A LINEAR ALGEBRA COMPILER FOR SMALL PROBLEM SIZES

A dissertation submitted to attain the degree of

DOCTOR OF SCIENCES OF ETH ZURICH
(Dr. sc. ETH Zurich)

by

DANIELE G. SPAMPINATO

Laurea Magistrale, Politecnico di Milano
Master i Teknologi / Sivilingeniør, Norwegian University of Science and Technology
born on 07.11.1984
citizen of Italy

accepted on the recommendation of

Prof. Markus Püschel (advisor)
Prof. Paolo Bientinesi
Prof. Albert Cohen
Prof. Franz Franchetti

2017
ABSTRACT

Many applications in media processing, control, graphics, and other domains require efficient small-scale linear algebra computations. However, most existing high performance libraries for linear algebra, such as ATLAS or Intel MKL are more geared towards large-scale problems (matrix sizes in the hundreds and larger), specific interfaces (e.g., BLAS and LAPACK), and cannot specialize to problem instances. In other domains, program generators have proven effective to automatically produce specialized code for important building blocks. An example is the SPIRAL generator for linear transforms and its extension to support other functionalities, such as matrix-matrix multiplication and Viterbi decoding. Learning from both libraries and generators, in this dissertation we aim for the automatic generation of fast code for small scale, linear algebra computations.

We introduce a program synthesis framework for small-scale, linear algebra computations of fixed size. These include computations on matrices, vectors, and scalars using basic operators as well as higher-level computations such as the Cholesky decomposition, solvers for the continuous-time Lyapunov and Sylvester equations, and the inversion of triangular matrices. The input to our framework is a linear algebra program composed of several fixed size basic and higher-level computations; the output is a corresponding C function optionally including intrinsics to efficiently use SIMD vector extensions.

In our framework, support for small-scale, basic linear algebra computations is provided by the LGEN compiler. LGEN generates code using two levels of mathematical domain-specific languages (DSLs). The DSLs are used to perform tiling, loop fusion, and vectorization at a high level of abstraction, before the final code is generated. In particular, LGEN’s vectorization approach can easily extend to different vector ISAs. In addition, search is used to select among alternative generated implementations.

LGEN also supports computations whose matrices have structure, such as symmetric or triangular, that reduces the cost of the computation. For example, dense linear systems of equations are solved by first reducing to triangular form and problems in optimization may yield matrices with different kinds of structures. The BLAS interface provides a small set of structured matrix computations, chosen to serve a certain set of higher-level functions supported by LAPACK. However, if a given computation contains a structure that is not supported by standard interfaces then its computation using a more general version of it (e.g., multiplying two general matrices instead of two structured ones) would lose the benefits of the structure. We address this problem by combining the LGEN compiler with techniques from polyhedral compilation to mathemati-
cally capture matrix structures. In this dissertation, we consider triangular and symmetric matrices and discuss the approach extensibility to a much larger set including blocked structures.

Finally, support for higher-level linear algebra computations is included building on LGEN and the FLAME-based algorithm synthesis tool CLick. When a higher-level computation is encountered in an input linear algebra program, a set of basic computations is synthetized starting from one of its algorithmic definitions. If desired, the framework’s autotuner explores alternative algorithmic variants to select the one that yields the best performance.

We evaluate the performance of the code generated with our framework for several linear algebra computations, including library compliant and noncompliant basic and higher-level computations and the computation of a Kalman filter. Experimental results show that our framework produces code that performs in many cases considerably better than well-established, commercial and non-commercial libraries, prior software generators, and compilers.
Le prestazioni di molte applicazioni in ambiti quali l’elaborazione dei segnali, i sistemi di controllo e la computer grafica dipendono dall’esecuzione efficiente di calcoli di algebra lineare con matrici di piccole dimensioni. Tuttavia la maggior parte delle librerie ad alte prestazioni per l’algebra lineare, come per esempio ATLAS e Intel MKL, sono progettate per problemi su larga scala, dove le matrici hanno almeno centinaia di righe e colonne. Esse presentano inoltre specifiche interfacce software, tipicamente BLAS e LAPACK, e sono difficilmente specializzate per particolari istanze di un problema. In alternativa alle librerie, i generatori di programmi sono rivelati mezzi efficaci per la produzione di importanti componenti software. Tra questi possiamo menzionare SPIRAL per le trasformate lineari e alcune sue estensioni per il supporto di moltiplicazioni tra matrici e la decodifica di Viterbi. Partendo dall’esperienza delle librerie e dei generatori, la presente dissertazione descrive alcuni progressi nella generazione automatica di codice per l’algebra lineare su piccola scala.

Presentiamo un framework per la sintesi di codice per calcoli algebra lineare su piccola scala con matrici di dimensioni fisse. Le espressioni supportate includono sia operazioni di base su matrici, vettori e scalari, sia operazioni di più alto livello, come la decomposizione di Cholesky, la soluzione delle equazioni tempo continue di Sylvester e Lyapunov e l’inversione di matrici triangolari. Il framework proposto riceve in input un programma composto da uno o più espressioni matriciali di dimensioni fisse. I calcoli espressi possono essere sia di base che di alto livello. L’output del framework consiste in una funzione C per il calcolo del programma dato in input. Opzionalmente, la funzione in output può fare uso di istruzioni SIMD per un miglior utilizzo delle unità vettoriali presenti in molti microprocessori moderni.

Il framework proposto introduce il supporto per calcoli di algebra lineare di base attraverso il compilatore LGEN. Quest’ultimo dispone di un processo di generazione del codice basato sull’uso di due linguaggi matematici. Questi linguaggi sono utilizzati per l’applicazione, ad un più alto livello di astrazione, di trasformazioni quali la ripartizione e la fusione di cicli e la vettorializzazione del codice. In particolare, l’approccio di vettorializzazione adottato da LGEN può essere esteso con facilità a diversi set d’istruzioni SIMD. Inoltre, dal momento che uno stesso programma matriciale può essere implementato in molti modi, LGEN esplora automaticamente varie alternative alla ricerca della versione migliore.

LGEN include anche il supporto per matrici strutturate, quali ad esempio matrici simmetriche e triangolari, comportando una riduzione del costo del calcolo. Per esempio, la soluzione di un sistema lineare con una matrice dei coefficienti
densa è normalmente ottenuta riducendo il problema ad una serie di sistemi triangolari, mentre problemi di ottimizzazione possono produrre matrici con svariate strutture. L’interfaccia BLAS dispone di un certo numero di calcoli con matrici strutturate per offrire supporto a funzioni di più alto livello contenute in LAPACK. Tuttavia, se un calcolo prevede l’uso di matrici strutturate non supportate dall’interfaccia, l’unica soluzione è quella di utilizzare una funzionalità più generica, con la conseguente perdita dei vantaggi recati dalla presenza delle strutture. Nella presente dissertazione, questo problema è affrontato estendendo LGen con un supporto per strutture matriciali grazie all’uso di tecniche tipiche del modello di compilazione poliedrale. Nel testo sono prese in considerazione matrici triangolari e simmetriche, descrivendo come l’approccio possa essere esteso ad un più vasto insieme di strutture comprensenti anche strutture a blocchi.

Infine, il framework proposto offre supporto per calcoli di algebra lineare di alto livello attraverso una soluzione che combina l’architettura di LGen con quella di ClicK, un tool per la sintesi automatica di algoritmi per l’algebra lineare basato su FLAME. Quando un programma in input contiene un calcolo di alto livello, questo viene scomposto in una serie di calcoli di base partendo da una delle sue possibili definizioni algoritmiche. Se richiesto, il framework esplora in modo automatico diverse varianti algoritmiche per selezionare quella con migliori prestazioni.

Valutiamo le prestazioni del codice generato con il framework proposto su una serie di calcoli di algebra lineare che include funzionalità sia di base che di alto livello, conformi e non alle interfacce BLAS e LAPACK, e il calcolo del filtro di Kalman. I risultati sperimentali ottenuti mostrano che le prestazioni del codice generato con il framework in esame sono in molti casi migliori di quelle ottenute con importanti librerie commerciali e non, generatori software preesistenti e compilatori.
I would like to use this page to thank all those people that directly and indirectly supported me throughout my doctoral studies. My gratitude goes first of all to my advisor Markus Püschel. He trusted and encouraged me in a very delicate moment of my life. With him I would also like to thank many past and present colleagues at the Department of Computer Science at ETH Zurich, in particular Victoria Caparrós Cabezas, Luca Della Toffola, Georg Ofenbeck, François Serre, Nikos Kyrtatas, Gagandeep Singh, Alen Stojanov, and Marcela Zuluaga: Brilliant colleagues and invaluable friends! Many thanks to Paolo Bientinesi and Diego Fabregat-Traver for the fruitful collaboration behind the system presented in Chapter 5. Thanks to Albert Cohen and Franz Franchetti for serving on my examination committee. Thanks to Petros Koumoutsakos and many of the past members of the CSE-lab, in particular Panagiotis Angelikopoulos, Basil Bayati, Mattia Gazzola, Sorban Georgescu, Babak Hejazialhosseini, Florian Milde, Diego Rossinelli, Gerardo Taurielo, and Wim van Rees. They were my first colleagues at ETH Zurich and I have learnt a lot from them. Special thanks to Daniele Foresti who brought me to Switzerland. We always manage to end up in the same country :) Thanks to all fellow Oerlikoners: Simone Bottan, Stefano Fusco, Luciana Marziotte, Matthias Leiß, and Sebastian Wex. Life in Zurich would not have been the same without such an awesome flat! I will never forget the JUCLAA group at the Claretian mission in Zurich, in particular Anthony Obikonu Igbokwe, Pilar Alonso, Ana Altair, Natalia Arenas, Ada Camey, Simon Foppa, Carolina Sonnekalb, Carlos Ugarte, and Alejandro Terova. Thanks to all the friends I could share my time with, both in Zurich and back in my hometown. Here I would like to mention in particular Franziska Baumgartner, Elena Castellote, Michela D'Amelio, Alessandra Di Lio, Serena Pierabella, Filippo Rossi, and Valerio Zerbi. Un particolare ringraziamento alla mia famiglia: i miei nonni, i miei genitori — Giuseppina Rota e Armando Spampinato — e mia sorella Beatrice. Grazie per il vostro amoroso sostegno! Thank you Laura for your love. We can truly say that our story grew stronger than the Alps! And this, I hope, is only the beginning.

Weeell...next!
CONTENTS

ABSTRACT iii

ACKNOWLEDGMENTS vii

1 INTRODUCTION 1
  1.1 Goal of This Dissertation 4
  1.2 Contributions of this Dissertation 6
  1.3 Related Work 9
    1.3.1 High-performance Linear Algebra Libraries 9
    1.3.2 Domain-specific Languages and Generators 10
    1.3.3 Optimizing Compilers 12
  1.4 Organization of this Dissertation 13

2 LINEAR ALGEBRA COMPUTATIONS 15
  2.1 Basic Linear Algebra Computations 15
    2.1.1 Notation and Basic Operators 16
    2.1.2 BLACs and BLAS 17
    2.1.3 Partitioning Matrices 19
    2.1.4 Structured Matrices 20
  2.2 Higher-level Linear Algebra Computations 22
    2.2.1 The FLAME Notation 23
    2.2.2 Triangular Systems 24
    2.2.3 The Inverse of a Triangular Matrix 26
    2.2.4 The Continuous-time, Triangular Sylvester and Lyapunov
        Equations 26
    2.2.5 The Cholesky Decomposition 29
  2.3 Computing Small Matrix-matrix Multiplications 29

3 A BASIC LINEAR ALGEBRA COMPILER 33
  3.1 Overview of the LGen Compiler 33
    3.1.1 LL: Low-level Linear Algebra Language 35
  3.2 Scalar Code Generation 35
    3.2.1 Input in LL 36
    3.2.2 Step 1: Tiling in LL 38
    3.2.3 Step 2: From LL to Σ-LL 39
    3.2.4 Step 3: Loop Transformations 42
    3.2.5 Step 4: From Σ-LL to C-IR 44
    3.2.6 Step 5: Code-level optimizations 45
3.2.7  Step 6: Performance Test and Autotuning  45
3.2.8  Summary of LGEN’s Choices During Code Generation  45
3.3  Vector Code Generation  45
  3.3.1  v-BLACs: Computational Building Blocks  46
  3.3.2  Load and Store Building Blocks  49
3.4  Relationship to Prior SPIRAL-like Attempts at Linear Algebra  54

4  A BASIC LINEAR ALGEBRA COMPILER FOR STRUCTURED MATRICES  57
  4.1  The Polyhedral Model  59
    4.1.1  Polyhedral Compilation Process  60
    4.1.2  Polyhedral Sets and Maps  61
  4.2  Code Generation with Structured Matrices  62
    4.2.1  Internal Representation of Structures in LGEN  62
    4.2.2  Scalar Code Generation  63
  4.3  Vectorization Approach  70
    4.3.1  Internal Representation of Tiled Structures  71
    4.3.2  Vector Code Generation  72
  4.4  Extensibility to New Structures  73

5  PROGRAM SYNTHESIS FOR LINEAR ALGEBRA  77
  5.1  Automatic Algorithm Discovery  78
    5.1.1  FLAME: Mechanical Derivation of Algorithms  78
    5.1.2  CLICK: FLAME Automated  81
  5.2  Linear Algebra Program Synthesis Overview  83
  5.3  Linear Algebra Program Synthesis Applied  88
    5.3.1  Vectorized Code Generation for the Cholesky Decomposition  88
    5.3.2  Vectorized Code Generation for the Kalman Filter  96

6  EXPERIMENTAL RESULTS  99
  6.1  Basic Linear Algebra Computations  99
    6.1.1  Targeting High-end Systems  100
    6.1.2  Targeting Embedded Systems  105
  6.2  Basic Linear Algebra Computations with structured Matrices  110
  6.3  Higher-level Linear Algebra Computations  112

7  CONCLUSIONS  119
  7.1  Current Limitations  120
  7.2  Future Directions  121

A  MATRIX-MATRIX MULTIPLICATION IN OL  123
  A.1  OL: The Operator Language  123
  A.2  Code Generation  125
B FLAME DERIVATION STEPS 131
INTRODUCTION

A significant part of processor time worldwide is spent on executing mathematical code in domains such as machine learning, control, communication, signal processing, graphics, or computer vision.

The mathematics used in these domains may differ widely, but the actual computations in the end often fall into the domain of linear algebra, meaning computations on matrices and vectors. The problem sizes and computers used range from the very large (e.g., simulations on a supercomputer or learning in the cloud) to the very small (e.g., a control system on an embedded processor). Both scenarios have in common the need for fast execution, for example, to save energy, to enable real-time, or to maximize output quality.

Current high-performance libraries and program generators for linear algebra have been mostly designed for large scale applications targeting computational science. They provide excellent performance for large problem sizes, but for smaller ones shortcomings exist: First, libraries are usually not highly optimized for small size computations even though they are needed in many applications; second, fixed input size computations that occur in many applications are usually not supported by specialized functions in libraries; third, many computations cannot always be directly mapped to the interface of existing library functions, thus causing overhead.

LIBRARY SUPPORT FOR SMALL VS LARGE SCALE COMPUTATIONS. For large-scale, dense linear algebra, commercial and non-commercial high-performance software exists, usually built around the Basic Linear Algebra Subprograms (BLAS) [26] and the Linear Algebra Package (LAPACK) [2] interfaces. For example, the Intel Math Kernel Library (MKL) [59] is a high-performance library that provides BLAS and LAPACK functions.

As an example, we consider the LAPACK function POTRF that computes the Cholesky decomposition of a symmetric positive-definite matrix. The Cholesky decomposition appears in several applications, including linear least squares, nonlinear optimizations, Monte Carlo simulations, and control systems. Figure 1.1(b) shows the percentage of peak performance attained by the POTRF function from Intel MKL for large matrices on a modern Intel processor with three levels of caches. By large we mean those matrices that fit into the last level
Figure 1.1: Percentage of peak performance (single thread) achieved computing a Cholesky decomposition (function POTRF from LAPACK) with small (a) vs large (b) matrix sizes on a modern Intel processor. The smallest large matrix ($n \times n$ with $n = 320$) fits in L3 cache, while the largest small matrix ($n = 124$) fits completely in the L2 cache of a single core. In (a) the performance of Intel MKL is compared with that achieved by straightforward, non-optimized code compiled with Intel icc (with vectorization enabled) and code generated with the framework introduced in this dissertation.

cache of the processor or exceed it. The performance of POTRF in this range is between 50% and 78% of the machine’s peak.

For smaller scale computations, however, the library’s performance degrades rapidly. Figure 1.1(a) shows the percentage of peak performance attained by Intel MKL for the same computation using smaller matrices, where the largest matrix still fits into the second level of cache of a single processor core. In this scenario, the performance of the library drops down to 25% of the machine’s peak for the largest size, which is less than half the performance obtained with the smallest test in Figure 1.1(b).

**Limitations of optimizing compilers.** In alternative to the libraries, one may choose to rely on optimizing compilers, but they can rarely match the performance obtained by high-performance libraries. Their limitations in performing domain-specific optimizations are well known [14, 77, 117, 121] with reasons that can vary from their inability to apply transformations at an algorithmic level to the fact that certain transformations are hard to prove legal in general. In our example, Figure 1.1(a) also shows the percentage of performance achieved by straightforward code with hardcoded sizes for the Cholesky decomposition compiled with Intel icc. The performance is in this case at most 12% of the machine’s peak, i.e., one fourth of the performance attained by the smallest test in Figure 1.1(b).
Table 1.1: Single iteration of a Kalman filter at time-step $k$. Matrices are denoted with upper case letters and vectors with lower case ones. The filter involves two phases: a predict phase and an update phase. In the predict phase, a new estimate $x_k$ of the system’s state (e.g., position and velocity of a car) is computed along with an error covariance matrix $P_k$ that measures its accuracy. In the update phase, the prediction is combined with the current observation information $z_k$ to refine the state estimate.

\[
\begin{align*}
\text{Predict:} & \\
& \begin{cases} 
  x_{k|k-1} = Fx_{k-1|k-1} + Bu \\
  P_{k|k-1} = FP_{k-1|k-1}F^T + Q 
\end{cases} \\
\text{Update:} & \\
& \begin{cases} 
  x_{k|k} = x_{k|k-1} + P_{k|k-1}H^T \\
  P_{k|k} = P_{k|k-1} - P_{k|k-1}H^T \times (HP_{k|k-1}H^T + R)^{-1}(z_k - Hx_{k|k-1}) 
\end{cases}
\end{align*}
\]

**Limitations of standard interfaces.** For a particular application, the needed linear algebra computations may not directly map to the standard interfaces provided by BLAS or LAPACK. For instance, consider the computation of a Kalman filter [97], which is widely used in the control of dynamic systems such as vehicles and robots. Table 1.1 shows a single iteration of a basic Kalman filter (i.e., limited to a linear model of a system).

The filter iteratively performs computations on matrices (upper case) and vectors (lower case): at every time step $k$, it estimates a vector $x_k$ of $n$ state parameters of the system (e.g., position and velocity of a car) and maintains a covariance matrix $P_k$ of the estimation error that is used to update the estimate with observations $z_k$ in (1.3). The notation $s|t$ in the subscript indicates an estimate (e.g., $x_{s|t}$) at time $s$ given observations of the system up to time $t$. The size of the involved matrices depends on the dimension of the estimated state $n$ which typically is a small (fixed) value.

Decomposing small size computations into a set of standard BLAS or LAPACK functions may lead to suboptimal computations. For instance, the matrices $P_{k|k-1}$ and $Q$ in (1.2) are both symmetric, which could result in a reduced operation count in computing (1.2) compared to using a generic algorithm (i.e., using $3n^3$ floating point operations as opposed to $4n^3$ as explained in Figure 1.2). However, using BLAS functions requires a split of the computation (1.2) into two subtasks that fail to recognize the symmetry of the resulting matrix.

**Computing on embedded systems.** The performance gap between small- and large-scale computations is also noticeable in embedded systems. Embedded processors are used in automotive electronics, network devices, smartphones...
and tablets, and many other ubiquitous systems with a reduced power budget. However, their energy efficiency comes at a price: a reduced set of resources. Examples include in-order execution units, smaller numbers of instruction-issue ports, and less efficient access to unaligned memory locations. Because of these, additional optimizations are needed to obtain highest performance, and hand-tuning for these processors is even more common than on their high-end counterparts.

In summary, neither existing high-performance libraries, nor general purpose optimizing compilers provide a sufficient solution to the problem of providing high performance for smaller scale linear algebra computations. However, we believe that an effective answer should combine their complementary strengths for this domain: The detailed domain-specific optimizations of libraries and the generality of compilers.

1.1 GOAL OF THIS DISSERTATION

In an ideal world, a programmer would express a linear algebra computation as presented in a book from the application domain, and a compiler, or program

---

1 In the specific case of Intel MKL, the (non-standard) gemmt function can be used in lieu of gemm to update only the lower or the upper half of the output matrix.
generator, would produce code that is precisely specialized to this computation and highly optimized for the target processor. In this dissertation, we want to make progress towards this goal.

Our goal is to develop a domain-specific framework for the synthesis of fast code for linear algebra computations of small, fixed size using as an input only their mathematical description and knowledge of the instruction set architecture (ISA) of the target processor.

By “small”, we mean that the working set (including all input and output matrices and vectors) of the computation fits into the local caches of a single microprocessor core. The thesis behind our goal is that a domain-specific generator is an effective solution to the problem of synthesizing fast code for small-scale, linear algebra computations for three major reasons:

(i) Flexibility: A mathematical language enables the natural and concise description of a large class of linear algebra computations.

(ii) Performance: By being domain-specific, the generator can apply all needed optimizations, including those of mathematical nature, that are crucial to achieving high-performance on modern microprocessors.

(iii) Portability: With a proper design, a generator can offer both performance and code portability across different hardware architectures.

We achieve our goal through a sequence of intermediate steps of increasing scope. To define them, we consider as prototypical example the Kalman filter in Table 1.1. It shows four categories of linear algebra computations illustrated in Figure 1.3:

- **Basic linear algebra computations (BLACs):** Computations on matrices, vectors, and scalars using basic operators: multiplication, addition, and transposition. For example, (1.1) in Table 1.1.

- **BLACs with structured matrices (sBLACs):** For example, in (1.2) the matrices $P_*$ and $Q$ are symmetric. Mathematically, sBLACs include the computations supported by the BLAS interface with the only exception of triangular solvers which we include in our next category.

- **Higher-level computations:** LAPACK-level computations, such as the Cholesky decomposition and solvers for linear systems required to efficiently compute the expressions involving inverses in (1.3) and (1.4). Higher-level computations are normally implemented in terms of sBLACs.

- **Linear algebra computations:** The outer scope, containing computations, such as the Kalman filter in Table 1.1, composed of several subcomputations from the previous three categories.
Figure 1.3: Four classes of linear algebra computations used in this dissertation: Basic linear algebra computations (BLACs, e.g., $x = Fx + By$), BLACs with structured matrices (sBLACs, e.g., $P = FPF^T$ where $P$ is symmetric), higher-level computations (normally implemented in terms of sBLACs, e.g., the Cholesky decomposition $U^TU = P$ where matrix $U$ is upper triangular and $P$ is symmetric, positive definite), and linear algebra computations, such as the Kalman filter in Table 1.1, defined using computations from the previous three categories.

Figure 1.4: Synthesis framework for small-scale, linear algebra computations of fixed size.

The four categories above enable the execution of a large class of linear algebra computations. In the next section we explain in more details the specific contributions made to enable their support.

1.2 CONTRIBUTIONS OF THIS DISSERTATION

The main contribution of this dissertation is a framework for the synthesis of efficient code for small-scale, linear algebra computations of fixed size using information from the ISA of the target microarchitecture. The framework is illustrated in Figure 1.4. The input to our framework is a linear algebra computation, such as the Kalman filter in Table 1.1, expressed as a linear algebra program, i.e., using a sequence of BLACs, sBLACs, and higher-level functionalities. The output is a C function that computes it, optionally vectorized using SIMD intrinsics. The knowledge of the ISA comes in terms of parameters, e.g., the length...
of the floating point vector register, and ISA-specific building blocks required to generate the final output code.

Implementing our framework required to build support for BLACs, sBLACs, and higher-level functionalities, which resulted in the five main contributions presented next.

**Supporting BLACs.**

1. We present a novel approach to generating efficient code for small-scale, basic linear algebra computations of fixed size, and its implementation in the LGEN compiler [70, 102].

A high-level illustration of our approach is provided in Figure 1.5. The input to LGEN is a fixed-size, basic linear algebra expression; the output is a corresponding C function optionally including intrinsics to efficiently use SIMD vector extensions. LGEN generates code using two levels of mathematical DSLs. The DSLs are used to perform tiling, loop fusion, and vectorization at a high level of abstraction, before the final code is generated. In addition, search is used to select among alternative generated implementations. Our approach closely follows the one of the SPIRAL [90, 91] code generator for transforms.

2. We introduce two new mathematical DSLs for describing basic linear algebra computations [102].

The low-level linear algebra language (LL) and its Σ-LL extension, to include summations in the language, are MATLAB-like DSLs that build on ideas from the Operator Language (OL) [33], a previous attempt at linear algebra used in the SPIRAL generator [90, 91], and the Hierarchically Tiled Arrays (HTAs) [51]. LL and Σ-LL allow the concise and purely mathematical representations of both computations and data accesses in a BLAC.

3. We present a vectorization methodology that is easily portable to new vector architectures [70, 102].

The vectorization methodology introduced in LGEN is based on the description of an input BLAC in terms of a small number of efficiently vectorized building blocks for computation and data access. Porting LGEN to a different architecture only requires their reimplementation based on a new target ISA.

**Supporting sBLACs.**
4. We propose an extensible methodology for the generation of optimized code for small scale BLACs with structured matrices [103].

We extend the LGen approach with techniques from polyhedral compilation [9, 111] to mathematically capture matrix structures. The methodology is extensible to include a large set of possible matrix structures. In this dissertation, we use lower triangular, upper triangular, and symmetric matrices as prototypical examples.

SUPPORTING HIGHER-LEVEL COMPUTATIONS.

5. We propose an automatic synthesis approach that encompasses both algorithmic discovery and code generation, which enables code generation for small-scale, higher-level computations.

We combine the LGen approach with an extension of the FLAME-based algorithm synthesis tool Click [13, 29, 30]. This new combined methodology enables the exploration of different algorithmic descriptions of an input higher-level computation. Every algorithmic variant can be formulated as an LL program (i.e., a program only composed of sBLACs) and compiled into optimized C code, optionally vectorized with intrinsics.
EVALUATION. We compare the performance of code generated using our approach against several commercial and non-commercial libraries, software generators, and optimizing compilers. In particular, for every category of linear algebra computations illustrated in Figure 1.3, we generate code for three different groups of tests:

- **Library-compliant**: Computations that can be computed with a single BLAS (in the case of BLACs and sBLACs) or LAPACK (in the case of higher-level computations) function call.

- **Almost library-compliant**: sBLACs or higher-level computations supported by BLAS or LAPACK but that are not specialized to some of the structures appearing in the input.

- **Non library-compliant**: sBLACs or higher-level computations that must be implemented using multiple BLAS or LAPACK functions.

We generate code targeting four different vector ISAs: Intel Supplemental Streaming SIMD Extensions 3 (SSSE3), Intel Streaming SIMD Extensions 4 (SSE4), Intel Advanced Vector Extensions (AVX), and the ARM NEON extension. In particular, we generate code for BLACs targeting SSSE3, SSE4, and NEON, while we target AVX when generating code for sBLACs and higher-level computations.

1.3 RELATED WORK

The optimization of linear algebra and related functions has been an active research topic for decades. In the following we review prior related work organized along three dimensions: High-performance linear algebra libraries, generators and DSLs, and optimizing compilers.

1.3.1 High-performance Linear Algebra Libraries

Several highly optimized, handwritten BLAS and LAPACK libraries [26, 27, 74] for large-scale problems exist including OpenBLAS [92, 119] (which is based on GotoBLAS [41, 43]), libFLAME [108], a library derived using the FLAME approach [13, 49], and vendor libraries, such as the Intel MKL [59] and the IBM Engineering and Scientific Subroutine Library (ESSL) [56]. ReLAPACK [87] exploits recursion to implement high-performance LAPACK computations. The BLAS-like Library Instantiation Software (BLIS) [109] is a framework for instantiating a set of functions larger than BLAS from a set of microkernels.

The performance of the libraries above can be suboptimal for smaller problem sizes compared to what is achievable (e.g., [100] and this dissertation) and the interface they provide may not match a desired computation. For this reason, the Intel Integrated Performance Primitives (IPP) [57] includes a section called
Intel IPP MX devoted to small scale linear algebra operations with a non-BLAS interface. The latest Intel MKL has introduced fast general matrix multiplication (GEMM) kernels on small matrices in double precision as well as a specific functionality for batches of matrices with same parameters, such as size and leading dimensions (GEMM\_BATCH). On the same direction, recent proposals are trying to extend the BLAS interface to include batched routines. IPP, MKL, and ReLAPACK will be among our benchmarks.

1.3.2 Domain-specific Languages and Generators

The libraries mentioned above are implemented and optimized by hand. Various approaches have worked on automation.

Linear algebra generators. Among the earliest efforts are the Portable High Performance ANSI C (PHiPAC) [14] and the Automatically Tuned Linear Algebra Software (ATLAS) [117] generators, which iteratively tune implementation parameters, such as block size and loops order, using the runtime as feedback (autotuning). Alternative tuning methodologies include machine learning techniques [122]. A model-based approach to completely avoid a tuning phase is available in the context of both ATLAS [121] and BLIS [76].

These approaches are typically geared towards BLAS and large sizes. The recent LIBXSMM library [1, 54] provides an assembly-level code generator specifically for small dense and sparse matrix-matrix multiplication and convolutions on Intel platforms. Its scope is however limited to the latter two functionalities.

Algorithmic synthesis. The Formal Linear Algebra Methods Environment (FLAME) [49] provides a methodology for automatically deriving algorithms for higher level linear algebra functions [13] given as mathematical equations. The supported functions are mostly those covered by the LAPACK library and the generated algorithms rely on the availability of a BLAS library. The methodology is completely automated by the CL\text{\textcopyright}k compiler [29, 30], and an extended version of it is considered as a fronted for higher-level computations in Chapter 5.

Dsl-based approaches. The CLAK compiler [31] finds efficient mappings of matrix equations onto building blocks from high-performance libraries such as BLAS and LAPACK. DxTer [78] transforms blocked algorithms such as those generated by CL\text{\textcopyright}k and applies transformations and refinements to output high-performance distributed-memory implementations. Both CLAK and DxTer do not generate BLAS or BLAC functions, with or without structures.

LINVIEW [83] is a framework for incremental maintenance of analytical queries expressed in terms of linear algebra programs. The goal of the system is to prop-
agate within a (large) computation only the changes caused by (small) variations
in the input matrices.

The Build to Order BLAS (BTO) [11, 101] is a domain-specific compiler for
matrix computations. BTO is not bound to the BLAS interface and optimizes for
loop fusion, data partitioning, and parallelism using autotuning. BTO focuses on
memory bound computations (BLAS 1-2 operations) and relies on a compiler for
vectorization.

The Vehicle for Optimized Basic Linear Algebra (VOBLA) [10] is a linear alge-
bra DSL with the goal of generating high-performance OpenCL code for BLAS
functionalities targeting GPUs. The DSL describes linear algebra computations
using basic operators and array access patterns. Access patterns are used to
separate matrix structures from storage formats. VOBLA applies optimizations
at a lower level of abstraction than the mathematical one, making its approach
rather different from ours. VOBLA is first translated into PENCIL [5] code, a
C99-based intermediate representation, and only at this point the Polyhedral
Parallel Code Generator (PPCG) [112] is used to optimize the computation and
generate parallel GPU code.

**TEMPLATE-BASED APPROACHES.** A different generative approach is adopted
by Eigen [48], uBLAS [116], and the Matrix Template Library (MTL) [45] among
others. They use C++ expression templates to optimize the code at compile time.
Optimizations include loop fusion, unrolling, and SIMD vectorization [44]. How-
ever, an approach based on C++ metaprogramming lacks autotuning capabilities
and requires a non-negligible effort in extensibility. Another approach based
on metaprogramming is taken by the Hierarchically Tiled Arrays (HTAs) [51],
which offer data types with the ability to dynamically partition matrices and
vectors, automatically handling situations of overlapping areas. HTAs priority,
however, is to improve programmability reducing the amount of code required
to handle tiling and data distribution in parallel programs, leaving any optimiza-
tion to the programmer (or program generator).

**GENERATORS IN OTHER DOMAINS.** Other program generators were devel-
oped for signal processing. Genfft [37] generates the small size FFTs (codelets)
needed in FFTW [38]. SpirAL [90, 91] is a high-performance library generator for
linear transforms, such as the discrete Fourier transform [113] and the Walsh-
Hadamard transform [61]. Further studies [21] have shown that the SpirAL
approach can be extended beyond the domain of transforms to support other
applications including general matrix-matrix multiplication [33, 110], Viterbi
decoding [22], and synthetic aperture radar [81]. SpirAL uses domain-specific
languages (DSLs) for optimizations such as loop merging [36] and vectoriza-
tion [35]. A more detailed analysis of the extension of SpirAL to support matrix-
matrix multiplication is provided in Section 3.4. Our work aims to build a func-
tionality similar to genfft in [37] for linear algebra using an approach similar to Spiral.

Other examples of DSL-based approaches are query [99] and stencil [52, 55, 93] compilers. The former provides a new methodology based on a stack of DSLs with progressive lowering of abstractions, while the latter aim at simplifying reasoning on complex reductions and data-dependent access patterns typical of stencil programs.

**DSL Frameworks for Performance.** Finally, frameworks for the development of DSLs for performance include PetaBricks [3], which allows for algorithmic autotuning, and systems based on multi-staging techniques, such as frameworks based on Lightweight Modular Staging (LMS) [86, 96, 105] and Terra [25].

### 1.3.3 Optimizing Compilers

A third approach to improve the performance of linear algebra code is the use of optimizing compilers.

**Polyhedral Tools.** Optimizing compilers based on the polyhedral model [32] reschedule computation and data accesses to enhance locality and expose parallelization and vectorization opportunities. They can perform loop transformations for imperfectly nested loops such as parallelization, fusion, and reordering [15, 16, 89], multi-level tiling [53, 66], and vectorization [47, 67].

In this work we use polyhedral tools, i.e., isl [111] and CLooG [9], to manipulate structures in sBLACs at a higher level of abstraction.

**Auto-vectorization.** Multiplatform vectorization techniques such as those in [84, 85] use abstract SIMD representations making optimizations such as alignment detection portable across different architectures.

Other recent vectorization techniques include approaches for loops exhibiting little loop-level parallelism [6, 8, 123]. All these apply code transformations to make code portions with partial SIMD parallelism, such as computations on complex data layouts, amenable to vectorization.

Whole-function vectorization methodologies are available for data-parallel languages such as OpenCL [65] and Intel ispc [88] as well as for C programs [94].

**Superoptimization.** Superoptimizers target small portions of code with the aim of finding a shorter and faster sequence of instructions with the same output. The original approach in [79] takes a small, loop-free codelet as an input and performs exhaustive search in the space of valid instruction sequences. Several research directions have explored alternative methodologies including the
exclusive use of equality-preserving transformations [63], formulating the problem as a stochastic search [98], and considering domain-specific transformations at higher level of abstraction than assembly instructions [80].

The scope of these and other optimizing compilers and techniques is more general than LGen’s. However, such generality often misses out on optimization opportunities [77]. In our context, we can take advantage of the specific domain to synthesize vectorized C code of higher performance.

1.4 ORGANIZATION OF THIS DISSERTATION

This dissertation is organized around the four categories of computations mentioned in Section 1.1. In Chapter 2 we review the required mathematical concepts and notation to describe the three categories shown in Figure 1.3, i.e., BLACs, sBLACs, and higher-level computations. The framework required to generate code for the latter categories is then introduced. In particular, Chapter 3 presents LGen for BLACs, including its DSLs and vectorization methodology. Chapter 4 explains the extension required to support structures. Chapter 5 describes our methodology to synthetize code for higher-level computations, thus enabling support for complex linear algebra computations such as the Kalman filter previously discussed. In Chapter 6 we show experimental results. Finally, we conclude in Chapter 7 discussing current limitations of our approach and future directions.
LINEAR ALGEBRA COMPUTATIONS

In this chapter we introduce notation and background for linear algebra computations that will be the focus of our work.

In Section 2.1 we present basic linear algebra computations, i.e., computations over matrices, vectors, and scalars expressed using only basic operators, such as addition and multiplication. In addition, we introduce notation for matrix tiling, an operation that creates matrices of matrices. Tiling is essential in high-performance linear algebra algorithms.

Matrices can have structure, such as being symmetric or triangular. These structures influence the way algorithms are constructed. For instance, the algorithm for a lower triangular matrix-vector multiplication may require less operations than the one involving a full matrix.

In Section 2.2 we introduce linear algebra computations that operate at a higher-level, such as solvers of linear systems and matrix decompositions. For these computations there are usually several algorithmic variants defined in terms of basic linear algebra computations over tiled input and output matrices.

In this dissertation we are interested in small scale computations. For this reason, we conclude this chapter discussing in Section 2.3 as an example the structure of an efficient, small matrix-matrix multiplication. The techniques used there are fundamental to enable fast code for a large class of linear algebra computations.

2.1 BASIC LINEAR ALGEBRA COMPUTATIONS

In this dissertation we refer to basic linear algebra computations (BLACs\(^1\)) as computations over real matrices, vectors and scalars using four fundamental operators: scalar multiplication, matrix multiplication and addition, and transposition. In the remainder of this section we introduce the notation for matrices and operators and provide simple algorithms for some important computations.

---

\(^1\) Not to be confused with the Netlib’s Basic Linear Algebra Communication Subprograms (BLACS).
2.1.1 Notation and Basic Operators

We represent full matrices with upper case letters (e.g., \(A, B, \ldots\)), vectors with lower case letters (e.g., \(a, b, \ldots\)), and scalars with Greek lower case letters (e.g., \(\alpha, \beta, \ldots\)). An \(m \times n\) real matrix \(A \in \mathbb{R}^{m \times n}\) and a real (column) vector \(x \in \mathbb{R}^{m}\) of length \(m\) can be described by listing their scalar entries

\[
A = (\alpha_{i,j}) = \begin{pmatrix}
\alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,n-1} \\
\alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,n-1} \\
\vdots & \vdots & \ddots & \vdots \\
\alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,n-1}
\end{pmatrix},
\]

\[
x = \begin{pmatrix}
\chi_0 \\
\chi_1 \\
\vdots \\
\chi_{m-1}
\end{pmatrix},
\]

where \(i, j\) indicates a position at row \(i\) and column \(j\). When convenient, we may use the MATLAB-like notation \(A(i, j) = \alpha_{i,j}\).

**Basic Operators.** We consider four basic operators on matrices.

Transposition:

\[
()^T : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{n \times m},
\]

\[
C = (\gamma_{i,j}) = (A)^T = A^T, \text{ where } \gamma_{i,j} = \alpha_{j,i}.
\]

Addition:

\[
+: \mathbb{R}^{m \times n} \times \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{m \times n},
\]

\[
C = A + B, \text{ where } \gamma_{i,j} = \alpha_{i,j} + \beta_{i,j}.
\]

Scalar-matrix multiplication:

\[
\cdot : \mathbb{R} \times \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{m \times n},
\]

\[
C = \alpha \cdot B = \alpha B, \text{ where } \gamma_{i,j} = \alpha \beta_{i,j}.
\]

Matrix-matrix multiplication:

\[
\cdot : \mathbb{R}^{m \times k} \times \mathbb{R}^{k \times n} \rightarrow \mathbb{R}^{m \times n}, \quad (2.1)
\]

\[
C = A \cdot B = AB, \text{ where } \gamma_{i,j} = \sum_{\ell=0}^{k-1} \alpha_{i,\ell} \beta_{\ell,j}.
\]

The above operations extend to vectors by choosing the column dimension \(n = 1\). For example, matrix-vector multiplication is a special case of (2.1):

\[
y = Ax, \quad y \in \mathbb{R}^m, x \in \mathbb{R}^k, A \in \mathbb{R}^{m \times k},
\]
and we use transposition to refer to row vectors:

\[ x^T \in \mathbb{R}^{1 \times n} \iff x^T = (x_0 \ x_1 \ \cdots \ x_{n-1}) \].

Any valid combination of the four basic operators produces a BLAC, e.g., \( \alpha = x^T A y \) and \( D = \alpha (A + B)^T C + \beta x y^T \). The output is on the left-hand side and can also appear as an input on the right-hand side. In addition to the four basic operators on matrices and vectors, we also allow for auxiliary scalar operations, such as divisions or square roots.

2.1.2 BLACs and BLAS

Starting from 1979 a subset of BLACs was identified as building block for the development of high-performance linear algebra libraries. The Basic Linear Algebra Subprograms (BLAS) specification \([26, 27, 74]\) was designed to provide a standard FORTRAN (and later also C) interface for libraries that would provide their computation. The BLAS is divided into three categories of computations, called levels, that are associated to the asymptotic runtime cost of the BLACs they contain. This cost is usually quantified in terms of floating point operations (flops). Next we list some of the most relevant computations supported by the BLAS and provide an algorithmic formulation of prototypical examples from each level.

BLAS 1. The BLAS level 1 \([74]\) provides BLACs that require \( \mathcal{O}(n) \) flops on \( \mathcal{O}(n) \) data. Some computations at this level are shown in Table 2.1 and straightforward algorithms are provided in Algorithms 2.1–2.2.

BLAS 2. The BLAS level 2 \([27]\) provides BLACs that compute \( \mathcal{O}(n^2) \) flops on \( \mathcal{O}(n^2) \) data. Table 2.1 shows prototypical computations from this level. Their
Algorithm 2.1 (AXPY) \( y = \alpha x + y, \quad \alpha \in \mathbb{R}, \ x,y \in \mathbb{R}^n. \) Cost = 2n flops.

\[
\text{for } i = 0 : n - 1 \text{ do} \\
\quad y(i) := \alpha x(i) + y(i) \\
\text{end for}
\]

Algorithm 2.2 (DOT) \( \delta = x^T y, \quad \delta \in \mathbb{R}, \ x,y \in \mathbb{R}^n. \) Cost = 2n flops.

\[
\delta = 0 \\
\text{for } i = 0 : n - 1 \text{ do} \\
\quad \delta := \delta + x(i)y(i) \\
\text{end for}
\]

corresponding straightforward algorithms are shown in Algorithms 2.3 and 2.4, respectively.

Algorithm 2.3 (GEMV) \( y = Ax + y, \quad x \in \mathbb{R}^n, \ y \in \mathbb{R}^m, \ A \in \mathbb{R}^{m \times n}. \) Cost = 2mn flops.

\[
\text{for } i = 0 : m - 1 \text{ do} \\
\quad \text{for } j = 0 : n - 1 \text{ do} \\
\quad\quad y(i) := A(i,j)x(j) + y(i) \\
\quad \text{end for} \\
\text{end for}
\]

Algorithm 2.4 (GER) \( A = A + xy^T, \quad x \in \mathbb{R}^n, \ y \in \mathbb{R}^m, \ A \in \mathbb{R}^{m \times n}. \) Cost = 2mn flops.

\[
\text{for } i = 0 : m - 1 \text{ do} \\
\quad \text{for } j = 0 : n - 1 \text{ do} \\
\quad\quad A(i,j) := A(i,j) + x(i)y(j) \\
\quad \text{end for} \\
\text{end for}
\]

Finally, the BLACs supported by the level 3 BLAS [26] compute \( O(n^3) \) flops on \( O(n^2) \) data. Table 2.1 shows the most important computation at this level: The general matrix-matrix multiplication update (GEMM). Algorithm 2.5 shows a possible way to compute it.

It is known that a fast implementation of GEMM enables high-performance level 3 BLAS [39, 42, 50, 64], and with it any other computation that can be expressed using it, such as the higher-level computations discussed in the next section. The reason is that, on modern microarchitectures with deep memory hierarchies, GEMM benefits from its high operational intensity, i.e., the ratio of flops (i.e., \( O(n^3) \)), to the amount of data transferred between main memory and the processor, (i.e., \( \Omega(n^2) \) bytes). This means that, if computation is performed efficiently, GEMM can achieve a performance close to the processor’s peak.
Algorithm 2.5 (GEMM) $C = AB + C$, $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$, $C \in \mathbb{R}^{m \times n}$.

Cost $= 2mnk$ flops.

\[ \begin{array}{ll}
\text{for } i = 0 : m - 1 \text{ do} \\
& \text{for } j = 0 : n - 1 \text{ do} \\
& & \text{for } l = 0 : k - 1 \text{ do} \\
& & & C(i, j) := A(i, l)B(l, j) + C(i, j) \\
& & \text{end for} \\
& \text{end for} \\
\text{end for} \\
\end{array} \]

2.1.3 Partitioning Matrices

We can partition a matrix into smaller matrices, effectively writing it as a matrix of matrices. For instance, consider the following $2 \times 3$ matrix:

\[ A = \begin{pmatrix}
\alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2} \\
\alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2}
\end{pmatrix}. \quad (2.2) \]

Using partitioning, we can describe $A$ in various ways. For example:

\[ A = \begin{pmatrix}
\alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2} \\
\alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2}
\end{pmatrix} = \begin{pmatrix}
a_0^T \\
a_1^T
\end{pmatrix}, \quad (2.3) \]

\[ A = \begin{pmatrix}
\alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2} \\
\alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2}
\end{pmatrix} = \begin{pmatrix}
a_0 & a_1 & a_2
\end{pmatrix}, \quad (2.4) \]

\[ A = \begin{pmatrix}
\alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2} \\
\alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2}
\end{pmatrix} = \begin{pmatrix}
A_{0,0} & a_{0,1}
\end{pmatrix}. \quad (2.5) \]

We may refer to the submatrices in a partition using the parenthesis operator extended with a MATLAB-like colon notation. For example, in (2.3), $A(0,:) = a_0^T$, in (2.4) $A(:,0) = a_0$, and in (2.5) $A(0:2,0:2) = A_{0,0}$. In the remainder of this dissertation we will distinguish between partitioning and tiling. With partitioning we will refer in general to the creation of matrix of matrices. However, whenever this operation is performed creating submatrices of homogeneous size (except for possible leftover rows and columns) we will refer to it as tiling and call the resulting submatrices tiles. For instance, in the following two examples, only the second partitioning is a tiling:

\[ B = \begin{pmatrix}
\beta_{0,0} & \beta_{0,1} & \beta_{0,2} & \beta_{0,3} & \beta_{0,4} & \beta_{0,5} \\
\beta_{1,0} & \beta_{1,1} & \beta_{1,2} & \beta_{1,3} & \beta_{1,4} & \beta_{1,5}
\end{pmatrix}, \quad (2.6) \]

\[ B = \begin{pmatrix}
\beta_{0,0} & \beta_{0,1} & \beta_{0,2} & \beta_{0,3} & \beta_{0,4} & \beta_{0,5} \\
\beta_{1,0} & \beta_{1,1} & \beta_{1,2} & \beta_{1,3} & \beta_{1,4} & \beta_{1,5}
\end{pmatrix}. \]
We refer to a tiling operation using the notation $[B]_{p,q}$ meaning that we create tiles of size $p \times q$ starting from element $B(0,0)$. For example, we describe (2.6) more compactly as $B = [B]_{2,2} = (B_{i,j})$, where $B_{i,j} \in \mathbb{R}^{2\times 2}$.

2.1.4 Structured Matrices

We also consider BLACs whose matrices have a structure. For example the following matrix $U \in \mathbb{R}^{n\times n}$ is upper triangular:

$$U = (u_{i,j}) = \begin{pmatrix} u_{0,0} & u_{0,1} & \cdots & u_{0,n-1} \\ 0 & u_{1,1} & \cdots & u_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{n-1,n-1} \end{pmatrix}.$$ (2.7)

In general we write an upper triangular matrix as $U$ and denote its structure with $U_n$ or simply $U$, depending whether we need or not to specify the dimension:

$$U \in U_n \Leftrightarrow U \in \mathbb{R}^{n\times n} \text{ and (2.7) holds.}$$

When structures occur in a BLAC they should be exploited to reduce its cost. For example, if we assume that both matrices $A$ and $B$ in Algorithm 2.5 are upper triangular we could reformulate it to require roughly one-sixth the number of flops, as shown in Algorithm 2.6.

**Algorithm 2.6** Multiplication of two upper triangular matrices:

$$C = U_0 U_1 + C, \quad U_0, U_1 \in U_n, \quad C \in \mathbb{R}^{n\times n}. \text{ Cost } \approx \frac{n^3}{3} \text{ flops.}$$

```plaintext
for i = 0 : n do
    for j = i : n do
        for l = i : j do
            C(i, j) := U_0(i, l)U_1(l, j) + C(i, j)
        end for
    end for
end for
```
2.1 Basic Linear Algebra Computations

A = \begin{pmatrix}
\alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2} \\
\alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2}
\end{pmatrix}

Row-major order, \text{ld}(A) = 3

\begin{pmatrix}
\alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2} \\
\alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2}
\end{pmatrix}

Row-major order, \text{ld}(A) = 4

Figure 2.1: Two possible full storage schemes with different leading dimensions for a 2 × 3 matrix A stored in row-major order.

Other structures we consider in this dissertation include lower triangular and symmetric matrices:

\begin{align*}
L \in L_n & \iff L \in \mathbb{R}^{n \times n}, L = (\lambda_{i,j}) = \\
& \begin{pmatrix}
\lambda_{0,0} & 0 & \cdots & 0 \\
\lambda_{1,0} & \lambda_{1,1} & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
\lambda_{n-1,0} & \lambda_{n-1,1} & \cdots & \lambda_{n-1,n-1}
\end{pmatrix}, \\
S \in S_n & \iff S \in \mathbb{R}^{n \times n}, S = (\sigma_{i,j}) = \\
& \begin{pmatrix}
\sigma_{0,0} & \sigma_{0,1} & \cdots & \sigma_{0,n-1} \\
\sigma_{0,1} & \sigma_{1,1} & \cdots & \sigma_{1,n-1} \\
\vdots & \vdots & \ddots & \vdots \\
\sigma_{n-1,0} & \sigma_{n-1,1} & \cdots & \sigma_{n-1,n-1}
\end{pmatrix}.
\end{align*}

We refer to BLACs with structured matrices as sBLACs. The BLAS interface supports certain sBLACs at level 2 and 3 as shown in Table 2.2. The first two characters in a BLAS label refer to the structure a computation is specialized for (e.g., GE for general in Table 2.1, TR for triangular, and SY for symmetric in Table 2.2).

Storage schemes. Associated with matrices are different storage schemes that determine how they are stored in main memory. The full storage scheme is the most common one, where every element in a matrix has a corresponding location in a memory array. For example, the 2 × 3 matrix in (2.2) requires an array of length at least six. Elements can be stored in row-major (i.e., traversing the matrix row by row: \( \alpha_{0,0}, \alpha_{0,1}, \alpha_{0,2}, \alpha_{1,0}, \ldots \)) or column-major (i.e., column by column traversal: \( \alpha_{0,0}, \alpha_{1,0}, \alpha_{0,1}, \alpha_{1,1}, \ldots \)) order. The distance between the beginning of two subsequent rows (or columns in column-major order) within a matrix \( M \) is referred to as its leading dimension, in short \( \text{ld}(M) \). It may coincide with the length of its rows (or columns) but may also be larger than that, e.g., when accessing a matrix within a matrix. Figure 2.1 shows two examples of full storage schemes with different leading dimensions for (2.2). For instance, assuming that (2.2) is fully stored in row-major order, \( \text{ld}(A) = \text{ld}(A(0 : 2, 0 : 2)) = 3 \).
The leading dimension is needed to compute the location of a matrix element \( A(i, j) \) within its array storage using the formula \( \text{pos}(A(i, j)) = \text{ld}(A) \cdot i + j \) in case of row-major order or \( \text{pos}(A(i, j)) = \text{ld}(A) \cdot j + i \) in case of column-major order. For example, considering matrix (2.2) \( \text{pos}(A(1, \theta)) = \text{ld}(A) \cdot 1 + 0 = 3 \). All BLAS functions support leading dimensions for their matrices. For example, the following is the complete C++ BLAS interface for computing GEMM in double precision:

```cpp
void cblas_dgemm (CBLAS_ORDER order,
                  CBLAS_TRANSPOSE transA, CBLAS_TRANSPOSE transB,
                  int m, int n, int k,
                  double alpha, double *A, int lda,
                  double *B, int ldb,
                  double beta, double *C, int ldc);
```

where `order` is `RowMajor` or `ColMajor`, `transA` and `transB` define if matrices A and B are to be transposed or not, and `lda`, `ldb`, and `ldc` are the leading dimensions of the matrices A, B, and C, respectively.

For structured matrices often also the full storage scheme is used, with the common convention that half of the elements are not accessed (e.g., the zero elements of a triangular matrix or the elements in the lower (or upper) half of a symmetric matrix). Since this implies waste, they could be stored more compactly using a packed storage scheme where no memory is associated to elements that should not be accessed thus saving half of the space. The BLAS allows the use of alternative storages when possible. For example, the functions SPMV and TPMV from BLAS 2 compute exactly the same sBLACs as SYMV and TRMV in Table 2.2, but with the assumption that the matrices S and T are packed.

### 2.2 Higher-level Linear Algebra Computations

With higher-level linear algebra computations we mean algorithms expressed in terms of BLACs or sBLACs. A prominent collection of such computations is the Linear Algebra Package (LAPACK) \([2]\). LAPACK computations are built on top of BLAS functions and can be grouped into four major categories: solvers for linear systems of equations, matrix decompositions, eigenvalue problems, and linear least squares. In this text we consider computations from only two of them: Solvers for linear systems and matrix decompositions. In the remainder of this section we introduce algorithmic notation (Section 2.2.1) and discuss in more details five computations: A solver for triangular systems (Section 2.2.2), the inverse of a triangular matrix (Section 2.2.3), solvers for the triangular, continuous-time Sylvester and Lyapunov equations (Section 2.2.4), and the Cholesky decomposition (Section 2.2.5).
2.2.1 *The FLAME Notation*

In the previous section, we described straightforward sBLAC algorithms using a notation where accesses were represented as functions on loop indices. In this section, as algorithms for higher-level functions grow in complexity, we adopt the notation defined by the Formal Linear Algebra Methods Environment (FLAME) [49], a methodology that enables a systematic derivation of correct algorithms for a certain class of linear algebra computations. FLAME will be thoroughly discussed in Chapter 5. Here, we are only interested in its notation.

The FLAME notation provides a loop-based representation of a computation on partitioned matrices and vectors using an algorithmic worksheet as the one shown in Algorithm 2.7. This algorithm computes a matrix-matrix multiplication using an outer product update as a building block. For comparison, Algorithm 2.7 is analogous to the index-based Algorithm 2.8. The FLAME notation is free of loop indices and traverses the input and output matrices based on an initial partitioning decision. Matrices can be partitioned either horizontally, vertically, or both. For instance, in Algorithm 2.7, line (a), the matrices $A$ and $B$ are partitioned horizontally and vertically respectively, while the output matrix $C$ is always updated entirely. The resulting partitions are labelled as (L)eft and (R)ight if partitioned horizontally, and (T)op and (B)ottom if partitioned vertically. Inside the loop, the partitions are updated at every iteration. They are first refined in (c) to isolate the vectors required to compute the update in (d) (also called the update statement). Note that the notation differentiates assignment ($:=$) from equality ($=$). After the computation of the the update statement, the thick lines in the repartitions are moved forward in (e) making new portions of the matrices available for the next iteration. The algorithm terminates as soon as the matrices are completely traversed (condition in (b)).
Algorithm 2.7 Matrix multiplication: outer product version.
\[ C = AB + C, \quad A \in \mathbb{R}^{m \times k}, \quad B \in \mathbb{R}^{k \times n}, \quad C \in \mathbb{R}^{m \times n}. \]
The algorithm is given in FLAME notation. Cost = 2mkn flops.

(a) Partition \( A \rightarrow (A_L \mid A_R) \), \( B \rightarrow \left( \begin{array}{c} B_T \\ B_B \end{array} \right) \)
where \( A_L \) is \( m \times 0 \) and \( B_T \) is \( 0 \times n \)

(b) while \( \text{size}(A_L) < \text{size}(A) \) do

(c) Repartition
\[
\begin{array}{c}
(A_L \mid A_R) \rightarrow (A_0 \mid a_1 \mid A_2), \\
\left( \begin{array}{c} B_T \\ B_B \end{array} \right) \rightarrow \left( \begin{array}{c} B_0 \\ b_1 \\ B_2 \end{array} \right)
\end{array}
\]
where \( a_1 \) is \( m \times 1 \), and \( b_1 \) is \( 1 \times n \)

(d) \( C := C + a_1 b_1^T \)

(e) Continue with
\[
\begin{array}{c}
(A_L \mid A_R) \leftarrow (A_0 \mid a_1 \mid A_2), \\
\left( \begin{array}{c} B_T \\ B_B \end{array} \right) \leftarrow \left( \begin{array}{c} B_0 \\ b_1 \\ B_2 \end{array} \right)
\end{array}
\]
endwhile

Algorithm 2.8 Matrix multiplication: Outer product version analogous to Algorithm 2.7. \( C = AB + C, \quad A \in \mathbb{R}^{m \times k}, \quad B \in \mathbb{R}^{k \times n}, \quad C \in \mathbb{R}^{m \times n}. \]

\[
\begin{array}{l}
\text{for } i = 0 : k - 1 \text{ do} \\
\quad C := C + A(:, i)B(i,:)
\end{array}
\]
end for

In the remainder of this section, we will use the above notation to describe five higher-level computations, i.e., triangular systems of equations, the continuous-time triangular Sylvester and Lyapunov equations, the triangular inverse, and the Cholesky decomposition, assuming the availability of sBLACs as building blocks.

2.2.2 Triangular Systems

Solving a triangular system of equations comes in two variants:

\[ TX = B, \quad T \in \mathcal{L}_n \cup \mathcal{U}_n, \quad X, B \in \mathbb{R}^{n \times m}, \tag{2.8} \]
\[ XT = B, \quad T \in \mathcal{L}_n \cup \mathcal{U}_n, \quad X, B \in \mathbb{R}^{m \times n}, \tag{2.9} \]
where \( X \) is unknown output matrix. If the triangular matrices in (2.8)–(2.9) are invertible, the two solutions could be computed as \( X = T^{-1}B \) and \( X = B T^{-1} \), respectively. In general with linear matrix equations, the inverse is seldom explicitly computed. In the case of triangular systems the use of forward and backward substitution for respectively lower and upper triangular matrices are a better choice both in terms of efficiency and accuracy. An example of forward substitution is shown in Algorithm 2.9. At every iteration the algorithm updates \( B_B \)

\[
\text{Algorithm 2.9 Solving a lower triangular system (Forward substitution algorithm).}
\]

\[ L X = B, \quad L \in \mathcal{L}_n, \quad X, B \in \mathbb{R}^{n \times m}. \]

\( X \) overwrites \( B \).

Cost \( \approx n^2 m \) flops.

There are two functions for the solution of triangular systems of equations: \text{TRSM} for problems with multiple vectors on the right-hand side (i.e., \( m > 1 \) in (2.8) and (2.9)), and \text{TRSV} for problems with a single vector on the right-hand side (i.e., \( m = 1 \) in (2.8) and (2.9)). In both cases the input matrix \( B \) is overwritten with the result \( X \).

Solvers for triangular systems are part of the BLAS because they are used to implement linear solvers for unstructured matrices. However, we do not consider them sBLACs as they require an inverse operator, which is not a basic operator.
2.2.3 The Inverse of a Triangular Matrix

Sometimes it is necessary to explicitly compute the inverse of a matrix. For instance, in the context of least square estimation, matrix inversion is used to compute the covariance matrix of the estimators of linear regression parameters [106]. LAPACK provides functions for inverting general, symmetric and triangular matrices. We will focus on triangular matrices and Algorithm 2.10 gives a possible algorithm to invert lower triangular ones. It requires computing a reciprocal (line (i)), two scalar products (lines (ii) and (iv)), and a rank-1 update (line (iii)).

Algorithm 2.10 Inverting a lower triangular matrix.
$L = L^{-1}$, $L \in \mathcal{L}_n$.
Cost $\approx n^3/3$ flops.

$$
\text{Partition } L \rightarrow \begin{pmatrix}
L_{TL} & 0 \\
L_{BL} & L_{BR}
\end{pmatrix}
$$

where $L_{TL}$ is $0 \times 0$

while size($L_{TL}$) < size($L$) do

Repartition

$$
\begin{pmatrix}
L_{TL} & 0 \\
L_{BL} & L_{BR}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
L_{0,0} & 0 & 0 \\
L_{1,0}^T & \lambda_{1,1} & 0 \\
L_{2,0} & L_{2,1} & L_{2,2}
\end{pmatrix}
$$

where $\lambda_{1,1}$ is $1 \times 1$

(i) $\lambda_{1,1} := 1/\lambda_{1,1}$
(ii) $l_{2,1} := -\lambda_{1,1}l_{2,1}$
(iii) $L_{2,0} := L_{2,0} + l_{2,1}l_{1,0}^T$
(iv) $l_{1,0}^T := \lambda_{1,1}l_{1,0}^T$

Continue with

$$
\begin{pmatrix}
L_{TL} & 0 \\
L_{BL} & L_{BR}
\end{pmatrix}
\leftarrow
\begin{pmatrix}
L_{0,0} & 0 & 0 \\
L_{1,0}^T & \lambda_{1,1} & 0 \\
L_{2,0} & L_{2,1} & L_{2,2}
\end{pmatrix}
$$

endwhile

2.2.4 The Continuous-time, Triangular Sylvester and Lyapunov Equations

Two other linear solvers that we consider solve the continuous-time, triangular Sylvester and Lyapunov equations, which are important routines in control the-
ory applications \([12, 20]\). The triangular Sylvester equation can be formulated as

\[
T_0 X + X T_1 = C, \quad T_0 \in \mathcal{L}_m \cup \mathcal{U}_m, \quad T_1 \in \mathcal{L}_n \cup \mathcal{U}_n, \quad X, C \in \mathbb{R}^{m \times n},
\]

(2.10)

where \(X\) is the unknown output matrix. The Bartels-Stewart algorithm \([7, 40]\) for computing the solution \(X\) is shown in Algorithm 2.11 for the case where \(T_0 \in \mathcal{L}, \quad T_1 \in \mathcal{U}\), and \(X\) overwrites \(C\). LAPACK provides a function TRSYL that solves the Sylvester equation for real quasitriangular matrices.

**Algorithm 2.11** Solving the triangular Sylvester equation (Bartels-Stewart Algorithm).

\(LX + XU = C\), \(L \in \mathcal{L}_m\), \(U \in \mathcal{U}_n\), and \(X, C \in \mathbb{R}^{m \times n}\). \(X\) overwrites \(C\).

Cost \(\approx (m + n)mn\) flops.

---

**Partition** \(C \rightarrow (C_L \mid C_R)\), \(U \rightarrow \begin{pmatrix} U_{TL} & U_{TR} \\ 0 & U_{BR} \end{pmatrix}\),

where \(C_L\) is \(m \times 0\) and \(U_{TL}\) is \(0 \times 0\).

while \(\text{size}(C_L) < \text{size}(C)\) do

**Repartition**

\[
\begin{pmatrix} C_L \mid C_R \end{pmatrix} \rightarrow \begin{pmatrix} C_0 \mid c_1 \mid C_2 \end{pmatrix},
\]

\[
\begin{pmatrix} U_{TL} & U_{TR} \\ 0 & U_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} U_{0,0} & u_{0,1} & U_{0,2} \\ 0 & v_{1,1} & u_{1,2} \\ 0 & 0 & U_{2,2} \end{pmatrix}
\]

where \(c_1\) is \(m \times 1\) and \(v_{1,1}\) is \(1 \times 1\).

\(c_1 := c_1 - C_0 u_{0,1}\)

\(c_1 := (L + v_{1,1} I_m)^{-1} c_1\) (solve with, e.g., Algorithm 2.9)

Continue with

\[
\begin{pmatrix} C_L \mid C_R \end{pmatrix} \rightarrow \begin{pmatrix} C_0 \mid c_1 \mid C_2 \end{pmatrix},
\]

\[
\begin{pmatrix} U_{TL} & U_{TR} \\ 0 & U_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} U_{0,0} & u_{0,1} & U_{0,2} \\ 0 & v_{1,1} & u_{1,2} \\ 0 & 0 & U_{2,2} \end{pmatrix}
\]

endwhile

---

The triangular Lyapunov equation is a specialized version of the Sylvester equation (2.10), where a single triangular matrix appears in the equation and the right-hand side is symmetric:

\[
TX + XT^T = S, \quad T \in \mathcal{L}_n \cup \mathcal{U}_n, \quad X, S \in S_n.
\]
The triangular Lyapunov equation could also be solved using Algorithm 2.11; however, more specialized algorithms such as the one in Algorithm 2.12 can reduce the number of flops by a half.

Algorithm 2.12 Solving the triangular Lyapunov equation.

\[
LX + XL^T = S, \quad L \in \mathcal{L}_n \text{ and } X, S \in \mathbb{S}_n. \text{ X overwrites } S.
\]

Cost \( \approx n^3 \) flops.

\[
\begin{pmatrix}
L_{TL} & 0 \\
L_{BL} & L_{BR}
\end{pmatrix}, \quad \begin{pmatrix}
S_{TL} & S_{TR} \\
S_{BL} & S_{BR}
\end{pmatrix}
\]

where \( L_{TL} \) is \( 0 \times 0 \) and \( S_{TL} \) is \( 0 \times 0 \)

while size(\( L_{TL} \)) < size(\( L \)) do

Repartition

\[
\begin{pmatrix}
L_{0,0} & 0 \\
\lambda_{1,1} & 0 \\
\end{pmatrix}, \quad \begin{pmatrix}
S_{0,0} & s_{0,1} \\
\sigma_{1,1} & s_{1,2} \\
\end{pmatrix}
\]

where \( \lambda_{1,1} \) and \( \sigma_{1,1} \) are \( 1 \times 1 \)

\[
s_{1,0}^T := s_{1,0}^T - \lambda_{1,0} s_{0,0},
\]

\[
s_{1,0} := s_{1,0}^T (L_{0,0} + \lambda_{1,1} I_m)^{-1} \quad (m = \text{col}(L_{0,0}))
\]

\[
\sigma_{1,1} := \sigma_{1,1} - s_{1,0}^T (\lambda_{1,0}^T - \lambda_{1,0} s_{1,0}) T
\]

\[
\sigma_{1,1} := \sigma_{1,1} / (2 \lambda_{1,1})
\]

Continue with

\[
\begin{pmatrix}
L_{0,0} & 0 \\
\lambda_{1,1} & 0 \\
\end{pmatrix}, \quad \begin{pmatrix}
S_{0,0} & s_{0,1} \\
\sigma_{1,1} & s_{1,2} \\
\end{pmatrix}
\]

endwhile

LAPACK does not provide functions for solving the Lyapunov equation. However, other solutions such as RECSY [62], a library for solving triangular Sylvester-type equations, provide a specialized alternative. RECSY provides two different functions, i.e., RECSYCT and RECLYCT, that target specifically the triangular Sylvester and Lyapunov equations, respectively.
2.2.5 The Cholesky Decomposition

Matrix decomposition is another important category of higher-level functions we consider. The goal of matrix decomposition is to factorize a matrix into a product of two or more matrices.

For example, when solving $Ax = b$ for $x$, with $A$ general and invertible, usually first the LU decomposition of $A$ is computed, which factors it into two triangular matrices. In this way, $x$ can be obtained by solving two triangular systems as discussed in Section 2.2.2 instead of computing the less efficient $A^{-1}b$.

If the matrix $A$ is symmetric, positive-definite (SPD), its Cholesky decomposition is computed instead. The Cholesky decomposition is used to factor an SPD matrix into the product of a triangular matrix by its transpose. This could be expressed by either of the two following computations where the real matrix $P$ is SPD:

\[
U^T U = P, \quad U \in \mathbb{U}_n, \\
L L^T = P, \quad L \in \mathbb{L}_n,
\]

where $U$ and $L$ are the outputs of the decompositions. The LAPACK’s POTRF function computes the Cholesky decompositions, overwriting half of the input SPD matrix with the output triangular matrix. Algorithm 2.13 shows one possible approach to compute (2.11).

2.3 Computing Small Matrix-Matrix Multiplications

In Section 2.1, we mentioned that a fast matrix-matrix multiplication enables fast level 3 sBLACs and higher-level computations. In this last section, we describe in greater detail the optimization of

\[
C = AB + C,
\]

with $A \in \mathbb{R}^{m \times k}$, and $B \in \mathbb{R}^{k \times n}$, targeting a single core microprocessor. The size of the computation is small, meaning that all matrices fit into the microprocessor’s L1 cache.

Consider (2.12) and Algorithm 2.5 that computes it. To take advantage of temporal and spatial locality, an efficient implementation of GEMM needs to be defined in terms of tiled matrices that fit into high levels of the memory hierarchy [82]. Based on our assumption all matrices fit in L1. As a consequence we want to apply tiling to take advantage of registers.
Algorithm 2.13 The Cholesky decomposition.
\( U^T U = P, \quad U \in \mathbb{U}_n, \) and \( P \) is SPD.
\( U \) overwrites the upper half of \( P \). Cost \( \approx n^3 / 3 \) flops.

Partition \( P \rightarrow \begin{pmatrix} P_{TL} & P_{TR} \\ P_{BL} & P_{BR} \end{pmatrix} \)
where \( P_{TL} \) is \( 0 \times 0 \)
while size(\( P_{TL} \)) < size(\( P \)) do
  Repartition
  \[
  \begin{pmatrix} P_{TL} & P_{TR} \\ P_{BL} & P_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} P_{0,0} & p_{0,1} & P_{0,2} \\ P_{1,0} & p_{1,1} & p_{1,2} \\ P_{2,0} & p_{2,1} & P_{2,2} \end{pmatrix}
  \]
  where \( p_{1,1} \) is \( 1 \times 1 \)

\[
\begin{align*}
\pi_{1,1} &:= \pi_{1,1} - p_{0,1}^T p_{0,1} \\
\pi_{1,1} &:= \sqrt{\pi_{1,1}} \\
p_{1,2} &:= p_{1,2} - p_{0,1}^T p_{0,2} \\
p_{1,2} &:= (1 / \pi_{1,1}) p_{1,2}
\end{align*}
\]
Continue with
\[
\begin{pmatrix} P_{TL} & P_{TR} \\ P_{BL} & P_{BR} \end{pmatrix} \leftrightarrow \begin{pmatrix} P_{0,0} & p_{0,1} & P_{0,2} \\ P_{1,0} & p_{1,1} & p_{1,2} \\ P_{2,0} & p_{2,1} & P_{2,2} \end{pmatrix}
\]
endwhile

TILING FOR REGISTERS. Algorithm 2.14 applies loop tiling [118] to compute (2.12) using a smaller matrix multiplication on tiles of matrices \( A, B, \) and \( C \).

Algorithm 2.14 \( C = AB + C \) (tiled version 0).
\( A \in \mathbb{R}^{m \times k}, \ B \in \mathbb{R}^{k \times n}, \) and \( C \in \mathbb{R}^{m \times n} \).
Parameters \( m_r, n_r, \) and \( k_c \) are tile sizes and divide \( m, n, \) and \( k \) respectively; index intervals are denoted with \( p_q = (p : p + q) \) with \( p, q \in \{(i, m_r), (j, n_r), (l, k_c)\} \).
Cost = \( 2mnk \) flops.

\[
\text{for } i = 0 : m_r : m \text{ do} \\
\quad \text{for } j = 0 : n_r : n \text{ do} \\
\quad\quad \text{for } l = 0 : k_c : k \text{ do} \\
\quad\quad\quad C(i, m_r, j, n_r) := A(i, m_r, l, k_c) B(l, k_c, j, n_r) + C(i, m_r, j, n_r)
\quad\quad \end{for}
\quad \end{for}
\]
An efficient choice is to maintain in register an \( m_r \times n_r \) micro-tile of \( C \) with the minimum necessary to compute it. Their values and that of \( k_c \) are related to the structure of the innermost multiplication in Algorithm 2.14.

**Taking advantage of ILP.** The \( m_r \times n_r \) elements of the micro-tile of \( C \) can all be updated independently. This suggests the use of a rank-1 update for the micro-tile yielding the structure in Algorithm 2.15.

**Algorithm 2.15** \( C = AB + C \) (tiled version 1).

\[ A \in \mathbb{R}^{m \times k}, \ B \in \mathbb{R}^{k \times n}, \text{ and } C \in \mathbb{R}^{m \times n}. \]

The innermost computation multiplication is computed as a sequence of micro-rank-1 updates.

Cost = 2mnk flops.

\[
\text{for } i = 0 : m_r \text{ do}
\]

\[
\text{for } j = 0 : n_r \text{ do}
\]

\[
\text{for } l = 0 : k_c \text{ do}
\]

\[
\text{for } p = 0 : k_c \text{ do}
\]

\[ C(i m_r, j n_r) := A(i m_r, l + p)B(l + p, j n_r) + C(i m_r, j n_r) \]

\[
\text{end for}
\]

\[
\text{end for}
\]

\[
\text{end for}
\]

\[
\text{end for}
\]

The new innermost multiplication is exactly the micro-kernel identified by the BLIS approach [109] or analogously the micro-MMM in [121]. This choice also gives us a range of values for the tiling parameters.

Following [121], a good choice is to maintain in register a micro-tile \( C(i m_r, j n_r) \), a column \( A(i m_r, l + p) \), a row \( B(l + p, j n_r) \), and the t temporary elements required to hide the latency of multiplication before its result can be added to \( C \):

\[ m_r n_r + m_r + n_r + t \leq n_{\text{reg}}, \quad (2.13) \]

where \( n_{\text{reg}} \) is the number of available floating point registers. The choice of \( k_c \) is not related to the register file. As the loop around the micro-multiplication could be unrolled to reduce loop overhead, the L1-I cache is the only bound on the choice of \( k_c \).

The remaining step is the efficient implementation of the micro-multiplication in Algorithm 2.15.

**Taking advantage of SIMD parallelism.** Efficiently implementing the micro-multiplication in Algorithm 2.15 on a \( v \)-way vector microarchitecture relies on ISA-specific solutions, making this task a performance-portability problem. Typically this micro-computation is implemented separately by an expert developer for each target ISA and packed into libraries [109] and autotuners [117].
Examples of vector ISAs from the Intel x86 family of architectures are the Streaming SIMD Extensions (SSE) and the Advanced Vector Extensions (AVX), while from the ARM family the NEON advanced SIMD extension.

The available instructions are normally defined in the assembly language of the architecture. For instance, the following assembly instruction computes a pointwise multiplication between two vector registers $\text{ymm0}$ and $\text{ymm1}$ in AVX saving the result in $\text{ymm2}$:

$\text{vmulpd ymm2, ymm0, ymm1}$

There $\text{ymm0}$, $\text{ymm1}$, and $\text{ymm2}$ are 256-bit wide registers each holding four doubles. In alternative to the assembly, vector instructions can also be accessed directly in higher-level languages like C or C++ by means of intrinsics. For instance, the multiplication above could be expressed using intrinsics as follows

```c
__m256d v0, v1, v2;
...
v2 = __mm_mul_pd(v0, v1);
```

where $\text{__m256d}$ is a data type that maps the vector of doubles $v0$–$v2$ to 256-bit registers.

Intrinsics can be mapped to one assembly instruction (like in the multiplication example above) or more. More information on intrinsics for both Intel and ARM architectures are available in [4, 58].

In the next chapters we introduce a domain-specific framework for linear algebra computations that translates BLACs, sBLACs, and higher-level computations into optimized C code, optionally vectorized with intrinsics. The approach behind our code generator includes techniques similar to the ones shown above for matrix multiplication.
A BASIC LINEAR ALGEBRA COMPILER

In this chapter, we present LGEN, a compiler for small basic linear algebra computations (BLACs) on operands with fixed size. BLACs were introduced in Section 2.1.1 and involve general matrices, vectors and scalars. An extension of LGEN to handle structured matrices is described later in Chapter 4. As stated in the introduction our focus are small computations, meaning that the total size of all matrices and vectors in a given BLAC fit into the last level of cache local to a single core (typically L1 or L2 cache).

The design of LGEN follows closely the one of the Spiral code generator for the different domain of linear transforms. Spiral was discussed in Section 1.3 together with other prior work extending the Spiral framework beyond the domain of linear transforms.

We start with explaining the general structure of LGEN in Section 3.1. In Section 3.2, we discuss in greater detail the steps taken by LGEN to generate the scalar code, using matrix-matrix multiplication as running example. How this same approach can be used to produce efficient vectorized code is shown in Section 3.3. In Section 3.4, we conclude this chapter comparing LGEN’s approach with that of a prior Spiral-based attempt at generating matrix-matrix multiplication.

3.1 OVERVIEW OF THE LGEN COMPILER

In this section we present the structure of LGEN illustrated in Figure 3.1. It presents a Spiral-like design characterized by:

- **Multiple DSLs:** A domain-specific representation makes it easier to define and apply automatically transformation rules that yield code-level optimizations. Spiral shows that this approach is particularly beneficial when two mathematical DSLs are used to manipulate the input transform both at an algorithmic and implementation level before translating the computation into a C-like representation.

- **Iterative code generation:** Different algorithmic formulations and implementation choices typically yield different performance results. Both Spiral
and LGEN compile and evaluate the performance of all generated C functions, searching for the best generated code.

The input to LGEN is a BLAC as specified in Section 2.1.1 in what we call low-level linear algebra language (LL). In Figure 3.1, Step 1, a tiling strategy is chosen. This is done by annotating the BLAC with a tile size (e.g., \(3 \times 2\)), which is then propagated to derive the tile sizes of all in- and outputs. Hierarchical tiling is possible. Next in Step 2, the resulting LL expression is converted to another language called \(\Sigma\)-LL that is still based on linear algebra but makes access patterns and loops explicit. At this level, loop merging and possible loop exchanges are performed (Step 3). Next, a C-intermediate representation (C-IR) is generated (Step 4) to perform loop unrolling, scalar replacement, and conversion into static single assignment (SSA) form (Step 5). Finally, the C function is generated; its performance is used in an autotuning feedback loop (Step 6).

If vectorized code (with intrinsics) is desired, the vector length \(v\) of the ISA is an input together with the BLAC and impacts the tiling decision in Step 1. The generated code uses a small set of pre-implemented building blocks, called \(v\)-BLACs, in a process further explained in Section 3.3.

Next we introduce the notation of LGEN’s frontend language and discuss in more details a compilation example.
3.1.1 LL: Low-level Linear Algebra Language

The Low-level Linear Algebra Language (LL) is a mathematical DSL with a MATLAB-like syntax. An LL statement can be used to express a BLAC (as defined in Section 2.1.1) on general matrices. Example of BLACs captured by LL include BLAS computations (e.g., the rank-1 update $A = A + \alpha xy^T$) and others (e.g., $\alpha = x^T Ay$).

**The Tiling Operator.** LL also includes a tiling operator in addition to the four basic operators (i.e., transposition, addition, scalar- and matrix-matrix multiplication). It captures the concept of multilevel tiling following ideas from the hierarchically tiled array (HTA) type [51]. Tiling is denoted using the same notation as introduced for matrices in Section 2.1.3, namely as

$$[\text{op}]_{r,c},$$  \hspace{1cm} (3.1)

where op is any LL statement, expression, or quantity (i.e., matrix, vector, and scalar) and $r$ and $c$ are positive integers. A simple example is tiling a plain or level-0 matrix of size $m \times n$ as $[A]_{r,c}$. This creates a level-1 tiled matrix of dimension $\left[\frac{m}{r}\right] \times \left[\frac{n}{c}\right]$ where every element is an $r \times c$ matrix. The resulting matrix is homogeneous if $r|m$ and $c|n$ or heterogeneous if tiles at the borders have a different size.

Tiling an operation in LL (including an assignment) means tiling the implicit output matrix that it computes. For example, the statement $C = AB + C$ where $C \in \mathbb{R}^{m \times n}$ has an implicit output $M$ in all similar to $C$, and the level-1 tiling

$$[C = AB + C]_{r,c}$$  \hspace{1cm} (3.2)

can be thought as applied to $M$.

The grammar of an LL statement is described by the grammar in Table 3.1. Since there is a straightforward mapping between statements and BLACs\(^1\) we will often stick to the mathematical notation in our discussion unless needed otherwise.

We now describe step by step the program generation process (Figure 3.1) for scalar (non-vectorized) C code.

3.2 Scalar Code Generation

For the following explanations, we use a GEMM computation of the form

$$C = AB + C$$  \hspace{1cm} (3.3)

\(^1\) With the only exception of $=$, which is not the mathematical equivalence but the assignment operator.
Table 3.1: Grammar of an LL statement in Backus-Naur form. The subscript of a non-terminal is a non-negative level of tiling (i.e., \( \ell \geq 1 \) for \( \langle \text{tiled-stmt} \rangle_\ell \) and \( \langle \text{tiled-expr} \rangle_\ell \), while \( \ell \geq 0 \) elsewhere). A statement or expression at level zero is plain, i.e., not tiled. The positive integers \( r \) and \( c \) are respectively the number of rows and columns of a tile at level \( \ell \).

\[
\langle \text{ll-stmt} \rangle_\ell ::= \langle \text{tiled-stmt} \rangle_\ell \mid \langle \text{quantity} \rangle_\ell = \langle \text{ll-expr} \rangle_\ell
\]

\[
\langle \text{ll-expr} \rangle_\ell ::= \langle \text{tiled-expr} \rangle_\ell \mid ( \langle \text{ll-expr} \rangle_\ell ) \mid \langle \text{basic-op} \rangle_\ell
\]

\[
\langle \text{tiled-stmt} \rangle_\ell ::= [ \langle \text{ll-stmt} \rangle_{\ell-1} ]_{r,c}
\]

\[
\langle \text{tiled-expr} \rangle_\ell ::= [ \langle \text{ll-expr} \rangle_{\ell-1} ]_{r,c}
\]

\[
\langle \text{basic-op} \rangle_\ell ::= \langle \text{ll-expr} \rangle_\ell + \langle \text{ll-expr} \rangle_\ell
\]
\[
\langle \text{ll-expr} \rangle_\ell \ast \langle \text{ll-expr} \rangle_\ell
\]
\[
\langle \text{ll-expr} \rangle_\ell^T
\]

\[
\langle \text{quantity} \rangle_\ell ::= \langle \text{matrix} \rangle_\ell \mid \langle \text{vector} \rangle_\ell \mid \langle \text{scalar} \rangle_\ell
\]

A : Matrix(5, 9);
B : Matrix(9, 5);
C : Matrix(5, 5);

C = A*B + C;

Figure 3.2: Example of LL program for the BLAC in (3.3).

as running example. We will consider different sizes depending on the details to be illustrated.

3.2.1 Input in LL

The input to LGEn is an LL program and a set of compilation options. An LL program is composed of a set of quantity declarations followed by a level-0 LL statement for which we want to generate fast code. The grammar of an LL program is shown in Table 3.2, and Figure 3.2 shows the LL program for (3.3) for some choice of fixed sizes. The set of compilation options contains information such as floating point precision (i.e., float or double), vectorization option (i.e., enabled or not), and parameters from the model of the hardware platform (e.g., number of registers, ISA, vector register length, etc.).

INTERNAL REPRESENTATION. First, the input is parsed into an expression tree as shown in Figure 3.3. We implemented a type system that determines the implicit output matrix associated with every operator in the tree (empty square nodes) and makes sure that the expression is well-formed (e.g., that dimensions
Table 3.2: Grammar of an LL program in Extended Backus-Naur form. The rule for an LL statement $\langle ll-stmt \rangle_0$ is given in Table 3.1. The values $r$ and $c$ are positive integers.

$$\langle ll-prog \rangle ::= \langle decl \rangle \langle ll-stmt \rangle_0;$$

$$\langle decl \rangle ::= \langle decl \rangle \{ \langle decl \rangle \} \mid \langle name-list \rangle : ( \langle mat-decl \rangle \mid \langle vec-decl \rangle \mid \langle sca-decl \rangle )$$

$$\langle mat-decl \rangle ::= \text{Matrix}(r, c);$$

$$\langle vec-decl \rangle ::= \text{Vector}(r);$$

$$\langle sca-decl \rangle ::= \text{Scalar};$$

Figure 3.3: LGEn’s internal representation of BLAC (3.3) where $A \in \mathbb{R}^{3\times2}$ and $B \in \mathbb{R}^{2\times4}$. Square nodes are input and output matrices, while empty square nodes are implicit matrices derived for every operator in the tree. Gray boxes are mathematical and physical layouts. A mathematical layout is defined for every matrix appearing in the tree while physical layouts are initially available only for inputs and outputs.

match). Explicit matrices in the tree are associated to two descriptors that we call mathematical and physical layouts. The mathematical layout of a matrix describes its mathematical properties, while its physical layout how the matrix is represented in memory. The physical layout is initially void for all implicit matrices at this point of the compilation. Physical layouts of implicit matrices will appear later at Step 4, after the last mathematical transformations are applied.
Table 3.3: Rewrite rules to propagate tiling decisions; \( \langle e_L \rangle \), \( \langle e_R \rangle \), and \( \langle e \rangle \) are LL expressions. The call \( \text{cols}(\langle e \rangle) \) returns the number of columns of \( \langle e \rangle \)'s implicit matrix.

\[
\begin{align*}
\langle e_L \rangle &= \langle e_R \rangle \\
\langle e_L \rangle + \langle e_R \rangle &= \langle e_L \rangle + \langle e_R \rangle \\
[\langle \text{scalar} \rangle \cdot \langle e \rangle]_{r,c} &= [\langle \text{scalar} \rangle]_{1,1} \cdot \langle e \rangle]_{r,c} \\
\langle e_L \rangle \cdot \langle e_R \rangle &= \langle e_L \rangle \cdot \langle e_R \rangle \\
[\langle e \rangle^T]_{r,c} &= \langle e \rangle]_c^T 
\end{align*}
\]

3.2.2 Step 1: Tiling in LL

Tiling is a crucial locality optimization in linear algebra for all levels of the memory hierarchy as discussed in Section 2.3. Thus, the first step in LGEn (Figure 3.1) is to fix a tiling strategy (the search will then be able to explore different ones), which has to be done in a way that works for all BLACs.

For example, the first tiling decision on \((3.3)\) may be \((3.2)\) with \(r = 2\) and \(c = 2\) yielding \( [C = AB + C]_{2,2} \). In the next step, this top-level tiling decision is propagated down the expression tree to derive the associated tiling decisions for the in- and output matrices. In our example, assume that \(A, B \in \mathbb{R}^{4 \times 4}\). This is done using rewriting with the rules shown in Table 3.3:

\[
[C = AB + C]_{2,2} \rightarrow [C]_{2,2} = [AB + C]_{2,2} \\
\quad \rightarrow [C]_{2,2} = [AB]_{2,2} + [C]_{2,2} \\
\quad \rightarrow [C]_{2,2} = [A]_{2,k}[B]_{k,2} + [C]_{2,2}, \quad (3.4)
\]

Note that matrix-matrix multiplication introduces a degree of freedom \(1 \leq k \leq 4\).

Any positive integers \(r\), \(c\), and \(k\) are allowed, as even expressions with poor divisibility can then be tiled; accordingly, LGEn has to handle the left-over code efficiently, a challenge that is most interesting for vectorized code (see Section 3.3). For example, Figure 3.4 depicts the structures resulting from two possible choices of \(k\) in \((3.4)\). The level-1 tiled matrix \([A]_{2,k}\) is \(2 \times 2\) in both cases, but is homogeneous for \(k = 2\) and heterogeneous for \(k = 3\).

Multilevel tiling is done by applying the tiling rules in Table 3.3 to previously tiled equations. The only constraint is that the created new tiles are not composed of subtiles of different sizes. For example, further tiling of \([A]_{2,3}\) in Figure 3.4 with \((r, c) \in \{(1,2), (2,2)\}\) would not be allowed. Multilevel tiling is used for vectorization in Section 3.3.
Since we consider small problem sizes, we tile scalar code for locality in the registers (and for vectorization as explained later). In this case, following ideas discussed in Section 2.3, we require $rc \leq n_{\text{reg}}$ (the number of floating point registers) and bind $k$ depending on the size of the L1 I-cache.

3.2.3 Step 2: From LL to $\Sigma$-LL

After the tiling has been fixed, LGEN translates the resulting LL statement into a language called $\Sigma$-LL, a generalization of SPIRAL’s $\Sigma$-SPL [36]. It is still purely mathematical and represents a natural intermediate step when translating a BLAC into loop code. This makes access patterns and loops explicit as matrices and matrix sums, respectively, enabling optimizations like loop merging and loop exchange at a high level of abstraction.

$\Sigma$-LL gathers and scatters. $\Sigma$-LL includes gather and scatter operators to extract or insert submatrices from or to larger matrices with the purpose of formally capturing data accesses as matrices. Assuming $A$ is $3 \times 3$, then the top left $2 \times 2$ submatrix can be extracted using gather matrices as

$$A(0 : 1, 0 : 1) = G_L A G_R,$$

with $G_L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ and $G_R = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$.

Similarly, defining the scatter matrices $S_L = G_R$ and $S_R = G_L$, we can insert a $2 \times 2$ matrix $B$ into a $3 \times 3$ matrix $A$ as

$$A = S_L B S_R.$$

In general, gather and scatter matrices are parametrized using affine functions on indices. More formally, we define the $\Sigma$-LL gather operator as

$$g = [i, j]_{k, \ell}^{m,n} : \mathbb{R}^{m \times n} \to \mathbb{R}^{k \times \ell},$$

$$A \mapsto A g = G_L(h_{i,1}^{k \to m}) A G_R(h_{j,1}^{\ell \to n}) = A(i : i + k - 1, j : j + \ell - 1).$$
Figure 3.5: Tiled matrix-vector multiplication $[y]_{2,1} = [A]_{2,2} [x]_{2,1}$ with $A \in \mathbb{R}^{4 \times 4}$ and $x \in \mathbb{R}^4$. White regions, created using scatters, contain zeros.

Note that we write the function on the right (as common for indexing) because gathers operate from the right. Indeed, if $g' = [i', j']_{k_{u,v}}$ and $gg' = [i, j]_{m,n} [i', j']_{u,v} = [i + i', j + j']_{m,n}$, then $(A g) g' = A (gg')$.

The scatter operator is the dual of the gather:

$$s = k_{m,n}[i, j]: \mathbb{R}^{k \times \ell} \rightarrow \mathbb{R}^{m \times n},$$

$$A \mapsto s A = S_L(h_{i_1 \rightarrow m}) A S_R(h_{j_1 \rightarrow n}),$$

where $B = s A$ is defined through $B(i : i + k - 1, j : j + \ell - 1) = A$ and $B$ is zero elsewhere. In this case $s'(s A) = (s's)A$ for the natural definition of $s's$. We will often omit the domain and range parameters to simplify representation.

**COMPLETE ALGORITHM SPECIFICATION IN Σ-LL.** Using gathers and scatters, tiled computations can be expressed as summations. Consider, for example, the tiled matrix multiplication (excluding the update) in (3.4) with $k = 2$. The computation on the tiles is visualized in Figure 3.5.
In $\Sigma$-LL, this is expressed as

$$C = \sum_{i,j,p} [i,j] \left( \sum_{i',j',p' \in \{0,1\}} [i',j'] \left( \begin{array}{c} 2 \times 2 \\ \begin{array}{cc} 2 \times 2 & 2 \times 2 \\ scalar & scalar \\ \end{array} \\ \end{array} \right) \right) \right)$$

$$= \sum_{i,j,p} [i,j] [i',j'] (A[i,p][i',p'] B[p,j][p',j']). \tag{3.5}$$

The latter corresponds to a tiled implementation with six loops.

In summary, $\Sigma$-LL makes the index functions explicit as symbolic objects that can be manipulated through rewriting. Possible loops are made explicit as mathematical summations. Since the language is mathematical, rewrite rules are simply mathematical identities.

rewriting LL to $\Sigma$-LL. $\Sigma$-LL statements are plain, meaning that they cannot contain the tiling operator (3.1). The translation from LL to $\Sigma$-LL is done by
Table 3.4: Simplification properties for the LL gather and scatter operators. (3.11) holds if \( i \in (b_0 : s_0 : m) \) and \( j \in (b_1 : s_1 : n) \). \( A, B, \) and \( A_{i,j} \) are LL expressions and \( A_{i,j} \) depends on indices \( i, j \).

\[
A[i, j]\{m, n\}[i', j'][m', n'] = A[i + i', j + j'][m, n]
\]

(3.6)

\[
m', n'[i, j]k, \ell[m', n'[i', j']A = k, \ell[m, n[i + i', j + j']A
\]

(3.7)

\[
k, \ell[m, n[i, j] (A[i, j]\{m, n\}) = (k, \ell[m, n[i, j])A
\]

(3.8)

\[
m', k'[i, p]A_k, k'[n, p, j]B = m', n'[i, j](AB)
\]

(3.9)

\[
A[i, p]\{m, k\}B[p, j]\{k, n\} = (AB)[i, j]\{m, n\}
\]

(3.10)

\[
\left( \sum_{i' \in (b_2 : s_2 : m)} \sum_{j' \in (b_1 : s_1 : n)} k, \ell[m, n[i', j']A_{i', j'} \right) [i, j]\{m, n\} = A_{i,j}
\]

(3.11)

Table 3.5: Rules to recursively translate LL into \( \Sigma \)-LL. \( \langle e \rangle, \langle e_L \rangle, \) and \( \langle e_R \rangle \) are LL (possibly tiled) expressions.

\[
\langle e_L \rangle_{r,c} + \langle e_R \rangle_{r,c} \rightarrow \sum_{i,j} r_{i,j}^c (\langle e_L \rangle[i, j]_{i, j} + \langle e_R \rangle[i, j]_{i, j})
\]

(3.12)

\[
\langle e_L \rangle_{r, k} \cdot \langle e_R \rangle_{k, c} \rightarrow \sum_{i,j,p} r_{i,j}^{c_{i,j,p}} (\langle e_L \rangle[i, p]_{i, j} \cdot \langle e_R \rangle[p]_{i, j})
\]

(3.13)

\[
\langle \text{scalar} \rangle_{1,1} \cdot \langle e \rangle_{r, c} \rightarrow \sum_{i,j} r_{i,j} (\langle \text{scalar} \rangle \cdot \langle e \rangle[i, j]_{i, j})
\]

(3.14)

\[
\langle e \rangle_{r, c}^T \rightarrow \sum_{i,j} c_{i,j}(\langle e \rangle[i, j]_{i, j})^T
\]

(3.15)

rewriting the expression tree with tiling information until a plain statement is constructed. The rewrite rules are shown in Table 3.5.

3.2.4 Step 3: Loop Transformations

At this step a \( \Sigma \)-LL statement can be transformed by manipulating summations, gathers, and scatters. In the final code this would correspond, e.g., to loop fusions or loop exchange.

simplifications and loop fusion. A \( \Sigma \)-LL expression obtained from a straightforward translation such as (3.5), will not represent an efficient program since every gather and scatter requires separated read and write code blocks. Using mathematical identities such as those in Table 3.4, LGEN reduces the num-
ber of these objects. For example, also in Σ-LL gathers and scatters can be fused by composing the index functions (a strided access of a strided access yields a strided access):

\[
\sum_{i,j,p} [i + i', j + j'] (A[i + i', p + p']B[p + p', j + j']) \rightarrow \sum_{i,j,p} [i + i', j + j'] (A[i + i', p + p']B[p + p', j + j'])
\]

Gathers and scatters can cancel each other if subsequent reads and writes are not to overlapping locations enabling the fusion of subexpressions into the same index space. Consider the application of the process illustrated so far to the full BLAC (3.3) assuming we want to generate a naïve triple loop. This can be obtained by tiling the BLAC with unitary tile sizes and applying the rewrite rules in Table 3.5, yielding

\[
C = \sum_{0 \leq i,j < 4} 1,1^4,4[i,j] ( \sum_{0 \leq l,t,p < 4} 1,1^4,4[l,t] (A[l,p] 1,1^4,4^4 B[p,t] 1,1^4,4) + C[i,j] )
\]

Now we can distribute the gather \([i,j] 1,1^4,4\) in (3.16) over the innermost summation using rule (3.11) to obtain

\[
C = \sum_{0 \leq i,j < 4} [i,j] ( \sum_{0 \leq p < 4} A[i,p]B[p,j] + C[i,j] )
\]

**Loop ordering.** The order of loops in the generated loop nests is in principle a further degree of freedom in the code generation process. Instead of enlarging the search space we determine nest-wise local orderings, using what we call a priority matrix (Π). Every time a summation is created using rules from Table 3.5, its indices are associated with new rows of Π. Columns of Π are related to factors that can influence performance. In our study we consider three factors: instruction-level parallelism (ilp), temporal locality (tl), and spatial locality (sl).

Every entry Π(i, f) estimates the (positive) impact on factor f of increasing index i before other indices of the summation. For example, Π(i, tl) > Π(j, tl) means that increasing index i before j has better temporal locality. In general, this value can depend on the operation and on the tiling level. For example, following the discussion in [121] about loop ordering for matrix multiplication...
We use the following criteria for rule (3.13) in Table 3.5 assuming we are at an outer level of tiling:

- **Temporal locality**: varying \( p \) has larger impact as we keep operating on same output elements, \( \Pi(p, tl) = 1, \Pi(t, tl) = 0, t \in i, j \).

- **Spatial locality**: Row-major indices have larger impact, \( \Pi(i, sl) = 0, \Pi(p, sl) = 1 \), and \( \Pi(j, sl) = 2 \).

- **Ilp**: moving along the dimensions of the output matrix has larger impact, \( \Pi(t, ilp) = 1, t \in i, j, \Pi(p, ilp) = 0 \).

Using the above criteria we obtain the following matrix \( \Pi \):

\[
\begin{array}{ccc}
\Pi & \text{tl} & \text{sl} & \text{ilp} \\
\hline
i & 0 & 0 & 1 \\
j & 0 & 2 & 1 \\
p & 1 & 1 & 0 \\
\end{array}
\]

We determine the order of the indices in two steps: (a) we sort the columns by the priority given to the performance factors; (b) we sort the rows of \( \Pi \) in ascending lexicographical order. In our example, assuming priority \((tl, sl, ilp)\), with \( tl \) being highest, we would determine the order \((i, j, p)\); giving higher priority to spatial locality, i.e., assuming priority \((sl, tl, ilp)\), yields \((i, p, j)\).

### 3.2.5 Step 4: From \( \Sigma \)-LL to C-IR

Next, \( \Sigma \)-LL expressions are converted into a C-intermediate representation (C-IR). Translation into C-IR first requires binding the physical layouts of implicit matrices appearing in a \( \Sigma \)-LL expression tree (i.e., those associated to gathers, scatters, summations, and basic operators) to temporary arrays. This process is similar to the one illustrated in Figure 3.3 for explicit input and output matrices. A domain-specific representation of the matrices at the C-IR level, by combining their physical and mathematical layouts, is particularly useful for accesses data with vector instructions as discussed later in Section 3.3. At the moment we only support contiguous, full storage schemes but the approach should be modular enough to accommodate for a variety of storage schemes.

After matrix binding, memory references are used in combination with code templates associated with the \( \Sigma \)-LL operators to produce C-IR code. The access patterns are deduced from the index mapping functions of the gathers and scatters and incorporated in the reference objects. For example, the code template for scalar addition can be described through the pseudocode in Figure 3.6. The codelet makes the following assumptions: (a) \texttt{left} and \texttt{right} are scalar expressions; (b) the reference objects (e.g., \texttt{inL}) have access to both mathematical and physical layouts of the (implicit or explicit) quantities associated to \texttt{left} and
3.3 Vector Code Generation

The previous section described the generation of scalar code for fixed-size BLACs using LGEN. However, to obtain high performance, vectorization for SIMD ISAs is crucial. In this section, we explain how LGEN generates C code.

```
genAdd(B, expr, left, right):
    # code for expr = left + right
    inL = getReference(left)
    inR = getReference(right)
    out = getReference(expr)
    B.append(Mov(Add(inL[0,0], inR[0,0]), out[0,0]))
```

Figure 3.6: Code generation template for scalar addition; the object B is a C-IR basic block of function to be generated.

right. They contain all the information necessary to locate the position of the scalars within possibly larger matrices.

3.2.6 Step 5: Code-level optimizations

LGEN takes advantage of the C-IR formulation of the computation for applying a set of standard compiler optimizations, such as loop unrolling of the innermost loops, scalar replacement, and conversion into SSA form. Finally, the C-IR code is unparsed into C.

3.2.7 Step 6: Performance Test and Autotuning

After the C function is generated, it is executed and its performance is measured and used for autotuning. The number of functions that can be generated depends on the degrees of freedom introduced by tiling in Step 1. If more than one function can be generated, LGEN explores them either by exhaustive or by random search.

3.2.8 Summary of LGEN’s Choices During Code Generation

Before concluding this section, we summarize in Table 3.6 the four major choices currently made by LGEN during its code generation process. The table points out in which of the six steps explained above these choices appear and briefly recap how they are made.

3.3 Vector Code Generation

In the previous section we described the generation of scalar code for fixed-size BLACs using LGEN. However, to obtain high performance, vectorization for SIMD ISAs is crucial. In this section, we explain how LGEN generates C code.
Table 3.6: The four main choices made by LGEN during code generation. The second column shows in which of the six steps in Section 3.2 they appear while the third column briefly summarizes how they are made.

<table>
<thead>
<tr>
<th>Choice</th>
<th>When made</th>
<th>How made</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiling</td>
<td>Step 1 &amp; 6</td>
<td>Tile sizes are considered degrees of freedom. The space of possible values is modelled as in (2.13) and explored with exhaustive or random search.</td>
</tr>
<tr>
<td>Loop fusion</td>
<td>Step 3</td>
<td>A single choice is made applying the simplification properties in Table 3.4 whenever possible.</td>
</tr>
<tr>
<td>Loop ordering</td>
<td>Step 3</td>
<td>Single choice made using the priority matrix described in Step 3.</td>
</tr>
<tr>
<td>Loop unrolling</td>
<td>Step 5</td>
<td>Single choice: All innermost loops are fully unrolled.</td>
</tr>
</tbody>
</table>

including intrinsics to explicitly use vector instructions. The vector length (e.g., 4 for SSE float) is denoted with $v$.

An important feature of our vectorization approach is extensibility. This means that porting to a new vector architecture is a straightforward, non-creative effort. Our solution does this conceptually similar to how it was done in SPIRAL [35]. Specifically, we identify a few basic vectorized building blocks, called $v$-BLACs, that need to be available to our system: porting to a new instruction set then simply requires their implementation.

The generation process extends the one for scalar code introduced in Section 3.2 in the following way: (i) LGEN receives the input BLAC together with the vector length $v$ of the ISA as part of the compilation options; (ii) before tiling for registers, we apply a first level of tiling ($v$-tiling) to match to the $v$-BLACs; (iii) $v$-BLACs are associated with a set of pre-implemented codelets that are generated at C-IR level; (iv) data processed by the $v$-BLACs are loaded and stored using a pack-compute-unpack approach. Thanks to this approach also left-over code (for parts smaller than $v \times v$) is vectorized by embedding into $v$-BLACs.

We now describe the $v$-BLACs, tiling, their use to generate code, and how data is accessed using load and store building blocks.

### 3.3.1 $v$-BLACs: Computational Building Blocks

A $v$-BLAC is a BLAC with the following characteristics: (a) only one operator is used; (b) it can be efficiently implemented on a vector ISA. For this we consider the four basic operators in LL (multiplication, addition, scalar multiplication, transposition) applied to all possible valid combinations of matrices and vectors of size $1 \times v$, $v \times 1$, or $v \times v$, yielding the 18 $v$-BLACs in Table 3.7.
Table 3.7: The 18 required ν-BLACs for vectorization in LGen.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Required ν-BLACs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>+ + +</td>
</tr>
<tr>
<td>(3 ν-BLACs)</td>
<td></td>
</tr>
<tr>
<td>Scalar Multiplication</td>
<td></td>
</tr>
<tr>
<td>(7 ν-BLACs)</td>
<td></td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td></td>
</tr>
<tr>
<td>(5 ν-BLACs)</td>
<td></td>
</tr>
<tr>
<td>Transposition</td>
<td>$^T$ $^T$ $^T$</td>
</tr>
<tr>
<td>(3 ν-BLACs)</td>
<td></td>
</tr>
</tbody>
</table>

blac_nu2_xpy(B, refx, refy, out):
  B.append( Mov( mmLoaduPd( refx[0,0] ), vx ) )
  B.append( Mov( mmLoaduPd( refy[0,0] ), vy ) )
  B.append( mmStoreuPd( mmAddPd(vx, vy ), out[0,0] ) )

Figure 3.7: ν-BLAC C-IR codelet for $x + y$ and $ν = 2$; $x$ and $y$ are either $ν \times 1$ or $1 \times ν$.

Figure 3.7 shows a C-IR codelet for the implementation of the ν-BLAC $x + y$ using double precision SSE intrinsics. In this example, $x$ and $y$ can be either row or column vectors. This shows that multiple ν-BLACs, even if mathematically different, can be implemented using a single codelet. Porting LGen to a new vector ISA only requires the implementation of the codelets associated with the 18 ν-BLACs in Table 3.7 (and associated packing routines explained later in Section 3.3.2) using the intrinsics available for the new target ISA.

In Figure 3.7 we show an implementation that uses unaligned load instructions. An approach to enable the use of aligned accesses, for architectures where this matters, is described later in Section 3.3.2. At the moment, the only considered data layout assumes that vectors and matrix rows are contiguous in memory.
TILING FOR V-BLACS. Once $v$ is provided to LGEN, the latter performs a first level of tiling with $(r, c) \in \{(1, v), (v, 1), (v, v)\}$. For example, consider (3.3) where $A$ is $3 \times 4$ and $B$ is $4 \times 1$ (being a vector we denote it $b$ in this example). Assuming $v = 2$, we tile with $(r, c) = (v, v)$ to get

$$[c]_{v,1} = [A]_{v,v}[b]_{v,1} + [c]_{v,1},$$

or visually

$$\nu(\begin{array}{c} 1 \end{array}) = \nu(\begin{array}{c} 1 \end{array}) + \nu(\begin{array}{c} 3 \end{array}).$$

For simplicity, we separately consider the $\nu$-LL statements $[z]_{v,1} = [A]_{v,v}[b]_{v,1}$ and $[c]_{v,1} = [z]_{v,1} + [c]_{v,1}$. Following the procedure described in Section 3.3, we obtain

$$z = \sum_{p \in \{0,2\}} \frac{v}{3}[0] \left( A[0,p]^{3,4} b[p]^{4} \right) \quad (3.17)$$

$$+ \sum_{t \in \{0,2\}} \frac{1}{3}[2] \left( A[2,t]^{3,4} b[t]^{4} \right) \quad (3.18)$$

$$c = \frac{1}{3}[0] \left( z[0]^{3} + c[0]^{3} \right) \quad (3.19)$$

$$+ \frac{1}{3}[2] \left( z[2]^{3} + c[2]^{3} \right) \quad (3.20)$$

The equations (3.17)–(3.18) describe the same computation but performed using tiles of different size. The same holds for (3.19)–(3.20). In particular, (3.17)–(3.19) map directly to $v$-BLAC codelets, while (3.20) needs additional work to be mapped. We describe both situations next.

CODE GENERATION WITH V-BLACS. The codelets for the $v$-BLACs and the load and store building blocks are pre-implemented and are retrieved for a given ISA using the parameter $v$ and the required floating point precision. Hence the code generation template for scalar code in Figure 3.6 is extended as shown in Figure 3.8.

LEFT-OVER CODE HANDLING. Expressions such as $z[2]^{3} + c[2]^{3}$ (adding two vectors of length one) in (3.20) do not conform to any $v$-BLAC with $v = 2$. The basic idea to vectorize these is to embed them into a (larger) $v$-BLAC of appropriate type. For example, a very small matrix-vector multiplication is embedded as shown here:

$$\begin{array}{c} = \begin{array}{c} \end{array} \Rightarrow \begin{array}{c} = \begin{array}{c} \end{array} \end{array} \nu \end{array}$$
```python
genAdd(B, expr, left, right, opts):
    # code for expr = left + right
    inL = getReference(left)
    inR = getReference(right)
    out = getReference(expr)
    isa, nu, prec = opts['isa'], opts['nu'], opts['precision']
    vecSize = sizeof(left)
    loader = getLoader(isa, nu, prec)
    storer = getStorer(isa, nu, prec)
    nublac = getNuBLAC(isa, Add, vecSize, nu, prec)

    loader(B, [inL, inR])
    nublac(B, inL, inR, out)
    storer(B, out)
```

Figure 3.8: Code generation template for addition; the object B refers to a basic block of the code.

In LGEN, this is done by the load and store routines, called Loaders and Storers, that perform the embedding of the operands. These are also pre-implemented using intrinsics and selected upon code generation (Figure 3.8, lines 13 and 15). In the remaining part of this section we describe in greater detail how this process works and address an issue that can occur when combining codelets.

### 3.3.2 Load and Store Building Blocks

LGEN generates C-IR code by combining codelets from the target ISA’s Loaders, Storers, and ν-BLACs. All of these codelets are implemented following a load-compute-store approach, meaning that they first load data from memory into registers, then they process the data, and finally they store the results back to memory. As a result, the generated C-IR code consists of chains of codelets, where data flow from one codelet to the next one. As an example, consider the partial computation $z^T = a^T B$ from (3.3) where the matrix $B$ has now size $11 \times 3$ and we consider a single row of $A$ of length $11$. Assume also that $B$ is 16-byte aligned and stored in row-major order, that our target ISA is SSSE3, and that we decide to vectorize choosing $\nu = 4$. In Figure 3.9 we illustrate the flow of computations required to take into account the leftover (the $1 \times 3$ matrix $T$) produced by ν-tiling.

The $1 \times 3$ leftover tile of $a^T$ and the $3 \times 3$ leftover tile of $B$ are loaded to the temporary quantities $T_0$ and $T_1$ (of sizes $1 \times \nu$ and $\nu \times \nu$). A similar approach is followed with the outputs using Storers. We focus on two consequences of this approach.
First, embedding elements comes at the cost of extra computations. However, the padding elements of an output quantity (e.g., \( t^T_6 \) in Figure 3.9) are not stored back. This allows the compiler to remove operations associated to them performing dead code elimination.

Second, the code associated to the store-load chain \( t^T_2 \rightarrow t^T_3 \rightarrow t^T_4 \) should be removed, holding the result from the multiplication in register for the second computation (Add \( \nu \)-BLAC). However, as shown in Figure 3.10, on architectures lacking mask-load and -store instructions, the special size of the output vector (\( 1 \times 3 \)) requires additional data rearrangement overhead (e.g., vector shuffles).

Data between two consecutive codelets are stored in a local array. However, in the example shown in Figure 3.9 the use of the temporary matrices \( T_0, \ldots, T_6 \) is superfluous since the result of each codelet could be passed directly to the next codelet through registers. Store-load elimination (SLE) is a compiler technique that can be used to eliminate redundant memory accesses. As we explain in the following, the challenge is in removing unnecessary shuffles.

**PROBLEMS USING SLE WITH INTRINSICS.** Standard SLE works in the following way: Whenever a pair of load and store intrinsics with matching access patterns is found, it is replaced with an assignment between vector variables. By access pattern we refer to the mapping between memory locations and positions within a vector variable. Loads that do not follow any store with the same access pattern are left unchanged. The same holds for stores that are not followed by loads with the same access pattern. For example, Figure 3.11 depicts the store-load chain from Figure 3.10 with access patterns depicted with arrows.
/* Begin nuBLAC mul */
// ... nuBLAC multiplication
__m128 mul_res = ...;
_mm_storeu_ps(t2, mul_res);
/* End nuBLAC mul */

/* Begin Storer 1x4 -> 1x3 */
__m128 v0 = _mm_loadu_ps(t2);
_mm_storel_pi((__m64*)(t3), v0);
_mm_store_ss(t3 + 2,
_mm_shuffle_ps(v0, v0, _MM_SHUFFLE(3, 3, 2)));
/* End Storer 1x4 -> 1x3 */

/* Begin Loader 1x3 -> 1x4 */
_mm_storeu_ps(t4,
_mm_shuffle_ps(

_mm_loadl_pi(_mm_setzero_ps(), (__m64*)(t3)),
_mm_load_ss(t3 + 2),
_MM_SHUFFLE(1, 0, 1, 0)
)
); /* End Loader 1x3 -> 1x4 */

Figure 3.10: SSSE3 code snippet for the store-load chain $t_2^T \rightarrow t_3^T \rightarrow t_4^T$ in Figure 3.9).

before SLE:
$v_1 = \_mm\_shuffle\_ps(v_0, v_3,
_MM\_SHUFFLE(3, 3, 2));$
$\_mm\_storel\_pi(addr, v_4);$
$v_2 = \_mm\_loadl\_pi(addr);$
$v_3 = \_mm\_load\_ss(addr+2, v_1);$ $v_4 = \_mm\_shuffle\_ps(v_2, v_3,$
_MM\_SHUFFLE(3, 3, 2)); +

after SLE:
$v_1 = \_mm\_shuffle\_ps(v_0, v_3,$
_MM\_SHUFFLE(3, 3, 2));$
$v_2 = v_3;$ $v_3 = v_1;$
$v_4 = \_mm\_shuffle\_ps(v_3, v_1,$
_MM\_SHUFFLE(1, 1, 0));

Figure 3.11: Example of SLE with load and store instrinsics. Both $v_0$ and $v_4$ contain a leftover of length three.
Finding store-load pairs to eliminate is thus equivalent to finding stores and loads whose outgoing and ingoing arrows can be “wired” together. Since storing the three values $a$, $b$, $c$ to memory is implemented in the same way as loading them, we can safely wire up outgoing and ingoing connections. In other words, applying SLE to this piece of code replaces the wired store-load pairs with assignments between variables $v_2$ and $v_0$, and between $v_3$ and $v_1$ (bottom-right code snippet). However, shuffles are left untouched by the analysis.

**Better SLE with generic C-IR vector load and store instructions.** To facilitate the application of SLE and avoid unnecessary shuffle instructions like the ones shown before, we use in our C-IR load and store instructions that do not correspond to specific intrinsics, but are generic enough to represent all possible vector accesses to memory. These instructions are used during SLE and are translated to specific intrinsics only during unparsing C-IR into C code.

General purpose compilers, such as LLVM [73], provide generic vector instructions at the IR level. Our approach however, is closer in spirit to the work in [95] and [35], where generic vector instructions are defined at a higher level of abstraction than usual vector data types and are geared towards working with matrices. This enables the efficient handling of our domain of interest.

The full syntax of a generic load is \texttt{GenLoad(addr, poslist, orientation)} and the one of a generic store is \texttt{GenStore(addr, v, poslist, orientation)}. The parameter \texttt{addr} is a memory address, \texttt{v} is a vector variable, and \texttt{poslist} is a list that maps memory locations to positions within the vector \texttt{v}. More specifically, the \texttt{i}th element of \texttt{poslist} maps the \texttt{i}th element starting from \texttt{addr} to a list of positions within a vector. For example, \texttt{GenLoad(addr, [ [0],[1],[2],[3] ], hor)} loads four consecutive elements starting from \texttt{addr} to the four positions of the returned vector, while \texttt{GenLoad(addr, [ [0,1,2,3] ], hor)} loads one element at \texttt{addr} to all four positions of the returned vector. The parameter \texttt{orientation} can take the value \texttt{hor} or \texttt{vert} and determines whether the access refers to a row or a column of a mathematical matrix. A generic load/store with \texttt{orientation} set to \texttt{vert} is interpreted as a strided memory access, and the stride is obtained from the mathematical layout of the accessed matrix.

Using these generic load/store instructions, for example, the code segment of Figure 3.11 is transformed into the one shown in Figure 3.12. Applying SLE on the latter will leave us with a single assignment, without any shuffle instructions. An example implementation of the generic load and store in Figure 3.12 on NEON is shown in Figure 3.13. Note that the "non-dual" mapping of generic loads and stores to code does not affect SLE.

**Optimal alignment detection.** In Section 3.3.1, we mentioned that data accesses are performed by default using unaligned instructions. Several processors with vector architectures offer both aligned and (slower) unaligned loads and stores. Although for some architectures their performance difference is neg-
Figure 3.12: Example of SLE with generic load and store C-IR instructions analogous to the one in Figure 3.11.

Figure 3.13: Unpacking-packing of a leftover of length three using a non-dual NEON implementation of the generic store and load.

ligible for others this is not the case. For example, on embedded processors such as Intel Atom and ARM Cortex-A8 and -A9 aligned instructions are at least twice as fast. Furthermore, on Intel Atom, unaligned instructions require two out of two issue ports for execution, making it impossible to issue an unaligned load simultaneously with an unaligned store. In contrast the aligned ones require only one port [60].

For these reasons we incorporated into LGEN an alignment detection algorithm based on abstract interpretation [18] at the C-IR level. The idea behind the algorithm is as follows: First the C-IR code is analyzed by applying abstract interpretation using the abstract domain of congruences [46]. This analysis is similar to the congruence detection technique described in [72] and determines for each memory access, whether all the addresses used in this access during program execution are guaranteed to be divisible by the ISA-specific alignment
length. In the affirmative case, an aligned load or store intrinsic is generated when unparsing to C code. Otherwise an unaligned instruction is generated.

Note that if the alignment of the input and output arrays is not known at compile time, LGEN can generate various code versions, one for each of the possible alignments, including the control to select the proper one at runtime.

In our context, all addresses are affine combinations of induction variables. Based on this restriction, one can prove that our alignment analysis is precise [69], i.e., each aligned memory access is detected and there are no false negatives.

3.4 RELATIONSHIP TO PRIOR SPIRAL-LIKE ATTEMPTS AT LINEAR ALGEBRA

We conclude this chapter comparing our approach to the one taken by the Operator Language (OL) [21, 33], a previous SPIRAL-like DSL and framework used to generate GEMM among other functionalities. OL is an extension of the Signal Processing Language (SPL) [120] originally used in SPIRAL to describe linear transforms.

OL inherits features from SPL that are typical of the domain of transforms. In particular, OL is designed to represent and manipulate operators on vectors using a point-free notation, meaning that the input and output quantities of a computation are not explicitly represented. For example, the multiplication matrix-matrix multiplication $C = AB$, with $A \in \mathbb{R}^{m,k}$, $B \in \mathbb{R}^{k,n}$, and $C \in \mathbb{R}^{m,n}$, is represented only in terms of its bilinear operator:

$$\text{MMM}_{m,k,n} : \mathbb{R}^{mk} \times \mathbb{R}^{kn} \rightarrow \mathbb{R}^{mn}; (\text{vec}(A), \text{vec}(B)) \mapsto \text{vec}(AB),$$

where the function vec produces a vector of size $mn$ from a matrix of size $m \times n$ linearized in row-major order. In contrast to LL, OL’s notation yields a more complex and less conventional representation of linear algebra expressions (an example of generation of straightforward code for $\text{MMM}_{m,k,n}$ is provided in Appendix A). For this same reason, developing and debugging OL formulations may represent a complex task for a developer used to the notation adopted by linear algebra sources such as [24, 40, 107].

Further, representing input and output features such as the structure of a matrix, might result less natural with a point-free notation. Structures in LL are associated to matrices using an approach discussed in detail in the next chapter.

Finally, tiling with factors that do not divide the input dimensions is captured by OL for matrix-matrix multiplication, but leftover computations are not easily expressible [21]. This could reflect in equally complex formulations of leftover computations in a BLAC.
OL AS A MATHEMATICAL INTERMEDIATE REPRESENTATION. Despite disadvantages in representing linear algebra computations such as those listed above, we think that the use of OL in the context of a multi-domain code generator should be investigated further. In particular, its ability to capture computations from different mathematical domains could make OL an intermediate mathematical language for expressing cross-domain optimizations.
A BASIC LINEAR ALGEBRA COMPILER FOR STRUCTURED MATRICES

The cost of computing a BLAC can be significantly reduced if the matrices have structure. For example, multiplying two upper triangular matrices requires only about one-sixth of the total amount of instructions necessary to multiply two general matrices (see Section 2.1.4). Further, the storage scheme of a structured matrix must be taken into account to ensure correct access to the data. For example, adding a symmetric matrix to a general one may require different access patterns for the two matrices. In this chapter we extend LGEN to support sBLACs, i.e., the class of basic linear algebra computations with structured matrices introduced in Section 2.1.4.

In the previous chapter we described how LGEN could be used to generate code for a BLAC using as a running example a GEMM of the form:

\[ C = AB + C. \]  

(4.1)

Assume now the goal of generating code for a \( \nu \)-way vector ISA, with \( \nu = 2 \), for the sBLAC

\[ A = LU + S, \quad A, L, U, S \in \mathbb{R}^{4 \times 4}, \]  

(4.2)

which is analogous to (4.1) but includes a lower and an upper triangular matrix, i.e., \( L \) and \( U \), and a symmetric matrix \( S \).

PROGRAM GENERATION WITH LGEN. We briefly recall how the previously described LGEN (Figure 3.1) would generate code for (4.2) in six steps:

- Step 1: Tiling in LL. The input BLAC, represented as an LL statement, is tiled recursively with fixed parameters (a degree of freedom; perfect divisibility is not required) and propagates the tiling decision to the operands. If code for a \( \nu \)-way vector ISA is desired, the lowest level block size has to be \( \nu \) to decompose the computation into pieces, called \( \nu \)-BLACs, that can be mapped well to vector code. In our example, we consider only one level of tiling with \( \nu = 2 \):

\[ [A = LU + S]_{\nu,\nu} \overset{\nu=2}{\mapsto} [A]_{2,2} = [L]_{2,2}[U]_{2,2} + [S]_{2,2}. \]  

(4.3)
• **Step 2: Conversion from LL to Σ-LL.** A fully tiled LL statement is rewritten into a second DSL called Σ-LL. This representation is still mathematical and makes loops and data accesses explicit. The latter are captured as explicit gather and scatter operators on matrices to allow for reasoning and fusion through rewriting. In our example, from (4.3) LGEN would derive the following Σ-LL statement:

\[
A = \sum_{i,j \in \{0,2\}} \left( \sum_{l,r,k \in \{0,2\}} ^{2,2} 4,4 \left[ i,j \right] \right) \left( \sum_{l,r,k \in \{0,2\}} ^{2,2} 4,4 \left[ l,r \right] \left( L[l,k]_4^4 U[k,j]_2^2 \right) \right) \left[ i,j \right]_4^4 U[k,r]_2^2 \left[ i,j \right]_4^4 \left( 4,4 \times 4,4 \right) + S[i,j]_4^4 \right) .
\]

• **Step 3: Loop-level transformations.** At this step a Σ-LL statement can be transformed by manipulating summations, gathers, and scatters. In the final code this would correspond, e.g., to loop fusions or loop exchange. For example, starting from (4.4), LGEN could fuse loops by distributing the first gather \([i,j]_4^4 \) (second line) over the innermost summation to obtain

\[
A = \sum_{i,j \in \{0,2\}} \left( \sum_{k \in \{0,2\}} L[i,k]_4^4 U[k,j] + S[i,j] \right) .
\]

• **Step 4: Conversion from Σ-LL to C-IR.** At this point the Σ-LL representation in (4.5) has the following features: (a) the number of summations and their order is defined, (b) (if vectorization is enabled) the entire formulation is decomposed into \(\nu\)-BLACs, meaning that all operations are performed on \(\nu\)-tiles of the input matrices. The translation between Σ-LL and C-IR (LGEN’s C-like IR) is performed by mapping summations to loops, \(\nu\)-BLACs to codelets, and gathers and scatters to data accesses. \(\nu\)-BLACs are the 18 single-operation BLACs that operate on tiles of size \(\nu \times \nu\), \(1 \times \nu\), and \(\nu \times 1\) with the four BLAC operators introduced in Section 3.3.1. They are preimplemented once for every vector ISA. The gathers and scatters are associated to a collection of vectorized data access basic blocks, called Loaders and Storers, that are used to perform low level optimizations including handling leftovers.

• **Step 5: Code-level optimizations.** Before unparsing to C code LGEN may apply further optimizations such as loop unrolling, scalar replacement, conversion into SSA form, and alignment detection.
• **Step 6: Performance tests and autotuning.** Finally, LGEN unparses the C-IR into C code vectorized with intrinsics and tests its performance. Autotuning is used to find the best result among available variants.

One may notice that certain computations in (4.5) are redundant, e.g., \( L[0, 2]U[2, j] \) and \( L[2, 2]U[2, 0] \). Also, a common standard for symmetric matrices stores only one side of the matrix, e.g., the lower one. In this case, access to \( S[0, 2] \) should be replaced with \( S[2, 0]^T \).

In the following sections we will show how to perform these analyses and transformations with LGEN. In particular, we will propose a methodology that combines LGEN’s internal DSLs with ideas from polyhedral compilation.

In Section 4.1, we introduce notions from the polyhedral model needed to represent matrix structures and reason about them. In the same section, we also present two tools from the polyhedral toolchain used to extend LGEN.

In Section 4.2, we explain how structured matrices are represented internally by LGEN and describe how code is generated from an sBLAC. Section 4.3 provides a more detailed overview of the vectorization approach.

In this chapter, we mainly focus on lower triangular, upper triangular, and symmetric matrices as prototypical examples. However, the methodology is extensible to include a much larger set of matrix structures as we discuss in Section 4.4.

### 4.1 The Polyhedral Model

The polyhedral model [32] is a mathematical representation for a class of loop-based, parallel programs. By a loop-based program we mean one composed of statements possibly surrounded by loop nests. Loops can be perfectly or imperfectly nested, depending on whether they contain statements only in the innermost loop or not. Finally, conditions on loop indices are expressed with affine constraints in Presburger arithmetic (i.e., constraints built with only additions on integer variables and constants).

To illustrate the model, we will consider the following implementation in C of the GEMM computation in Algorithm 2.5, where we label the single statement in the innermost loop as \( S_0 \):

```c
for(i=0; i<m; ++i)
  for(j=0; j<n; ++j)
    for(l=0; l<k; ++l)
      S0: \( C[i*n+j] += A[i*k+l] * B[l*n+j]; \)
```

The polyhedral model of a program’s statement consists of:

• A set of integer points bounded by a polytope called the *iteration domain* of the statement. The dimensions of the polytope are associated with the
number of loops surrounding the statement and its boundaries with the
boundaries of the loops. For instance, statement \( s_0 \) in the above program
is executed \( kmn \) times for different values of \((i, j, \ell)\). The set of all values
of the indices for which the statement has to be executed (i.e., its iteration
domain) is therefore the following three-dimensional polytope or polyhe-
dron:

\[
\sigma_{s_0} = \{(i, j, \ell) \in \mathbb{Z}^3 \mid 0 \leq i < m, 0 \leq j < n, 0 \leq \ell < k\},
\]

where, for example, the tuple \((0, 1, 2)\) is associated with the execution

- A function that provides an execution order for every point in the poly-
tope called the schedule of the statement. For example, the schedule for the
program above is represented by a function that maps every tuple in the
iteration domain of statement \( s_0 \) to a tuple that represents a logical time
for its execution:

\[
f(i, j, \ell) = (i, j, \ell), \quad (i, j, \ell) \in \sigma_{s_0}.
\]

(4.6)

In the context of linear algebra computation, we will call iteration space the
union of of the iteration domains of all the statements required to compute it.
For example, the iteration space of the implementation of the matrix-matrix mul-
tiplication shown above coincides with the iteration domain of its only statement
\( s_0 \) (i.e., \( \sigma_{s_0} \)).

Next, we explain how the polyhedral model is used in a polyhedral compila-
tion process.

4.1.1 Polyhedral Compilation Process

Polyhedral compilation tools such as Pluto [16] and LLVM/Polly [47], manip-
ulate polyhedral representations, such as the one just described, to apply code
transformations, such as loop parallelization and vectorization.

The process normally requires to (a) translate a program to its polyhedral
representation, (b) to apply transformations that produce new iteration domains
or schedules or both, and (c) to generate a new program which loops visit (or
scan) every point of every polytope following their schedules.

The Chunky Loop Generator (CLooG) [9] is a polyhedral tool that handles
the latter code generation step. Given a set of statements with their respective
iteration domains and schedules, CLooG generates a loop-based program that
properly combines them. For example, assume that some transformation pro-
duces a new function \( f'(i, j, \ell) = (i, \ell, j) \) for tuples in \( \sigma_{s_0} \). Then, passing the
triplet \(<s_0, \sigma_{s_0}, f'>\) as an input to CLooG produces the following C code:

\[
\text{for}(i=\theta; i<m; ++i)
\]
```cpp
for(l=0; l<k; ++l)
    for(j=0; j<n; ++j)
        C[i*n+j] += A[i*k+l] * B[l*n+j];
```

In this chapter we describe an approach based on an extension of CLooG for manipulating sBLACs at a high level of abstraction. In particular, we will use polyhedral sets to represent both regions in matrices and iteration spaces of computations and polyhedral maps to represent access patterns of matrices and schedules.

In the remainder of this section, we fix the notation for polyhedral sets and maps in accordance with the integer set library (isl) [111], which is used in LGEN to implement such concepts.

### 4.1.2 Polyhedral Sets and Maps

The two essential concepts used in our definition of structures are *polyhedral sets* and relations (called *maps* in [111]) of $n$-tuples of integers bounded by $m$ affine constraints. A set of such $n$-tuples is defined as

$$
\sigma = \bigcup_i \{ t \in \mathbb{Z}^n \mid \exists c \in \mathbb{Z}^e : A_i t + E_i c + z_i \geq 0 \},
$$

where $A_i \in \mathbb{Z}^{m \times n}$, $E_i \in \mathbb{Z}^{m \times e}$, $z_i \in \mathbb{Z}^m$, and $\geq$ is componentwise. The existential quantifier allows us to define tuples at a stride. For example, the following sets

$$
\sigma_1 = \{(i,j) \mid 0 \leq i < 4 \land 0 \leq j < 4 \},
$$

$$
\sigma_2 = \{(i,j) \mid \exists a, b : 0 \leq i,j < 4 \land i = 2a \land j = 2b \}
$$

(4.8)

can be used to represent all integer points in a square of size $4 \times 4$ ($\sigma_1$) or those at a stride 2 ($\sigma_2$). The first set would be given by

$$
A_0 = \begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}, \ E_0 = 0, \ z_0 = (0 0 3 3)^T.
$$

The second set requires the inclusion of $i - 2a \geq 0$, $i - 2a \leq 0$, $j - 2b \geq 0$, and $j - 2b \leq 0$.

Maps in [111] are relations between sets and defined as:

$$
\rho = \bigcup_i \{(t_0, t_1) \in \mathbb{Z}^{n_0} \times \mathbb{Z}^{n_1} \mid \exists c \in \mathbb{Z}^e : A_i t_0 + B_i t_1 + E_i c + z_i \geq 0 \}.
$$

Polyhedral maps can be used to describe schedules such as (4.6) as follows:

$$
\rho_f = \{((i, j, p), (p, i, j))\}.
$$
A : Matrix(4, 4);
L : LowerTriangular(4);
U : UpperTriangular(4);
S : Symmetric(l, 4);

A = L*U+S;

Figure 4.1: LL implementation of the sBLAC (4.2).

4.2 CODE GENERATION WITH STRUCTURED MATRICES

In this section we discuss how structures are defined in LGEN using polyhedral sets and maps. The approach is designed to be extensible: adding a new structure to LGEN requires the inclusion of two different interfaces, one towards the user and one towards LGEN itself.

From a user perspective, a structured matrix is just another type of matrix within an LL input program. For example, Figure 4.1 shows a simple LL implementation of (4.2).

A structured matrix, however, also needs an internal interface to LGEN to enable its decomposition in Σ-LL. We build this interface using the isl library and the polyhedral formalism introduced in the previous section.

4.2.1 Internal Representation of Structures in LGEN

We associate every matrix with a pair of dictionaries called SInfo and AInfo.

SInfo associates regions of a matrix to structures. Its entries have the form M : σ, where M is a matrix type and σ a polyhedral set that represents an area of the matrix that can be considered of type M.

Matrix types are denoted with the notation used for matrix structures in Section 2.1.4, i.e., L, U, and S for, respectively, lower triangular, upper triangular, and symmetric matrices. In addition, with a similar notation, we refer to a general matrix type with G and to a zero matrix type with Z.

For example, the matrix L in (4.2) has the following SInfo:

\[
\text{L.SInfo} = \begin{cases} 
G : \{(i,j) | 0 \leq i < 4 \land 0 \leq j \leq i\} \\
Z : \{(i,j) | 0 \leq i < 4 \land i < j < 4\}
\end{cases}.
\]

This means that every scalar element in region L.SInfo[G] has general structure, while every element in L.SInfo[Z] has a zero structure. Note that this method allows the definition of blocked structures (e.g., where the top left quadrant is symmetric), which appear in several applications.
Similarly, we define the $S_{\text{Info}}$ dictionaries of $A$, $U$, and $S$:

$$U.S_{\text{Info}} = \{ G : \{(i, j) | 0 \leq i < 4 \land 0 \leq j < 4\} \},$$

$$Z : \{(i, j) | 0 \leq i < 4 \land 0 \leq j < i\},$$

$$A.S_{\text{Info}} = S.S_{\text{Info}} = \{ G : \{(i, j) | 0 \leq i, j < 4\} \}.$$

An $A_{\text{Info}}$ associates regions of a matrix to information on how to access their elements. Entries for $A_{\text{Info}}$ have the general form

$$\sigma : (g : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{r \times c}, p : \mathbb{R}^{r \times c} \rightarrow \mathbb{R}^{r \times c}),$$

where $\sigma$ is a polyhedral set, $g$ is a gather operator (see Section 3.2.3), and $p$ a permutation operator that can be applied to a gathered block. In other terms, in region $\sigma$ a block should be accessed using the composed operator $p(g(\cdot))$. For example, assuming the symmetric matrix $S$ in (4.2) stores only its lower part, it has the following $A_{\text{Info}}$:

$$\{(i, j) | 0 \leq i < 4, 0 \leq j \leq i\} : ([i,j]_{1,1}^{4,4}, \text{id}) \),

$$\{(i, j) | 0 \leq i < 4, 0 \leq j < 4\} : ([i,j]_{1,1}^{4,4}, \text{id}) \},$$

where $\text{id}$ is the identity permutation. Accessing element $(0, 3)$ would yield $\text{id}(S[3, 0]) = S[3, 0]$. For matrices $A$, $L$, and $U$ the accesses are unmodified:

$$A.A_{\text{Info}} = \{(i, j) | 0 \leq i < 4, 0 \leq j \leq i\} : ([i,j]_{1,1}^{4,4}, \text{id}) \},$$

$$L.A_{\text{Info}} = \{(i, j) | 0 \leq i < 4, 0 \leq j \leq i\} : ([i,j]_{1,1}^{4,4}, \text{id}) \},$$

$$U.A_{\text{Info}} = \{(i, j) | 0 \leq i < 4, 0 \leq j \leq i\} : ([i,j]_{1,1}^{4,4}, \text{id}) \}.$$

In the remainder of this section, we present our approach, and its implementation within LGEn, to generating optimized code for sBLACs using the polyhedral representation of matrix structures just presented.

### 4.2.2 Scalar Code Generation

As a running example we will use the sBLAC from (4.2):

$$A = LU + S, \quad A, L, U, S \in \mathbb{R}^{4 \times 4},$$

which contains three differently structured matrices. First, we assume scalar (non-vectorized) code as output.

The steps in the generation closely follow the ones of the original LGEn summarized at the beginning of this chapter, however with several important changes as described here.
Table 4.1: Examples of structure inference rules.

\[
M \times M \rightarrow M, \quad M \in \{G, L, U\}, \quad \star \in \{+, \cdot\} \\
\alpha M \rightarrow M, \quad M \in \{G, L, U, S\} \\
L^T = U, \quad U^T = L, \quad S^T = S \\
MM^T is S, \quad M is M \in \{G, L, U, S\} \\
M is L, \quad U \Rightarrow [M]_{r,r} is L, \quad U
\]

(4.9)  
(4.10)  
(4.11)  
(4.12)  
(4.13)  

Figure 4.2: Architecture of the \(\Sigma\)-CLooG rewriting module. The module is an extension of the original rewriting system of LGEN presented in Section 3.2.3.

**Step 1: Tiling and Structure Inference.** Given the LL program in Figure 4.1 as an input, Step 1 proceeds as discussed in Section 3.2.2. In addition, structure information is propagated to the implicit matrices in the expression tree using type inference rules from well-known mathematical properties such as those provided in Table 4.1. In our example, both LU and LU + S are of type G.

**Step 2: From LL to \(\Sigma\)-LL.** The rewriting system mentioned in Section 3.2.3 is substituted with an intermediate new module called \(\Sigma\)-CLooG based on the CLooG generator presented in Section 4.1.1. \(\Sigma\)-CLooG is schematically shown in Figure 4.2. It consists of two main components: (i) the statement generator StmtGen and (ii) an extended version of CLooG with a backend to output \(\Sigma\)-LL.
The input to $\Sigma$-CLooG is an sBLAC in LL from Step 1 and its output a translation of the input into an equivalent $\Sigma$-LL formulation. For example, given the sBLAC (4.2), the following $\Sigma$-LL statement is a possible output:

\[
A = \sum_{i=0}^{2} \left( \sum_{j=0}^{i} [i, j](L[i, 0]U[0, j] + S[i, j]) + \sum_{j=i+1}^{3} [i, j](L[i, 0]U[0, j] + S[j, i]) \right) + \sum_{j=0}^{3} [3, j](L[3, 0]U[0, j] + S[3, j]) + \sum_{k=1}^{3} \sum_{i=k}^{3} \sum_{j=k}^{3} [i, j](L[i, k]U[k, j]).
\]  

(4.14) (4.15) (4.16) (4.17)

Note that redundant multiplications (with zero) do not occur and that the symmetry of $S$ has been taken into account (i.e., only the part below the diagonal is accessed).

To achieve this, the input sBLAC is transformed using the information $SInfo$ and $AInfo$ of the matrices. This information is used to produce a set of CLooG statements. Every such statement is a triplet $\langle$body, domain, schedule$\rangle$ where: (a) domain is a polyhedral set $\sigma$ representing the iteration domain of the statement; (b) schedule is a polyhedral map $\rho$ that determines the traversal or scanning order of the domain’s tuples; (c) body is a $\Sigma$-LL expression $B$. For example, the statement

\[
s = \langle \sigma = \{(i, k, j) \mid k = 0 \land 0 \leq i < 4 \land 0 \leq j \leq i \}, \\
\rho = ((i, k, j), (k, i, j)), \\
B = [i, j](L[i, k]U[k, j] + S[i, j])\rangle
\]

(4.18)

is used to generate (4.14) and (4.16) (which is (4.14) with $i = 3$, split off). In particular, the domain specifies the range of the indices appearing in the body and the schedule their order. Next we describe how StmtGen recursively creates statements such as (4.18) by processing bottom-up the LL expression tree of the input sBLAC. We first explain the creation of domains and bodies. Then, once the entire tree is processed, we present how schedules are determined from it.

**Step 2.1: Generating Domains and Bodies for Operations on Leaves.**

The first operation performed by StmtGen is the creation of a unique index space for the input sBLAC. For our running example, three indices are needed:

\[
A_{[i,j]} = L_{[i,k]}U_{[k,j]} + S_{[i,j]}.
\]
Such an index space is then used to expand the SInfo and AInfo dictionaries of the occurring matrices. In our case, the regions of the structured matrices are expanded to prisms:

\[
\begin{align*}
L.SInfo &= \{ (i, k, j) | 0 \leq i < 4 \wedge 0 \leq k \leq i \} \\
U.SInfo &= \{ (i, k, j) | 0 \leq k < 4 \wedge k \leq j \leq 4 \} \\
A.SInfo &= S.SInfo = \{ (i, k, j) | 0 \leq i < 4 \wedge 0 \leq i < 4 \}.
\end{align*}
\]

Similarly, the following AInfos are computed:

\[
\begin{align*}
L.AInfo &= \{ (i, k, j) | 0 \leq i < 4 \wedge 0 \leq k \leq i \} : \begin{bmatrix} i, k \end{bmatrix}_{4,4}, id \\
U.AInfo &= \{ (i, k, j) | 0 \leq k < 4 \wedge k \leq j \leq 4 \} : \begin{bmatrix} k, j \end{bmatrix}_{4,4}, id \\
S.AInfo &= \{ (i, k, j) | 0 \leq i < 4, 0 \leq j \leq i \} : \begin{bmatrix} i, j \end{bmatrix}_{4,4}, id \\
A.AInfo &= \{ (i, k, j) | 0 \leq i, j < 4 \} : \begin{bmatrix} i, j \end{bmatrix}_{4,4}, id \}.
\end{align*}
\]

Next, StmtGen builds a set of statements for every operator in the input sBLAC bottom-up, starting from the inputs. In our case the first operation is LU. To build statements for LU we begin by determining its iteration space. In general, the iteration space for matrix multiplication is a cuboid (Figure 4.3(a)). However given the presence of zero regions in (4.19) and (4.20), the redundant zero computations can be excluded (Figure 4.3(b)) by computing the iteration space as

\[
iterSpace_{LU} = L.SInfo[\mathcal{G}] \cap U.SInfo[\mathcal{G}] = \{ (i, k, j) | 0 \leq k < 4 \wedge k \leq i, j < 4 \}.
\]

In general (e.g., for vectorization), our approach computes the iteration spaces for all combinations of nonzero operands (e.g., \( \mathcal{G}\mathcal{G} \), \( \mathcal{G}\mathcal{L} \), etc.) using Algorithm 4.1.

**Algorithm 4.1** Computing the iteration space for matrix-matrix multiplication.

Input: matrices \( I_0 \) and \( I_1 \).

Output: The iteration space (iterSpace) of \( I_0I_1 \).

\[
\text{iterSpace} \leftarrow \emptyset \\
\text{for} \ (M_0 : \sigma_0) \in I_0.SInfo : M_0 \neq \emptyset \ \text{do} \\
\ \ \ \ \text{for} \ (M_1 : \sigma_1) \in I_1.SInfo : M_1 \neq \emptyset \ \text{do} \\
\ \ \ \ \ \ \ \ // Iteration space based on all pairs of non-zero, input regions. \\
\ \ \ \ \ \ \ \ \ \text{iterSpace} = \text{iterSpace} \cup (\sigma_0 \cap \sigma_1); \\
\ \ \ \ \text{end for} \\
\ \ \ \text{end for}
\]

4.2 Code Generation with Structured Matrices

Output Initialization. Matrix-matrix multiplication accumulates along the k-axis of its iteration space. In general, the \( \text{StmtGen} \) module has to decide whether it is necessary to initialize the output or not. Our input sBLAC is not an update computation and therefore \( \text{StmtGen} \) cannot use the output as an accumulator without initializing it first. At this point, it could decide either to introduce zeroing statements and then accumulate across the whole \( \text{iterSpace}_{LU} \) or to identify the set of points of \( \text{iterSpace}_{LU} \) that first touch the output, splitting it from all remaining points of \( \text{iterSpace}_{LU} \).

We describe the latter option, which reduces to the problem of identifying the region of points in the iteration space with minimum coordinates along the axes of accumulation. In our example, the computation accumulates along the k-axis and \( \text{StmtGen} \) needs to identify the following initialization set:

\[
\text{iterSpace}_{LU}^{\text{init}} = \{(i, k, j) | (i, k, j) \in \text{iterSpace}_{LU},\]
\[
\#(i, k', j) \in \text{iterSpace}_{LU} : k' \leq k\}
\[
= \{(i, 0, j) | 0 \leq i < 4 \land 0 \leq j < 4\}.
\]

Splitting the region above from the rest of the iteration space \( \text{StmtGen} \) also obtains the accumulation points:

\[
\text{iterSpace}_{LU}^{\text{acc}} = \text{iterSpace}_{LU} \setminus \text{iterSpace}_{LU}^{\text{init}}
\]
\[
= \{(i, k, j) | 1 \leq k < 4 \land k \leq i, j < 4\}.
\]

The two partial iteration spaces above are illustrated in Figure 4.4.
Figure 4.4: Iteration space of LU split into output initialization (black dots) and output accumulation (gray dots) space.

**Fixing Domains and Bodies.** To derive the final domains and bodys associated with the two partial iteration spaces above, these need to be intersected with the regions in the respective \( A\text{Info} \) dictionaries, which explain how matrices are accessed. Since in our examples only symmetric matrices have special access, nothing changes:

\[
\begin{align*}
dom_{\text{init}}^{LU} &= \text{iterSpace}_{\text{init}}^{LU}, \\
dom_{\text{acc}}^{LU} &= \text{iterSpace}_{\text{acc}}^{LU},
\end{align*}
\]

and using the gathers from \( A\text{Info} \) \( \text{StmtGen} \) constructs the associated bodies (which in this case are the same):

\[
\begin{align*}
B_{\text{init}}^{LU} &= B_{\text{acc}}^{LU} = [i, j](L[i, k]U[k, j]).
\end{align*}
\]

Two statements are thus obtained:

\[
\begin{align*}
s_{\text{init}}^{LU} &= \langle \text{dom}_{\text{init}}^{LU}, \emptyset, B_{\text{init}}^{LU} \rangle, \\
s_{\text{acc}}^{LU} &= \langle \text{dom}_{\text{acc}}^{LU}, \emptyset, B_{\text{acc}}^{LU} \rangle.
\end{align*}
\]

The schedules are left empty as they will be computed last. The general version of this approach for arbitrarily structured inputs is shown in Algorithm 4.2.

**Step 2.2: Generating Domains and Bodies for Operations Recursively.** As mentioned, the generation of domains and bodies is done bottom-up. In our example, the operation following LU is the addition \( LU + S \). For the computation of its CLooG statements, \( \text{StmtGen} \) uses an approach similar to the one used before. However, as LU is not an input matrix, its set of (already gener-
Algorithm 4.2 Building CLooG statements for matrix-matrix multiplication. One statement is created for every combination of input and output regions that intersect the iteration space. Schedules are generated separately.

Input: iterSpace\textsuperscript{init}, iterSpace\textsuperscript{acc}, I_0, I_1, and T.

Output: CLooG statements (stmts) for T = I_0 I_1.

\[
\text{stmts} \leftarrow \emptyset
\]

\[
\text{for } (\sigma_0 : (g_0, p_0)) \in I_0.A\text{Info} \text{ do}
\]

\[
\text{for } (\sigma_1 : (g_1, p_1)) \in I_1.A\text{Info} \text{ do}
\]

\[
\text{for } (\sigma_T : (g_T, p_T)) \in T.A\text{Info} \text{ do}
\]

\[
\text{for } \sigma_{\text{space}} \in \{ \text{iterSpace}^{\text{init}}, \text{iterSpace}^{\text{acc}} \} \text{ do}
\]

\[
\text{dom} \leftarrow \sigma_0 \cap \sigma_1 \cap \sigma_T \cap \sigma_{\text{space}}
\]

\[
\text{if } \text{dom} \neq \emptyset \text{ then}
\]

Gather + permute inputs and multiply.
\[
m \leftarrow p_0(g_0(I_0)) \cdot p_1(g_1(I_1))
\]

Permute + scatter output.
\[
B \leftarrow g_T^{-1}(p_T^{-1}(m))
\]

Save new statement.
\[
\text{stmts} \leftarrow \text{stmts} \cup \{(\text{dom}, \emptyset, B)\}
\]

\[
\text{end if}
\]

\[
\text{end for}
\]

\[
\text{end for}
\]

\[
\text{end for}
\]

\[
\text{end for}
\]

ated) CLooG statements is used as input this time. As before, StmtGen computes first the iteration space. Using (4.21) it gets the trivial result

\[
\text{iterSpace} = \{(i, k, j) \in \sigma \mid (M : \sigma) \in A.S\text{Info}, M \neq \emptyset\}
\]

\[
= \{(i, k, j) \in A.S\text{Info}[\mathcal{G}]\}
\]

\[
= \{(i, k, j) \mid 0 \leq i, j < 4\},
\]

where A is the output of the operation.

Next, it derives the CLooG statements, i.e., a possible splitting into domains and the associated bodies using a general algorithm for matrix addition that operates analogous to Algorithm 4.2. It does this by intersecting iterSpace with (a) the domain of s^{\text{init}}_{\text{LU}} and (b) the regions from S.A\text{Info} in (4.22). Since there are two such regions (here denoted with \sigma_{S,0} and \sigma_{S,1}) two domains are computed. Both have initialization accesses only and accumulating accesses do not occur:

\[
\text{dom}_0 = \text{iterSpace} \cap \text{dom}_{\text{LU}}^{\text{init}} \cap \sigma_{S,0}
\]

\[
= \{(i, 0, j) \mid 0 \leq i < 4, 0 \leq j \leq i\},
\]

\[
\text{dom}_1 = \text{iterSpace} \cap \text{dom}_{\text{LU}}^{\text{init}} \cap \sigma_{S,1}
\]

\[
= \{(i, 0, j) \mid 0 \leq i < 4, i < j < 4\}.
\]
Using $s_{LU}^{init}$ and $s_{AInfo}$ we compute the associated two bodies:

$$B_0 = [i, j] \left( [i, j]([L[i, k]U[k, j]][i, j] + S[i, j]) \right)$$

$$= [i, j] \left( L[i, k]U[k, j] + S[i, j] \right),$$

$$B_1 = [i, j] \left( [i, j]([L[i, k]U[k, j]][i, j] + S[j, i]) \right)$$

$$= [i, j] \left( L[i, k]U[k, j] + S[j, i] \right).$$

With the new domains and bodies, StmtGen can finally construct the statements that lead to the final output in (4.14)–(4.17):

$$s_0 = (\text{dom}_0, 0, B_0), \quad s_1 = (\text{dom}_1, 0, B_1), \quad s_2 = s_{LU}^{acc}.$$

Before feeding the statements to CLooG, StmtGen needs to complete them with schedules.

We emphasize that the method sketched here on a simple example can correctly derive and exploit intermediate structures including blocks in multi-level blocking of expressions as complex as, e.g., $A = (L_0 + L_1)S_1 + xx^T$.

**Step 2.3: Building the Schedules.** After step 2.2 the root operator contains all necessary statements for the given sBLAC albeit without schedules. To add the schedules, StmtGen first computes a global order over the index space of the sBLAC. This can be done by assuming performance models such as those of the original LGen presented in Section 3.2.4. For example, assuming they provide the order $(k, i, j)$, StmtGen can define the schedule $\rho = \{(i, k, j), (k, i, j)\}$. Completing $s_0$, $s_1$, and $s_2$ with $\rho$, CLooG produces the expression in (4.14)–(4.17) as the input to the next step in LGen.

**Steps 3 to 6: From $\Sigma$-LL to Output Code.** For scalar code generation the remaining three steps are similar to the original LGen (Sections 3.2.4–3.2.7). From the $\Sigma$-LL statement in (4.14)–(4.17), LGen generates the C code in Figure 4.5.

### 4.3 Vectorization Approach

Enabling vectorization, introduces at least one level of tiling for $\nu$-BLACs as explained in Section 3.3.1. We now discuss how this affects the internal representation of structures. We again use our example sBLAC (4.2) assuming we want to vectorize for a microarchitecture with a 2-way vector ISA ($\nu = 2$).
for( int i = 0; i <= 2; i++ ) {
    for( int j = 0; j <= i; j++ ) {
    }
    for( int j = i + 1; j <= 3; j++ ) {
    }
}
for( int j = 0; j <= 3; j++ ) {
}
for( int k = 1; k <= 3; k++ ) {
    for( int i = k; i <= 3; i++ ) {
        for( int j = k; j <= 3; j++ ) {
            A[4*i+j] += L[4*i+k] * U[4*k+j];
        }
    }
}

Figure 4.5: Output C code for sBLAC (4.2).

4.3.1 Internal Representation of Tiled Structures

When a structured matrix is $\nu$-tiled for vector instructions, it is viewed as a matrix of $\nu \times \nu$ blocks. Viewed like this, the matrix will still have structure. For example, an $L$ or $U$ matrix has the same structure when $\nu$-tiled. In principle, this could be derived automatically. We chose to incorporate this information into our system by providing the associated definitions of $S\text{Info}$ and $A\text{Info}$ for the blocked matrix in each case and for a generic block size. This definition can then be instantiated for specific cases. For example, for a $\nu$-tiled symmetric matrix (instantiated for $\nu = 2$) one gets

$$
[S]_{2,2}.S\text{Info} = \left\{ \begin{array}{l}
S : \{(0,2),(2,0)\} \\
S : \{(0,0),(2,2)\}
\end{array} \right\},
$$

$$
[S]_{2,2}.A\text{Info} = \left\{ \begin{array}{l}
\{(0,0),(2,0),(2,2)\} : ([i,j]_{2,2}^{4,4}, \text{id}) \\
\{(0,2)\} : ([j,i]_{2,2}^{4,4}, (\cdot)^{T})
\end{array} \right\}.
$$

This specifies, for example, that the tile at $(0,2)$ is accessed as $S[2,0]^T$. Next we sketch how these new definitions interact with the approach of $\Sigma$-CLoooG.
4.3.2 Vector Code Generation

Vectorization introduces a coarser basic block definition for the matrices. From the $\Sigma$-CLooG perspective, this only means the construction of sparser domains of the statements, where polyhedral points are accessed at a stride as in (4.8). The approach taken in Algorithm 4.2 (and similarly those taken by the other operators) would then derive more structure combinations. For example, consider the computation of $[L]_{2,2}[U]_{2,2}$. The iteration space would be constructed based on the following combination of structures:

$$
\begin{align*}
L & \ U & G & L & G \\
G & L & U & U & G & G \\
& & & & & G
\end{align*}
$$

This yields four initialization statements for the four different structure combinations (i.e., $\mathcal{L} U$, $\mathcal{L} G$, $\mathcal{G} U$, and $\mathcal{G} G$ in the first output square) and a single accumulation statement. Completing with addition, and using the schedule defined in Section 4.2.2, step 2.3, it produces the following $\Sigma$-LL output:

$$
A = [0, 0](L[0, 0]U[0, 0] + S[0, 0]) \\
+ [0, 2](L[0, 0]U[0, 2] + (S[2, 0])^T) \\
+ [2, 0](L[2, 0]U[0, 0] + S[2, 0]) \\
+ [2, 2](L[2, 0]U[0, 2] + S[2, 2]) \\
+ [2, 2](L[2, 2]U[2, 2]).
$$

The above expression is completely decomposed into $\nu$-BLACs and thus in principle mappable to vector code. However, it features different kinds of tiles (e.g., $L[0, 0]$ is $L$, $L[2, 0]$ is $G$, and $S[0, 0]$ is $S$) that enforce different kinds of computations (e.g., $L[0, 0]U[0, 0]$ is an $\mathcal{L} U$ multiplication while $L[2, 0]U[0, 0]$ is $\mathcal{G} U$). Simply ignoring the structure by using generic $\nu$-BLACs is not possible since, by convention, data accesses above the diagonal are not allowed for $L$, $U$, and $S$.

**Mapping structures to vector code.** As explained in Section 3.3, the translation between $\Sigma$-LL and C-IR is based on three collections of codelets called Loaders, Storers, and $\nu$-BLACs. The first two handle data accesses while the latter does the computation. When mapping structured $\nu$-BLACs to vector code we use the generic computation but extend the Loaders and Storers to prevent illegal accesses. For example, consider the load of the lower triangular block $L[0, 0]$. The expected behavior of the Loader would be the following:

$$
\begin{pmatrix}
\ell_{0,0} \\
\ell_{1,0} & \ell_{1,1}
\end{pmatrix}
\xrightarrow{\text{Load}}
\begin{pmatrix}
\ell_{0,0} & 0 \\
\ell_{1,0} & \ell_{1,1}
\end{pmatrix}.
\quad (4.23)
$$
Here, a 0 is inserted by the Loader in place of x and used in the computation. Once matrices are loaded, the computations can be performed using the original 18 ν-BLACs (a slight inefficiency) introduced in Section 3.3.1.

4.4 Extensibility to New Structures

We designed our approach to be extensible to new structures. Provided a structure can be described using isl (see Section 4.1.2), including it into LGEn’s types requires the addition of:

- A structure definition: $\text{SInfo}$ and $\text{AInfo}$ dictionaries (Section 4.2.1).
- A set of Loaders and Storers: vectorized codelets for accessing ν-sized matrices with the new structure (Section 4.3.2).

As an example, we briefly discuss the addition of banded matrices with bandwidth k illustrated below:

As in the triangular case, the scalar definition of their $\text{SInfo}$ would contain two regions, a general region for the band and one or two zero regions outside the band. For example, assuming the matrix above has size $m \times n$, it would have the following $\text{SInfo}$:

$$
\begin{align*}
\text{G} : & \{ (i,j) \mid 0 \leq i < m \land \max(0, i - k) \leq j < \min(n, i + k) \} \\
\text{Z} : & \{ (i,j) \mid 0 \leq i < m \land (0 < j < i - k \lor i + k \leq j < n) \}
\end{align*}
$$

(4.24)

Producing vector code for a ν-way vector ISA would require special Loaders and Storers at the sub- and super-diagonals. For example, consider ν-tiling the matrix above assuming it has dimensions $4\nu \times 4\nu$. If ν|k, then the new $4 \times 4$ matrix of matrices would have tiles of the following types:

$$
\begin{bmatrix}
\text{G} & \text{G} & \text{L} & \text{Z} \\
\text{G} & \text{G} & \text{G} & \text{L} \\
\text{U} & \text{G} & \text{G} & \text{G} \\
\text{Z} & \text{U} & \text{G} & \text{G}
\end{bmatrix}
$$

(4.25)
where the types $L$ and $U$ are strictly lower and upper triangular, respectively. Otherwise, if $\nu \not| k$, tiling would create the following type subregions:

\[
\begin{bmatrix}
\mathcal{L} & \mathcal{K} & \bar{L} & \mathcal{Z} \\
\mathcal{J} & \mathcal{G} & \mathcal{K} & \bar{L} \\
\bar{U} & \mathcal{J} & \mathcal{G} & \mathcal{K} \\
\mathcal{Z} & \bar{U} & \mathcal{J} & \mathcal{G}
\end{bmatrix},
\]

(4.26)

where $\mathcal{J}$ and $\mathcal{K}$ are upper and lower Hessenberg types, respectively. If a banded matrix is also symmetric then support can be added by combining the $\text{sInfo}$ description sketched in (4.25) and (4.26) with an $\text{AInfo}$ similar to the one described for the general case in Section 4.2.1.

Blocked structures as, for example,

\[
\begin{bmatrix}
\mathcal{G} & \mathcal{L} \\
\mathcal{S} & \mathcal{U}
\end{bmatrix},
\]

can be added by recursively fusing the $\text{sInfo}$ and $\text{AInfo}$ dictionaries of the occurring structures. This is possible since isl supports unions of regions as shown in (4.7).

EXPERIENCE IN EXTENSIBILITY. As part of a three-month project at the Department of Mathematics of ETH Zurich, a student tested the approach by extending LGEN with two additional types: The tridiagonal type and the uniform composition type.

The tridiagonal type is a special type of banded matrix that can described with (4.24) for $k = 1$. Matrices of this type were then used to generate code for solvers of tridiagonal systems with LGEN. We will present an extension of LGEN for higher level computations, including solving systems of equations, in the next chapter.

The uniform composition type was defined as a parametric type, where a basic structure is replicated several times both vertically and horizontally. Matrices with such structure can occur in the context of multi-agent control systems [19]. An example is the following matrix obtained by composing $2 \times 3$ upper triangular matrices:

The following is an example of the declaration in LL of the matrix above using the new uniform composition type:

\[ A : \text{Uniform} < \text{UpperTriangular}(4) > (2,3); \]
where \( A \) is an \( 8 \times 12 \) matrix composed of 6 smaller, upper triangular matrices of size \( 4 \times 4 \). In general, a uniform composition type replicates an existing LL matrix type \( M \) in a new \( m \times n \) grid structure:

\[
\langle M \rangle_{m \times n} := \begin{bmatrix}
M & \cdots & M \\
\vdots & \ddots & \vdots \\
M & \cdots & M
\end{bmatrix}^m, \quad M \in \{L, U, S, \ldots\}.
\]
In this chapter we make one final step towards the goal of this work presented in Section 1.1: Generating efficient code starting from the specification of a linear algebra computation. We begin by going back to our motivating example from Chapter 1, the Kalman filter, that we show again in Table 5.1. The table shows a single iteration of a basic Kalman filter.

In Section 1.1, discussing the filter in more details, we identified two classes of computations: Basic linear algebra computations (BLACs), possibly with structures (sBLACs, e.g., the first two equations 5.1 and 5.2), and higher-level computations (e.g., the Cholesky decomposition and solvers for linear systems needed to efficiently compute the expressions involving inverses in (5.3) and (5.4)).

In Chapters 3 and 4, we explained how to generate code for BLACs and sBLACs with LGen. In this chapter we include support for higher-level computations, presenting a system for program synthesis that, given a linear algebra computation, such as the Kalman filter in Table 5.1, outputs efficient code, optionally vectorized using intrinsics.

The design of our new generator combines and expands the approaches of two systems: The LGen compiler, introduced in the two previous chapters, and the CLICK [29, 30] generator. CLICK automatizes the FLAME methodology [13, 107] and synthesizes blocked algorithms, expressed using BLAS functions, for certain higher-level linear algebra functions; LGen, as described so far, generates

Table 5.1: Single iteration of a Kalman filter.

\[ x_{k|k-1} = F x_{k-1|k-1} + B u \]  
\[ P_{k|k-1} = F P_{k-1|k-1} F^T + Q \]  
\[ x_{k|k} = x_{k|k-1} + P_{k|k-1} H^T \]  
\[ \times (H P_{k|k-1} H^T + R)^{-1} (z_k - H x_{k|k-1}) \]  
\[ P_{k|k} = P_{k|k-1} - P_{k|k-1} H^T \]  
\[ \times (H P_{k|k-1} H^T + R)^{-1} H P_{k|k-1}. \]
optimized code for a single, basic linear algebra computation on structured matrices. The combination of these two approaches involves a number of extensions presented in Section 5.2 and described in practice in Section 5.3.

First, however, we review FLAME and C\textsc{lick} in the next section.

5.1 AUTOMATIC ALGORITHM DISCOVERY

In Section 2.2, we introduced as background the FLAME notation. In particular, we highlighted that it describes a linear algebra algorithm as composed of five parts (parts (a)–(e) in Algorithm 2.7). In this section, we show as prior work and background that those five parts can be derived mechanically using the FLAME methodology. For more details on FLAME, we refer to [13, 107].

5.1.1 FLAME: Mechanical Derivation of Algorithms

The FLAME methodology starts from a specification of a computation, such as the lower triangular system presented in Section 2.2.3:

\[
LX = B, \quad L \in \mathcal{L}_n, \quad X, B \in \mathbb{R}^{n \times m},
\]  

and systematically derives a correct algorithm for it, such as the one shown in Algorithm 5.1. The latter differs from Algorithm 2.9 in two ways: (i) The output is stored separately and (ii) the algorithm is blocked, i.e., the thick lines in Algorithm 5.1(e), move more than one element and row at a time (e.g., \(\lambda_{1,1}\) is now a matrix \(L_{11}\) of size \(b \times b\)).

The five parts of a FLAME algorithm (i.e., (a)–(e) in Algorithm 5.1) are derived using the three steps that we briefly summarize next. A more detailed derivation of the same algorithm is discussed in Appendix B.

**STEP F1: DETERMINING THE PARTITIONED MATRIX EXPRESSION.** The first step partitions the operands in (5.5). For example, since \(L\) is lower triangular, partitioning it into quadrants\(^1\) maintains the properties on the two diagonal partitions:

\[
L \rightarrow \left( \begin{array}{c|c}
L_{TL} & 0 \\
\hline
L_{BL} & L_{BR}
\end{array} \right), \quad L_{TL} \in \mathcal{L}_k, \quad L_{BR} \in \mathcal{L}_{n-k}, \quad 0 \leq k \leq n,
\]

\(\text{\textsuperscript{1}}\) The subscripts indicate (L)eft, (R)ight, (T)op, and (B)ottom partitions.
Algorithm 5.1 Solving a lower triangular system (Forward substitution algorithm, blocked variant 2).
LX = B, \( L \in \mathcal{L}_n \), X, B \( \in \mathbb{R}^{n \times m} \).
Initially, the temporary matrix \( T = B \). Flops \( \approx mn^2 \).

\[ \begin{align*}
LX & = B, \quad L \in \mathcal{L}_n, \quad X, B \in \mathbb{R}^{n \times m}. \\
\text{Initially, the temporary matrix } T & = B. \quad \text{Flops } \approx mn^2.
\end{align*} \]

(a) **Partition** \( L \rightarrow \begin{pmatrix} L_{TI} & 0 \\ L_{BL} & L_{BR} \end{pmatrix} \), \( T \rightarrow \begin{pmatrix} T_T \\ T_B \end{pmatrix} \), \( X \rightarrow \begin{pmatrix} X_T \\ X_B \end{pmatrix} \)

where \( L_{TI} \) is \( 0 \times 0 \) and \( X_T \) is \( 0 \times m \)

(b) **while** \( \text{size}(X_T) < \text{size}(X) \) **do**

(c) **Repartition**

\[ \begin{pmatrix} L_{TI} & 0 \\ L_{BL} & L_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} L_{0,0} & 0 & 0 \\ L_{1,0} & L_{1,1} & 0 \\ L_{2,0} & L_{2,1} & L_{2,2} \end{pmatrix}, \quad \begin{pmatrix} T_T \\ T_B \end{pmatrix} \rightarrow \begin{pmatrix} T_0 \\ T_1 \\ T_2 \end{pmatrix}, \]

\[ \begin{pmatrix} X_T \\ X_B \end{pmatrix} \rightarrow \begin{pmatrix} X_0 \\ X_1 \\ X_2 \end{pmatrix} \]

where \( L_{1,1} \) is \( b \times b \) and \( X_1 \) is \( b \times m \)

(d) \( X_1 := T_1 L_{1,1}^{-1} \)
\( T_2 := T_2 - L_{2,1} X_1 \)

(e) **Continue with**

\[ \begin{pmatrix} L_{TI} & 0 \\ L_{BL} & L_{BR} \end{pmatrix} \leftarrow \begin{pmatrix} L_{0,0} & 0 & 0 \\ L_{1,0} & L_{1,1} & 0 \\ L_{2,0} & L_{2,1} & L_{2,2} \end{pmatrix}, \quad \begin{pmatrix} T_T \\ T_B \end{pmatrix} \leftarrow \begin{pmatrix} T_0 \\ T_1 \\ T_2 \end{pmatrix}, \]

\[ \begin{pmatrix} X_T \\ X_B \end{pmatrix} \leftarrow \begin{pmatrix} X_0 \\ X_1 \\ X_2 \end{pmatrix} \]

**endwhile**

where the size of the partitions depend on \( k \) and varies during the execution of the loop-based algorithm. This choice also imposes a consistent partitioning of \( X \) and \( B \), such as the following:

\[ B \rightarrow \begin{pmatrix} B_T \\ B_B \end{pmatrix}, \quad X \rightarrow \begin{pmatrix} X_T \\ X_B \end{pmatrix}, \quad \text{(5.7)} \]

where \( \text{rows}(B_T) = \text{rows}(X_T) = \text{cols}(L_{TI}). \)
Table 5.2: Loop invariants of (5.5) obtained from the PME in (5.8). The matrix-like notation on the right alternatively describe the loop invariants providing a link between the predicates and their positions in the PME.

\[(L_{TL}X_T = B_T) \equiv \left( \begin{array}{c} L_{TL}X_T = B_T \\ T_B = B_B - L_BLX_T \end{array} \right)\]  
\[ (L_{TL}X_T = B_T) \land (T_B = B_B - L_BLX_T) \equiv \left( \begin{array}{c} L_{TL}X_T = B_T \\ T_B = B_B - L_BLX_T \end{array} \right)\]

Plugging (5.6) and (5.7) in (5.5) determines a partitioned matrix expression (PME) of (5.5):

\[
\begin{pmatrix}
L_{TL} & 0 \\
L_BL & L_{BR}
\end{pmatrix}
\begin{pmatrix}
X_T \\
X_B
\end{pmatrix}
= \begin{pmatrix}
B_T \\
B_B
\end{pmatrix}
\]

\[
\rightarrow \begin{pmatrix}
L_{TL}X_T = B_T \\
L_{BR}X_B = B_B - L_BLX_T
\end{pmatrix}, \quad (5.8)
\]

which recursively expresses the initial computation in terms of its partitioned operands (computing both \(X_T\) and \(X_B\) requires to solve a smaller triangular system).

The partial computations in a PME could show dependencies between each other. For instance, the following order must be respected for a correct computation of (5.8):

1. \(L_{TL}X_T = B_T\)  
2. \(T_B = B_B - L_BLX_T\)  
3. \(L_{BR}X_B = T_B\)

where \(T = B\) is introduced as a temporary matrix, as the algorithm is not allowed to overwrite the input in this example.

**Step F2: Determining Loop Invariants.** From a PME, it is possible to derive loop invariants, i.e., predicates that hold before and after the algorithm’s loop as well as before and after each of its iterations.

For example, analyzing the PME in (5.8), we can determine the two loop invariants in Table 5.2. The presence of output partitions in a loop invariant indicate that their computation has been already performed.

**Step F3: Building the Algorithm.** Deriving a loop invariant also produces an initial partitioning and loop guard. In our example, as shown in the
Previous paragraph, Algorithm 5.1, parts (a) and (b), can be associated with both invariants in Table 5.2.

Algorithm 5.1, parts (c) and (e), are prescribed by the choice of the initialization and the loop guard. For instance, in Algorithm 5.1 parts (c) and (e), we chose $b > 1$ many rows of $X_T$. This imposes to progress by the same amount of rows in $B_T$ and by a $b \times b$ block in $L$, as we want to maintain the property of having triangular partitions on the diagonal.

At this point choosing a loop invariant makes the difference, as different ones yield different update statements. We choose the second one which yields Algorithm 5.1 (for this reason labelled “variant 2”). Specifically, Algorithm 5.1, part (d), is computed by comparing the state of the computation after repartition (i.e., Algorithm 5.1, part (c)) with the state of the computation after moving the thick lines in Algorithm 5.1, part (e).

**Methodology scope.** A necessary condition for the methodology to be applicable is that the PME has to exist, with no cyclic dependencies among the different statements. Examples of operations that can be handled include all computations introduced in Chapter 2.

The whole procedure described with three steps above can be automatized as demonstrated by the Click generator.

### 5.1.2 Click: FLAME Automated

*Click [29, 30]* is an algorithm generator that implements the FLAME methodology previously explained in the three major stages illustrated in Figure 5.1.

*Click* takes as input the description of a target operation in the form of an equation annotated with the properties of the operands. For instance, the input for our running example in (5.5) is specified in Figure 5.2. In our example, the input computation is also tagged with the label TRSM for better supporting pattern matching operations as described next.

**Generation of partitioned matrix expressions.** This stage implements Step F1 in Section 5.1.1. Supported by a linear algebra knowledge-base and an engine for the inference of mathematical properties and structures, *Click*
82 program synthesis for linear algebra

\[ X = \text{TRSM}(L, B) \equiv \begin{cases} 
\text{equation} &: \{LX = B, \ L \in \mathbb{R}^{n \times n}, \ X, B \in \mathbb{R}^{n \times m}\}; \\
\text{properties} &: \{\text{Output}(X) \land \text{Input}(L, B) \\
&\land \text{LowerTriangular}(L)\}; 
\end{cases} \]

Figure 5.2: CLIck input example. It consists of a single linear algebra operation with inputs and output annotations.

1. \(X_T = \text{TRSM}(L_{TL}, B_T)\)
2. \(T_B = B_B - L_{BL}X_T\)
3. \(X_B = \text{TRSM}(L_{BR}, T_B)\)

Figure 5.3: Left: Decomposition of TRSM’s PME (5.8) into its building blocks. Right: Graph of dependencies. Here, the grey line highlights the source position of the statements within the PME, similarly to the notation on the right-hand side of (5.13).

generates possible PMEs from the input. In our example, one of three resulting PMEs for the TRSM input in Figure 5.2 is (5.8).

PMEs are not always unique. In fact, there exist \(2^d - 1\), where \(d\) is the number of problem dimensions (in the case of TRSM, \(d = ||m, n|| = 2\), since each dimension may either be partitioned or not (minus the case where no operand is partitioned).

Then pattern matching and term rewriting is applied to identify resulting subproblems (e.g., (5.9)–(5.11)) taking into account available properties and structures (e.g., \(L_{TL}\) and \(L_{BR}\) in (5.9) and (5.11) are also lower triangular). In particular, pattern matching allows the identification of recursive calls to the same higher-level function. For instance, the PME (5.8) is represented in CLIck as

\[
\begin{pmatrix}
X_T = \text{TRSM}(L_{TL}, B_T) \\
X_B = \text{TRSM}(L_{BR}, B_B - L_{BL}X_T)
\end{pmatrix}.
\]

(5.14)

**Loop invariants identification.** Following Step F2 in Section 5.1.1, from each PME a family of loop invariants is generated. First, CLIck uses tree tiling to decompose the PME into computations compatible with the BLAS interface. Then loop invariants are selected that satisfy a graph of dependencies among these computations.

For example, the dependencies between subproblems listed in (5.9)–(5.11) for the PME in (5.8) are represented as the graph illustrated on the right of Figure 5.3 (note also in this case the use of the recursive calls to TRSM), where the loop invariant in (5.13) is obtained from subgraph \{1, 2\}. 

loop invariants identification. Following Step F2 in Section 5.1.1, from each PME a family of loop invariants is generated. First, CLIck uses tree tiling to decompose the PME into computations compatible with the BLAS interface. Then loop invariants are selected that satisfy a graph of dependencies among these computations.

For example, the dependencies between subproblems listed in (5.9)–(5.11) for the PME in (5.8) are represented as the graph illustrated on the right of Figure 5.3 (note also in this case the use of the recursive calls to TRSM), where the loop invariant in (5.13) is obtained from subgraph \{1, 2\}. 

loop invariants identification. Following Step F2 in Section 5.1.1, from each PME a family of loop invariants is generated. First, CLIck uses tree tiling to decompose the PME into computations compatible with the BLAS interface. Then loop invariants are selected that satisfy a graph of dependencies among these computations.

For example, the dependencies between subproblems listed in (5.9)–(5.11) for the PME in (5.8) are represented as the graph illustrated on the right of Figure 5.3 (note also in this case the use of the recursive calls to TRSM), where the loop invariant in (5.13) is obtained from subgraph \{1, 2\}. 

loop invariants identification. Following Step F2 in Section 5.1.1, from each PME a family of loop invariants is generated. First, CLIck uses tree tiling to decompose the PME into computations compatible with the BLAS interface. Then loop invariants are selected that satisfy a graph of dependencies among these computations.

For example, the dependencies between subproblems listed in (5.9)–(5.11) for the PME in (5.8) are represented as the graph illustrated on the right of Figure 5.3 (note also in this case the use of the recursive calls to TRSM), where the loop invariant in (5.13) is obtained from subgraph \{1, 2\}. 

loop invariants identification. Following Step F2 in Section 5.1.1, from each PME a family of loop invariants is generated. First, CLIck uses tree tiling to decompose the PME into computations compatible with the BLAS interface. Then loop invariants are selected that satisfy a graph of dependencies among these computations.

For example, the dependencies between subproblems listed in (5.9)–(5.11) for the PME in (5.8) are represented as the graph illustrated on the right of Figure 5.3 (note also in this case the use of the recursive calls to TRSM), where the loop invariant in (5.13) is obtained from subgraph \{1, 2\}. 

loop invariants identification. Following Step F2 in Section 5.1.1, from each PME a family of loop invariants is generated. First, CLIck uses tree tiling to decompose the PME into computations compatible with the BLAS interface. Then loop invariants are selected that satisfy a graph of dependencies among these computations.

For example, the dependencies between subproblems listed in (5.9)–(5.11) for the PME in (5.8) are represented as the graph illustrated on the right of Figure 5.3 (note also in this case the use of the recursive calls to TRSM), where the loop invariant in (5.13) is obtained from subgraph \{1, 2\}. 

loop invariants identification. Following Step F2 in Section 5.1.1, from each PME a family of loop invariants is generated. First, CLIck uses tree tiling to decompose the PME into computations compatible with the BLAS interface. Then loop invariants are selected that satisfy a graph of dependencies among these computations.

For example, the dependencies between subproblems listed in (5.9)–(5.11) for the PME in (5.8) are represented as the graph illustrated on the right of Figure 5.3 (note also in this case the use of the recursive calls to TRSM), where the loop invariant in (5.13) is obtained from subgraph \{1, 2\}.
Algorithm Construction. Finally, each loop invariant is translated into an algorithm. At this final stage, Cl1ck implements Step F3 in Section 5.1.1 to find a loop guard, a traversal of the operands, and those loop statements that ensure that the loop invariant is satisfied before entering the loop, at the beginning and end of each iteration, and at the end of the loop. This process relies on structural pattern matching and term rewriting.

5.2 Linear Algebra Program Synthesis Overview

We present a new code generator that builds on Cl1ck, discussed in Section 5.1.2, and LGen, as presented in Chapter 4. The goal of our new system is to generate fast vectorized code for small to medium size linear algebra computations as complex as the Kalman filter shown in Table 5.1. In this section we provide a general description of our generator in terms of main changes and extensions to previous work, while in the following section we discuss in more details the generation process with a running example.

Generator Overview. Our generator integrates the algorithm discovery features from Cl1ck, and the code generation capabilities of LGen into a single framework, as shown in Figure 5.4.

In particular, Steps 1–3 in Figure 5.4 build on Cl1ck; these steps take as input a linear algebra program and generate algorithms for each higher level function it contains. We consider Steps 1–3 the algorithmic level of the new generator.

Using the generated algorithms, the LL backend (Step 4) composes a basic linear algebra program that solely contains loops, sBLACs, and auxiliary scalar operations such as divisions and square roots. Finally, Steps 5–7 in Figure 5.4 are designed after LGen and transform the basic linear algebra program into efficient, optionally SIMD-vectorized, C code. We consider Steps 5–7 the implementation level of the new generator.

A simple combination of Cl1ck’s and LGen’s processing steps is not sufficient to achieve our goal. To implement our generator, in collaboration with the authors of Cl1ck, we introduced a number of significant changes and additions to the original systems. More specifically, The extensions to Cl1ck were developed by D. Fabregat-Traver, the extensions to LGen by the author of this dissertation, while the LL backend by both. Next we describe these extensions in greater detail.

Extensions to Cl1ck. Steps 1–3 in Figure 5.4 are obtained by extending the original Cl1ck with the following changes:

- New mathematical DSLs. To go from a single functionality to a program, a new group of mathematical DSLs was designed to represent and manipulate as a language all concepts introduced in Section 5.1, such as input
operations, PMEs, loop invariants, and algorithms. We call these languages LA languages (from linear algebra) and the grammar in Table 5.3 describes them at once. The LA language is used to describe input programs composed of one or more equations with input and output annotations. LA is obtained from rules (1)–(7) in Table 5.3 (excluding the choice of \( \langle \text{algorithm} \rangle \) in rule (1)); an example of LA program, analogous to CLick’s original input in Figure 5.2, is shown in Figure 5.5. Language p-LA (partitioned LA; rules (1)–(6) and (8)–(10) in Table 5.3) is an extension of LA and is used to represent PMEs and loop invariants (e.g., (5.14) and (5.13) in Section 5.1). Finally, lp-LA (loop-based p-LA; rules (1)–(6) and (9)–(15) in Table 5.3) is an extension of p-LA that can describe algorithms such as the one in Algorithm 5.1. A sequence of algorithms and sBLACs is an example of an lp-LA program.

- A new flexible ISA manager. As described in Section 5.1.2, the original CLick relied on a fixed ISA provided by the BLAS interface. To enable the integration with LGEN, we redesigned Steps 1–3 in Figure 5.4 to allow for a more flexible interfacing to support LGEN’s wider set of sBLACs, as well as scalar auxiliary operations (square roots, ...).
5.2 LINEAR ALGEBRA PROGRAM SYNTHESIS OVERVIEW

**program** TRSM

Matrix L <n, n, Input, LowerTriangular>
Matrix B <m, n, Input>
Matrix X <m, n, Output>

LX == B

end

Figure 5.5: Example of LA program for a triangular linear system.

- **Nested PME generation.** To proceed with the discovery of algorithms, the PME of a target operation such as TRSM in Figure 5.5 must consist of only ISA-supported operations or recursive calls to smaller instances of the same operation (e.g., (5.14)). If any other unknown higher level equation is encountered in the formulation, a new PME generation task must be initiated to complete the algorithmic definition of the target operation. A mechanism to dynamically learn new PMEs was incorporated into Step 1 in Figure 5.4 (feedback loop).

**EXTENSIONS TO LGEN.** Steps 5–7 in Figure 5.4 are obtained by extending the original LGEN with the following changes:

- **Basic linear algebra programs in LL.** The DSL LL was extended from describing single sBLACs to entire basic linear algebra programs. The grammar is provided in Table 5.4. It includes loops, sBLACs, scalar auxiliary operations, and gather operators. Loops, sBLACs and auxiliary operations are called LL statements. An example of LL program is provided in Figure 5.6.

- **LL statement-level optimizations.** Step 5 in Figure 5.4 is extended with additional rewriting rules for the replacement of LL statements that may exhibit poor performance; for example, a group of scalar computations can be rewritten as a vectorizable sBLAC.

- **A domain-specific load/store analysis.** Differently from a subroutine-based approach, at the C-IR level we can take advantage of a global perspective over the entire computation. This enables the replacement of explicit memory loads and stores with efficient data rearrangement between vector variables. The approach is domain-specific as all pointers are associated with mathematical matrices.

**THE LL BACKEND.** The LL backend (Step 4 in Figure 5.4) is the component of our new generator that links generated algorithms in lp-LA to their basic linear algebra formulation in LL. It performs two tasks:
Table 5.3: Superset grammar in EBNF for the LA, p-LA and lp-LA languages. The non-terminal \( \langle \text{type} \rangle \) is any of Matrix, Vector, and Scalar. id and dim are any variable name. expression represents any well-defined combination of scalars, vectors and matrices with operators \(+, -, \cdot, (\cdot)^T\) (transposition) and \((\cdot)^{-1}\) (inverse).

\[
\begin{align*}
\langle \text{la-program} \rangle & ::= \text{'program id'} \\
& \quad \{ \langle \text{declaration} \rangle \} \{ \langle \text{equation} \rangle \} \{ \langle \text{algorithm} \rangle \} \text{'end'} \\
\langle \text{declaration} \rangle & ::= \langle \text{type} \rangle \text{'id'} \{', id'} \langle \text{dim} \rangle \langle \text{dim} \rangle \langle \text{iotype} \rangle \\
& \quad \{',', \langle \text{property} \rangle \} \{',', \langle \text{ow} \rangle \} \text{'}\rangle \\
\langle \text{iotype} \rangle & ::= \langle \text{Input} \rangle \langle \text{Output} \rangle \langle \text{InOut} \rangle \\
\langle \text{property} \rangle & ::= \langle \text{LowerTriangular} \rangle \langle \text{UpperTriangular} \rangle \\
& \quad \langle \text{Symmetric} \rangle \langle \text{PositiveDefinite} \rangle \\
\langle \text{ow} \rangle & ::= \langle \text{Overwrites} \rangle \text{'}id' \rangle \\
\langle \text{equation} \rangle & ::= \langle \text{expression} \rangle \langle \text{expression} \rangle \\
\langle \text{blk-of-asgn} \rangle & ::= \left( \langle \text{assignment} \rangle | \ldots \right. \\
& \quad \left. \vdots \right| \left. \ldots \right) \\
\langle \text{assignment} \rangle & ::= \langle \text{id} = \langle \text{expression} \rangle \} \{ \langle \text{func-call} \rangle \} \\
\langle \text{func-call} \rangle & ::= \text{'}id(id\{', id'} \} \} \\
\langle \text{algorithm} \rangle & ::= \langle \text{algorithm id} \} \langle \text{id} \rangle \{', id'} \} \} \\
& \quad \{ \langle \text{declaration} \rangle \} \\
& \quad \{ \langle \text{partition} \rangle \} \\
& \quad \langle \text{while( size(id) < dim(id) )} \} \\
& \quad \{ \langle \text{repartition} \rangle \} \\
& \quad \{ \langle \text{assignment} \rangle \} \\
& \quad \{ \langle \text{continue} \rangle \} \\
& \quad \text{'end'} \\
& \quad \text{'end'} \\
\langle \text{partition} \rangle & ::= \langle \text{blk-of-ids} \} \'= \text{partition(id)'} \} \\
\langle \text{repart} \rangle & ::= \langle \text{blk-of-ids} \} \'= \text{repartition('} \langle \text{blk-of-ids} \} \} \\
\langle \text{continue} \rangle & ::= \langle \text{blk-of-ids} \} \'= \text{continue('} \langle \text{blk-of-ids} \} \} \\
\langle \text{blk-of-ids} \rangle & ::= \left( \langle \text{id} \rangle \ldots \right. \\
& \quad \left. \vdots \right| \left. \ldots \right) \\
\end{align*}
\]
Table 5.4: Grammar of the extended LL language which extends the one defined in Table 3.2. \texttt{idx-expr} and \texttt{idx-cond} are respectively a numerical expression and condition on indices (including the use of \texttt{min/max} and \texttt{floor/ceil}). A non-terminal \texttt{(ll-expr)} marked with a subscript \texttt{s} denote a scalar LL expression.

\[\langle ll-program \rangle ::= \{ \langle declaration \rangle \} \{ \langle ll-statement \rangle \}\]

\[\langle declaration \rangle ::= \text{Table 5.3, Rule (2)}\]

\[\langle ll-statement \rangle ::= \langle ll-for \rangle \mid \langle ll-if \rangle \mid \langle ll-assignment \rangle\]

\[\langle ll-for \rangle ::= \text{'For (idx-expr ; idx-expr ; idx-expr) {'} \{ \langle ll-statement \rangle \} \text{'}\]

\[\langle ll-if \rangle ::= \text{'If (idx-cond) {'} \{ \langle ll-statement \rangle \} \text{'}\]

\[\langle ll-assignment \rangle ::= \langle quantity \rangle \text{'='} \langle ll-expr \rangle \text{';'}\]

\[\langle ll-expr \rangle ::= \text{'('} \langle ll-expr \rangle \text{'} \mid \langle basic-op \rangle\]

\[\langle basic-op \rangle ::= \langle ll-expr \rangle \text{'+'} \langle ll-expr \rangle \mid \langle ll-expr \rangle \text{'*'} \langle ll-expr \rangle \mid \langle ll-expr \rangle \text{'T'} \mid \langle ll-expr \rangle \text{'/'} \langle ll-expr \rangle \mid \sqrt{\langle ll-expr \rangle} \mid \ldots\]

\[\langle quantity \rangle ::= \langle gather \rangle \mid \langle matrix \rangle \mid \langle vector \rangle \mid \langle scalar \rangle\]

\[\langle gather \rangle ::= (\langle matrix \rangle \mid \langle vector \rangle)[\langle ll-expr \rangle_{\langle ll-expr \rangle_{\langle ll-expr \rangle_{\langle ll-expr \rangle_{\langle ll-expr \rangle}}}}\]

\begin{verbatim}
For (i = 0; i < 3; i = i + 1) {
    M[i, i] = -(M[0, i]_{4,4})^T M[0, i]_{4,4} + M[i, i];
}
For (j = i + 1; j < 4; j = j + 1) {
    M[i, j] = A[i, j]/M[i, i];
}
\end{verbatim}

Figure 5.6: Simple example of an LL program (declarations are omitted). Domain and range are omitted with scalar gathers (i.e., gathers with range $1 \times 1$).

- \textit{lp-LA to LL translation}. An lp-LA program from Step 3 contains a sequence of algorithms (encoding, for example, Algorithm 5.1) and sBLACs. sBLACs are directly added to the LL program, while algorithms must be translated into LL loop nests. To this end, every non-sBLAC operation in the algorithm is replaced by another algorithm to compute it. The process iterates until the loop nest only contains sBLACS and scalar operations.
program CHOL
Matrix A <n, n, Input, Symmetric, PositiveDefinite>
Matrix X <n, n, Output, UpperTriangular, Overwrites(A)>

\[ X^T X = A \]
end

Figure 5.7: LA program for the Cholesky decomposition.

- Handling temporary and zero-size matrices. Temporary matrices may degrade performance due to memory allocation and increased memory traffic. Thus we implemented traditional compiler analysis and optimization passes in Step 4 to reduce them in the final LL program. Furthermore, some of the LL statements may contain zero-sized matrices at the beginning or at the end of a loop. We introduce a loop peeling pass to reduce the number of if-statements otherwise needed.

5.3 Linear Algebra Program Synthesis Applied

In the first part of this section, we describe the generation process of the Cholesky decomposition, which we have described with an unblocked algorithm in Section 2.2.5. Finally we address how the same process applies to the more complex case of a single iteration of the Kalman filter shown in Table 5.1.

5.3.1 Vectorized Code Generation for the Cholesky Decomposition

Given a symmetric positive definite matrix \( A \in \mathbb{R}^{n \times n} \), the Cholesky decomposition (CHOL) computes an upper triangular matrix \( X \in \mathbb{R}^{n \times n} \) such that \( X^T X = A \). In our running example, we generate double precision code for \( n = 4 \), with vectorization enabled, for an Intel CPU with AVX intrinsics (\( v = 4 \)). The LA input program for CHOL is shown in Figure 5.7. As indicated in the program definition, the code will overwrite the contents of matrix \( A \) with the solution \( X \).

Nested PME Generation. Our generator starts by deriving the (unique) PME for CHOL (Step 1 of Figure 5.4). The operands \( X \) and \( A \) are partitioned in 2 by 2 parts as

\[
\begin{pmatrix}
X_{TL} & X_{TR} \\
0 & X_{BR}
\end{pmatrix}
\text{ and }
\begin{pmatrix}
A_{TL} & A_{TR} \\
A_{BL} & A_{BR}
\end{pmatrix}.
\]

The original operands are replaced by the partitioned ones in the input equation (Figure 5.7, line 4), obtaining the partitioned equations

\[ X_{TL}^T X_{TL} + X_{TL}^T X_{BR} = A_{TL} \]
\[ X_{TL}^T X_{TR} + X_{TR}^T X_{BR} = A_{TR} \]
\[ X_{TL}^T X_{BR} + X_{TR}^T X_{BR} = A_{BL} \]
\[ X_{TL}^T X_{BR} = A_{BR} \]

\[ X_{TL}^T X_{BR} = A_{BR} \]

The * in the bottom-left quadrant indicates that it is the transpose of the top-right one, and its computation is thus not required.
As for CLick, during the partitioning of the operands, properties for their parts are inferred; these properties enable the identification of subproblems via constrained structural pattern matching, and are also carried on to enable the generation of more efficient code.

After identifying the equation in the top-left quadrant as a recursive call to CHOL and marking $X_{TL}$ as known\(^3\), the generator encounters in the top-right quadrant the expression $X_{TL}^T X_{TR} = A_{TR}$, which is neither an sBLAC nor a recursive call to CHOL. Since it cannot identify the operation, a new instance of the PME generator is run to find a recursive definition for the operation (feedback loop in Step 1 of Figure 5.4). To this end, properties of the equation are collected and the input

```
program TRSM
  Matrix X_{TL} <k, k, Input, UpperTriangular>
  Matrix A_{TR} <k, r, Input>
  Matrix X_{TR} <k, r, Output, Overwrites(A_{TR})>

  X_{TL}^T X_{TR} == A_{TR}
end
```

is created to initiate the generation of PMEs.

In this case, the operation is a TRSM-like operation\(^4\), similar to that introduced in Section 5.1.2.

All PMEs of TRSM contain only calls to sBLACs and TRSM itself; the PME generation thus concludes producing the PME expression in p-LA shown in Figure 5.8.

The mechanism for nested generation of PMEs is critical not only to find a complete recursive definition of the initial program (CHOL in this case) but also because the PMEs for TRSM will be necessary for the generation of algorithms (Step 3 in Figure 5.4).

**LOOP INVARIANT IDENTIFICATION.** Next, in Step 2 of Figure 5.4, the PME is decomposed into its building blocks, i.e., sBLACs and function calls and a graph of dependencies among them is built. For CHOL’s PME, the building blocks and graph of dependencies are shown in Figure 5.9.

\(^3\) Now the generator knows which operation computes $X_{TL}$ and it becomes (symbolically) an input.

\(^4\) Note that we use TRSM as the name for the operation for readability reasons. The generator does not need to know this; it assigns the operation a random name.
program CHOL
Matrix \( X_{TL} <k, k, \text{Output}, \text{UpperTriangular}, \text{Overwrites}(A_{TL}) > \)
Matrix \( X_{TR} <k, r, \text{Output}, \text{Overwrites}(A_{TR}) > \)
Matrix \( X_{BR} <r, r, \text{Output}, \text{UpperTriangular}, \text{Overwrites}(A_{BR}) > \)
Matrix \( A_{TL} <k, k, \text{Input}, \text{Symmetric}, \text{PositiveDefinite} > \)
Matrix \( A_{TR} <k, r, \text{Input} > \)
Matrix \( A_{BR} <r, r, \text{Input}, \text{Symmetric}, \text{PositiveDefinite} > \)

\[
\begin{align*}
X_{TL} &= \text{CHOL}(A_{TL}) \\
X_{TR} &= \text{TRSM}(X_{TL}^T, A_{TR}) \\
X_{BR} &= \text{CHOL}(A_{BR} - X_{TR}^T X_{TR})
\end{align*}
\]

end

Figure 5.8: p-LA program for the PME of the Cholesky decomposition. The * in the bottom-left quadrant indicates that it is the transpose of the top-right one, and its computation is thus not required.

1. \( X_{TL} = \text{CHOL}(A_{TL}) \)
2. \( X_{TR} = \text{TRSM}(X_{TL}^T, A_{TR}) \)
3. \( T_1 = A_{BR} - X_{TR}^T X_{TR} \)
4. \( X_{BR} = \text{CHOL}(T_1) \)

Figure 5.9: Left: Decomposition of CHOL’s PME into its building blocks. Right: Graph of dependencies.

Before moving forward with the selection of subgraphs as loop invariants, two transformation passes for the sequence of building blocks are required. First, the generator has to ensure that each statement abides to the interface of each operation. In this case, CHOL overwrites its input; therefore, statement 4 in Figure 5.9 is replaced by

4. \( X_{BR} = T_1 \)
5. \( X_{BR} = \text{CHOL}(X_{BR}) \)

Second, following an analysis close to copy propagation and dead-code elimination, the generator is able to remove temporary \( T_1 \), a critical step for performance. The resulting sequence is the same as in Figure 5.9, with \( T_1 \) replaced by \( X_{BR} \).

Among the possible subgraphs, three of them lead to feasible loop invariants. The p-LA loop invariant corresponding to subgraph \( \{1, 2\} \) in Figure 5.9 is shown in Figure 5.10.

**Algorithm construction.** In Step 3 of Figure 5.4 each of the loop invariants is transformed into an algorithm using rewriting of the loop invariant
program CHOL
    Matrix $X_{TL} <k, k, \text{Output}, \text{UpperTriangular}, \text{Overwrites}(A_{TL})>$
    Matrix $X_{TR} <k, r, \text{Output}, \text{Overwrites}(A_{TR})>$
    Matrix $A_{TL} <k, k, \text{Input}, \text{Symmetric, PositiveDefinite}>$
    Matrix $A_{TR} <k, r, \text{Input}>$

    \[
    \begin{pmatrix}
    X_{TL} = CHOL(A_{TL}) & X_{TR} = TRSM(X_{TL}^T, A_{TR})
    \end{pmatrix}
    \]
end

Figure 5.10: p-LA program for one of the Cholesky decomposition’s loop invariants associated with subgraph \{1, 2\} in Figure 5.9.

into considerably long expressions; details can be found in [28]. Here, we discuss one conceptually critical detail. The repartition and continue statements to traverse the matrices, imply a rewriting of the loop invariant in terms of the repartitioned submatrices. For instance, in the case of the p-LA loop invariant in Figure 5.10, the top-right quadrant after the repartitioning results in

\[
(X_{0,1} \mid X_{0,2}) = TRSM(X_{0,0}^T, (A_{0,1} \mid A_{0,2})).
\]

Only by learning the PMEs derived in the nested PME generation, the generator can flatten the expression into

\[
X_{0,1} = TRSM(X_{0,0}^T, A_{0,1}) \\
X_{0,2} = TRSM(X_{0,0}^T, A_{0,2})).
\]

Once the full algorithm has been derived and the sequence of assignments in the loop has been fixed, the occurrences of $X$ can be safely replaced with the overwritten counterparts of $A^5$. The lp-LA algorithm corresponding to the p-LA loop invariant in Figure 5.10 is shown in Figure 5.11.

**LL backend: lp-LA to LL translation.** In Step 4 of Figure 5.4, our generator translates the lp-LA algorithm in Figure 5.11 into an LL program.

First, the algorithm is refined until it only contains sBLACS and scalar operations. Our example problem size is $4 \times 4$, which matches the architecture vector size $v$, therefore the block size $b$ is set to one. Among the four loop statements in Figure 5.11, lines 9 and 11 are sBLACs and can be directly translated into LL notation. Lines 10 and 12 instead need to be further processed. For the recursive call to CHOL, since $A_{11}$ is a scalar, the backend queries the corresponding ISA instruction associated to a scalar CHOL base case, which returns $A_{11} = \sqrt{A_{11}}$.

---

5 The dependencies due to the aliasing of $A$ and $X$ are carefully propagated and treated so that this final replacement does not lead to incorrect algorithms.
program CHOL
algorithm \([A] = \text{CHOL}(A, b)\)
\[
\begin{bmatrix}
A_{TL} & A_{TR} \\
A_{BL} & A_{BR}
\end{bmatrix} = \text{partition}(A, A_{TL}, 0, 0)
\]

while( size(A_{TL}) < size(A) )
\[
\begin{bmatrix}
A_{0,0} & A_{0,1} & A_{0,2} \\
A_{1,0} & A_{1,1} & A_{1,2} \\
A_{2,0} & A_{2,1} & A_{2,2}
\end{bmatrix} = \text{repartition}(\begin{bmatrix}
A_{TL} & A_{TR} \\
A_{BL} & A_{BR}
\end{bmatrix}, A_{1,1}, b, b)
\]

\[
A_{1,1} = A_{1,1} - A_{0,1}^T A_{0,1}
\]
\[
A_{1,1} = \text{CHOL}(A_{1,1})
\]
\[
A_{1,2} = A_{1,2} - A_{0,1}^T A_{0,2}
\]
\[
A_{1,2} = \text{TRSM}(A_{1,1}, A_{1,2})
\]
\[
\begin{bmatrix}
A_{TL} & A_{TR} \\
A_{BL} & A_{BR}
\end{bmatrix} = \text{continue}(\begin{bmatrix}
A_{0,0} & A_{0,1} & A_{0,2} \\
A_{1,0} & A_{1,1} & A_{1,2} \\
A_{2,0} & A_{2,1} & A_{2,2}
\end{bmatrix})
\]
end
end
end

Figure 5.11: lp-LA algorithm to compute CHOL. The unblocked algorithm to compute the recursive call to TRSM can be obtained by setting block size \(b = 1\).

Figure 5.11, line 12 corresponds to a TRSM of size \(1 \times n\) (i.e., \(A_{12}\) is a row vector). Therefore, the backend queries for an unblocked algorithm (feedback in Step 4 of Figure 5.4) for TRSM. At this point, all statements are either scalar operations or sBLACs, and the backend proceeds constructing the LL program.

Next, partitionings are replaced by gathers, and lp-LA while-loops are rewritten as LL loops. This requires the introduction of explicit indices. One index is introduced per loop, and the matrix parts that are affected by the index are labeled accordingly. The index stride is given by the block size \(b\) of the algorithm, and the starting position and sizes of the gather operation are expressed in terms of the index.

For instance, the repartition of matrix \(A\) in Figure 5.11 induces the presence of an index \(i\) to traverse the matrix diagonally. Gathers are created for each partition in the loop statements using a scheme similar to the following:
For \( i = 0; \ i < 4; \ i = i + 1 \) \{ \\
\begin{align*}
    A[i, i] &= -\left( A[0, i]_{i, i}^{4,4} \right)^T A[0, i]_{i, i}^{4,4} + A[i, i]; \\
    A[i, i] &= \sqrt{A[i, i];} \\
    A[i, i] &= A[i, i + 1]_{i, i + 1}^{4,4} = \\
    &\quad -\left( A[0, i]_{i, i}^{4,4} \right)^T A[0, i + 1]_{i, i + 1}^{4,4} + A[i, i + 1]_{i, i + 1}^{4,4}; \\
\end{align*}
\}
\}

\textbf{Figure 5.12:} LL program for the input computation in Figure 5.7.

\[
\begin{pmatrix}
    A_{0,0} & A_{0,1} & A_{0,2} \\
    A_{1,0} & A_{1,1} & A_{1,2} \\
    A_{2,0} & A_{2,1} & A_{2,2}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
    A[0, i]_{i, i}^{n,n} & A[0, i + b]_{i, i}^{n,n} \\
    A[i, i]_{i, i}^{n,n} & A[i, i + b]_{i, i + b}^{n,n} \\
\end{pmatrix},
\]

where in our case \( n = 4 \) and \( b = 1 \).

The \( \ell p-\text{LA} \) while loop in Figure 5.11, line 5 is replaced with the LL index-based formulation

\[
\text{For} \ (i = 0; \ i < n; \ i = i + b) \ { \ldots } \).
\]

The final LL program is reported in Figure 5.12, where domains and ranges of gathers that yield scalars are omitted and square roots and divisions appear from performing the Cholesky decomposition and the TRSM on scalar values.

Before proceeding further, the generator applies loop peeling as some of the gathers could produce zero-size blocks. Finally, loops associated with algorithms with unit block size are also unrolled to expose all statements required to compute small instances of higher level functions to further optimizations (e.g., the computation of a small TRSM in lines 6–8 of Figure 5.12). The final program is shown in Figure 5.13.

**LL statement-level optimizations.** The LL program shown in Figure 5.13 is composed of sBLACs that can be mapped to vectorized building blocks (i.e., the v-BLACs) but also several scalar computations that could result in a lower vectorization efficiency. We address this issue by introducing a set of rewriting rules in Step 5 of Figure 5.4 aimed at increasing the amount of vectorizable LL statements. This technique is similar in spirit to the one used to identify superword-level parallelism (SPL) \([71]\).

For instance consider the pair of rules \( R_0 \) and \( R_1 \) in Table 5.5. \( R_0 \) expands two TRSMs into a larger one while \( R_1 \) transforms an element-wise division of a vector by a scalar into a scalar division followed by a scalar multiplication. The
A[0, 0] = \sqrt{A[0, 0]};
A[0, 1] = A[0, 1]/A[0, 0];
A[0, 2] = A[0, 2]/A[0, 0];
A[0, 3] = A[0, 3]/A[0, 0];
A[1, 1] = \sqrt{A[1, 1]};
A[1, 2] = A[1, 2]/A[1, 1];
A[1, 3] = A[1, 3]/A[1, 1];
A[2, 2] = \sqrt{A[2, 2]};

Figure 5.13: LL program in Figure 5.12 after loop peeling and unrolling.

Table 5.5: Example of rewriting rules to expose more v-BLACs. X, B ∈ \mathbb{R}^{m \times n_1}; U ∈ \mathbb{R}^{m \times m}; X, B ∈ \mathbb{R}^{m \times (n_0 + n_1)}; x, b ∈ \mathbb{R}^k; and \lambda, \tau ∈ \mathbb{R}. Statement S_0 appears in the computation before S_1 and no operation writes to X_1, B_1, or L in between.

\begin{align}
R_0 : & \quad S_0 : X_0 = U^{-1}B_0, \quad S_1 : X_1 = U^{-1}B_1 \\
& \quad X = (X_0 | X_1), B = (B_0 | B_1), X = U^{-1}B \\
R_1 : & \quad \operatorname{op}(x) = \operatorname{op}(b)/\lambda, \quad \operatorname{op}(\cdot) = (\cdot) or (\cdot)^T \\
& \quad \tau = 1/\lambda, \quad \operatorname{op}(x) = \tau \cdot \operatorname{op}(b)
\end{align}

(5.15) (5.16)

application of rules R_0 and R_1 to lines 2–4 and 8–9 in Figure 5.13 yields two additional scalar multiplication v-BLACs as shown in Figure 5.14. Similar rules can be introduced for all basic operators creating new opportunities to improve code vectorization by identifying more v-BLACs in the code.

\begin{align}
\tau_0 = 1/A[0, 0]; & \quad \tau_1 = 1/A[1, 1]; \\
A[0, 1]_{1, 3} = \tau_0 A[0, 1]_{1, 3}; & \quad A[1, 2]_{1, 2} = \tau_1 A[1, 2]_{1, 2};
\end{align}

(a) (b)

Figure 5.14: Application of rules in Table 5.5 to (a) lines 2–4 and (b) lines 8–9 in the code in Figure 5.13, which determines additional v-BLACs (second line of both (a) and (b)).
5.3 Linear Algebra Program Synthesis Applied

```c
/* Begin Storer of sca. mul. in Fig. 19a */
Genstore(A+1, smul19a, [0, 1, 2], hor);
...
/* Begin Storer of sca. mul. in Fig. 19a */
Genstore(A+6, smul19b, [0, 1], hor);
...
/* Begin Loader for Fig. 16 l.10 */
__m256d vA02_vert = Genload(A+2, [0, 1], vert);
```

Figure 5.15: C-IR code snippet for the load of \( A[0,2]^{4,4} \) in line 10 of Figure 5.13.

**Tiling and Loop-level Optimization.** Every LL statement is now either an auxiliary scalar computation or an sBLAC. Steps 5 and 6 of Figure 5.4 proceed following LGen’s steps 1–3 in Section 3.1, Figure 3.1: all sBLACs are tiled and a \( \Sigma \)-LL formulation is produced. In our example, this yields a \( \Sigma \)-LL program analogous to the LL input in Figure 5.13 as all operands are already smaller or equal than a block \( \times \).

**Code-level Optimizations.** The last step in Figure 5.4 converts a \( \Sigma \)-LL program into C-IR. At this point our generator performs optimization across what originally were lp-LA operations (e.g., lines 6–9 in Figure 5.11). We focus on an improved scalarization of vector accesses enabled by a domain-specific load/store analysis. The goal of the analysis is to reduce explicit memory loads and stores in the final vectorized C code. We explain the approach with an example. Consider, the gather \( A[0,2]^{4,4} \) in Figure 5.13, line 10. The elements gathered from matrix \( A \) are defined by two previous computations, i.e., the two scalar multiplications in Figure 5.14. Their C-IR store/load sequence is provided in Figure 5.15, which would yield the AVX code in Figure 5.16(a). However, by analyzing their overlap we can deduce that element 1 of the first vector (smul19a) goes into element 0 of the result one, while element 0 of the second vector (smul19b) into element 1. This results in the more efficient wiring and associated AVX code shown in Figure 5.16(b).

**Algorithmic Autotuning Extension.** Finally, the optimized C-IR code is unparsed into C code and its performance is measured. At this point, if any variants are available the generator resumes from either Step 5 in Figure 5.4, if the variant depends on implementation choices (e.g., tiling), or Step 2, if it is a new algorithmic variant. In our example, our generator would repeat the process just discussed for the two remaining loop invariants identified from the PME in Figure 5.8. Finally the fastest code is provided as an output.
Figure 5.16: Resulting AVX code for the C-IR snippet in Figure 5.15 without (a) and with (b) load/store analysis. In (a) vA02_vert is obtained by explicitly storing to and loading from memory while in (b) by shuffling vector variables.

5.3.2 Vectorized Code Generation for the Kalman Filter

We now briefly discuss the more complex case of generating code for a single iteration of the Kalman filter (KF) shown in Table 5.1.

First, an LA program is created decomposing KF into computations that can be processed by our generator (e.g., sBLACS, matrix decompositions, and linear systems). In particular, (5.1)–(5.2) are sBLACs and can be directly described in LA. The second pair of equations (5.3)–(5.4) must first be decomposed into six computations. Referring to (5.3) we need to compute:

(i) Two sBLACs: \( P_0 = H P_{k\mid k-1} H^T + R \) and \( v_0 = z_k - H x_{k\mid k-1} \).

(ii) The Cholesky decomposition of \( U^T U = P_0 \).

(iii) The solution of the two triangular systems: \( U^T v_1 = v_0 \) and \( U v_2 = v_1 \).

(iv) The sBLAC \( x_{k\mid k} = x_{k\mid k-1} + P_{k\mid k-1} H^T v_2 \).

Currently this decomposition is manually performed but could be automated using approaches such as [31].

KF now takes the following form as LA program:

```
program KF
    Vector u <m, Input> % Control vector
    Vector x, y <n, InOut> % A priori (y) and a posteriori (x) state estimate
    Vector z <k, Input> % State observation
```
Matrix \( F <n, n, \text{Input}> \) % State-transition model
Matrix \( B <n, m, \text{Input}> \) % Control-input model
Matrix \( H <k, n, \text{Input}> \) % Observation model
% A priori (\( Y \)) and a posteriori (\( P \)) estimate covariance
Matrix \( P, Y <n, n, \text{InOut}, \text{Symmetric}, \text{PositiveDefinite}> \)
% Covariance of the process noise
Matrix \( Q <n, n, \text{Input}, \text{Symmetric}, \text{PositiveDefinite}> \)
% Covariance of the observation noise
Matrix \( R <k, k, \text{Input}, \text{Symmetric}, \text{PositiveDefinite}> \)

% Temporaries
Vector \( t_0 <k, \text{InOut}> \)
Vector \( t_1, t_2 <k, \text{InOut, Overwrites}(t_0)> \)

Matrix \( M_0 <k, n, \text{InOut}> \)
Matrix \( M_1 <n, k, \text{InOut}> \)
Matrix \( M_2, M_3 <k, n, \text{InOut, Overwrites}(M_0)> \)
Matrix \( S <k, k, \text{InOut, Symmetric, PositiveDefinite}> \)
Matrix \( U <k, k, \text{InOut, UpperTriangular, Overwrites}(S)> \)

% Predict
\( y == Fx + Bu \)
\( Y == FP F^T + Q \)

% Update
\( M_0 == HY \)
\( S == M_0 H^T + R \)
\( U^T U == S \)

\( v_0 == z - Hy \)
\( U^T v_1 == v_0 \)
\( Uv_2 == v_1 \)

\( U^T M_1 == M_0 \)
\( UM_2 == M_1 \)

\( M_3 == YH^T \)
% From (5.3)
\( x == y + M_3 v_2 \)
% From (5.4)
\( P == Y - M_3 M_2 \)

end
Second, the LA program is provided as input to our generator which executes the steps previously discussed for the case of Cholesky decomposition. In particular, algorithms for the Cholesky decomposition and the solution of triangular systems have to be synthesized and optimizations over a basic linear algebra description of KF are performed in Steps 5–7 in Figure 5.4.

The final output is a single vectorized C function that performs the desired computation.
EXPERIMENTAL RESULTS

In this chapter we discuss experimental results for the three stages of our work described in Chapters 3–5. In particular, we start with our approach for compiling basic linear algebra computations with LG\textsc{en} presented in Chapter 3. Then, we assess LG\textsc{en}’s extension for the support of basic linear algebra computations with structured matrices discussed in Chapter 4. Finally, we evaluate the code generator introduced in Chapter 5 for both single higher-level linear algebra computations and the Kalman filter program shown in Section 5.3.2.

6.1 BASIC LINEAR ALGEBRA COMPUTATIONS

In this section we show performance benchmarks of BLAC code generated by LG\textsc{en}.

EXPERIMENTAL CATEGORIES. We divide our experiments into the four categories listed in Table 6.1 depending on the type of functionality generated.

In the first three cases we use matrices with narrow rectangular shapes (panels) or small squares (blocks). This choice is due to their importance [43, 109]. The sizes are either \( n \times 4 \) or \( 4 \times n \), chosen to fit into L1 D-cache, or \( 4 \times 4 \). For Micro BLACs, the matrices are \( n \times n \) with \( 2 \leq n \leq 10 \).

MEASURING PROCESS. All experiments involve single-precision code. For all plots, the y-axis shows performance in flops per cycle (f/c), and the x-axis shows the value of the input’s varying dimensions as number of float elements. The flop count is derived from the BLAC while cycles are explicitly measured.

All experiments are run under warm cache conditions, executed multiple times for at least \( 10^8 \) cycles. The reported measurement is the average number of cycles per execution. This process is repeated 15 times to compute median and quartile information. Each point in the plots is the median of 15 repetitions and it is accompanied by whiskers that show the most extreme data points falling into the range \([1.5q_1, 1.5q_3]\), where \( q_1 \) and \( q_3 \) are the first and third quartiles.
Table 6.1: BLAC experimental categories. In the first three cases we use matrices with narrow rectangular shapes (panels) or small squares (blocks) with sizes either $n \times 4$ or $4 \times n$, chosen to fit into L1 D-cache, or $4 \times 4$. For Micro BLACs, the matrices are $n \times n$ with $2 \leq n \leq 10$.

<table>
<thead>
<tr>
<th>Category</th>
<th>Label</th>
<th>BLAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple BLACs</td>
<td>$s_{mv}$</td>
<td>$y = Ax$</td>
</tr>
<tr>
<td></td>
<td>$s_{mm}$</td>
<td>$C = AB$</td>
</tr>
<tr>
<td>BLAS-like</td>
<td>$s_{axpy}$</td>
<td>$y = \alpha x + y$</td>
</tr>
<tr>
<td></td>
<td>$s_{gemv}$</td>
<td>$y = \alpha Ax + \beta y$</td>
</tr>
<tr>
<td></td>
<td>$s_{gemm}$</td>
<td>$C = \alpha AB + \beta C$</td>
</tr>
<tr>
<td>Non-BLAS</td>
<td>$s_{gesummv}$</td>
<td>$y = \alpha Ax + \beta Bx$</td>
</tr>
<tr>
<td></td>
<td>$s_{blinf}$</td>
<td>$\alpha = x^T Ay$</td>
</tr>
<tr>
<td></td>
<td>$s_{gemam}$</td>
<td>$C = \alpha(A_0 + A_1)^T B + \beta C$</td>
</tr>
<tr>
<td>Micro BLACs</td>
<td>$s_{mv}$</td>
<td>Three BLACs from previous cases using very small matrices and vectors.</td>
</tr>
<tr>
<td></td>
<td>$s_{mm}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$s_{blinf}$</td>
<td></td>
</tr>
</tbody>
</table>

6.1.1 Targeting High-end Systems

We run our tests on an Intel Xeon X5680 (Westmere EP microarchitecture), 3.3 GHz, SSE 4.2, 32 kB L1 D-cache, under RHEL Server 6 with kernel v.2.6.32. Intel’s SpeedStep and Turbo Boost technologies were disabled during the tests. The theoretical peak performance of the platform is 8 f/c. However, our plots are scaled to 6 f/c for better readability.

Competitors. We considered only single precision code and compared against (a) Intel MKL v.11, (b) Intel IPP v.7.1, (c) Eigen v.3.1.3, (d) BTO v1.3, and (e) handwritten code. The latter is straightforward, loop-based, scalar C code and comes in two versions: with hardcoded problem sizes (fixed size) and with problem sizes passed as parameters (general size).

Hardware and software configuration. MKL and IPP are provided as binary code. Code obtained from LGen, Eigen, BTO, as well as all the handwritten kernels were compiled using icc v.13.1 with flags -O3 -xHost -fargument-noalias -fno-alias -ipo -ipo.

LGen uses a random search with a sample size of 10. BTO’s kernels were generated disabling multithreading, and enabling loop tiling. In Eigen we used...
fixed-size Map interfaces to existing arrays, no-alias assignments, and enabled SSE code generation.

In MKL, we implemented \texttt{sgesummv} with two calls to \texttt{cblas\_sgemv}, \texttt{sblinf} as a combination of \texttt{cblas\_sgemv} and \texttt{cblas\_sdot}, and \texttt{sgemam} as a call to \texttt{MKL\_Somatadd}\(^1\) followed by \texttt{cblas\_sgemm}.

**Case 1: Simple BLACS.** Figure 6.1 shows performance results for the BLACs \texttt{smv} and \texttt{smm}. For \texttt{smv} with vertical \(A\) (Figure 6.1(a)), LGEN performs between 1.8\(\times\) and 3\(\times\) better than Eigen. With horizontal \(A\) (Figure 6.1(b)) and for larger \(n\) LGEN performs within 10\% of IPP and Eigen. For \texttt{smm} we consider four scenarios. In the panel-block case (Figure 6.1(c)) LGEN performs about 2.5\(\times\) faster than MKL. In the block-panel computation (Figure 6.1(d)) the improvements reduce to 10\% for larger sizes. For the panel-panel products the speed-up is again a factor of about 3\(\times\) over the competition in Figure 6.1(e) and about 2\(\times\) for the rank-4 update in Figure 6.1(f). Unfortunately, we could not compare against BTO due to exceptions raised by the generated code. Downwards spikes, such as in Figure 6.1(e), are related to suboptimal tiling decisions resulting either from the random selection during search or from current multilevel tiling limitations (see Section 3.2.2).

**Case 2: BLAS-like Computations.** The results are shown in Figure 6.2. For \texttt{saxpy} (Figure 6.2(a)) MKL and the icc-compiled fixed size code attain the same performance which is about 15\% higher than LGEN. \texttt{sgemv} (Figures 6.2(b)–(c)) and \texttt{sgemm} (Figures 6.2(d)–(g)) have a performance behavior very close to the one previously observed for \texttt{smv} and \texttt{smm}.

**Case 3: Non-BLAS Computations.** The results are shown in Figure 6.3. Eigen’s ability to generate fused loops results in comparable performance between BLACs in Case 2 and 3 (e.g., \texttt{smv}-based expressions in Figures. 6.3(d), 6.3(b), and 6.2(c)). On the other hand, we notice that slight changes in computational patterns (e.g., from \texttt{smv} in Figure 6.1(b) to \texttt{sgesummv} in Figure 6.3(b)) can diminish the capability of icc to apply loop-level optimizations. The combination of BTO’s autotuning capabilities and icc’s autovectorization achieves similar performance to LGEN (about 4 f/c) for the case of \texttt{sgesummv} with horizontal panels (Figure 6.3(b)). For \texttt{sgemam} (Figures 6.3(e)–(h)) all competing curves except MKL perform below 1 f/c.

**Case 4: Micro BLACS.** Finally, we report on small size code in Figure 6.4. In this case LGEN produces fully unrolled code with vectorized left-over computations. In case of \texttt{smv} and \texttt{smm}, LGEN exhibits improvements between 1.25\(\times\) and

---

\(^1\) \texttt{MKL\_Somatadd} is a non-BLAS function provided by Intel MKL.
Figure 6.1: Simple BLACs. (a)–(b): $y = Ax$; (c)–(f): $C = AB$. 

(a) $A$ is $n \times 4$. 
(b) $A$ is $4 \times n$. 
(c) $A$ is $n \times 4$, $B$ is $4 \times 4$. 
(d) $A$ is $4 \times 4$, $B$ is $4 \times n$. 
(e) $A$ is $4 \times n$, $B$ is $n \times 4$. 
(f) $A$ is $n \times 4$, $B$ is $4 \times n$. 

102 EXPERIMENTAL RESULTS
6.1 Basic Linear Algebra Computations

Figure 6.2: BLACs that closely match BLAS. (a): $y = \alpha x + y$; (b)–(c): $y = \alpha A x + \beta y$; (d)–(g): $C = \alpha A B + \beta C$. 

(a) $x$ of length $n$.  
(b) $A$ is $n \times 4$.  
(c) $A$ is $4 \times n$.  
(d) $A$ is $n \times 4$, $B$ is $4 \times 4$.  
(e) $A$ is $4 \times 4$, $B$ is $4 \times n$.  
(f) $A$ is $4 \times n$, $B$ is $n \times 4$.  
(g) $A$ is $n \times 4$, $B$ is $4 \times n$. 

- LGen
- Handwritten fixed size
- Handwritten gen size
- MKL 11.0
- Eigen 3.1.3
- BTO 1.3
- IPP 7.1
Figure 6.3: BLACs that need more than one BLAS call. (a)–(b): \( y = \alpha Ax + \beta Bx \); (c)–(d): \( \alpha = x^T Ay \); (e)–(h): \( C = \alpha(A_0 + A_1)^T B + \beta C \).
3.5× compared to icc fixed size, which is the best competing code. For \textit{sblinf} we achieve a speedup as high as up to 6× with respect to Eigen.

**Remarks.** From the plots we can observe that certain shapes are favored by the existing libraries and generators we compared with. For instance, looking at Figure 6.3 we can quickly identify that the most competitive plots (b), (d), (f), and (h) involve horizontal panels. In contrast, \textsc{LGen} produces across most functions and sizes a performance in the 3–6 f/c range. Also worth noting is that the compiler fails to optimize straightforward loop code, even when specialized to the problem size.

### 6.1.2 Targeting Embedded Systems

In this section we present performance experiments conducted to evaluate the code generated by \textsc{LGen} on four embedded processors: Intel Atom, ARM Cortex-A8, ARM-Cortex-A9, and ARM1176. Table 6.2 summarizes relevant information about the computing platforms.
Table 6.2: Properties of the platforms used for the experiments. AI stands for arithmetic instruction, LS for load/store, f/c are flops/cycle.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Atom D2550</th>
<th>Cortex-A8</th>
<th>Cortex-A9</th>
<th>ARM1176</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector ISA</td>
<td>SSSE3</td>
<td>NEON</td>
<td>NEON</td>
<td>-</td>
</tr>
<tr>
<td>D-L1 [kB]</td>
<td>24</td>
<td>32</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>I-L1 [kB]</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>Peak [f/c]</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Execution</td>
<td>in-order</td>
<td>in-order</td>
<td>out-of-order</td>
<td>in-order</td>
</tr>
<tr>
<td>Issues 2 AI</td>
<td>yes</td>
<td>yes (FMA)</td>
<td>yes (FMA)</td>
<td>no</td>
</tr>
<tr>
<td>Issues LS+AI</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Board</td>
<td>Mini-PC</td>
<td>BeagleBone Black</td>
<td>Kayla DevKit</td>
<td>Raspberry Pi</td>
</tr>
<tr>
<td>OS kernel</td>
<td>Linux 3.8</td>
<td>Linux 3.8</td>
<td>Linux 3.1</td>
<td>Linux 3.6</td>
</tr>
</tbody>
</table>

The two Cortex-A processors present two critical microarchitectural differences: (a) Scalar floating point operations are more efficient on Cortex-A9 and (b) Cortex-A8 can issue a NEON load/store instruction together with a NEON arithmetic operation, while this is not possible on Cortex-A9.

The peak performance values in this table were computed without considering the impact of loads and stores and assuming an ideal ratio of additions and multiplications. In the following we describe our tests, competitors, and provide details about the configuration of our hardware and software environment.

**CHosen blacs.** In this experimental context we focus on only two computations from Table 6.1: `sblinf` and `sgemam`. In particular, `sblinf` is a memory-intensive computation while `sgemam` a compute-intensive one, where by memory-intensive we mean a ratio of operations to data movement of $O(1)$. A larger set of experiments is discussed in [69]. Unless stated otherwise, all matrices and vectors are 16-byte aligned.

**Competitors.** Our selected competitors are: (a) Intel MKL 11.1 (Intel Atom only), (b) Intel IPP 8.0 (Intel Atom only), (c) Eigen 3.2 (all processors), (d) ATLAS 3.10.1 (all processors), and (e) compilers taking as input handwritten straightforward, scalar code as described before (all processors). In the last case, we considered again both code with fixed problem sizes that are known at compile time and code with unknown problem sizes that are passed as arguments.

**Hardware and software configuration.** On Intel Atom, cycles are measured using the `rdtsc` instruction. On the ARM Cortex-A8 and ARM1176 we
used the cycle counter of the performance monitor unit (PMU). On the ARM Cortex-A9 we used the Linux perf infrastructure.

We disabled hyper-threading on Intel Atom and CPU throttling on the three ARM processors. LGEN was configured to use a random search with sample size of 10.

For both Intel MKL and ATLAS, we implemented \textit{sblinf} as a combination of \texttt{cblas_sgemv} and \texttt{cblas_sdot}. \textit{sgemam} was implemented in MKL with a call to \texttt{MKL_Somatadd} followed by \texttt{cblas_sgemm} and in ATLAS with a call to \texttt{cblas_saxpy} followed by \texttt{cblas_sgemm}. For Eigen we used Map interfaces over existing arrays, no-alias assignments, and we enabled vector code generation by defining \texttt{EIGEN\_VECTORIZE}.

ATLAS was built natively using gcc 4.7 on all platforms. On the Intel Atom we used the provided architectural defaults, while for the other three processors we executed a full search to find the best values for the ATLAS parameters. For all four processors, \texttt{gemm} was retuned after the installation to improve the performance of ATLAS for small matrix computations, as it is described in the errata section of the ATLAS website\textsuperscript{2}.

On the Intel Atom tests were compiled with icc 14 (flags -O3 -xHost -fargument-noalias -fno-alias -no-ip -no-ip -no-prec-div); on the ARM processors with clang 3.4 (flags -O3 -mcpu=<cpuname>) and gcc 4.7 (flags -O3 -ffast-math -fsingle-precision-constant -fstrict-aliasing -mcpu=<cpuname> -march=armv7-a -mtune=<cpuname> -mfpu=neon -mfloat-abi=hard).

**Labelling Conventions.** For plots we use the following labelling convention: \texttt{LGen} for LGEN without generic C-IR loads and stores (see Section 3.3.2), \texttt{LGen -GLS} for LGEN using generic C-IR loads and stores, and \texttt{LGen -GLS -AD} for LGEN using both generic C-IR loads and stores and alignment detection. Alignment detection only applies on Atom, since the ARM NEON intrinsics do not provide aligned loads and stores. Also, the NEON \texttt{v-BLACs} were implemented directly with generic loads and stores; thus a comparison to the previous LGEN is omitted.

**Intel Atom.** In Figure 6.5(a) we show the results for the computation of \textit{sblinf}. \texttt{LGen -GLS -AD} performs better than all competitors, achieving speedups of up to $2.8 \times$ with respect to MKL. The presence of several downward spikes is due to the amount of unaligned instructions available in the code. The size of the panel matrix $\mathbf{A}$ strongly influences performance, bringing it down to $1 f/c$ whenever $n \mod 4 \in \{1, 3\}$ (which yields only 25\% aligned accesses).

In Figure 6.5(b) we show the performance results for \textit{sgemam}. The performance of \texttt{LGen -GLS} is around 30\% higher than the one of LGEN. Alignment detection adds another 30\% of improvement over the performance of LGEN for matrix

\textsuperscript{2} \url{http://math-atlas.sourceforge.net/errata.html}
Figure 6.5: (a) $\alpha = x^T Ay$ and (b) $C = \alpha(A_0 + A_1)^TB + \beta C$ on the Intel Atom.

Figure 6.6: (a) $\alpha = x^T Ay$ and (b) $C = \alpha(A_0 + A_1)^TB + \beta C$ on the ARM Cortex-A8.

sizes that favor this optimization (i.e. divisible by 4). Eigen, the best competitor, performs better than LG\text{en} for larger matrices, but never better than LG\text{en-GLS} and LG\text{en-GLS-AD}.

**ARM Cortex-A8.** In all experiments conducted on Cortex-A8 (Figure 6.6) the competitors achieve lower performance than LG\text{en} (in most cases less than 0.2 f/c). The main reason is the mixing of scalar and vector instructions, which on Cortex-A8 leads to poor performance. This does not apply to LG\text{en}, as it generates completely vectorized code, even when handling leftovers. The performance of LG\text{en} is mostly in the range 1-1.3 f/c, being up to $9 \times$ faster than the best competitor.

**ARM Cortex-A9.** Figure 6.7 shows the experimental results for Cortex-A9. For s\text{blinf} (Figure 6.7(a)) Eigen is the best competitor, achieving 10–40% lower performance than LG\text{en}. For s\text{gemam} (Figure 6.7(b)) LG\text{en} is more than $2 \times$ faster than the optimizing compilers and 25% faster than Eigen, with a performance
of between 0.8 and 1 f/c. For wider matrices ATLAS approaches LGen’s performance to within 10%.

**ARM1176.** The ARM1176 processor does not present vector units and optimizations such as tiling, loop unrolling, loop fusion, and loop exchange have a significant impact on the quality of the generated code. In all the experiments in Figure 6.8 LGen is up to $4 \times$ faster than ATLAS, which is in all cases the best competitor. Drops in performance can be noticed for large values of $n$ due to reaching the L1 D-cache size (16 kB). Another general remark is that for all tested BLACs, LGen’s generated code compiled with gcc is more efficient than the one compiled with clang. Finally, for large values of $n$ the performance results of LGen are less stable because of the random search with a sample size that is relatively small compared to the large space of tiling options.
Table 6.3: sBLAC experimental categories. For $S$ we specify whether lower (l) or upper (u) part is used.

<table>
<thead>
<tr>
<th>Category</th>
<th>Label</th>
<th>sBLAC</th>
<th>Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAS</td>
<td>$dsyrk$</td>
<td>$S_u = AA^T + S_u$</td>
<td>$A \in \mathbb{R}^{n \times 4}$</td>
</tr>
<tr>
<td></td>
<td>$dtrsv$</td>
<td>$x = L \backslash x$</td>
<td>$L \in \mathbb{R}^{n \times n}$</td>
</tr>
<tr>
<td>BLAS-like</td>
<td>$dlusmm$</td>
<td>$A = LU + S_l$</td>
<td>$L, U \in \mathbb{R}^{n \times n}$</td>
</tr>
<tr>
<td></td>
<td>$dsylmm$</td>
<td>$A = S_u L + A$</td>
<td>$S_u, L \in \mathbb{R}^{n \times n}$</td>
</tr>
<tr>
<td>Non-BLAS</td>
<td>composite</td>
<td>$A = (L_0 + L_1)S_l + xx^T$</td>
<td>$L_0, S_l \in \mathbb{R}^{n \times n}$</td>
</tr>
</tbody>
</table>

6.2 Basic Linear Algebra Computations with Structured Matrices

In this section we provide an experimental evaluation of our approach in the context of sBLACs. We divide our experiments into the three categories, shown in Table 6.3, according to the compatibility of the sBLACs with the BLAS. Specifically we chose two sBLACs that match BLAS, two sBLACs that are available in BLAS but without support for structures (BLAS-like), and an sBLAC that can only be implemented using more than one BLAS or BLAS-like function (Non-BLAS). Matrices are implemented using double precision arrays 32-bytes aligned. All of them are fully stored in row-major order and in the case of triangular and symmetric matrices only half of the matrices are used.

Experimental setup. We executed our tests on an Intel Core i7-2600 CPU (Sandy Bridge microarchitecture) 3.3 GHz, AVX, 32 kB L1 D-cache, 256 kB L2 cache, under Ubuntu 14.04 with Linux kernel v3.13. We disabled Intel Turbo Boost to minimize measurement instabilities. We compare with: (a) the Intel MKL library v11.2, (b) naïve code compiled with Intel icc v15, and (c) code generated by LGEn without support for structures. Further, starting with v11.2 it adds support for small-scale, double precision matrix multiplication (dgemm). BLAS tests are implemented using the matching BLAS functions. BLAS-like tests are implemented using $dtrmm$ (in the case of $dlusmm$) and $dsymm$ (in the case of $dsylmm$). The composite test is implemented using $MKL_{\text{Somatadd}}, dsyr$, and $dsymm$. We do not rearrange matrices when testing MKL (e.g., zeroing a half triangular matrix when used in place of a general one).

Naïve code is scalar, unoptimized, handwritten, straightforward code with hardcoded sizes of the matrices. The goal is to compare with compiler optimizations.
All tests were compiled using icc with flags -O3 -xHost -fargument-noalias -fno-alias -no-ip -no-ip. The MKL tests use flags obtained from the Intel MKL Link Line Advisor\(^3\).

**Measuring Approach.** All plots show performance in flops per cycles (f/c) on the y-axis and the size parameter \(n\) in doubles on the x-axis. Assuming balanced additions and multiplications, the peak performance of the CPU is 8 f/c. The parameter \(n\) is always increased up to the L2 cache boundaries. All tests were run with warm cache. Every point on the graphs is the median of 30 repetitions and quartile informations are reported with whiskers. In most cases, however, these are too small to be visible.

**Remarks on Plots.** We compute performance as the ratio of flop count taking structures into account (f underneath each plot) to measured time to solution. This way the plots can provide an estimate of the CPU utilization as well as carrying information about time speedup (as \(\frac{f/c_1}{f/c_2} = \frac{c_2}{c_1}\)). Every plot shows the L1 boundary determined by working set size (size of all inputs and outputs of an sBLAC). All plots have the same legend show on top of each figure.

**Category BLAS.** For *dsyrk* (Figures 6.9(a)–(b)) LGEN is up to 2.5× faster than MKL when data fit in L1 and around 1.6× when data fit in L2. Comparing with icc-compiled code LGEN is in general 1.6× faster. In all cases, icc performs unrolling and vectorization of the innermost loop of length four. However, icc does not modify the loop nest to increase reuse by blocking. In the case of *dtrsv* (Figs. 6.9(c)–(d)) all competitors perform equally. For larger sizes casting the computation in terms of matrix-vector multiplication becomes more crucial, an optimization that also icc applies by unrolling and vectorizing the innermost of the two loops in the handwritten code. In this case we could not generate code using the old LGEN approach as it lacks the structure support required by the triangular solve.

**Category BLAS-like.** The first test in the BLAS-like category is *dlusmm* (Figures 6.10(a)–(b)). Here LGEN is up to 3.5× faster than icc and up to 2× faster than MKL for data in L1 (1.4× for data in L2). In this case exploiting the structure of both L and U avoids about one third of redundant computations. icc on the other hand fails to perform and take advantage of proper tiling for locality. In *dsylnm* (Figures 6.10(c)–(d)) LGEN is up to 7× faster than icc-compiled code and, for sizes up to the L1 boundary, about 1.4× faster than MKL. Further investigations revealed that code generated with LGEN produces high pressure on the shuffle unit of the CPU. This could be due to an excessive amount of transpositions in

\(^3\)https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor
In this section we evaluate the generator presented in Chapter 5 for five different computations: Four linear algebra building blocks (the Cholesky decomposition, the solution of triangular, continuous-time Sylvester and Lyapunov equations, the innermost loops and could be handled by introducing block permutations in between (non-fused) gathers.

**Category non-BLAS.** Figure 6.11 shows results for composite. Although more complicated, this sBLAC contains a multiplication term structurally similar to the one in `dsylmm`, thus the similarity between the two performance profiles.

### 6.3 Higher-Level Linear Algebra Computations

In this section we evaluate the generator presented in Chapter 5 for five different computations: Four linear algebra building blocks (the Cholesky decomposition, the solution of triangular, continuous-time Sylvester and Lyapunov equations,
Figure 6.10: Category BLAS-like: (a)–(b) \( \text{dlusmm} \left( \frac{1}{3} (2n^3 + n) + n^2 \text{ flops} \right) \) and (c)–(d) \( \text{dsylmm} \left( n^3 + n^2 \text{ flops} \right) \). In (b) and (d) all sizes are multiple of the vector length (\( \nu = 4 \)).

Figure 6.11: Category Non-BLAS: (a)–(b) \( \text{composite} \left( f = n^3 + \frac{5}{2} (n^2 + n) \right) \). In (b) all sizes are multiple of the vector length (\( \nu = 4 \)).
Table 6.4: Selected benchmarks. All matrices $\in \mathbb{R}^{n \times n}$. $\text{tu}(\cdot)$ indicates the operand is considered upper triangular. The hat $\hat{\cdot}$ indicates that the computation is performed inplace.

<table>
<thead>
<tr>
<th>Name</th>
<th>Label</th>
<th>Computation</th>
<th>Flops (f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cholesky dec.</td>
<td>dpotrf</td>
<td>$\text{tu}(A)^T \text{tu}(A) = \hat{A} \approx n^3/3$</td>
<td></td>
</tr>
<tr>
<td>Sylvester eq.</td>
<td>dtrsyl</td>
<td>$\text{LC} + \text{CU} = \hat{C} \approx 2n^3$</td>
<td></td>
</tr>
<tr>
<td>Lyapunov eq.</td>
<td>dtrlya</td>
<td>$\text{LS} + \text{SL}^T = \hat{S} \approx n^3$</td>
<td></td>
</tr>
<tr>
<td>Triangular inv.</td>
<td>dtrtri</td>
<td>$L = \hat{L}^{-1} \approx n^3/3$</td>
<td></td>
</tr>
<tr>
<td>Kalman filter</td>
<td>kf</td>
<td>See Table 5.1</td>
<td>$\approx 11n^3$</td>
</tr>
</tbody>
</table>

and the inverse of a triangular matrix) and the Kalman filter. Their definitions and flop count are listed in Table 6.4.

**Experimental Setup.** All our tests are single-threaded and executed on an Intel Core i7-2600 CPU (Sandy Bridge) with 3.3 GHz, AVX, 32 kB L1 D-cache, 256 kB L2 cache, under Ubuntu 14.04 with Linux kernel v3.13. Turbo Boost is disabled. In the case of $\text{dpotrf}$, $\text{dtrsyl}$, $\text{dtrlya}$, and $\text{dtrtri}$ we compare with: (a) the Intel MKL library v11.3.2, (b) ReLAPACK [87], and straightforward code (c) compiled with Intel icc v16, and (d) clang v4 with the polyhedral Polly optimizer [47]. For $\text{dtrsyl}$ we also compare with RECSY [62], a library specifically for designed for these solvers. For $\text{kf}$ we only compare with a library-based implementation using MKL.

The straightforward code is scalar, handwritten, loop-based code with hardcoded sizes of the matrices. It is included to show optimizations performed by the compiler. For icc we use the flags -O3 -xHost -fargument-noalias -fno-alias -no-ipo -no-ip. Tests with MKL are linked to the sequential version of the library using flags from the Intel MKL Link Line Advisor\(^4\). Finally, tests with clang/Polly were compiled with flags -O3 -mllvm -polly -mllvm -polly-vectorizer=stripmine. All code considered is in double precision.

**Plot Navigation.** The plots present performance in flops per cycles ($f/c$) on the $y$-axis and the size $n$ on the $x$-axis, restricted to ensure that the working set fits into L2 cache. The peak performance of the CPU is 8 $f/c$. All tests were run with warm cache. Every point on the graphs is the median of 30 repetitions and quartile information is reported with whiskers (sometimes too small to be visible). The performance in each case is computed with the flop count in Table 6.4. For our generated code, we show performance for each algorithmic

variant using colored, dashed lines without markers, and with a bold black line the fastest of all (the exception is $kf$ where we only used the fastest subroutines where needed).

**Experimental results.** Figure 6.12 shows the performance results. For $dpotrf$ (Figure 6.12(a)) our generated code is on average $2\times$ and $1.8\times$ faster than MKL and ReLAPACK, respectively. Compared to icc and clang/Polly we obtained a larger speedup of $4.2\times$ and $5.6\times$ showing the limitations of compilers. For $dtrsyl$ (Figure 6.12(b)) our system reaches up to $2$ f/c and is typically $2.8\times$, $2.6\times$, $12\times$, $1.5\times$, and $2\times$ faster than MKL, ReLAPACK, RECSY, icc, and clang/Polly, respectively. The computation of $dtrlya$ (Figure 6.12(c)) attains around $1.7$ f/c for the larger sizes, thus being $5\times$ faster than the libraries and $2\times$ than the compilers. Note that missing a specialized interface for this function MKL performs more than $2\times$ slower than icc. Figure 6.12(d) shows $dtrtri$ where we achieve up to $3.5$ f/c, with an average speedup of about $2.5\times$, $2.3\times$, $4.2\times$, and $4.6\times$ with respect to MKL, RELAPACK, icc and clang/Polly.

Finally, for $kf$ (Figure 6.12(e)) our synthesized code is on average $1.4\times$ faster than MKL. Note that typical problem sizes for $kf$ (i.e., the size of the system state) often lie on the left half, where we observe even larger speedups. Figure 6.12(f) shows for example the performance of $kf$ when fixing the state size to $n = 28$ and varying the size of the observation between $k = 4$ and $k = 28$.

**Bottleneck analysis.** We analyzed our generated code with ERM [17], a tool that performs a generalized roofline analysis to determine hardware bottlenecks. ERM creates the computation DAG using the LLVM Interpreter and a number of microarchitectural parameters capturing throughout and latency information of instructions and the memory hierarchy. We ran ERM with the parameters describing our Sandy Bridge target platform and for the first four linear algebra routines.

Table 6.5 summarizes for each routine and three sizes the hardware bottleneck found with ERM. In general we notice that the generated code is limited by two main factors. For small sizes it is the cost of divisions/square roots, which on Sandy Bridge can only be issued every 44 cycles. The fraction of these is asymptotically small (approximately $1/n^2$ for $dpotrf$, and $1/n$ for $dtrsyl$, $dtrlya$, and $dtrtri$) but matters for small sizes as they are also sequentially dependent.

For larger sizes, the number of loads and stores between registers and L1 grows mostly due to spills. Our generator tends to fuse several innermost loops of several neighbouring sBLACs, which can result in an increased register pressure due to several intermediate computations. This insight may pave the way to further optimize the code, for instance introducing scheduling strategies supported by analytical models [76].

Table 6.5 shows on the third column the ratio of issued shuffles and blends to the total number of issued instructions, excluding loads and stores. This number
Experimental Results

Our synthesized code
ReLAPACK
Intel MKL 11.3.2
RECSY '09
Intel icc 16
clang 4 + Polly 3.9

Figure 6.12: Performance plots for the computations in Table 6.4. Colored, dashed lines without markers in (a)–(d) indicate algorithmic variants.
Table 6.5: Summary of bottleneck analysis with ERM. The shuffle/blend issue rate is the ratio of shuffle/blend issues to the total issued instructions (excluding loads and stores). The performance limit due to shuffles and blends is estimated by ERM and given in f/c.

<table>
<thead>
<tr>
<th>Computation</th>
<th>Sizes</th>
<th>Bottleneck</th>
<th>Issue rate</th>
<th>Perf limit (f/c)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Shuffles &amp; blends</td>
<td>Shuffles</td>
</tr>
<tr>
<td>(dpotrf)</td>
<td>4 divs</td>
<td>L₁ stores</td>
<td>50%</td>
<td>6.5</td>
</tr>
<tr>
<td></td>
<td>76 divs</td>
<td>L₁ stores</td>
<td>15%</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>124 divs</td>
<td>L₁ stores</td>
<td>10%</td>
<td>8</td>
</tr>
<tr>
<td>(dtrsy1)</td>
<td>4 divs</td>
<td>L₁ stores</td>
<td>35%</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>76 divs</td>
<td>L₁ stores</td>
<td>35%</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>124 divs</td>
<td>L₁ loads</td>
<td>35%</td>
<td>8</td>
</tr>
<tr>
<td>(dtrly1)</td>
<td>4 divs</td>
<td>L₁ stores</td>
<td>40%</td>
<td>6.7</td>
</tr>
<tr>
<td></td>
<td>76 divs</td>
<td>L₁ stores</td>
<td>40%</td>
<td>6.7</td>
</tr>
<tr>
<td></td>
<td>124 divs</td>
<td>L₁ loads</td>
<td>37%</td>
<td>6.8</td>
</tr>
<tr>
<td>(dtrtri)</td>
<td>4 divs</td>
<td>L₁ loads</td>
<td>62%</td>
<td>3.2</td>
</tr>
<tr>
<td></td>
<td>76 L₁ loads</td>
<td>32%</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>124 L₁ loads</td>
<td>32%</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

can be considered as an estimate of how much data rearrangement overhead is introduced by our vectorization strategy. As estimated by ERM’s bottleneck analysis the peak performance would almost never be affected by the introduced shuffles and blends (forth and fifth column).
CONCLUSIONS

We restate our goal from the introduction:

"Our goal is to develop a domain-specific framework for the synthesis of fast code for linear algebra computations of small, fixed size using as an input only their mathematical description and knowledge of the instruction set architecture of the target processor."

Currently, there is no consolidated solution that provides high-performance code for arbitrary small linear algebra computations. With this dissertation, we aimed to make a first step in this direction focusing on four intermediate objectives.

First, we introduced LG\textsuperscript{en}, a compiler for basic linear algebra computations. LG\textsuperscript{en} uses mathematical DSLs that enable optimizations at a high level of abstraction, in a way that closely resembles Spiral for linear transforms. In this work, we used the DSLs for vectorization and for loop optimizations making our approach easily portable to different vector ISAs.

Second, we addressed the problem of generating efficient code for small, basic linear computations where structured matrices appear, such as triangular or symmetric. Structures enable the elimination of redundant accesses and computations and the introduction of new functions such as the solution of triangular systems. We extended LG\textsuperscript{en} with a new methodology to represent and manipulate such computations based on the polyhedral representation of the structures they contain. Our approach is extensible to include new structures just and we gave an example by extending to band and composite matrices.

Third, we extended our framework to support higher-level computations, including the Cholesky decomposition and triangular solvers. Algorithms for these computations are synthesized using an extension of the FLAME-based Click compiler. They are represented as a sequence of basic linear algebra computations, which are then further optimized and compiled into code.

Fourth, the extension to higher-level computations made possible the handling of entire linear algebra programs with loops, thus enabling code generation for simple applications like the Kalman filter that was selected as an example.
We evaluated LGEN targeting both Intel and ARM ISAs showing that code generated with our framework performs in many cases better than well-established libraries, prior code generators, and general-purpose compilers applied to straightforward implementations.

7.1 CURRENT LIMITATIONS

We discuss a list of current limitations of LGEN and explain how they may be overcome in future work.

**Fixed size code.** Currently, LGEN generates code that is specialized to the input sizes of the operands. It is important to stress that many relevant applications in signal processing, control, and machine learning fulfill this constraint. However, for many other applications a library for general input sizes is still desirable. To support CLICK-generated algorithms we already added limited support for general size, so a further extension should be feasible. One possible solution to achieving this could be the recursion step closure technique developed in [114].

**Data types.** At present, LGEN only supports real floating point data; extension to complex numbers will mainly impact the vectorization strategy.

**Contiguous data.** Second, our approach always assumes matrices represented as full m x n arrays, with redundant regions not to be accessed (e.g., the lower half of an upper triangular matrix). For the structures discussed here, this is standard. However, for matrices such as diagonal or tridiagonal matrices, a special format (i.e., storing only the nonzero elements) could be advantageous. In the generator we did implement a separation between the physical and the mathematical layout of matrices but did not use it yet for this case. As a requirement, the structure of the new layout should be describable using affine constraints.

**Search methods and modeling.** Our search strategies are very limited at the moment: exhaustive and random search. Thus better code may be within LGEN’s scope but cannot be found. It should be straightforward to extend LGEN with well-known search or learning algorithms used in various autotuning approaches [23, 75, 100, 104, 115]. Along with autotuning, a fully model-based optimization strategy following [43, 68, 76, 121] should also be considered as part of the framework.

**Aligned accesses.** Although we proved that our alignment detection methodology is precise for our generated code, we could potentially achieve further
improvement by (a) exposing more aligned memory accesses, e.g., introducing leftovers on both sides producing an effect similar to loop peeling in the resulting code, and (b) investigating techniques similar to [67] that replace unaligned accesses with aligned ones combined with shuffles.

7.2 Future Directions

Concluding, we briefly address a few lines of research that could be investigated to extend the code synthesis approach presented in this dissertation.

Structured v-BLAC Superoptimization. Currently all v-BLACs, LGen’s computational building blocks, are handwritten for every new target ISA and structures are not taken into account (e.g., as appears in the $v \times v$ blocks on the diagonal after blocking a triangular matrix). For small $v$ the penalty is likely negligible, but for larger $v$ and relatively small matrices, some performance could be gained by special v-sBLACs tailored to specific structure. A domain-specific, superoptimization approach, similar in spirit to the one presented in [80] for permutations, could be investigated in such a context.

Multicore Parallelism. When a linear algebra computation is composed of many small subcomputations our system could take advantage of modern multicore systems. The idea is to identify task parallelism among independent subcomputations and allocate them to different cores of a processor.

Batched Small Linear Algebra Computations. While in the previous paragraph we were interested in splitting a linear algebra computation into many independent subtasks, here we consider a complementary problem. Our framework could be extended to handle batched computations, i.e., a group of independent computations that could be processed by one single function to better take advantage of data and task parallelism.

Integration into Production Systems. Another interesting direction would be investigating what is required to integrate our approach into well-established mathematical environments, such as MATLAB, and computer algebra systems, such as Mathematica. This direction could help expand to a much larger set of applications and provide ground for potential cross-domain performance optimizations which we discuss next.

Cross-Domain Performance Optimizations. Cross-domain performance optimization opportunities should be investigated at the boundaries of different mathematical domains such as linear algebra and the domain of linear trans-
forms. For this purpose, an intermediate mathematical representation such as OL could be used to express mutually beneficial transformations.
MATRICES-MATRIX MULTIPLICATION IN OL

In this appendix, we provide a description of the approach used to generate straightforward C code for matrix-matrix multiplication using the OL DSL. More details about OL and its applications can be found in [21, 22, 33, 110].

A.1 OL: THE OPERATOR LANGUAGE

The Operator Language (OL) [21, 33] is an extension of the Signal Processing Language (SPL) originally used in SPIRAL to describe linear transforms such as the discrete Fourier transform (DFT) [120]. The two main components of the language are operators and higher-order operators.

Operators. The core components of OL are operators of arity $(r, s)$ with $r, s \geq 1$, i.e., functions that take $r$ real (complex) input vectors and produce $s$ real (complex) output vectors. Example of linear operators of arity $(1, 1)$ are the $n \times n$ identity:

$$I_n : \mathbb{R}^n \rightarrow \mathbb{R}^n; \ x \mapsto x,$$

and the transpose of an $m \times n$ matrix:

$$L_{mn}^T : \mathbb{R}^{mn} \rightarrow \mathbb{R}^{mn}; \ \text{vec}(A) \mapsto \text{vec}(A^T),$$

where the function vec produces a vector of size $mn$ from a matrix of size $m \times n$ linearized in row-major order. The following are examples of bilinear operators of arity $(2, 1)$ in OL.

The pointwise product:

$$P_n : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^n; \ (x, y) \mapsto (x_iy_i).$$

The scalar product:

$$R_n : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}; \ (x, y) \mapsto \sum_i x_iy_i.$$
The Kronecker product:
\[ K_{m \times n} : \mathbb{R}^m \times \mathbb{R}^n \rightarrow \mathbb{R}^{mn}; \ (x, y) \mapsto (x_i y). \]

The matrix-matrix multiplication:
\[ \text{MMM}_{m,k,n} : \mathbb{R}^{mk} \times \mathbb{R}^{kn} \rightarrow \mathbb{R}^{mn}; \ (\text{vec}(A), \text{vec}(B)) \mapsto \text{vec}(AB). \]

Operators in OL can be expressed using formulas, i.e., operators combined together using higher-order operators.

**Higher-order operators: the cross product.** The cross product takes two operators of arity \((r_0, s_0)\) and \((r_1, s_1)\) and applies the first operator to the first \(r_0\) input vectors and the second operator to the remaining \(r_1\) producing in total \(s_0 + s_1\) output vectors. For instance, the operator
\[ L_{mk}^k \times I_{kn} \]
has arity \((2, 2)\) and works by transposing the first input (application of \(L_{mk}^k\) to a linearized \(m \times k\) matrix) and maintaining the second one unchanged (application of \(I_{kn}\) to a \(k \times n\) matrix).

**Higher-order operators: the tensor product.** The following tensor product between two bilinear operators
\[ R_k \otimes \text{MMM}_{m,1,n} \]
computes the sum of \(k\) outer products between \(k\) different chunks of the input (linearized) matrices. The tensor product is used in OL to represent loops at a high level of abstraction without the introduction of indices.

**Higher-order operators: composition.** The composition of two operators applies the second operator to the output of the first one. Conventionally the order of the operators is from right to left. For instance, the composition
\[ (R_k \otimes \text{MMM}_{m,1,n}) \circ (L_{mk}^k \times I_{kn}) \] (A.1)
applies the operator \(R_k \otimes \text{MMM}_{m,1,n}\) to the output of \(L_{mk}^k \times I_{kn}\).

In the remainder of this section, we briefly describe the main steps in the code generation process of matrix-matrix multiplication using the OL framework.
A.2 CODE GENERATION

In OL, the algorithm generation step is performed according to divide-and-conquer rewriting rules called breakdown rules.

**Matrix-matrix multiplication breakdown rules.** The operator (A.1) is one possible way to formulate matrix-matrix multiplication in terms of outer products and can be used in a breakdown rule:

\[ \text{MMM}_{m,k,n} \rightarrow (R_k \otimes \text{MMM}_{m,1,n}) \circ (L_k^{mk} \times I_{kn}). \]  

(A.2)

To provide a complete example of derivation, consider the following two breakdown rules in addition to (A.2):

\[ \text{MMM}_{m,k,n} \rightarrow K_{m \times 1} \otimes \text{MMM}_{1,k,n}, \]  

(A.3)

\[ \text{MMM}_{1,1,n} \rightarrow K_{1 \times n}. \]  

(A.4)

Rule (A.3) decomposes the computation of \( \text{MMM}_{m,k,n} \) into \( m \) (row) vector-matrix multiplications, while Rule (A.4) captures that a multiplication of a scalar by a row vector is equivalent to a Kronecker product. Breakdown rules such as (A.4) are considered base cases as their right-hand side can be directly mapped to code. Other examples of operators in OL that can be mapped directly to code are: \( I_n, L_n^{mn}, P_n, R_n, \) and \( K_{m \times n} \).

Using the three rules above we can derive the following operator for matrix-matrix multiplication:

\[ \text{MMM}_{m,k,n} \rightarrow (R_k \otimes \text{MMM}_{m,1,n}) \circ (L_k^{mk} \times I_{kn}) \]  

\[ \rightarrow (R_k \otimes (K_{m \times 1} \otimes \text{MMM}_{1,1,n})) \circ (L_k^{mk} \times I_{kn}) \]  

\[ \rightarrow (R_k \otimes (K_{m \times 1} \otimes K_{1 \times n})) \circ (L_k^{mk} \times I_{kn}). \]  

(A.5)

Depending on how breakdown rules are plugged into each other, the OL generator can derive different algorithmic versions of the same functionality. Search is used to explore the space of alternative derivations and select the one that yields code with the highest performance.

The direct translation to code of (A.5) produces the naïve code shown in Figure A.1, where an extra for loop is instantiated to collect the temporary (linearized) transposition of matrix \( A \). This operation, however, could be merged with the computation performed in Figure A.1, line 9. To enable loop level optimizations such as loop merging, OL is extended by another DSL called \( \Sigma \)-OL that we present next.

**Loop-level transformation in \( \Sigma \)-OL.** The second mathematical DSL used by the OL framework is \( \Sigma \)-OL, and extension of OL designed for repre-
\[ L_{mk}^k \cdot \text{vec}(A) \]
\[
\text{for}(t=0; t<m*k; ++t) \{
    t[(t\%k)\cdot m + t/k] = A[t];
\}
\]

\[
/\ast \text{vec}(C) = (R_k \otimes (K_{m\times 1} \otimes K_{1\times n}))(t, \text{vec}(B)) /\ast
\]
\[
\text{for}(p=0; p<k; ++p) \{
    \text{for}(i=0; i<m; ++i) \{
        \text{for}(j=0; j<n; ++j) \{
            C[i\cdot n+j] += t[p\cdot m+i] \cdot B[p\cdot n+j];
        \}
    \}
\}
\]

Figure A.1: Direct code generation of the operator (A.5) derived from the functionality \( \text{MMM}_{m,k,n} \). The extra for loop collects the temporary, linearized transposition of matrix \( A \) into vector \( t \). Vector \( t \) is finally used to compute the outer product. The output \( C \) is assumed to be zero-initialized.

senting mathematically and explicitly loops and data accesses. \( \Sigma \)-OL extends \( \Sigma \)-SPL [36].

\( \Sigma \)-OL extends OL with three new concepts: index mapping functions, parametrized operators, and iterative sums. Index mapping functions are used to represent functions on indices such as those used to access arrays in code. For example the two functions used in Figure A.1, line 3 to access elements from the arrays \( t \) and \( A \) are modelled as the transposition and identity index mapping functions:

\[
\begin{align*}
\ell_{mk}^{mk} & : \mathbb{I}_{mk} \rightarrow \mathbb{I}_{mk}; \ i \mapsto \lfloor i/k \rfloor + m(i \mod k), \\
i_{mk}^{mk} & : \mathbb{I}_{mk} \rightarrow \mathbb{I}_{mk}; \ i \mapsto i,
\end{align*}
\]

where \( \mathbb{I}_{mk} = \{0, \ldots, mk-1\} \subset \mathbb{N} \). The respective matrix operators are created by means of the parametrized operator \( \text{perm} \):

\[
L_{mk}^{mk} = \text{perm}(\ell_{mk}^{mk}),
I_{mk} = \text{perm}(i_{mk}^{mk}).
\]

Other two parametrized operators are the gather and scatter operators. They are linear operators used to represent general accesses to vectors. For example, we can use a gather to extract two elements from a vector of length four starting from the second element at stride two:

\[
\begin{pmatrix}
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix} \begin{pmatrix}
x \end{pmatrix} = G(h_{12}^{2+4})x,
\]

where \( x(1:2:) \) is MATLAB-like notation for a stride access from element \( b = 1 \) until the end of the vector by a factor \( s = 2 \) and \( G(h_{12}^{2+4}) \) is a gather operator that
Table A.1: Rules to convert OL to $\Sigma$-OL. $A : \mathbb{R}^m \times \mathbb{R}^n \rightarrow \mathbb{R}^k$ is any arity-(2,1) operator.

\[
R_k \otimes A \rightarrow \sum_{\sigma \leq i < k} A \circ (G(h_{im,1}^{m \rightarrow k}) \times G(h_{in,1}^{n \rightarrow k}))
\]  
\[K_{p \times q} \otimes A \rightarrow \sum_{\sigma \leq i < p} \sum_{\sigma \leq j < q} S(h_{(i+q \cdot j)k,1}^{p \rightarrow q}) \circ A \circ (G(h_{im,1}^{m \rightarrow p}) \times G(h_{jn,1}^{n \rightarrow q}))
\]

depends on the stride index mapping function $h$. The stride function is defined in general as:

\[
h_{b,s}^{d \rightarrow r} : I_d \rightarrow I_r; \quad i \mapsto b + is.
\]  

A scatter operator $S(f)$ is the transpose of a gather $G(f)$ and can be used to create a vector with non-zero elements at a stride:

\[
\begin{bmatrix} 0 & \chi_0 & \chi_1 \end{bmatrix}^T = G(h_{i,2}^{2 \rightarrow 4})^T \begin{bmatrix} \chi_0 \\ \chi_1 \end{bmatrix} = S(h_{i,2}^{2 \rightarrow 4}) \begin{bmatrix} \chi_0 \\ \chi_1 \end{bmatrix}.
\]

Similar concepts were introduced in the case of $\Sigma$-LL in Section 3.2. Here, however, they work on vectors rather than on matrices.

Finally, loops are represented in $\Sigma$-OL as iterative sums over indices. For example, the tensor product in (A.2) applies the outer product $\text{MMM}_{m,1,n}$ to $k$ chunks of the input vectors and accumulates the result. The following is an analogous formulation of the operator using gathers and an iterative sum:

\[
R_k \otimes \text{MMM}_{m,1,n} = \sum_{\sigma \leq i < k} \text{MMM}_{m,1,n} \circ (G(h_{im,1}^{m \rightarrow k}) \times G(h_{in,1}^{n \rightarrow k}))
\]

where at every iteration two chunks of $m$ and $n$ consecutive elements are obtained from the input vectors.

Using these three new concepts, i.e. iterative sums, parametrized operators, and index mapping functions, it is possible to perform in $\Sigma$-OL code level optimizations such as loop merging at a higher level of abstraction as we show next.

**Loop Merging Example.** The generation of the redundant memory copy in the first for loop of Figure A.1 is prevented by: (i) converting the OL operator to a $\Sigma$-OL one using conversion rules such as those in Table A.1 obtained generalizing equivalences such as (A.7), and (ii) applying loop merging and index simplification rules such as those in Tables A.2 and A.3 whenever possible.

One possible sequence of applications is the following:
Table A.2: Loop merging rules. A, A_i, B, C, and D are operators, while f, f_i, and p are index mapping functions. Operators A and B can be composed with C and D respectively, as well as A_i with B.

\[
(A \times B) \circ (C \times D) \rightarrow (A \circ C) \times (B \circ D)
\]  \hspace{1cm} (A.12)

\[
G(f_1) \circ G(f_2) \rightarrow G(f_2 \circ f_1)
\]  \hspace{1cm} (A.13)

\[
S(f_1) \circ S(f_2) \rightarrow S(f_1 \circ f_2)
\]  \hspace{1cm} (A.14)

\[
G(f) \circ \text{perm}(p) \rightarrow G(p \circ f)
\]  \hspace{1cm} (A.15)

Table A.3: Simplification rules of index mapping functions.

\[
L_{mk}^{mk} \rightarrow \text{perm}(t_{k}^{mk \rightarrow mk})
\]  \hspace{1cm} (A.16)

\[
I_{mk}^{mk} \rightarrow \text{perm}(1^{mk \rightarrow mk})
\]  \hspace{1cm} (A.17)

\[
\ell_{mk \rightarrow mk}^{mk} \circ h_{ki,l}^{mk} \rightarrow h_{i,m}^{mk}
\]  \hspace{1cm} (A.18)

\[
h_{s',s}^{n \rightarrow r} \circ h_{b_{s},s}^{d \rightarrow n} \rightarrow h_{b_{s}',s'}^{d \rightarrow r}
\]  \hspace{1cm} (A.19)

\[
h_{l_{n},n}^{n \rightarrow n} \rightarrow t^{n \rightarrow n}
\]  \hspace{1cm} (A.20)

\[
f \circ 1, 1 \circ f \rightarrow f
\]  \hspace{1cm} (A.21)
As a final step, the OL framework maps the final expression above to the naïve triple loop code in Figure A.2, where the direct access to the array $A$ is merged with the outer product computation within the innermost loop.

Next we briefly address how tiling and vectorization are applied using breakdown rules in OL.

**Tiling in OL.** The breakdown rule (A.2) recursively build on matrix-matrix multiplication with $k = 1$. This can be generalized by replacing $k = 1$ in the rule with the tiling choice $k = k_b$:

$$
\text{MMM}_{m,k,n} \to (R_{k/k_b} \otimes \text{MMM}_{m,k_b,n}) \circ ((L_{k/k_b}^{m/k_{b}} \otimes I_{k_b}) \times I_{kn}),
$$

(A.22)

The operator $L_{k/k_b}^{m/k_{b}} \otimes I_{k_b}$ in (A.22) has the effect to de-interleave the rows of the $k/k_b$ different tiles of size $m \times k_b$. The parameter $k_b$ is a degree of freedom in the algorithm construction process, and its choice has the effect of enlarging the search space explored by the OL generator. More details on expert-level tiling and packing of matrix-matrix multiplication in OL can be found in [110].

**Vectorization in OL.** Vectorization is applied by tagging the input functionality with the length $\nu$ of the vector registers:

$$
\text{vec}(\text{MMM}_{m,k,n}).
$$

(A.22)
The tag is then used to control the rewriting process using tagged breakdown rules such as

$$\underbrace{\text{MMM}_{m,k,n/\nu} \otimes K_{1 \times \nu}}_{\text{vec}(\nu)} \rightarrow (\text{MMM}_{m,k,n/\nu} \otimes P_{\nu}) \circ (\text{dup}_{k}^{\nu} \times I_{m\nu}), \quad (A.23)$$

where $\text{dup}_{k}^{\nu} : \mathbb{R}^{k} \rightarrow \mathbb{R}^{\nu k}$ consecutively replicates $\nu$ times every element of a vector. The goal is the derivation of an OL formula based on a few platform-specific base cases that map well to the given paradigm. For example, $\text{MMM}_{m,k,n/\nu} \otimes P_{\nu}$ which appears in the rule above is an example of paradigm-specific base case for vectorization, where the $\nu$-way pointwise multiplication can be implemented solely with arithmetic and memory access intrinsics. More details on the vectorization of the base cases are provided in [34, 35].
FLAME DERIVATION STEPS

In this appendix we provide a full description of the FLAME algorithm derivation process discussed in Section 5.1.1. A thorough introduction to the FLAME methodology and its applications can be found in [107].

We start from the lower triangular system:

\[ LX = B, \quad L \in \mathcal{L}_n, \quad X, B \in \mathbb{R}^{n \times m}, \quad (B.1) \]

and systematically derive a blocked algorithm for it, such as the one shown in Algorithm B.1. The reason of a temporary matrix \( T \) and why the algorithm is labelled with “variant 2” will be clear shortly. FLAME uses a constructive approach that at the same time ensures the correctness of the derivated algorithm.

To better clarify the approach, we will apply it step by step to (B.1) showing how the five parts of a FLAME algorithm (i.e., (a)–(e) in Algorithm B.1) are derived.

**Step f1: Determining the Partitioned Matrix Expression.** The first step requires to fix a partitioning of the operands and use it in (B.1). For example, since \( L \) is lower triangular, partitioning it into quadrants\(^1\) maintains the properties on the two diagonal partitions:

\[ L \rightarrow \begin{pmatrix} L_{TL} & 0 \\ L_{BL} & L_{BR} \end{pmatrix}, \quad L_{TL} \in \mathcal{L}_k, \quad L_{BR} \in \mathcal{L}_{n-k}, \quad 0 \leq k \leq n, \quad (B.2) \]

where the size of the partitions will vary during the execution of the loop-based algorithm.

This choice also imposes a consistent partitioning of \( X \) and \( B \), such as the following:

\[ B \rightarrow \begin{pmatrix} B_T \\ B_B \end{pmatrix}, \quad X \rightarrow \begin{pmatrix} X_T \\ X_B \end{pmatrix}, \quad (B.3) \]

where \( \text{rows}(B_T) = \text{rows}(X_T) = \text{cols}(L_{TL}) \).

\(^1\) The subscript notation indicate (L)eft, (R)ight, (T)op, and (B)ottom partitions.
Algorithm B.1 Solving a lower triangular system (Forward substitution algorithm, blocked variant 2). 

\( L \mathbf{X} = \mathbf{B}, \quad L \in \mathcal{L}_n, \ X, \mathbf{B} \in \mathbb{R}^{n \times m}. \) Initially, the temporary matrix \( T = \mathbf{B} \).

(a) Partition \( L \to \begin{pmatrix} L_{TL} & 0 \\ L_{BL} & L_{BR} \end{pmatrix}, \ T \to \begin{pmatrix} T_T \\ T_B \end{pmatrix}, \ X \to \begin{pmatrix} X_T \\ X_B \end{pmatrix} \)

where \( L_{TL} \) is \( 0 \times 0 \) and \( X_T \) is \( 0 \times m \)

(b) while \( \text{size}(X_T) < \text{size}(X) \) do

(c) Repartition

\[
\begin{pmatrix} L_{TL} & 0 \\ L_{BL} & L_{BR} \end{pmatrix} \to \begin{pmatrix} L_{0,0} & 0 & 0 \\ L_{1,0} & L_{1,1} & 0 \\ L_{2,0} & L_{2,1} & L_{2,2} \end{pmatrix}, \quad \begin{pmatrix} T_T \\ T_B \end{pmatrix} \to \begin{pmatrix} T_0 \\ T_1 \\ T_2 \end{pmatrix}, \quad \begin{pmatrix} X_T \\ X_B \end{pmatrix} \to \begin{pmatrix} X_0 \\ X_1 \\ X_2 \end{pmatrix}
\]

where \( L_{1,1} \) is \( b \times b \) and \( X_1 \) is \( b \times m \)

(d) \( X_1 := T_1 L_{1,1}^{-1} \)

\( T_2 := T_2 - L_{2,1} X_1 \)

(e) Continue with

\[
\begin{pmatrix} L_{TL} & 0 \\ L_{BL} & L_{BR} \end{pmatrix} \to \begin{pmatrix} L_{0,0} & 0 & 0 \\ L_{1,0} & L_{1,1} & 0 \\ L_{2,0} & L_{2,1} & L_{2,2} \end{pmatrix}, \quad \begin{pmatrix} T_T \\ T_B \end{pmatrix} \to \begin{pmatrix} T_0 \\ T_1 \\ T_2 \end{pmatrix}, \quad \begin{pmatrix} X_T \\ X_B \end{pmatrix} \to \begin{pmatrix} X_0 \\ X_1 \\ X_2 \end{pmatrix}
\]

endwhile

Plugging (B.2) and (B.3) in (B.1) determines a partitioned matrix expression (PME) of (B.1):

\[
\begin{pmatrix} L_{TL} & 0 \\ L_{BL} & L_{BR} \end{pmatrix} \begin{pmatrix} X_T \\ X_B \end{pmatrix} = \begin{pmatrix} B_T \\ B_B \end{pmatrix}
\]

\[
\rightarrow \begin{pmatrix} L_{TL} X_T \\ L_{BL} X_T + L_{BR} X_B \end{pmatrix} = \begin{pmatrix} B_T \\ B_B \end{pmatrix}
\]

\[
\rightarrow \begin{pmatrix} L_{TL} X_T = B_T \\ L_{BR} X_B = B_B - L_{BL} X_T \end{pmatrix}
\]  
(B.4)
Table B.1: Loop invariants of (B.1) obtained from the PME in (B.4). The matrix-like notation on the right alternatively describes the loop invariants providing a link between the predicates and their source in the PME.

\[
\begin{align*}
L_{TL}X_T &= B_T \equiv \left( \frac{L_{TL}X_T = B_T}{-} \right) \\
L_{TL}X_T &= B_T \land T_B = B_B - L_{BL}X_T \equiv \left( \frac{L_{TL}X_T = B_T}{T_B = B_B - L_{BL}X_T} \right)
\end{align*}
\]  

which recursively expresses the initial computation in terms of its partitioned operands (computing both \(X_T\) and \(X_B\) requires to solve a smaller triangular system).

The partial computations in a PME may show dependencies between each other. For instance, the following order must be respected for a correct computation of (B.4):

1. \(L_{TL}X_T = B_T\)  
2. \(T_B = B_B - L_{BL}X_T\)  
3. \(L_{BR}X_B = T_B\)

where \(T = B\) is introduced as a temporary matrix, as the algorithm is not allowed to overwrite the input in this example, and it is partitioned vertically similarly to \(B\).

**STEP F2: DETERMINING LOOP INVARIANTS.** From a PME, it is possible to derive loop invariants, i.e., predicates that hold before and after the algorithm’s loop as well as before and after each of its iterations.

For example, analyzing the PME in (B.4), we can determine the two loop invariants in Table B.1. The presence of output partitions in a loop invariant indicate that their computation has been already performed.

A loop invariant is deemed feasible when:

- It exists an initial partitioning of the operands after which the invariant still hold. For instance, both invariants in Table B.1 hold with the initial partitions defined in Algorithm B.1, part (a), which sets \(k = 0\) in (B.2).
- It exists a loop guard \(G\) such that:

\[
P_{inv} \land G \implies P_{post},
\]

where \(P_{inv}\) is the loop invariant to be vetted and \(P_{post}\) a predicate that must hold when the algorithm terminates. For instance, choosing \(P_{post} := LX = \)
B, the loop guard \( G := \text{size}(X_T) < \text{size}(X) \) in Algorithm B.1, part (b), is one possible choice that satisfies condition (B.10).

Note that the predicate
\[
L_{TL}X_T = B_T \land T_B = B_B - L_{BL}X_T \land L_{BR}X_B = T_B
\]
cannot be considered a valid loop invariant, as no initial partitioning exists that would make the predicate true.

**STEP F3.1: FIXING THE INITIALIZATION AND THE LOOP GUARD.** All feasible loop invariants are marked with the initial partitioning and loop guards identified in the previous step. When a loop invariant is chosen to produce an algorithm, they are added to the algorithm in construction. In our example, as shown in the previous paragraph, Algorithm B.1, parts (a) and (b), can be associated with both invariants in Table B.1.

**STEP F3.2: PROGRESSING THROUGH THE OPERANDS.** Algorithm B.1, parts (c) and (e), are prescribed by the choice of the initialization and the loop guard. As the size of \( X_T \) has to progress until reaching the size of \( X \), it follows that \( X_T \) must be repartitioned at each iteration. In an unblocked algorithm, such as Algorithm 2.9, the size varies by a single row (or column, or element, depending on the condition), while in a blocked one, by more than one.

For instance, in Algorithm B.1 parts (c) and (e), we chose \( b > 1 \) many rows of \( X_T \). This imposes to progress by the same amount of rows in \( B_T \) and by a \( b \times b \) block in \( L \), as we want to maintain the property of having triangular partitions on the diagonal.

**STEP F3.3: DETERMINING THE UPDATE.** The last step of the FLAME methodology prescribes how the computation should be updated between loop iterations. At this point choosing one or the other loop invariant makes the difference, as they would yield different update statements. We choose the second one which yields Algorithm B.1 (for this reason labelled “variant 2”).

Specifically, Algorithm B.1, part (d), is computed by comparing the state of the computation after repartition (i.e., Algorithm B.1, part (c)) with the state of the computation after moving the thick lines in Algorithm B.1, part (e).

The state of the computation at these two points is determined substituting the partitions in the loop invariant with the decompositions defined in Algo-
rithm B.1, parts (c) and (e). In particular, we know that after repartition the following fact hold:

\[
L_T L X_T = B_T \land T_B = B_B - L_B L X_T
\]

\[
\equiv L_{0,0} X_0 = B_0 \land \left( \frac{T_1}{T_2} \right) = \left( \frac{B_1}{B_2} \right) - \left( \frac{L_{1,0}}{L_{2,0}} \right) X_0
\]

\[
\equiv L_{0,0} X_0 = B_0 \land \left( \frac{T_1}{T_2} \right) = \left( \frac{B_1 - L_{1,0} X_0}{B_2 - L_{2,0} X_0} \right), \tag{B.11}
\]

as well as that the following predicate must be true after the computation of a loop iteration:

\[
L_T L X_T = B_T \land T_B = B_B - L_B L X_T
\]

\[
\equiv \left( \begin{array}{l}
L_{0,0} \\
L_{1,0}
\end{array} \right) \left( \begin{array}{l}
0 \\
L_{1,1}
\end{array} \right) \left( \begin{array}{l}
X_0 \\
X_1
\end{array} \right) = \left( \frac{B_0}{B_1} \right)
\]

\[
\land T_2 = B_2 - \left( \begin{array}{l}
L_{2,0} \\
L_{2,1}
\end{array} \right) \left( \begin{array}{l}
X_0 \\
X_1
\end{array} \right)
\]

\[
\equiv \left( \begin{array}{l}
L_{0,0} X_0 \\
L_{1,1} X_1
\end{array} \right) = \left( \frac{B_0}{B_1 - L_{1,0} X_0} \right) \tag{B.12}
\]

\[
\land T_2 = B_2 - L_{2,0} X_0 - L_{2,1} X_1. \tag{B.13}
\]

The difference between (B.11) and (B.12)–(B.13) is finally the update required to get from one state to the other. More specifically, since we know from (B.11) that \( T_1 = B_1 - L_{1,0} X_0 \) and \( T_2 = B_2 - L_{2,0} X_0 \), we can make (B.12) true by solving \( L_{1,1} X_1 = T_1 \), and (B.13) by computing \( T_2 = T_2 - L_{2,1} X_1 \). This two computations are added to Algorithm B.1, part (d), thus terminating the derivation of the algorithm.
BIBLIOGRAPHY


[34] F. Franchetti and M. Püschel. Short vector code generation for the discrete Fourier transform. In International Parallel and Distributed Processing Symposium (IPDPS), 2003. pages 130


CURRICULUM VITÆ

DANIELE GIUSEPPE SPAMPINATO

Doctoral student in Computer Science with research and teaching assistant duties
ETH Zurich, Switzerland 2011 – 2017

President of the association of scientific staff (Verein des Mittelbaus der Informatik - VMI)
Department of Computer Science, ETH Zurich, Switzerland 2014 – 2015

Research and teaching assistant
CSElab, ETH Zurich, Switzerland 2010 – 2011

Laurea Magistrale in Computer Engineering (equivalent M.Sc.)
Politecnico di Milano, Italy 2007 – 2009

Master i Teknologi / Sivilingeniør (equivalent M.Sc.)
Norwegian University of Science and Technology, Norway 2007 – 2009

Laurea in Computer Engineering (equivalent B.Sc.)
Politecnico di Milano, Italy 2003 – 2006

AWARDS AND HONORS

Top Industrial Managers for Europe (T.I.M.E.) Label 2013

IBM 2009 EMEA Best Student Recognition Award 2009