Generation of Efficient Linear Algebra Programs." In: *ACM Trans. Math. Softw.* 47.3 (June 2021). ISSN: 0098-3500. DOI: 10.1145/3446632.

[10] Gerald Baumgartner et al. "Synthesis of High-Performance Parallel Programs for a Class of Ab Initio Quantum Chemistry Models." In: *Proceedings of the IEEE* 93.2 (Jan. 1, 2005), pp. 276–292. DOI: 10.1109/JPROC.2004.840311.

[11] Dan Benanav, Deepak Kapur, and Paliath Narendran. "Complexity of Matching Problems." In: *Journal of Symbolic Computation* 3.1 (Feb. 4, 1987), pp. 203–216. ISSN: 0747-7171. DOI: 10.1016/S0747-7171(87)80027-5.

[12] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. "Julia: A Fresh Approach to Numerical Computing." In: *SIAM Review* 59.1 (Jan. 2017), pp. 65–98. ISSN: 0036-1445, 1095-7200. DOI: 10.1137/141000671.

[13] Sanjukta Bhowmick, Victor Eijkhout, Yoav Freund, Erika Fuentes, and David Keyes. "Application of Machine Learning to the Selection of Sparse Linear Solvers." In: *Int. J. High Perf. Comput. Appl* (2006).

[14] Paolo Bientinesi. "Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms." PhD Thesis. 2006.

[15] Paolo Bientinesi, Brian Gunter, and Robert A. van de Geijn. "Families of Algorithms Related to the Inversion of a Symmetric Positive Definite Matrix." In: *ACM Transactions on Mathematical Software (TOMS)* 35.1 (2008), pp. 1–22.

[16] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. "Representing Linear Algebra Algorithms in Code: The FLAME Application Program Interfaces." In: *ACM Transactions on Mathematical Software* 31.1 (Mar. 1, 2005), pp. 27–59. DOI: 10.1145/1055531.1055533.

[17] Paolo Bientinesi and Robert A. van de Geijn. "Goal-Oriented and Modular Stability Analysis." In: *SIAM J. Matrix Analysis Applications* 32.1 (Jan. 1, 2011), pp. 286–308. DOI: 10.1137/080741057.

[18] Phillip G. Bradford, Gregory J. E. Rawlins, and Gregory E. Shannon. "Efficient Matrix Chain Ordering in Polylog Time." In: *SIAM Journal on Computing* 27.2 (Apr. 1, 1998), pp. 466–490. ISSN: 0097-5397. DOI: 10.1137/S0097539794270698.

[19] Coen Bron and Joep Kerbosch. "Algorithm 457: Finding All Cliques of an Undirected Graph." In: *Communications of the ACM* 16.9 (Sept. 1, 1973), pp. 575–577. ISSN: 0001-0782. DOI: 10.1145/362342.362367.

[20] Lorenzo Chelini, Andi Drebes, Alex Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. "Progressive Raising in Multi-Level IR." In: (2021).

[21] Francis Y. Chin. "An O(n) Algorithm for Determining a Near-Optimal Computation Order of Matrix Chain Products." In: *Communications of the ACM* 21.7 (July 1, 1978), pp. 544–549. ISSN: 0001-0782. DOI: 10.1145/359545.359556.

[22] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. "Automatic Generation of Efficient Sparse Tensor Format Conversion Routines." In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '20: 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation. London UK: ACM, June 11, 2020, pp. 823–838. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3385963.

[23] Julianne Chung, Matthias Chung, J. Tanner Slagel, and Luis Tenorio. "Stochastic Newton and Quasi-Newton Methods for Large Linear Least-Squares Problems." In: *CoRR* math.NA (Jan. 1, 2017). URL: http://arxiv.org/abs/1702.07367v1.

[24] George E. Collins. "Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decompostion." In: *Automata Theory and Formal Languages*. Springer, 1975, pp. 134–183.

[25] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd. The MIT Press, 2009.

[26] Artur Czumaj. "Very Fast Approximation of the Matrix Chain Product Problem." In: *Journal of Algorithms* 21.1 (July 1, 1996), pp. 71–79. ISSN: 0196-6774. DOI: 10.1006/jagm.1996.0037.

[27] Timothy A. Davis. "Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra." In: *ACM Transactions on Mathematical Software* 45.4 (Dec. 25, 2019), pp. 1–25. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/3322125.

[28] Nachum Dershowitz. "Termination of Rewriting." In: *Journal of Symbolic Computation* 3.1-2 (1987), pp. 69–115.

[29] Edoardo Di Napoli, Elmar Peise, Markus Hrywniak, and Paolo Bientinesi. "High-Performance Generation of the Hamiltonian and Overlap Matrices in Flapw Methods." In: *Computer Physics Communications*. High Performance Computing for Advanced Modeling and Simulation of Materials 211 (Feb. 1, 2017), pp. 61–72. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2016.10.003.

[30] Alan J. J. Dick. "An Introduction to Knuth-Bendix Completion." In: *The Computer Journal* 34.1 (Jan. 1, 1991), pp. 2–15. DOI: 10.1093/comjnl/34.1.2.

[31] Yin Ding and Ivan W. Selesnick. "Sparsity-Based Correction of Exponential Artifacts." In: *Signal Processing* 120 (Mar. 2016), pp. 236–248. ISSN: 01651684. DOI: 10.1016/j.sigpro.2015.09.017. arXiv: 1509.07234.

[32] Elizabeth D. Dolan and Jorge J. Moré. "Benchmarking Optimization Software with Performance Profiles." In: *Mathematical programming* 91.2 (2002), pp. 201–213.

[33] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. "A Set of Level 3 Basic Linear Algebra Subprograms." In: *ACM Trans. Math. Softw.* 16.1 (Jan. 1, 1990), pp. 1–17. DOI: 10.1145/77626.79170.

[34] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. "An Extended Set of FORTRAN Basic Linear Algebra Subprograms." In: *ACM Transactions on Mathematical Software* 14.1 (Mar. 1, 1988), pp. 1–17. DOI: 10.1145/42288.42291.

[35] Mohamed Elsayed Mohamed Ahmed El Seidy. "Efficient Online Processing for Advanced Analytics." EPFL, 2017.

[36] Diego Fabregat-Traver and Paolo Bientinesi. "Automatic Generation of Loop-Invariants for Matrix Operations." In: *ICCSA Workshops* (Jan. 1, 2011), pp. 82–92. ISSN: 978-1-4577-0142-9. DOI: 10.1109/ICCSA.2011.41.

[37] Diego Fabregat-Traver and Paolo Bientinesi. "Knowledge-Based Automatic Generation of Partitioned Matrix Expressions." In: *CASC* 6885.4 (Jan. 1, 2011), pp. 144–157. ISSN: 978-3-642-23567-2. DOI: 10.1007/978-3-642-23568-9_12.

[38] Diego Fabregat-Traver and Paolo Bientinesi. "A Domain-Specific Compiler for Linear Algebra Operations." In: *High Performance Computing for Computational Science - VECPAR 2012*. Ed. by Michel Daydé, Osni Marques, and Kengo Nakajima. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 346–361. ISBN: 978-3-642-38718-0. DOI: 10.1007/978-3-642-38718-0_33.

[39] Diego Fabregat-Traver and Paolo Bientinesi. "Computing Petaflops over Terabytes of Data." In: *ACM Transactions on Mathematical Software* 40.4 (June 1, 2014), pp. 1–22. DOI: 10.1145/2560421.

[40] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. "Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples." In: *arXiv.org* cs.PL.6 (Nov. 22, 2016), pp. 422–436. DOI: 10.1145/3140587.3062351.

[41] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. "Component-Based Synthesis for Complex APIs." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages - POPL 2017*. The 44th ACM SIGPLAN Symposium. Paris, France: ACM Press, 2017, pp. 599–612. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009851.

[42] Hal Finkel, Alex McCaskey, Tobi Popoola, Dmitry Lyakh, and Johannes Doerfert. "Really Embedding Domain-Specific Languages into C++." In: 2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar). GA, USA: IEEE, Nov. 2020, pp. 65–73. ISBN: 978-0-7381-1042-4. DOI: 10.1109/LLVMHPCHiPar51896.2020.00012.

[43] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M. Veras, Daniele G. Spampinato, Jeremy R. Johnson, Markus Püschel, James C. Hoe, and Jose M. F. Moura. "SPIRAL: Extreme Performance Portability." In: *Proceedings of the IEEE* 106.11 (Nov. 1, 2018), pp. 1935–1968. DOI: 10.1109/JPROC.2018.2873289.

[44] Matteo Frigo and Steven G. Johnson. "The Design and Implementation of FFTW3." In: *Proceedings of the IEEE* 93.2 (Jan. 1, 2005), pp. 216–231. DOI: 10.1109/JPROC.2004.840301.

[45] Gianluca Frison, Dimitris Kouzoupis, Andrea Zanelli, and Moritz Diehl. "BLASFEO - Basic Linear Algebra Subroutines for Embedded Optimization." In: *CoRR* cs.MS (Jan. 1, 2017). URL: http://arxiv.org/abs/1704.02457v3.

[46] Michael R. Garey and David S. Johnson. *Computers and Intractability*. Vol. 29. W. H. Freeman, 1979.

[47] Jürgen Giesl, Albert Rubio, Christian Sternagel, Johannes Waldmann, and Akihisa Yamada. "The Termination and Complexity Competition." In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2019, pp. 156–166.

[48] Philip Ginsbach, Bruce Collie, and Michael F. P. O'Boyle. "Automatically Harnessing Sparse Acceleration." In: *Proceedings of the 29th International Conference on Compiler Construction*. CC '20: 29th International Conference on Compiler Construction. San Diego CA USA: ACM, Feb. 22, 2020, pp. 179–190. ISBN: 978-1-4503-7120-9. DOI: 10.1145/3377555.3377893.

[49] Sadashiva S. Godbole. "On Efficient Computation of Matrix Chain Products." In: *IEEE Transactions on Computers* 100.9 (1973), pp. 864–866.

[50] Gene H. Golub, Per Christian Hansen, and Dianne P. O'Leary. "Tikhonov Regularization and Total Least Squares." In: *SIAM Journal on Matrix Analysis and Applications* 21.1 (July 31, 2006), pp. 185–194. DOI: 10.1137/S0895479897326432.

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

[52] Claude Gomez and Tony Scott. "Maple Programs for Generating Efficient Fortran Code for Serial and Vectorised Machines." In: *Computer Physics Communications* 115.2-3 (1998), pp. 548–562.

[53] Kazushige Goto and Robert A. van de Geijn. "Anatomy of High-Performance Matrix Multiplication." In: *ACM Transactions on Mathematical Software* 34.3 (May 2008), pp. 1–25. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/1356052.1356053.

[54] Peter Gottschling, David S. Wise, and Adwait Joshi. "Generic Support of Algorithmic and Structural Recursion for Scientific Computing." In: *International Journal of Parallel, Emergent and Distributed Systems* 24.6 (Dec. 2009), pp. 479–503. ISSN: 1744-5760, 1744-5779. DOI: 10.1080/17445760902758560.

[55] Robert M. Gower and Peter Richtárik. "Randomized Quasi-Newton Updates Are Linearly Convergent Matrix Inversion Algorithms." In: *SIAM J. Matrix Analysis Applications* 38.4 (Jan. 1, 2017), pp. 1380–1409. DOI: 10.1137/16M1062053.

[56] Gaël Guennebaud, Benoît Jacob, et al. *Eigen*. URL: http://eigen.tuxfamily.org.

[57] Sumit Gulwani. "Automating String Processing in Spreadsheets Using Input-Output Examples." In: *ACM SIGPLAN Notices* 46.1 (Jan. 26, 2011), p. 317. DOI: 10.1145/1925844.1926423.

[58] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. "Synthesis of Loop-Free Programs." In: *ACM SIGPLAN Notices* 46.6 (2011), pp. 62–73.

[59] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. "FLAME: Formal Linear Algebra Methods Environment." In: *ACM Transactions on Mathematical Software* 27.4 (Dec. 1, 2001), pp. 422–455. DOI: 10.1145/504210.504213.

[60] Samuel Z. Guyer and Calvin Lin. "Broadway: A Compiler for Exploiting the Domain-Specific Semantics of Software Libraries." In: *Proceedings of the IEEE* 93.2 (2005), pp. 342–357.

[61] William W. Hager. "Updating the Inverse of a Matrix." In: *SIAM Review* 31.2 (June 1, 1989), pp. 221–239. ISSN: 0036-1445. DOI: 10.1137/1031049.

[62] Charles R. Harris et al. "Array Programming with NumPy." In: *Nature* 585.7825 (Sept. 17, 2020), pp. 357–362. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-020-2649-2.

[63] William R. Harris and Sumit Gulwani. "Spreadsheet Table Transformations from Examples." In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '11*. The 32nd ACM SIGPLAN Conference. San Jose, California, USA: ACM Press, 2011, p. 317. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993536.

[64] Alexander Herold and Jörg H. Siekmann. "Unification in Abelian Semigroups." In: *Journal of Automated Reasoning* 3.3 (1987), pp. 247–283.

[65] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1996. 688 pp. ISBN: 978-0-89871-355-8.

[66] Nicholas J. Higham. *Functions of Matrices: Theory and Computation*. Philadelphia, PA, USA: SIAM, 2008. ISBN: 978-0-89871-646-7.

[67] John E. Hopcroft and Richard M. Karp. "An $n^5/2$ Algorithm for Maximum Matchings in Bipartite Graphs." In: *SIAM Journal on computing* 2.4 (1973), pp. 225–231.

[68] T. C. Hu and M. T. Shing. "Computation of Matrix Chain Products. Part I." In: *SIAM Journal on Computing* 11.2 (May 1, 1982), pp. 362–373. ISSN: 0097-5397. DOI: 10.1137/0211028.

[69] T. C. Hu and M. T. Shing. "Computation of Matrix Chain Products. Part II." In: *SIAM Journal on Computing* 13.2 (May 1, 1984), pp. 228–251. ISSN: 0097-5397. DOI: 10.1137/0213017.

[70] Aurélie Hurault, Kyungim Baek, and Henri Casanova. "Selecting Linear Algebra Kernel Composition Using Response Time Prediction." In: *Softw., Pract. Exper.* 45.12 (Jan. 1, 2015), pp. 1659–1676. DOI: 10.1002/spe.2307.

[71] Roman Iakymchuk and Paolo Bientinesi. "Modeling Performance Through Memory-Stalls." In: *ACM SIGMETRICS Performance Evaluation Review* 40.2 (Oct. 8, 2012), pp. 86–91. ISSN: 0163-5999. DOI: 10.1145/2381056.2381076.

[72] Klaus Iglberger, Georg Hager, Jan Treibig, and Ulrich Rüde. "Expression Templates Revisited: A Performance Analysis of Current Methodologies." In: *SIAM J. Scientific Computing* 34.2 (Jan. 1, 2012), pp. C42–C69. DOI: 10.1137/110830125.

[73] Intel. *Intel Math Kernel Library*. URL: http://software.intel.com/en-us/intel-mkl.

[74] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. "Oracle-Guided Component-Based Program Synthesis." In: *ICSE* 1 (Jan. 1, 2010), p. 215. ISSN: 9781605587196. DOI: 10.1145/1806799.1806833.

[75] Víctor M. Jiménez and Andrés Marzal. "Computing the K Shortest Paths - A New Algorithm and an Experimental Comparison." In: *Algorithm Engineering* 1668 (Chapter 4 Jan. 1, 1999), pp. 15–29. ISSN: 978-3-540-66427-7. DOI: 10.1007/3-540-48318-7_4.

[76] Rajeev Joshi, Greg Nelson, and Keith H Randall. "Denali - A Goal-Directed Superoptimizer." In: *PLDI* (Jan. 1, 2002), p. 304. ISSN: 1581134630. DOI: 10.1145/512529.512566.

[77] Peter Kabal. "Minimum Mean-Square Error Filtering: Autocorrelation/Covariance, General Delays, and Multirate Systems." In: (Jan. 1, 2011). URL: http://www-mmsp.ece.mcgill.ca/Documents/Reports/2011/KabalR2011d.pdf.

[78] Rudolf Emil Kalman. "A New Approach to Linear Filtering and Prediction Problems." In: *Journal of basic Engineering* 82.1 (Mar. 1, 1960), pp. 35–45. DOI: 10.1115/1.3662552.

[79] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, and Henning Meyerhenke. "Mathematical Foundations of the Graphblas." In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–9.

[80] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. "The Tensor Algebra Compiler." In: *Proceedings of the ACM on Programming Languages* 1 (OOPSLA Oct. 12, 2017), pp. 77–29. DOI: 10.1145/3133901.

[81] Donald E. Knuth and Peter B. Bendix. "Simple Word Problems in Universal Algebras." In: *Automation of Reasoning*. Springer, 1983, pp. 342–376.

[82] Manuel Krebber. "Non-Linear Associative-Commutative Many-to-One Pattern Matching with Sequence Variables." In: *CoRR* cs.SC (Jan. 1, 2017). URL: http://arxiv.org/abs/1705.00907v1.

[83] Manuel Krebber and Henrik Barthels. "MatchPy: Pattern Matching in Python." In: *Journal of Open Source Software* 3.26 (June 2018), p. 2. DOI: 10.21105/joss.00670.

[84] Manuel Krebber, Henrik Barthels, and Paolo Bientinesi. "Efficient Pattern Matching in Python." In: *Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing*. In conjunction with SC17: The International Conference for High Performance Computing, Networking, Storage and

Analysis. Denver, Colorado, Nov. 2017. DOI: 10.1145/3149869.3149871.

[85] Manuel Krebber, Henrik Barthels, and Paolo Bientinesi. "MatchPy: A Pattern Matching Library." In: *Proceedings of the 15th Python in Science Conference*. Austin, Texas, July 2017. arXiv: 1710.06915.

[86] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation." In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Seoul, Korea (South): IEEE, Feb. 27, 2021, pp. 2–14. ISBN: 978-1-72818-613-9. DOI: 10.1109/CGO51591.2021.9370308.

[87] Charles Lawrence Lawson, Richard J. Hanson, David R. Kincaid, and Frederick Thomas Krogh. "Basic Linear Algebra Subprograms for Fortran Usage." In: *ACM Trans. Math. Softw.* 5.3 (Jan. 1, 1979), pp. 308–323. DOI: 10.1145/355841.355847.

[88] Heejo Lee, Jong Kim, Sung Je Hong, and Sunggu Lee. "Processor Allocation and Task Scheduling of Matrix Chain Products on Parallel Systems." In: *IEEE Transactions on Parallel and Distributed Systems* 14.4 (Apr. 2003), pp. 394–407. ISSN: 1558-2183. DOI: 10.1109/TPDS.2003.1195411.

[89] Percy Liang, Michael I. Jordan, and Dan Klein. "Learning Programs: A Hierarchical Bayesian Approach." In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 2010, pp. 639–646.

[90] Dhruv C. Makwana and Neel Krishnaswami. "Numlin: Linear Types for Linear Algebra." In: (2019).

[91] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. "Jungloid Mining - Helping to Navigate the API Jungle." In: *PLDI* (Jan. 1, 2005), p. 48. ISSN: 1595930566. DOI: 10.1145/1065010.1065018.

[92] Bryan Marker, Jack Poulson, Don Batory, and Robert van de Geijn. "Designing Linear Algebra Algorithms by Transformation: Mechanizing the Expert Developer." In: *International Conference on High Performance Computing for Computational Science*. Springer, 2012, pp. 362–378.

[93] Bryan Marker, Martin Schatz, Devin Matthews, Isil Dillig, Robert van de Geijn, and Don Batory. *DxTer: An Extensible Tool for Optimal Dataflow Program Generation*. Technical Report TR-15-03, The University of Texas at Austin, 2015.

[94] Henry Massalin. "Superoptimizer: A Look at the Smallest Program." In: *ACM SIGARCH Computer Architecture News* 15.5 (Nov. 1987), pp. 122–126. ISSN: 0163-5964. DOI: 10.1145/36177.36194.

[95] MathWorks. *Matlab Documentation*. URL: https://www.mathworks.com/help/matlab/.

[96] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, Jan. 1, 1997.

[97] Noël M. Nachtigal, Satish C. Reddy, and Lloyd N. Trefethen. "How Fast Are Nonsymmetric Matrix Iterations?" In: *SIAM Journal on Matrix Analysis and Applications* 13.3 (July 1, 1992), pp. 778–795. ISSN: 0895-4798. DOI: 10.1137/0613049.

[98] Uwe Naumann. "Optimal Jacobian Accumulation Is NP-Complete." In: *Mathematical Programming* 112.2 (Nov. 30, 2007), pp. 427–441. ISSN: 0025-5610, 1436-4646. DOI: 10.1007/s10107-006-0042-z.

[99] Thomas Nelson, Geoffrey Belter, Jeremy G. Siek, Elizabeth Jessup, and Boyana Norris. "Reliable Generation of High-Performance Matrix Algebra." In: *ACM Transactions on Mathematical Software* 41.3 (June 2015), pp. 1–27. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/2629698.

[100] Elias D. Nino-Ruiz, Adrian Sandu, and Xinwei Deng. "A Parallel Implementation of the Ensemble Kalman Filter Based on Modified Cholesky Decomposition." In: *Journal of Computational Science* 36 (Sept. 2019), p. 100654. ISSN: 18777503. DOI: 10.1016/j.jocs.2017.04.005.

[101] Kazufumi Nishida, Yasuaki Ito, and Koji Nakano. "Accelerating the Dynamic Programming for the Matrix Chain Product on the GPU." In: *2011 Second International Conference on Networking and Computing*. 2011 Second International Conference on Networking and Computing. Nov. 2011, pp. 320–326. DOI: 10.1109/ICNC.2011.62.

[102] Elmar Peise and Paolo Bientinesi. "Performance Modeling for Dense Linear Algebra." In: *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. SCC '12. USA: IEEE Computer Society, Nov. 10, 2012, pp. 406–416. ISBN: 978-0-7695-4956-9. DOI: 10.1109/SC.Companion.2012.60.

[103] Elmar Peise and Paolo Bientinesi. "A Study on the Influence of Caching: Sequences of Dense Linear Algebra Kernels." In: High Performance Computing for Computational Science – VECPAR 2014. Vol. 8969. Springer International Publishing, June 30, 2014, pp. 245–258. ISBN: 978-3-319-17352-8. DOI: 10.1007/978-3-319-17353-5_21.

[104] Elmar Peise and Paolo Bientinesi. "Algorithm 979: Recursive Algorithms for Dense Linear Algebra—The ReLAPACK Collection." In: *ACM Transactions on Mathematical Software* 44.2 (Oct. 10, 2017), pp. 1–19. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/3061664.

[105] Eduardo Pelegrí-Llopart and Susan L. Graham. "Optimal Code Generation for Expression Trees: An Application BURS Theory." In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: Association for Computing Machinery, Jan. 13, 1988, pp. 294–308. ISBN: 978-0-89791-252-5. DOI: 10.1145/73560.73586.

[106] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodík, and Dinakar Dhurjati. "Scaling up Superoptimization." In: *ASPLOS* (Jan. 1, 2016), pp. 297–310. ISSN: 9781450340915. DOI: 10.1145/2872362.2872387.

[107] Federico Pizzuti, Michel Steuwer, and Christophe Dubach. "Position-Dependent Arrays and Their Application for High Performance Code Generation." In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing - FHPNC 2019*. The 8th ACM SIGPLAN International Workshop. Berlin, Germany: ACM Press, 2019, pp. 14–26. ISBN: 978-1-4503-6814-8. DOI: 10.1145/3331553.3342614.

[108] Christos Psarras, Henrik Barthels, and Paolo Bientinesi. "The Linear Algebra Mapping Problem." In: *CoRR* abs/1911.09421 (2019). arXiv: 1911.09421 [cs.MS].

[109] Prakash Ramanan. "An Efficient Parallel Algorithm for the Matrix-Chain-Product Problem." In: *SIAM Journal on Computing* 25.4 (1996), pp. 874–893. DOI: 10.1137/0225039.

[110] Wolfram Research. *Mathematica Documentation*. URL: https://reference.wolfram.com/language/.

[111] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Third edition, Global edition. Prentice Hall Series in Artificial Intelligence. London: Pearson, 2016. 1132 pp. ISBN: 978-1-292-15396-4.

[112] Conrad Sanderson and Ryan Curtin. "Armadillo: A Template-Based C++ Library for Linear Algebra." In: *Journal of Open Source Software* 1.2 (2016), p. 26.

[113] Eric Schkufza, Rahul Sharma 0001, and Alex Aiken. "Stochastic Superoptimization." In: *ASPLOS* (Jan. 1, 2013), p. 305. ISSN: 9781450318709. DOI: 10.1145/2451116.2451150.

[114] Ravi Sethi and Jeffrey D. Ullman. "The Generation of Optimal Code for Arithmetic Expressions." In: *Journal of the ACM (JACM)* 17.4 (1970), pp. 715–728.

[115] Amir Shaikhha, Mohammed Elseidy, Stephan Mihaila, Daniel Espino, and Christoph Koch. "Synthesis of Incremental Linear Algebra Programs." In: *ACM Transactions on Database Systems* 45.3 (Aug. 26, 2020), 12:1–12:44. ISSN: 0362-5915. DOI: 10.1145/3385398.

[116] Amir Shaikhha and Lionel Parreaux. "Finally, a Polymorphic Linear Algebra Language." Version 1.0. In: (2019). In collab. with Michael Wagner, 29 pages. DOI: 10.4230/LIPICS.ECOOP.2019.25.

[117] Jeremy G. Siek, Ian Karlin, and Elizabeth R. Jessup. "Build to Order Linear Algebra Kernels." In: *Distributed Processing Symposium (IPDPS)*. Distributed Processing Symposium (IPDPS). IEEE, Jan. 1, 2008, pp. 1–8. ISBN: 978-1-4244-1693-6. DOI: 10.1109/IPDPS.2008.4536183.

[118] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. "Sketching Stencils." In: *ACM SIGPLAN Notices* 42.6 (June 10, 2007), pp. 167–178. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/1273442.1250754.

[119] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. "Programming by Sketching for Bit-Streaming Programs." In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2005, pp. 281–294.

[120] Daniele G. Spampinato, Diego Fabregat-Traver, Paolo Bientinesi, and Markus Püschel. "Program Generation for Small-Scale Linear Algebra Applications." In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. Vienna, Austria: Association for Computing Machinery, Feb. 24, 2018, pp. 327–339. ISBN: 978-1-4503-5617-6. DOI: 10.1145/3168812.

[121] Daniele G. Spampinato and Markus Püschel. "A Basic Linear Algebra Compiler for Structured Matrices." In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. CGO '16. Barcelona, Spain: Association for Computing Machinery, Feb. 29, 2016, pp. 117–127. ISBN: 978-1-4503-3778-6. DOI: 10.1145/2854038.2854060.

[122] Paul Springer and Paolo Bientinesi. "Design of a High-Performance GEMM-like Tensor–Tensor Multiplication." In: *ACM Transactions on Mathematical Software* 44.3 (Apr. 26, 2018), pp. 1–29. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/3157733.

[123] Damian Straszak and Nisheeth K. Vishnoi. "On a Natural Dynamics for Linear Programming." In: *CoRR* cs.DS (Nov. 22, 2015). arXiv: 1511.07020. URL: http://arxiv.org/abs/1511.07020.

[124] Steve A. Strate and Roger L. Wainwright. "Parallelization of the Dynamic Programming Algorithm for the Matrix Chain Product on a Hypercube." In: *Proceedings of the 1990 Symposium on Applied Computing*. IEEE, 1990, pp. 78–84.

[125] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. "Equality Saturation - A New Approach to Optimization." In: *POPL '09*. Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA: ACM Press, Jan. 1, 2009, pp. 264–276. ISBN: 978-1-60558-379-2. DOI: 10.1145/1480881.1480915.

[126] Tom Tirer and Raja Giryes. "Image Restoration by Iterative Denoising and Backward Projections." In: *arXiv.org* cs.CV (Oct. 18, 2017), pp. 138–142. ISSN: 978-1-5386-1565-2. DOI: 10.1109/sampta.2017.8024474.

[127] Sid Ahmed Ali Touati and Denis Barthou. "On the Decidability of Phase Ordering Problem in Optimizing Compilation." In: *Conf. Computing Frontiers* (Jan. 1, 2006), p. 147. ISSN: 1595933026. DOI: 10.1145/1128022.1128042.

[128] Field G. Van Zee, Ernie Chan, Robert A. van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. "The Libflame Library for Dense Matrix Computations." In: *Computing in Science & Engineering* 11.6 (Nov. 2009), pp. 56–63. ISSN: 1521-9615. DOI: 10.1109/MCSE.2009.207.

[129] Field G. Van Zee and Robert A. van de Geijn. "BLIS - A Framework for Rapidly Instantiating BLAS Functionality." In: *ACM Trans. Math. Softw.* 41.3 (Jan. 1, 2015), pp. 1–33. DOI: 10.1145/2764454.

[130] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. "SPORES: Sum-Product Optimization Via Relational Equality Saturation for Large Scale Linear Algebra." In: *Proceedings of the VLDB Endowment* 13.12 (Aug. 2020), pp. 1919–1932. ISSN: 2150-8097. DOI: 10.14778/3407790.3407799.

[131] Zhang Xianyi, Martin Kroeker, Werner Saar, Wang Qian, Zaheer Chothia, Chen Shaohu, and Luo Wen. *OpenBLAS*. URL: http://www.openblas.net.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Final Version as of August 9, 2021 (classicthesis v4.6).