Knowledge-Based Automatic Generation of Linear Algebra Algorithms and Code

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

vorgelegt von

Dipl.-Ing. Diego Fabregat Traver

aus Castellón de la Plana, Spanien.

Berichter: Prof. Paolo Bientinesi, Ph.D.
Prof. Dr. Uwe Naumann
Prof. Dr. Christian Bischof

Tag der mündlichen Prüfung: 06.12.2013

Abstract

This dissertation focuses on the design and the implementation of domain-specific compilers for linear algebra matrix equations. The development of efficient libraries for such equations, which lie at the heart of most software for scientific computing, is a complex process that requires expertise in a variety of areas, including the application domain, algorithms, numerical analysis and high-performance computing. Moreover, the process involves the collaboration of several people for a considerable amount of time. With our compilers, we aim to relieve the developers from both designing algorithms and writing code, and to generate routines that match or even surpass the performance of those written by human experts.

We present two compilers, CLAK and CLlCK, that take as input the description of a target equation together with domain-specific knowledge, and generate efficient customized algorithms and routines. CLAK targets high-level matrix equations, possibly encompassing the solution of multiple instances of interdependent problems. This compiler generates algorithms consisting in a sequence of library-supported building blocks. It builds on top of a methodology that combines a model of human expert reasoning with the power of computers; the search for algorithms makes use of the available domain knowledge to prune the search space and to tailor the algorithms to the application. Along the process, CLAK prioritizes the reduction of the computational cost, the elimination of redundant computations, and the selection of the most suitable building blocks. For one target equation, many algorithms, with different properties, are generated.

CLlCK, instead, addresses the generation of algorithms for specialized building blocks. To this end, this compiler adopts the FLAME methodology for the derivation of formally correct loop-based algorithms. CLlCK takes a three-stage approach: First, the PME(s) —a recursive definition of the target operation in a divide and conquer fashion— is found; then, the PME is analyzed to identify a family of loop invariants; finally, each loop invariant is transformed into a corresponding loop-based algorithm. CLlCK fully automates the application of this methodology; in this dissertation, we dissect the mechanisms necessary to make it possible.

As we show, for our compilers, the exploitation of both linear algebra and application-specific knowledge is crucial to generate efficient customized solvers. In order to facilitate the management of knowledge and to increase productivity, we raise the abstraction level and provide the users with an expressive domain-specific language that allows them to reason at the matrix equation level. The *users* are

thus able to state *what* needs to be solved, providing as much domain knowledge as possible, and delegating the *compilers* to find *how* to efficiently solve it.

We illustrate the potential of our compilers by applying them to real world problems. For instance, for a challenging problem arising in computational biology, the exploitation of the available knowledge leads to algorithms that lower the complexity of existing methods by orders of magnitude. Our algorithms, at the heart of the publicly available library OmicABEL, have become the state-of-the-art. We also carry out a thorough study of the application of our compilers to derivative operations, quantifying how much tools used in algorithmic differentiation can benefit from incorporating the techniques discussed in this dissertation. The experiments demonstrate the compilers' potential to produce efficient derivative versions of part of LAPACK and of the entire BLAS, an effort that requires thousands of routines and for which a manual approach is unfeasible.

This dissertation provides evidence that a linear algebra compiler, which increases experts' productivity and makes efficiency accessible to non-experts, is within reach.

Acknowledgements

First and foremost, I thank Paolo Bientinesi for being such an inspiring advisor. He has been a model of scientific passion and honesty, hard work, and perseverance. I am extremely grateful for his support and for the respect he has always shown towards my ideas; from the very first day, he gave me the freedom to make my own decisions, while steering me towards meaningful goals. We have shared an excellent relationship, and I cannot think of a better advisor for me.

I am also grateful to Prof. Christian Bischof for his firm belief in automation and his direct and indirect supervision. I wish to thank Prof. Bischof and Prof. Uwe Naumann as well for agreeing to review this dissertation, and Prof. Martin Bücker and Matthias Petschow for their valuable comments and suggestions on early drafts of this manuscript.

Financial support from the Deutsche Forschungsgemeinschaft (German Research Association) through grant GSC 111 and the DAAD (Deutscher Akademischer Austausch Dienst) through project 50225798 PARSEMUL is gratefully acknowledged; I extend the acknowledgement to the RWTH Computing Center (RZ) for the access to their computational resources and their support.

I wish to express my deep gratitude to everyone at AICES who makes an effort to provide us (the students) with the best environment and resources so that we can concentrate on our research. The many organized team-building activities, talks, seminars, and courses are also a great complement to our daily work. A special thanks goes to the service team for their support, relieving AICES students from many administrative tasks; particularly, thanks to Nicole Faber, Nadine Bachem, Annette de Haes, and Joelle Janssen.

I am indebted to Prof. Enrique Quintana Ortí, under whose supervision I completed my undergraduate studies. He awakened my interest in high-performance matrix computations, and guided my first steps into science. I owe him having introduced me to Paolo, and encouraging me to pursue a Ph.D.

Science is much more fun when exchanging ideas and joining efforts. In this respect, I am thankful to Dr. Yurii Aulchenko and Prof. Bücker for sharing with me a number of exciting problems that opened up multiple research directions. Their collaboration greatly contributes to this dissertation. I also thank the FLAME research group at Austin, Texas, and especially Prof. Robert van de Geijn, for developing the foundations of an important part of my research, and for their insight and numerous suggestions.

Of course, I thank all AICES students for the relaxing chat over all sorts of international sweets and cakes, and in particular, Aravind Balan for his predisposition to give me a hand whenever needed.

For the countless hours of scientific discussion and fun both at work and outside, I am grateful to the HPAC group: Paolo Bientinesi, Edoardo di Napoli, Matthias Petschow, Roman Iakymchuk, Elmar Peise, Lucas Beyer, Daniel Tameling, Paul Springer and Viola Wierschem. I am especially grateful to Matthias, Roman, Edo, and Paolo, who accompanied me during this entire trip. You have been like a family to me.

I cannot list everyone who joined our countless game nights. We played thousands of games, ate tones of pizzas, and most importantly, we had lots of fun. I also thank Edo and Jess for organizing many dinner and (bad) movie nights. Having Edo commenting on bad sci-fi movies is priceless.

I want to spend a few words to thank my Spanish friends in Aachen. Sports and food have been the best excuse to hang out and talk about the good things we have at home and how much we miss them. I do not forget my friends in Spain. There is no need to name them, they know who they are. Distance and time does not matter for them; every time we meet, it feels like I never left. I know you will always be there for me.

I owe the biggest and deepest thanks to my family, who always took care of me, supported me, and encouraged me to pursue my dreams. You made everything at your reach to turn my life into a wonderful adventure.

Finally, I find no words to express my infinite gratitude to Ana. You stood by me through thick and thin during all these years, and, despite the distance, you have been my biggest emotional support. *Todo al negro.*

Contents

1 Introduction **1**
- 1.1 Our compilers: A short overview . . . 3
  - 1.1.1 CLAK . . . 3
  - 1.1.2 CLICK . . . 5
- 1.2 Contributions . . . 6
- 1.3 Outline of the thesis . . . 8

2 CLAK **11**
- 2.1 Defining input equations . . . 12
- 2.2 Heuristics for the generation of algorithms . . . 16
  - 2.2.1 Dealing with the inverse operator . . . 16
  - 2.2.2 Mapping onto kernels . . . 19
- 2.3 Compiler's engine . . . 22
  - 2.3.1 Matrix algebra . . . 22
  - 2.3.2 Interface to building blocks . . . 24
  - 2.3.3 Inference of properties . . . 25
- 2.4 A detailed example: GWAS (Part I) . . . 26
- 2.5 Multiple instances of problems: GWAS (Part II) . . . 30
- 2.6 Cost analysis . . . 32
- 2.7 Code generation: Matlab and Fortran . . . 33
- 2.8 Scope and limitations . . . 38
- 2.9 Summary . . . 39

3 CLAK: High-Performance BLAS and LAPACK Derivatives **41**
- 3.1 The challenge . . . 44
- 3.2 CLAK for high-performance derivatives . . . 46
  - 3.2.1 The derivative operator . . . 46
  - 3.2.2 The AD mode: Inference of properties and activity patterns . . . 47
- 3.3 An example: Differentiating $AX = B$ . . . 49
- 3.4 Experimental results . . . 52
  - 3.4.1 Example 1: Solution of a SPD linear system . . . 52
  - 3.4.2 Example 2: Symmetric rank update . . . 55
- 3.5 Summary . . . 57

4 CLICK 59
4.1 Automating FLAME: A three-stage approach . . . 60
4.2 Input to CLICK . . . 63
4.3 PME generation . . . 64
4.3.1 Operands partitioning . . . 64
4.3.2 Matrix arithmetic and pattern matching . . . 70
4.3.3 Non-uniqueness of the PME . . . 74
4.3.4 Recursive algorithms . . . 76
4.4 Loop invariant identification . . . 77
4.4.1 Decomposition of the PME . . . 78
4.4.2 Graph of dependencies . . . 79
4.4.3 DAG subsets selection . . . 81
4.4.4 A more complex example: the coupled Sylvester equation 84
4.5 Algorithm construction . . . 89
4.5.1 Repartitioning of the operands . . . 90
4.5.2 Predicates P_{before} and P_{after} . . . 91
4.5.3 Finding the updates . . . 95
4.5.4 The final algorithms and routines . . . 97
4.6 Towards a one-click code generation . . . 100
4.7 Scope and limitations . . . 100
4.8 Summary . . . 102

5 CLICK: High-Performance Specialized Kernels 103
5.1 A complete example: The derivative of the Cholesky factorization 104
5.1.1 Generation of the PME . . . 105
5.1.2 Loop invariant identification . . . 109
5.1.3 Algorithm construction . . . 112
5.1.4 The final algorithms . . . 115
5.1.5 Experimental results . . . 115
5.2 The derivative of TRSM . . . 119
5.2.1 Experimental results . . . 120
5.3 Summary . . . 121

6 Related Work 123

7 Conclusions 129
7.1 Results . . . 129
7.2 Future work . . . 131

A BLAS and LAPACK Routines 133

B Code Samples **135**
B.1 gSPDSOLVE 135
B.2 gSYRK 136
B.3 gCHOL 137
B.4 gTRSM 141

Chapter 1

Introduction

This dissertation focuses on the design and the implementation of domain-specific compilers for linear algebra matrix equations. Matrix equations constitute the computational bottleneck of most scientific and engineering applications; the development of efficient libraries for such equations has proven to be a complex and time consuming task that requires expertise in a variety of areas, from the application domain, through numerical analysis, to high-performance computing. A typical development process begins with an application expert modeling a problem in terms of matrix equations, continues with the discovery of efficient algorithms to solve them, and completes with the writing of high-performance code; normally, the process requires the collaboration of several people for months or even years. With our compilers, we aim to relieve the developers from both designing algorithms and writing code, and to generate routines that match or even surpass the performance of those written by human experts.

Given a target problem in terms of one or more matrix equations, no further work is required only if these equations can be solved directly by an existing high-performance library (e.g., LAPACK [2]). Most often, however, libraries do not offer routines that take advantage of domain knowledge, and their use results in suboptimal solutions; the burden is thus shifted to the developers who have to modify or extend existing libraries to tailor the computation to their needs. We present here two examples of such equations.

Example 1: The genome-wide association study. At the heart of the genome-wide association study (GWAS),[1] an important problem in computational biology [36], lies the generalized least-squares (GLS) problem:

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

where $M \in R^{n \times n}$, $X \in R^{n \times p}$, and $y \in R^n$; $b \in R^p$ is the sought-after solution. While LAPACK offers routines for closely related problems, such as the ordinary least-

[1] When carried out by means of the variance components method based on linear mixed models [3, 71].

squares $b := (X^T X)^{-1} X^T y$, some effort is needed to extend it to solve a GLS. Moreover, GWAS requires the solution of not one single GLS, but many of them; specifically, it requires solving the two-dimensional grid of $m \times t$ GLSs

$$b_{ij} := (X_i^T M_j^{-1} X_i)^{-1} X_i^T M_j^{-1} y_j, \qquad \text{with } 1 \leq i \leq m \text{ and } 1 \leq j \leq t, \tag{1.1}$$

where M_j is built as a function of a matrix Φ, the identity matrix I, and a scalar h_j: $M_j := h_j \Phi + (1 - h_j) I$. The key to an efficient solver is to exploit all the available knowledge: The specific structure of the matrix M_j, and the interdependence among the GLS problems that allows the reuse of computation across them. Unfortunately, this knowledge is more complex than what traditional libraries may take. In fact, the only alternative offered by these libraries is the computation of $m \times t$ such GLSs independently, in a black box fashion. In practice, the computational cost of this approach makes it unfeasible. The burden of designing competitive routines is put on the user.

Example 2: The derivative of Cholesky. The derivative of the Cholesky factorization (gCHOL) represents a building block required, for instance, in sensitivity analyses and optimization problems [59]. One way of computing this derivative is to solve the equation

$$GL^T + LG^T = B \tag{1.2}$$

for unknown G, where L and $G \in R^{n \times n}$ are lower triangular, and $B \in R^{n \times n}$ is symmetric. Even though traditional libraries offer solvers for closely-related operations, none of them supports the solution of Equation (1.2). For this type of operations, an unexperienced programmer is likely to only find inefficient unblocked algorithms; the discovery of efficient blocked algorithms, which enable data reuse to overcome the memory bandwidth bottleneck, requires a high-performance expert.

To further complicate matters, multiple algorithms may exist to solve one single target equation. It is well known that the performance of an algorithm depends on multiple factors, including the problem size and the underlying architecture, and that in general no single algorithm performs best in all scenarios. Therefore, to attain high performance in a range of scenarios, it is desirable to develop not one but a family of algorithms. This adds extra complexity to the development task: Multiple routines must be coded and maintained.

Since the mechanisms required for the management of domain knowledge and the derivation of families of algorithms are beyond the scope of traditional general-purpose compilers, we focus on the development of domain-specific linear algebra compilers. Our goal is to allow application experts to reason at the matrix equation level, and to relieve them from the design and the coding of algorithms. Following the approach of other domain-specific compilers, we provide a high-level interface, in the form of a domain-specific language, that allows the expert to express the matrix equations to be solved, together with as much knowledge as possible. Then,

our compilers take this information and automatically produce efficient solvers. More specifically, they take care of the exploitation of knowledge, the derivation of families of algorithms, the efficient mapping onto library-supported building blocks, and the code generation. In short, the users are able to state *what* needs to be solved, providing as much domain knowledge as possible, and delegating the compilers to find *how* to efficiently solve it.

Our ultimate goal is not different from that of the first compilers [37]. At that time, the community was skeptical because obviously a human expert could generate better code than a program (compiler). Nowadays, nobody conceives computer science, and especially the branch of programming, without general-purpose compilers and interpreters. Due to the complexity of matrix equations, similar and even stronger objections may be raised against our work. However, while the community concern is understandable, we argue that a) the loss in performance will be, in general, marginal, b) the gain in productivity is substantial, and c) for non-trivial problems, especially given that it is desirable to generate families of algorithms, it is very likely that our compilers find algorithms that even experts would miss.

1.1 Our compilers: A short overview

Applications require customized routines for both high-level matrix equations (e.g., Example 1), and specialized building blocks (e.g., Example 2). Since the concepts behind the algorithms for these two classes of operations are different, we developed two different prototypes of domain-specific compilers: 1) CLAK, for matrix equations, and 2) CLlCK, for building blocks.

1.1.1 CLAK

CLAK is our compiler for matrix equations comprising, for instance, linear systems, matrix inversions, and least-squares-like problems. With CLAK, we aim at modeling the reasoning of a human expert for the derivation of algorithms, and extending it with computers' exploration power. The approach may be described as follows: The same way that a traditional compiler breaks a program into assembly instructions directly supported by the processor, attempting different types of optimizations, CLAK breaks a target operation down to library-supported kernels, tailoring the algorithm to the application. In general, the decomposition is not unique, and the number of possible algorithms may be large; our compiler makes use of knowledge to prune the space of algorithms during the search, yielding only the most promising ones.

Example: The genome-wide association study. CLAK takes as input the description of a target operation expressed in a high-level mathematical notation (which we discuss in Section 2.1); for instance, Box 1.1 contains the description of the GWAS equation. Given this input, CLAK generates a family algorithms that

cast the computation in terms of kernels from the BLAS and LAPACK libraries. Algorithms 1.1 and 1.2 are two members of the family; in brackets, we specify the kernel corresponding to each statement (for a list of acronyms and their meaning, please see Appendix A).

```
Equation GWAS

    Matrix X   <Input, FullRank, ColumnPanel>;
    Vector Y   <Input>;
    Scalar h   <Input>;
    Matrix Phi <Input, SymmetricLower>;
    Vector b   <Output>;

    Matrix M   <Intermediate, SPD>;

    b{ij} = inv( trans(X{i}) * inv(M{j}) * X{i} ) *
                trans(X{i}) * inv(M{j}) * y{j};
    M{j}  = h{j} * Phi + (1 - h{j}) * I;
```

Box 1.1: Description of the GWAS equation.

Algorithm 1.1: GWAS variant 1.

1 **for** j **in** 1:t
2 $M_j := h_j\Phi + (1-h_j)I$ (SC-ADD)
3 $LL^T = M_j$ (POTRF)
4 $y_j := L^{-1}y_j$ (TRSV)
5 **for** i **in** 1:m
6 $W := L^{-1}X_i$ (TRSM)
7 $S := W^TW$ (SYRK)
8 $GG^T = S$ (POTRF)
9 $b_{ij} := W^Ty_j$ (GEMV)
10 $b_{ij} := G^{-1}b_{ij}$ (TRSV)
11 $b_{ij} := G^{-T}b_{ij}$ (TRSV)

Algorithm 1.2: GWAS variant 2.

1 $Z\Lambda Z^T = \Phi$ (SYEVR)
2 **for** i **in** 1:m
3 $K_i := X_i^TZ$ (GEMM)
4 **for** j **in** 1:t
5 $D := h_j\Lambda + (1-h_j)I$ (SC-ADD)
6 $y_j := Z^Ty_j$ (GEMV)
7 **for** i **in** 1:m
8 $V := K_iD^{-1}$ (SCAL)
9 $A := VK_i^T$ (GEMM)
10 $QR = A$ (GEQRF)
11 $b_{ij} := Vy_j$ (GEMV)
12 $b_{ij} := Q^Tb_{ij}$ (ORMQR)
13 $b_{ij} := R^{-1}b_{ij}$ (TRSV)

We choose these two example algorithms for their practical relevance. We recall that GWAS computes the $m \times t$ grid of GLS problems displayed in Equation (1.1). In a typical study, m takes large values (from millions to hundreds of millions), and t is either 1 —*Scenario 1*— or ranges from thousands to hundreds of thousands —*Scenario 2*—. Even though the mathematical problem is the same for

both scenarios, its parameters are not; different ways of exploiting the available domain knowledge, result in radically different algorithms that suit best each specific case. As illustrated by Figure 1.1, while Algorithm 1.1 is best suited for Scenario 1 (very small values of t), Algorithm 1.2 is to be preferred for Scenario 2 (large values of t). A deeper discussion of this application is carried out in Section 2.6.

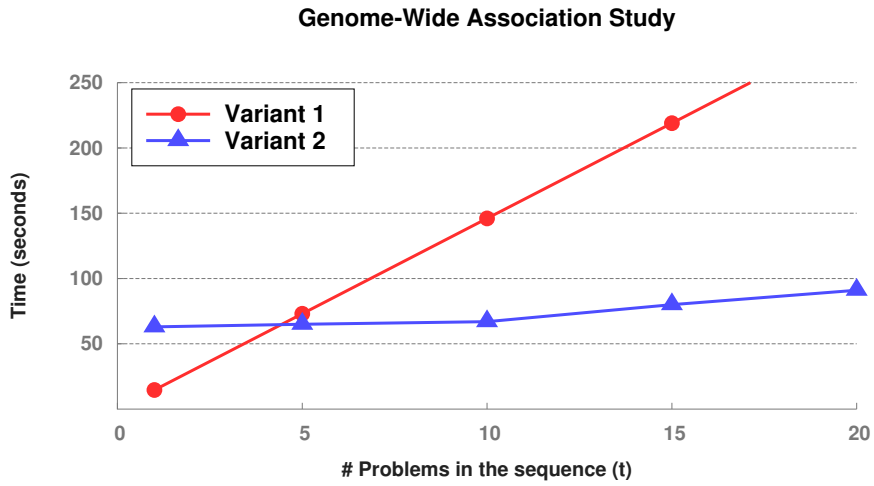


Figure 1.1: Performance of Algorithms 1.1 and 1.2 for GWAS. Values for n, p, and m are 1,000, 4, and 1,000,000, respectively.

1.1.2 CLICK

CLICK targets the generation of algorithms and code for building blocks such as matrix and vector products, and matrix factorizations. CLICK builds on a methodology born in the context of the FLAME project [33] for the generation of loop-based blocked algorithms. Based on formal methods, the methodology takes a high-level description of a target operation, and derives a family of provably correct algorithms. The key lies in finding the operation's Partitioned Matrix Expression (PME), a divide-and-conquer definition of the operation, from which a pool of loop invariants are identified; each loop invariant leads to a different algorithm, resulting in a family of them.

Example: The derivative of Cholesky. In line with FLAME's methodology, CLICK takes two predicates —Precondition and Postcondition— as formalism for the input. Box 1.2 contains the input to CLICK for the derivative of the Cholesky factorization (gCHOL); given this description, CLICK derives four algorithms to compute gCHOL. One example (Variant 1) is given in Figure 1.2 in FLAME notation; the complete family is collected in Section 5.1.4.

In Section 5.1.5, we carry out a thorough study of the performance of the algorithms. Here, we briefly discuss the need for deriving multiple variants. Figure 1.3

$$G = gChol(L,B) \equiv \begin{cases} P_{\text{pre}} : \{\texttt{Output}(G) \wedge \texttt{Input}(L) \wedge \texttt{Input}(B) \wedge \\ \qquad \texttt{Matrix}(G) \wedge \texttt{Matrix}(L) \wedge \texttt{Matrix}(B) \wedge \\ \qquad \texttt{LowerTriangular}(G) \wedge \texttt{LowerTriangular}(L) \wedge \\ \qquad \texttt{Symmetric}(B)\} \\ P_{\text{post}} : \{GL^T + LG^T = B\} \end{cases}$$

Box 1.2: Description of the derivative of the Cholesky factorization.

Partition $B \to \left(\begin{array}{c|c} B_{TL} & \star \\ \hline B_{BL} & B_{BR} \end{array}\right), L \to \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right), G \to \left(\begin{array}{c|c} G_{TL} & 0 \\ \hline G_{BL} & G_{BR} \end{array}\right)$

where $B_{TL}, L_{TL},$ and G_{TL} are 0×0

while $\texttt{size}(B_{TL}) < \texttt{size}(B)$ **do**

Repartition

$$\left(\begin{array}{c|c} B_{TL} & \star \\ \hline B_{BL} & B_{BR} \end{array}\right) \to \left(\begin{array}{c|c|c} B_{00} & \star & \star \\ \hline B_{10} & B_{11} & \star \\ \hline B_{20} & B_{21} & B_{22} \end{array}\right), \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \to \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \left(\begin{array}{c|c} G_{TL} & 0 \\ \hline G_{BL} & G_{BR} \end{array}\right) \to \left(\begin{array}{c|c|c} G_{00} & 0 & 0 \\ \hline G_{10} & G_{11} & 0 \\ \hline G_{20} & G_{21} & G_{22} \end{array}\right)$$

where $B_{11}, L_{11},$ and G_{11} are $b \times b$

$$\begin{array}{ll} G_{10} = B_{10} - L_{10}G_{00}^T & (\text{TRMM}) \\ G_{10} = G_{10}L_{00}^{-T} & (\text{TRSM}) \\ G_{11} = B_{11} - G_{10}L_{10}^T - L_{10}G_{10}^T & (\text{SYR2K}) \\ G_{11} = \text{gChol}(G_{11}, L_{11}) & (\text{gCHOL}) \end{array}$$

Continue with

$$\left(\begin{array}{c|c} B_{TL} & \star \\ \hline B_{BL} & B_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} B_{00} & \star & \star \\ \hline B_{10} & B_{11} & \star \\ \hline B_{20} & B_{21} & B_{22} \end{array}\right), \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \left(\begin{array}{c|c} G_{TL} & 0 \\ \hline G_{BL} & G_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} G_{00} & 0 & 0 \\ \hline G_{10} & G_{11} & 0 \\ \hline G_{20} & G_{21} & G_{22} \end{array}\right)$$

endwhile

Figure 1.2: One of the four algorithms generated by CLlCK for the derivative of the Cholesky factorization.

contains performance results using one thread (left) and eight threads (right). We make two points: First, despite the fact that all variants have the same computational cost, we observe differences in performance of up to 2.4x; second, while Variants 2 and 3 attain the best performance when one single thread is used, a different one, Variant 4, performs best in the multi-threaded case.

1.2 Contributions

This dissertation makes the following contributions.

- **Methodology for the derivation of algorithms for matrix equations.** We propose a methodology inspired by the reasoning of a human expert. The

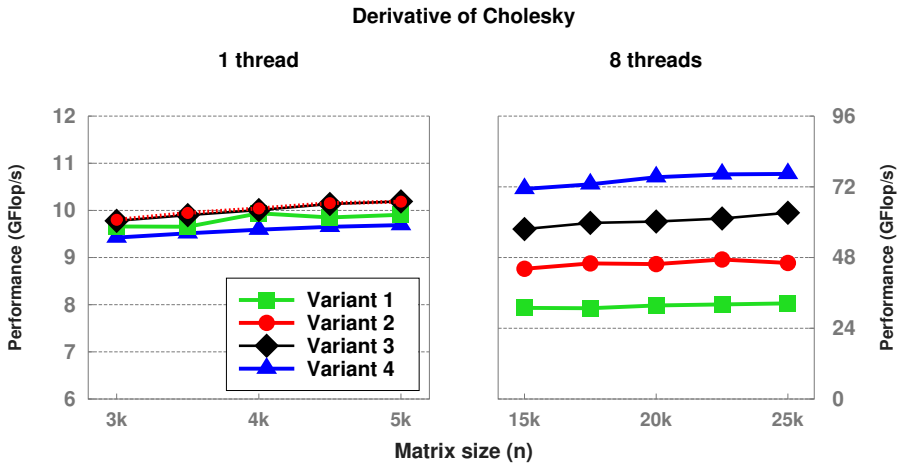


Figure 1.3: Performance of the four variants for the derivative of Cholesky.

methodology takes into account knowledge from the application domain, numerical linear algebra, and high-performance libraries; given the mathematical specification of a target equation, it yields a family of algorithms that exploit as much available domain knowledge as possible. The algorithms produced consist in efficient mappings onto a sequence of optimized library-supported kernels.

- **Automatic generation of algorithms and code for matrix equations.** We introduce CLAK, a prototype of linear algebra compiler that automates the application of the aforementioned methodology for matrix equations. For well-studied operations, including linear systems, matrix inversions, and least-squares problems, CLAK finds already known algorithms; here, we concentrate on 1) the discovery of novel algorithms for challenging equations, and 2) the application of CLAK to problems that require customized libraries comprising a large number of routines, where a manual approach is unfeasible. As case studies, we make use of the genome-wide association study (GWAS), and derivative operations arising in the context of algorithmic differentiation (AD).

- **Complete automation of FLAME's methodology.** The FLAME project provides a methodology for the systematic derivation of loop-based algorithms for a class of linear algebra building blocks. While FLAME researchers believed that the methodology could be fully mechanized, i.e., automatically carried out by a computer, little evidence existed. We introduce a second compiler, CLlCK, which demonstrates that the automatic application of the methodology is indeed feasible. We illustrate the application of CLlCK to standard operations, such as the LU factorization and the triangular Sylvester equation, and to unsupported kernels arising in AD, such as the

derivative of the Cholesky factorization.

- **Knowledge management.** We implement an engine to manipulate the available knowledge associated to the target problem, and to dynamically deduce structure and properties of expressions from that of individual operands. This engine encodes linear algebra rules and theorems related to properties such as matrix positive-definiteness, orthogonality and rank. Such a knowledge management is often sought after when performing symbolic computations; however, even the most advanced computer-algebra systems, such as Mathematica [70] and Maple [45], lack powerful deduction modules. Thanks to the knowledge management engine, our compilers can produce highly customized routines.

Two more contributions are made to specific fields of computational science and engineering.

- **Algorithmic Differentiation.** We perform a thorough performance comparison of the routines generated by our compilers with those of ADIFOR (a characteristic AD tool) for the derivative of a number of BLAS and LAPACK operations. The results quantify how much AD tools can benefit from the application of the techniques described in this dissertation when targeting linear algebra problems. Furthermore, the use of characteristic operations, such as linear systems, matrix factorizations, and matrix products, provide evidence that our compilers are capable of generating a differentiated version of a subset of LAPACK and of the entire BLAS.

- **Genome-wide association studies.** The application of CLAK to Equation (1.1) yielded novel algorithms for the computation of linear mixed-models in the context of genome-wide association studies. These algorithms are now included in the OmicABEL package[2] as part of the widely-used R library GenABEL [4], and are considered the state-of-the-art.

1.3 Outline of the thesis

The organization of this dissertation follows. Chapters 2 and 3 are devoted to CLAK. Chapter 2, introduces the compiler's engine. The input language and the class of accepted matrix equations are presented in Section 2.1. Section 2.2, discusses the heuristic-based model of the human expert reasoning, while Sections 2.3 and 2.5 uncover the core modules of the compiler's engine that support the application of these heuristics. Cost analysis and code generation are discussed in Sections 2.6 and 2.7, respectively.

Chapter 3 is dedicated to showing the broad applicability and extensibility of the compiler; to this end, we target the generation of libraries for the derivative

[2] `http://www.genabel.org/packages/OmicABEL`

of BLAS and LAPACK operations, as they arise in the field of algorithmic differentiation. The challenge behind the development of such libraries is discussed in Section 3.1. The extension to CLAK for the support of derivative operations is presented in Section 3.2. Section 3.4 carries out a deep comparison of CLAK's routines with those produced by ADIFOR to evaluate CLAK's potential.

Chapters 4 and 5 concentrate on CLlCK. Chapter 4 details thoroughly the steps behind the complete automation of the FLAME methodology. An overview of the approach is given in Section 4.1, while Sections 4.3 through 4.5 are dedicated to each of the three major steps in the process: 1) The generation of PMEs, 2) the identification of loop invariants, and 3) the construction of the algorithms. LAPACK and RECSY[3] operations are used as examples to illustrate the application of CLlCK.

Chapter 5 focuses on showing the application of CLlCK to non-standard operations. Section 5.1, gives a complete example of CLlCK's application to the derivative of the Cholesky factorization, while Section 5.2 provides a more general example by means of a kernel representative of many other specialized kernels. In both cases, performance results evidence that CLlCK not only increases productivity, but it also delivers efficient customized routines.

Chapter 6 reviews related work, and Chapter 7 concludes with a summary of the results of this thesis and a discussion of open research directions.

[3] RECSY is a specialized library for control theory equations.

Chapter 2

CLAK

We introduce CLAK, a compiler for linear algebra matrix equations. CLAK takes as input the mathematical description of a target equation together with domain knowledge, and returns a family of algorithms and routines that solve the equation. The goal is to replicate the exploitation of knowledge and the optimizations carried out by a human expert, and combine them with the computational power of a computer to address from simple to challenging problems.

Simple examples of matrix operations are $x := Q^T L y$, $b := (X^T X)^{-1} v$, and $B_i := A_i^T M^{-1} A_i$; in all cases, the quantities on the right-hand side of the assignment are known (matrices in capital letters and vectors in lower case), and the left-hand side has to be computed. Despite their mathematical simplicity, these operations pose challenges so significant that even the best tools for linear algebra produce suboptimal results. For instance, Matlab[1] uses a cubic—instead of quadratic—algorithm in the first equation, incurs possibly critical numerical errors in the second one, and fails to reuse intermediate results—and thus save computation—in the last one.

Let us take a closer look at $x := Q^T L y$, with $Q, L \in R^{n \times n}$, and $x, y \in R^n$: Algorithms 2.1 and 2.2 display two alternative ways of computing x. In the algorithm on the left, the one used by Matlab,[2] the input equation is decomposed into a GEMM (matrix-matrix multiplication), followed by a GEMV (matrix-vector multiplication), for a total of $O(n^3)$ floating point operations (flops); the algorithm on the right, generated by CLAK, instead maps the equation onto two GEMVs, for a cost of $O(n^2)$ flops. The difference lays in how the input operation is decomposed and mapped onto available kernels. In more complex matrix equations, it is not uncommon to face dozens and dozens of alternative decompositions, all corresponding to viable, but not equally effective, algorithms. We will illustrate how unfruitful branches can be avoided by propagating knowledge, as the algorithm unfolds, from the input operands to intermediate results.

[1] When comparing to Matlab, we refer to version 7.11.0.584 (R2010b).

[2] With the extra help of parentheses, Matlab can be forced to use Algorithm 2.2. We highlight the fact that in absence of user's intervention, Matlab falls back on a suboptimal algorithm.

Algorithm 2.1: Matlab's algorithm for $x := Q^T L y$	
$T := Q^T L$	(GEMM)
$x := Ty$	(GEMV)

Algorithm 2.2: CLAK's alternative algorithm for $x := Q^T L y$	
$t := Ly$	(GEMV)
$x := Q^T t$	(GEMV)

Challenging matrix equations appear in applications as diverse as machine learning, sensitivity analysis, and computational biology. In most cases, one has to solve not one instance of the problem, but thousands or even billions of them. A characteristic example is given by the aforementioned computation of mixed models in the context of the genome-wide association study, which requires the solution of up to 10^{12} (trillions) instances of the generalized least-squares problem

$$b := (X^T M^{-1} X)^{-1} X^T M^{-1} y, \quad \text{where} \quad M = h^2 \Phi + (1 - h^2) I.$$

These instances are related to one another suggesting that intermediate results could be saved and reused; unfortunately, none of the currently available libraries allows this.

We developed CLAK with the objective of overcoming the deficiencies discussed so far. Very much like a standard compiler takes a computer program and maps it onto the instruction set provided by the processor, our approach is to decompose the input equations into kernels provided by linear algebra libraries such as BLAS and LAPACK. The mapping is, in general, not unique, and the number of alternatives may be very large. For this reason, our compiler carries out a search within the space of possible algorithms, and yields the most promising ones. The search is guided by a number of heuristics which, in conjunction with a mechanism for inferring properties, aim at simulating the thought-process of an expert in the field. Moreover, by means of dependency analyses, CLAK actively seeks to avoid redundant computation, both within a single equation and across multiple instances of them.

The application of these techniques heavily relies on pattern matching and symbolic computations; due to its powerful engine for pattern matching and expression rewriting, we chose Mathematica [70] to implement the compiler's engine. In this chapter, we discuss the mechanisms incorporated into CLAK to automate the process of generation of algorithms and code: From the input to the compiler, through the mapping onto building blocks and knowledge management, to the code generation.

2.1 Defining input equations

We consider equations that involve scalar, vector and matrix operands, combined with the binary operators "$+$" (addition) and "$*$" (multiplication, used both for scaling and matrix products), and the unary operators "$-$" (negation), "T" (transposition), and "-1" (inversion, for scalars and square matrices). Equations come

with what we refer to as *knowledge*: Each operand is annotated with a list of zero or more properties such as "square", "orthogonal", "full rank", "symmetric", "symmetric positive definite", "diagonal", and so on. Additionally, we allow operands to be subscripted, indicating that the problem has to be solved multiple times. As an example, Box 2.1 illustrates the description of the solution of multiple linear systems that share the same symmetric coefficient matrix: $x_i := A^{-1}b_i$.

```
Equation MultSymmSolve
    Matrix A <Input, SymmetricLower>;
    Vector b <Input>;
    Vector x <Output>;

    x{i} = inv(A) * b{i};
```

Box 2.1: Description of the solution of multiple linear systems $x_i := A^{-1}b_i$, with a common symmetric coefficient matrix A. The description includes the definition of the operands together with their properties, and the target equation where the operands are labeled with the corresponding subindex.

The notation is straightforward: First, the operands are declared specifying their type and a number of properties; then, the equation to be solved is stated in a high-level notation similar to that of Matlab. Essentially, the valid equations are formed by a left-hand side consisting of an output operand, the unknown, and a right-hand side consisting of input operands combined using the aforementioned operators. In the remainder of this section, we briefly formalize the syntax of the language accepted by our compiler, and thus the class of accepted input equations. These details do not affect the description of the mechanisms behind CLAK; readers mainly interested in the generation of algorithms may skip the rest of this section.

As in a natural language, CLAK defines a collection of admissible words, the *tokens*. These tokens are grouped in lexical categories, as shown in Table 2.1; for each category (indicated in the left column), we provide the regular expression that defines it (middle column) and its meaning (right column). We use the traditional notation for the specification of regular expressions:

- `*` matches zero or more occurrences of the expression on the left,
- `+` matches one or more occurrences of the expression on the left,
- `?` matches zero or one occurrences of the expression on the left,
- `|` matches one of the many options,
- `[]` indicates a class of characters, e.g., `[a-z]` matches any lower case letter, and

- `"[*"` indicates the literals within the quotes, in this case an opening square bracket followed by a star.

Category	**Regular Expression**	**Meaning**
id	`[a-zA-Z][a-zA-Z0-9_]*`	Identifier
optype	`Scalar` | `Vector` | `Matrix`	Type of operand
iotype	`Input` | `Output` | `InOut` | `Intermediate`	I/O type of the operand
property	`Square` | `ColumnPanel` | `RowPanel` | `Diagonal` | `LowerTriangular` | `UpperTriangular` | `Symmetric` | `SymmetricLower` | `SymmetricUpper` | `SPD` | `SPDLower` | `SPDUpper` | `Orthogonal` | `FullRank`	Properties of the operands
subscript	`{[a-z](,[a-z])*}`	Operand subscripts
number	`[0-9]+(\.[0-9]+)? ([Ee][+-]?[0-9]+)?`	Numeric constant
opeq	`=`	Equality operator
opadd	`[+-]`	Addition operator
opmul	`[*]`	Multiplication operator
unary	`trans` | `inv`	Transpose and inverse operators
init	`init`	Initial contents of an InOut operand

Table 2.1: Lexical categories of CLAK's input language.

We clarify the meaning of a few keywords included in the *iotype* and the *property* lexical categories:

iotype. The input/output (I/O) type of the operands may be not only `Input` or `Output`, but also `InOut` and `Intermediate`. `InOut` is used to specify operands that are overwritten; this type is used in conjunction with the macro `init`, which specifies the initial contents of the operand. For instance, the equation

```
A = 3 * init(A),
```

scales `InOut` matrix `A` and overwrites it with the result.

The `Intermediate` type is used for temporary operands, and it allows the specification of properties of expressions that are empirical, i.e., properties that cannot be deduced analytically. An example of `Intermediate` operand will appear in Section 2.4.

property. Typically, symmetric matrices are only partially stored in either their upper or lower triangle; the suffixes `Lower` and `Upper` for symmetric and SPD matrices indicate in which part of the matrix is data actually stored, and are essential for the generation of code. The `ColumnPanel` and `RowPanel` keywords are used for rectangular matrices: A matrix $A \in R^{m \times n}$ is a column panel if $m > n$, i.e., it has more rows than columns, while it is a row panel if it has more columns than rows.

Tokens are combined to construct program sentences or *statements*. The structure of a correct CLAK program is specified by means of a grammar G, defined as the quadruple $G = (N, \Sigma, P, \langle S \rangle)$, where

- N is the set of *non-terminal* symbols (<Model>, <OpDecl>, <Equation>, <Expression>, <Term>, <Factor>).
- Σ is the set of *terminal* symbols (id, optype, iotype, property, subscript, number, opeq, opadd, opmul, unary, init).
- $P \subseteq N \times (N \cup \Sigma)^*$ is the set of *production rules* (provided in Box 2.2).
- $\langle S \rangle \in N$ is the *starting symbol*. In this case, the starting symbol is <Model>.

As is the case for most programming languages, this is a context-free grammar [69]. More specifically, it is an LLR(1) grammar: **L**eft-to-right parsing, **L**eftmost derivation, with **R**egular expressions in the production's right-hand sides, and **1** lookahead token.

Any program that can be produced by means of the production rules, starting from the non-terminal symbol <Model>, and only containing terminal symbols, is grammatically correct. However, similarly to natural languages, not every grammatically-correct statement is meaningful. Therefore, a number of semantic rules are needed. We will not get into the details of the semantic analysis, but we emphasize that these are the rules in charge to assert, for instance, that the left-hand side operand is an output operand, and the expression in the right-hand side only includes input operands.

Finally, we recall CLAK is written in Mathematica; therefore, we also developed a parser that reads a file containing the description of a target operation in CLAK's language and translates it into a Mathematica representation accepted by CLAK's core.

<Model>	→	`"Equation"` **id**
		(<OpDecl> `";"`)$^+$
		(<Equation> `";"`)$^+$
<OpDecl>	→	**optype id** `"<"` **iotype** (`","` **property**)*`">"`
<Equation>	→	**id opeq** <Expression>
<Expression>	→	<Term> (**opadd** <Term>)*
<Term>	→	<Factor> (**opmul** <Factor>)*
<Factor>	→	**id** (**subscript**)$^?$ |
		number |
		`"("` <Expression> `")"` |
		unary `"("` <Expression> `")"` |
		init `"("` **id** `")"` |
		(**opadd**)$^+$ <Factor>

Box 2.2: Production rules of the grammar defining CLAK's language. String literals within double quotes must be found as-is in the input program.

2.2 Heuristics for the generation of algorithms

Starting from a target equation, CLAK explores a subset of the space of possible algorithms, dynamically generating a "tree of decompositions". For instance, Figure 2.1 contains the complete tree generated for the solution of a linear system, when the coefficient matrix is symmetric positive definite (SPD). The root node corresponds to the input equation $x := A^{-1}b$, and every edge represents the mapping onto a building block; in the example, the three branches are originated by three different factorizations of the matrix A: a Cholesky factorization, a QR factorization, and an eigendecomposition. Once the process is over, the operations along the edges from the root to each leaf constitute a valid algorithm. In practice, the tree is built in two phases, corresponding to the blue (dark) and green (light) nodes, respectively. In the first phase, CLAK deals with the inverse operator via matrix factorizations; in the second phase, the decomposition completes with the mapping of expressions onto kernels. In order to limit the size of the tree, the compiler uses the heuristics described hereafter.

2.2.1 Dealing with the inverse operator

The inversion of matrices is a delicate operation. There are only rare occurrences of problems in which one is interested in the actual matrix inverse; most often, the operation appears in the context of linear systems, least squares problems, or more

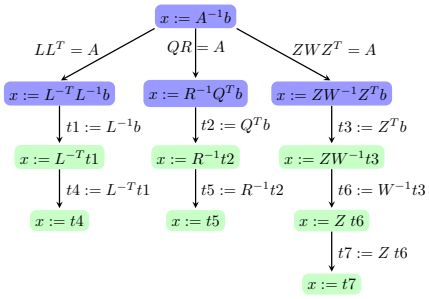


Figure 2.1: Full tree spawned by the compiler when processing the solution of a linear system of equations $x := A^{-1}b$, with an SPD coefficient matrix A, and a single right-hand side b.

complex expressions; in the majority of cases, the inversion can—and should—be avoided altogether. Because of this, CLAK splits the generation of algorithms in two phases, the first of which is solely devoted to the treatment of inverses; the objective is to reduce the input equations to an expression in which the inverse is only applied to matrices in factored form, i.e., triangular or diagonal (see blue subtree in Figure 2.1). In the second phase, the resulting expression is mapped onto computational kernels (see green branches in Figure 2.1).

This first phase takes as input the target equation, and generates the subtree characterized by leaf nodes that require no further treatment of the inverses. This is an iterative process in which the tree is constructed in a breadth-first fashion; at each iteration, the current expression is inspected for inverse operators, the innermost of which is handled. The inversion is applied to either a full matrix, such as A^{-1}, or to a non-simplifiable expression, e.g., $(A^TA)^{-1}$ with A rectangular.

Inversion of a full matrix. In the first case, the matrix is factored by means of one or more of the many matrix decompositions provided by LAPACK, but instead of exhaustively trying all possibilities, the factorizations are chosen according to the properties of the matrix. For instance, if A is a symmetric positive definite matrix, we limit the viable options to the Cholesky factorization ($LL^T = A$), the QR factorization ($QR = A$), and the eigendecomposition (ZWZ^T); vice versa, the LU ($LU = A$), and LDL ($LDL^T = A$) factorizations are not considered. As depicted in Figure 2.2, the compiler constructs as many branches as factorizations, while altering the initial expression. All the branches are subsequently explored.

Limiting the search to a subset of all possible factorizations has two advantages: On the one hand, non-promising algorithms are discarded and the search space is pruned early on; on the other hand, the algorithms are tailored to the specific properties of the application. Table 2.2 contains the set of factorizations currently used by CLAK, together with the matrix properties that enable them.

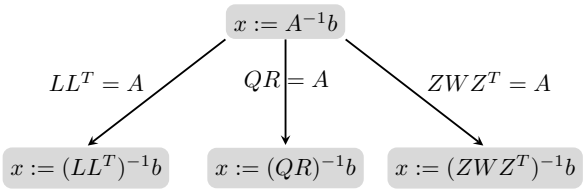


Figure 2.2: Solution of an SPD linear system. In the first iteration, CLAK considers three possible factorizations for the coefficient matrix A; three branches are originated, corresponding to a Cholesky factorization (left), a QR factorization (middle), and an eigendecomposition (right).

Matrix Property	**Factorizations**
Symmetric	LDL, QR, Eigendecomposition
SPD	Cholesky, QR, Eigendecomposition
Column Panel (FullRank)	QR
Column Panel (RankDef)	SVD
Row Panel (FullRank)	LQ
Row Panel (RankDef)	SVD
General	LU, SVD

Table 2.2: Factorizations currently used by CLAK, and matrix properties that enable them.

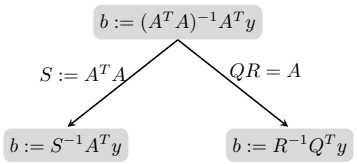


Figure 2.3: Snippet of the tree spawned by CLAK when processing the ordinary least-squares equation $b := (A^TA)^{-1}A^Ty$, where $A \in R^{m\times n}(m > n)$ is full rank.

Inversion of an expression. We concentrate now on the case of an inverse operator applied to a non-simplifiable expression. A characteristic example comes from the normal equations, arising for instance as part of the ordinary least-squares problem

$$b := (A^TA)^{-1}A^Ty, \tag{2.1}$$

where $A \in R^{m\times n}$ (with $m > n$) is full rank. In this scenario, as depicted in Figure 2.3, our compiler explores two alternative routes: 1) the multiplication of the expression A^TA, thus reducing it to the inverse of a single SPD operand S; and 2) the decomposition of one of the matrices in the expression, in this case A, thus spawning a branch per suitable factorization. As dictated by Table 2.2, in Equation (2.1) A is decomposed by means of a QR factorization.

The treatment of inverses continues until the inverse operator is only applied to triangular or diagonal matrices. For the example in Figure 2.3, the left branch would be further processed by factoring the matrix S, yielding three more nodes; the right branch instead, since R is a triangular matrix, is complete.

2.2.2 Mapping onto kernels

The goal of this second phase is to find efficient mappings from expressions to kernels provided by numerical libraries, e.g., BLAS and LAPACK. The number of possible mappings grows exponentially with the number of operators in the expression, therefore heuristics are necessary to constrain the amount of explored alternatives. We discuss two examples of such heuristics.

Common segments. The objective is to reduce the complexity of the algorithms by avoiding redundant computations; common segments of the expression are identified, thus allowing the reuse of intermediate results. We emphasize that this is by no means a trivial optimization. In fact, even for the simplest cases, sophisticated tools such as Matlab do not adopt it. For instance, when evaluating the operation

$$\alpha := x^Tyx^Ty,$$

where x and y are vectors of size n, Matlab makes use of Algorithm 2.3, which computes the products from left to right; CLAK instead recognizes that the expression x^Ty appears twice, and generates Algorithm 2.4, reducing the number of flops from $5n$ to $2n$.

Algorithm 2.3: Matlab's computation for $\alpha := x^T y x^T y$

```
t1 := x' * y
t2 := t1 * x'
alpha := t2 * y
```

Algorithm 2.4: CLAK's algorithm for $\alpha := x^T y x^T y$

```
t1 := x' * y
alpha := t1 * t1
```

More challenging is the case where one of the occurrences of the common segment appears in transposed or inverted form. As an example, let us consider the expression

$$v := X^T L^{-1} L^{-T} X,$$

where both operands X and L are matrices, and L is triangular. In order to recognize that $L^{-T}X$ is the transpose of $X^T L^{-1}$, and in general, to recognize that two segments are the negation, inverse, or transpose of one another, our compiler incorporates a large set of ground linear algebra knowledge. This is covered in Section 2.3.

Prioritization. In an attempt to minimize the cost of the generated algorithms, the kernels available to the compiler are classified according to a precedence system. In Table 2.3, we give an example of a subset of these kernels, sorted from high to low priority. The precedences are driven by the dimensionality of the operands in the kernels: The idea is to reduce the number of required flops by keeping the dimensionality of the resulting operands as low as possible. [3] For the example in Table 2.3, the first two kernels reduce the dimensionality of the output operand with respect to that of the input, while the third kernel maintains it, and the fourth increases it. Finally, the inversion of a triangular matrix is given the lowest precedence: A matrix will only be inverted if no other option is available.

The benefits of the prioritization were already outlined in this chapter's introduction (Algorithms 2.1 and 2.2): There, by favoring the matrix-vector over the matrix-matrix product, the complexity was lowered by an order of magnitude. Here, we provide more examples. Consider the operation

$$\alpha := x^T z x^T y,$$

where x, y, and z are vectors, and α is a scalar. When inspecting the expression for kernels, CLAK finds two inner products ($x^T z$, and $x^T y$), and one outer product (zx^T). While all three options lead to valid algorithms, the inner products are favored, producing, for instance, Algorithm 2.5; the cost of this algorithm is $O(n)$ flops, instead of a cost of $O(n^2)$, had the compiler favored the outer product (Algorithm 2.6).

[3] The problem at hand—minimizing the cost of a sequence of (symbolic) matrix products—resembles the matrix chain multiplication problem. However, CLAK targets general size computations, and operates with symbolic sizes. Hence, it does not necessarily have enough information to determine whether the cost of a specific parenthesization has a lower cost than another.

#	Kernels	Example	Dim(in)	Dim(out)
1	inner product	$\alpha := x^T y$	1/1	0
2	matrix-vector operations	$y := Ax$, $b := L^{-1}x$	2/1	1
3	matrix-matrix operations	$C := AB$, $B := L^{-1}A$	2/2	2
4	outer product	$A := xy^T$	1/1	2
5	inversion of a triangular matrix	$C := L^{-1}$	–	–

Table 2.3: Example of the classification of kernels based on a system of precedences. The kernels that reduce the dimensionality of the output operands with respect to the input ones are given higher precedence. The inversion is only selected when no other option exists.

Algorithm 2.5: Computation of $\alpha := x^T z x^T y$ favoring inner products

```
t1 := x' * z
t2 := x' * y
alpha := t1 * t2
```

Algorithm 2.6: Computation of $\alpha := x^T z x^T y$ favoring outer products

```
T1 := z * x'
t2 := x' * T1
alpha := t2 * y
```

A third example is given by

$$\beta := v^T L^{-1} L^{-T} u,$$

where L is a square lower triangular matrix, and v and u are vectors. The inspection for kernels yields the following matches: $v^T L^{-1}$, L^{-1}, and $L^{-T}u$. Again, the inversion of L is avoided, unless no alternatives exist; this is captured by the precedences listed in Table 2.3, which give priority to the solution of linear systems over the inversion of matrices. Therefore, the second option (L^{-1}) is dismissed, and the compiler only explores the branches spawned by the first and third kernels. While the inversion of L would lead to a cubic algorithm (Algorithm 2.7), the ones generated (e.g., Algorithm 2.8) have a quadratic cost.

Algorithm 2.7: Computation of $\beta := v^T L^{-1} L^{-T} u$ favoring the inversion of matrices

```
T1 := inv(L)
t2 := v' * T1
t3 := t2 * T1'
beta := t3 * u
```

Algorithm 2.8: Computation of $\beta := v^T L^{-1} L^{-T} u$ favoring the solution of triangular systems

```
t1 := v' / L
t2 := L' \ u
beta := t1 * t2
```

Notice that if implemented naively, the rules discussed so far may lead to an infinite process: For instance, a matrix could be factored and built again, as in

$(A^TA)^{-1} \xrightarrow{QR=A} ((QR)^TQR)^{-1} \xrightarrow{A:=QR} (A^TA)^{-1}$; also, a matrix could be factored indefinitely, as in $A \xrightarrow{Q_1R_1=A} Q_1R_1 \xrightarrow{Q_2R_2=Q_1} Q_2R_2R_1 \xrightarrow{\cdots} \ldots \xrightarrow{Q_iR_i=Q_{i-1}} Q_iR_i \ldots R_2R_1$. To avoid such situations, our compiler incorporates a mechanism to measure and guarantee progress.

2.3 Compiler's engine

The availability of knowledge is crucial for a successful application of the heuristics. Equally important is the capability of algebraically manipulating expressions with the objective of simplifying them or finding common segments. Here, we detail the different modules that constitute CLAK's engine, and how these modules enable: 1) the algebraic manipulation of expressions, 2) the mapping onto building blocks, and 3) the management of both input and inferred knowledge.

2.3.1 Matrix algebra

The *Matrix algebra* module deals with the algebraic manipulation of expressions. It incorporates a considerable amount of knowledge regarding properties of the operators, such commutativity and distributivity, and linear algebra equalities, such as "the inverse of an orthogonal matrix equals its transpose". This knowledge is encoded as an extensive list of *rewrite rules* that allow the compiler to rearrange expressions, simplify them, and find subexpressions that are the inverse, transpose, etc, of one another.

A rewrite rule consists of a left-hand and a right-hand side. The left-hand side contains a pattern, possibly restricted via constraints to be satisfied by the operands; the right-hand side specifies how the pattern, if matched, should be replaced. For instance, the rule

$$Q^{-1} \wedge \text{Orthogonal}(Q) \rightarrow Q^T$$

reads as follows: The inverse of a matrix Q, provided that Q is orthogonal, may be replaced with the transpose of Q. Box 2.3 includes examples of rewrite rules relative to transposition, product, and inversion of matrices.

$$\begin{aligned}
(A \times B)^T &\rightarrow B^T \times A^T \\
Q^T \times Q \wedge \text{Orthogonal}(Q) &\rightarrow I \\
A \times I \wedge \neg\text{Scalar}(A) &\rightarrow A \\
(A \times B)^{-1} \wedge \text{Square}(A) \wedge \text{Square}(B) &\rightarrow B^{-1} \times A^{-1} \\
A^{-1} \times A &\rightarrow I
\end{aligned}$$

Box 2.3: Examples of rewrite rules included in the Matrix Algebra module. I is the identity matrix.

The example in Box 2.4 gives an idea of how the compiler is capable of eliminating unnecessary calculations by means of algebraic transformations. The initial expression is $(X^TX)^{-1}X^TL^{-1}y$; we assume X is a full rank column panel, and L a lower triangular matrix. As dictated by the heuristics presented in Section 2.2, one alternative in the processing of the expression is through a QR factorization of the matrix X: The symbol X is replaced by QR—line 2—(where Q and R are orthogonal and upper triangular, respectively), and a series of transformations are triggered. First, the transposition is distributed over the product—line 3—; next, due to the orthogonality of Q, the product Q^TQ is removed as it equals the identity—line 4—. Since R is square, the inverse may be distributed over the product R^TR resulting in $R^{-1}R^{-T}$—line 5—. Another simplification rule establishes that the product of a square matrix with its inverse equals the identity; because of this, the $R^{-T}R^T$ is removed—line 6—. After all these algebraic steps, the expression $((QR)^TQR)^{-1}(QR)^TL^{-1}y$ simplifies to $R^{-1}Q^TL^{-1}y$. Box 2.3 contains the necessary set of rewrite rules for this manipulation.

1) $b := (X^TX)^{-1}X^TL^{-1}y;$
2) $b := ((QR)^TQR)^{-1}(QR)^TL^{-1}y;$
3) $b := (R^TQ^TQR)^{-1}R^TQ^TL^{-1}y;$
4) $b := (R^TR)^{-1}R^TQ^TL^{-1}y;$
5) $b := R^{-1}R^{-T}R^TQ^TL^{-1}y;$
6) $b := R^{-1}Q^TL^{-1}y.$

Box 2.4: Example of expression simplification carried out by CLAK. X is a full rank column panel, and L a lower triangular matrix.

Rewrite rules are algebraic identities, i.e., they may be applied in both directions. For instance, the expression $(AB)^T$ may be rewritten as B^TA^T, and vice versa, leading to multiple equivalent representations for the same expression. Since this fact complicates the manipulation and identification of building blocks, one may be tempted to use rules as "always distributing the transpose over the product" for reducing every expression to a canonical form. Unfortunately, there exists no "best" representation for matrix expressions. Indeed, imposing a canonical form would lower the effectiveness of the compiler.

A typical example is given by the distribution of the product over the addition: $(A+B)C$ may be transformed into $AC+BC$ and vice versa, but neither representation is superior in all scenarios. Consider, for instance, the expression $\alpha xx^T+\beta yx^T+\beta xy^T$, where α and β are scalars, and x and y are vectors. In this format, it is straightforward to realize that the expression is symmetric—the first term is symmetric, and the second and third are one the transpose of the other—; if instead x^T is factored out as in $(\alpha x+\beta y)x^T+\beta xy^T$, the symmetry is not visible,

and redundant computation would be performed. This is an example in which the distribution of the product over the addition seems to be the choice to favor.

On the contrary, let us consider the expression in Box 2.5: $(ZWZ^T + ZZ^T)^{-1}$, where Z is square and orthogonal, and W is diagonal. Factoring Z and Z^T out—$(Z(W+I)Z^T)^{-1}$, where I is the identity matrix—is an indispensable first step towards the simplification of the expression. Next, since all matrices are square, the inverse may be distributed over the product, and the orthogonality of Z allows the rewriting of its inverse as its transpose, resulting in $Z(W+I)^{-1}Z^T$. This transformation—absolutely crucial in practical cases—is only possible thanks to the initial factoring; hence, this is a contrasting example in which the distribution of the product is not the best option. In light of this dichotomy, CLAK always operates with multiple alternative representations.

$$
\begin{aligned}
M &:= (ZWZ^T + ZZ^T)^{-1}; \\
M &:= (Z(W+I)Z^T)^{-1}; \\
M &:= Z^{-T}(W+I)^{-1}Z^{-1}; \\
M &:= Z(W+I)^{-1}Z^T;
\end{aligned}
$$

Box 2.5: Another example of expression manipulation carried out by the compiler. Z is square and orthogonal; W is diagonal.

2.3.2 Interface to building blocks

We have claimed repeatedly that the goal of CLAK is to decompose the target equation in terms of building blocks that can be directly mapped to library invocations; it remains to be discussed what are the available building blocks. The exact list is configurable, and is provided to the compiler via the *Interface to building blocks* module. This module contains a list of patterns associated to the corresponding computational kernels. As of now, this list includes a subset of the operations provided by BLAS and LAPACK, e.g., matrix products and the solution of linear systems; a sample is given in Box 2.6.

CLAK is by no means limited to this set of operations. Should an additional or a different set of building blocks be available, say RECSY [41, 42] or an extension of the BLAS library [16], this can be made accessible to the compiler with only minimal effort, by including in this module the corresponding patterns. For instance, in order to add support for the operation $w := \alpha x + \beta y$, as proposed in the extension to the BLAS library, we only need to incorporate the pattern

```
plus[ times[ alpha_, x_ ], times[ beta_, y_ ] ] /;
            isVectorQ[x,y] && isScalarQ[alpha, beta];
```

the compiler is then ready to make use of this building block in the generation of algorithms.

MATRIX PRODUCTS:

```
plus[times[alpha_, A_, B_], times[beta_, C_]]
plus[times[alpha_, trans[A_], B_], times[beta_, C_]]
plus[times[alpha_, trans[A_], A_], times[beta_, C_]]
times[A_, trans[A_]] /; isTriangularQ[A]
```

LINEAR SYSTEMS:

```
plus[times[inv[A_], B_]] /; isTriangularQ[A] && isMatrixQ[B]
plus[times[inv[A_], b_]] /; isTriangularQ[A] && isVectorQ[b]
```

Box 2.6: A snippet of the interface to available building blocks.

2.3.3 Inference of properties

Properties play a central role in the search for efficient algorithms; the more knowledge is available, the more opportunities arise for optimizations. A distinguishing feature of CLAK is the propagation of properties: We developed an engine for inferring properties of expressions from those of the individual operands. Thanks to this engine, the initial knowledge (from the input equation) is augmented dynamically.

This mechanism is activated every time a mapping takes place: 1) when mapping onto factorizations, properties are propagated from the input matrix to its factors; 2) when mapping onto other kernels, properties are propagated from the segment to the output quantity. The gained knowledge on the intermediate operands is then used by the compiler for further tailoring the algorithms. Boxes 2.7 and 2.8 provide examples of inference of knowledge in factorizations and products, respectively.

EIGENDECOMPOSITION ($ZWZ^T = A$):

Input	A: matrix, square, symmetric
Output	Z: matrix, square, orthogonal
	W: matrix, square, diagonal

QR ($QR = A$):

Input	A: matrix, column-panel, full rank
Output	Q: matrix, orthogonal, column-panel, full rank
	R: matrix, square, upper triangular, full rank

Box 2.7: Inference of properties for two representative factorizations.

It is important to notice that the inference of rules and the mapping onto kernels are completely independent actions. For instance, in the absence of the second rule in Box 2.8, CLAK would still be able to match a product of the form A^TA (provided the pattern is included in the *Interface to building blocks* module); however, if A is a full rank, column panel matrix, the compiler would not be able to infer, and then exploit, the positive definiteness of S.

$W := L^{-1}X$:		
	Input	L: matrix, square, full rank
		X: matrix, column-panel, full rank
	Output	W: matrix, column-panel, full rank
$S := W^T W$:		
	Input	W: matrix, column-panel, full rank
	Output	S: matrix, square, SPD

Box 2.8: Inference of properties for two mappings onto kernels.

We regard the inference engine as a growing database of linear algebra knowledge. In its current form, the database is populated with a sample of rules and theorems, but the flexible design of the module allows it to be easily extended with new inference rules.

2.4 A detailed example: GWAS (Part I)

We use the computationally challenging genome-wide association study (GWAS) problem to illustrate the potential of CLAK's engine and heuristics. We recall that, as part of GWAS, one has to solve the equation

$$\begin{cases} b_{ij} := (X_i^T M_j^{-1} X_i)^{-1} X_i^T M_j^{-1} y_j \\ M_j := h_j \Phi + (1 - h_j) I \end{cases} \quad \text{with} \quad \begin{array}{l} 1 \le i \le m \\ 1 \le j \le t, \end{array} \tag{2.2}$$

where X_i, y_j, h_j, and Φ are known quantities, and b_{ij} is sought after. The size and properties of the operands are as follows: $b_{ij} \in R^p$, $X_i \in R^{n \times p}$ is a full rank column panel ($n > p$), $M_j \in R^{n \times n}$ is symmetric positive definite, $y_j \in R^n$, $h_j \in R$, $\Phi \in R^{n \times n}$ is symmetric, and I is the identity matrix. Box 2.9 contains the representation of Equation (2.2) in CLAK's language.

Due to the complexity of GWAS, a large number of alternatives are generated. For the sake of this discussion, we focus now on the solution of a single instance of Equation (2.2), as if both m and t were 1, and defer the case of multiple instances to next section. In Figure 2.4, we provide a snippet of the tree spawned by CLAK while constructing algorithms. Among the dozens of different branches, we describe three representative ones.

At the root node, the compiler starts by dealing with the innermost inverse, M^{-1}, and equivalently, $(h_j \Phi + (1 - h_j) I)^{-1}$. As explained in Section 2.2, the options are either to reduce the expression to a single operand (M, which is known to be SPD), or to factor one of the matrices in the expression, in this case Φ. The former choice leads directly to Node 2 (modulo the order in which addition and scaling are performed), while the latter opens up a number of branches, corresponding to all the admissible factorizations of Φ; the middle branch in Figure 2.4 follows the eigendecomposition of Φ. One might argue that based on the available

```
Equation GWAS

    Matrix X   <Input, FullRank, ColumnPanel>;
    Vector Y   <Input>;
    Scalar h   <Input>;
    Matrix Phi <Input, SymmetricLower>;
    Vector b   <Output>;

    Matrix M   <Intermediate, SPD>;

    b{ij} = inv( trans(X{i}) * inv(M{j}) * X{i} ) *
                trans(X{i}) * inv(M{j}) * y{j};
    M{j}  = h{j} * Phi + (1 - h{j}) * I;
```

Box 2.9: CLAK's representation of the GWAS equation.

knowledge (M is SPD), the compiler should decide against the eigendecomposition, since a Cholesky factorization is about ten times as fast. In actuality, although the eigendecomposition is suboptimal for the solution of one single instance, in the general case (Equation (2.2)) it leads to the fastest algorithms of all [19].[4]

Let us now concentrate on the subtree rooted at Node 2. The input equation was reduced to $b := (X^T M^{-1} X)^{-1} X^T M^{-1} y$; again, CLAK looks for the innermost inverse, M^{-1}, and spawns a branch per factorization allowed for SPD matrices: QR, Cholesky, and eigendecomposition (Table 2.2); here, we only describe the Cholesky factorization ($LL^T = M$), which generates Node 3: The equation becomes $b := (X^T (LL^T)^{-1} X)^{-1} X^T (LL^T)^{-1} y$, and the inference engine asserts a number of properties for L: square, lower triangular, and full rank. The innermost inverse now is $(LL^T)^{-1}$; since L is square, rewrite rules allow the distribution of the inverse over the product LL^T, resulting in Node 4.

Once more, the compiler looks at the innermost inverse operators: In this case, they all are applied to triangular matrices and, according to our guidelines, they do not require further treatment. Therefore the focus shifts on the expression $(X^T L^{-T} L^{-1} X)^{-1}$; L is already in factored form, while according to a heuristic, the factorizations of X would not be useful; hence the compiler resorts to mappings onto kernels. Matching the expression against the list of available kernels yields two segments: L^{-1} and $L^{-1}X$. The latter has higher priority, so it is exposed ($W := L^{-1}X$), every occurrence is replaced with W (generating Node 5), and it is established that W is a full rank, column panel (Box 2.8).

Similar to the example depicted in Figure 2.3, the inspection of Node 5 causes two branches to be constructed: In the right one, CLAK multiplies out $S := W^T W$,

[4] This is because the eigendecomposition can be reused across the entire two-dimensional sequence, while the Cholesky factorization cannot.

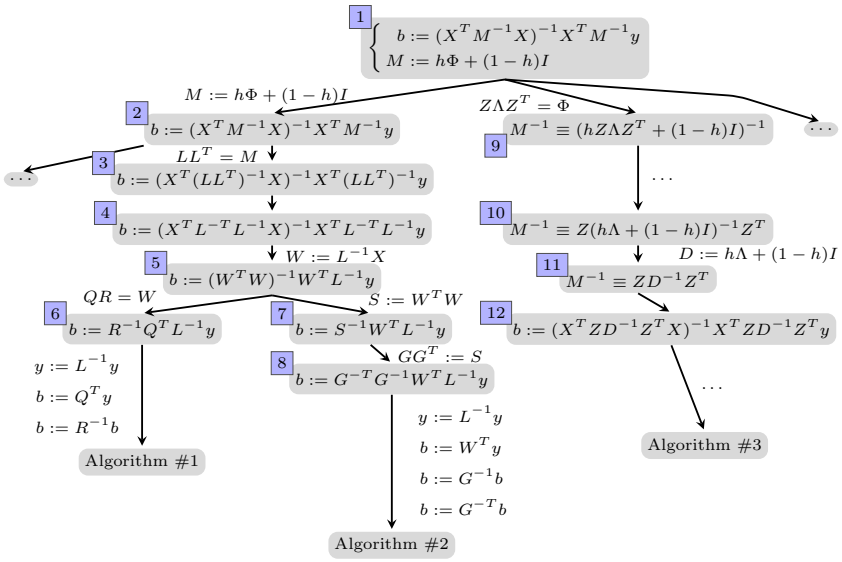


Figure 2.4: Snippet of the tree spawned by CLAK while constructing algorithms for the computation of GWAS.

producing the *SPD* matrix S (Node 7). In the left one, in accordance to the properties of W, the matrix is factored via a QR factorization; after replacing W with the product QR, the simplifications exposed in Box 2.4 are carried out, resulting in Node 6. At this point, all inverses are processed, as the remaining ones are only applied to triangular matrices. For this node, the first phase (as described in Section 2.2.1) is completed, thus the remaining expression is now to be mapped onto available kernels. The compiler identifies the following building blocks: R^{-1}, $R^{-1}Q^T$, $Q^T L^{-1}$, L^{-1}, and $L^{-1}y$. The first four are either matrix inversions or matrix-matrix operations, while the last one corresponds to a matrix-vector operation. Based on the list of priorities (Table 2.3), the matrix-vector operation $L^{-1}y$ is chosen. The same reasoning is applied subsequently, leading to the sequence of operations $y' := L^{-1}y$, $b := Q^T y'$, and $b := R^{-1}b$. A similar discussion leads from Node 7 to Algorithm #2.

Finally, we focus on the subtree rooted at Node 9. After the eigendecomposition of Φ, the innermost inverse is given by $M^{-1} \equiv (hZAZ^T + (1-h)I)^{-1}$. Analogous to the reasoning previously illustrated in Box 2.5, CLAK carries out a number of algebraic transformations that lead to the simplified expression $M^{-1} \equiv Z(h\Lambda + (1-h)I)^{-1} Z^T$ (Node 10). Here, the innermost inverse is applied to a diagonal object (Λ is diagonal and h a scalar); no more factorizations are needed, and $D := h\Lambda + (1-h)I$ is exposed (Node 11). The inverse of M is then replaced in $b := (X^T M^{-1} X)^{-1} X^T M^{-1} y$, resulting in Node 12. The subsequent steps develop similarly to the case of Node 4, generating Algorithm #3.

Once the search completes, the algorithms are built by assembling the operations that label each edge along the path from the root node to each of the leafs. The

three algorithms are provided in Algorithms 2.9 (#1: QR-GWAS), 2.10 (#2: CHOL-GWAS), and 2.11 (#3: EIG-GWAS), together with the corresponding screenshot of Mathematica's output. In brackets, we provide the names of the matching building blocks. Next, we discuss how the produced algorithms are tailored for the computation of multiple instances of problems.

Algorithm 2.9: QR-GWAS

$M := h\Phi + (1-h)I$	(SCAL-ADD)
$LL^T = M$	(POTRF)
$W := L^{-1}X$	(TRSM)
$QR = W$	(GEQRF)
$y := L^{-1}y$	(TRSV)
$b := Q^T y$	(GEMV)
$b := R^{-1}b$	(TRSV)

```
tmp10 = 1 id - h id
tmp23 = tmp10 + h Phi
L40 L40^T = tmp23
tmp58 = L40^-1 X
Q78 R79 = tmp58
tmp122 = L40^-1 y
tmp215 = Q78^T tmp122
tmp311 = R79^-1 tmp215
b = tmp311
```

Algorithm 2.10: CHOL-GWAS

$M := h\Phi + (1-h)I$	(SCAL-ADD)
$LL^T = M$	(POTRF)
$W := L^{-1}X$	(TRSM)
$S := W^T W$	(SYRK)
$GG^T = S$	(POTRF)
$y := L^{-1}y$	(TRSV)
$b := W^T y$	(GEMV)
$b := G^{-1}b$	(TRSV)
$b := G^{-T}b$	(TRSV)

```
tmp10 = 1 id - h id
tmp23 = tmp10 + h Phi
L40 L40^T = tmp23
tmp58 = L40^-1 X
tmp80 = tmp58^T tmp58
L125 L125^T = tmp80
tmp216 = L40^-1 y
tmp312 = tmp58^T tmp216
tmp355 = L125^-1 tmp312
tmp400 = L125^-T tmp355
b = tmp400
```

Algorithm 2.11: EIG-GWAS

$Z\Lambda Z^T = \Phi$	(SYEVR)
$D := h\Lambda + (1-h)I$	(SCAL-ADD)
$K := X^T Z$	(GEMM)
$V := KD^{-1}$	(SCAL)
$A := VK^T$	(GEMM)
$QR = A$	(GEQRF)
$y := Z^T y$	(GEMV)
$b := Vy$	(GEMV)
$b := Q^T b$	(GEMV)
$b := R^{-1}b$	(TRSV)

```
Z7 W8 Z7^T = Phi
tmp28 = 1 id - h id
tmp50 = tmp28 + h W8
tmp70 = X^T Z7
tmp103 = tmp70 tmp50^-1
tmp182 = tmp103 tmp70^T
Q274 R275 = tmp182
tmp341 = Z7^T y
tmp384 = tmp103 tmp341
tmp457 = Q274^T tmp384
tmp573 = R275^-1 tmp457
b = tmp573
```

2.5 Multiple instances of problems: GWAS (Part II)

It is not uncommon that scientific and engineering applications require the solution of not a single instance of a problem, but an n-dimensional grid of them. Typically, libraries and languages for scientific computing follow a black-box approach, i.e., they provide a routine to solve one instance, which is then used repeatedly for the entire grid. While this approach is effective for problems that are completely independent from one another, its rigidity leads to a suboptimal strategy when the problems are related, and intermediate results may be reused. To overcome this limitation, CLAK breaks the black-box approach by 1) exposing the computation within the single-instance algorithm, 2) performing an analysis of data dependencies, and 3) rearranging the operations so that redundant computations are avoided.

The generation of algorithms for grids of problems is divided in two steps: First, the compiler creates a family of algorithms for a single instance, following the techniques described in Sections 2.2 and 2.3; then, each of the algorithms is customized for the solution of the entire grid. We proceed by describing the latter step, using as a case study the GWAS example; specifically, we focus on Algorithm 2.11.

The single-instance algorithm is wrapped with as many loops as different dimensions; in this case, with a double loop along the m and t dimensions. Both loop transpositions —`for i, for j` and `for j, for i`— are generated and analyzed; we concentrate on the latter. Next, CLAK identifies operations that are loop-invariant, i.e., operations that do not change across iterations of the loop; and applies the so-called *code motion* optimization, which consists in moving loop-invariant operations to the preheader of the loop, i.e., the region right before of the loop.

In details, invariant operations are identified by means of an analysis of the dependencies between operands and loop indices: The compiler labels each input operand according to the input equation description (Algorithm 2.12). The subscripts are then propagated with a single pass, from top to bottom, through the algorithm: For each operation, the union of the indices appearing in the right-hand side is attached to the operand(s) on the left-hand side, and to all their occurrences thereafter (Algorithm 2.13).

Algorithm 2.12: EIG-GWAS. Initial dependencies.

```
1  for j in 1:t
2    for i in 1:m
3      ZΛZ^T = Φ
4      D := h_j Λ + (1 − h_j) I
5      K := X_i^T Z
6      V := K D^{-1}
7      A := V K^T
8      QR = A
9      y := Z^T y_j
10     b := V y_j
11     b := Q^T b
12     b := R^{-1} b
```

Algorithm 2.13: EIG-GWAS. Propagated dependencies.

```
1  for j in 1:t
2    for i in 1:m
3      ZΛZ^T = Φ
4      D_j := h_j Λ + (1 − h_j) I
5      K_i := X_i^T Z
6      V_ij := K_i D_j^{-1}
7      A_ij := V_ij K_i^T
8      Q_ij R_ij = A_ij
9      y_j := Z^T y_j
10     b_ij := V_ij y_j
11     b_ij := Q_ij^T b_ij
12     b_ij := R_ij^{-1} b_ij
```

At this point, the subindices in the left-hand side operands indicate each operation's dependencies. Operations not labeled with the i subindex may now be moved to the preheader of the innermost loop; these operations are computed once per iteration over j, and reused across the loop over i. Additionally, the operation in line 3 is invariant with respect to both loops; thus it is moved outside the loops. The rearranged algorithm is provided in Algorithm 2.14.

We note that Algorithm 2.14 still performs some redundant computation: line 6 depends on the iterator i but not on j, thus being computed redundantly for each iteration over j. Similarly to the application of code motion, CLAK identifies this situation and drags the operation outside the loops, precomputes each of the i products, and reuses the results in the loops.[5] The final algorithm for the two-dimensional grid of problems is provided in Algorithm 2.15. In the following section, we discuss the impact of the analysis of dependencies.

[5] This optimization reduces the computational cost but increases the temporary storage requirements. Currently, CLAK focuses on flop-efficient algorithms; in future versions of the compiler, the application of this and similar optimizations should be configurable.

Algorithm 2.14: EIG-GWAS. After applying code motion.

```
1  Z Λ Z^T = Φ
2  for j in 1:t
3      D_j := h_j Λ + (1 − h_j) I
4      y_j := Z^T y_j
5      for i in 1:m
6          K_i := X_i^T Z
7          V_ij := K_i D_j^{-1}
8          A_ij := V_ij K_i^T
9          Q_ij R_ij = A_ij
10         b_ij := V_ij y_j
11         b_ij := Q_ij^T b_ij
12         b_ij := R_ij^{-1} b_ij
```

Algorithm 2.15: EIG-GWAS. Final algorithm.

```
1  Z Λ Z^T = Φ
2  for i in 1:m
3      K_i := X_i^T Z
4  for j in 1:t
5      D_j := h_j Λ + (1 − h_j) I
6      y_j := Z^T y_j
7      for i in 1:m
8          V_ij := K_i D_j^{-1}
9          A_ij := V_ij K_i^T
10         Q_ij R_ij = A_ij
11         b_ij := V_ij y_j
12         b_ij := Q_ij^T b_ij
13         b_ij := R_ij^{-1} b_ij
```

2.6 Cost analysis

As detailed throughout this chapter, CLAK produces a family of algorithmic variants to solve one target equation. Since the amount of generated algorithms might be (fairly) large, it is convenient to provide the user with a metric to select the variant that best suits his needs. As part of the "Interface to building blocks" module, each of the building blocks onto which CLAK may map the algorithms is labeled with its asymptotic cost. By combining the individual costs, the compiler documents each of the output algorithms with their computational cost.

As an example, we provide the cost of the three selected algorithms for GWAS: QR-GWAS, CHOL-GWAS, and EIG-GWAS. Table 2.4 includes the cost of the three algorithms after the tailoring for one instance of GLS problem, as well as for the two most common grids of GLS problems in GWAS (Equation 2.2): a one-dimensional grid, where $t = 1$, and a two-dimensional grid, where $t \approx 10^5$.

Scenario	QR-GWAS	CHOL-GWAS	EIG-GWAS
One instance	$O(n^3)$	$O(n^3)$	$O(n^3)$
1D grid	$O(n^3 + mpn^2)$	$O(n^3 + mpn^2)$	$O(n^3 + mpn^2 + mp^2n)$
2D grid	$O(tn^3 + mtpn^2)$	$O(tn^3 + mtpn^2)$	$O(n^3 + mpn^2 + mtp^2n)$

Table 2.4: Computational cost for the three selected algorithms generated by CLAK.

QR-GWAS and CHOL-GWAS share the same computational cost for both types of grids, suggesting a very similar behavior in practice. When compared to EIG-GWAS, these two algorithms present a lower cost for the one-dimensional case; in contrast, for the two-dimensional case, the cost of EIG-GWAS is considerably lower. This analysis suggests that QR-GWAS and CHOL-GWAS are better suited for the one-dimensional grid, while EIG-GWAS is better suited for the two-dimensional one. Experimental results in [20, 25] confirm these predictions.

These numbers illustrate two of the problems that motivate this research, and emphasize the benefits of our approach. First, a generic solver for a GLS, e.g., the Matlab routine `lscov` (LAPACK does not provide an expert routine for this operation), has a computational cost of $O(n^3)$, as the algorithms generated by CLAK. However, Matlab provides no means to exploit domain-knowledge such as the structure of the matrix M, or the linkage among problems; thus, unless the user develops his own algorithms, the only approach supported by Matlab is to use `lscov` for each individual problem in the grid, for a total cost of $O(mtn^3)$ versus a cost of $O(n^3 + mpn^2 + mtp^2n)$ for CLAK's best algorithm. Second, the availability of multiple variants allow the user to choose the one that fits best his needs. If the solution of a one-dimensional grid ($t = 1$) is sought after, CHOL-GWAS or QR-GWAS should be used; if the target is the two-dimensional grid, EIG-GWAS is to be preferred.

Besides their theoretical interest, these three algorithms are also of practical relevance. State-of-the-art tools offer routines only for the 1D case; the alternative for the 2D case is to repeatedly use the algorithms for the 1D scenario t times in a black-box fashion. In both scenarios, our algorithms improve the state-of-the-art ones: For the 1D scenario, QR-GWAS and CHOL-GWAS perform half the computation of the best existing algorithms, and for the 2D case EIG-GWAS reduces their computational cost by $O(n)$ [19, 20, 25].

2.7 Code generation: Matlab and Fortran

Algorithms have been generated; we now turn the attention towards their translation into code. We incorporated into CLAK two prototypes of code generators, which produce Matlab and Fortran routines.

As an example, we provide the routines generated for the EIG-GWAS algorithm (Algorithm 2.15): Routine 2.16 and Routine 2.17 contain, respectively, the Matlab and Fortran implementations. The name of the routines, `GWAS_26_2`, stands for the operation (GWAS), the algorithm number (26 out of the 99 generated), and the permutation number (second generated permutation of the iterators i and j, as described in Section 2.5). As a direct consequence of our approach, each of the operations in Algorithm 2.15 has a unique mapping onto available building blocks. In the case of Matlab, its high level notation for matrix operations such as product, addition, inverse and transposition —which are internally mapped onto the corresponding kernels provided by high-performance libraries—, relieves the

user (or the code generator) from tedious and error-prone low-level details and enables a one to one translation of the algorithm statements into code.

In contrast, the generation of Fortran code is much more complicated. The mapping onto library routines is now explicit, and the user is exposed to the libraries' internals. In the case of BLAS and LAPACK, special attention must be paid to both the overwriting of input operands and the implicit storage of the operands with special structure. The former enforces further data dependency analysis and expression rewriting, while the latter requires carefully tracking the storage representation of the operands of each kernel. We illustrate these issues via the QR factorization in line 10 of Algorithm 2.15, and the subsequent use of the Q and R operands in lines 12 and 13. The factorization is performed by the call to LAPACK's `dgeqrf` routine in line 52 of the Fortran code. The routine factors the operand `temp9`; however, instead of creating two new operands for the corresponding Q and R matrices, the contents of the computed R are stored in the elements on and above the diagonal of `temp9`, and Q is implicitly stored in the combination of the elements below the diagonal and the extra buffer `tau10`. As a consequence, the compiler has to ensure 1) that if `temp9` is needed in subsequent calls, a copy is kept, and 2) that subsequent uses of Q and R refer now to parts of `temp9`. Further, the compiler must detect that Q is not stored explicitly as an orthogonal matrix but implicitly as the product of a number of Householder reflectors [29], and consequently map the statement representing the matrix product $Q_{ij}^T b_{ij}$ to the special routine `dormqr`, instead of the more general `dgemv`.

This operation clearly exemplifies the challenge behind a Fortran (and similarly a C) code generator; and also how CLAK alleviates the developer's burden not only in generating specialized algorithms but also in the tedious and error prone translation into code.

Routine 2.16: Matlab code for EIG-GWAS (Algorithm 2.15) generated by CLAK.

```
function [b] = GWAS_26_2(X, y, h, Phi, sm, sn, nXs, nys)
   b = zeros(sm, nXs * nys);
   T3 = zeros(sm, sn * nXs);
   [Z1, W1] = eig( Phi );

   for i = 1:nXs
      T3(:, sn*(i-1)+1:sn*i) = X(:, sm*(i-1)+1:sm*i)' * Z1;
   end
   for j = 1:nys
      T1 = 1 * eye(sn) + - h(j) * eye(sn);
      T2 =  T1 + h(j) * W1;
      T6 =  Z1' * y(:, j);
      for i = 1:nXs
         T4 =  T3(:, sn*(i-1)+1 : sn*i) / T2;
         T5 =  T4 * T3(:, sn*(i-1)+1 : sn*i)';
         [Q1, R1] = qr( T5, 0 );
         T7 =  T4 * T6;
         T8 =  Q1' * T7;
         T9 = R1 \ T8;
         b(:, i + (j-1)*nXs) = T9;
      end
   end
end
```

Routine 2.17: Fortran code for EIG-GWAS (Algorithm 2.15) generated by CLAK (I).

```
SUBROUTINE GWAS_26_2( X, csX, dsX, y, csy, h, Phi, csPhi,
                      b, csb, sn, sm, nXs, nys )
   INTEGER sn, sm, nXs, nys, csX, dsX, csy, csPhi, csb
   DOUBLE PRECISION X(csX, dsX, nXs), y(csy, nys), h(nys),
                    Phi(csPhi, sn), b(csb, nXs, nys)

   DOUBLE PRECISION ZERO
   PARAMETER (ZERO=0.0D+0)
   DOUBLE PRECISION ONE
   PARAMETER (ONE=1.0D+0)

   DOUBLE PRECISION tmp50(sn), tmp182(sm, sm), temp13(sm),
                    temp8(sm, sn), tmp28(sn),
                    tmp70(sm, sn, nXs), tmp384(sm), [...]
   [...]
   EXTERNAL dtrsv, dgemm, dgeqrf, dgemv, dsyevr, dscal,
            dormqr, dcopy

   call dsyevr( 'V', 'A', 'L', sn, Phi( 1, 1 ), sn, ddummy,
                ddummy, idummy, idummy, ddummy, nCompPairs5,
                W8( 1 ), Z7( 1, 1 ), sn, isuppz4, ... )
   DO i = 1, nXs
      call dgemm( 'T', 'N', sm, sn, sn, ONE, X( 1, 1, i ),
                  sn, Z7( 1, 1 ), sn, ZERO,
                  tmp70( 1, 1, i ), sm )
   END DO
   DO j = 1, nys
      DO iter1 = 1, sn
         tmp28( iter1 ) = 1 + ( - h(j))
      END DO
      DO iter1 = 1, sn
         tmp50( iter1 ) = tmp28(iter1) + h(j) * W8(iter1)
      END DO
      call dgemv( 'T', sn, sn, ONE, Z7( 1, 1 ), sn,
                  y( 1, j ), 1, ZERO, tmp341( 1 ), 1 )
```

Routine 2.17b: Fortran code for EIG-GWAS (Algorithm 2.15) generated by CLAK (II).

```
      DO i = 1, nXs
         DO iter1 = 1, sn
            call dcopy( sm, tmp70( 1, iter1, i ), 1,
                        temp8( 1, iter1 ), 1 )
         END DO
         DO iter1 = 1, sn
            call dscal( sm, 1/tmp50(iter1),
                        temp8( 1, iter1 ), 1 )
         END DO
         call dgemm( 'N', 'T', sm, sm, sn, ONE,
                     temp8( 1, 1 ), sm, tmp70( 1, 1, i ),
                     sm, ZERO, tmp182( 1, 1 ), sm )
         DO iter1 = 1, sm
           call dcopy( sm, tmp182( 1, iter1 ), 1,
                       temp9( 1, iter1 ), 1 )
         END DO
         call dgeqrf( sm, sm, temp9( 1, 1 ), sm,
                      tau10( 1 ), work12, sm*192, info )
         call dgemv( 'N', sm, sn, ONE, temp8( 1, 1 ), sm,
                     tmp341( 1 ), 1, ZERO, tmp384( 1 ), 1 )
         call dcopy( sm, tmp384( 1 ), 1, temp13( 1 ), 1 )
         call dormqr( 'L', 'T', sm, 1, sm, temp9( 1, 1 ),
                      sm, tau10( 1 ), temp13( 1 ), sm,
                      work15( 1 ), sm*192, info )
         call dcopy( sm, temp13( 1 ), 1, temp16( 1 ), 1 )
         call dtrsv( 'U', 'N', 'N', sm, temp9( 1, 1 ), sm,
                     temp16( 1 ), 1 )
         DO iter1 = 1, sm
            b( iter1, i, j ) = temp16( iter1 )
         END DO
      END DO
   END DO
   RETURN
END
```

2.8 Scope and limitations

As input, CLAK accepts target operations of the form

```
op = <Expression>,
```

where `op` is a single output operand, and `<Expression>` is a combination of input operands and the operators $+$, $*$, $-$, T, -1. In addition, as we illustrate in the next chapter, operations easily translated into such a form, e.g., linear systems like $AX = B$, where X is the unknown, are also allowed (with a slight modification of the grammar presented in Section 2.1). For a given operation in this class, CLAK returns a family of algorithmic variants, documented with their computational cost, and translated into the corresponding Matlab and Fortran routines.

While powerful, CLAK presents a number of limitations that should be addressed in the future:

Accepted knowledge and equations. We successfully showcased how to handle and exploit different pieces of domain knowledge such as operands' properties and the relation among instances of problems. Yet, the power of the compiler would be increased with the support for an extended set of properties, e.g., banded matrices. Also, while broad, the range of supported equations is still limited. We plan an extension to deal with more complex operations, ranging from explicit equations (as opposed to only assignments) to determinants, logarithms, and matrix functions in general.

Choosing an algorithm. Each of the generated algorithms is currently documented with a rather simple performance analysis based on the operation count (flops). This is, in general, not a reliable metric, and we aim at incorporating more advanced techniques for performance prediction which account for the underlying architecture and libraries. A promising research project relies on a sample-based approach: The idea is to create performance models not for the competing algorithms, but only for those routines that are used as building blocks. By combining the models, it is then possible to make accurate performance predictions [49].

Code generation. CLAK's code generator currently produces sequential code, which may take advantage of multi-threaded implementations of the BLAS library to exploit shared-memory parallelism. Nevertheless, the variety of available computing platforms (e.g, multi- and many-core processors, clusters, and co-processors such as GPGPUs) demands the generation of algorithms that are tailored not only to the application but also to the architecture. To this end, we envision the development of a number of modules responsible for the tailoring to each specific architecture and type of parallelism.

Stability analysis. The main goal of CLAK (beyond increasing development productivity) is the generation of efficient algorithms. However, while mathematically correct, the produced algorithms may be numerically unstable when executed in finite precision arithmetic. For instance, for the ordinary least-squares (OLS) problem $x := (A^TA)^{-1}Ab$, CLAK generates, among other stable ones, Algorithm 2.18; depending on the condition number of the matrix A, the algorithm may yield highly inaccurate results [29].

Algorithm 2.18: Potentially unstable algorithm to solve the OLS problem.

$S := A^TA$	(SYRK)
$LL^T = S$	(POTRF)
$x := A^Tb$	(GEMV)
$b := L^{-T}x$	(TRSV)
$b := L^{-1}x$	(TRSV)

Unfortunately, a completely automatic analysis is extremely challenging, and, to the best of our knowledge, no methodology exists that addresses this issue. Thus, currently, the generated algorithms need to be manually validated or tested for numerical robustness.

2.9 Summary

We introduced CLAK, a linear algebra compiler for the generation of application-tailored algorithms and routines. CLAK takes as input the mathematical description of a target operation together with domain-specific knowledge; in a process that closely replicates the reasoning of a human expert, the target equation is mapped onto a sequence of calls to high-performance library-supported kernels. Along the process, the compiler applies a number of optimizations and exploits the available knowledge to produce specialized algorithms. The contents of this chapter extend our work published in [23, 24].

This chapter makes the following contributions:

- A model of expert reasoning. We observed the reasoning carried out by linear algebra experts in the derivation of algorithms, and set up a heuristic-based model of this reasoning, making it systematic. A number of optimizations, from simple to advanced ones but all overlooked by sophisticated environments like Matlab, are also discussed and incorporated into our compiler.

- Advanced management of domain knowledge. We exposed the importance of exploiting domain-specific knowledge, and introduced an engine that enables the inference of operands' properties to dynamically expand the available knowledge. This allows for a more precise tailoring of the algorithms

and routines. Such an inference engine is not provided [6] by any (symbolic) linear algebra package.

- High-performance algorithms for GWAS. Part of the examples in this chapter are taken from an operation arising in the context of genome-wide association studies, a popular tool in computational biology. The generated algorithms led to high-performance out-of-core routines that largely outperform state-of-the-art libraries [20, 25]. These routines have been incorporated into a widely-used R package for statistical genomics, GenABEL [4], as part of the OmicABEL library [19]. [7]

[6] At the time of this dissertation writing.

[7] Available at `http://www.genabel.org/packages/OmicABEL`

Chapter 3

CLAK: High-Performance BLAS and LAPACK Derivatives

In Chapter 2, we detailed the mechanisms behind CLAK, and demonstrated its potential by applying it to the challenging GWAS problem. The purpose of this chapter is to provide evidence of the broad applicability and extensibility of our approach. Moreover, we show how our automated system comes in handy where a manual approach is not viable. We concentrate on the field of Algorithmic Differentiation (AD) —often referred to as Automatic Differentiation—, and illustrate the application of CLAK to generate efficient algorithms and code for computing the derivative of BLAS and LAPACK operations.

Derivatives are needed in a wide range of fields: from cost-optimization in finance, through sensitivity analysis of simulation models, to parameter estimation and design optimization. A popular technique for the computation of derivatives is that of AD [31, 48]. AD tools take a function given as a computer program, and change the semantics of the program to compute both the function and its derivative with respect to a set of selected input parameters. The conceptual idea behind AD is to first decompose the input program into elementary operations (addition, multiplication, division, ...) and functions (sin, cos, log, exp, ...), and then differentiate them. The derivatives are accumulated according to the chain rule, resulting in the computation of the derivative of the overall program.

Depending on how the chain rule is applied, AD distinguishes between two basic modes: forward mode and reverse mode. We focus on the forward mode, which follows the control flow of the program, accumulating the derivative of intermediate variables with respect to the input variables. For each of these modes, there exist two approaches to semantic transformation: Source transformation and operator overloading (see [13] for a short review). In this chapter, we consider the source transformation approach, which consists in rewriting the input program f to produce an *extended program* that computes both f and its derivative f'. For instance, the statement $a = b * c$ is rewritten into two statements: $a = b * c$ and $a' = b' * c + b * c'$. This scheme is semantic-oblivious, e.g., whenever a routine

call is encountered, no knowledge of the operation is used; the scheme is blindly applied recursively to the routine's code. Although this is an effective and scalable approach, when the input program relies on highly-optimized BLAS and LAPACK kernels, the extended program suffers from a significant loss in performance.

Due to the widespread usage of BLAS and LAPACK in scientific software, the availability of an optimized differentiated version of these libraries is relevant to the AD community. In this chapter, we extend CLAK to enable the generation of algorithms and routines for the derivative of BLAS and LAPACK operations.

Preliminaries. In the remainder of this chapter, we consider exclusively AD based on source transformation and the forward mode. We also restrict the discussion to AD tools for imperative programming languages with statements that return scalars;[1] this is an important class of languages in scientific computing, which includes C and Fortran 77. For any comparison with CLAK, we will use ADIFOR [15] as representative of such tools.

Below, we briefly introduce the key concepts for the discussion in this chapter:

1. Dependent, independent, and active variables.
In AD, an input variable is called *independent* if derivatives with respect to that variable are desired. An output variable is called *dependent* if its derivatives with respect to the independent variables are to be computed. *Active* variables depend on one or more of the independent variables, and also contribute to the computation of one or more dependent variables.

Let us consider, as an example, the following pseudocode

Pseudocode 1 : Dependent, independent, and active variables.

Input: v, y
Output: w, z

1: $\alpha := f(v)$
2: $x := g(v)$
3: $w := h(x)$
4: $z := \alpha x + y$

and let us assume that the derivative of z with respect to v is desired. In this case, z and v are the dependent and independent variables, respectively. Also, since both α and x depend on v (lines 1 and 2) and contribute to the computation of z (line 4), they are active. Since y does not depend on v and w does not contribute to the computation of z, they are inactive.

2. Activity pattern.
The *activity pattern* indicates which operands in a statement are active and which ones are not, making possible computation savings [35]. Based on an activity

[1] As opposed to vector-valued statements in languages like Matlab.

analysis, statements that do not contribute to the dependent variable are not differentiated, e.g., line 3 in Pseudocode 1 is not differentiated. The only inactive variables left are those that do not depend on the independent variable, e.g., y, whose derivative is thus 0. Accordingly, in the derivative of the statement in line 4

$$\frac{dz}{dv} := \frac{d\alpha}{dv} * x + \alpha * \frac{dx}{dv} + \frac{dy}{dv},$$

the term $\frac{dy}{dv}$ equals 0, and the expression simplifies to

$$\frac{dz}{dv} := \frac{d\alpha}{dv} * x + \alpha * \frac{dx}{dv}.$$

Assuming z, x, and y are vectors of size n, and α is a scalar, exploiting the activity pattern saves n out of $4n$ flops. The resulting extended program is given in Pseudocode 2; functions g_f and g_g calculate the derivative of f and g, respectively.

Pseudocode 2 : Differentiated Pseudocode 1 for the computation of $\frac{dz}{dv}v'$.

Input: v, v', y
Output: w, z, z'

1: $\alpha := f(v)$
2: $\alpha' := g\_f(v, v')$
3: $x := g(v)$
4: $x' := g\_g(v, v')$
5: $w := h(x)$
6: $z := \alpha x + y$
7: $z' := \alpha' x + \alpha x'$

3. Multiple derivatives.
The independent variable v is not necessarily a scalar, it may also be a vector-valued variable. When $v \in R^p$, derivatives with respect to each v_i are desired, and each derivative formula is to be computed p times:

$$\frac{dz}{dv_i} := \frac{d\alpha}{dv_i} * x + \alpha * \frac{dx}{dv_i}, \qquad 1 \leq i \leq p.$$

Conceptually, the derivative operands gain an extra dimension: scalars become vectors, vectors become matrices, and matrices become three-dimensional arrays.

Disadvantages of AD's approach. AD tools based on source transformation present two significant disadvantages when differentiating BLAS (and LAPACK) operations. The main drawback lies in the low performance of the generated code. The difficulty resides in the automatic generation of high-performance code for BLAS-like operations. Manually optimized BLAS implementations, such as the Intel's Math Kernel Library (MKL) [47], attain from 10% (BLAS 1) to more than

90% (BLAS 3) of the architectures' peak performance. Even the most prominent projects focused on automatically producing optimized BLAS libraries, such as ATLAS [67, 68], require hand-tuned microkernels to achieve a relatively high percent of such performance. AD-generated code, instead, achieves only about 5% of the peak (see performance results in Section 3.4); for BLAS 3 operations, this means code that is almost 20 times slower.

The second disadvantage is related to the treatment of different activity patterns. It may occur that a same routine, e.g., vec_mul in Pseudocode 3, is encountered multiple times in the original program, each time with a different activity pattern.

Pseudocode 3 : vec_mul. Element-wise vector product.

```
1: function v := vec_mul(x, y, z)
2: for i := 1 to length(x) do
3:         v(i) := x(i) * y(i) * z(i)
4: end for
```

Suppose that in every occurrence x is active and y is inactive; then, there are two possible cases: z is active and z is inactive. It is not uncommon that in such a situation source transformation tools generate one single differentiated version of the routine that covers for both scenarios. In the example above, such a tool generates g_vec_mul (Pseudocode 4); when the routine is called in the extended program with z inactive, a zero vector is passed for z's derivative g_z. Notice the unnecessary computation in line 4.

Pseudocode 4 : Derivative of vec_mul. Argument y is inactive.

```
1: function (v, g_v) := g_vec_mul(x, g_x, y, z, g_z)
2: for i := 1 to length(x) do
3:         v(i) := x(i) * y(i) * z(i)
4:         g_v(i) := g_x(i) * y(i) * z(i) + x(i) * y(i) * g_z(i)
5: end for
```

While the approach may beneficial in that it simplifies the code generation process and reduces the, possibly very large, size of the extended program, it may also result in considerable redundant computation.

3.1 The challenge

The need for an efficient differentiated version of BLAS was brought up by the AD community already in 2000 at the *Automatic Differentiation Conference* (AD 2000). Given the simplicity, from a mathematical perspective, of BLAS operations, analytic formulas for their derivatives can be derived easily. For instance, the formula for the derivative of AXPY ($y := \alpha x + y$, $\alpha \in R$, $x, y \in R^n$) with respect to

an independent variable v is

$$\frac{dy}{dv} := \frac{d\alpha}{dv}x + \alpha\frac{dx}{dv} + \frac{dy}{dv}. \tag{3.1}$$

M. H. Bucker and P. Hovland approached the aforementioned efficiency problem by deriving and coding such derivatives manually. Although successful in providing efficient routines for a small subset of BLAS 1 and 2 operations, the authors realized that in practice, due to the vast number of operations and variants to support, the problem becomes unmanageable. On the one hand, BLAS already contains a fairly large number of operations (about 40); each of them accepting several options to indicate multiple flavors of the operation, and supporting multiple datatypes. For instance, the routine for a general matrix-matrix product (GEMM: $C := \alpha A^{(T)}B^{(T)} + \beta C$), accepts two options to indicate whether A and B are to be transposed or not (for a total of four combinations), and provides support for four datatypes (single and double precision, real and complex data). On the other hand, the activity pattern adds another multiplicative factor to the number of flavors: Each of the n input operands may be active or inactive, resulting in $2^n - 1$ different formulas. For instance, Equation (3.1) yields seven different equations:

1. $\frac{dy}{dv} := \frac{d\alpha}{dv}x + \alpha\frac{dx}{dv} + \frac{dy}{dv}$
2. $\frac{dy}{dv} := \frac{d\alpha}{dv}x + \alpha\frac{dx}{dv}$
3. $\frac{dy}{dv} := \frac{d\alpha}{dv}x + \frac{dy}{dv}$
4. $\frac{dy}{dv} := \frac{d\alpha}{dv}x$
5. $\frac{dy}{dv} := \alpha\frac{dx}{dv} + \frac{dy}{dv}$
6. $\frac{dy}{dv} := \alpha\frac{dx}{dv}$
7. $\frac{dy}{dv} := \frac{dy}{dv}$

To give a taste of the magnitude of the challenge, in Table 3.1 we provide the number of variants required for a subset of BLAS 3 operations. Let m be the number of possible options (which typically take one of two values) and n the number of input operands (which may be active or inactive), a differentiated BLAS must support, for each of the operations, $2^m \times (2^n - 1)$ variants; further, the support of four different datatypes multiplies the above quantity by four. For the operations in the table, this totals 2624 variants. The differentiation of the entire library would require more than 8000 variants.

Clearly, the manual development and maintenance of such a library is unfeasible. The most common approach these days is to either sacrifice efficiency or to manually code only the specific variants needed in each case and plug them in the extended program. Still, the need remains: The issue was brought up again by P. Hovland in the *Seventh European Workshop on Algorithmic Differentiation* in 2008, where he insisted on the benefits that efficient differentiated versions of BLAS and LAPACK would bring to the community.

Operation	# Options	# Operands	# Variants
GEMM $(C := \alpha AB + \beta C)$	2	5	$2^2 \times (2^5 - 1) \times 4 = 496$
SYMM $(C := \alpha AB + \beta C)$	2	5	496
SYRK $(C := \alpha AA^T + \beta C)$	2	4	240
SYR2K $(C := \alpha AB^T + \alpha BA^T + \beta C)$	2	5	496
TRMM $(C := \alpha AB)$	4	3	448
TRSM $(C := \alpha A^{-1}B)$	4	3	448
		Total:	2624

Table 3.1: Number of variants required to fully support a differentiated version of a subset of BLAS 3 operations.

3.2 CLAK for high-performance derivatives

We propose a third alternative that has the potential to automatically generate derivative code for BLAS and part of LAPACK while attaining high performance. Similarly to Hovland and Bücker, the idea is to raise the abstraction level from scalar operations to matrix equations (the analytic formulas) and exploit CLAK's capabilities to find efficient mappings onto BLAS and LAPACK kernels.

To enable CLAK to find algorithms for derivative operations, the engine presented in the previous chapter is augmented with a number of features. We discuss the inclusion of support for the derivative operator and the extension of the *Inference of properties* module.

3.2.1 The derivative operator

The support for the derivative operator implied two modifications to CLAK's engine. First, we modified the input language so to accept a new unary operator: `dv(·)`. The new grammar allows, on the one hand, the declaration of derivative operands, e.g.,

```
Vector dv(x)<Input>,
```

and, on the other hand, the specification of derivative equations by means of the following extra production rule:

<Factor> → `"dv("` <Expression> `")"`.

An example of valid expression is `dv(A * B + C)`.

Second, we incorporated into CLAK the rewrite rules necessary to encode the so-called chain rule, so that the system can differentiate a given expression. These rules are displayed in Box 3.1. A collection of results for the derivative of the matrix operations in Box 3.1 and more complex ones can be found in [28].

After these modifications have been incorporated into the engine, the derivative of AXPY (Box 3.2) may be expressed as illustrated by Box 3.3. We recall that the macro `init` is used when an operand is overwritten to refer to its initial contents.

1. $\texttt{dv}(A+B) \longrightarrow \texttt{dv}(A)+\texttt{dv}(B)$
2. $\texttt{dv}(A-B) \longrightarrow \texttt{dv}(A)-\texttt{dv}(B)$
3. $\texttt{dv}(A\times B) \longrightarrow \texttt{dv}(A)\times B+A\times\texttt{dv}(B)$
4. $\texttt{dv}(A^T) \longrightarrow \texttt{dv}(A)^T$
5. $\texttt{dv}(A^{-1}) \longrightarrow -A^{-1}\times\texttt{dv}(A)\times A^{-1}$

Box 3.1: Rewrite rules encoding the chain rule.

```
Equation Axpy
    Scalar alpha<Input>;
    Vector x<Input>;
    Vector y<InOut>;

    y = alpha * x + init(y);
```

Box 3.2: Description of AXPY in CLAK.

```
Equation gAxpy
    Scalar    alpha <Input>;
    Scalar dv(alpha)<Input>;
    Vector    x <Input>;
    Vector dv(x)<Input>;
    Vector dv(y)<InOut>;

    dv(y) = dv(alpha) * x + alpha * dv(x) + init(dv(y));
```

Box 3.3: Description of the derivative of AXPY in CLAK. All of α, x, and y are active.

3.2.2 The AD mode: Inference of properties and activity patterns

The derivative of AXPY in Box 3.3 corresponds to the case where all input operands are active. However, manually writing such descriptions for all possible patterns is time consuming and error-prone; for instance, GEMM presents 31 different patterns (see Table 3.1). Furthermore, it is the user who has to deduce the properties of the derivative operands. To simplify this task, we include an "AD mode" in CLAK to carry out this process automatically.

In the AD mode, CLAK takes the description of the original equation (Box 3.2) as input and, by means of the chain rule (Box 3.1), it generates the derivative expression

```
dv(y) = dv(alpha) * x + alpha * dv(x) + init(dv(y)).
```

Then, it deduces a number of properties for the operands, and generates the input corresponding to each activity pattern.

Inference. We extended the Inference of properties module, described in Section 2.3.3, to infer properties for the operands of derivative formulas. First, the module deduces the type of operand for the derivative operands. As we mentioned earlier, when differentiating with respect to a vector-valued variable ($v \in R^p$), the derivative operands gain an extra dimension. Alternatively, following CLAK's design, we regard the problem as multiple instances of the scalar-valued case, where the derivative operands vary along the p dimension. Thus, `dv(x)` and `dv(y)` are assigned the type `Vector`, `dv(alpha)` is a `Scalar`, and the three of them are attached a subindex:

```
dv(y{i}) = dv(alpha{i}) * x + alpha * dv(x{i}) + init(dv(y{i})).
```

Next, based on the description of the original function f, the module determines which operands are input and which ones are output to the derivative f'. In the forward mode, the following rules apply:

1. The (nonlinear) input operands to f are also input operands to f'.

2. The output of f, if required, becomes an input to f'.

3. The derivative of active inputs to f are also inputs to f'.

4. The derivative of the output operand is sought after, and therefore it is the output in f'.

We illustrate the application of these rules by means of AXPY: The input operands in AXPY —`alpha`, `x` and `init(y)`— and their derivative counterparts —`dv(alpha)`, `dv(x)` and `dv(init(y))`—, are input to its derivative; the final contents of `y`, if they appeared in the derivative formula, would also be an input; the final contents of `dv(y)` are sought after.

Finally, further knowledge of the properties of the derivative operands may also be inferred from the properties of the original operands. As an example, the derivative of matrix operands presenting some type of zero-pattern, maintain such pattern; for instance, the derivative of diagonal and triangular matrices are diagonal and triangular, respectively. Similarly, the derivative of constant scalars and matrices, e.g, the identity matrix, is zero (either the scalar 0 or the zero matrix). Other properties instead are inferred only partially. This is the case of the positive definiteness of an SPD operand `A`: While the derivative inherits the symmetry (if the (i,j) and (j,i) entries of `A` are equal, so are in the derivative `dv(A)`), the positive-definiteness of `dv(A)` is not guaranteed.

Activity patterns. Once properties are deduced, the input corresponding to each possible activity pattern is generated. This is accomplished by first replacing the derivative of the inactive variables with 0, and then simplifying the resulting expression. Box 3.4 reproduces the input generated for the case where x and y are active, and α is inactive.

```
Equation gAxpy
    Scalar alpha<Input>;
    Vector dv(x)<Input>;
    Vector dv(y)<InOut>;

    dv(y{i}) = alpha * dv(x{i}) + init(dv(y{i}));
```

Box 3.4: Description of the derivative of AXPY in CLAK. x and y are active, α is inactive.

For each such description, CLAK produces algorithms and code as illustrated in Chapter 2.

3.3 An example: Differentiating $AX = B$

We provide now a brief example of the process carried out by CLAK to generate algorithms for derivative operations. We use as example the derivative of the linear system $AX = B$, where the coefficient matrix $A \in R^{n \times n}$ is SPD, and X and $B \in R^{n \times m}$. The description of the equation in CLAK's language is provided in Box 3.5; the extension for the generation of derivative code is activated via the "`--AD-mode`" argument:

```
./clak  --AD-mode  SPDSolve.ck
```

```
Equation SPDSolve
    Matrix A<Input, SPD>;
    Matrix B<Input>;
    Matrix X<Output>;

    A * X = B;
```

Box 3.5: SPDSolve.ck. Description of the SPD system $AX = B$ in CLAK.

First, CLAK takes the description and, by means of the rules in Box 3.1 (the chain rule), it produces the most general derivative formula (the one where all input operands are active):

```
dv(A) * X + A * dv(X) = dv(B)
```
(3.2)

Next, properties for the operands are inferred: Since A, B, and X are matrices, so are `dv(A)`, `dv(B)`, and `dv(X)`; each derivative operand is attached the index i. Also, the inference rules for the input and output determine that A, B, X, `dv(A)`, and `dv(B)`, are input to Equation (3.2), and `dv(X)` is the output. In terms of structure, `dv(A)` inherits the symmetry of A, and neither B and X, nor `dv(B)` and `dv(X)`, present any structure.

Then, formulas for each of the possible activity patterns are produced:

```
1. dv(A{i}) * X + A * dv(X{i}) = dv(B{i})
2. dv(A{i}) * X + A * dv(X{i}) = 0
3.                A * dv(X{i}) = dv(B{i})
```

and, for each of them, CLAK input is generated (Boxes 3.6 to 3.8).

```
Equation gSPDSolve
    Matrix    A <Input, SPD>;
    Matrix dv(A)<Input, Symmetric>;
    Matrix dv(B)<Input>;
    Matrix    X <Input>;
    Matrix dv(X)<Output>;

    dv(A{i}) * X + A * dv(X{i}) = dv(B{i});
```

Box 3.6: Description of the derivative of the SPD linear system $AX = B$ in CLAK. A and B are active.

```
Equation gSPDSolve
    Matrix    A <Input, SPD>;
    Matrix dv(A)<Input, Symmetric>;
    Matrix    X <Input>;
    Matrix dv(X)<Output>;

    dv(A{i}) * X + A * dv(X{i}) = 0;
```

Box 3.7: Description of the derivative of the SPD linear system $AX = B$ in CLAK. A is active, B is inactive.

Finally, for each of the many activity patterns, algorithms are generated. We outline the process for the general derivative (Box 3.6)

$$A'X + AX' = B'.$$

```
Equation gSPDSolve
    Matrix    A <Input, SPD>;
    Matrix dv(B)<Input>;
    Matrix dv(X)<Output>;

    A * dv(X{i}) = dv(B{i});
```

Box 3.8: Description of the derivative of the SPD linear system $AX = B$ in CLAK. A is inactive, B is active.

First, a step of algebraic manipulation rewrites the equation so that the input operands lie on the right-hand side, and the output on the left-hand side:

$$X' = A^{-1}(B' - A'X). \tag{3.3}$$

Then, CLAK applies the heuristics discussed in Chapter 2. Since the inverse operator is applied to A, a full SPD matrix, the matrix is factored using multiple factorizations (Cholesky, QR, and eigendecomposition); in the case of a Cholesky factorization, $LL^T = A$, the inference engine deduces properties for L (lower triangular, square, full rank), and Equation (3.3) is rewritten as

$$X' = (LL^T)^{-1}(B' - A'X).$$

Now, since L is square, the inverse may be distributed over the product, resulting in

$$X' = L^{-T}L^{-1}(B' - A'X). \tag{3.4}$$

In Equation 3.4, the inverse is applied only to triangular operands, and no further processing of inverses is required; CLAK proceeds with the mapping of the equation onto kernels. The compiler matches multiple kernels: L^{-1}, $L^{-1}B'$, $L^{-1}A'$, and $B' - A'X$. The inversion has the lowest priority and is therefore discarded; since all operands in the remaining three kernels are matrices, the kernels have the same priority and the three paths are considered; for the sake of brevity, we only describe the latter ($S := B' - A'X$). The decomposition of the remaining expression,

$$X' = L^{-T}L^{-1}S,$$

completes with the identification of two TRSMs: $T := L^{-1}S$, and $X' := L^{-T}T$. The resulting algorithm is assembled in Algorithm 3.1.

The process completes with the tailoring of the algorithm to the computation of multiple instances $v \in R^p$ (the case of scalar-valued v is captured by $p = 1$). The analysis of dependencies determines that, since A does not vary along the derivative direction, the Cholesky factorization in line 1 may be performed once and reused; the remaining operations depend on the iterator i, thus must be kept within the loop. The final algorithm for the computation of multiple derivatives is provided in Algorithm 3.2.

Algorithm 3.1: Single-instance gSPD	
$LL^T = A$	(POTRF)
$S := B' - A'X$	(GEMM)
$T := L^{-1}S$	(TRSM)
$X' := L^{-T}T$	(TRSM)

Algorithm 3.2: Multiple-instance gSPD	
$LL^T = A$	(POTRF)
for i **in** 1:p	
$S_i := B'_i - A'_i X$	(GEMM)
$T_i := L^{-1}S_i$	(TRSM)
$X'_i := L^{-T}T_i$	(TRSM)

AD tools based on a black box approach, e.g. ADIFOR, generate code that computes every statement p times regardless of whether the computation is redundant. In contrast, not only does Algorithm 3.2 benefit from a mapping onto optimized BLAS and LAPACK kernels, it also reduces the computational cost with respect to that of ADIFOR's equivalent routine. While the cost of ADIFOR to compute the derivate is $\frac{2}{3}pn^3 + 4pn^2m$, CLAK reduces the cost to $\frac{1}{3}n^3 + 4pn^2m$.

3.4 Experimental results

We present now performance results for the derivative of two example operations: the previously discussed solution of a linear system with an SPD coefficient matrix (LAPACK's POSV routine), and the so-called symmetric rank-update (SYRK) BLAS operation. In both cases, we compare the performance of the routines generated by CLAK with those generated by the AD tool ADIFOR (version 2.0).

We recall that the code generated by ADIFOR mixes the computation of both the function and its derivative. For a fair comparison, the timings corresponding to the routines generated by CLAK also include the computation of both operation and derivative. Samples of the routines generated by CLAK are provided in Appendix B.

The experiments were performed on an SMP system consisting of two Intel Xeon E5450 multi-core processors. Each processor comprises four cores operating at 3 GHz. The system is equipped with 16 GB of RAM. The routines were compiled using the GNU C (version 4.4.5) and Fortran (version 4.4.6) compilers, and linked to the Intel MKL library (version 12.1). The compiler flags "-O2 -mcmodel=medium" were used. Computations were performed in double precision.

3.4.1 Example 1: Solution of a SPD linear system

We commence by presenting experimental results for the solution of the SPD system discussed in the previous section: $AX = B$, where $A \in R^{n\times n}$, and B and $X \in R^{n\times m}$. In the experiments, we concentrate on the activity pattern where all operands are active, i.e.,

$$A'X + AX' = B'.$$

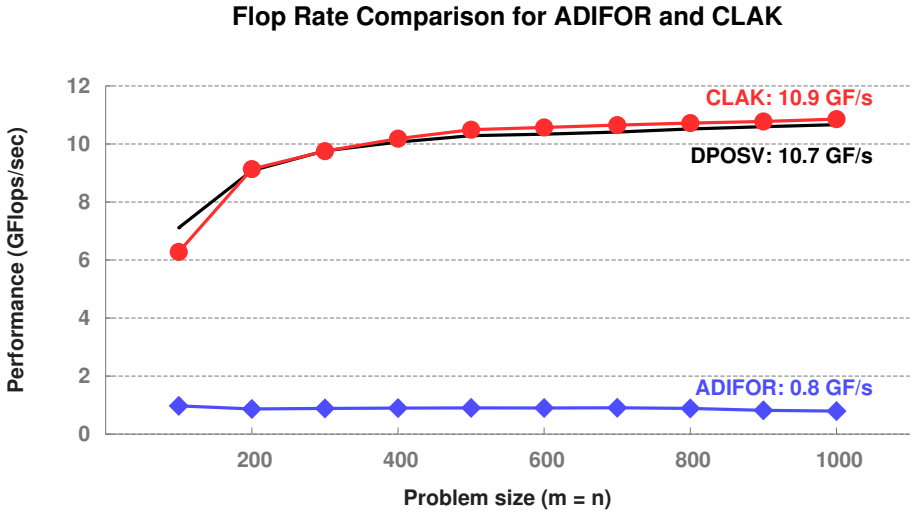


Figure 3.1: Performance comparison between the routines generated by ADIFOR and CLAK for the solution of an SPD system and its derivative. The flop rate attained by MKL for the SPD system (DPOSV) is given as a reference. Results obtained for a single derivative ($p = 1$), and a single core.

First, we give a sense of the performance differences between the code generated by ADIFOR and that generated by CLAK. In Figure 3.1, we report on the flop rate attained for a single derivative ($p = 1$) and an increasing size of the matrices ($m = n$). As a reference, the top of the figure represents the theoretical peak performance of the architecture for a single core (12 GFlops/sec), and the line labeled "DPOSV" indicates the flop rate attained by MKL's routine for the solution of the original SPD system. While ADIFOR's code delivers rather poor performance (below 1 GFlop/sec), CLAK's code attains a performance of 11 GFlops/sec, comparable to that of LAPACK, and close to the peak performance. The message is that the usage of CLAK's routine in the extended program sustains LAPACK's performance levels, and prevents a loss in performance.

Next, in Figure 3.2, we show how this gap in performance translates into large speedups. The speedup (ratio of execution time for ADIFOR over execution time for CLAK) ranges from 5x for a small coefficient matrix (100×100) and one right-hand side, up to 35x for large problems.

In Figure 3.3, we provide further experiments where multiple derivatives ($p > 1$) are computed. The goal is to emphasize the even larger speedups achieved thanks to the analysis of dependencies carried out by CLAK. As anticipated by the computational cost formulas, the largest ratio is attained when computing multiple derivatives with a single right-hand side, for a speedup of about 80x.

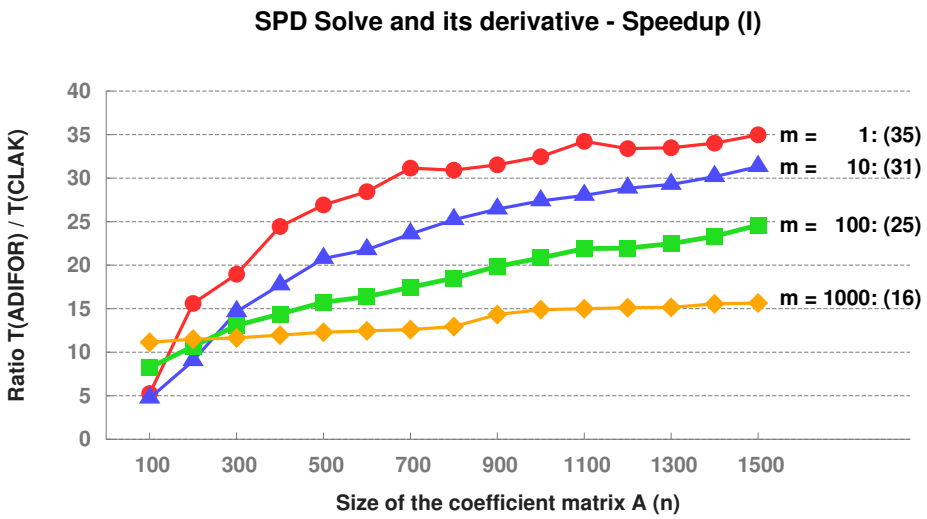


Figure 3.2: Speedup of CLAK's routine over ADIFOR's for a variety of coefficient matrices and number of right-hand sides. Results obtained for a single derivative ($p = 1$), and a single core. In brackets, the attained speedup.

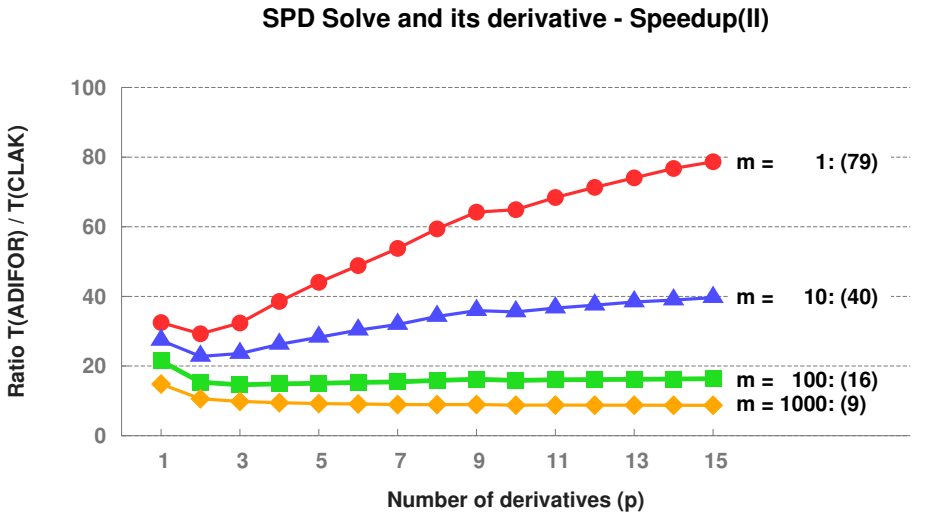


Figure 3.3: Speedup of CLAK over ADIFOR when differentiating with respect to a vector-valued variable ($p > 1$). Results obtained for $n = 1{,}000$, and a single core. In brackets, the attained speedup.

Activity Pattern	Cost for ADIFOR	Cost for CLAK
$C' := \alpha' AA^T + \alpha A'A^T + \alpha AA'^T + \beta' C + \beta C'$	$(2p+1)n^2k$	$(3p+1)n^2k$
$C' := \alpha' AA^T + \beta' C + \beta C'$	$(\ p+1)n^2k$	$(\ p+1)n^2k$
$C' := \alpha A'A^T + \alpha AA'^T + \beta C'$	$(2p+1)n^2k$	$(2p+1)n^2k$
$C' := \beta' C + \beta C'$	$(\frac{3}{2}p+k)n^2$	$(\frac{3}{2}p+k)n^2$
$C' := \beta C'$	$(\frac{1}{2}p+k)n^2$	$(\frac{1}{2}p+k)n^2$

Table 3.2: Derivative of `dsyrk`. Cost of the routines generated by CLAK and ADIFOR for a variety of activity patterns. The cost of computing `dsyrk` itself (n^2k) is included.

3.4.2 Example 2: Symmetric rank update

We now concentrate on the second example, the computation of the BLAS SYRK operation: $C := \alpha AA^T + \beta C$, where α and $\beta \in R$, $A \in R^{n\times k}$, and $C \in R^{n\times n}$ is symmetric. This operation involves four input operands (α, A, β, C), and thus admits 15 different activity patterns; a subset of them is collected in Table 3.2 together with their computational cost.

Figure 3.4 provides further evidence of the large gap between the performance attained by CLAK's and ADIFOR's code; the experiments were run for the most general derivative (row 1 in Table 3.2), and $p = 1$. Again, the top of the graph represents the peak performance of the architecture, and the line labeled with "DSYRK" shows the performance attained by MKL for the computation of DSYRK only. As it was the case for the SPD system, the performance of ADIFOR's code is about 0.7 GFlops/sec, while CLAK's routine attains a performance of almost 10 GFlops/sec, similar to that of BLAS for DSYRK (10.5 GFlops/sec), and close to the peak.

The differences in performance translate into large speedups across the spectrum of problem sizes. In Figure 3.5, we report on the speedup of CLAK over ADIFOR for the most general derivative, and $p = 1$. The speedup ranges from 4x to 10x, the larger the size of the matrix A the higher the efficiency attained by BLAS, and thus the larger the speedup.

To conclude the study, we illustrate the benefit of having routines available for all activity patterns. In Figure 3.6, we provide timings for the routines generated by CLAK for the computation of the different activity patterns in Table 3.2. Each pattern is labeled as follows: the label "aAbC" (for α, A, β, and C) means all four variables are active; whenever one of the characters is set to 0, it means that the corresponding variable is inactive. For instance, "a0bC" means α, β, and C are active, while A is inactive. As the figure shows, time to solution may be further reduced by using the most specific routine. For instance, using the routine for the case "0A0C", instead of the most general "aAbC", results in an extra 40% speedup.

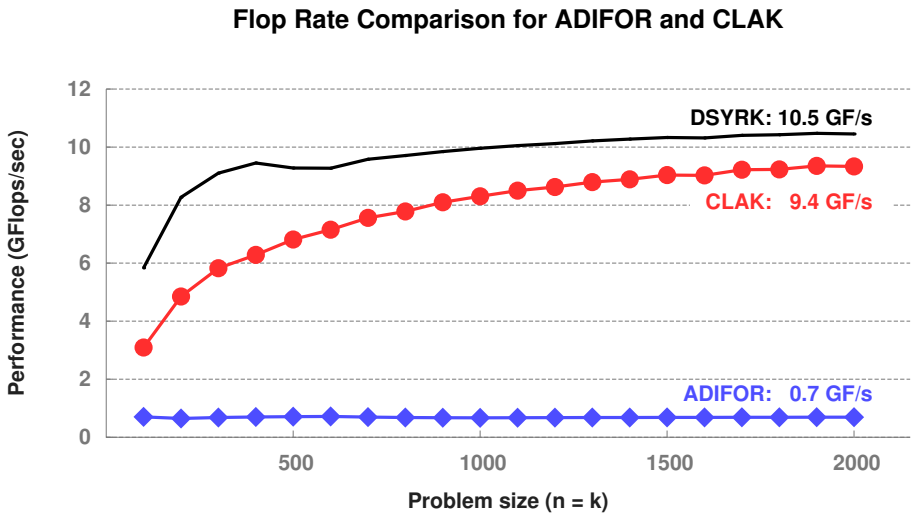


Figure 3.4: Performance comparison between the code generated by ADIFOR and CLAK for the computation of `dsyrk` and its derivative. The flop rate attained by MKL for DSYRK is given as a reference. Results obtained for a single derivative ($p = 1$), and a single core.

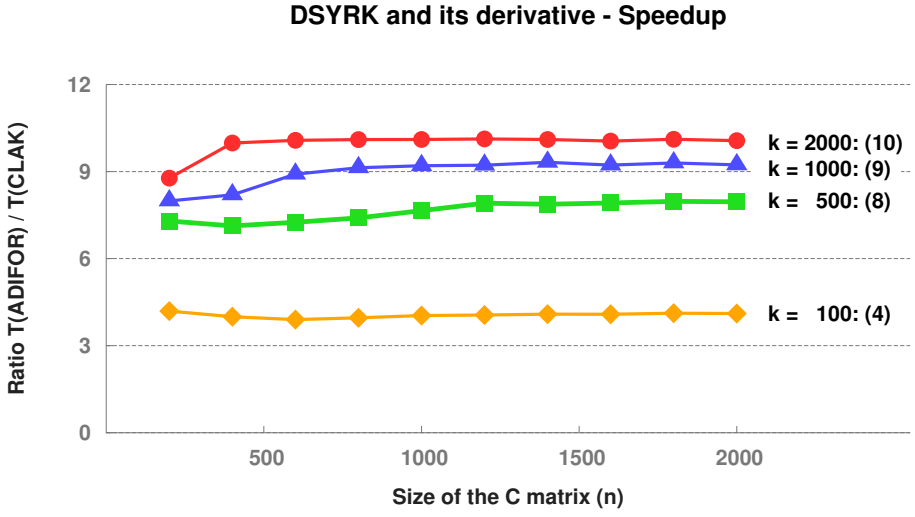


Figure 3.5: Speedup of CLAK's code with respect to ADIFOR's for a variety of matrix sizes. Results obtained for a single derivative ($p = 1$), and a single core. In brackets, the attained speedup.

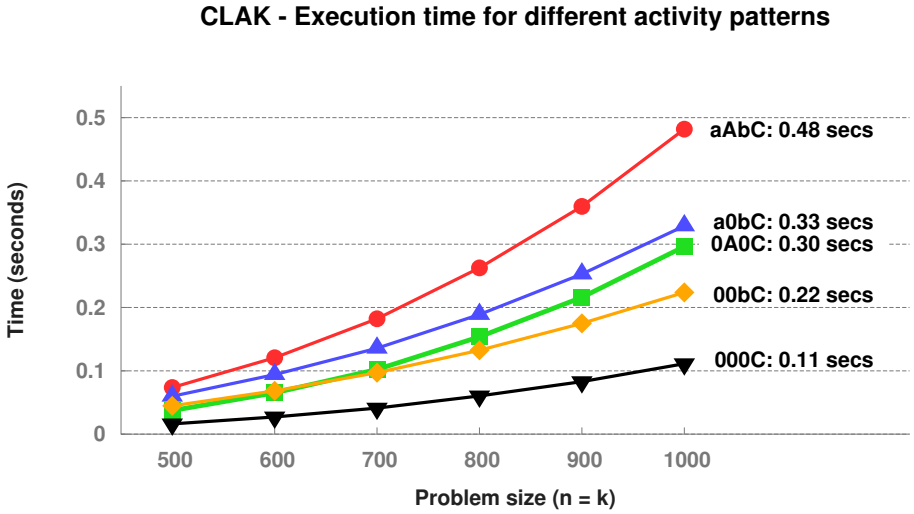


Figure 3.6: Timings for different activity patterns arising in the differentiation of the `dsyrk` BLAS operation. Timings for a single derivative ($p = 1$), and a single core.

3.5 Summary

In this chapter, we focused on illustrating the broad applicability and the extensibility of our compiler. To this end, we chose the application of CLAK to matrix operations arising in the field of algorithmic differentiation; as examples, we used the computation of the derivative of BLAS and LAPACK operations.

The core of the compiler presented in the previous chapter was extended by adding support for the derivative operator (with the encoding of the chain rule to generate the analytical derivative formulas), and by augmenting the engine for the inference of properties to determine properties of the derivative operands. The mapping of the derivative formulas onto BLAS and LAPACK kernels resulted in efficient algorithms and routines that, compared to the code generated by ADIFOR, attained considerable speedups.

Beyond further showcasing the properties of our approach and compiler, this chapter also makes a contribution to the AD community in terms of the study of the potential benefits, should high-performance differentiated versions of BLAS and LAPACK be available. First, the experimental results provide evidence that large speedups should be expected for a broad range of BLAS and LAPACK operations, as illustrated by the characteristic examples used in the experiments (a linear system and a matrix product). Second, the analysis of data dependencies at a higher level of abstraction (the analytic formulas), may lead to reductions in the computational cost with respect to traditional approaches, resulting in further speedups. And third, we show the convenience of providing specific routines for every activity pattern, especially in the context of expensive matrix computations

where the computational savings may be substantial.

While it is not uncommon that these benefits can be achieved by manually coding the kernels that represent the bulk of the computation, having a differentiated version of the libraries available, and enabling AD tools to make use of them, would have an impact in productivity. The work presented in this chapter represents a step forward towards this goal, demonstrating that efficient differentiated versions of BLAS and LAPACK are within reach.

Chapter 4

CL1CK

In this second part of the dissertation, we focus on the generation of algorithms for computational building blocks, such as matrix products and factorizations. For this class of operations, instead of the decomposition of a target operation into a sequence of building blocks, we seek the derivation of loop-based blocked algorithms. The design of such algorithms is in general a complex task, which has long been considered a fine art. Fortunately, in the last decade, in the frame of the Formal Linear Algebra Methods Environment (FLAME) project [33], a methodology has been developed for the systematic derivation of provably correct blocked algorithms [7, 8]. We adopt the FLAME methodology, and develop CL1CK, a compiler that demonstrates how the methodology can be applied fully automatically, i.e., without any human intervention.

The FLAME methodology enables the derivation of multiple algorithmic variants for one same target operation. In fact, for many operations, such as the Cholesky and LU factorizations, all the previously known algorithms are systematically discovered and unified under a common root [61]. For more involved operations, such as the triangular continuous-time Sylvester equation and the reduction of a generalized eigenproblem to standard form, the generated family of algorithms include new and better performing ones [52, 55]. A quick review of FLAME-related literature [8–10, 52, 55, 61] reveals that the methodology has been consistently tested against well-known matrix operations available from numerical linear algebra libraries such as BLAS, LAPACK, and RECSY. Indeed, the project also provides libFLAME [62], a library regarded as a modern rewrite of LAPACK, that codes hundreds of algorithms derived via this methodology.

We emphasize that the methodology is not restricted to these example problems, and is of far more general applicability. We are especially interested in specialized operations not supported by high-performance libraries, ranging from slight variations of available kernels, such as the matrix product $C = AB$ with A and B triangular, to the derivative of matrix factorizations. While in this chapter we make use of classical examples to illustrate our work, in Chapter 5 we provide an example where the methodology yields high-performance algorithms for two

operations not directly supported by any numerical library.

Even though systematic, the methodology heavily relies on pattern matching and symbolic manipulation of algebraic expressions, hence becoming a tedious and error-prone process. In fact, in [55], a mistake in the derivation led to an incorrect algorithm. As the complexity of the target equation increases, the methodology requires longer and more involved algebraic manipulation, quickly surpassing what is manageable by hand. The situation is aggravated by the fact that not one but multiple algorithmic variants are desired.

We developed CLlCK with the objective of relieving the developer from this burden and enabling the automatic generation of entire libraries. In this chapter, we describe how CLlCK is capable of generating a family of loop-based blocked algorithms to compute a target operation from its sole mathematical description.

4.1 Automating FLAME: A three-stage approach

The FLAME methodology enables the systematic derivation of formally correct loop-based linear algebra algorithms. The main idea is that the correctness is not proved a posteriori, once the algorithm is built; instead, the proof of correctness and the algorithm grow hand in hand. To this end, loop invariants are identified first, and then, for each of them, a skeleton of proof is created and the algorithm is built so that the proof is satisfied. To automate the application of this methodology, we first characterize the minimal input information about the target operation necessary to automate the entire process, and then we follow the constructive three-stage approach illustrated by Figure 4.1. Here, we outline these stages; we devote the next sections to discuss each of them in detail.

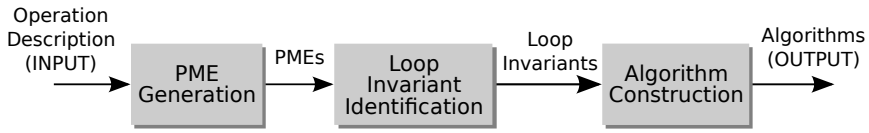


Figure 4.1: The three stages in the algorithm generation process.

Input. We define a target operation by means of two predicates: the Precondition (P_{pre}) and the Postcondition (P_{post}). The postcondition states the equation to be solved, while the precondition enumerates the properties of the operands. Box 4.1 contains the definition of the inversion of a triangular matrix L; L is overwritten with its inverse; the notation $\hat{L}$ indicates the initial contents of L. This is the only information about the operation required by CLlCK to automate the generation of algorithms.

PME generation. The first stage of the process takes the description of the input operation, and yields its *Partitioned Matrix Expression*. The PME is a decom-

$$\begin{cases} P_{\text{pre}}: \{\texttt{Overwritten}(L) \wedge \texttt{Matrix}(L) \wedge \texttt{LowerTriangular}(L)\} \\ P_{\text{post}}: \{L = \hat{L}^{-1}\} \end{cases}$$

Box 4.1: Formal description for the inversion of a lower triangular matrix.

position of the target problem into simpler sub-problems in a divide-and-conquer fashion; it exposes how each part of the output matrices is computed from parts of the input matrices. Equation (4.1) represents the PME for the triangular inverse, which states that the inverse may be decomposed as a two-sided triangular system and two smaller inverses.

$$\left(\begin{array}{c|c} L_{TL} := \hat{L}_{TL}^{-1} & 0 \\ \hline L_{BL} := -\hat{L}_{BR}^{-1}\hat{L}_{BL}\hat{L}_{TL}^{-1} & L_{BR} := \hat{L}_{BR}^{-1} \end{array}\right) \tag{4.1}$$

Loop invariant identification. The second stage of the process deals with the identification of loop invariants. A loop invariant is a boolean predicate that encodes the state of the computation at specific points of a loop: It must be satisfied before the loop is entered and at the top and the bottom of each iteration [30]. Loop invariants can be extracted as subsets of the computation encapsulated in the PME. Equation (4.2) contains one loop invariant (out of eight) for the triangular inverse; it indicates that the inverse of the top-left part of $\hat{L}$ has been computed, while the other parts of $\hat{L}$ remain to be computed.

$$\left(\begin{array}{c|c} L_{TL} := \hat{L}_{TL}^{-1} & 0 \\ \hline \neq & \neq \end{array}\right) \tag{4.2}$$

Algorithm construction. In the third and last stage, each loop invariant is transformed into its corresponding loop-based algorithm. To this end, FLAME's methodology provides a template for a proof of correctness (Figure 4.2a); the predicates P_{pre}, P_{post}, and P_{inv} are replaced, respectively, with the precondition, postcondition, and the loop invariant at hand. Then, the algorithm statements (boldface labels) are filled in so that the proof is satisfied (Figure 4.2b). The details in Figure 4.2 are not important now; they will become clear by the end of the chapter.

By automating each of these three stages, we achieve, for the first time, the complete automation of the FLAME methodology: From the mathematical description of the target operation, a family of algorithms that compute it are generated. In the following, we describe in detail how CLlCK carries out each of the three stages.

$\{P_{\text{pre}}\}$

Partition

$\{P_{\text{inv}}\}$

While G **do**

$\{P_{\text{inv}} \wedge G\}$

Repartition

LoopBody

Continue With

$\{P_{\text{inv}}\}$

end

$\{P_{\text{inv}} \wedge \neg G\}$

$\{P_{\text{post}}\}$

(a) FLAME template for a formal proof of correctness for algorithms consisting of an initialization step followed by a loop.

Algorithm: $L := \hat{L}^{-1}$

$\{L = \hat{L} \wedge \mathit{LowerTriangular}(L)\}$

Partition $L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)$

where L_{TL} is 0×0

$\{\left(\begin{array}{c|c} L_{TL} := \hat{L}_{TL}^{-1} & 0 \\ \hline \neq & \neq \end{array}\right)\}$

While $\mathit{size}(L_{TL}) < \mathit{size}(L)$ **do**

$\{\left(\begin{array}{c|c} L_{TL} := \hat{L}_{TL}^{-1} & 0 \\ \hline \neq & \neq \end{array}\right) \wedge \mathit{size}(L_{TL}) < \mathit{size}(L)\}$

Repartition

$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$

where L_{11} is $b \times b$

$L_{10} := -L_{11}^{-1} L_{10}$ (TRSM)

$L_{10} := L_{10} L_{00}^{-1}$ (TRSM)

$L_{11} := L_{11}^{-1}$ (INV)

Continue with

$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$

$\{\left(\begin{array}{c|c} L_{TL} := \hat{L}_{TL}^{-1} & 0 \\ \hline \neq & \neq \end{array}\right)\}$

endwhile

$\{\left(\begin{array}{c|c} L_{TL} := \hat{L}_{TL}^{-1} & 0 \\ \hline \neq & \neq \end{array}\right)\} \wedge \neg(\mathit{size}(L_{TL}) < \mathit{size}(L))$

$\{L := \hat{L}^{-1}\}$

(b) Algorithm to compute the inverse of a lower triangular matrix derived from loop invariant (4.2).

Figure 4.2: FLAME template for a proof of correctness, and example of FLAME algorithm. The shaded predicates are part of the proof of correctness.

4.2 Input to CLlCK

To unequivocally describe a target operation, we choose the language traditionally used to reason about program correctness: Equations shall be specified by means of the predicates Precondition (P_{pre}) and Postcondition (P_{post}) [30]. The precondition enumerates the operands that appear in the equation and describes their properties, while the postcondition specifies the equation to be solved.

We commence the discussion using the Cholesky factorization as an example: Given a symmetric positive definite (SPD) matrix A, the goal is to find a lower triangular matrix L such that $LL^T = A$. In Box 4.2, we provide the description of such a factorization; the notation $L := \Gamma(A)$ indicates that L is the Cholesky factor of A.

$$L := \Gamma(A) \equiv \begin{cases} P_{\text{pre}}: \{\texttt{Output}(L) \wedge \texttt{Matrix}(L) \wedge \texttt{LowerTriangular}(L) \wedge \\ \qquad \texttt{Input}(A) \wedge \texttt{Matrix}(A) \wedge \texttt{SPD}(A)\} \\ P_{\text{post}}: \{LL^T = A\} \end{cases}$$

Box 4.2: Formal description for the Cholesky factorization.

The definition is unambiguous, and it includes all the information specific to the operation needed by CLlCK to fully automate the derivation process.

Pattern Learning

CLlCK takes the pair of predicates in Box 4.2 and creates the pattern in Box 4.3 that identifies the Cholesky factorization. The pattern establishes that matrices L and A are one the Cholesky factor of the other, provided that the constraints in the precondition are satisfied, and that L and A are related as dictated by the postcondition ($LL^T = A$).

```
equal[ times[ L_, trans[L_] ], A_ ] /;
        isOutputQ[L] && isInputQ[A] &&
        isMatrixQ[L] && isMatrixQ[A] &&
        isLowerTriangularQ[L] && isSPDQ[A]
```

Box 4.3: Mathematica pattern representing the Cholesky factorization.

For instance, in the expression

$$XX^T = A - BC,$$

in order to determine whether $X = \Gamma(A - BC)$, the following facts need to be asserted: i) X is an unknown (output) lower triangular matrix; ii) the expression

$A - BC$ is a known (input) quantity (A, B and C are known); iii) the matrix $A - BC$ is symmetric positive definite.

The strategy for decomposing an equation in terms of simpler problems greatly relies on pattern matching. Initially, CLlCK only knows the patterns for a basic set of operations: addition, multiplication, inversion, and transposition of matrices, vectors and scalars. This information is built-in in the compiler. More complex patterns are instead dynamically learned during the process of algorithm derivation. As CLlCK's pattern knowledge increases, also does its capability of tackling complex operations.

4.3 PME generation

This section centers around the first stage of the derivation process, the generation of PMEs. As Figure 4.3 shows, such process involves three steps: 1) the partitioning of the operands in the equation, 2) matrix arithmetic involving the partitioned operands, and 3) a sequence of iterations, each consisting of algebraic manipulation and pattern matching, that yield the sought-after PMEs.

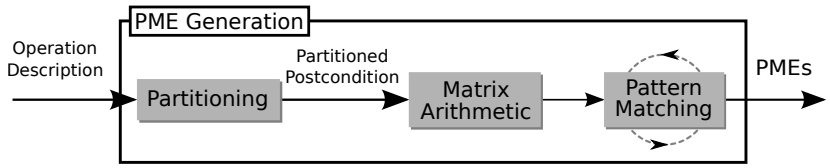


Figure 4.3: Steps for the automatic generation of PMEs.

4.3.1 Operands partitioning

We illustrate all the steps performed by CLlCK to transform the description of the input equation into one or more PMEs. The idea is to first rewrite the postcondition in terms of partitioned matrices and then apply pattern matching to identify known operations. To this end, we introduce a set of rules to partition and combine operands and to assert properties of expressions involving sub-operands. The application of these rules to the postcondition yields one or more predicates called *partitioned postcondition*. In next section, an iterative process consisting of algebraic manipulation and pattern matching will take us to the PMEs.

Operands partitioning and direct inheritance

The discussion commences with a set of rules for partitioning matrices and vectors and for transferring properties to sub-matrices and sub-vectors. These rules are part of the basic engine of CLlCK. Depending on constraints imposed by both the structure of the input operands and the postcondition, only few partitioning rules will be meaningful.

As shown in Box 4.4, a generic matrix A can be partitioned in four different ways. The 1×1 rule (Box 4.4(d)) is special, as it does not affect the operand; we refer to it as the *identity*. For a vector, only the 2×1 and 1×1 rules apply, while for scalars only the identity is admissible. When referring to any of the parts resulting from a non-identity rule, we use the terms *sub-matrix* or *sub-operand*, and for 2×2 partitionings, we also use the term *quadrant*.

$$A_{m\times n} \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \quad \text{where } A_{TL} \text{ is } k_1 \times k_2$$

(a) 2×2 rule

$$A_{m\times n} \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \quad \text{where } A_T \text{ is } k_1 \times n$$

(b) 2×1 rule

$$A_{m\times n} \rightarrow \left(\begin{array}{c|c} A_L & A_R \end{array}\right) \quad \text{where } A_L \text{ is } m \times k_2$$

(c) 1×2 rule

$$A_{m\times n} \rightarrow \left(\begin{array}{c} A \end{array}\right) \quad \text{where } A \text{ is } m \times n$$

(d) 1×1 (identity) rule

Box 4.4: Rules for partitioning a generic matrix operand A. We use the subscript letters T, B, L, and R for Top, Bottom, Left, and Right, respectively.

The inheritance of properties plays an important role in subsequent stages of the algorithm derivation. Thus, when the operands have a special structure, it is beneficial to choose partitioning rules that respect it. For a symmetric matrix, for instance, it is convenient to create sub-matrices that exhibit the same property. The 1×2 and 2×1 rules break the structure of a symmetric matrix, as neither of the two sub-matrices inherit the symmetry. Therefore, we only allow 1×1 or 2×2 partitionings, with the extra constraint that the TL quadrant has to be square.

Box 4.5 illustrates the admissible partitionings for lower triangular (L) and symmetric (M) matrices. On the left, the identity rule is applied and the operands remain unchanged. On the right instead, a constrained 2×2 rule is applied, so that some of the resulting quadrants inherit properties. For a lower triangular matrix L, both L_{TL} and L_{BR} are square and lower triangular, L_{TR} is zero, and L_{BL} is a generic matrix. For a symmetric matrix M, both M_{TL} and M_{BR} are square and symmetric, and $M_{BL} = M_{TR}^T$ (or vice versa $M_{TR} = M_{BL}^T$).

Theorem-aware inheritance Although frequent, direct inheritance of properties is only the simplest form of inheritance. Here we expose a more complex situation. Let A be an SPD matrix. Because of symmetry, the only allowed partitioning rules are the ones listed in Box 4.5(b); applying the 2×2 rule, we obtain

$$A_{m\times m} \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{BL}^T \\ \hline A_{BL} & A_{BR} \end{array}\right), \quad \text{where } A_{TL} \text{ is } k \times k \tag{4.3}$$

$$L_{m\times m} \rightarrow \left(\begin{array}{c} L \end{array}\right) \quad \text{where } L \text{ is } m \times m \qquad \text{or} \qquad L_{m\times m} \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \quad \text{where } L_{TL} \text{ is } k \times k$$

(a) Viable partitionings for a lower triangular matrix.

$$M_{m\times m} \rightarrow \left(\begin{array}{c} M \end{array}\right) \quad \text{where } M \text{ is } m \times m \qquad \text{or} \qquad M_{m\times m} \rightarrow \left(\begin{array}{c|c} M_{TL} & M_{BL}^T \\ \hline M_{BL} & M_{BR} \end{array}\right) \quad \text{where } M_{TL} \text{ is } k \times k$$

(b) Viable partitionings for a symmetric matrix.

Box 4.5: Partitioning rules for structured matrices.

and both A_{TL} and A_{BR} are symmetric. More properties about the quadrants of A can be stated. For example, it is well known that *if A is SPD, then all its principal sub-matrices are SPD* [29]. As a consequence, the quadrants A_{TL} and A_{BR} inherit such a property. Moreover, it can be proved that given a 2×2 partitioning of an SPD matrix as in (4.3), the matrices $A_{TL} - A_{BL}^T A_{BR}^{-1} A_{BL}$ and $A_{BR} - A_{BL} A_{TL}^{-1} A_{BL}^T$ (known as Schur complements) are also symmetric positive definite. The knowledge emerging from this theorem is included in CLlCK's engine. In Section 4.3.2 it will become apparent how this information is essential for the generation of PMEs.

Combining the partitionings

The partitioning rules are now applied to rewrite the postcondition equation. Since in general each operand can be decomposed in multiple ways, not one, but many partitioned postconditions are created. As an example, in the Cholesky factorization (Box 4.2) both the 1×1 and 2×2 rules are viable for both L and A, leading to four different sets of partitionings:

- Both L and A are partitioned in 1×1.
- L and A are partitioned in 1×1 and 2×2, respectively.
- L and A are partitioned in 2×2 and 1×1, respectively.
- Both L and A are partitioned in 2×2.

Table 4.1 contains the resulting four partitioned postconditions. It is apparent that some of the expressions in the fourth column are not algebraically well-defined. Consequently, in addition to constraints on each individual operand, the partitioning rules need to be such that the partitioned operands can be combined together according to standard matrix arithmetic. For instance, in the expression $X + Y$, if the 2×1 rule is applied to matrix X, the $+$ operator imposes that the same rule is applied to Y too.

#	L	A	Partitioned Postcondition
1	$L \to (L)$	$A \to (A)$	$(L)(L)^T = (A)$
2	$L \to (L)$	$A \to \left(\begin{array}{c\|c} A_{TL} & A_{BL}^T \\ \hline A_{BL} & A_{BR} \end{array}\right)$	$(L)(L)^T = \left(\begin{array}{c\|c} A_{TL} & A_{BL}^T \\ \hline A_{BL} & A_{BR} \end{array}\right)$
3	$L \to \left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)$	$A \to (A)$	$\left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)\left(\begin{array}{c\|c} L_{TL}^T & L_{BL}^T \\ \hline 0 & L_{BR}^T \end{array}\right) = (A)$
4	$L \to \left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)$	$A \to \left(\begin{array}{c\|c} A_{TL} & A_{BL}^T \\ \hline A_{BL} & A_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)\left(\begin{array}{c\|c} L_{TL}^T & L_{BL}^T \\ \hline 0 & L_{BR}^T \end{array}\right) = \left(\begin{array}{c\|c} A_{TL} & A_{BL}^T \\ \hline A_{BL} & A_{BR} \end{array}\right)$

Table 4.1: Application of the different combinations of partitioning rules to the postcondition.

With reference to Table 4.1, the rules in the second row lead to an expression whose left-hand and right-hand sides are a 1×1 and a 2×2 object, respectively. The reverse is true in the third row. Since, both lead to ill-defined partitioned postconditions, they are discarded. Despite leading to a well defined expression, the first row of the table is discarded too, as it leads to an expression in which none of the operands has been partitioned, while the goal is to obtain a *Partitioned* Matrix Expression. In light of these additional restrictions, the only viable set of rules for the Cholesky factorization is the one given in the last row of Table 4.1, with the additional constraint that the A_{TL} and L_{TL} quadrants are square.

In summary, partitioning rules must satisfy both the constraints due to the nature of the individual operands, and those due to the operators appearing in the postcondition. Next, we detail the algorithm used by CLICK to generate only the viable sets of partitioning rules.

Automation

We show how CLICK performs the partitioning process automatically. A naive approach would exhaustively search among all the rules applied to all the operands, leading to a search space of exponential size in the number of operands. Instead, CLICK utilizes an algorithm that traverses the postcondition (represented as a tree) just once, and yields only the viable sets of partitioning rules.

The algorithm builds around two main ideas: 1) the properties of an operand impose restrictions on the viable rules; 2) the operators in the postcondition also constrain the partitionings of their operands. The input to the algorithm is a target operation, in the form of the predicates P_{pre} and P_{post}. As an example, we look at the triangular Sylvester equation

$$AX + XB = C,$$

$$X := \Omega(A,B,C) \equiv \begin{cases} P_{\text{pre}} : \{\texttt{Input}(A) \wedge \texttt{Matrix}(A) \wedge \texttt{UpperTriangular}(A) \wedge \\ \qquad \texttt{Input}(B) \wedge \texttt{Matrix}(B) \wedge \texttt{UpperTriangular}(B) \wedge \\ \qquad \texttt{Input}(C) \wedge \texttt{Matrix}(C) \wedge \texttt{Output}(X) \wedge \texttt{Matrix}(X)\} \\ P_{\text{post}} : \{AX + XB = C\}. \end{cases}$$

Box 4.6: Formal description for the triangular Sylvester equation.

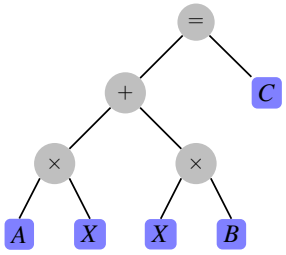


Figure 4.4: Tree representation for the Sylvester equation $AX + XB = C$.

defined formally in Box 4.6. Henceforth, we will also refer to the equation as the *Sylvester equation*. First, the algorithm transforms the postcondition to prefix notation (Figure 4.4) and collects name and dimensionality of each operand. A list of disjoint sets, one per dimension of the operands is then created. For the Sylvester equation, this initial list is

$$[\,\{A_r\},\{A_c\},\{B_r\},\{B_c\},\{C_r\},\{C_c\},\{X_r\},\{X_c\}\,],$$

where r and c stand for *rows* and *columns* respectively. The algorithm traverses the tree, in a post-order fashion, to determine if and which dimensions are bound together. Two dimensions are bound to one another if the partitioning of one implies the partitioning of the other. If two dimensions are found to be bound, then their corresponding sets are merged. As the algorithm moves from the leaves to the root of the tree, it keeps track of the dimensions of the operands' subtrees.

The algorithm starts by visiting the node corresponding to the upper triangular operand A. There it establishes that the identity and the 2×2 partitioning rules are the only admissible ones. Thus, the rows and the columns of A are bound together, and the list becomes

$$[\,\{A_r,A_c\},\{B_r\},\{B_c\},\{C_r\},\{C_c\},\{X_r\},\{X_c\}\,].$$

The next node to be visited is that of the operand X. Since X has no specific structure, its analysis causes no bindings. Then, the node corresponding to the $\times$ operator is analyzed. The dimensions of A and X have to agree according to the matrix product, therefore, a binding between A_c and X_r is imposed:

$$[\,\{A_r,A_c,X_r\},\{B_r\},\{B_c\},\{C_r\},\{C_c\},\{X_c\}\,].$$

#	A	B	C	X
1	(A)	$\left(\begin{array}{c\|c} B_{TL} & B_{TR} \\ \hline 0 & B_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} C_L & C_R \end{array}\right)$	$\left(\begin{array}{c\|c} X_L & X_R \end{array}\right)$
2	$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array}\right)$	(B)	$\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right)$	$\left(\begin{array}{c} X_T \\ \hline X_B \end{array}\right)$
3	$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} B_{TL} & B_{TR} \\ \hline 0 & B_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right)$

Table 4.2: Viable combinations of partitioning rules for the Sylvester equation.

At this stage, the dimensions of the product AX are also determined to be $A_r \times X_c$.

The procedure continues by analyzing the subtree corresponding to the product XB. Similarly to the product AX, the lack of structure in X does not cause any binding, while the triangularity of B imposes a binding between B_r and B_c leading to

$$[\,\{A_r, A_c, X_r\}, \{B_r, B_c\}, \{C_r\}, \{C_c\}, \{X_c\}\,].$$

Then, the node for the $\times$ operator is analyzed, and a binding between X_c and B_r is found:

$$[\,\{A_r, A_c, X_r\}, \{B_r, B_c, X_c\}, \{C_r\}, \{C_c\}\,].$$

The dimensions of the product XB are determined to be $X_r \times B_c$.

The next node to be considered is the corresponding to the $+$ operator. It imposes a binding between the rows and the columns of the products AX and XB, i.e., between A_r and X_r, and between X_c and B_c. Since each of these pairs of dimensions already belong to the same set, no modifications are made to the list. The algorithm establishes that the dimensions of the $+$ node are $A_r \times B_c$. Next, the node associated to the operand C is analyzed. Since C has no particular structure, its analysis does not cause any modification. The last node to be processed is the equality operator $=$. This node binds the rows of C to those of A (C_r, A_r) and the columns of C to those of B (C_c, B_c). The final list consists of two separate groups of dimensions:

$$[\,\{A_r, A_c, X_r, C_r\}, \{B_r, B_c, X_c, C_c\}\,].$$

Having created g groups of bound dimensions, the algorithm generates 2^g combinations of rules (the dimensions within each group being either partitioned or not), resulting in a family of partitioned postconditions, one per combination. In practice, since the combination including solely identity rules does not lead to a PME, only $2^g - 1$ combinations are acceptable. In our example, the algorithm found two groups of bound dimensions; therefore three possible combinations of rules are generated: 1) only the dimensions in the second group are partitioned, 2) only the dimensions in the first group are partitioned, or 3) all dimensions are partitioned. The resulting partitionings are listed in Table 4.2.

By means of this algorithm, CLlCK efficiently generates, for every target operation, only the acceptable sets of partitioning rules.

4.3.2 Matrix arithmetic and pattern matching

This section covers the second and third steps in the stage of PME generation (Figure 4.3). Given a partitioned postcondition, within the *Matrix Arithmetic* step, symbolic arithmetic is performed and the = operator is distributed over the partitions, originating multiple equations. In Equation (4.4), we display the result of these actions for the Cholesky factorization, where the symbol $\star$ means that the equation in the top-right quadrant is the transpose of the one in the bottom-left quadrant.

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)\left(\begin{array}{c|c} L_{TL}^T & L_{BL}^T \\ \hline 0 & L_{BR}^T \end{array}\right) = \left(\begin{array}{c|c} A_{TL} & A_{BL}^T \\ \hline A_{BL} & A_{BR} \end{array}\right) \Rightarrow \left(\begin{array}{c|c} L_{TL}L_{TL}^T = A_{TL} & \star \\ \hline L_{BL}L_{TL}^T = A_{BL} & L_{BL}L_{BL}^T + L_{BR}L_{BR}^T = A_{BR} \end{array}\right). \tag{4.4}$$

The last step, *Pattern Matching*, carries out an iterative process during which CLlCK finds the solution for each of the equations resulting from the previous step. When such a solution is found, it is expressed as an assignment to the unknown(s) of an implicit or explicit function of known quantities. The unknown(s) or output(s) are then labeled as computable (the system knows now the formula to compute them) and become known quantities to the remaining equations. Upon completion, the process delivers the sought-after PME.

Success of this step is dependent on the ability to identify expressions with known structure and properties. In order to facilitate pattern matching, we force equations to be in their *canonical form*. We state that an equation is in canonical form if a) its left-hand side only consists of those terms that contain at least one unknown object, and b) its right-hand side only consists of those terms that solely contain known objects.

The iterative process comprises three separate actions: 1) algebraic manipulation: The equations are rearranged in canonical form; 2) structural pattern matching: Equations are matched against known patterns; 3) exposing new available operands: Once a known pattern is matched, the equation becomes an assignment, and the unknown operands are flagged as known.

We clarify the iterative process by illustrating, action by action, how CLlCK works through the Cholesky factorization. During the discussion, green and red are used to highlight the known and unknown operands, respectively. The **first iteration** is depicted in Box 4.7, in which the top-left formula (4.7(a)) displays the initial state. Initially, parts of A are known, and parts of L are unknown.

Algebraic manipulation

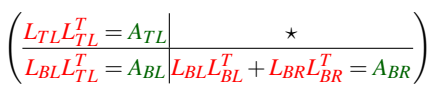
All the equations in Box 4.7(a) are in canonical form, and no manipulation is required.

Structural pattern matching

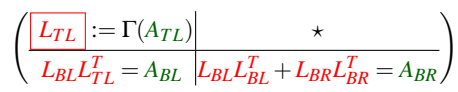
The three equations are tested against known patterns. CLlCK recognizes the top-left quadrant, which matches the pattern for the Cholesky factorization in Box 4.3. The system rewrites the equation as an assignment (Box 4.7(b)), and labels the output quantity, L_{TL}, as computable.

Exposing new available operands

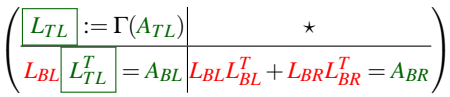
Having matched the equation in the top-left quadrant, CLlCK turns the unknown quantity L_{TL} into L_{TL}, and propagates the information to all the other quadrants: Box 4.7(c). The first iteration ends.



(a) Initial state. No manipulation is required.



(b) The top-left equation is identified as a Cholesky sub-problem.



(c) The operand L_{TL} becomes known for the rest of equations.

Box 4.7: First iteration towards the PME generation.

In this first iteration, one unknown operand, L_{TL}, has become known, and one equation has turned into an assignment. The **second iteration** is shown in Box 4.8.

Algebraic manipulation

Box 4.8(a) reproduces the final state from the previous iteration. The remaining equations are still in canonical form, thus no operation takes place.

Structural pattern matching

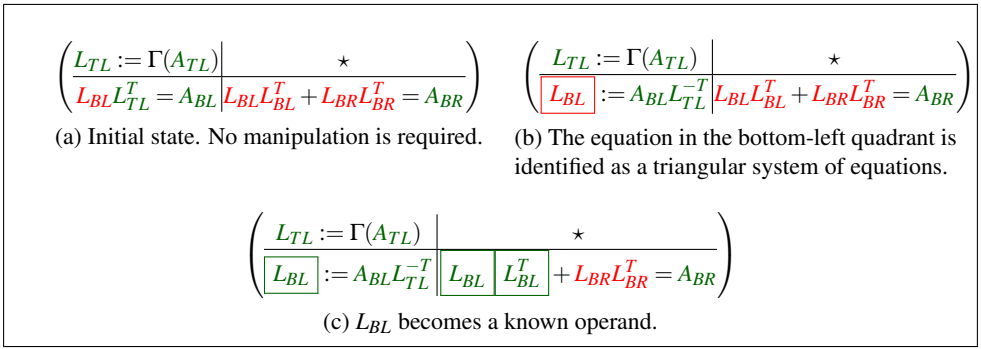
Among the two outstanding equations, the one in the bottom-left quadrant is identified (Box 4.8(b)), as it matches the pattern of a triangular system of equations with multiple right-hand sides (TRSM). The pattern for a TRSM is

```
equal[ times[ X_, trans[L_] ], B_ ] /;
      isOutputQ[X] && isInputQ[L] && isInputQ[B] &&
      isLowerTriangularQ[L].
```

For the sake of brevity, we assume that CLlCK learned such pattern from a previous derivation; in practice, a nested task of PME generation could be initiated, yielding the required pattern.

Exposing new available operands

Once the TRSM is identified, the output operand L_{BL} becomes available and turns to green in the bottom-right quadrant (Box 4.8(c)).



(a) Initial state. No manipulation is required.

(b) The equation in the bottom-left quadrant is identified as a triangular system of equations.

(c) L_{BL} becomes a known operand.

Box 4.8: Second iteration towards the PME generation.

The process continues until all the equations are turned into assignments. The third and **final iteration** for the Cholesky factorization is shown in Box 4.9.

Algebraic manipulation

The bottom-right equation is not in canonical form anymore: The product $L_{BL}L_{BL}^T$, now a known quantity, does not lay in the right-hand side. A simple manipulation brings the equation back to canonical form (Box 4.9(a)).

Structural pattern matching

Only the equation in the bottom-right quadrant remains unprocessed. At a first glance, one might recognize a Cholesky factorization, but the corresponding pattern in Box 4.2 requires A to be SPD. The question is whether the expression $A_{BR} - L_{BL}L_{BL}^T$ represents an SPD matrix. In order to answer the question, CLlCK applies rewrite rules and symbolic simplifications.

In Section 4.3.1, we explained that the following facts regarding the quadrants of A are known:

- SPD(A_{TL})
- SPD(A_{BR})
- SPD($A_{TL} - A_{BL}^T A_{BR}^{-1} A_{BL}$)
- SPD($A_{BR} - A_{BL}A_{TL}^{-1}A_{BL}^T$)

In order to determine whether $A_{BR} - L_{BL}L_{BL}^T$ is equivalent to any of the expressions listed above, CLlCK makes use of the knowledge acquired throughout the previous iterations. Specifically, in the first two iterations it was discovered that

- $L_{TL}L_{TL}^T = A_{TL}$, and
- $L_{BL} = A_{BL}L_{TL}^{-T}$.

Using these identities as rewrite rules, the expression $A_{BR} - L_{BL}L_{BL}^T$ is manipulated: First, $L_{BL} = A_{BL}L_{TL}^{-T}$ is used to replace the instances of L_{BL}, yielding $A_{BR} - A_{BL}L_{TL}^{-T}L_{TL}^{-1}A_{BL}^T$, and equivalently, $A_{BR} - A_{BL}(L_{TL}L_{TL}^T)^{-1}A_{BL}^T$; then, by virtue of the identity $L_{TL}L_{TL}^T = A_{TL}$, $L_{TL}L_{TL}^T$ is replaced by A_{TL}, yielding $A_{BR} - A_{BL}A_{TL}^{-1}A_{BL}^T$, which is known to be SPD. Now that CLICK can assert the SPDness of $A_{BR} - L_{BL}L_{BL}^T$, it successfully associates the equation in the bottom-right quadrant with the pattern for a Cholesky factorization, and L_{BR} is labeled as computable.

Exposing new available operands

Once the expression in the bottom-right quadrant is identified, the system exposes the quantity L_{BR} as known. Since no equation is left, the process completes and the PME—formed by the three assignments—is returned as output.

$$\left(\begin{array}{c|c} L_{TL} := \Gamma(A_{TL}) & \star \\ \hline L_{BL} := A_{BL}L_{TL}^{-T} & L_{BR}L_{BR}^T = A_{BR} - L_{BL}L_{BL}^T \end{array}\right)$$

(a) Simple algebraic manipulation takes the bottom-right equation back to canonical form.

$$\left(\begin{array}{c|c} L_{TL} := \Gamma(A_{TL}) & \star \\ \hline L_{BL} := A_{BL}L_{TL}^{-T} & \boxed{L_{BR}} := \Gamma(A_{BR} - L_{BL}L_{BL}^T) \end{array}\right)$$

(b) The bottom-right equation is identified as a Cholesky factorization.

$$\left(\begin{array}{c|c} L_{TL} := \Gamma(A_{TL}) & \star \\ \hline L_{BL} := A_{BL}L_{TL}^{-T} & \boxed{L_{BR}} := \Gamma(A_{BR} - L_{BL}L_{BL}^T) \end{array}\right)$$

(c) The operand L_{BR} becomes known.

Box 4.9: Final iteration towards the PME generation.

By means of the described process, PMEs for a target equation are automatically generated. The PME for the Cholesky factorization is given in Box 4.10. We point out that the decomposition encoded by the PME is correct independently of the size of the quadrants (as long as A_{TL} and L_{TL} are square, as specified by the initial partitioning of the matrices L and A).

$$\left(\begin{array}{c|c} L_{TL} := \Gamma(A_{TL}) & \star \\ \hline L_{BL} := A_{BL}L_{TL}^{-T} & L_{BR} := \Gamma(A_{BR} - L_{BL}L_{BL}^T) \end{array}\right)$$

Box 4.10: Partitioned Matrix Expression for the Cholesky factorization.

Before proceeding with the second stage (*Loop Invariant Identification*), we briefly discuss the existence of multiple PMEs for a single operation, and the relation of the object PME with recursive divide-and-conquer algorithms.

4.3.3 Non-uniqueness of the PME

For the Cholesky factorization, CLlCK identifies that only one set of partitioning rules is feasible (Table 4.1), which corresponds to one way of decomposing the problem and to the generation of one PME. In general, the PME *is not unique* since, for one target operation, multiple sets of viable rules may be found, each of them leading to a different problem decomposition and a different PME. To illustrate such a situation, we look once more at the triangular Sylvester equation (Box 4.6).

The procedure described in Section 4.3.1 is used to obtain the sets of admissible partitioning rules, listed in Table 4.2. Each of them is then applied to the postcondition equation, obtaining three different partitioned postconditions, as shown in Table 4.3 (left). By applying the iterative process described in Section 4.3.2, three PMEs are generated: Table 4.3 (right). In Box 4.11, we illustrate the steps performed by CLlCK to transform the second partitioned postcondition into a PME.

$$\left(\frac{A_{TL}X_T + A_{TR}X_B + X_T B = C_T}{A_{BR}X_B + X_B B = C_B} \right)$$

(a) Initial state.

$$\left(\frac{A_{TL}X_T + A_{TR}X_B + X_T B = C_T}{\boxed{X_B} := \Omega(A_{BR}, B, C_B)} \right)$$

(b) The bottom equation is identified as a Sylvester equation, where all the input operands are known.

$$\left(\frac{A_{TL}X_T + A_{TR}\boxed{X_B} + X_T B = C_T}{\boxed{X_B} := \Omega(A_{BR}, B, C_B)} \right)$$

(c) The operand X_T becomes available for the equation in the top quadrant.

$$\left(\frac{A_{TL}X_T + X_T B = C_T - A_{TR}X_B}{X_B := \Omega(A_{BR}, B, C_B)} \right)$$

(d) State after algebraic manipulation.

$$\left(\frac{\boxed{X_T} := \Omega(A_{TL}, B, C_T - A_{TR}X_B)}{X_B := \Omega(A_{BR}, B, C_B)} \right)$$

(e) The equation in the top quadrant is also identified as a Sylvester equation.

$$\left(\frac{X_T := \Omega(A_{TL}, B, C_T - A_{TR}X_B)}{X_B := \Omega(A_{BR}, B, C_B)} \right)$$

(f) Resulting PME.

Box 4.11: Generation of the PME corresponding to the second partitioned postcondition (Table 4.3) of the Sylvester equation.

Learning the PMEs

As we discuss in the *Loop Invariant Identification* stage, the generated PMEs represent a pool of loop invariants. However, this is not the only place where the information encoded in the PMEs is used. In the last stage of the algorithm generation process, the *Algorithm Construction*, CLlCK makes use of the PMEs to

#	Partitioned Postcondition	Partitioned Matrix Expression
1	$(A)\left(\begin{array}{c\|c} X_L & X_R \end{array}\right) + \left(\begin{array}{c\|c} X_L & X_R \end{array}\right)\left(\begin{array}{c\|c} B_{TL} & B_{TR} \\ \hline 0 & B_{BR} \end{array}\right) = \left(\begin{array}{c\|c} C_L & C_R \end{array}\right)$	$\left(X_L := \Omega(A, B_{TL}, C_L) \middle\| X_R := \Omega(A, B_{BR}, C_R - X_L B_{TR})\right)$
2	$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array}\right)\left(\begin{array}{c} X_T \\ \hline X_B \end{array}\right) + \left(\begin{array}{c} X_T \\ \hline X_B \end{array}\right)(B) = \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right)$	$\left(\begin{array}{c} X_T := \Omega(A_{TL}, B, C_T - A_{TR} X_B) \\ \hline X_B := \Omega(A_{BR}, B, C_B) \end{array}\right)$
3	$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array}\right)\left(\begin{array}{c\|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) + \left(\begin{array}{c\|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right)\left(\begin{array}{c\|c} B_{TL} & B_{TR} \\ \hline 0 & B_{BR} \end{array}\right) = \left(\begin{array}{c\|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} X_{TL} := \Omega(A_{TL}, B_{TL}, C_{TL} - A_{TR} X_{BL}) & \begin{array}{l} X_{TR} := \Omega(A_{TL}, B_{BR}, \\ \quad C_{TR} - A_{TR} X_{BR} - X_{TL} B_{TR}) \end{array} \\ \hline X_{BL} := \Omega(A_{BR}, B_{TL}, C_{BL}) & X_{BR} := \Omega(A_{BR}, B_{BR}, C_{BR} - X_{BL} B_{TR}) \end{array}\right)$

Table 4.3: Partitioned postconditions and Partitioned Matrix Expressions for the triangular Sylvester equation.

rewrite expressions involving partitioned operands into multiple expressions. For instance, the triangular Sylvester equation in its implicit form $X = \Omega(A,B,C)$, with the following combination of partitioned operands:

$$\left(\begin{array}{c} X_T \\ \hline X_B \end{array}\right) := \Omega\left(\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array}\right), (B), \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right)\right),$$

needs to be rewritten as

$$\left(\begin{array}{c} X_T := \Omega(A_{TL},B,C_T - A_{TR}X_B) \\ \hline X_B := \Omega(A_{BR},B,C_B) \end{array}\right).$$

To enable such transformations, CL1CK produces one rewrite rule per PME, and incorporates them into its knowledge-base: For the example of the Sylvester equation, the system translates the PMEs in Table 4.3, into the rewrite rules in Box 4.12.

1. $\left(X_L \mid X_R\right) := \Omega\left((A), \left(\begin{array}{c|c} B_{TL} & B_{TR} \\ \hline 0 & B_{BR} \end{array}\right), \left(C_L \mid C_R\right)\right) \longrightarrow \left(X_L := \Omega(A,B_{TL},C_L) \,\middle|\, X_R := \Omega(A,B_{BR},C_R - X_L B_{TR})\right)$

2. $\left(\begin{array}{c} X_T \\ \hline X_B \end{array}\right) =: \Omega\left(\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array}\right), (B), \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right)\right) \longrightarrow \left(\begin{array}{c} X_T := \Omega(A_{TL},B,C_T - A_{TR}X_B) \\ \hline X_B := \Omega(A_{BR},B,C_B) \end{array}\right)$

3. $\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) := \Omega\left(\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array}\right), \left(\begin{array}{c|c} B_{TL} & B_{TR} \\ \hline 0 & B_{BR} \end{array}\right), \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right)\right) \longrightarrow$

$$\left(\begin{array}{c|c} X_{TL} := \Omega(A_{TL},B_{TL},C_{TL} - A_{TR}X_{BL}) & X_{TR} := \Omega(A_{TL},B_{BR},C_{TR} - A_{TR}X_{BR} - X_{TL}B_{TR}) \\ \hline X_{BL} := \Omega(A_{BR},B_{TL},C_{BL}) & X_{BR} := \Omega(A_{BR},B_{BR},C_{BR} - X_{BL}B_{TR}) \end{array}\right)$$

Box 4.12: Rewrite rules representing the knowledge acquired by CL1CK after generating the three PMEs for the Sylvester equation.

4.3.4 Recursive algorithms

The concept of PME leads naturally to recursive divide-and-conquer algorithms. Such algorithms consist of three main parts: The decomposition of the problem into smaller sub-problems, the computation of these sub-problems, and the composition of the solution from the partial results. In this scheme, the PME acts as the decomposition operator, showing how to decompose the problem into sub-problems; then, each partial result is obtained, and the output matrix is composed by assembling its subparts.

In the case of Cholesky, the PME determines that the operation may be computed as 1) a recursive call involving smaller matrices, followed by 2) the solution of a triangular system, and 3) one more recursive call applied to a matrix product. The base case of the recursion involves the corresponding scalars from L and A,

$\lambda := \Gamma(\alpha)$. This base case is equivalent to the equation $\lambda \times \lambda = \alpha$ for unknown λ, whose solution is $\lambda := \sqrt{\alpha}$. Algorithm 4.1 presents such a recursive algorithm for the computation of the Cholesky factorization.

Algorithm 4.1: Recursive algorithm to compute the Cholesky factorization.

```
L := Cholesky(A)
  if size(A) is 1×1
  then
    L := √A
  else
    (n,n) := size(A)
    L → ( L_TL | 0 )
        ( L_BL | L_BR ),   with  L_TL ∈ R^{n/2 × n/2}
    A → ( A_TL | A_TR )
        ( A_BL | A_BR ),   with  A_TL ∈ R^{n/2 × n/2}

    L_TL := Cholesky(A_TL)
    L_BL := A_BL L_TL^{-T}
    L_BR := Cholesky(A_BR − L_BL L_BL^T)

    L ← ( L_TL | 0 )
        ( L_BL | L_BR )
  end
```

While it is worth noticing that once the PME is generated, recursive algorithms can already be derived, for performance reasons we focus on the derivation of families of loop-based algorithms.

4.4 Loop invariant identification

We focus now on the second stage of the algorithm generation process, the Loop Invariant Identification. We recall that a loop invariant expresses the contents of the output matrices (the state of the computation) at different points of a loop. Inherently, a loop invariant describes an intermediate result towards the complete computation of the target operation. Thus, loop invariants can be identified by selecting different subsets of the operations in the PME that satisfy the data dependencies.

In this second stage, CLlCK takes the PME(s) of the target equation, and produces a family of loop invariants. The identification of loop invariants consists of three steps (Figure 4.5): 1) each of the assignments in the PME is decomposed into its building blocks, the *tasks*; 2) an analysis of dependencies among tasks is carried out to build a dependency graph; 3) the graph is traversed, selecting all possible subgraphs that satisfy the dependencies. The subgraphs correspond to predicates that are candidates to becoming loop invariants. CLlCK checks the feasibility

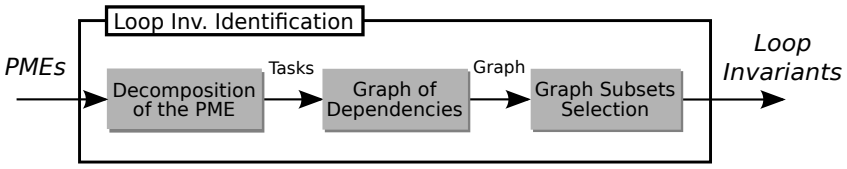


Figure 4.5: Steps for the identification of loop invariants from a PME.

of such predicates, discarding the non-feasible ones and promoting the remaining ones to loop invariants.

4.4.1 Decomposition of the PME

CLlCK commences by analyzing the assignments in the PME. All assignments share the same structure: the left-hand side includes one or more output quantities, which are computed according to the expression on the right-hand side. Similarly to the approach discussed in Chapter 2, CLlCK decomposes the right-hand sides into one or more building blocks.

Here, additionally to kernels such as matrix products and additions, the expressions to decompose may also include implicit functions. Functions will be considered building blocks; when one or more of the arguments is an expression, the expression is also decomposed. We illustrate the application of the decomposition rules by example. The following patterns will arise in the discussion:

1. `times[inv[A_], B_] /; isTriangularQ[A]`
2. `times[B_, inv[A_]] /; isTriangularQ[A]`
3. `plus[times[A_, B_], C_]`
4. `f_[`x_1`, ..., `x_n`] /;` $\forall_i$ `isOperandQ[`x_i`]`
5. `f_[`x_1`, ..., `x_n`] /;` $\exists_i$ `isExpressionQ[`x_i`]`.

The first two patterns represent triangular systems (TRSM), the third one corresponds to a matrix product $AB+C$ (GEMM), the fourth matches functions where all arguments are simple operands, and the fifth matches functions where at least one argument is an expression.

As an example, we choose the LU factorization, which is defined as

$$\{L,U\} := LU(A) \equiv \begin{cases} P_{\text{pre}} : \{\texttt{Output}(L) \wedge \texttt{Matrix}(L) \wedge \texttt{LowerTriangular}(L) \wedge \\ \qquad \texttt{UnitDiagonal}(L) \wedge \texttt{Output}(U) \wedge \texttt{Matrix}(U) \wedge \\ \qquad \texttt{UpperTriangular}(U) \wedge \texttt{Input}(A) \wedge \texttt{Matrix}(A) \wedge \exists\, \texttt{LU}(A)\} \,. \\ \\ P_{\text{post}} : \{LU = A\} \end{cases}$$

The corresponding PME comprises four assignments:

$$\left(\begin{array}{c|c} \{L_{TL}, U_{TL}\} := LU(A_{TL}) & U_{TR} := L_{TL}^{-1} A_{TR} \\ \hline L_{BL} := A_{BL} U_{TL}^{-1} & \{L_{BR}, U_{BR}\} := LU(A_{BR} - L_{BL} U_{TR}) \end{array}\right).$$

The decomposition of the assignments can be performed independently from one another; CLlCK arbitrarily commences from the top-left quadrant: $\{L_{TL}, U_{TL}\} := LU(A_{TL})$. Since the right-hand side matches the pattern associated to a function with simple operands, no decomposition is necessary; the system returns one task: $\{L_{TL}, U_{TL}\} := LU(A_{TL})$.

The analysis proceeds with the top-right quadrant: $U_{TR} := L_{TL}^{-1} A_{TR}$. The expression is identified as a TRSM operation;[1] CLlCK recognizes it as a basic task and returns it. Similarly, in the bottom-left quadrant a third task is matched and yielded.

Only one assignment remains to be studied: $\{L_{BR}, U_{BR}\} := LU(A_{BR} - L_{BL} U_{TR})$, whose right-hand side corresponds to a function with an expression as input argument. In such a case, the system first decomposes the expression, yielding a number of tasks, and then returns the function itself as a task. The expression $A_{BR} - L_{BL} U_{TR}$ matches the pattern for a matrix product (GEMM), corresponding to a basic task. Accordingly, CLlCK returns two tasks: $A_{BR} := A_{BR} - L_{BL} U_{TR}$, and $\{L_{BR}, U_{BR}\} := LU(A_{BR})$. In total, the following five tasks are produced:

1. $\{L_{TL}, U_{TL}\} := LU(A_{TL})$;
2. $U_{TR} := L_{TL}^{-1} A_{TR}$;
3. $L_{BL} := A_{BL} U_{TL}^{-1}$;
4. $A_{BR} := A_{BR} - L_{BL} U_{TR}$;
5. $\{L_{BR}, U_{BR}\} := LU(A_{BR})$.

4.4.2 Graph of dependencies

Once the decomposition into tasks is available, CLlCK proceeds with the study of the dependencies among them. Three different kinds of dependencies may occur.

- **True dependency.** One of the input arguments of a task is also the result of a previous task:

$$A := B + C$$
$$X := A + D$$

  The order of the assignments cannot be reversed because the second one requires the value of A computed in the first one.

[1] The triangularity of L_{TL} is inferred during the partitioning of L.

- **Anti dependency.** One of the input arguments of a task is also the result of a subsequent task:

$$X := A + D$$
$$A := B + C$$

  The order of the statements cannot be reversed because the first one needs the value of A before the second one overwrites it.

- **Output dependency.** The result of a task is also the result of a different task:

$$A := B + C$$
$$A := D + E$$

  The second assignment cannot be performed until the first is computed to ensure the correct final value of A.

Since, in general, there is no explicit ordering among the produced tasks, the distinction between true and anti dependencies is not straightforward. However, since assignments from different quadrants compute different parts of the output matrices, any time the output of a statement is found as an input argument of another one, it implies a true dependency: first the quantity is computed, then it is used elsewhere. The only exception is the case when the occurrence of the operand as input is labeled with a hat, refering to the initial contents of an overwritable operand. For instance, given the following pair of tasks:

$$X_{BL} = \hat{X}_{BL} - A_{BL}B_{TL}$$
$$X_{BR} = \hat{X}_{BR} - \hat{X}_{BL}B_{TR},$$

the second needs the initial contents of X_{BL} before it is overwritten by the first, thus imposing an anti dependency.

Similarly, it is not easy to distinguish the direction of an output dependency. Since output dependencies only occur among tasks belonging to the same quadrant (each quadrant writes to a different part of the output matrices), the order is determined because one of the involved tasks comes from the decomposition of the other one, imposing an order in their execution.

We detail the analysis of the dependencies following the example of the *LU* factorization. During the analysis we use **boldface** to highlight the dependencies. The study commences with Task 1, whose output is $\{L_{TL}, U_{TL}\}$. CLICK finds that the operands L_{TL} and U_{TL} are input arguments for Tasks 2 and 3, respectively.

1. $\{\mathbf{L_{TL}}, \mathbf{U_{TL}}\} := LU(A_{TL})$
2. $U_{TR} := \mathbf{L_{TL}}^{-1} A_{TR}$
3. $L_{BL} := A_{BL} \mathbf{U_{TL}}^{-1}$

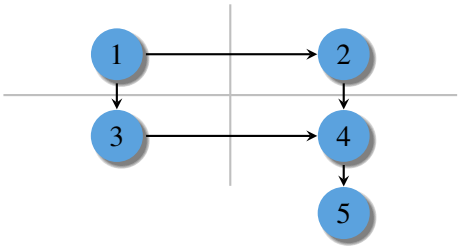


Figure 4.6: Final graph of dependencies for the *LU* factorization.

This means that two true dependencies exist: one from Task 1 to Task 2 and another from Task 1 to Task 3. Next, CLlCK inspects Task 2, whose output is U_{TR}. U_{TR} is also identified as input for Task 4.

2. $\mathbf{U_{TR}} := L_{TL}^{-1} A_{TR}$

4. $A_{BR} := A_{BR} - L_{BL}\mathbf{U_{TR}}$

Hence, a true dependency from Tasks 2 to 4 is imposed. A similar situation arises when inspecting Task 3, originating a true dependency from Task 3 to Task 4.

The analysis continues with Task 4; this computes an update of A_{BR}, which is then used as input by Task 5, thus, creating one more true dependency.

4. $\mathbf{A_{BR}} := A_{BR} - L_{BL}U_{TR}$

5. $\{L_{BR}, U_{BR}\} := LU(\mathbf{A_{BR}})$

Task 5 remains to be analyzed. Since its output, $\{L_{BR}, U_{BR}\}$, does not appear in any of the other tasks, no new dependencies are found.

In Figure 4.6, the list of the dependencies for the *LU* factorization is mapped onto a graph in which node *i* represents Task *i*. We note that, by construction of the PME —where equations are matched individually, not allowing interdependencies among assignments—, no cyclic dependencies may arise among tasks; thus, the resulting graph is a direct acyclic graph (DAG).

4.4.3 DAG subsets selection

Once CLlCK has generated the dependency graph, it selects all the possible subgraphs that satisfy the dependencies. Each of them corresponds to a different loop invariant, provided that it is feasible. A loop invariant is feasible if it satisfies a number of constraints imposed by the FLAME methodology.

CLlCK finds all possible subgraphs by means of Algorithm 4.2. The algorithm starts by sorting the nodes in the graph; since the graph is a DAG, the nodes may be sorted by levels according to the longest path from the root. For the *LU* factorization the sorted DAG is shown in Figure 4.7. Then, the algorithm creates the list of subgraphs of the DAG incrementally, by levels. At first it initializes the list of subgraphs with the empty subset, $l = [\{\}]$, which is equivalent to selecting none of

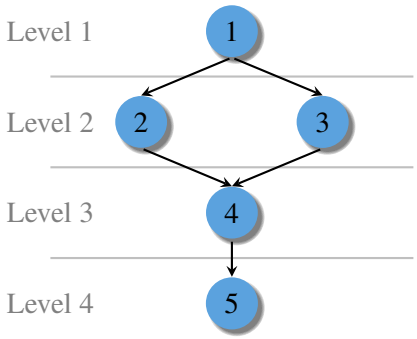


Figure 4.7: Graph of dependencies for the *LU* factorization with its nodes sorted by levels according to the distance from the root node.

the PME tasks. Then, at each level it extends the set of subgraphs by adding all those resulting from appending accesible nodes to the existing subgraphs. A node at a given level is accesible from a subgraph *sg* if all the dependencies of the node are satisfied by *sg*.

Algorithm 4.2: Generation of all the subgraphs of a DAG g.

```
l = [{}]
g' = sortByLevels(g)
for each level i in g':
  for each subgraph sg in l:
    acc = accesibleNodesFrom(sg, g', i)
    subsets = nonEmptySubsets(acc)
    for each subset ss in subsets:
      l = append(l, union(sg, ss))
  end
end
```

In the first iteration of the *LU* example, the only accesible node from $\{\}$ at level 1 is node 1, hence, union($\{\}$, $\{1\}$) is added to l, which becomes $[\{\},\{1\}]$. Now, the level is increased to 2; no node in level 2 is accesible from $\{\}$, while both nodes 2 and 3 are accesible from $\{1\}$. The union of $\{1\}$ with the non-empty subsets of $\{2,3\}$ —$\{2\}$, $\{3\}$ and $\{2,3\}$— are added to l, resulting in $l = [\{\},\{1\},\{1,2\},\{1,3\},\{1,2,3\}]$. At level 3, CLICK discovers that node 4 is accesible from subgraph $\{1,2,3\}$, thus $\{1,2,3,4\}$ is added to l. Finally, node 5 is accesible from $\{1,2,3,4\}$. The final list of subgraphs is:

$$[\{\},\{1\},\{1,2\},\{1,3\},\{1,2,3\},\{1,2,3,4\},\{1,2,3,4,5\}].$$

Checking the feasibility of the loop invariants

The seven subgraphs included in the final list correspond to predicates that are candidates to becoming loop invariants. As a final step to complete the identification

of loop invariants, CLICK must check each predicate to establish its feasibility.

The FLAME methodology (see the skeleton in Figure 4.2a) imposes two constraints for such a predicate to be a feasible loop invariant (P_{inv}):

1) There must exist a basic initialization of the operands, i.e., an initial partitioning, that renders the predicate P_{inv} true:

$$\boxed{\begin{array}{l}\{P_{\text{pre}}\}\\ \textbf{Partition}\\ \{P_{\text{inv}}\}.\end{array}}$$

2) P_{inv} and the negation of the loop guard, G, must imply the postcondition, P_{post}:

$$\boxed{P_{\text{inv}} \wedge \neg G \implies P_{\text{post}}.}$$

The partitioning and the traversal of the operands play a central role in checking the feasibility of loop invariants. The partitionings were already fixed in the previous stage (*PME generation*); however, the traversal of the operands is not determined by their partitioning, and thus is yet to be established. In the LU example, all three operands, —L, U, and A— are partitioned in 2×2 quadrants. In principle, each of the operands can be traversed in one of four ways: from the top-left to the bottom-right corners ($\boxed{\searrow}$), bottom-left to top-right ($\boxed{\nearrow}$), bottom-right to top-left ($\boxed{\nwarrow}$), and top-right to bottom-left ($\boxed{\swarrow}$). CLICK determines the traversal of the operands based on the sorted DAG (Figure 4.7): The DAG indicates that computation starts from Task 1 ($\{L_{TL}, U_{TL}\} := LU(A_{TL})$), which involves the top-left quadrants of the operands (L_{TL}, U_{TL}, and A_{TL}); thus the three operands are traversed from the top-left to the bottom-right corners.

The system is now ready to check the feasibility of the seven predicates. We illustrate the process by example; we use the the predicate corresponding to the subgraph $\{1, 2\}$:

$$\left(\begin{array}{c|c} \{L_{TL}, U_{TL}\} = LU(A_{TL}) & U_{TR} = L_{TL}^{-1} A_{TR} \\ \hline \neq & \neq \end{array}\right). \tag{4.5}$$

The initial partitioning of the operands (statement **Partition** in the skeleton) is given by the rewrite rules in Box 4.13. Notice that the top-left, top-right, and bottom-left quadrants are, respectively, of size 0×0, $0 \times m$, and $m \times 0$, i.e., they are empty. Thus, the initial partitioning renders the loop invariant (4.5) true.

CLICK checks now the second constraint: $P_{\text{inv}} \wedge \neg G \implies P_{\text{post}}$. The loop guard G follows from the traversal of the operands: Initially L_{TL}, U_{TL}, and A_{TL} are empty; the loop is executed until the matrices are traversed completely. Hence, G equals $A_{TL} < A$ (and, accordingly, $L_{TL} < L$ and $U_{TL} < U$). After the loop completes, the

$$L_{m\times m} \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right), \quad U_{m\times m} \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right), \text{ and } \quad A_{m\times m} \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$$

where L_{TL} is 0×0 where U_{TL} is 0×0 where A_{TL} is 0×0

Box 4.13: Initial partitioning of the operands for the LU factorization.

$$\left(\begin{array}{c|c} \{L_{TL}, U_{TL}\} = LU(A_{TL}) & U_{TR} = L_{TL}^{-1} A_{TR} \\ \hline \neq & \neq \end{array}\right) \wedge (A_{TL} = A \wedge L_{TL} = L \wedge U_{TL} = U)$$

$$\Longrightarrow \quad \{L, U\} = LU(A),$$

negation of the loop guard $\neg G$ means that the matrices L_{TL}, U_{TL}, and A_{TL} equal L, U, and A (and the rest of the quadrants are empty). Predicate (4.5) satisfies the second constraint:
and therefore it represents a feasible loop invariant.

Out of the seven candidate predicates, the subgraph $\{1, 2, 3, 4, 5\}$, corresponding to the full PME, fails to satisfy the first constraint, while the subgraph $\{\}$, corresponding to an empty predicate, fails to satisfy the second. The remaining five predicates satisfy both feasibility constraints and are promoted to valid loop invariants for the LU factorization (Table 4.4).

We remark that the full and empty predicates for every target operation always fail to satisfy the first and second constraints, respectively. Accordingly, these are always discarded.

4.4.4 A more complex example: the coupled Sylvester equation

To illustrate the potential of CLICK, we apply it to the coupled triangular Sylvester equation (Box 4.14), an example where the complexity of the graph of dependencies and the number of loop invariants are such that automation becomes an indispensable tool.

$$\{X, Y\} := \Psi(A, B, C, D, E, F) \equiv \begin{cases} P_{\text{pre}} : \{\texttt{Input}(A, B, C, D, E, F) \wedge \texttt{Output}(X, Y) \wedge \\ \qquad \texttt{Matrix}(A, B, C, D, E, F, X, Y) \wedge \\ \qquad \texttt{LowerTriangular}(A, D) \wedge \texttt{UpperTriangular}(B, E) \\ P_{\text{post}} : \begin{cases} AX + YB = C \\ DX + YE = F \end{cases} \end{cases}$$

Box 4.14: Formal description of the coupled Sylvester equation.

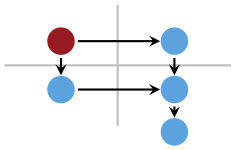
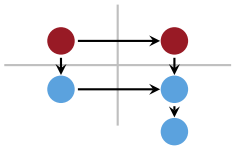
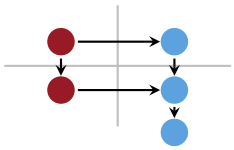
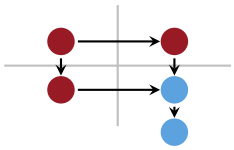
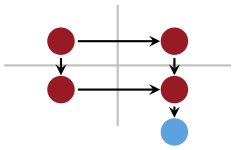
#	Subgraph	Loop invariant
1		$\left(\begin{array}{c\|c} \{L_{TL},U_{TL}\} := LU(A_{TL}) & \neq \\ \hline \neq & \neq \end{array}\right)$
2		$\left(\begin{array}{c\|c} \{L_{TL},U_{TL}\} := LU(A_{TL}) & U_{TR} := L_{TL}^{-1}A_{TR} \\ \hline \neq & \neq \end{array}\right)$
3		$\left(\begin{array}{c\|c} \{L_{TL},U_{TL}\} := LU(A_{TL}) & \neq \\ \hline L_{BL} := A_{BL}U_{TL}^{-1} & \neq \end{array}\right)$
4		$\left(\begin{array}{c\|c} \{L_{TL},U_{TL}\} := LU(A_{TL}) & U_{TR} := L_{TL}^{-1}A_{TR} \\ \hline L_{BL} := A_{BL}U_{TL}^{-1} & \neq \end{array}\right)$
5		$\left(\begin{array}{c\|c} \{L_{TL},U_{TL}\} := LU(A_{TL}) & U_{TR} := L_{TL}^{-1}A_{TR} \\ \hline L_{BL} := A_{BL}U_{TL}^{-1} & A_{BR} := A_{BR} - L_{BL}U_{TR} \end{array}\right)$

Table 4.4: The five loop invariants for the *LU* factorization.

Given the description in Box 4.14, CLlCK finds three feasible sets of partitioning rules, which, in time, lead to the three PMEs listed in Table 4.5.

#	Partitioned Matrix Expression
1	$\left(\{X_L,Y_L\} := \Psi(A,B_{TL},C_L,D,E_{TL},F_L) \,\middle\vert\, \{X_R,Y_R\} := \Psi(A,B_{BR},C_R - Y_L B_{TR},D,E_{BR},F_R - Y_L E_{TR})\right)$
2	$\left(\begin{array}{c} \{X_T,Y_T\} := \Psi(A_{TL},B,C_T,D_{TL},E,F_T) \\ \hline \{X_B,Y_B\} := \Psi(A_{BR},B,C_B - A_{BL}X_T,D_{BR},E,F_B - D_{BL}X_T) \end{array}\right)$
3	$\left(\begin{array}{c\|c} \{X_{TL},Y_{TL}\} := \Psi(A_{TL},B_{TL},C_{TL},D_{TL},E_{TL},F_{TL}) & \{X_{TR},Y_{TR}\} := \Psi(A_{TL},B_{BR},C_{TR} - Y_{TL}B_{TR}, \\ & \quad D_{TL},E_{BR},F_{TR} - Y_{TL}E_{TR}) \\ \hline \{X_{BL},Y_{BL}\} := \Psi(A_{BR},B_{TL},C_{BL} - A_{BL}X_{TL}, & \{X_{BR},Y_{BR}\} := \Psi(A_{BR},B_{BR},C_{BR} - A_{BL}X_{TR} - Y_{BL}B_{TR}, \\ \quad D_{BR},E_{TL},F_{BL} - D_{BL}X_{TL}) & \quad D_{BR},E_{BR},F_{BR} - D_{BL}X_{TR} - Y_{BL}E_{TR}) \end{array}\right)$

Table 4.5: The three Partitioned Matrix Expressions for the coupled Sylvester equation.

We demonstrate the identification of loop invariants for the third PME. First,

CLlCK traverses the PME, one quadrant at a time, to decompose the assignments into tasks. The analysis starts from the top-left assignment; since the right-hand side consists of a function where all the input arguments are simple operands, the function is yielded as a single task:

1. $\{X_{TL}, Y_{TL}\} := \Psi(A_{TL}, B_{TL}, C_{TL}, D_{TL}, E_{TL}, F_{TL})$.

Next, the top-right assignment is inspected. In this case, two of the input arguments are not simple operands. Thus, CLlCK analyzes both expressions, $C_{TR} - Y_{TL}B_{TR}$ and $F_{TR} - Y_{TL}E_{TR}$, to identify the sequence of tasks. Both expressions match the pattern for the matrix product (GEMM). As a result, CLlCK returns the sequence

2. $C_{TR} := C_{TR} - Y_{TL}B_{TR}$
3. $F_{TR} := F_{TR} - Y_{TL}E_{TR}$
4. $\{X_{TR}, Y_{TR}\} := \Psi(A_{TL}, B_{BR}, C_{TR}, D_{TL}, E_{BR}, F_{TR})$

A similar situation occurs when studying the bottom-left assignment, in which CLlCK yields three more tasks.

5. $C_{BL} := C_{BL} - A_{BL}X_{TL}$
6. $F_{BL} := F_{BL} - D_{BL}X_{TL}$
7. $\{X_{BL}, Y_{BL}\} := \Psi(A_{BR}, B_{TL}, C_{BL}, D_{BR}, E_{TL}, F_{BL})$

Only the assignment in the bottom-right quadrant remains to be analyzed. CLlCK recognizes that two of the input arguments to the function are expressions. In contrast to the previous cases, the two expressions are decomposed into more than one task. For instance, the expression $C_{BR} - A_{BL}X_{TR} - Y_{BL}B_{TR}$ is decomposed into two matrix products: $C_{BR} - A_{BL}X_{TR}$ and $C_{BR} - Y_{BL}B_{TR}$. It is important to notice that both products are independent of one another, i.e., they can be performed in any order. CLlCK keeps track of this fact for a correct analysis of dependencies. In total, the analisys of the bottom-right assignment yields the following five tasks, two per complex input argument and the function itself:

8. $C_{BR} := C_{BR} - A_{BL}X_{TR}$
9. $C_{BR} := C_{BR} - Y_{BL}B_{TR}$
10. $F_{BR} := F_{BR} - D_{BL}X_{TR}$
11. $F_{BR} := F_{BR} - Y_{BL}E_{TR}$
12. $\{X_{BR}, Y_{BR}\} := \Psi(A_{BR}, B_{BR}, C_{BR}, D_{BR}, E_{BR}, F_{BR})$.

CLlCK proceeds with the inspection of the tasks for dependencies. The analysis commences from Task 1, whose outputs (X_{TL} and Y_{TL}) are inputs to Tasks 2, 3, 5 and 6. The four corresponding true dependencies are created. Next, Tasks 2 and 3 are inspected. Their outputs, C_{TR} and F_{TR}, are input to Task 4, hence enforcing two more true dependencies. The algorithm proceeds by analyzing Task 4. One of

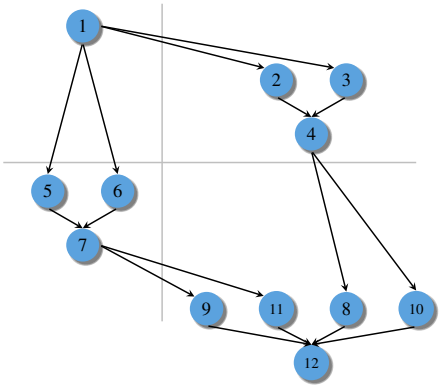


Figure 4.8: Graph of dependencies for the coupled Sylvester equation.

its output operands, X_{TR}, appears as an input argument of Tasks 8 and 10; two new dependencies arise. The study of Tasks 5, 6 and 7 is analogous to that of Tasks 2, 3, and 4. CLlCK finds true dependencies from Tasks 5 and 6 to Task 7, and from Task 7 to Tasks 9 and 11.

The analysis continues with the study of Tasks 8 and 9. Both tasks take as input and overwrite the quantity C_{BR}; however, as we pointed out earlier, they are independent from one another and can be computed in any order. Therefore, dependencies are created only from Tasks 8 and 9 to 12, which takes C_{BR} as input. The study of Tasks 10 and 11 is led by the same principle, originating the corresponding dependencies from both to Task 12. Finally, Task 12 is analyzed. Its output, $\{X_{BR}, Y_{BR}\}$, does not appear in any of the other tasks, thus no new dependencies are imposed. The final graph of dependencies is shown in Figure 4.8.

Once the graph is built, CLlCK executes Algorithm 4.2 and returns a list with the predicates that are candidates to becoming loop invariants. Then, the predicates are checked to establish their feasibility; the non-feasible ones are discarded. In the coupled Sylvester equation example, the system identifies 64 feasible loop invariants, which lead to 64 different algorithms that solve the equation. In Table 4.6 we list a subset of the returned loop invariants.

The large number of identified loop invariants and the corresponding algorithms, demonstrates the necessity for having a system that automates the process.

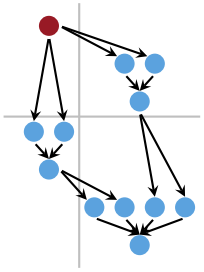
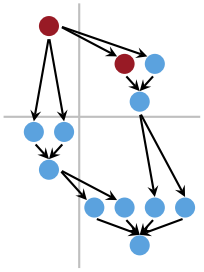
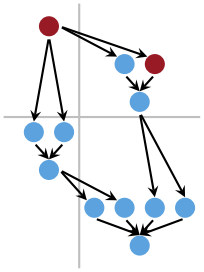
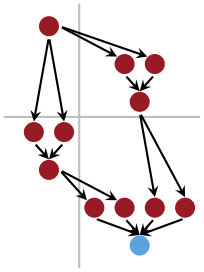
#	Subgraph	Loop invariant
1		$\left(\begin{array}{c\|c} \{X_{TL},Y_{TL}\}=\Psi(A_{TL},B_{TL},C_{TL},D_{TL},E_{TL},F_{TL}) & \neq \\ \hline \neq & \neq \end{array}\right)$
2		$\left(\begin{array}{c\|c} \{X_{TL},Y_{TL}\}=\Psi(A_{TL},B_{TL},C_{TL},D_{TL},E_{TL},F_{TL}) & X_{TR}=C_{TR}-Y_{TL}B_{TR} \\ \hline \neq & \neq \end{array}\right)$
3		$\left(\begin{array}{c\|c} \{X_{TL},Y_{TL}\}=\Psi(A_{TL},B_{TL},C_{TL},D_{TL},E_{TL},F_{TL}) & Y_{TR}=F_{TR}-Y_{TL}E_{TR} \\ \hline \neq & \neq \end{array}\right)$
⋮		⋮
64		$\left(\begin{array}{c\|c} \{X_{TL},Y_{TL}\}=\Psi(A_{TL},B_{TL},C_{TL}, D_{TL},E_{TL},F_{TL}) & \{X_{TR},Y_{TR}\}=\Psi(A_{TL},B_{BR}, C_{TR}-Y_{TL}B_{TR}, D_{TL},E_{BR}, F_{TR}-Y_{TL}E_{TR}) \\ \hline \{X_{BL},Y_{BL}\}=\Psi(A_{BR},B_{TL}, C_{BL}-A_{BL}X_{TL}, D_{BR},E_{TL}, F_{BL}-D_{BL}X_{TL}) & \{X_{BR},Y_{BR}\}=\{C_{BR}-A_{BL}X_{TR} -Y_{BL}B_{TR}, F_{BR}-D_{BL}X_{TR} -Y_{BL}E_{TR}\} \end{array}\right)$

Table 4.6: A subset of the 64 loop invariants for the coupled Sylvester equation.

4.5 Algorithm construction

We discuss now the final stage in the generation of algorithms, the Algorithm Construction. It is in this last stage where the loop body is derived, and the algorithms are finally built. The discussion centers around the template for a proof of correctness introduced in Section 4.1, which we reproduce here with further detail:

$\{P_{\text{pre}}\}$
Partition
$\{P_{\text{inv}}\}$
While G **do**
    $\{P_{\text{inv}} \wedge G\}$
    Repartition
    $\{P_{\text{before}} \equiv P_{\text{inv}}|_{\text{Repartition}}\}$
    Algorithm Updates
    $\{P_{\text{after}} \equiv P_{\text{inv}}|_{\text{Continue with}^{-1}}\}$
    Continue with
    $\{P_{\text{inv}}\}$
end
$\{P_{\text{inv}} \wedge \neg G\}$
$\{P_{\text{post}}\}$

The idea is to transform the template into an algorithm annotated with its proof of correctness by incrementally replacing the predicates in brackets, and by filling in the boldface labels with actual algorithm statements.

For each of the loop invariants found in the previous stage, one such template is filled in. Each loop invariant P_{inv} was annotated with the traversal of the operands, and the corresponding loop guard G. With this information, together with the input equation description (P_{pre} and P_{post}), the template can be partially filled in. As an example, in Box 4.7, we provide the partially filled in template for the third loop-invariant of the *LU* factorization (Table 4.4). We highlight in red the pieces that remain to be derived.

The construction of the algorithm completes in three steps, as depicted by Figure 4.9. The key is to find the updates that render the P_{inv} true at the end of the loop. First, the traversal of the operands is formalized via the Repartition and Continue with statements. The former exposes new parts of the matrices to be used in the updates; the latter combines these parts to ensure progress and termination of the loop. Then, the loop invariant P_{inv} is rewritten in terms of the repartitioned matrices, yielding the predicates P_{before} and P_{after}. Finally, the algorithm updates are derived so that the computation is taken from the state P_{before} to the state P_{after}

$\{$ `LowTri(L), UppTri(U), ...` $\}$
Partition $\left(\begin{array}{c\|c} *_{TL} & *_{TR} \\ \hline *_{BL} & *_{BR} \end{array}\right)$ **where** $*_{TL}$ is 0×0
$\{\left(\begin{array}{c\|c} \{L_{TL}, U_{TL}\} := LU(A_{TL}) & \neq \\ \hline L_{BL} := A_{BL}U_{TL}^{-1} & \neq \end{array}\right)\}$
While $size(A_{TL}) < size(A)$ **do**
$\{\left(\begin{array}{c\|c} \{L_{TL}, U_{TL}\} := LU(A_{TL}) & \neq \\ \hline L_{BL} := A_{BL}U_{TL}^{-1} & \neq \end{array}\right) \wedge (size(A_{TL}) < size(A))\}$
$\left(\begin{array}{c\|c} *_{TL} & *_{TR} \\ \hline *_{BL} & *_{BR} \end{array}\right) \longrightarrow ?$
$\{P_{\text{before}} \equiv P_{\text{inv}}\|_{\text{Repartition}}\}$
Algorithm Updates
$\{P_{\text{after}} \equiv P_{\text{inv}}\|_{\text{Continue with}^{-1}}\}$
$\left(\begin{array}{c\|c} *_{TL} & *_{TR} \\ \hline *_{BL} & *_{BR} \end{array}\right) \longleftarrow ?$
$\{\left(\begin{array}{c\|c} \{L_{TL}, U_{TL}\} := LU(A_{TL}) & \neq \\ \hline L_{BL} := A_{BL}U_{TL}^{-1} & \neq \end{array}\right)\}$
end
$\{\left(\begin{array}{c\|c} \{L_{TL}, U_{TL}\} := LU(A_{TL}) & \neq \\ \hline L_{BL} := A_{BL}U_{TL}^{-1} & \neq \end{array}\right) \wedge (A_{TL} = A)\}$
$\{LU = A\}$

Table 4.7: Partially filled in template for LU's third loop invariant (Table 4.4).

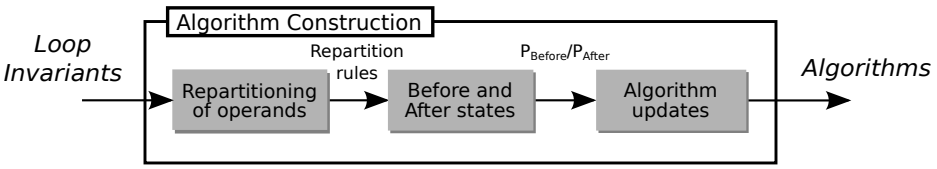


Figure 4.9: Steps for the construction of an algorithm from a given loop invariant.

and the proof of correctness is satisfied. In this section, we describe these steps, and detail how CLlCK automates them.

4.5.1 Repartitioning of the operands

The Repartition and Continue with statements encode how the algorithm marches through the operands, as determined by the loop invariant. In Box 4.15, we illustrate the statements for a generic matrix A and a lower triangular matrix L; A, initially partitioned in 1×2 parts, is traversed from left to right, while L, partitioned in 2×2 quadrants, is traversed from bottom-right to top-left. The thick lines have a meaning: Initially empty, A_L and L_{BR} grow larger at each iteration; when the loop

terminates, they equal the full operand.

$$\left(\begin{array}{c|c} A_L & A_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_0 & A_1 & A_2 \end{array}\right) \qquad \left(\begin{array}{c|c} A_L & A_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_0 & A_1 & A_2 \end{array}\right)$$

(a) Repartition (left) and Continue with (right) statements for a generic matrix A traversed from left to right.

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right) \qquad \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$$

(b) Repartition (left) and Continue with (right) statements for a lower triangular matrix L traversed from bottom-right to top-left.

Box 4.15: Two examples of Repartition and Continue with statements.

Given the traversal for an operand, CLlCK generates the corresponding Repartition and Continue with statements. Similarly to the inheritance of properties during the partitioning of the operands in the PME Generation (Section 4.3), the repartitioning also activates the propagation of properties. For instance, in the example in Box 4.15b, L_{00}, L_{11}, and L_{22} are lower triangular, and L_{01}, L_{02}, and L_{12} are the zero matrix.

The statements are encoded as lists of rewrite rules, which will be used to perform the subsequent textual substitution of the loop invariant. For instance, for matrix L in Box 4.15b, CLlCK produces the following two lists of rules.

- *Repartition rules:*

$$\left\{ L_{TL} \rightarrow \left(\begin{array}{c|c} L_{00} & 0 \\ \hline L_{10} & L_{11} \end{array}\right), L_{BL} \rightarrow \left(\begin{array}{c|c} L_{20} & L_{21} \end{array}\right), L_{BR} \rightarrow \left(L_{22}\right) \right\}$$

- *Continue with rules:*

$$\left\{ L_{TL} \rightarrow \left(L_{00}\right), L_{BL} \rightarrow \left(\begin{array}{c} L_{10} \\ \hline L_{20} \end{array}\right), L_{BR} \rightarrow \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right) \right\}$$

For each of the operands in the target operation, CLlCK generates such lists of rules and proceeds with the construction of the P_{before} and P_{after} predicates.

4.5.2 Predicates P_{before} and P_{after}

The P_{before} and P_{after} predicates express the loop invariant in terms of the repartitioned operands before and after the algorithm updates. These predicates are constructed in two steps: First, the loop invariant is rewritten, replacing the partitioned operands by their repartitioned counterparts; then, the resulting expressions are *flattened out*. The success of this process is dependent on CLlCK's ability to learn the operation's PMEs.

To illustrate the generation of P_{before} and P_{after}, we make use of the following loop invariant for the triangular Sylvester equation:[2]

$$\left(\begin{array}{c|c} X_{TL} = C_{TL} - A_{TR}X_{BL} & \neq \\ \hline X_{BL} = \Omega(A_{BR}, B_{TL}, C_{BL}) & X_{BR} = \Omega(A_{BR}, B_{BR}, C_{BR} - X_{BL}B_{TR}) \end{array}\right). \tag{4.6}$$

All four operands —X, A, B, and C— are initially partitioned in 2×2 quadrants; X and C are traversed from bottom-left to top-right, A is traversed from bottom-right to top-left, and B is traversed from top-left to bottom-right. The rewrite rules corresponding to the Repartition and Continue statements yielded in the previous step are provided in Boxes 4.16 and 4.17, respectively.

$$\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c} \left(\begin{array}{c} X_{00} \\ \hline X_{10} \end{array}\right) & \left(\begin{array}{c|c} X_{01} & X_{02} \\ \hline X_{11} & X_{12} \end{array}\right) \\ \hline (X_{20}) & \left(\begin{array}{c|c} X_{21} & X_{22} \end{array}\right) \end{array}\right) \quad \text{(a)}$$

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c} \left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline 0 & A_{11} \end{array}\right) & \left(\begin{array}{c} A_{02} \\ \hline A_{12} \end{array}\right) \\ \hline \left(\begin{array}{c|c} 0 & 0 \end{array}\right) & (A_{22}) \end{array}\right) \quad \text{(b)}$$

$$\left(\begin{array}{c|c} B_{TL} & B_{TR} \\ \hline 0 & B_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c} (B_{00}) & \left(\begin{array}{c|c} B_{01} & B_{02} \end{array}\right) \\ \hline \left(\begin{array}{c} 0 \\ \hline 0 \end{array}\right) & \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline 0 & B_{22} \end{array}\right) \end{array}\right) \quad \text{(c)}$$

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c} \left(\begin{array}{c} C_{00} \\ \hline C_{10} \end{array}\right) & \left(\begin{array}{c|c} C_{01} & C_{02} \\ \hline C_{11} & C_{12} \end{array}\right) \\ \hline (C_{20}) & \left(\begin{array}{c|c} C_{21} & C_{22} \end{array}\right) \end{array}\right) \quad \text{(d)}$$

Box 4.16: *Repartition rules*. Repartitioning towards the generation of P_{before}.

$$\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c} \left(\begin{array}{c|c} X_{00} & X_{01} \\ \hline X_{10} & X_{11} \\ \hline X_{20} & X_{21} \end{array}\right) & \left(\begin{array}{c} X_{02} \\ \hline X_{12} \\ \hline X_{22} \end{array}\right) \end{array}\right) \quad \text{(a)}$$

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c} (A_{00}) & \left(\begin{array}{c|c} A_{01} & A_{02} \end{array}\right) \\ \hline \left(\begin{array}{c} 0 \\ \hline 0 \end{array}\right) & \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline 0 & A_{22} \end{array}\right) \end{array}\right) \quad \text{(b)}$$

$$\left(\begin{array}{c|c} B_{TL} & B_{TR} \\ \hline 0 & B_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c} \left(\begin{array}{c|c} B_{00} & B_{01} \\ \hline 0 & B_{11} \end{array}\right) & \left(\begin{array}{c} B_{02} \\ \hline B_{12} \end{array}\right) \\ \hline \left(\begin{array}{c|c} 0 & 0 \end{array}\right) & (B_{22}) \end{array}\right) \quad \text{(c)}$$

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c} \left(\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \\ \hline C_{20} & C_{21} \end{array}\right) & \left(\begin{array}{c} C_{02} \\ \hline C_{12} \\ \hline C_{22} \end{array}\right) \end{array}\right) \quad \text{(d)}$$

Box 4.17: *Continue with rules*. Repartitioning towards the generation of P_{after}.

The construction of P_{before} commences with the application of the repartition

[2] This loop invariant is obtained from the third PME in Table 4.3, Section 4.3.3.

rules (Box 4.16) to the loop invariant; the expression is rewritten as

$$\left(\begin{array}{c|c} \left(\begin{array}{c} X_{00} \\ \hline X_{10} \end{array}\right) := \left(\begin{array}{c} C_{00} \\ \hline C_{10} \end{array}\right) - \left(\begin{array}{c} A_{02} \\ \hline A_{12} \end{array}\right)(X_{20}) & \left(\begin{array}{c|c} X_{01} & X_{02} \\ \hline X_{11} & X_{12} \end{array}\right) := \left(\begin{array}{c|c} 0 & 0 \\ \hline 0 & 0 \end{array}\right) \\ \hline (X_{20}) := \Omega((A_{22}),(B_{00}),(C_{20})) & \left(X_{21} \mid X_{22}\right) := \Omega\left((A_{22}), \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline 0 & B_{22} \end{array}\right), \left(C_{21} \mid C_{22}\right) - (X_{20})\left(B_{01} \mid B_{02}\right)\right) \end{array}\right). \tag{4.7}$$

Next, the assignments in each of the four quadrants must be simplified. The right-hand sides consist of either explicit algebraic operations, as in the top-left quadrant, or an implicit function with partitioned arguments, as in the bottom-right quadrant. In the first case, CLlCK applies basic built-in matrix algebra knowledge. For instance, the expression

$$\left(\begin{array}{c} X_{00} \\ \hline X_{10} \end{array}\right) := \left(\begin{array}{c} C_{00} \\ \hline C_{10} \end{array}\right) - \left(\begin{array}{c} A_{02} \\ \hline A_{12} \end{array}\right)(X_{20})$$

is multiplied out and the assignment is distributed, resulting in

$$\left(\begin{array}{c} X_{00} := C_{00} - A_{02}X_{20} \\ \hline X_{10} := C_{10} - A_{12}X_{20} \end{array}\right)$$

The second case, instead, requires a deeper understanding of the FLAME methodology. In the bottom-right quadrant of Equation (4.7), one finds a recursive call to Sylvester with partitioned operands. At first sight, simplifying the expression is far from straightforward; fortunately, as CLlCK generated the PMEs for Sylvester, it learned a number of rules (Box 4.12) to flatten out such a complicated expression. Concretely, the rule

$$\left(\begin{array}{c|c} X_L & X_R \end{array}\right) := \Omega\left((A), \left(\begin{array}{c|c} B_{TL} & B_{TR} \\ \hline 0 & B_{BR} \end{array}\right), \left(\begin{array}{c|c} C_L & C_R \end{array}\right)\right) \longrightarrow$$

$$\left(\begin{array}{c|c} X_L := \Omega(A, B_{TL}, C_L) & X_R := \Omega(A, B_{BR}, C_R - X_L B_{TR}) \end{array}\right)$$

corresponding to the first PME, enables the rewrite of

$$\left(\begin{array}{c|c} X_{21} & X_{22} \end{array}\right) := \Omega\left((A_{22}), \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline 0 & B_{22} \end{array}\right), \left(\begin{array}{c|c} C_{21} & C_{22} \end{array}\right) - (X_{20})\left(\begin{array}{c|c} B_{01} & B_{02} \end{array}\right)\right)$$

as

$$\left(\begin{array}{c|c} X_{21} := \Omega(A_{22}, B_{11}, C_{21} - X_{20}B_{01}) & X_{22} := \Omega(A_{22}, B_{22}, C_{22} - X_{20}B_{02} - X_{21}B_{12}) \end{array}\right).$$

In fact, the generation of P_{before} and P_{after} for the loop invariant under consideration requires all three previously learned PMEs. We illustrate this in Figure 4.10: Above the *Algorithm Updates*, we provide the predicate P_{before} prior and after the flattening; the flattening rule from PME 1 is used in the bottom-right quadrant. Below the updates, we find the predicate P_{after} also prior and after the flattening; rules from PMEs 2 and 3 are required to flatten the expressions in the bottom-right and bottom-left quadrants, respectively.

$$P_{\text{before}} \quad \left(\begin{array}{c|c} \left(\begin{array}{c} X_{00} \\ \hline X_{10} \end{array}\right) = \left(\begin{array}{c} C_{00} \\ \hline C_{10} \end{array}\right) - \left(\begin{array}{c} A_{02} \\ \hline A_{12} \end{array}\right)(X_{20}) & \left(\begin{array}{c|c} X_{01} & X_{02} \\ \hline X_{11} & X_{12} \end{array}\right) = \left(\begin{array}{c|c} 0 & 0 \\ \hline 0 & 0 \end{array}\right) \\ \hline (X_{20}) = \Omega(A_{22}, B_{00}, C_{20}) & \left(\begin{array}{c|c} X_{21} & X_{22} \end{array}\right) = \Omega\left((A_{22}), \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline 0 & B_{22} \end{array}\right), \left(\begin{array}{c|c} C_{21} & C_{22} \end{array}\right) - (X_{20})\left(\begin{array}{c|c} B_{01} & B_{02} \end{array}\right)\right) \end{array}\right).$$

$$\Downarrow \text{Flattening}$$

$$\left(\begin{array}{c|c|c} X_{00} = C_{00} - A_{02}X_{20} & X_{01} = 0 & X_{02} = 0 \\ \hline X_{10} = C_{10} - A_{12}X_{20} & X_{11} = 0 & X_{12} = 0 \\ \hline X_{20} = \Omega(A_{22}, B_{00}, C_{20}) & X_{21} = \Omega(A_{22}, B_{11}, C_{21} - X_{20}B_{01}) & X_{22} = \Omega(A_{22}, B_{22}, C_{22} - X_{20}B_{02} - X_{21}B_{12}) \end{array}\right)$$

Algorithm Updates

$$\left(\begin{array}{c|c|c} X_{00} = C_{00} - A_{01}X_{10} - A_{02}X_{20} & X_{01} = C_{01} - A_{01}X_{11} - A_{02}X_{21} & X_{02} = 0 \\ \hline X_{10} = \Omega(A_{11}, B_{00}, C_{10} - A_{12}X_{20}) & X_{11} = \Omega(A_{11}, B_{11}, C_{11} - A_{12}X_{21} - X_{10}B_{01}) & X_{12} = \Omega(A_{11}, B_{22}, C_{12} - X_{10}B_{02} - X_{11}B_{12} - A_{12}X_{22}) \\ \hline X_{20} = \Omega(A_{22}, B_{00}, C_{20}) & X_{21} = \Omega(A_{22}, B_{11}, C_{21} - X_{20}B_{01}) & X_{22} = \Omega(A_{22}, B_{22}, C_{22} - X_{20}B_{02} - X_{21}B_{12}) \end{array}\right)$$

$$\Uparrow \text{Flattening}$$

$$P_{\text{after}} \quad \left(\begin{array}{c|c} \left(\begin{array}{c|c} X_{00} & X_{01} \end{array}\right) = \left(\begin{array}{c|c} C_{00} & C_{01} \end{array}\right) - \left(\begin{array}{c|c} A_{01} & A_{02} \end{array}\right)\left(\begin{array}{c|c} X_{10} & X_{11} \\ \hline X_{20} & X_{21} \end{array}\right) & X_{02} = 0 \\ \hline \left(\begin{array}{c|c} X_{10} & X_{11} \\ \hline X_{20} & X_{21} \end{array}\right) = \Omega\left(\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline 0 & A_{22} \end{array}\right), \left(\begin{array}{c|c} B_{00} & B_{01} \\ \hline 0 & B_{11} \end{array}\right), \left(\begin{array}{c|c} C_{10} & C_{11} \\ \hline C_{20} & C_{21} \end{array}\right)\right) & \left(\begin{array}{c} X_{12} \\ \hline X_{22} \end{array}\right) = \Omega\left(\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right), (B_{22}), \left(\begin{array}{c} C_{12} \\ \hline C_{22} \end{array}\right) - \left(\begin{array}{c|c} X_{10} & X_{11} \\ \hline X_{20} & X_{21} \end{array}\right)\left(\begin{array}{c} B_{02} \\ \hline B_{12} \end{array}\right)\right) \end{array}\right)$$

Figure 4.10: P_{before} and P_{after} for Sylvester's loop invariant (4.6).

4.5.3 Finding the updates

Finding the algorithm updates is equivalent to finding the computation that takes the loop invariant from the state in P_{before} to the state in P_{after}. Intuitively, the updates are identified by comparing the two states. In practice, the comparison heavily relies on pattern matching and expression rewriting. In this section, we expose, via the Sylvester and Cholesky examples, how CLlCK derives the updates.

In our first example, we continue the Sylvester case study from the previous section. The comparison of the P_{before} and P_{after} predicates (Figure 4.10) is carried out quadrant by quadrant. The quantities X_{02}, X_{20}, X_{21}, and X_{22} hold the same value before and after the updates; hence, for these quadrants, no computation is required. X_{00} and X_{10}, instead, are partially computed, i.e., they hold a value distinct from *Null* (0) at the beginning of the iteration (P_{before}), but require further computation to render the loop invariant true at the end of the iteration (P_{after}). CLlCK processes both quadrants in a similar fashion; here we discuss the update for X_{10}. The value for X_{10} in P_{before} is

$$X_{10} := C_{10} - A_{12}X_{20} \tag{4.8}$$

while the required value at the end of the loop is

$$X_{10} := \Omega(A_{11}, B_{00}, C_{10} - A_{12}X_{20}). \tag{4.9}$$

An inspection of both expressions quickly reveals the fact that the quantity stored in P_{before}, (4.8), equals the quantity required as third argument for Ω in P_{after}, (4.9). Accordingly, the sought-after update is

$$X_{10} := \Omega(A_{11}, B_{00}, X_{10}),$$

where the previously computed X_{10} is used as third argument.

This intuition is formalized via rewrite rules: CLlCK takes the assignment in P_{before}, and creates a rule of the form *right-hand side* $\rightarrow$ *left-hand side*. The rule is used to rewrite the assignment in P_{after}:

$$X_{10} := \Omega(A_{11}, B_{00}, C_{10} - A_{12}X_{20}) \;\; /. \;\; C_{10} - A_{12}X_{20} \rightarrow X_{10},$$

obtaining the update $X_{10} := \Omega(A_{11}, B_{00}, X_{10})$.

Deriving the updates in the remaining quadrants —X_{01}, X_{11}, and X_{12}— is straightforward. Since no computation is stored in P_{before}, the required update equals the right-hand side of the expression in P_{after}. Combining all quadrants, the final set of updates for the Sylvester example is

$$\begin{aligned}
X_{10} &:= \Omega(A_{11}, B_{00}, X_{10}) \\
X_{11} &:= \Omega(A_{11}, B_{11}, C_{11} - A_{12}X_{21} - X_{10}B_{01}) \\
X_{12} &:= \Omega(A_{11}, B_{22}, C_{12} - X_{10}B_{02} - X_{11}B_{12} - A_{12}X_{22}) \\
X_{00} &:= X_{00} - A_{01}X_{10} \\
X_{01} &:= C_{01} - A_{01}X_{11} - A_{02}X_{21}.
\end{aligned}$$

As a second example, we choose the following loop invariant for the Cholesky factorization:

$$\left(\begin{array}{c|c} L_{TL} := \Gamma(A_{TL}) & \star \\ \hline L_{BL} := A_{BL}L_{TL}^{-T} & L_{BR} := A_{BR} - L_{BL}L_{BL}^{T} \end{array}\right). \tag{4.10}$$

The P_{before} and P_{after} predicates are given in Box 4.18. In this case, only quadrants "1,1", "2,1", and "2,2" differ between states and need to be updated. To derive the updates for the operands L_{11} and L_{22}, it suffices to build and apply rewrite rules as previously described; the yielded updates are

$$\begin{aligned} L_{11} &:= \Gamma(L_{11}) \\ L_{22} &:= L_{22} - L_{21}L_{21}^{T}. \end{aligned}$$

However, often times, a direct replacement is not sufficient to find the updates. This is the case, for instance, of L_{21}: The expression

$$L_{21} := A_{21} - L_{20}L_{10}^{T} \tag{4.11}$$

is not directly found in the P_{after} counterpart,

$$L_{21} := A_{21}L_{11}^{-T} - A_{20}L_{00}^{-T}L_{10}^{T}L_{11}^{-T}. \tag{4.12}$$

The reason why this happens is that the assignments in these two predicates are not written in any sort of canonical form. In such a situation, a human inspects the other quadrants for already computed subexpressions that may be used to rewrite (4.11) or (4.12). For instance, the expression $A_{20}L_{00}^{-T}$ has already been computed and stored in L_{20}; therefore, $A_{20}L_{00}^{-T}$ may be replaced in the after state (4.12) by L_{20}, resulting in

$$L_{21} := A_{21}L_{11}^{-T} - L_{20}L_{10}^{T}L_{11}^{-T}.$$

Now, the right-hand side of the before $(A_{21} - L_{20}L_{10}^{T})$ is exposed in that of the after; the simple replacement

$$L_{21} := A_{21}L_{11}^{-T} - L_{20}L_{10}^{T}L_{11}^{-T} \;\; /. \;\; A_{21} - L_{20}L_{10}^{T} \to L_{20},$$

yields the required update: $L_{21} := L_{21}L_{11}^{-T}$.

CLICK makes this search systematic by first rewriting both assignments so that redundant subexpressions are eliminated, and then applying the direct replacement of the before in the after. To this end, CLICK creates the two lists of rewrite rules shown in Box 4.19; one list per predicate, one rule per assignment. The rules are then applied to the assignments of the corresponding predicate. While the application of the rules in Box 4.19a to the before state does not modify L_{21}, the application of the rules in Box 4.19b to the after state, results in

$$L_{21} := A_{21}L_{11}^{-T} - L_{20}L_{10}^{T}L_{11}^{-T}. \tag{4.13}$$

Now, the right-hand side of L_{21}'s before state may be directly replaced in (4.13), yielding the exact same update as derived by hand.

$$
\begin{array}{ll}
P_{\text{before}} & \left(\begin{array}{c|c|c}
L_{00} := \Gamma(A_{00}) & \star & \star \\ \hline
L_{10} := A_{10}L_{00}^{-T} & L_{11} := A_{11} - L_{10}L_{10}^{T} & \star \\ \hline
L_{20} := A_{20}L_{00}^{-T} & L_{21} := A_{21} - L_{20}L_{10}^{T} & L_{22} := A_{22} - L_{20}L_{20}^{T}
\end{array}\right) \\ \hline
\textit{Updates} & \begin{array}{l} L_{11} := \Gamma(L_{11}) \\ L_{21} := L_{21}L_{11}^{-T} \\ L_{22} := L_{22} - L_{21}L_{21}^{T}. \end{array} \\ \hline
P_{\text{after}} & \left(\begin{array}{c|c|c}
L_{00} := \Gamma(A_{00}) & \star & \star \\ \hline
L_{10} := A_{10}L_{00}^{-T} & L_{11} := \Gamma(A_{11} - L_{10}L_{10}^{T}) & \star \\ \hline
L_{20} := A_{20}L_{00}^{-T} & L_{21} := A_{21}L_{11}^{-T} - A_{20}L_{00}^{-T}L_{10}^{T}L_{11}^{-T} & L_{22} := A_{22} - L_{20}L_{20}^{T} - L_{21}L_{21}^{T}
\end{array}\right)
\end{array}
$$

Box 4.18: Predicates P_{before} and P_{after}, and algorithm updates for Cholesky's loop invariant (4.10).

$$
\begin{aligned}
\Gamma(A_{00}) &\rightarrow L_{00} \\
A_{10}L_{00}^{-T} &\rightarrow L_{10} \\
A_{11} - L_{10}L_{10}^{T} &\rightarrow L_{11} \\
A_{20}L_{00}^{-T} &\rightarrow L_{20} \\
A_{21} - L_{20}L_{10}^{T} &\rightarrow L_{21} \\
A_{22} - L_{20}L_{20}^{T} &\rightarrow L_{22}
\end{aligned}
$$

(a) Rules created to rewrite the P_{before} assignments.

$$
\begin{aligned}
\Gamma(A_{00}) &\rightarrow L_{00} \\
A_{10}L_{00}^{-T} &\rightarrow L_{10} \\
\Gamma(A_{11} - L_{10}L_{10}^{T}) &\rightarrow L_{11} \\
A_{20}L_{00}^{-T} &\rightarrow L_{20} \\
A_{21}L_{11}^{-T} - A_{20}L_{00}^{-T}L_{10}^{T}L_{11}^{-T} &\rightarrow L_{21} \\
A_{22} - L_{20}L_{20}^{T} - L_{21}L_{21}^{T} &\rightarrow L_{22}
\end{aligned}
$$

(b) Rules created to rewrite the P_{after} assignments.

Box 4.19: Rules to rewrite the P_{before} and P_{after} predicates.

4.5.4 The final algorithms and routines

By repeating the process for every loop invariant obtained in the second stage (*Loop Invariant Identification*), CLlCK constructs a family of algorithms (one per loop invariant). In Figure 4.11, we present the algorithm corresponding to Sylvester's loop invariant (4.6), the one discussed in Sections 4.5.2 and 4.5.3. On the left side, we reproduce the blocked variant of the algorithm; the recursive calls to Sylvester may be computed by the unblocked variant (on the right side) which is easily obtained by setting the block size b to 1.

The FLAME project offers a number of application programming interfaces (APIs) that simplify the translation of algorithms into routines that closely resemble the algorithmic notation. CLlCK incorporates a C code generator that makes use of the FLAME/C API for this programming language; the implementation generated for the blocked algorithm in Figure 4.11 (left) is displayed in Routine 4.3.

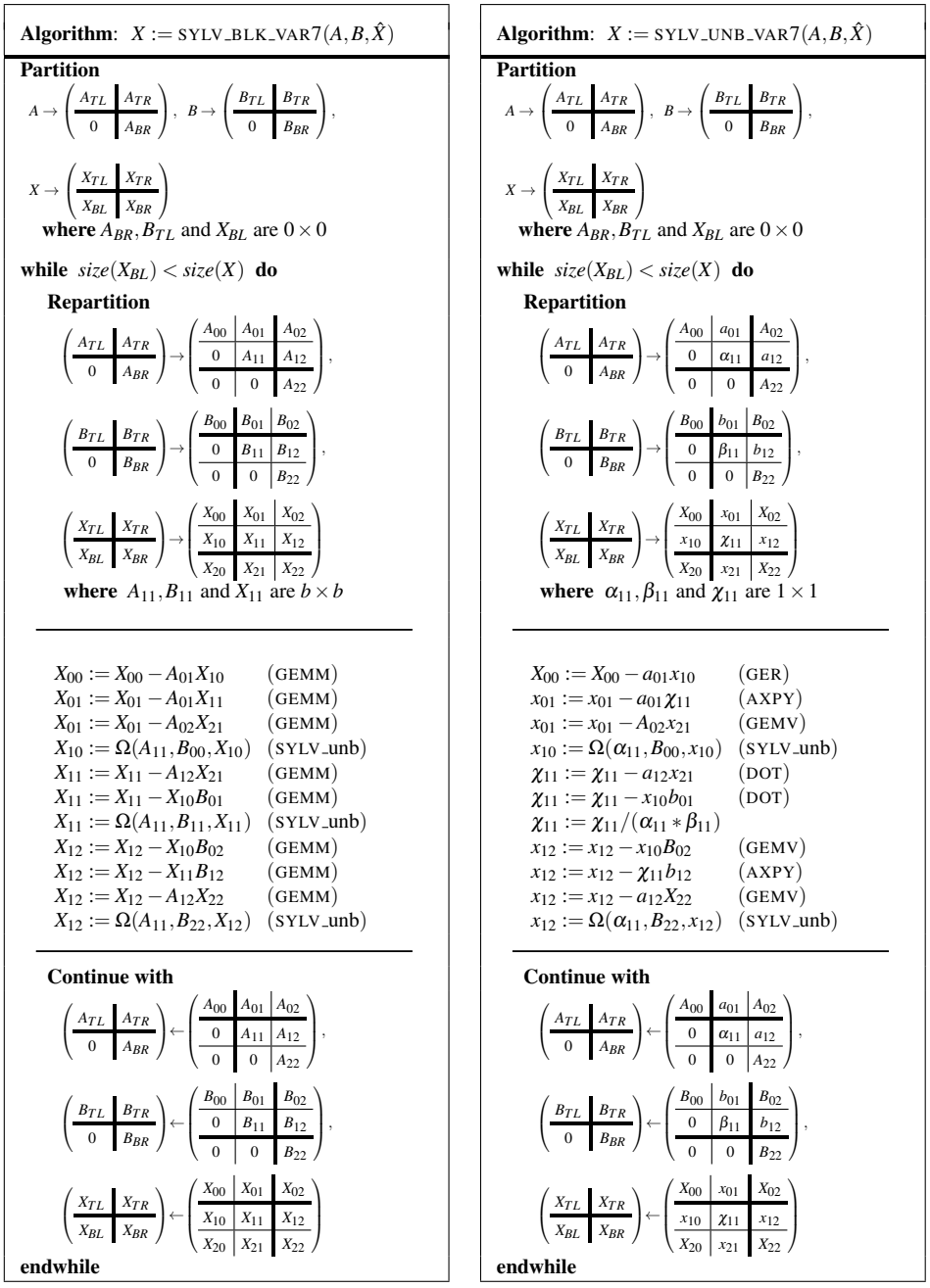


Figure 4.11: Blocked (left) and unblocked (right) versions of Sylvester's variant 7 (out of 16). C is overwritten with the solution X. In the unblocked version, greek, lowercase and uppercase letters are used, respectively, for scalars, vectors and matrices.

Routine 4.3: FLAME/C code for Sylvester's blocked variant 7 as generated by CLlCK.

```
void sylv_blk_var7( FLA_Obj A, FLA_Obj B, FLA_Obj X, int nb )
{
  FLA_Obj ATL, ATR, ABL, ABR, A00, A01, A02, A10, A11, A12, A20, A21, A22;
  FLA_Obj BTL, BTR, BBL, BBR, B00, B01, B02, B10, B11, B12, B20, B21, B22;
  FLA_Obj XTL, XTR, XBL, XBR, X00, X01, X02, X10, X11, X12, X20, X21, X22;

  FLA_Part_2x2( A, &ATL, &ATR,
                   &ABL, &ABR, 0, 0, FLA_BR );
  FLA_Part_2x2( B, &BTL, &BTR,
                   &BBL, &BBR, 0, 0, FLA_TL );
  FLA_Part_2x2( X, &XTL, &XTR,
                   &XBL, &XBR, 0, 0, FLA_BL );

  while ( FLA_Obj_length( XBL ) < FLA_Obj_length( X ) ||
          FLA_Obj_width( XBL ) < FLA_Obj_width( X ) )
  {
    FLA_Repart_2x2_to_3x3( XTL, XTR, &X00, &X01, &X02,
                                     &X10, &X11, &X12,
                           XBL, XBR, &X20, &X21, &X22, nb, nb, FLA_TR );
    [...]

    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A02, X21, FLA_ONE, X01 );
    FLA_sylv_unb(A11, B00, X10);
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A01, X10, FLA_ONE, X00 );
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, X10, B01, FLA_ONE, X11 );
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A12, X21, FLA_ONE, X11 );
    FLA_sylv_unb(A11, B11, X11);
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A01, X11, FLA_ONE, X01 );
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, X11, B12, FLA_ONE, X12 );
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, X10, B02, FLA_ONE, X12 );
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A12, X22, FLA_ONE, X12 );
    FLA_sylv_unb(A11, B22, X12);

    [...]
    FLA_Cont_with_3x3_to_2x2( &XTL, &XTR, X00, X01, X02,
                                          X10, X11, X12,
                              &XBL, &XBR, X20, X21, X22, FLA_BL );
  }
}
```

4.6 Towards a one-click code generation

The ultimate goal of the FLAME project in terms of automation is the development of a system that takes as input a high-level description of a target operation, and returns a family of algorithms and routines to compute the operation. From Bientinesi's dissertation [7]:

> *"Ultimately, one should be able to visit a website, fill in a form with information about the operation to be performed, choose a programming language, click the 'submit' button, and receive a library of routines that compute the operation."*

We made remarkable progress in this direction. First, we exposed in depth all the requirements to build such a system and fully automated the process; then we developed a user-friendly web interface where the user is freed from every low level detail. Figures 4.12 and 4.13 contain two screenshots of the web interface, corresponding to the triangular Sylvester equation used as example throughout this chapter. For a comparison, we recall the formal definition of the operation:

$$X := \Omega(A,B,C) \equiv \begin{cases} P_{\text{pre}} : \{\texttt{Input}(A) \wedge \texttt{UpperTriangular}(A) \wedge \\ \qquad\quad \texttt{Input}(B) \wedge \texttt{UpperTriangular}(B) \wedge \\ \qquad\quad \texttt{Input}(C) \wedge \texttt{Output}(X)\} \\ \\ P_{\text{post}} : \{AX + XB = C\}. \end{cases}$$

The first figure shows how the operation is input by the user; as the reader can appreciate, the formal description and the input to the interface match perfectly.

The second figure provides the output generated by the tool right after clicking the *Ok* button. It corresponds to the loop invariant number 7 from the third PME, i.e., the example used in Sections 4.5.2 and 4.5.3. On the left-hand panel we find the PME and the loop invariant; on the right-hand panel we see the generated algorithm. We emphasize that, in contrast to the several days that it would take by hand, CLlCK generated all 20 algorithms in only a few seconds.

4.7 Scope and limitations

Given the mathematical definition of a target operation in terms of the predicates Precondition and Postcondition, CLlCK produces a family of both algorithms and routines that compute it. CLlCK has been applied to a broad set of linear algebra operations. We list a few examples:

- Vector-vector, matrix-vector, and matrix-matrix products (e.g., BLAS operations).
- Matrix factorizations, such as LU and Cholesky.

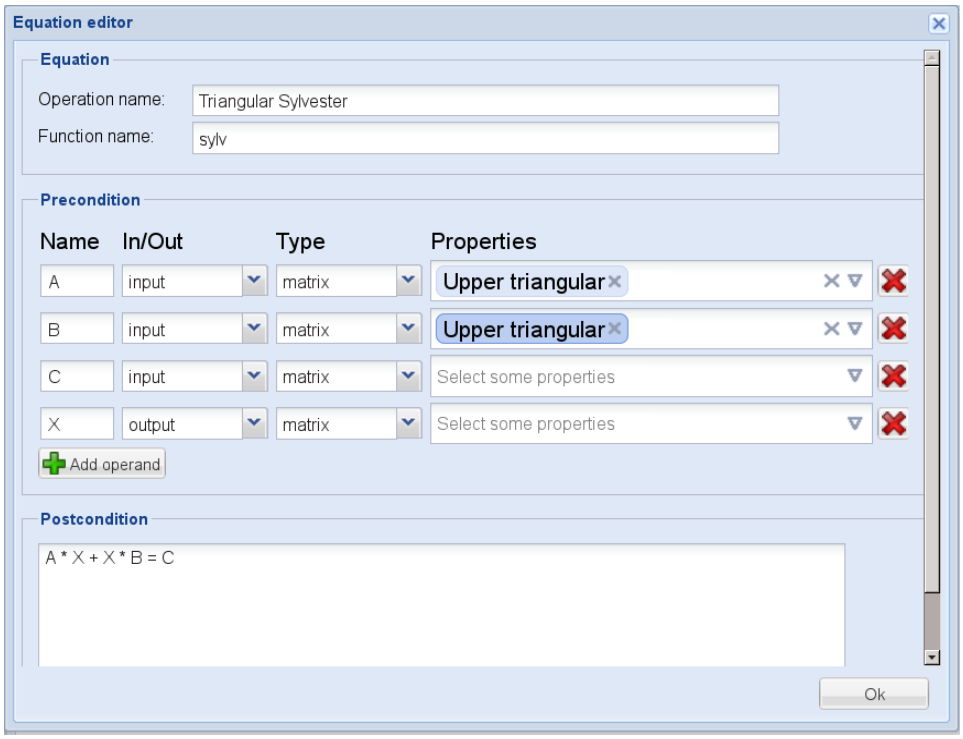


Figure 4.12: User-friendly web form to easily input the description of a target equation. In the example, we type the description of the Sylvester equation.

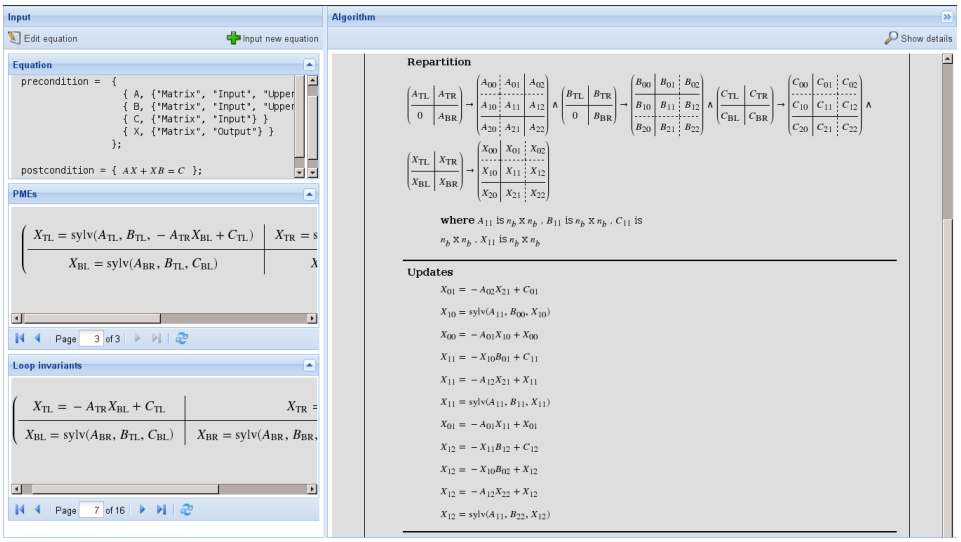


Figure 4.13: Output from CLlCK's web interface for the input in Figure 4.12. On the left panel, the interface shows the equation itself, its PMEs, and its loop invariants; on the right panel, the interface displays the algorithms.

- Inversion of matrices.
- Operations arising in control theory, such as Lyapunov and Sylvester equations.

However, while it has been established that the information encoded in the PME suffices to generate algorithms [7], a precise characterization of the scope of the methodology, i.e., the class of operations that admit a PME, is still missing.

Beyond the scope of the methodology, CLlCK presents a number of limitations similar to those discussed in Section 2.8 for CLAK, i.e, the lack of: 1) a module to automatically select the best algorithms, 2) code generators for multiple programming languages and programming paradigms, and 3) a mechanism to analyze the stability of the produced algorithms. In this case, instead, promising work from Bientinesi et al. [11] proposes an extension to the FLAME methodology for the systematic stability analysis of the generated algorithms. While still far fetched, this extension opens up the possibility for the future development of a module for the automatic stability analysis of CLlCK-generated algorithms.

4.8 Summary

We presented CLlCK, a prototype compiler for the automatic generation of loop-based linear algebra algorithms. From the sole mathematical description of a target equation, CLlCK is capable of generating families of algorithms that solve it. To this end, CLlCK adopts the FLAME methodology; the application of the methodology is divided in three stages: First, all PMEs for the target operation are generated; then, for each PME, multiple loop invariants are identified; finally, each loop invariant is used to build a provably correct algorithm. This chapter expands upon our work published in [21, 22].

The list of contributions made in this chapter follows.

- Minimum knowledge. For a given equation, we characterize the minimum knowledge required to automatically generate algorithms that solve it. This is the equation itself together with the properties of its operands.
- Full automation. We fully automate the generation of algorithms from the sole mathematical description of the operation. Previous work [7] required a PME and a loop invariant (both manually derived) as input, and the approach to automatically find the algorithm updates was limited.
- Feasibility. Several times this project has been deemed unfeasible. This chapter should serve as a precise reference on how to automate the process, and remove the skepticism.

For a wide class of linear algebra operations, the developers are now relieved from tedious, often unmanageable, symbolic manipulation, and only one CLlCK separates them from the sought-after algorithms.

Chapter 5

CL1CK: High-Performance Specialized Kernels

In the previous chapter, we demonstrated how CL1CK automates the application of the FLAME methodology by means of multiple standard operations, such as the LU and Cholesky factorizations. While the application of CL1CK to these operations shows the potential of the compiler, routines to compute them are already available from traditional libraries; in fact, most of the algorithms included in libFLAME [62] were derived using this methodology. In this chapter, instead, we concentrate on demonstrating the broad applicability of CL1CK by generating customized kernels for building blocks not supported by standard numerical libraries.

When developing application libraries, it is not uncommon to require kernels for building blocks that are closely related but not supported by libraries like BLAS or LAPACK. The situation arises so often that extensions to traditional libraries are regularly proposed [16]; unfortunately, the inclusion of every possible kernel arising in applications is unfeasible. While it may be possible to emulate the required kernels via a mapping onto two or more available kernels, this approach typically affects both routine's performance and developer's productivity; alternatively, efficient customized kernels may be produced on demand. We illustrate this issue by means of two example kernels arising in the context of algorithmic differentiation.

Consider a program that solves a linear system of equations $AX = B$, with symmetric positive definite coefficient matrix A, and multiple right-hand sides B; pseudocode for such a program follows. First, A is factored through a Cholesky factorization; then, two triangular linear systems are solved to compute the unknown X.

$$LL^T = A \qquad \text{(CHOL)}$$
$$LY = B \qquad \text{(TRSM)}$$
$$L^T X = Y \qquad \text{(TRSM)}$$

When interested in the derivative of this program, e.g., for a sensitivity analysis,

one must compute the following sequence of derivative operations,

$$\begin{aligned}
\frac{dL}{dv}\, L^T + L\, \frac{dL^T}{dv} &= \frac{dA}{dv} && \text{(gCHOL)}\\
\frac{dL}{dv}\, Y + L\, \frac{dY}{dv} &= \frac{dB}{dv} && \text{(gTRSM)}\\
\frac{dL^T}{dv}\, X + L^T\, \frac{dX}{dv} &= \frac{dY}{dv}, && \text{(gTRSM)}
\end{aligned}$$

none of which is supported by high-performance libraries. For both kernels, CLlCK is capable of generating high-performance algorithms in a matter of seconds.

The aim of this chapter is two-fold: First, we use the operation gCHOL to provide a complete self-contained example of the application of CLlCK, from the description of the operation to the final algorithms. Then, we present experimental results for both gCHOL and gTRSM; the corresponding routines attain high performance and scalability.

5.1 A complete example: The derivative of the Cholesky factorization

To initiate the derivation of algorithms for the derivative of the Cholesky factorization, CLlCK requires the mathematical description of the operation. Given a symmetric positive definite matrix A, the Cholesky factorization calculates a lower triangular matrix L such that

$$LL^T = A; \tag{5.1}$$

its derivative is

$$\frac{dL}{dv}\, L^T + L\, \frac{dL^T}{dv} = \frac{dA}{dv},$$

where L and $\frac{dA}{dv}$ are known, and $\frac{dL}{dv}$ is sought after. The quantities L and $\frac{dL}{dv}$ are lower triangular matrices, and $\frac{dA}{dv}$ is a symmetric matrix.[1] A formal description of the operation is given in Box 5.1; to simplify the notation and to avoid confusion, hereafter we replace $\frac{dA}{dv}$ and $\frac{dL}{dv}$ with B and G, respectively.

We recall that this description is the sole input required by CLlCK to generate algorithms; all the actions leading to the algorithms in Figure 5.4 are carried out automatically.

Pattern Learning Given the description of gCHOL in Box 5.1, CLlCK creates the pattern corresponding to gCHOL (Box 5.2), and incorporates it to its knowledge-base. The system is now capable of identifying gCHOL in the subsequent steps of the process.

[1] The rules to determine the properties of the operands of a derivative equation were given in Section 3.2.2

$$G = gChol(L,B) \equiv \begin{cases} P_{\text{pre}} : \{\texttt{Output}(G) \wedge \texttt{Input}(L) \wedge \texttt{Input}(B) \wedge \\ \qquad \texttt{Matrix}(G) \wedge \texttt{Matrix}(L) \wedge \texttt{Matrix}(B) \wedge \\ \qquad \texttt{LowerTriangular}(G) \wedge \texttt{LowerTriangular}(L) \wedge \\ \qquad \texttt{Symmetric}(B)\} \\ \\ P_{\text{post}} : \{GL^T + LG^T = B\} \end{cases}$$

Box 5.1: Formal description for the derivative of the Cholesky factorization.

```
equal[
  plus[
    times[ G_, trans[L_] ],
    times[ L_, trans[G_] ]
  ],
  B_
] /;  isInputQ[L] && isInputQ[B] && isOutputQ[G] &&
      isMatrixQ[L] && isMatrixQ[B] && isMatrixQ[G] &&
      isLowerTriangularQ[L] && isSymmetricQ[B] &&
      isLowerTriangularQ[G]
```

Box 5.2: Mathematica pattern representing gCHOL.

5.1.1 Generation of the PME

In this initial stage, CLlCK first identifies the feasible sets of partitionings for the operands; then, for each of these sets of partitionings, the system produces the corresponding partitioned postcondition, which gives raise to a number of equations; finally, these equations are matched against known patterns, yielding the PME(s).

Feasible Partitionings

To find the sets of valid partitionings for the operands, CLlCK applies the algorithm described in Section 4.3.1 to the tree representation of gCHOL (Figure 5.1). The algorithm starts by creating a list of disjoint sets, one per dimension of the operands:

$$[\{L_r\},\{L_c\},\{B_r\},\{B_c\},\{G_r\},\{G_c\}].$$

The tree is traversed in postorder. Since G is triangular, G_c and G_r are bound together —the only admissible partitionings for G are 1×1 and 2×2—; similarly, the triangularity of L imposes a binding between L_r and L_c. The corresponding sets of dimensions are merged, resulting in:

$$[\{L_r,L_c\},\{B_r\},\{B_c\},\{G_r,G_c\}].$$

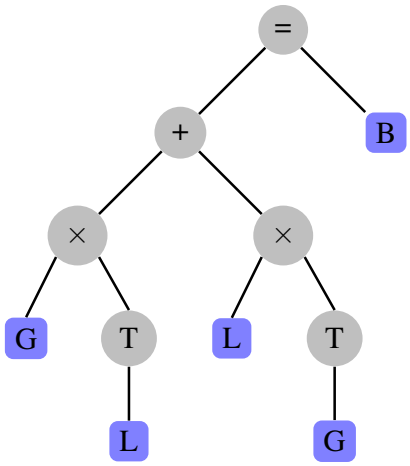


Figure 5.1: Tree representation of gCHOL.

Next, the node for the transpose of L is analyzed, not causing any binding. Hence, CLICK continues by studying the left-most "×" operator. Its children, G and L^T, are, respectively, of size $G_r \times G_c$ and $L_c \times L_r$; a binding between G_c and L_c is thus imposed:

$$[\{L_r, L_c, G_r, G_c\}, \{B_r\}, \{B_c\}].$$

A similar analysis of the subtree corresponding to $L \times G^T$, yields no new bindings. Then, the "+" node is visited. The node's children expressions are of size $G_r \times L_r$ and $L_r \times G_r$; no new bindings occur. CLICK now proceeds with the analysis of the node corresponding to B; due to the symmetry of B, the sets containing B_r and B_c are merged:

$$[\{L_r, L_c, G_r, G_c\}, \{B_r, B_c\}].$$

Finally, the "=" operator, with left-hand side of size $G_r \times L_r$ and right-hand side of size $B_r \times B_c$, imposes the union of the remaining two sets of dimensions. The final list of sets consists of a single group of dimensions:

$$[\{L_r, L_c, G_r, G_c, B_r, B_c\}].$$

Since the application of the identity rule (1×1) to all operands does not lead to a valid partitioned postcondition, the only set of feasible partitionings is the application of the 2×2 rule to every operand:

$$B_{m\times m} \to \left(\begin{array}{c|c} B_{TL} & B_{BL}^T \\ \hline B_{BL} & B_{BR} \end{array}\right), L_{m\times m} \to \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right), G_{m\times m} \to \left(\begin{array}{c|c} G_{TL} & 0 \\ \hline G_{BL} & G_{BR} \end{array}\right).$$

where B_{TL} is $k \times k$ where L_{TL} is $k \times k$ where G_{TL} is $k \times k$

The newly created submatrices inherit a number of properties: B_{TL} and B_{BR} are square and symmetric; B_{BL} and B_{TR} are the transpose of one another; L_{TL}, L_{BR},

G_{TL}, and G_{BR} are lower triangular; L_{TR}, and B_{TR} are the zero matrix; and L_{BL}, and G_{BL} present no structure. CLICK sets these properties and keeps track of them for future use.

Matrix Algebra and Pattern Matching

Next, the system replaces the operands in the postcondition by their partitioned counterparts, producing the partitioned postcondition

$$\left(\begin{array}{c|c} G_{TL} & 0 \\ \hline G_{BL} & G_{BR} \end{array}\right)\left(\begin{array}{c|c} L_{TL}^T & L_{BL}^T \\ \hline 0 & L_{BR}^T \end{array}\right)+\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)\left(\begin{array}{c|c} G_{TL}^T & G_{BL}^T \\ \hline 0 & G_{BR}^T \end{array}\right)=\left(\begin{array}{c|c} B_{TL} & B_{BL}^T \\ \hline B_{BL} & B_{BR} \end{array}\right).$$

The expression is multiplied out and the "=" operator distributed, yielding three equations:[2]

$$\left(\begin{array}{c|c} G_{TL}L_{TL}^T+L_{TL}G_{TL}^T=B_{TL} & * \\ \hline G_{BL}L_{TL}^T+L_{BL}G_{TL}^T=B_{BL} & G_{BL}L_{BL}^T+G_{BR}L_{BR}^T+L_{BL}G_{BL}^T+L_{BR}G_{BR}^T=B_{BR} \end{array}\right). \tag{5.2}$$

The iterative process towards the PME starts. We recall the use of coloring to help the reader following the description: green and red are used to highlight the known and unknown operands, respectively. The operands L and B are input to GCHOL, and so are all sub-matrices resulting from their partitioning. G and its parts are output quantities. All three equations in (5.2) are in canonical form —on the left-hand side appear only output terms, and on the right-hand side appear only input terms—. Hence, no initial algebraic manipulation is required:

$$\left(\begin{array}{c|c} G_{TL}L_{TL}^T+L_{TL}G_{TL}^T=B_{TL} & * \\ \hline G_{BL}L_{TL}^T+L_{BL}G_{TL}^T=B_{BL} & G_{BL}L_{BL}^T+G_{BR}L_{BR}^T+L_{BL}G_{BL}^T+L_{BR}G_{BR}^T=B_{BR} \end{array}\right). \tag{5.3}$$

CLICK inspects (5.3) for known patterns. A GCHOL operation is found in the top-left quadrant: The operation matches the pattern in Box 5.2, L_{TL} and B_{TL} are known matrices, G_{TL} is unknown, G_{TL} and L_{TL} are lower triangular, and B_{TL} is symmetric. The equation is rewritten as the assignment $G_{TL} := gChol(L_{TL}, B_{TL})$, and the unknown quantity, G_{TL}, is labeled as computable and becomes known; this information is propagated to every appearance of the operand:

$$\left(\begin{array}{c|c} G_{TL} := gChol(L_{TL}, B_{TL}) & * \\ \hline G_{BL}L_{TL}^T+L_{BL}G_{TL}^T=B_{BL} & G_{BL}L_{BL}^T+G_{BR}L_{BR}^T+L_{BL}G_{BL}^T+L_{BR}G_{BR}^T=B_{BR} \end{array}\right). \tag{5.4}$$

[2]The symbol $*$ means that the expression in the top-right quadrant is the transpose of that in the bottom-left one.

The bottom-left equation in (5.4) is not in canonical form anymore; a simple step of algebraic manipulation brings the equation back to canonical form:

$$\left(\begin{array}{c|c} G_{TL} := gChol(L_{TL}, B_{TL}) & * \\ \hline G_{BL}L_{TL}^T = B_{BL} - L_{BL}G_{TL}^T & G_{BL}L_{BL}^T + G_{BR}L_{BR}^T + L_{BL}G_{BL}^T + L_{BR}G_{BR}^T = B_{BR} \end{array}\right). \tag{5.5}$$

The bottom-left equation is identified as a triangular system (TRSM). The output operand, G_{BL}, is computable, and turns green in the bottom-right quadrant:

$$\left(\begin{array}{c|c} G_{TL} := gChol(L_{TL}, B_{TL}) & * \\ \hline G_{BL} := (B_{BL} - L_{BL}G_{TL}^T)L_{TL}^{-T} & G_{BL}L_{BL}^T + G_{BR}L_{BR}^T + L_{BL}G_{BL}^T + L_{BR}G_{BR}^T = B_{BR} \end{array}\right). \tag{5.6}$$

A step of algebraic manipulation takes place to reestablish the canonical form in the bottom-right equation, resulting in

$$\left(\begin{array}{c|c} G_{TL} := gChol(L_{TL}, B_{TL}) & * \\ \hline G_{BL} := (B_{BL} - L_{BL}G_{TL}^T)L_{TL}^{-T} & G_{BR}L_{BR}^T + L_{BR}G_{BR}^T = B_{BR} - G_{BL}L_{BL}^T - L_{BL}G_{BL}^T \end{array}\right). \tag{5.7}$$

One last equation remains to be identified. Since L_{BR} and G_{BR} are lower triangular, and the system can establish the symmetry of the right-hand side expression, $B_{BR} - G_{BL}L_{BL}^T - L_{BL}G_{BL}^T$, the equation is matched as a gCHOL. The output quantity, G_{BR}, becomes input. No equation is left, the process completes, and the PME for gCHOL (Box 5.3) is returned.

$$\left(\begin{array}{c|c} G_{TL} := gChol(L_{TL}, B_{TL}) & * \\ \hline G_{BL} := (B_{BL} - L_{BL}G_{TL}^T)L_{TL}^{-T} & G_{BR} := gChol(L_{BR}, B_{BR} - G_{BL}L_{BL}^T - L_{BL}G_{BL}^T) \end{array}\right)$$

Box 5.3: PME for the derivative of the Cholesky factorization.

PME Learning

Once the PME is found, CLICK generates and stores in its knowledge-base the rewrite rule displayed in Box 5.4, which states how to decompose a gCHOL problem with partitioned operands into multiple subproblems. We recall that this rule is essential for the flattening of the P_{before} and P_{after} predicates in later steps of the methodology.

$$\left(\begin{array}{c|c} G_{TL} & 0 \\ \hline G_{BL} & G_{BR} \end{array}\right) := \text{gChol}\left(\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right), \left(\begin{array}{c|c} B_{TL} & B_{BL}^T \\ \hline B_{BL} & B_{BR} \end{array}\right)\right) \longrightarrow$$

$$\left(\begin{array}{c|c} G_{TL} := gChol(L_{TL}, B_{TL}) & * \\ \hline G_{BL} := (B_{BL} - L_{BL}G_{TL}^T)L_{TL}^{-T} & G_{BR} := gChol(L_{BR}, B_{BR} - G_{BL}L_{BL}^T - L_{BL}G_{BL}^T) \end{array}\right)$$

Box 5.4: Rewrite rule associated to gCHOL's PME.

5.1.2 Loop invariant identification

From the PME, multiple loop invariants are identified in three successive steps: First, the PME is decomposed into a set of tasks; then, a graph of dependencies among tasks is built; and finally, the feasible subsets of the graph are returned as valid loop invariants.

Decomposition into tasks

CLlCK analyzes the assignments in each quadrant of the PME, and decomposes them into a series of tasks. The analysis commences from the top-left quadrant: $G_{TL} := gChol(L_{TL}, B_{TL})$. Since the right-hand side consists of a function whose input arguments are simple operands, no decomposition is required and the assignment is returned as a single task.

The next inspected assignment is $G_{BL} := (B_{BL} - L_{BL}G_{TL}^T)L_{TL}^{-T}$. No single pattern matches the right-hand side, which therefore must be decomposed. Similarly to the decomposition undergone by CLAK in Chapter 2, CLlCK first matches the expression $G_{BL} := B_{BL} - L_{BL}G_{TL}^T$ as a matrix-matrix product, and then identifies the remaining operation $G_{BL} := G_{BL}L_{TL}^{-T}$ as the solution of a triangular system. The two operations are yielded as tasks.

One last assignment remains to be studied: $G_{BR} := gChol(L_{BR}, B_{BR} - G_{BL}L_{BL}^T - L_{BL}G_{BL}^T)$. As in the top-left quadrant, it represents a gCHOL function; in this case, however, one of the input arguments is an expression. The decomposition is carried out in two steps: First, CLlCK matches the expression $G_{BR} := B_{BR} - G_{BL}L_{BL}^T - L_{BL}G_{BL}^T$ with the pattern associated to the BLAS 3 operation SYR2K and returns it; then, the function $G_{BR} := gChol(L_{BR}, G_{BR})$ is yielded. The complete list of generated tasks is

1. $G_{TL} := gChol(L_{TL}, B_{TL})$
2. $G_{BL} := B_{BL} - L_{BL}G_{TL}^T$
3. $G_{BL} := G_{BL}L_{TL}^{-T}$
4. $G_{BR} := B_{BR} - G_{BL}L_{BL}^T - L_{BL}G_{BL}^T$

5. $G_{BR} := gChol(L_{BR}, G_{BR})$

Graph of dependencies

A graph of dependencies among tasks is built; the analysis proceeds as follows (we recall the use of **boldface** to highlight the dependencies). The study commences with Task 1. A true dependency is found between Tasks 1 and 2: The output operand of Task 1, G_{TL}, appears as an input quantity to Task 2.

1. $\mathbf{G_{TL}} := gChol(L_{TL}, B_{TL})$
2. $G_{BL} := B_{BL} - L_{BL}\mathbf{G_{TL}^{T}}$.

Next, Task 2 is inspected. Its output operand, G_{BL}, is an input for Task 3:

2. $\mathbf{G_{BL}} := B_{BL} - L_{BL}G_{TL}^{T}$
3. $G_{BL} := \mathbf{G_{BL}}L_{TL}^{-T}$,

which imposes another dependency from Task 2 to 3. Since G_{BL} is also the output of Task 3, an output dependency occurs; the direction of the dependency is imposed during the decomposition of the assignment ($G_{BL} := (B_{BL} - L_{BL}G_{TL}^{T})L_{TL}^{-T}$) that originated these tasks: to ensure a correct result, first Task 2 is computed, and then Task 3.

Two more true dependencies are found from Task 3 to 4,

3. $\mathbf{G_{BL}} := B_{BL}L_{TL}^{-T}$
4. $G_{BR} := B_{BR} - \mathbf{G_{BL}}L_{BL}^{T} - L_{BL}\mathbf{G_{BL}^{T}}$,

and from Task 4 to 5

4. $\mathbf{G_{BR}} := B_{BR} - G_{BL}L_{BL}^{T} - L_{BL}G_{BL}^{T}$
5. $G_{BR} := gChol(L_{BR}, \mathbf{G_{BR}})$.

As for Tasks 2 and 3, an output dependency also exists between Tasks 4 and 5; the direction is imposed by the decomposition: First the argument to the function is computed (Task 4), and then the function itself (Task 5).

Finally, the output of Task 5, G_{BR} does not appear as input to any other task, and the analysis completes. The resulting graph of dependencies is depicted in Figure 5.2.

Graph subsets selection

Predicates candidate to be loop invariants are selected as subsets of the graph that satisfy the dependencies. To obtain the subsets, CLlCK utilizes Algorithm 4.2

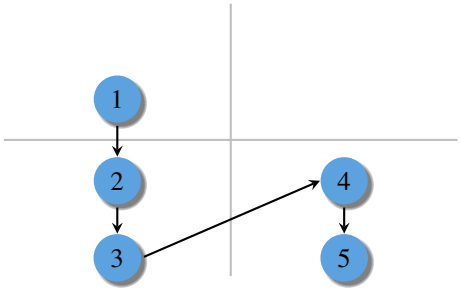


Figure 5.2: Task dependency graph obtained from the analysis of gCHOL's PME.

(Section 4.4.3). Since the application of the algorithm to gCHOL's graph is rather straightforward, we skip the description and give the final list of subsets:

$$[\{\},\{1\},\{1,2\},\{1,2,3\},\{1,2,3,4\},\{1,2,3,4,5\}].$$

According to the rules stated in Chapter 4, the predicates corresponding to the empty and full subgraphs are deemed not valid and discarded. The remaining four predicates lead to feasible loop invariants, which are collected in Table 5.1. In all four loop invariants, the three operands —L, B, and G— are traversed from the top-left to the bottom-right corner.

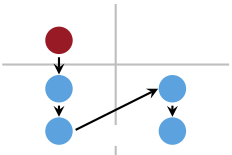
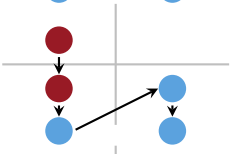
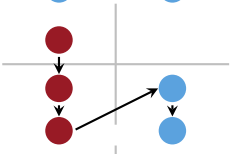
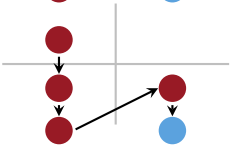
#	Subgraph	Loop-invariant
1		$\left(\begin{array}{c\|c} G_{TL} := gChol(L_{TL}, B_{TL}) & * \\ \hline \neq & \neq \end{array}\right)$
2		$\left(\begin{array}{c\|c} G_{TL} := gChol(L_{TL}, B_{TL}) & * \\ \hline G_{BL} := B_{BL} - L_{BL}G_{TL}^T & \neq \end{array}\right)$
3		$\left(\begin{array}{c\|c} G_{TL} := gChol(L_{TL}, B_{TL}) & * \\ \hline G_{BL} := (B_{BL} - L_{BL}G_{TL}^T)L_{TL}^{-T} & \neq \end{array}\right)$
4		$\left(\begin{array}{c\|c} G_{TL} := gChol(L_{TL}, B_{TL}) & * \\ \hline G_{BL} := (B_{BL} - L_{BL}G_{TL}^T)L_{TL}^{-T} & G_{BR} := B_{BR} - G_{BL}L_{BL}^T - L_{BL}G_{BL}^T \end{array}\right)$

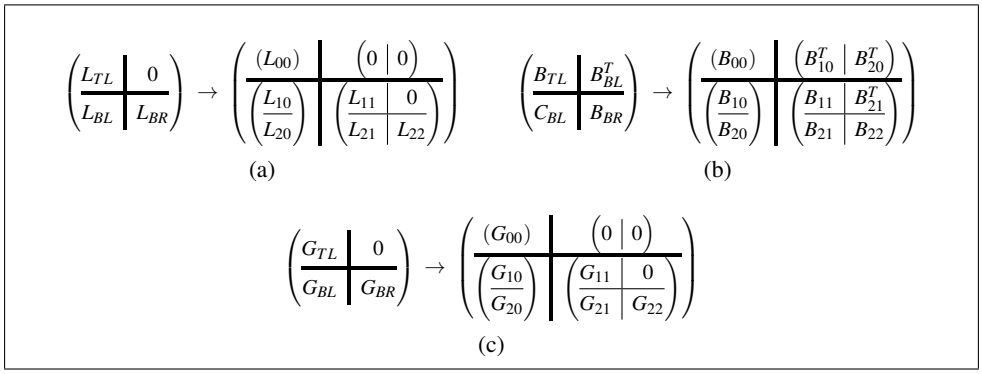
Table 5.1: Four loop invariants for the derivative of the Cholesky factorization.

5.1.3 Algorithm construction

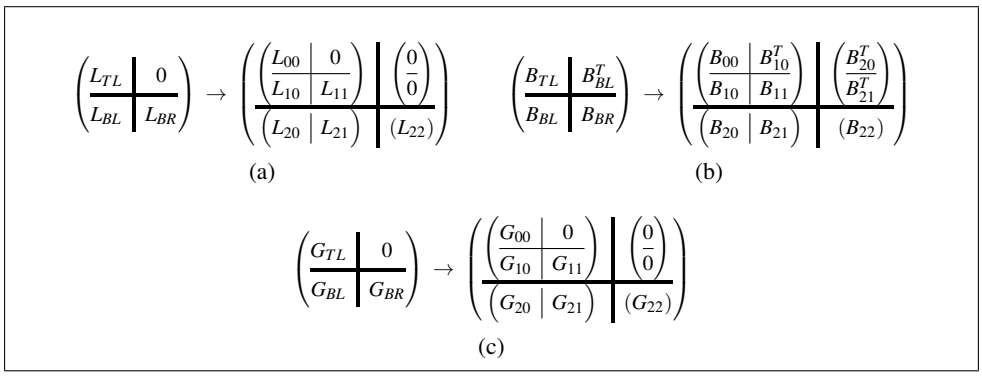
The final stage in the generation of algorithms consists in constructing, for each loop invariant, the corresponding algorithm that computes gCHOL. The construction is carried out in three steps: 1) The repartitioning of the operands (Repartition and Continue with statements), 2) the rewrite of the loop invariant in terms of the repartitioned operands (P_{before} and P_{after} predicates), and 3) the comparison of these predicates to find the Algorithm Updates. We continue the example by means of gCHOL's fourth loop invariant (Table 5.1).

Repartitioning of the operands

The loop invariant states that all three operands are traversed from top-left to bottom-right. This traversal must be captured by the Repartition and Continue with statements; Boxes 5.5 and 5.6, collect the corresponding *Repartition* and *Continue with* rules.



Box 5.5: *Repartition rules* for gCHOL's fourth variant.



Box 5.6: *Continue with rules* for gCHOL's fourth variant.

Predicates P_{before} and P_{after}

To construct the P_{before} and P_{after} predicates, CLlCK first rewrites the loop invariant in terms of the repartitioned operands. To this end, CLlCK applies the *Repartition* and *Continue with* rules, producing, respectively, the top and bottom expressions in Figure 5.3. Then, these expressions are flattened out using both basic matrix algebra and the PME rewrite rule in Box 5.4. The final P_{before} and P_{after} predicates are also given in Figure 5.3.

Finding the updates

The final step undergone by CLlCK consists in determining the updates that take the computation from the state in P_{before} to the state in P_{after}.

The contents of both predicates only differ in quadrants G_{11}, G_{21}, and G_{22}. In the case of G_{11} and G_{22}, the right-hand side of the expressions in P_{before} appear explicitly in the corresponding quadrants of P_{after}. Hence, for those quadrants, a direct replacement suffices to obtain the required updates:

$$\begin{aligned} G_{11} &:= gChol(L_{11}, G_{11}) \\ G_{22} &:= G_{22} - G_{21}L_{21}^T - L_{21}G_{21}^T. \end{aligned}$$

The before state for G_{21}, instead, is not explicitly found in the after state; thus, both must be rewritten so that no redundant subexpressions appear. To avoid clutter, we anticipate that only the after state is rewritten; CLlCK uses the rule from quadrant G_{20}:

$$(B_{20} - L_{20}G_{00}^T)L_{00}^{-T} \rightarrow G_{20},$$

to rewrite the expression

$$G_{21} := (B_{21} - B_{20}L_{00}^{-T}L_{10}^T + L_{20}G_{00}^T L_{00}^{-T}L_{10}^T - L_{20}G_{10}^T - L_{21}G_{11}^T)L_{11}^{-T}$$

into

$$G_{21} := (B_{21} - G_{20}L_{10}^T - L_{20}G_{10}^T - L_{21}G_{11}^T)L_{11}^{-T}.$$

Now, the right-hand side of the before $(B_{21} - G_{20}L_{10}^T - L_{20}G_{10}^T)$ is made explicit in the after, and may be replaced, resulting in the update:

$$G_{21} := (G_{21} - L_{21}G_{11}^T)L_{11}^{-T}.$$

The complete list of the sought-after updates is:

$$\begin{aligned} G_{11} &:= gChol(L_{11}, G_{11}) \\ G_{21} &:= (G_{21} - L_{21}G_{11}^T)L_{11}^{-T} \\ G_{22} &:= G_{22} - G_{21}L_{21}^T - L_{21}G_{21}^T. \end{aligned}$$

$$P_{\text{before}}: \quad \left(\begin{array}{c|c} (G_{00}) := gChol((L_{00}),(B_{00})) & * \\ \hline \left(\frac{G_{10}}{G_{20}}\right) := \left(\left(\frac{B_{10}}{B_{20}}\right) - \left(\frac{L_{10}}{L_{20}}\right)(G_{00})^T\right)(L_{00})^{-T} & \left(\frac{G_{11} \mid 0}{G_{21} \mid G_{22}}\right) := \left(\frac{B_{11} \mid B_{21}^T}{B_{21} \mid B_{22}}\right) - \left(\frac{G_{10}}{G_{20}}\right)\left(\frac{L_{10}}{L_{20}}\right)^T - \left(\frac{L_{10}}{L_{20}}\right)\left(\frac{G_{10}}{G_{20}}\right)^T \end{array}\right)$$

$$\Downarrow \text{ Flattening}$$

$$\left(\begin{array}{c|c|c} G_{00} := gChol(L_{00}, B_{00}) & 0 & 0 \\ \hline G_{10} := (B_{10} - L_{10}G_{00}^T)L_{00}^{-T} & G_{11} := B_{11} - G_{10}L_{10}^T - L_{10}G_{10}^T & 0 \\ \hline G_{20} := (B_{20} - L_{20}G_{00}^T)L_{00}^{-T} & G_{21} := B_{21} - G_{20}L_{10}^T - L_{20}G_{10}^T & G_{22} := B_{22} - G_{20}L_{20}^T - L_{20}G_{20}^T \end{array}\right)$$

Algorithm Updates

$$P_{\text{after}}: \quad \left(\begin{array}{c|c|c} G_{00} := gChol(L_{00}, B_{00}) & 0 & 0 \\ \hline G_{10} := (B_{10} - L_{10}G_{00}^T)L_{00}^{-T} & G_{11} := gChol(L_{11}, B_{11} - G_{10}L_{10}^T - L_{10}G_{10}^T) & 0 \\ \hline G_{20} := (B_{20} - L_{20}G_{00}^T)L_{00}^{-T} & \begin{array}{r} G_{21} := (B_{21} - B_{20}L_{00}^{-T}L_{10}^T + L_{20}G_{00}^T L_{00}^{-T}L_{10}^T \\ - L_{20}G_{10}^T - L_{21}G_{11}^T)L_{11}^{-T} \end{array} & \begin{array}{r} G_{22} := B_{22} - G_{20}L_{20}^T - L_{20}G_{20}^T \\ - G_{21}L_{21}^T - L_{21}G_{21}^T \end{array} \end{array}\right)$$

$$\Uparrow \text{ Flattening}$$

$$\left(\begin{array}{c|c} \left(\frac{G_{00} \mid 0}{G_{10} \mid G_{11}}\right) := gChol\left(\left(\frac{L_{00} \mid 0}{L_{10} \mid L_{11}}\right), \left(\frac{B_{00} \mid B_{10}^T}{B_{10} \mid B_{11}}\right)\right) & * \\ \hline \left(G_{20} \mid G_{21}\right) := \left(\left(B_{20} \mid B_{21}\right) - \left(L_{20} \mid L_{21}\right)\left(\frac{G_{00} \mid 0}{G_{10} \mid G_{11}}\right)^T\right)\left(\frac{L_{00} \mid 0}{L_{10} \mid L_{11}}\right)^{-T} & \begin{array}{r} (G_{22}) := (B_{22}) - \left(G_{20} \mid G_{21}\right)\left(L_{20} \mid L_{21}\right)^T \\ - \left(L_{20} \mid L_{21}\right)\left(G_{20} \mid G_{21}\right)^T \end{array} \end{array}\right)$$

Figure 5.3: P_{before} and P_{after} predicates for gCHOL's fourth loop invariant.

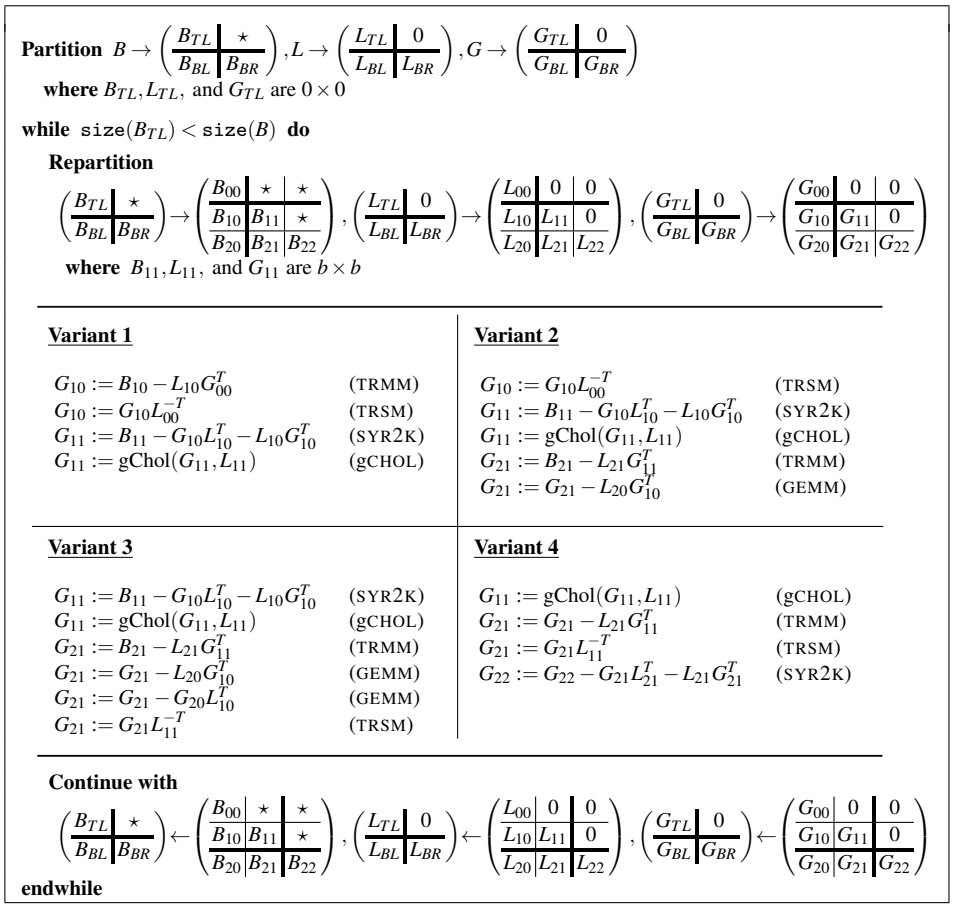


Figure 5.4: The four algorithms for the derivative of the Cholesky factorization generated by CL1CK.

5.1.4 The final algorithms

The process described in the previous section is repeated for each of the four loop invariants for gCHOL. As a result, CL1CK generates the four algorithms collected in Figure 5.4, and the corresponding routines listed in Appendix B.3. Given B, L, and G of size $n \times n$, the computational cost of the algorithms is $\frac{2}{3}n^3$. While the derivation of these algorithms would take hours to an expert, CL1CK generates them in less than 5 seconds; it takes, literally, more time to input the description of the operation than generating the algorithms.

5.1.5 Experimental results

We turn now the attention towards the experimental results. We show that the generated algorithms are not only of theoretical interest, but also of practical relevance. Details on the computing environment for the experiments can be found in

Section 3.4.

We first compare, in Figure 5.5, the four algorithms generated by CLlCK (labeled "Variant 1" to "Variant 4") with the routine generated by ADIFOR. We recall that ADIFOR produces a single routine for the computation of both the Cholesky factorization and its derivative; therefore, to ensure a fair comparison, we also include in the timings for our routines the execution time of both the Cholesky factorization (via LAPACK's DPOTRF) and its derivative. The gap in performance stands out: while ADIFOR's routine attains about 0.6 GFlops/s, all four CLlCK variants attain between 9.5 and 10 GFlops/s. CLlCK's fastest routine is 17 times faster than ADIFOR's.

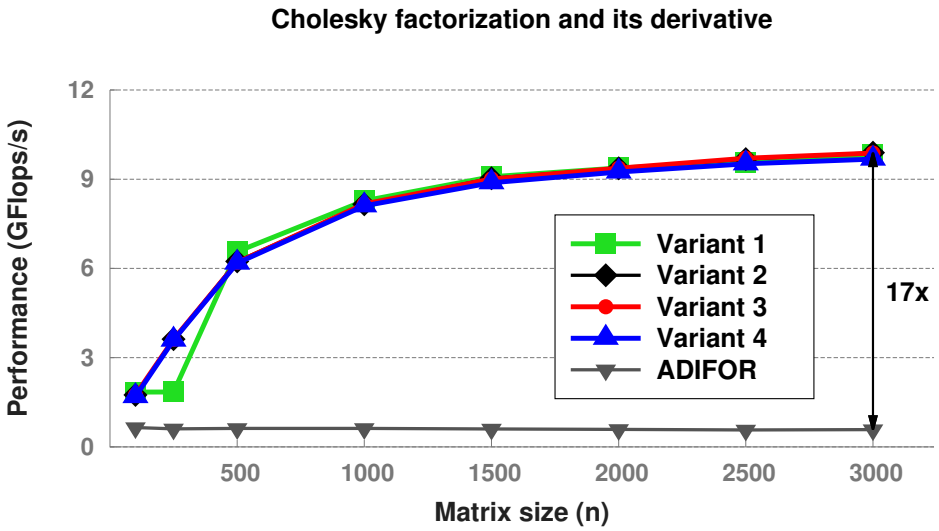


Figure 5.5: Comparison of the performance of CLlCK's four variants for gCHOL with the routine generated by ADIFOR. The experiments were run using a single thread.

We set now ADIFOR aside and concentrate on the performance of CLlCK's routines for the computation of gCHOL exclusively. Figure 5.6 shows results using a single core. The top line (12 GFlop/s) is the theoretical peak of the architecture, and the horizontal black line represents the performance of GEMM, arguably the practical peak (93% of the theoretical). As the figure shows, all 4 variants are very efficient (over 75% of the peak), and the best two —Variants 2 and 3— attain a performance of 10.2 GFlop/s, very close to that of GEMM (11.2 GFlop/s).

We also ran experiments for parallel versions of the routines; parallelism is achieved via a multi-threaded version of the BLAS library. We first look at the scalability of the routines, and then present performance results for their execution using 8 threads. In Figure 5.7, we show the speedup achieved for up to 8 cores; the problem size was fixed to $n = 20{,}000$. The diagonal gray line represents the perfect scalability (speedup equal to the number of cores). The scalability of Variants

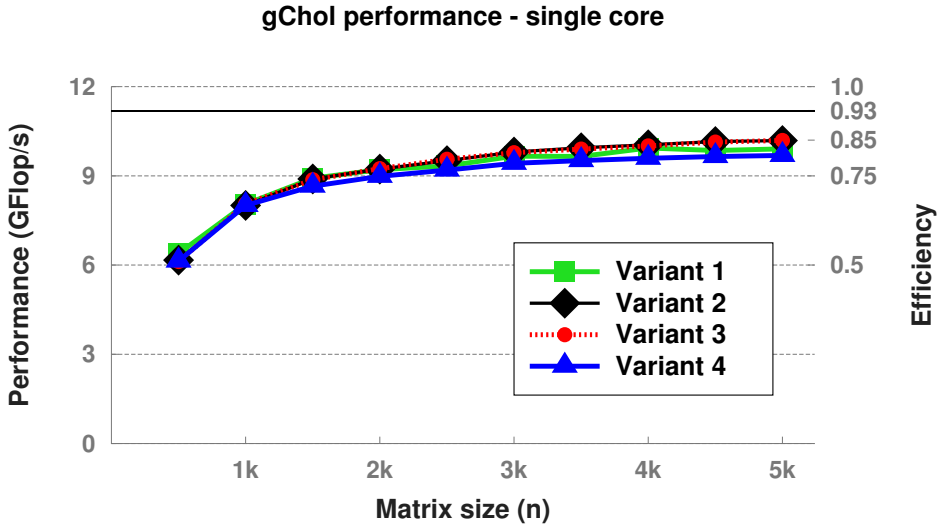


Figure 5.6: Comparison of the performance of CLlCK's four variants for gCHOL. The top border represents the architecture's theoretical peak performance, while the horizontal black line is the peak performance of GEMM (the practical peak).

1 and 2 is rather limited due to the type and shape of the operations that perform the bulk of the computation: In both cases roughly half the computation is carried out in the TRSM operation —$G_{10} := G_{10}L_{00}^{-T}$—, which, in this specific shape (a small number of rows in G_{10}), presents limited scalability; in Variant 1, the other half of the computation is carried out in the TRMM $B_{10} - L_{10}G_{00}^{T}$, that due to limitations inherent to the BLAS interface, requires memory allocation and copy, further limiting the algorithm's scalability. As for Variants 3 and 4, most of the computation is performed via matrix-products (lines 4 and 5 in Variant 3, and line 4 in Variant 4). While both scale well, the specific shape of the SYR2K in Variant 4 is better suited for shared-memory parallelism and attains almost perfect scalability; the resulting speedup is of almost 8x for 8 cores.

Finally, in Figure 5.8, we collect performance results for the four routines when using 8 cores. The top line (96 GFlop/s) is the theoretical peak of the architecture, and the horizontal black line represents the performance of GEMM, arguably the practical peak (87% of the theoretical). The performance of the fastest routine (76.5 GFlop/s) is close to that of GEMM (84 GFlop/s). Interestingly, while the best variants in the single-core case are variants 2 and 3, the best suited for multi-core architectures is variant 4. This is one more example of why multiple variants are desired.

The bottom line of this section is that CLlCK generated efficient and scalable routines at the effort of just one single click.

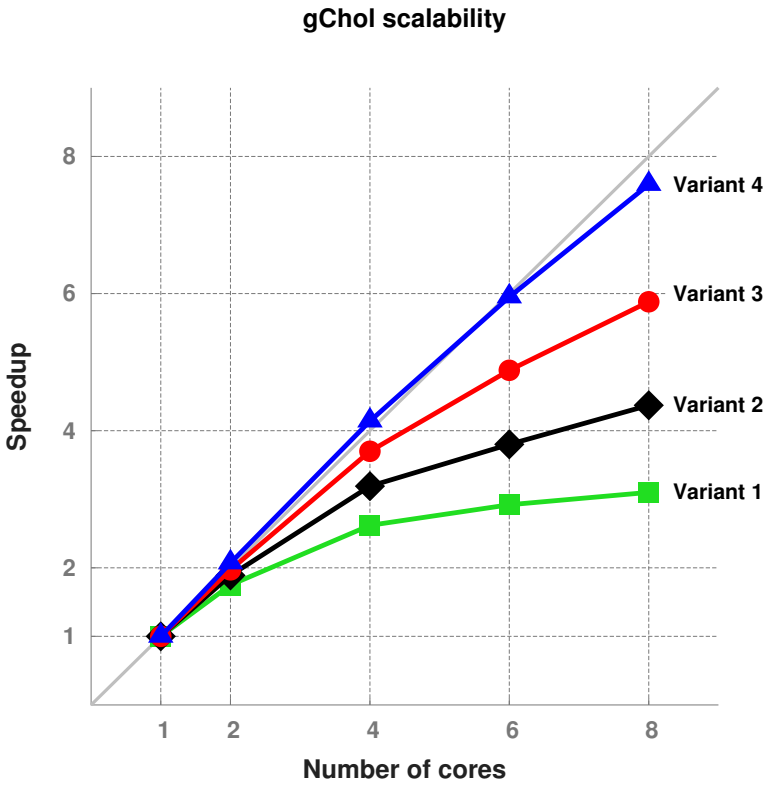


Figure 5.7: Scalability of the four variants for gCHOL. The close-to-perfect scalability of variant 4 stands out, and larger speedups are expected when more cores are available. The problem size is fixed to $n = 20{,}000$.

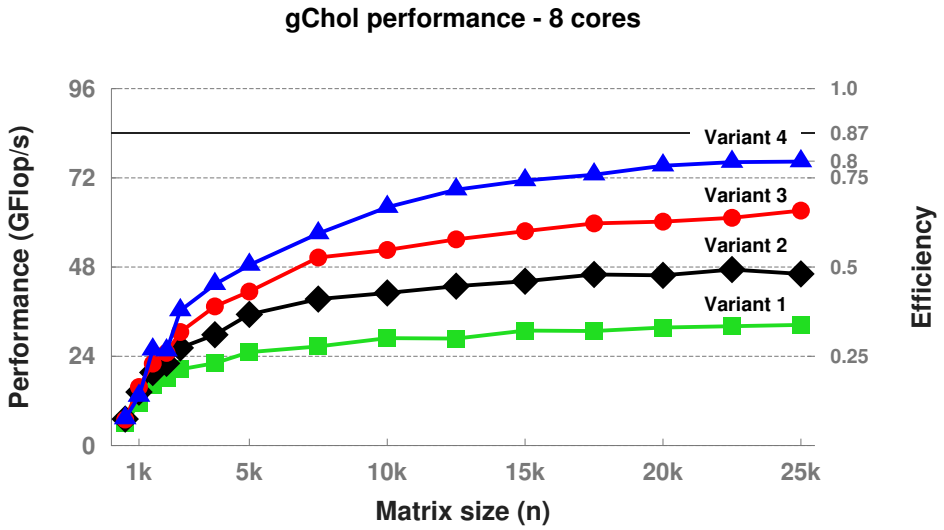


Figure 5.8: Comparison of the performance of CLlCK's four variants for gCHOL. The top border represents the architecture's theoretical peak performance, while the horizontal black line is the peak performance of GEMM (the practical peak).

5.2 The derivative of TRSM

To conclude the motivating example—the derivative of the building blocks involved in the solution of an SPD linear system—, we now focus on the solution of the gTRSM equation

$$L'X + LX' = B',$$

for unknown X', where matrices B, X, $X' \in R^{m\times n}$, and matrices L, $L' \in R^{n\times n}$ are lower triangular.

We discuss two possible approaches to compute this operation efficiently: 1) Mapping it onto calls to BLAS routines, in line with CLAK. 2) Developing a customized blocked routine. The first solution is relatively straightforward, but an overhead is often paid either in terms of extra computation or extra memory (operand) accesses; the second solution, instead, is much harder to implement (by hand), but more efficient. We show that CLlCK is capable of generating automatically customized kernels that outperform a straight mapping onto BLAS kernels.

The equation for gTRSM may be rewritten as

$$X' = L^{-1}(B' - L'X). \tag{5.8}$$

Two possible mappings of Equation (5.8) onto BLAS are given by Algorithms 5.1 (GEMM+TRSM) and 5.2 (TRMM+TRSM). The difference between these two algorithms lies in the kernel used to compute the matrix product $B' - L'X$. Algorithm 5.1 relies on the GEMM kernel, which computes "$\alpha AB + \beta C$" for general

matrices A, B, and C. Due to the triangularity of L', GEMM performs twice the required computation; the algorithm incurs in a 50% redundant computation ($3n^2m$ flops, instead of the only $2n^2m$ of Algorithm 5.2). Alternatively, Algorithm 5.2 exploits the triangularity of L by means of the specialized BLAS kernel TRMM (line 1), which performs the matrix product "αAB" with either A or B triangular; however, in this case the subtraction of the product $L'X$ from B' is not directly supported in the call to TRMM, and it must be calculated in a separate step (line 2). The algorithm performs roughly $2n^2m$, but it is penalized by the overhead due to the multiple sweeps through the operands, and the corresponding increase in memory traffic.

Algorithm 5.1: GEMM + TRSM

1	$X' := B' - L'X$	(GEMM - $2n^2m$)
2	$X' := L^{-1}X'$	(TRSM - n^2m)

Algorithm 5.2: TRMM + TRSM

1	$T := L'X$	(TRMM - n^2m)
2	$X' := B' - T$	(SUB - n^2)
3	$X' := L^{-1}X'$	(TRSM - n^2m)

The alternative is the use of customized blocked algorithms as generated by CLICK. gTRSM is seemingly simpler than gCHOL, but applying FLAME's methodology to the operation is far more complex due to the large number of loop invariants, and thus algorithms, found. CLICK generates more than a hundred routines; all of them require $\frac{2}{3}n^2m$ flops, and perform a single sweep through the operands. We skip the details and jump directly into the experimental results. In Appendix B.4, we provide the two variants used for the experiments in this section.

5.2.1 Experimental results

We study the performance of four routines to compute gTRSM: The first two correspond to GEMM+TRSM and TRMM+TRSM; the other two were generated by CLICK (variants 6 and 63). All four routines are written in C; the experiments were run in the same computing environment used for our previous tests (see Section 3.4).

Figure 5.9 contains the timings for the single-threaded version of the routines. Not surprisingly, GEMM+TRSM performs worst due to the 50% extra computation. Despite the extra memory accesses in TRMM+TRSM, the remaining three routines perform very similarly. It is worth emphasizing that the three best routines attain more than 90% of the architecture's peak performance.

Next, we concentrate on how the routines perform in a shared-memory environment. Figure 5.10 provides scalability results for the four routines; they are not only efficient, but also highly scalable: the speedup attained with 8 cores is of, at least, 7x. The routine TRMM+TRSM present the worst scalability of the four due to the limited scalability of the subtraction operation. This is reflected in the next experiment. In Figure 5.11, we collect the timings for the multi-threaded version of the routines using 8 cores. Due to its better scalability, CLICK's variant 6 out-

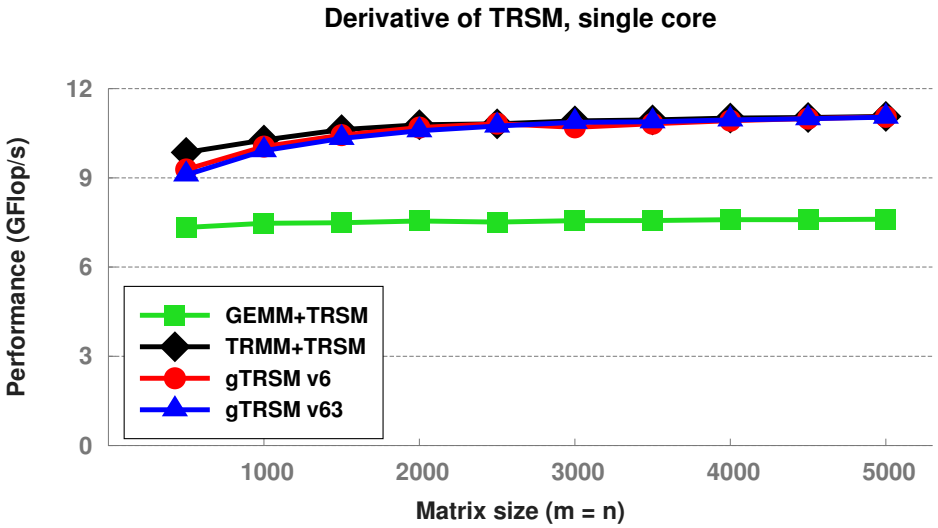


Figure 5.9: Performance comparison of the four routines for gTRSM. Experiments run using a single core.

performs TRMM+TRSM by a 6%, and most important, it attains about 90% of the peak performance.

5.3 Summary

In this chapter, we put the emphasis on the generation of customized kernels for building blocks not supported by standard numerical libraries. We used as examples two kernels arising in the derivation of a linear system: the derivative of the Cholesky factorization (gCHOL) and the derivative of a triangular system (gTRSM).

With gCHOL, we gave a complete step-by-step example of application of CLlCK; the example contributes a new case study (for a non-standard linear algebra operation) to FLAME's literature. As a result, CLlCK produced, in just a few seconds, four algorithms, two of which attain high performance and scalability. The algorithms are of interest, for instance, in the field of statistics, when computing the variance estimation by restricted maximum likelihood (REML). In fact, in [59] S.P. Smith discusses an unblocked version of what we referred to as "Variant 4" (Figure 5.4).

The gTRSM operation is a characteristic example of a class of arbitrary building blocks that often arise in applications and for which no standard library offers optimized kernels. Typically, these operations may be computed as a sequence of calls to supported kernels (in line with the approach discussed in Chapter 2 for CLAK), at the expense of an overhead due to extra computation or extra memory accesses. We demonstrated that CLlCK can generate routines based on blocked algorithms that are competitive with or even outperform those based on calls to a sequence of kernels.

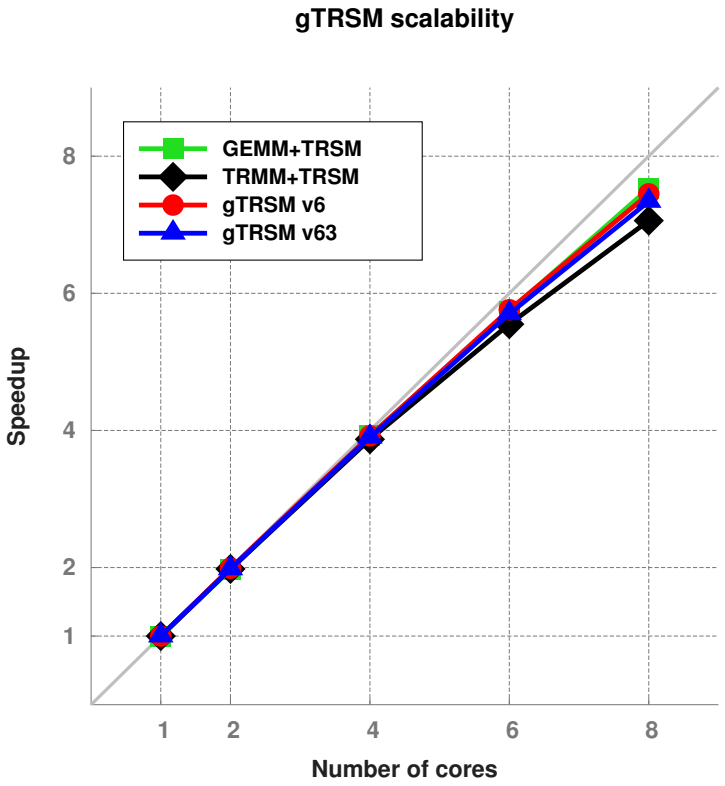


Figure 5.10: Scalability of the four variants to compute gTRSM. The gray diagonal line represents the perfect scalability. Problem size: $m = n = 10{,}000$.

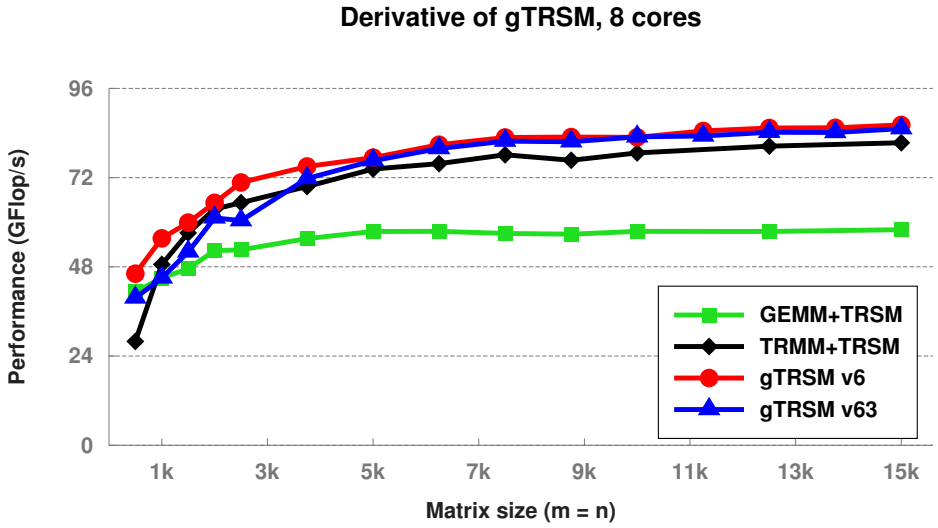


Figure 5.11: Performance comparison of the four routines for gTRSM. Experiments run using the eight available cores.

Chapter 6

Related Work

The concept of automatic program generation has been present since the appearance of digital computers. Quoting Prywes from his 1974's survey on *Automatic generation of software systems* [53]:

> *Research and development on automatic programming has been underway since early applications of digital computers, for nearly twenty-five years, and will continue for years. [...] Its ultimate objective is envisaged as a situation where software would be automatically generated for the businessman, industrialist or scientist, on demand.*

The overarching theme in every effort towards automatic program generation is the increase of productivity by reducing coding and maintainability cost.

Autotuning. Already in 1996, recognizing both the prominent role of the BLAS library in dense matrix computations and the difficulty of providing hand-tuned optimized implementations for a broad range of architectures, the PHiPAC project [12] addressed the problem of producing high-performance implementations of BLAS for a wide range of systems with minimal effort. Instead of hand-coding the routines, this approach consisted in a parameterized code generator and a series of scripts to generate optimized code by varying the parameter values; the best performing routines were selected empirically. ATLAS [68] improves PHiPAC's approach in that it explores a constrained search space, resulting in a faster optimization process. In later stages, ATLAS combines automatically tuned code with user-contributed hand-optimized micro-kernels. Similarly to these two projects, FFTW [27] provides an adaptive library for Fourier Transforms, based also on an empirical search via actual execution and timing.

Efforts on the automatic tuning of libraries have also been made on sparse matrix computations. For instance, OSKI [65] provides a collection of sparse kernels such as matrix-vector products and the solution of triangular systems. The difference with respect to ATLAS lies in that, since the sparsity pattern varies from matrix to matrix, the tuning is in general deferred until run-time. While tuning

affects execution time, the cost is amortized when reused across multiple calls to the same kernels with the same matrix, as it is the case, for instance, for iterative solvers.

The most important factor that separates both our systems, CLlCK and CLAK, from these projects is that, instead of tuning a given algorithm, we start from the mathematical description of the target equation and generate a family of algorithms to solve it. The application of tuning techniques is therefore complementary to our work.

FLAME CLlCK is the culmination of thorough work on the formal derivation of loop-based linear algebra algorithms. In [33], Gunnels et al. outline a series of steps for the derivation of correct loop-based algorithms given a loop invariant for the operation at hands. Bientinesi extends this work in [7], formalizing the entire methodology and introducing the Partitioned Matrix Expression (PME), an object from which loop invariants can be systematically extracted. Bientinesi also presents evidence that an automated system is within reach. CLlCK is the demonstration that the automation of FLAME's methodology is indeed feasible, and the generation of linear algebra libraries with minimal effort is possible.

FLAME also provides multiple Application Programming Interfaces (APIs) for several programming languages and programming paradigms. Among them, we find Elemental [51], a distributed-memory library that provides functionality similar to that of FLAME and LAPACK. Elemental makes use of FLAME's high-level notation in its routines and may be used as target language when code for distributed-memory architectures is desired. In fact, recently, FLAME researchers prototyped DxTer [46], a system that, starting from the representation of a blocked algorithm, like those generated by CLlCK, replicates the process carried out by domain experts to produce efficient distributed-memory implementations using Elemental as target library/domain-specific language. DxTer has been successful in generating code which is competitive with or even more efficient than that manually optimized by Elemental's developers. The combination of DxTer and CLlCK would enable the automatic generation of high-performance distributed-memory implementations of arbitrary kernels.

Deduction of loop invariants. Software correctness is a recurrent problem in computer science. While most software is (at most) thoroughly tested, as Dijkstra pointed out, *"Testing can only show the presence of bugs, not their absence"*. Proofs of correctness consist in identifying a series of predicates assessing properties of the code at multiple points of the program. Since developers rarely annotate their code with such predicates or any kind of formal documentation, a common approach to program verification is to (semi-)automatically infer these properties.

Among the many predicates to be inferred, deduction of loop invariants (which support the proof of correctness for loops) have proven most complex. Automatic deduction of loop invariants dates back to the 70's, mainly in the context of com-

piler technology, where asserting them would allow code reordering and performance optimizations [17, 66]. In the last two decades, the topic gained popularity and many techniques were developed, incorporating ideas from machine learning, artificial intelligence and data mining, among others [18, 26, 40]. Typically, the research efforts target general-purpose programs, and present practical limitations for complex loops.

In sharp contrast to the above projects, FLAME's methodology advocates a constructive approach in which, instead of proving the correctness a posteriori, it first identifies loop invariants and then builds the algorithms around them so that the invariant is satisfied and the program is correct by construction.

Divide-and-conquer decompositions. Even though the goal of the FLAME project is the derivation of loop-based algorithms, the underlying methodology strongly relies on finding a recursive decomposition of the target equation in a divide-and-conquer fashion. This decomposition presents similarities to D. R. Smith's approach to formal derivation of divide-and-conquer algorithms [57]. Indeed, in [58], Smith demonstrates the approach by means of sorting algorithms that may also be derived following FLAME's techniques. In our work, we target a class of linear algebra algorithms.

The starting point in Smith's approach for top-down decompositions are formal specifications of the functionality of the target problem. The functionality is expressed in terms of input and output domains, and input and output conditions. Such specifications are closely related to the traditional formalism —precondition and postcondition predicates— that we use in CLICK: the input and output domains consist of a Cartesian product of matrix, vector, and scalar domains; the input and output conditions are given by the information encoded in the precondition and the postcondition. In both approaches, the formal specification indicates what to solve, not how to solve it.

According to Smith [58], *"One of the principal difficulties in top-down design is knowing how to decompose a problem specification into subproblem specifications"*. In our framework, the PME represents the decomposition operator: It states how the computation of the target problem may be split into the computation of smaller subproblems. The PME itself also encodes the composition operator by indicating how the partial results, obtained from the computation of the subproblems, are combined to assemble the output matrices. The base case (primitive) may be represented by the unblocked version of the algorithms.

Domain-specific compilers After more than a decade of extensive research on domain-specific compilers in computational science, the benefits have been shown in a broad variety of fields. To name a few, the Tensor Contraction Engine (TCE) [5], the FEniCS project [43], and Spiral [54], which target, respectively, tensor contractions, differential equations, and linear transforms for digital signal processing. The general approach consists in defining a high-level domain-specific

input language, and automatically generating efficient code tailored to the target operation.

Spiral [54] is especially related to our work. In its search for efficient implementations, Spiral explores a space that comes from the combination of breakdown rules to decompose the transforms in a divide-and-conquer fashion, and parameterized rewrite rules to incorporate knowledge of the architecture. The PME from FLAME's methodology is closely related to the breakdown rules from Spiral. However, Spiral targets a limited set of equations, for which the "PMEs" are taken from the literature and encoded in the system, while in our case, the PMEs are automatically found for arbitrary input equations. A second difference between the two projects is that we derive loop-based algorithms that rely on computational kernels available from high-performance libraries, mainly BLAS, while Spiral derives recursive algorithms and generates its own optimized code for the base cases. We believe that a combination of the core ideas from both projects has the potential to become an alternative approach for the automatic generation of highly-optimized linear algebra libraries such as BLAS.

A more recent development is the Built-to-Order compiler (BTO) [6]. BTO addresses the generation of high-performance linear algebra kernels, with focus on memory bound operations (e.g., BLAS 1 and BLAS 2 kernels). The main target of this compiler are sequences of such memory bound operations to which tiling and loop fusion are applied to reduce memory traffic. Similar results could be achieved by CLlCK provided that our compiler is extended to accept sequences of operations as input.

High-level languages and libraries High-performance computations have been traditionally associated to low level languages such as C and Fortran. Aiming at relieving the application developers from tedious low level details, programming environments such as Matlab and R act as a convenient interface to optimized libraries, at the expense of performance.

In the last 15 years, domain-specific libraries have been developed with the objective of extending C++ to make it more appealing to computational scientists and the high-performance computing community. The introduction of Veldhuizen's Blitz++ library [64], based on the so-called *Expression Templates* (ET) [63], opened new ways to elegant yet efficient C++ code for linear algebra computations. Expression templates enable libraries that become linear algebra domain-specific languages embedded within C++. The main idea behind Blitz++ is the elimination of the overhead due to the creation and elimination of temporary operands, simulating, in a sense, loop fusion. Thus, it is specially suited for BLAS 1 and 2 operations, i.e., memory bound vector and matrix-vector kernels. This same approach was later adopted by the uBLAS [60] library with the objective of providing functionality similar to BLAS.

More recently, classical use of ET has been deemed insufficient when addressing more complex operations such as matrix-matrix products [39]. An extension,

often referred to as *Smart* Expression Templates (SET), is in use in modern libraries such as Blaze, Armadillo and Eigen [32, 38, 56]. The first two address matrix-vector operations by means of classical ET complemented with their own manually optimized code, while for matrix-matrix products they rely on calls to optimized BLAS routines provided by the user. Eigen differs from Blaze and Armadillo in that it provides its own code even for matrix-matrix products.

Beyond providing a user-friendly interface to high-performance kernels, these libraries focus on low level optimizations for matrix and vector products and additions. In contrast, CLAK targets high-level matrix equations and the discovery of algorithms to solve them. Also, CLAK incorporates a number of optimizations commonly applied by traditional general-purpose compilers [1]; these optimizations are the logic extension to matrix operands of techniques used by traditional compilers on vectors and scalars. ET-based libraries can complement the decomposition performed by CLAK when facing memory bound computations. Similar functionality and performance can also be achieved at compile-time by CLlCK generated algorithms, if combined with low-level techniques such as the use of intrinsics for vectorization.

Algorithmic Differentiation When discussing the applicability of our compilers to algorithmic differentiation, we compared the generated code with that produced by ADIFOR [15]. While ADIFOR serves its purpose as a reference tool to which we can compare our results, it is certainly not the only choice. Over the last decades, a lot of research has been carried out in the field. As a result, the AD landscape is populated with a large variety of tools. Prominent examples are Tapenade [34] and ADiMat [14]. The former can differentiate Fortran and C code, while the latter targets the differentiation of Matlab code; both tools are based on the source code transformation approach and can be used in forward and reverse mode. It is worth noting that, while more modern tools like Tapenade may deliver better performance than ADIFOR, the main results and contributions of our work still hold.

Closely related to our work, research in progress on the *dco* AD tool [48] explores similar ideas to those developed in this dissertation. Specifically, in [44] the authors study the application of AD techniques to the solution of linear systems. By raising the level of abstraction and avoiding a mere black-box approach, they show that the reuse of intermediate results leads to a reduction of the overhead incurred in computing the systems' derivatives.

Chapter 7

Conclusions

In this dissertation we addressed the development of domain-specific compilers for linear algebra operations. The goal was to relieve application developers from the laborious and time consuming tasks of algorithm design and code writing, while still matching or even surpassing the performance attained by experts. We presented two compilers, CLAK and CLlCK; they start from a high-level description of a target matrix operation, together with application domain knowledge, and return both a family of efficient algorithms that compute the operation and the corresponding routines in the language of choice. The main contribution of this thesis is the evidence that linear algebra compilers, which increase experts' productivity and make efficiency accessible to non experts, are within reach.

In the next section, we summarize the main results of our work, and provide references to our research publications. We conclude with a discussion of future research directions to broaden and strengthen the results from this dissertation.

7.1 Results

We developed prototypes of two linear algebra compilers: CLAK, targeting high-level matrix equations, and CLlCK, for building blocks.

- **CLAK: Compiler for matrix equations** [23, 24]. We presented the design of CLAK, a domain-specific compiler for linear algebra equations. CLAK models the reasoning of the thought-process of a human expert, and extends it with the exploration power of a computer. The generation of algorithms centers around the decomposition of a target equation into a sequence of calls to kernels provided by libraries such as BLAS and LAPACK. The decomposition is not unique, and even for simple equations many alternative algorithms can be generated; a number of heuristics, guided by both linear algebra and domain knowledge, are used to prune the search space while tailoring the algorithms to the application. In the discussion, we uncovered the modules that constitute the compiler's engine. The following modules

were discussed: 1) The algebraic manipulation of expressions and knowledge management, 2) the interface to the available building blocks, 3) the inference of properties for the dynamic deduction of knowledge, 4) the analysis of dependencies for the reduction of the computational cost, and 5) the Matlab and Fortran code generation.

- **CLlCK: Complete automation of FLAME's methodology** [21, 22]. The FLAME project provides a systematic methodology for the derivation of correct loop-based algorithms. While the FLAME literature offers many examples of the manual application of the methodology to traditional operations, little evidence existed that it could be made completely mechanical. CLlCK demonstrates that it can be, indeed, automatically carried out by a computer. Given the sole description of an operation, CLlCK derives families of algorithms that compute it. The compiler takes a three-stage approach: First, we detailed the generation of the PME(s), a recursive definition of the operation in a divide-and-conquer fashion. Then, we illustrated the analysis of the PME to identify a family of loop invariants. Finally, we described how each loop invariant is transformed into its corresponding loop-based algorithm. We demonstrated that the methodology applies not only to standard operations, but also to new kernels.

This dissertation also makes contributions to the fields of algorithmic differentiation, and computational biology.

- **BLAS and LAPACK derivatives for algorithmic differentiation (AD).** The code generated by source-transformation forward-mode AD tools for the derivative of BLAS and LAPACK operations suffers from low performance. We illustrated how our compilers automatically generate high-performance code for derivative operations. By raising the level of abstraction from scalars to matrices, the compilers produce derivative routines that exploit library-provided optimized kernels; then, by means of a data dependency analysis, the complexity of the resulting code may also be reduced. We observed speedups with respect to ADIFOR's routines ranging from 5x to 80x. Our work contributes, first, a study of the potential benefits, should high-performance differentiated versions of BLAS and LAPACK be available, and second, a demonstration that the automatic generation of efficient differentiated versions of these libraries is within reach.

- **High-Performance algorithms for GWAS** [20, 25, 50]. Genome-wide association studies carry out large-scale data analyses, and require performing computations ranging from teraflops to hundreds of petaflops. While state-of-the-art libraries are satisfactory for short to medium problem sizes, they are not practical for large-scale problems. When applied to the GWAS equation, CLAK yielded a family of specialized algorithms that efficiently

solve the equation and achieve a lower complexity compared to existing algorithms. CLAK's algorithms led to high-performance out-of-core routines that largely outperform state-of-the-art libraries [19]. These routines have been collected in the publicly available OmicABEL package,[1] as part of the GenABEL suite for statistical genomics.

7.2 Future work

The tools presented in this dissertation can be extended in a number of ways. Here, we briefly discuss the most promising extensions, which we believe are within reach.

- **Integration of performance analysis techniques.** In order to attain high-performance in a variety of scenarios, our compilers generate families of algorithms; a challenging and critical component in a compiler is the automatic selection of the best one. So far, each produced algorithm is accompanied with its computational cost; however, since the mere operation count is not a reliable metric, we aim at incorporating advanced techniques for performance prediction. A promising direction relies on a sample-based approach: The idea is to create performance models not for the competing algorithms, but only for those routines that are used as building blocks. By combining the models, it is then possible to make predictions and to accurately rank the algorithms [49].

- **Support for an extended class of equations.** While broad, the range of supported equations in CLAK is still rather limited. We aim at extending the scope of the compiler by handling more complex operations, ranging from explicit equations (as opposed to assignments) to determinants, logarithms, and matrix functions in general.

- **Algorithm analysis and code generation for parallel architectures.** Our compilers incorporate modules for the translation of the generated algorithms into code. However, only sequential and multi-threaded code (via multi-threaded implementations of BLAS and LAPACK) is produced. The variety of available computing platforms (e.g, multi- and many-core processors, clusters, and co-processors such as GPGPUs) demands the generation of algorithms that are tailored not only to the application but also to the architecture. To this end, we envision the development of a number of modules responsible for the tailoring to each specific architecture and type of parallelism; for instance, algorithms by blocks (out-of-order execution) for multi- and many-cores, distributed-memory for clusters, and the offload of computation to accelerators.

[1] Available at `http://www.genabel.org/packages/OmicABEL`

- **Support for the reverse mode of algorithmic differentiation (AD).** While we only explored the use of our compilers in the forward mode of AD, we believe that similar techniques may be applied to the reverse mode. The main extension to support the reverse mode involves the inclusion of transformation rules corresponding to the chain rule in the forward mode, and the support for additional operators like the trace of a matrix [28]. As ongoing work in the field evidences, similar results in terms of reduction of complexity and increase of performance would be appreciated by the AD community.

Appendix A

BLAS and LAPACK Routines

We list the BLAS and LAPACK operations used across this dissertation, together with their description.

	BLAS 1	
SCAL	Vector scaling	$y := \alpha y$
DOT	Dot product	$\alpha := x^T y$
AXPY	Vector scaling and addition	$y := \alpha x + y$
	BLAS 2	
GER	Outer vector product	$A := \alpha x y^T + A$
GEMV	Matrix-vector product	$y := \alpha A^{\bullet} x + \beta y$
TRSV	Triangular system with single right-hand side	$x := T^{\bullet^{-1}} b$

Table A.1: Collection of the BLAS 1 and 2 routines used in the algorithms presented in this dissertation. Greek, lowercase, and uppercase letters are used for scalars, vectors, and matrices, respectively; T is a triangular matrix; $A^{\bullet}$ indicates that matrix A may be used either transposed or not.

	BLAS 3	
GEMM	Matrix-matrix product	$C := \alpha A^{\bullet} B^{\bullet} + \beta C$
SYRK	Matrix-matrix product, C symmetric	$C := \alpha A A^T + \beta C$
		$C := \alpha A^T A + \beta C$
SYR2K	Matrix-matrix product, C symmetric	$C := \alpha A B^T + B A^T + \beta C$
		$C := \alpha A^T B + B^T A + \beta C$
TRMM	Matrix-matrix product, A triangular	$B := \alpha A^{\bullet} B$
		$B := \alpha B A^{\bullet}$
TRSM	Triangular system with	$X := \alpha T^{\bullet^{-1}} B$
	multiple right-hand sides	$X := \alpha B T^{\bullet^{-1}}$
	LAPACK	
POSV	SPD system with multiple right-hand sides	
POTRF	Cholesky factorization	
SYEVR	Eigendecomposition of a symmetric matrix	
GEQRF	QR factorization	
ORMQR	Matrix-matrix product with a Q matrix as returned by GEQRF	

Table A.2: Collection of the BLAS 3 and LAPACK routines used in the algorithms presented in this dissertation. Greek, lowercase, and uppercase letters are used for scalars, vectors, and matrices, respectively; T is a triangular matrix; $A^{\bullet}$ indicates that matrix A may be used either transposed or not.

Appendix B

Code Samples

We collect a sample of the routines generated by our compilers for the multiple experiments presented in this dissertation. In Sections B.1 and B.2, we provide examples of routines generated by CLAK for the derivative of the SPD linear system and the SYRK kernel, respectively; specifically, we include the routines for the most general derivatives. In Sections B.3 and B.4, we include, respectively, the four routines generated by CLICK for the derivative of Cholesky and two of those produced for gTRSM.

B.1 gSPDSOLVE

Routine B.1: gSPD. Operands A and B are active.

```
1  SUBROUTINE gSPD_11( A, B, gA, gB, sm, sn, ngAs )
2
3      INTEGER sm, sn, ngAs
4      DOUBLE PRECISION A(sm, sm), B(sm, sn), gA(sm, sm, ngAs),
5                       gB(sm, sn, ngAs)
6      DOUBLE PRECISION ONE
7      PARAMETER (ONE=1.0D+0)
8      INTEGER info, p
9
10     EXTERNAL dtrsm, dsymm, dpotrf
11
12     call dpotrf( 'L', sm, A( 1, 1 ), sm, info )
13     DO p = 1, ngAs
14         call dsymm( 'Left', 'Lower', sm, sn, ( - ONE), gA( 1, 1, p ),
15                     sm, B( 1, 1 ), sm, ONE, gB( 1, 1, p ), sm )
16         call dtrsm( 'Left', 'L', 'N', 'N', sm, sn, ONE, A( 1, 1 ), sm,
17                     gB( 1, 1, p ), sm )
18         call dtrsm( 'Left', 'L', 'T', 'N', sm, sn, ONE, A( 1, 1 ), sm,
19                     gB( 1, 1, p ), sm )
20     END DO
21
22     RETURN
23 END
```

B.2 gSYRK

Routine B.2: gSYRK. Operands α, A, β, and C are active.

```
1  SUBROUTINE gSYRK_1111( alpha, A, csA, beta, C, csC, galpha, gA,
2                         csgA, dsgA, gbeta, gC, csgC, dsgC, sm, sn, ngAs )
3
4    INTEGER sm, sn, ngAs, csA, csC, csgA, dsgA, csgC, dsgC
5    DOUBLE PRECISION alpha, A(csA, sn), beta, C(csC, sm), galpha(ngAs),
6                   gA(csgA, dsgA, ngAs), gbeta(ngAs), gC(csgC, dsgC, ngAs)
7    DOUBLE PRECISION tmp880(sm, sm), temp0(sm, sm), temp1(sm, sm)
8    INTEGER info, p, iter1, iter2
9
10   EXTERNAL dsyrk, dsyr2k, dcopy
11
12   DO p = 1, ngAs
13     DO iter1 = 1, sm
14       call dcopy( sm, gC( 1, iter1, p ), 1, temp0( 1, iter1 ), 1 )
15     END DO
16     call dsyrk( 'U', 'N', sm, sn, galpha( p ), A( 1, 1 ), csA,
17                 beta, temp0( 1, 1 ), sm )
18     DO iter1 = 1, sm
19       call dcopy( sm, C( 1, iter1 ), 1, temp1( 1, iter1 ), 1 )
20     END DO
21     call dsyr2k( 'UPPER', 'NO_TRANSPOSE', sm, sn, alpha, A( 1, 1 ), csA,
22                  gA( 1, 1, p ), csgA, gbeta( p ), temp1( 1, 1 ), sm )
23     DO iter2 = 1, sm
24       DO iter1 = 1, sm
25         tmp880( iter1, iter2 ) = temp0( iter1, iter2 ) +
26                                  temp1( iter1, iter2 )
27       END DO
28     END DO
29     DO iter2 = 1, sm
30       DO iter1 = 1, sm
31         gC( iter1, iter2, p ) = tmp880( iter1, iter2 )
32       END DO
33     END DO
34   END DO
35
36   RETURN
37 END
```

B.3 gCHOL

Routine B.3: gCHOL. Variant 1.

```
1  void gChol_blk_var1( FLA_Obj G, FLA_Obj L, int nb )
2  {
3    FLA_Obj GTL, GTR, GBL, GBR, G00, G01, G02, G10, G11, G12, G20, G21, G22;
4    FLA_Obj LTL, LTR, LBL, LBR, L00, L01, L02, L10, L11, L12, L20, L21, L22;
5
6    FLA_Part_2x2( G, &GTL, &GTR,
7                           &GBL, &GBR, 0, 0, FLA_TL );
8
9    FLA_Part_2x2( L, &LTL, &LTR,
10                          &LBL, &LBR, 0, 0, FLA_TL );
11
12   while ( FLA_Obj_length( GTL ) < FLA_Obj_length( G ) ||
13           FLA_Obj_width( GTL ) < FLA_Obj_width( G ) )
14   {
15     FLA_Repart_2x2_to_3x3( GTL, GTR, &G00, &G01, &G02,
16                                      &G10, &G11, &G12,
17                            GBL, GBR, &G20, &G21, &G22,
18                            nb, nb, FLA_BR );
19
20     FLA_Repart_2x2_to_3x3( LTL, LTR, &L00, &L01, &L02,
21                                      &L10, &L11, &L12,
22                            LBL, LBR, &L20, &L21, &L22,
23                            nb, nb, FLA_BR );
24
25     FLA_Trmmsx_external( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
26                          FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
27                          FLA_MINUS_ONE, G00, L10, FLA_ONE, G10);
28     FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR, FLA_TRANSPOSE,
29               FLA_NONUNIT_DIAG, FLA_ONE, L00, G10);
30     FLA_Syr2k( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
31                FLA_MINUS_ONE, G10, L10, FLA_ONE, G11 );
32     FLA_gChol_unb(G11, L11);
33
34     FLA_Cont_with_3x3_to_2x2( &GTL, &GTR, G00, G01, G02,
35                                           G10, G11, G12,
36                               &GBL, &GBR, G20, G21, G22, FLA_TL );
37
38     FLA_Cont_with_3x3_to_2x2( &LTL, &LTR, L00, L01, L02,
39                                           L10, L11, L12,
40                               &LBL, &LBR, L20, L21, L22, FLA_TL );
41   }
42 }
```

Routine B.4: gCHOL. Variant 2.

```
1  void gChol_blk_var2( FLA_Obj G, FLA_Obj L, int nb )
2  {
3    FLA_Obj GTL, GTR, GBL, GBR, G00, G01, G02, G10, G11, G12, G20, G21, G22;
4    FLA_Obj LTL, LTR, LBL, LBR, L00, L01, L02, L10, L11, L12, L20, L21, L22;
5
6    FLA_Part_2x2( G, &GTL, &GTR,
7                           &GBL, &GBR, 0, 0, FLA_TL );
8
9    FLA_Part_2x2( L, &LTL, &LTR,
10                          &LBL, &LBR, 0, 0, FLA_TL );
11
12   while ( FLA_Obj_length( GTL ) < FLA_Obj_length( G ) ||
13           FLA_Obj_width( GTL ) < FLA_Obj_width( G ) )
14   {
15     FLA_Repart_2x2_to_3x3( GTL, GTR, &G00, &G01, &G02,
16                                      &G10, &G11, &G12,
17                            GBL, GBR, &G20, &G21, &G22,
18                            nb, nb, FLA_BR );
19
20     FLA_Repart_2x2_to_3x3( LTL, LTR, &L00, &L01, &L02,
21                                      &L10, &L11, &L12,
22                            LBL, LBR, &L20, &L21, &L22,
23                            nb, nb, FLA_BR );
24
25     FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR, FLA_TRANSPOSE,
26               FLA_NONUNIT_DIAG, FLA_ONE, L00, G10);
27     FLA_Syr2k( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
28                FLA_MINUS_ONE, G10, L10, FLA_ONE, G11 );
29     FLA_gChol_unb(G11, L11);
30     FLA_Trmmsx_external( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
31                          FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
32                          FLA_MINUS_ONE, G11, L21, FLA_ONE, G21);
33     FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
34               FLA_MINUS_ONE, L20, G10, FLA_ONE, G21);
35
36     FLA_Cont_with_3x3_to_2x2( &GTL, &GTR, G00, G01, G02,
37                                           G10, G11, G12,
38                               &GBL, &GBR, G20, G21, G22, FLA_TL );
39
40     FLA_Cont_with_3x3_to_2x2( &LTL, &LTR, L00, L01, L02,
41                                           L10, L11, L12,
42                               &LBL, &LBR, L20, L21, L22, FLA_TL );
43   }
44 }
```

Routine B.5: gCHOL. Variant 3.

```
1  void gChol_blk_var3( FLA_Obj G, FLA_Obj L, int nb )
2  {
3    FLA_Obj GTL, GTR, GBL, GBR, G00, G01, G02, G10, G11, G12, G20, G21, G22;
4    FLA_Obj LTL, LTR, LBL, LBR, L00, L01, L02, L10, L11, L12, L20, L21, L22;
5
6    FLA_Part_2x2( G, &GTL, &GTR,
7                           &GBL, &GBR, 0, 0, FLA_TL );
8
9    FLA_Part_2x2( L, &LTL, &LTR,
10                          &LBL, &LBR, 0, 0, FLA_TL );
11
12   while ( FLA_Obj_length( GTL ) < FLA_Obj_length( G ) ||
13           FLA_Obj_width( GTL ) < FLA_Obj_width( G ) )
14   {
15     FLA_Repart_2x2_to_3x3( GTL, GTR, &G00, &G01, &G02,
16                                      &G10, &G11, &G12,
17                            GBL, GBR, &G20, &G21, &G22,
18                            nb, nb, FLA_BR );
19
20     FLA_Repart_2x2_to_3x3( LTL, LTR, &L00, &L01, &L02,
21                                      &L10, &L11, &L12,
22                            LBL, LBR, &L20, &L21, &L22,
23                            nb, nb, FLA_BR );
24
25     FLA_Syr2k( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
26                FLA_MINUS_ONE, G10, L10, FLA_ONE, G11 );
27     FLA_gChol_unb(G11, L11);
28     FLA_Trmmsx_external( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
29                          FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
30                          FLA_MINUS_ONE, G11, L21, FLA_ONE, G21);
31     FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
32               FLA_MINUS_ONE, L20, G10, FLA_ONE, G21);
33     FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
34               FLA_MINUS_ONE, G20, L10, FLA_ONE, G21);
35     FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR, FLA_TRANSPOSE,
36               FLA_NONUNIT_DIAG, FLA_ONE, L11, G21);
37
38     FLA_Cont_with_3x3_to_2x2( &GTL, &GTR, G00, G01, G02,
39                                           G10, G11, G12,
40                               &GBL, &GBR, G20, G21, G22, FLA_TL );
41
42     FLA_Cont_with_3x3_to_2x2( &LTL, &LTR, L00, L01, L02,
43                                           L10, L11, L12,
44                               &LBL, &LBR, L20, L21, L22, FLA_TL );
45   }
46 }
```

Routine B.6: gCHOL. Variant 4.

```
1  void gChol_blk_var4( FLA_Obj G, FLA_Obj L, int nb )
2  {
3    FLA_Obj GTL, GTR, GBL, GBR, G00, G01, G02, G10, G11, G12, G20, G21, G22;
4    FLA_Obj LTL, LTR, LBL, LBR, L00, L01, L02, L10, L11, L12, L20, L21, L22;
5
6    FLA_Part_2x2( G, &GTL, &GTR,
7                           &GBL, &GBR, 0, 0, FLA_TL );
8
9    FLA_Part_2x2( L, &LTL, &LTR,
10                          &LBL, &LBR, 0, 0, FLA_TL );
11
12   while ( FLA_Obj_length( GTL ) < FLA_Obj_length( G ) ||
13           FLA_Obj_width( GTL ) < FLA_Obj_width( G ) )
14   {
15     FLA_Repart_2x2_to_3x3( GTL, GTR, &G00, &G01, &G02,
16                                      &G10, &G11, &G12,
17                            GBL, GBR, &G20, &G21, &G22,
18                            nb, nb, FLA_BR );
19
20     FLA_Repart_2x2_to_3x3( LTL, LTR, &L00, &L01, &L02,
21                                      &L10, &L11, &L12,
22                            LBL, LBR, &L20, &L21, &L22,
23                            nb, nb, FLA_BR );
24
25     FLA_gChol_unb(G11, L11);
26     FLA_Trmmsx_external( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
27                          FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
28                          FLA_MINUS_ONE, G11, L21, FLA_ONE, G21);
29     FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR, FLA_TRANSPOSE,
30               FLA_NONUNIT_DIAG, FLA_ONE, L11, G21);
31     FLA_Syr2k( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
32                FLA_MINUS_ONE, G21, L21, FLA_ONE, G22 );
33
34     FLA_Cont_with_3x3_to_2x2( &GTL, &GTR, G00, G01, G02,
35                                           G10, G11, G12,
36                               &GBL, &GBR, G20, G21, G22, FLA_TL );
37
38     FLA_Cont_with_3x3_to_2x2( &LTL, &LTR, L00, L01, L02,
39                                           L10, L11, L12,
40                               &LBL, &LBR, L20, L21, L22, FLA_TL );
41   }
42 }
```

B.4 gTRSM

Routine B.7: gTRSM. Variant 6.

```
1  void gTRSM_blk_var6(FLA_Obj gL, FLA_Obj X, FLA_Obj L, FLA_Obj gX, int nb)
2  {
3    FLA_Obj gLTL, gLTR, gLBL, gLBR, gL00, gL01, gL02, gL10, gL11, gL12,
4                                    gL20, gL21, gL22;
5    FLA_Obj gXT, gXB, gX0, gX1, gX2;
6    FLA_Obj LTL, LTR, LBL, LBR, L00, L01, L02, L10, L11, L12, L20, L21, L22;
7    FLA_Obj XT, XB, X0, X1, X2;
8
9    FLA_Part_2x2( gL, &gLTL, &gLTR,
10                   &gLBL, &gLBR, 0, 0, FLA_TL );
11   FLA_Part_2x1( gX, &gXT,
12                   &gXB, 0, FLA_TOP );
13   [...]
14   while ( FLA_Obj_length( gXT ) < FLA_Obj_length( gX ) ||
15           FLA_Obj_width( gXT ) < FLA_Obj_width( gX ) )
16   {
17     FLA_Repart_2x2_to_3x3( gLTL, gLTR, &gL00, &gL01, &gL02,
18                                        &gL10, &gL11, &gL12,
19                            gLBL, gLBR, &gL20, &gL21, &gL22,
20                            nb, nb, FLA_BR );
21     FLA_Repart_2x1_to_3x1( gXT, &gX0,
22                                 &gX1,
23                            gXB, &gX2, nb, FLA_BOTTOM );
24     [...]
25
26     FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
27               FLA_MINUS_ONE, gL11, X1, FLA_ONE, gX1 );
28     FLA_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
29               FLA_NONUNIT_DIAG, FLA_ONE, L11, gX1 );
30     FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
31               FLA_MINUS_ONE, L21, gX1, FLA_ONE, gX2 );
32     FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
33               FLA_MINUS_ONE, gL21, X1, FLA_ONE, gX2 );
34
35     FLA_Cont_with_3x3_to_2x2( &gLTL, &gLTR, gL00, gL01, gL02,
36                                          gL10, gL11, gL12,
37                               &gLBL, &gLBR, gL20, gL21, gL22, FLA_TL );
38     FLA_Cont_with_3x1_to_2x1( &gXT, gX0,
39                                     gX1,
40                               &gXB, gX2, FLA_TOP );
41     [...]
42   }
43 }
```

Routine B.8: gTRSM. Variant 63.

```
1  void gTRSM_blk_var63(FLA_Obj gL, FLA_Obj X, FLA_Obj L, FLA_Obj gX, int nb)
2  {
3    FLA_Obj gLTL, gLTR, gLBL, gLBR, gL00, gL01, gL02, gL10, gL11, [...];
4    FLA_Obj gXTL, gXTR, gXBL, gXBR, gX00, gX01, gX02, gX10, gX11, [...];
5    FLA_Obj LTL, LTR, LBL, LBR, L00, L01, L02, L10, L11, L12, L20, L21, L22;
6    FLA_Obj XTL, XTR, XBL, XBR, X00, X01, X02, X10, X11, X12, X20, X21, X22;
7
8    FLA_Part_2x2( gL, &gLTL, &gLTR,
9                      &gLBL, &gLBR, 0, 0, FLA_TL );
10   [...]
11   while ( FLA_Obj_length( gXTR ) < FLA_Obj_length( gX ) ||
12           FLA_Obj_width( gXTR ) < FLA_Obj_width( gX ) )
13   {
14     FLA_Repart_2x2_to_3x3( gLTL, gLTR, &gL00, &gL01, &gL02,
15                                        &gL10, &gL11, &gL12,
16                            gLBL, gLBR, &gL20, &gL21, &gL22,
17                            nb, nb, FLA_BR );
18     [...]
19     FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
20               FLA_MINUS_ONE, gL11, X10, FLA_ONE, gX10 );
21     FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
22               FLA_MINUS_ONE, gL11, X11, FLA_ONE, gX11 );
23     FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
24               FLA_MINUS_ONE, gL11, X12, FLA_ONE, gX12 );
25     FLA_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
26               FLA_NONUNIT_DIAG, FLA_ONE, L11, gX10);
27     FLA_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
28               FLA_NONUNIT_DIAG, FLA_ONE, L11, gX11);
29     FLA_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
30               FLA_NONUNIT_DIAG, FLA_ONE, L11, gX12);
31     FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
32               FLA_MINUS_ONE, L21, gX10, FLA_ONE, gX20 );
33     FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
34               FLA_MINUS_ONE, L21, gX11, FLA_ONE, gX21 );
35     FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
36               FLA_MINUS_ONE, L21, gX12, FLA_ONE, gX22 );
37     FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
38               FLA_MINUS_ONE, gL21, X10, FLA_ONE, gX20 );
39     FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
40               FLA_MINUS_ONE, gL21, X11, FLA_ONE, gX21 );
41     FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
42               FLA_MINUS_ONE, gL21, X12, FLA_ONE, gX22 );
43
44     FLA_Cont_with_3x3_to_2x2( &gLTL, &gLTR, gL00, gL01, gL02,
45                                             gL10, gL11, gL12,
46                               &gLBL, &gLBR, gL20, gL21, gL22, FLA_TL );
47     [...]
48   }
49 }
```

Bibliography

[1] Aho, A. V., Sethi, R. & Ullman, J. D. *Compilers principles, techniques, and tools* (Addison-Wesley, Reading, MA, 1986).

[2] Anderson, E. *et al. LAPACK Users' Guide* (Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999), third edn.

[3] Astle, W. & Balding, D. Population structure and cryptic relatedness in genetic association studies. *Statist. Sci.* **24**, 451–471 (2009).

[4] Aulchenko, Y. S., Ripke, S., Isaacs, A. & van Duijn, C. M. Genabel: an R library for genome-wide association analysis. *Bioinformatics* **23**, 1294–6 (2007).

[5] Baumgartner, G. *et al.* Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. In *Proceedings of the IEEE*, 2005 (2005).

[6] Belter, G., Jessup, E. R., Karlin, I. & Siek, J. G. Automating the generation of composed linear algebra kernels. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, 59:1–59:12 (ACM, New York, NY, USA, 2009).

[7] Bientinesi, P. *Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms*. Ph.D. thesis, Department of Computer Sciences, The University of Texas (2006).

[8] Bientinesi, P., Gunnels, J. A., Myers, M. E., Quintana-Ortí, E. S. & van de Geijn, R. A. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Software* **31**, 1–26 (2005).

[9] Bientinesi, P. *et al.* Deriving dense linear algebra libraries. *Formal Aspects of Computing* 1–13 (2012).

[10] Bientinesi, P., Gunter, B. & van de Geijn, R. A. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Software* **35**, 1–22 (2008).

[11] Bientinesi, P. & van de Geijn, R. Goal-oriented and modular stability analysis. *SIAM Journal on Matrix Analysis and Applications* **32** (2011).

[12] Bilmes, J., Asanovic, K., Chin, C.-W. & Demmel, J. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *Proceedings of the 11th international conference on Supercomputing*, ICS '97, 340–347 (ACM, New York, NY, USA, 1997).

[13] Bischof, C. H. & Bücker, H. M. Computing derivatives of computer programs. In Grotendorst, J. (ed.) *Modern Methods and Algorithms of Quantum Chemistry: Proceedings, Second Edition*, vol. 3 of *NIC Series*, 315–327 (NIC-Directors, Jülich, 2000).

[14] Bischof, C. H., Bücker, H. M., Lang, B., Rasch, A. & Vehreschild, A. Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, 65–72 (IEEE Computer Society, Los Alamitos, CA, USA, 2002).

[15] Bischof, C. H., Carle, A., Khademi, P. & Mauer, A. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering* **3**, 18–32 (1996).

[16] Blackford, S. *et al.* An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* **28**, 135–151 (2002).

[17] Cousot, P. & Halbwachs, N. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, 84–96 (ACM, New York, NY, USA, 1978).

[18] Ernst, M. D. *Dynamically Discovering Likely Program Invariants*. Ph.D. thesis, University of Washington (2000).

[19] Fabregat-Traver, D., Aulchenko, Y. S. & Bientinesi, P. High-throughput genome-wide association analysis for single and multiple phenotypes. Tech. Rep. (2012). Available at: http://arxiv.org/abs/1207.2169.

[20] Fabregat-Traver, D., Aulchenko, Y. S. & Bientinesi, P. Solving sequences of generalized least-squares problems on multi-threaded architectures. *Applied Mathematics and Computation (AMC)* **234**, 606–617 (2014).

[21] Fabregat-Traver, D. & Bientinesi, P. Automatic generation of loop-invariants for matrix operations. In *Computational Science and its Applications, International Conference*, 82–92 (IEEE Computer Society, Los Alamitos, CA, USA, 2011).

[22] Fabregat-Traver, D. & Bientinesi, P. Knowledge-based automatic generation of partitioned matrix expressions. In *Computer Algebra in Scientific Computing*, vol. 6885 of *Lecture Notes in Computer Science*, 144–157 (Springer Berlin / Heidelberg, 2011).

[23] Fabregat-Traver, D. & Bientinesi, P. Application-tailored linear algebra algorithms: A search-based approach. *International Journal of High Performance Computing Applications (IJHPCA)* **27**, 425–438 (2013).

[24] Fabregat-Traver, D. & Bientinesi, P. A domain-specific compiler for linear algebra operations. In *High Performance Computing for Computational Science – VECPAR 2012*, vol. 7851 of *Lecture Notes in Computer Science* (Springer Berlin / Heidelberg, 2013).

[25] Fabregat-Traver, D. & Bientinesi, P. Computing petaflops over terabytes of data: The case of genome-wide association studies. *ACM Trans. Math. Software* (2014). To appear.

[26] Flanagan, C. & Qadeer, S. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, 191–202 (ACM, New York, NY, USA, 2002).

[27] Frigo, M. & Johnson, S. G. The design and implementation of FFTW3. *Proceedings of the IEEE* **93**, 216–231 (2005). Special issue on "Program Generation, Optimization, and Platform Adaptation".

[28] Giles, M. B. Collected matrix derivative results for forward and reverse mode algorithmic differentiation. In Bischof, C. H., Bücker, H. M., Hovland, P. D., Naumann, U. & Utke, J. (eds.) *Advances in Automatic Differentiation*, 35–44 (Springer, 2008).

[29] Golub, G. H. & Van Loan, C. F. *Matrix computations (3rd ed.)* (Johns Hopkins University Press, Baltimore, MD, USA, 1996).

[30] Gries, D. & Schneider, F. B. *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science (Springer Verlag, 1992).

[31] Griewank, A. & Walther, A. *Evaluating derivatives - Principles and techniques of algorithmic differentiation (2. ed.)* (SIAM, 2008).

[32] Guennebaud, G., Jacob, B. *et al.* Eigen v3. http://eigen.tuxfamily.org (2010).

[33] Gunnels, J. A., Gustavson, F. G., Henry, G. M. & van de Geijn, R. A. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Software* **27**, 422–455 (2001).

[34] Hascoët, L. & Pascual, V. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Software* **39**, 20:1–20:43 (2013).

[35] Hascoët, L., Naumann, U. & Pascual, V. "To Be Recorded" analysis in reverse-mode automatic differentiation. *Future Gener. Comput. Syst.* **21**, 1401–1417 (2005).

[36] Hindorff, L. *et al.* A catalog of published genome-wide association studies (2012). Available at: www.genome.gov/gwastudies. Accessed July 22nd, 2012.

[37] IBM. Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN. Tech. Rep. (1954).

[38] Iglberger, K., Hager, G., Treibig, J. & Rude, U. High performance smart expression template math libraries. In *International Conference on High Performance Computing and Simulation (HPCS)*, 367–373 (2012).

[39] Iglberger, K., Hager, G., Treibig, J. & Rde, U. Expression templates revisited: A performance analysis of current methodologies. *SIAM Journal on Scientific Computing* **34**, C42–C69 (2012).

[40] Ireland, A. & Stark, J. On the automatic discovery of loop invariants. In *Proceedings of the Fourth NASA Langley Formal Methods Workshop* (1997).

[41] Jonsson, I. & Kågström, B. Recursive blocked algorithms for solving triangular systems—Part I: One-sided and coupled Sylvester-type matrix equations. *ACM Trans. Math. Software* **28**, 392–415 (2002).

[42] Jonsson, I. & Kågström, B. Recursive blocked algorithms for solving triangular systems—Part II: Two-sided and generalized Sylvester and Lyapunov matrix equations. *ACM Trans. Math. Software* **28**, 416–435 (2002).

[43] Logg, A. & Wells, G. N. Dolfin: Automated finite element computing. *ACM Trans. Math. Software* **37**, 20:1–20:28 (2010).

[44] Lotz, J. & Naumann, U. Algorithmic differentiation of numerical methods: Tangent-linear and adjoint direct solvers for systems of linear equations. Tech. Rep. AIB-2012-10, RWTH Aachen (2012). Available at http://aib.informatik.rwth-aachen.de/2012/2012-10.ps.gz.

[45] Maplesoft, a division of Waterloo Maple Inc. *Maple* (Waterloo, Ontario).

[46] Marker, B., Poulson, J., Batory, D. & van de Geijn, R. Designing linear algebra algorithms by transformation: Mechanizing the expert developer. In *High Performance Computing for Computational Science – VECPAR 2012*, vol. 7851 of *Lecture Notes in Computer Science* (Springer Berlin / Heidelberg, 2013).

[47] MKL. *Intel Math Kernel Library*. URL http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation.

[48] Naumann, U. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. No. 24 in Software, Environments, and Tools (SIAM, Philadelphia, PA, 2012).

[49] Peise, E. & Bientinesi, P. Performance modeling for dense linear algebra. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis (PMBS12)*, SCC '12, 406–416 (IEEE Computer Society, Washington, DC, USA, 2012).

[50] Peise, E., Fabregat-Traver, D., Aulchenko, Y. & Bientinesi, P. Algorithms for large-scale whole genome association analysis. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, 229–234 (ACM, New York, NY, USA, 2013).

[51] Poulson, J., Marker, B., van de Geijn, R. A., Hammond, J. R. & Romero, N. A. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Software* (2012). To appear.

[52] Poulson, J., van de Geijn, R. & Bennighof, J. Parallel algorithms for reducing the generalized hermitian-definite eigenvalue problem. FLAME Working Note #56. Technical Report TR-11-05, The University of Texas at Austin, Department of Computer Sciences (2011).

[53] Prywes, N. S. Automatic generation of software systems. *SIGMIS Database* **6**, 7–17 (1974).

[54] Püschel, M. *et al.* SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"* **93**, 232– 275 (2005).

[55] Quintana-Ortí, E. S. & van de Geijn, R. A. Formal derivation of algorithms: The triangular Sylvester equation. *ACM Trans. Math. Software* **29**, 218–243 (2003).

[56] Sanderson, C. Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments. Tech. Rep., NICTA (2010).

[57] Smith, D. R. The design of divide and conquer algorithms. *Sci. Comput. Program.* **5**, 37–58 (1985).

[58] Smith, D. R. Readings in artificial intelligence and software engineering. chap. Top-down synthesis of divide-and-conquer algorithms, 35–61 (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1986).

[59] Smith, S. P. Differentiation of the Cholesky Algorithm. *Journal of Computational and Graphical Statistics* **4** (1995).

[60] uBLAS library, B. uBLAS online documentation. http://www.boost.org/doc/libs/1_3_0/libs/numeric/ublas/doc/index.htm.

[61] van de Geijn, R. A. & Quintana-Ortí, E. S. *The Science of Programming Matrix Computations* (`lulu.com`, 2008).

[62] Van Zee, F. G. `libflame`*: The Complete Reference* (`lulu.com`, 2009).

[63] Veldhuizen, T. Expression templates. *C++-Report* **7**, 26–31 (1995).

[64] Veldhuizen, T. L. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, 223–230 (Springer-Verlag, 1998).

[65] Vuduc, R. W. *Automatic performance tuning of sparse matrix kernels*. Ph.D. thesis, University of California, Berkeley, CA, USA (2004).

[66] Wegbreit, B. The synthesis of loop predicates. *Commun. ACM* **17**, 102–113 (1974).

[67] Whaley, R. C. *Automated Empirical Optimization of High Performance Floating Point Kernels*. Ph.D. thesis, Department of Computer Sciences, The Florida State University (2004).

[68] Whaley, R. C. & Dongarra, J. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing* (1998).

[69] Wilhelm, R. & Maurer, D. *Compiler Design* (Addison-Wesley, Wokingham, UK, 1995).

[70] Wolfram Research, Inc. *Mathematica, Version 8.0* (Champaign, IL, 2010).

[71] Yu, J. *et al.* A unified mixed-model method for association mapping that accounts for multiple levels of relatedness. *Nat. Genet.* **38**, 203–208 (2006).