The Science of Programming Matrix Computations

The Science of Programming Matrix Computations

Robert A. van de Geijn
The University of Texas at Austin

Enrique S. Quintana-Ortí
Universidad Jaume I

Copyright © 2007 by Robert A. van de Geijn and Enrique S. Quintana-Ortí.

10 9 8 7 6 5 4 3 2 1

All rights reserved. No part of this book may be reproduced, stored, or transmitted in any manner without the written permission of the publisher. For information, contact either of the authors.

No warranties, express or implied, are made by the publisher, authors, and their employers that the programs contained in this volume are free of error. They should not be relied on as the sole basis to solve a problem whose incorrect solution could result in injury to person or property. If the programs are employed in such a manner, it is at the user's own risk and the publisher, authors, and their employers disclaim all liability for such misuse.

Trademarked names may be used in this book without the inclusion of a trademark symbol. These names are used in an editorial context only; no infringement of trademark is intended.

Library of Congress Cataloging-in-Publication Data

van de Geijn, Robert A.
The Science of Programming Matrix Computations / Robert A. van de Geijn
and Enrique S. Quintana-Ortí.
p. cm.
ISBN ?????????????? (pbk.)
1. Algebra, Linear. 2. Programming. I. Title.
QA ????? 2007
???? ????????

First Edition, December 2007

Contents

List of Contributors **v**

Preface **vii**

1 Motivation **1**
1.1 A Motivating Example: the LU Factorization . . . 1
1.2 Notation . . . 2
1.3 Algorithmic Variants . . . 5
1.4 Presenting Algorithms in Code . . . 5
1.5 High Performance and Blocked Algorithms . . . 6
1.6 Numerical Stability . . . 8

2 Derivation of Linear Algebra Algorithms **9**
2.1 A Farewell to Indices . . . 9
2.2 Predicates as Assertions about the State . . . 13
2.3 Verifying Loops . . . 13
2.4 Goal-Oriented Derivation of Algorithms . . . 16
2.5 Cost Analysis . . . 20
2.6 Summary . . . 23
2.7 Other Vector-Vector Operations . . . 24
2.8 Further Exercises . . . 24

3 Matrix-Vector Operations **27**
3.1 Notation . . . 28

3.2 Linear Transformations and Matrices . . . 29
3.3 Algorithms for the Matrix-Vector Product . . . 33
3.4 Rank-1 Update . . . 40
3.5 Solving Triangular Linear Systems of Equations . . . 43
3.6 Blocked Algorithms . . . 51
3.7 Summary . . . 58
3.8 Other Matrix-Vector Operations . . . 58
3.9 Further Exercises . . . 58

4 The FLAME Application Programming Interfaces 61
4.1 Example: GEMV Revisited . . . 61
4.2 The FLAME@LAB Interface for M-script . . . 62
4.3 The FLAME/C Interface for the C Programming Language . . . 70
4.4 Summary . . . 86
4.5 Further Exercises . . . 86

5 High Performance Algorithms 87
5.1 Architectural Considerations . . . 87
5.2 Matrix-Matrix Product: Background . . . 89
5.3 Algorithms for GEMM . . . 92
5.4 High-Performance Implementation of GEPP, GEMP, and GEPM . . . 100
5.5 Modularity and Performance via GEMM: Implementing SYMM . . . 105
5.6 Summary . . . 108
5.7 Other Matrix-Matrix Operations . . . 110
5.8 Further Exercises . . . 111

6 The LU and Cholesky Factorizations 113
6.1 Gaussian Elimination . . . 113
6.2 The LU Factorization . . . 114
6.3 The Basics of Partial Pivoting . . . 122
6.4 Partial Pivoting and High Performance . . . 125
6.5 The Cholesky Factorization . . . 132
6.6 Summary . . . 135
6.7 Further Exercises . . . 136

A The Use of Letters 137

B Summary of FLAME/C Routines **139**
B.1 Parameters 139
B.2 Initializing and Finalizing FLAME/C 141
B.3 Manipulating Linear Algebra Objects 141
B.4 Printing the Contents of an Object 143
B.5 A Subset of Supported Operations 143

List of Contributors

A large number of people have contributed, and continue to contribute, to the FLAME project. For a complete list, please `http://www.cs.utexas.edu/users/flame/`

Below we list the people who have contributed directly to the knowledge and understanding that is summarized in this text.

Paolo Bientinesi

Ernie Chan

Kazushige Goto

John A. Gunnels

Tze Meng Low

Margaret E. Myers

Gregorio Quintana-Ortí

Field G. Van Zee

Preface

The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand.

– E.W. Dijkstra

This book shows how to put the above words of wisdom to practice when programming algorithms for dense linear algebra operations.

Programming as a Science

One definition of *science* is *knowledge that has been reduced to a system.* In this book we show how for a broad class of matrix operations the derivation and implementation of algorithms can be made systematic.

Notation

Traditionally, algorithms in this area have been expressed by explicitly exposing the indexing of elements in matrices and vectors. It is not clear whether this has its roots in how matrix operations were originally coded in languages like Fortran77 or whether it was because the algorithms could be more concisely stated, something that may have been important in the days when the typesetting of mathematics was time-consuming and the printing of mathematical books expensive.

The notation adopted in this book attempts to capture the pictures of matrices and vectors that often accompany the explanation of an algorithm. Such a picture typically does not expose indexing. Rather, it captures regions (submatrices and subvectors) that have been, or are to be, updated in a consistent manner. Similarly, our notation identifies regions in matrices and vectors, hiding indexing details. While algorithms so expressed require more space on a page, we believe the notation improves the understanding of the algorithm as well as the opportunity for comparing and contrasting different algorithms that compute the same operation.

Application Programming Interfaces

The new notation creates a disconnect between how the algorithm is expressed and how it is then traditionally represented in code. We solve this problem by introducing Application Programming Interfaces (APIs) that allow the code to closely mirror the algorithms. In this book we refer to APIs for Matlab's M-script language, also used by Octaveand LabView MathScript, as well as for the C programming language. Since such APIs can easily be defined for almost any programming language, the approach is largely language independent.

Goal-Oriented Programming

The new notation and the APIs for representing the algorithms in code set the stage for growing proof of correctness and program hand-in-hand, as advocated by Dijkstra. For reasons that will become clear, high-performance algorithms for computing matrix operations must inherently involve a loop. The key to developing a loop is the ability to express the state (contents) of the variables, being updated by the loop, before and after each iteration of the loop. It is the new notation that allows one to concisely express this state, which is called the loop-invariant in computer science. Equally importantly, the new notation allows one to systematically identify all reasonable states that can be maintained by a loop that computes the desired matrix operation. As a result, the derivation of loops for computing matrix operations becomes systematic, allowing hand-in-hand development of multiple algorithms and their proof of correctness.

High Performance

The scientific computing community insists on attaining high performance on whatever architectures are the state-of-the-art. The reason is that there is always interest in solving larger problems and computation time is often the limiting factor. The second half of the book demonstrates that the formal derivation methodology facilitates high performance. The key insight is that the matrix-matrix product operation can inherently achieve high performance, and that most computation intensive matrix operations can be arranged so that more computation involves matrix-matrix multiplication.

A High-Performance Library: libFLAME

The methods described in this book have been used to implement a software library for dense and banded linear algebra operations, libFLAME. This library is available under Open Source license. For information, visit `http://www.cs.utexas.edu/users/flame/`.

Intended Audience

This book is in part a portal for accessing research papers, tools, and libraries that were and will be developed as part of the Formal Linear Algebra Methods Environment (FLAME) project that is being pursued by researchers at The University of Texas at Austin and other institutions. The basic knowledge behind the FLAME methodology is presented in a way that makes it accessible to novices (e.g., undergraduate students with a limited background in linear algebra and high-performance computing). However, the approach has been used to produce state-of-the-art high-performance linear algebra libraries, making the book equally interesting to experienced researchers and the practicing professional.

The audience of this book extends beyond those interested in the domain of linear algebra algorithms. It is of interest to students and scholars with interests in the theory of computing since it shows how to make the formal derivation of algorithms practical. It is of interest to the compiler community because the notation and APIs present programs at a much higher level of abstraction than traditional code does, which creates new opportunities for compiler optimizations. It is of interest to the scientific computing community since it shows how to develop routines for a matrix operation when that matrix operation is not supported by an existing library. It is of interest to the architecture community since it shows how algorithms and architectures interact. It is of interest to the generative programming community, since the systematic approach to deriving algorithms supports the mechanical derivation of algorithms and implementations.

Related Materials

While this book is meant to be self-contained, it may be desirably to use it in conjunction with books and texts that focus on various other topics related to linear algebra. A brief list follows.

- Gilbert Strang. *Linear Algebra and its Application, Third Edition.* Academic Press, 1988.

  Discusses the mathematics of linear algebra at a level appropriate for undergraduates.

- G. W. Stewart. *Introduction to Matrix Computations.* Academic Press, 1973.

  A basic text that discusses the numerical issues (the effects of roundoff when floating-point arithmetic is used) related to the topic.

- James W. Demmel. *Applied Numerical Linear Algebra.* SIAM, 1997.

  Lloyd N. Trefethen and David Bau, III. *Numerical Linear Algebra.* SIAM, 1997.

  David S. Watkins. *Fundamentals of Matrix Computations, Second Edition.* John Wiley and Sons, Inc., 2002.

  Texts that discuss numerical linear algebra at the introductory graduate level.

- Gene H. Golub and Charles F. Van Loan. *Matrix Computations, Third Edition.* The Johns Hopkins University Press, 1996

  Advanced text that is best used as a reference or as a text for a class with a more advanced treatment of the topics.

- G. W. Stewart. *Matrix Algorithms Volume 1: Basic Decompositions.* SIAM, 1998.

  A systematic, but more traditional, treatment of many of the matrix operations and the related numerical issues.

- Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms, Second Edition.* SIAM, 2002.

  An advanced book on the numerical analysis of linear algebra algorithms.

In addition, we recommend the following manuscripts for those who want to learn more about formal verification and derivation of programs.

- David Gries. *The Science of Programming.* Springer-Verlag, 1981.

  A text on the formal derivation and verification of programs.

- Paolo Bientinesi. *Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms.* Department of Computer Sciences, University of Texas at Austin, August 2006.

  Chapter 2 of this dissertation systematically justifies the structure of the worksheet and explains in even more detail how it relates to formal derivation methods. It shows how the FLAME methodology can be made mechanical and how it enables the systematic stability analysis of the algorithms that are derived. We highly recommend reading this dissertation upon finishing this text.

Since formating the algorithms takes center stage in our approach, we recommend the classic reference for the LaTeX document preparation systems:

- Leslie Lamport. *LaTeX: A Document Preparation System, Second Edition.* Addison-Wesley Publishing Company, Inc., 1994.

  User's guide and reference manual for typesetting with LaTeX.

Webpages

A companion webpage has been created for this book. The base address is

`http://www.cs.utexas.edu/users/flame/books/TSoPMC/`

(TSoPMC: The Science of Programming Matrix Computations). In the text the above path will be referred to as `$BASE/`. On these webpages we have posted errata, additional materials, hints for the exercises, tools, and software. We suggest the reader visit this website at this time to become familiar with its structure and content.

Wiki: `www.linearalgebrawiki.org`

Many examples of operations, algorithms, derivations, and implementations similar to those discussed in this book can be found at

`http://www.linearalgebrawiki.org/`

Why `www.lulu.com`?

We considered publishing this book through more conventional channels. Indeed three major publishers of technical books offered to publish it (and two politely declined). The problem, however, is that the cost of textbooks has spiralled out of control and, given that we envision this book primarily as a reference and a supplemental text, we could not see ourselves adding to the strain this places on students. By publishing it ourselves through `www.lulu.com`, we have reduced the cost of a copy to a level where it is hardly worth printing it oneself. Since we retain all rights to the material, we may or may not publish future editions the same way, or through a conventional publisher.

Please visit `$BASE/books/TSoPMC/` for details on how to purchase this book.

Acknowledgments

This research was partially sponsored by NSF grants ACI-0305163, CCF-0342369, CCF-0540926, and CCF-0702714. Additional support came from the *J. Tinsley Oden Faculty Fellowship Research Program* of the Institute for Computational Engineering and Sciences (ICES) at UT-Austin, a donation by Dr. James Truchard (President, CEO, and Co-Founder of National Instruments), and an unrestricted grant from NEC Solutions (America), Inc.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Chapter 1

Motivation

Programming high-performance routines for computing linear algebra operations has long been a fine art. In this book we show it to be a science by exposing a systematic approach that given an operation yields high-performance implementations for computing it. The methodology builds upon a new notation for expressing algorithms, new advances regarding the formal derivation of linear algebra algorithms, a new style of coding, and the use of high-performance implementations of a few key linear algebra operations. In this chapter we preview the approach.

Don't Panic: A reader who is less well-versed in linear algebra should not feel intimidated by this chapter: It is meant to demonstrate to more experienced readers that there is substance to the book. In Chapter 2, we start over, more slowly. Indeed, a novice may wish to skip Chapter 1, and return to it later.

1.1 A Motivating Example: the LU Factorization

Consider the linear system $Ax = b$, where the square matrix A and the right-hand side vector b are the input, and the solution vector x is to be computed. Under certain conditions, a unique solution to this equation exists. *Gaussian elimination* is a technique, that is typically first introduced in high school, for computing this solution.

In Chapter 6, we will link Gaussian elimination to the computation of the *LU factorization* of matrix A. For now it suffices to understand that, given a square matrix A, the LU factorization computes a *unit lower triangular* matrix L and an *upper triangular* matrix U so that $A = LU$. Given that x must satisfy $Ax = b$, we find that $LUx = b$, or $L(Ux) = b$. This suggests that once the LU factorization has been computed, the solution to the linear system can be computed by first solving the triangular linear system $Ly = b$ for y, after which x is computed from the triangular linear system $Ux = y$.

A sketch of a standard algorithm for computing the LU factorization is illustrated in Figure 1.1 (left). There,

(a) Partition

$$A \rightarrow \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right), \quad L \rightarrow \left(\begin{array}{c|c} 1 & 0 \\ \hline l_{21} & L_{22} \end{array}\right), \quad U \rightarrow \left(\begin{array}{c|c} \upsilon_{11} & u_{12}^T \\ \hline 0 & U_{22} \end{array}\right).$$

(b)
- $\upsilon_{11} := \alpha_{11}$, overwriting α_{11}. (No-op)
- $u_{12}^T := a_{12}^T$, overwriting a_{12}^T. (No-op)
- $l_{21} := a_{21}/\upsilon_{11}$, overwriting a_{21}.
- $A_{22} := A_{22} - l_{21}u_{12}^T$.

(c) Continue by computing the LU factorization of A_{22}. (Back to Step (a) with $A = A_{22}$.)

Algorithm: $A := \text{LU_UNB_VAR5}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
  where A_{TL} is 0×0
while $m(A_{TL}) < m(A)$ **do**

  Repartition
  $$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$
  where α_{11} is a scalar

  $\alpha_{11} := \upsilon_{11} = \alpha_{11}$
  $a_{12}^T := u_{12}^T = a_{12}^T$
  $a_{21} := l_{21} = a_{21}/\upsilon_{11}$
  $A_{22} := A_{22} - l_{21}u_{12}^T$

  Continue with
  $$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$
endwhile

Figure 1.1: Left: Typical explanation of an algorithm for computing the LU factorization, overwriting A with L and U. Right: Same algorithm, using our notation.

matrices L and U overwrite the lower and upper triangular parts of A, respectively, and the diagonal of L is not stored, since all its entries equal one. To show how the algorithm sweeps through the matrix the explanation is often accompanied by the pictures in Figure 1.2 (left). The thick lines in that figure track the progress through matrix A as it is updated.

1.2 Notation

In this book, we have adopted a non traditional notation that captures the pictures that often accompany the explanation of an algorithm. This notation was developed as part of our Formal Linear Algebra Methods Environment (FLAME) project [16, 3]. We will show that it facilitates a style of programming that allows the algorithm to be captured in code as well as the systematic derivation of algorithms [4].

In Figure 1.1(right), we illustrate how the notation is used to express the LU factorization algorithm so that it reflects the pictures in Figure 1.2. For added clarification we point to Figure 1.2 (right). The next few chapters will explain the notation in detail, so that for now we leave it to the intuition of the reader.

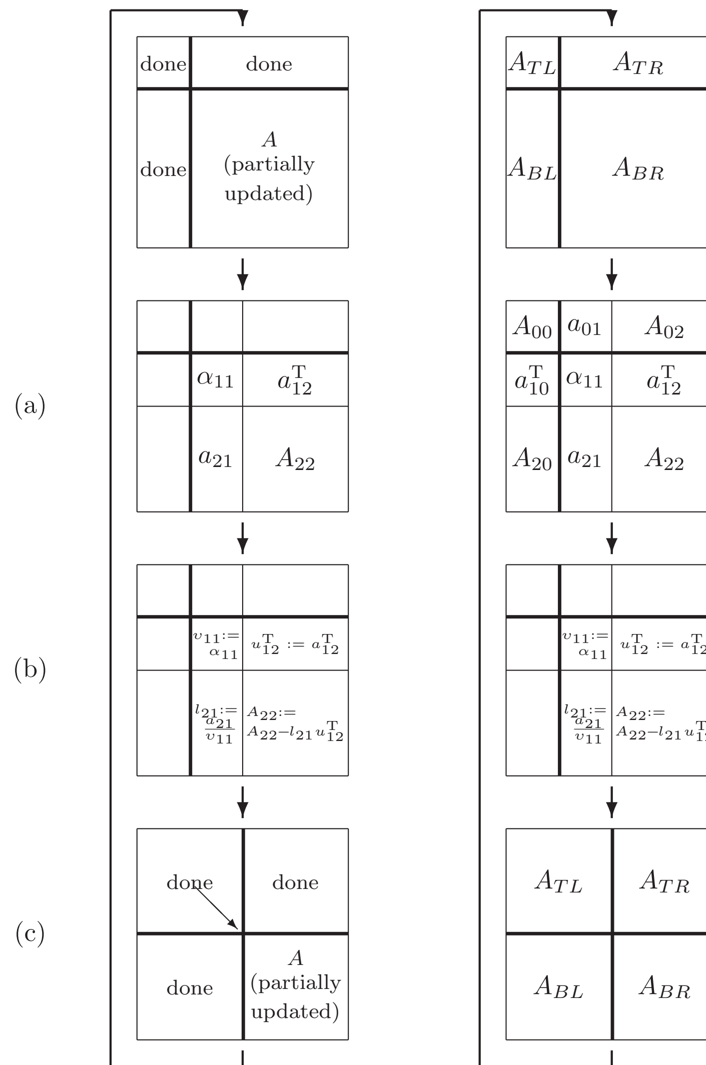

Figure 1.2: Progression of pictures that explain the LU factorization algorithm. Left: As typically presented. Right: Annotated with labels to explain the notation in Fig. 1.1(right).

Algorithm: $A := \text{LU_UNB}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
  where A_{TL} is 0×0
while $n(A_{TL}) < n(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

where α_{11} is a scalar

Variant 1:	
$a_{01} := L_{00}^{-1} a_{01}$	(TRSV)
$a_{10}^T := a_{10}^T U_{00}^{-1}$	(TRSV)
$\alpha_{11} := \alpha_{11} - a_{10}^T a_{01}$	(APDOT)
Variant 2:	
$a_{10}^T := a_{10}^T U_{00}^{-1}$	(TRSV)
$\alpha_{11} := \alpha_{11} - a_{10}^T a_{01}$	(APDOT)
$a_{12}^T := a_{12}^T - a_{10}^T A_{02}$	(GEMV)
Variant 3:	
$a_{01} := L_{00}^{-1} a_{01}$	(TRSV)
$\alpha_{11} := \alpha_{11} - a_{10}^T a_{01}$	(APDOT)
$a_{21} := (a_{21} - A_{20} a_{01}) / \alpha_{11}$	(GEMV,INVSCAL)
Variant 4:	
$\alpha_{11} := \alpha_{11} - a_{10}^T a_{01}$	(APDOT)
$a_{21} := (a_{21} - A_{20} a_{01}) / \alpha_{11}$	(GEMV,INVSCAL)
$a_{12}^T := a_{12}^T - a_{10}^T A_{02}$	(GEMV)
Variant 5:	
$a_{21} := a_{21} / \alpha_{11}$	(INVSCAL)
$A_{22} := A_{22} - a_{21} a_{12}^T$	(GER)

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

endwhile

Algorithm: $A := \text{LU_BLK}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
  where A_{TL} is 0×0
while $n(A_{TL}) < n(A)$ **do**
  Determine block size n_b
  Repartition

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

where A_{11} is $n_b \times n_b$

Variant 1:	
$A_{01} := L_{00}^{-1} A_{01}$	(TRSM)
$A_{10} := A_{10} U_{00}^{-1}$	(TRSM)
$A_{11} := \text{LU_UNB}(A_{11} - A_{10} A_{01})$	(GEMM,LU)
Variant 2:	
$A_{10} := A_{10} U_{00}^{-1}$	(TRSM)
$A_{11} := \text{LU_UNB}(A_{11} - A_{10} A_{01})$	(GEMM,LU)
$A_{12} := A_{12} - A_{10} A_{02}$	(GEMM)
Variant 3:	
$A_{01} := L_{00}^{-1} A_{01}$	(TRSM)
$A_{11} := \text{LU_UNB}(A_{11} - A_{10} A_{01})$	(GEMM,LU)
$A_{21} := (A_{21} - A_{20} A_{01}) U_{11}^{-1}$	(GEMM,TRSM)
Variant 4:	
$A_{11} := \text{LU_UNB}(A_{11} - A_{10} A_{01})$	(GEMM,LU)
$A_{21} := (A_{21} - A_{20} A_{01}) U_{11}^{-1}$	(GEMM,TRSM)
$A_{12} := L_{11}^{-1} (A_{12} - A_{10} A_{02})$	(GEMM,LU)
Variant 5:	
$A_{11} := \text{LU_UNB}(A_{11})$	(LU)
$A_{21} := A_{21} U_{11}^{-1}$	(TRSM)
$A_{12} := L_{11}^{-1} A_{12}$	(TRSM)
$A_{22} := A_{22} - A_{21} A_{12}$	(GEMM)

Continue with

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

endwhile

Figure 1.3: Multiple algorithms for computing the LU factorization. Matrices L_{ii} and U_{ii}, $i = 0, 1, 2$, denote, respectively, the unit lower triangular matrices and upper triangular matrices stored over the corresponding A_{ii}. Expressions involving L_{ii}^{-1} and U_{ii}^{-1} indicate the need to solve a triangular linear system.

Exercise 1.1 *Typesetting algorithms like those in Figure 1.1 (right) may seem somewhat intimidating. We have created a webpage that helps generate the LaTeX source as well as a set of LaTeXcommands (FLaTeX). Visit* `$BASE/Chapter1` *and follow the directions on the webpage associated with this exercise to try out the tools.*

1.3 Algorithmic Variants

For a given linear algebra operation there are often a number of loop-based algorithms. For the LU factorization, there are five algorithmic variants, presented in Figure 1.3 (left), that perform the same arithmetic computations on the same data, but in a different order. The first algorithm was proposed by Gauss around the beginning of the 19th century and the last by Crout in a paper dated in 1941 [5]. This raises the question of how to find all loop-based algorithms for a given linear algebra operation.

The ability to find all loop-based algorithms for a given operation is not just an academic exercise. Dense linear algebra operations occur as subproblems in many scientific applications. The performance attained when computing such an operation is dependent upon a number of factors, including the choice of algorithmic variant, details of the computer architecture, and the problem size. Finding and using the appropriate variant is in general worth the effort.

The topic of this book is the systematic derivation of a family of correct algorithms from the mathematical specification of a linear algebra operation. The key technique that enables this comes from formal derivation methods that date back to early work of Floyd [11], Dijkstra [8, 7], and Hoare [19], among others. One of the contribution of the FLAME project has been the formulation of a sequence of steps that leads to algorithms which are correct. These steps are first detailed for a simple case involving vectors in Chapter 2 and are demonstrated for progressively more complex operations throughout the book. They are straightforward and easy to apply even for people who lack a strong background in linear algebra or formal derivation methods. The scope of the derivation methodology is broad, covering most important dense linear algebra operations.

1.4 Presenting Algorithms in Code

There is a disconnect between how we express algorithms and how they are traditionally coded. Specifically, code for linear algebra algorithms traditionally utilizes explicit indexing into the arrays that store the matrices. This is a frequent source of errors as code is developed. In Figure 1.4 we show how the introduction of an appropriate *Application Programming Interface* (*API*), in this case for the C programming language, allows the code to closely resemble the algorithm. In particular, intricate indexing is avoided, diminishing the opportunity for the introduction of errors. APIs for M-script and the C programming languages are detailed in Chapter 4 and Appendix B. (Matlab Octave and LabView MathScript share a common scripting language, which we will refer to as M-script.)

Similar APIs can be easily defined for almost every programming language. Indeed, as part of the FLAME project, APIs have also been developed for Fortran, the traditional language of numerical linear algebra; func-

```
1   void LU_UNB_VAR5( FLA_Obj A )
2   {
3     FLA_Obj ATL,   ATR,      A00,  a01,     A02,
4             ABL,   ABR,      a10t, alpha11, a12t,
5                              A20,  a21,     A22;
6
7     FLA_Part_2x2( A,    &ATL, &ATR,
8                         &ABL, &ABR,     0, 0, FLA_TL );
9
10    while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ){
11      FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,    &A00,  /**/ &a01,     &A02,
12                           /* ******** */    /* ********************** */
13                                             &a10t, /**/ &alpha11, &a12t,
14                             ABL, /**/ ABR,    &A20,  /**/ &a21,     &A22,
15                             1, 1, FLA_BR );
16      /*------------------------------------------------------------*/
17
18      FLA_Inv_scal( alpha11, a21 );              /* a21 := a21 / alpha11    */
19      FLA_Ger( FLA_MINUS_ONE, a21, a12t, A22 );  /* A22 := A22 - a21 * a12t */
20
21      /*------------------------------------------------------------*/
22      FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,    A00,  a01,     /**/ A02,
23                                                    a10t, alpha11, /**/ a12t,
24                                /* ********** */    /* ******************* */
25                                &ABL, /**/ &ABR,    A20,  a21,     /**/ A22,
26                                FLA_TL );
27    }
28  }
```

Figure 1.4: C code for the algorithm in Figure 1.1.

tional programming languages such as HASKELL and MATHEMATICA; and the LABVIEW G graphical programming language.

Exercise 1.2 *The formating in Figure 1.4 is meant to, as closely as possible, resemble the algorithm in Figure 1.1(right). The same webpage that helps generate LaTeX source can also generate an outline for the code. Visit* `$BASE/Chapter1` *and duplicate the code in Figure 1.4 by following the directions on the webpage associated with this exercise.*

1.5 High Performance and Blocked Algorithms

The scientific applications that give rise to linear algebra operations often demand that high performance is achieved by libraries that compute the operations. As a result, it is crucial to derive and implement high-performance algorithms.

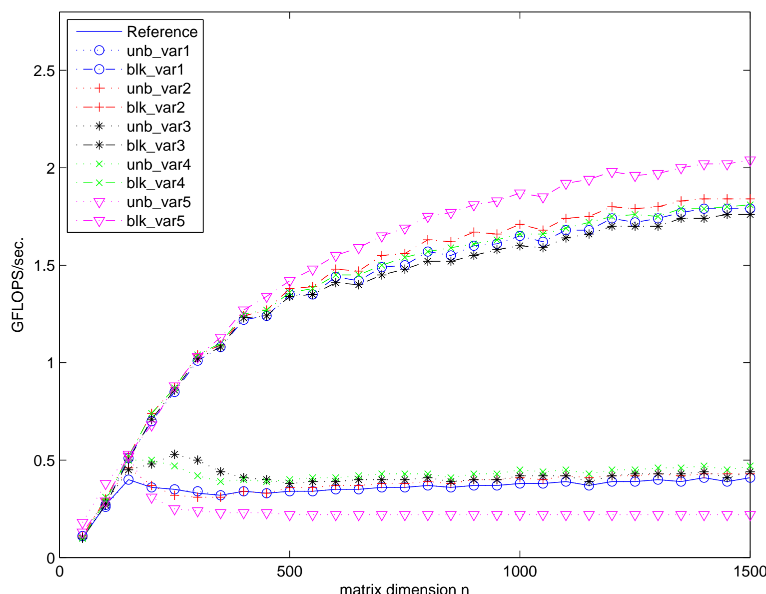

Figure 1.5: Performance of unblocked and blocked algorithmic variants for computing the LU factorization.

The key insight is that cache-based architectures, as are currently popular, perform *floating-point operations* (*flops*) at very fast rates, but fetch data from memory at a (relatively) much slower rate. For operations like the *matrix-matrix multiplication* (GEMM) this memory bandwidth bottleneck can be overcome by moving submatrices into the processor cache(s) and amortizing this overhead over a large number of flops. This is facilitated by the fact that GEMM involves a number of operations of cubic order on an amount of data that is of quadratic order. Details of how high performance can be attained for GEMM are exposed in Chapter 5.

Given a high-performance implementation of GEMM, other operations can attain high performance if the bulk of the computation can be cast in terms of GEMM. This is property of *blocked algorithms*. Figure 1.3 (right) displays blocked algorithms for the different algorithmic variants that compute the LU factorization. We will show that the derivation of blocked algorithms is typically no more complex than the derivation of their unblocked counterparts.

The performance of a code (an implementation of an algorithm) is often measured in terms of the rate at which flops are performed. The maximal rate that can be attained by a target architecture is given by the product of the *clock rate* of the processor times the number of flops that are performed per *clock cycle*. The rate of computation for a code is computed by dividing the number of flops required to compute the operation by the time it takes for it to be computed. A *gigaflops* (or GFLOPS) indicates a billion flops per second. Thus,

an implementation that computes an operation that requires f flops in t seconds attains a rate of

$$\frac{f}{t} \times 10^{-9} \text{ GFLOPS.}$$

Throughout the book we discuss how to compute the cost, in flops, of an algorithm.

The number of flops performed by an LU factorization is about $\frac{2}{3}n^3$, where n is the matrix dimension. In Figure 1.5 we show the performance attained by implementations of the different algorithms in Figure 1.3 on an Intel® Pentium® 4 workstation. The clock speed of the particular machine is 1.4 GHz and a Pentium 4 can perform two flops per clock cycle, for a peak performance of 2.8 GFLOPS, which marks the top line in the graph. The block size n_b was taken to equal 128. (We will eventually discuss how to determine a near-optimal block size.) Note that blocked algorithms attain much better performance than unblocked algorithms and that not all algorithmic variants attain the same performance.

In Chapter 5 it will become clear why we favor loop-based algorithms over recursive algorithms, and how recursion does enter the picture.

1.6 Numerical Stability

The numerical properties, like stability of algorithms, for linear algebra operations is a very important issue. We recommend using this book in conjunction with one or more of the books, mentioned in the preface, that treat numerical issues fully. In [2] it is discussed how the systematic approach for deriving algorithms can be extended so that a stability analysis can be equally systematically derived.

Chapter 2

Derivation of Linear Algebra Algorithms

This chapter introduces the reader to the systematic derivation of algorithms for linear algebra operations. Through a very simple example we illustrate the core ideas: We describe the notation we will use to express algorithms; we show how assertions can be used to establish correctness; and we propose a goal-oriented methodology for the derivation of algorithms. We also discuss how to incorporate an analysis of the cost into the algorithm.

Finally, we show how to translate algorithms to code so that the correctness of the algorithm implies the correctness of the implementation.

2.1 A Farewell to Indices

In this section, we introduce a notation for expressing algorithms that avoids the pitfalls of intricate indexing and will allow us to more easily derive, express, and implement algorithms. We present the notation through a simple example, the *inner product* of two vectors, an operation that will be used throughout this chapter for illustration.

Given two vectors, x and y, of length m, the *inner product* or *dot product* (DOT) of these vectors is given by

$$\alpha := x^{\mathrm{T}} y = \sum_{i=0}^{m-1} \chi_i \psi_i$$

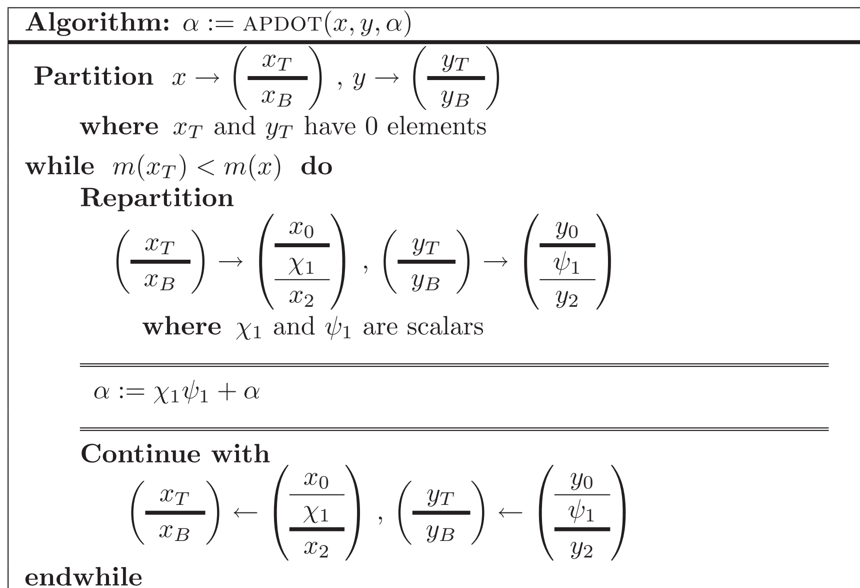

Figure 2.1: Algorithm for computing $\alpha := x^{\mathrm{T}}y + \alpha$.

where χ_i and ψ_i equal the ith elements of x and y, respectively:

$$x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{m-1} \end{pmatrix} \quad \text{and} \quad y = \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{m-1} \end{pmatrix}.$$

Remark 2.1 *We will use the symbol ":=" ("becomes") to denote* assignment *while the symbol "=" is reserved for* equality.

Example 2.2 *Let*

$$x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad \textit{and} \quad y = \begin{pmatrix} 2 \\ 4 \\ 1 \end{pmatrix}.$$

Then $x^{\mathrm{T}}y = 1 \cdot 2 + 2 \cdot 4 + 3 \cdot 1 = 13$. *Here we make use of the symbol "$\cdot$" to denote the* arithmetic product.

A traditional loop for implementing the updating of a scalar by adding a dot product to it, $\alpha := x^{\mathrm{T}}y + \alpha$, is given by

$k := 0$
while $k < m$ **do**
  $\alpha := \chi_k \psi_k + \alpha$
  $k := k + 1$
endwhile

Our notation presents this loop as in Figure 2.1. The name of the algorithm in that figure reflects that it performs a *alpha plus dot product* (APDOT). To interpret the algorithm in Figure 2.1 note the following:

- We bid farewell to intricate indexing: In this example only indices from the sets $\{T, B\}$ (*Top* and *Bottom*) and $\{0, 1, 2\}$ are required.

- Each vector has been subdivided into two subvectors, separated by thick lines. This is how we will represent systematic movement through vectors (and later matrices).

- Subvectors x_T and y_T include the "top" elements of x and y that, in this algorithm, have already been used to compute a partial update to α. Similarly, subvectors x_B and y_B include the "bottom" elements of x and y that, in this algorithm, have not yet been used to update α. Referring back to the traditional loop, x_T and y_T consist of elements $0, \ldots, k-1$ and x_B and y_B consist of elements $k, \ldots, m-1$:

$$\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) = \left(\begin{array}{c} \chi_0 \\ \vdots \\ \chi_{k-1} \\ \hline \chi_k \\ \vdots \\ \chi_{m-1} \end{array}\right) \quad \text{and} \quad \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} \psi_0 \\ \vdots \\ \psi_{k-1} \\ \hline \psi_k \\ \vdots \\ \psi_{m-1} \end{array}\right).$$

- The initialization before the loop starts

  Partition $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$, $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$
  where x_T and y_T have 0 elements

  takes the place of the assignment $k := 0$ in the traditional loop.

- The loop is executed as long as $m(x_T) < m(x)$ is *true*, which takes the place of $k < m$ in the traditional loop. Here $m(x)$ equals the length of vector x so that the loop terminates when x_T includes all elements of x.

- The statement

Repartition

$$\left(\frac{x_T}{x_B}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{array}\right), \left(\frac{y_T}{y_B}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right)$$

where χ_1 and ψ_1 are scalars

exposes the top elements of x_B and y_B, χ_1 and ψ_1 respectively, which were χ_k and ψ_k in the traditional loop.

- The exposed elements χ_1 and ψ_1 are used to update α in

  $$\alpha := \chi_1\psi_1 + \alpha,$$

  which takes the place of the update $\alpha := \chi_k\psi_k + \alpha$ in the traditional loop.

Remark 2.3 *It is important not to confuse the single elements exposed in our repartitionings, such as χ_1 or ψ_1, with the second entries of corresponding vectors.*

- The statement

  Continue with

  $$\left(\frac{x_T}{x_B}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{array}\right), \left(\frac{y_T}{y_B}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right)$$

  moves the top elements of x_B and y_B to x_T and y_T, respectively. This means that these elements have now been used to update α and should therefore be added to x_T and y_T.

Exercise 2.4 *Follow the instructions at* `$BASE/Chapter2/` *to duplicate Figure 2.1.*

Exercise 2.5 *Consider the following loop, which computes $\alpha := x^T y + \alpha$ backwards:*

$k := m - 1$
while $k \geq 0$ **do**
  $\alpha := \chi_k\psi_k + \alpha$
  $k := k - 1$
endwhile

Modify the algorithm in Figure 2.1 so that it expresses this alternative algorithmic variant. Typeset the resulting algorithm.

2.2 Predicates as Assertions about the State

To reason about the correctness of algorithms, predicates will be used to express assertions about the state (contents) of the variables. Recall that a *predicate*, P, is simply a *Boolean expression* that evaluates to *true* or *false* depending on the state of the variables that appear in the predicate. The placement of a predicate at a specific point in an algorithm means that it must evaluate to *true* so that it asserts the state of the variables that appear in the predicate. An assertion is a predicate that is used in this manner.

For example, after the command

$\alpha := 1$

which assigns the value 1 to the scalar variable α, we can assert that the predicate "$P : \alpha = 1$" holds (is true). An assertion can then be used to indicate the state of variable α after the assignment as in

$\alpha := 1$
$\{P : \alpha = 1\}$

Remark 2.6 *Assertions will be enclosed by curly brackets,* { }, *in the algorithms.*

If P_{pre} and P_{post} are predicates and S is a sequence of commands, then $\{P_{pre}\}S\{P_{post}\}$ is a predicate that evaluates to *true* if and only if the execution of S, when begun in a state satisfying P_{pre}, terminates in a finite amount of time in a state satisfying P_{post}. Here $\{P_{pre}\}S\{P_{post}\}$ is called the *Hoare triple*, and P_{pre} and P_{post} are referred to as the *precondition* and *postcondition* for the triple, respectively.

Example 2.7 $\{\alpha = \beta\}\,\alpha := \alpha + 1\,\{\alpha = \beta + 1\}$ *evaluates to true. Here "*$\alpha = \beta$*" is the precondition while "*$\alpha = (\beta + 1)$*" is the postcondition.*

2.3 Verifying Loops

Consider the loop in Figure 2.1, which has the form

while G **do**
  S
endwhile

Here, G is a Boolean expression known as the *loop-guard* and S is the sequence of commands that form the *loop-body*. The loop is executed as follows: If G is *false*, then execution of the loop terminates; otherwise S is executed and the process is repeated. Each execution of S is called an *iteration*. Thus, if G is initially *false*, no iterations occur.

We now formulate a theorem that can be applied to prove correctness of algorithms consisting of loops. For a proof of this theorem (using slightly different notation), see [15].

Step	Annotated Algorithm: $\alpha := x^T y + \alpha$
1a	$\{\alpha = \hat{\alpha} \wedge 0 \le m(x) = m(y)\}$
4	**Partition** $x \rightarrow \left(\frac{x_T}{x_B}\right)$, $y \rightarrow \left(\frac{y_T}{y_B}\right)$ **where** x_T and y_T have 0 elements
2	$\{(\alpha = x_T^T y_T + \hat{\alpha}) \wedge (0 \le m(x_T) = m(y_T) \le m(x))\}$
3	**while** $m(x_T) < m(x)$ **do**
2,3	$\{((\alpha = x_T^T y_T + \hat{\alpha}) \wedge (0 \le m(x_T) = m(y_T) \le m(x))) \wedge (m(x_T) < m(x))\}$
5a	**Repartition** $\left(\frac{x_T}{x_B}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{array}\right)$, $\left(\frac{y_T}{y_B}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right)$ **where** χ_1 and ψ_1 are scalars
6	$\{(\alpha = x_0^T y_0 + \hat{\alpha}) \wedge (0 \le m(x_0) = m(y_0) < m(x))\}$
8	$\alpha := \chi_1 \psi_1 + \alpha$
5b	**Continue with** $\left(\frac{x_T}{x_B}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{array}\right)$, $\left(\frac{y_T}{y_B}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right)$
7	$\{(\alpha = x_0^T y_0 + \chi_1 \psi_1 + \hat{\alpha}) \wedge (0 < m(x_0) + 1 = m(y_0) + 1 \le m(x))\}$
2	$\{(\alpha = x_T^T y_T + \hat{\alpha}) \wedge (0 \le m(x_T) = m(y_T) \le m(x))\}$
	endwhile
2,3	$\{((\alpha = x_T^T y_T + \hat{\alpha}) \wedge (0 \le m(x_T) = m(y_T) \le m(x))) \wedge \neg (m(x_T) < m(x))\}$
1b	$\{\alpha = x^T y + \hat{\alpha}\}$

Figure 2.2: Annotated algorithm for computing $\alpha := x^T y + \alpha$.

Theorem 2.8 (Fundamental Invariance Theorem) *Given the loop*

while G **do** S **endwhile**

and a predicate P_{inv} *assume that*

1. $\{P_{inv} \wedge G\}S\{P_{inv}\}$ *holds – execution of* S *begun in a state in which* P_{inv} *and* G *are* true *terminates with* P_{inv} true *–, and*
2. *execution of the loop begun in a state in which* P_{inv} *is* true *terminates.*

Then, if the loop is entered in a state where P_{inv} *is* true*, it will complete in a state where* P_{inv} *is* true *and the loop-guard* G *is false.*

Here the symbol "$\wedge$" denotes the *logical and* operator.

This theorem can be interpreted as follows. Assume that the predicate P_{inv} holds before and after the loop-body. Then, *if* P_{inv} holds before the loop, obviously it will also hold before the loop-body. The commands in the loop-body are such that it holds again after the loop-body, which means that it will again be *true* before the loop-body in the next iteration. We conclude that it will be *true* before and after the loop-body every time through the loop. When G becomes false, P_{inv} will still be *true*, and therefore it will be *true* after the loop completes (if the loop can be shown to terminate), we can assert that $P_{inv} \wedge \neg G$ holds after the completion of the loop, where the symbol "$\neg$" denotes the *logical negation*. This can be summarized by

$$\begin{array}{lcl} & & \{P_{inv}\} \\ & & \textbf{while } G \textbf{ do} \\ \{P_{inv} \wedge G\} & & \quad \{P_{inv} \wedge G\} \\ S & \Longrightarrow & \quad S \\ \{P_{inv}\} & & \quad \{P_{inv}\} \\ & & \textbf{endwhile} \\ & & \{P_{inv} \wedge \neg G\} \end{array}$$

if the loop can be shown to terminate. Here $\Longrightarrow$ stands for "implies". The assertion P_{inv} is called the *loop-invariant* for this loop.

Let us again consider the computation $\alpha := x^{\mathrm{T}} y + \alpha$. Let us use $\hat{\alpha}$ to denote the *original contents* (or state) of α. Then we define the precondition for the algorithm as

$$P_{pre} : \alpha = \hat{\alpha} \wedge 0 \leq m(x) = m(y),$$

and the postcondition as

$$P_{post} : \alpha = x^{\mathrm{T}} y + \hat{\alpha};$$

that is, the result the operation to be computed.

In order to prove the correctness, we next annotate the algorithm in Figure 2.1 with assertions that describe the state of the variables, as shown in Figure 2.2. Each command in the algorithm has the property that, when entered in a state described by the assertion that precedes it, it will complete in a state where the assertion immediately after it holds. In that *annotated algorithm*, the predicate

$$P_{inv} : (\alpha = x_T^{\mathrm{T}} y_T + \hat{\alpha}) \wedge (0 \leq m(x_T) = m(y_T) \leq m(x))$$

is a loop-invariant as it holds

1. immediately before the loop (by the initialization and the definition of $x_T^{\mathrm{T}} y_T = 0$ when $m(x_T) = m(y_T) = 0$),
2. before the loop-body, and
3. after the loop-body.

Now, it is also easy to argue that the loop terminates so that, by the Fundamental Invariance Theorem, $\{P_{inv} \wedge \neg G\}$ holds after termination. Therefore,

$$\begin{array}{rcl} P_{inv} \wedge \neg G & \equiv & \underbrace{(\alpha = x_T^{\rm T} y_T + \hat{\alpha}) \wedge (0 \le m(x_T) = m(y_T) \le m(x))}_{P_{inv}} \wedge \underbrace{\neg(m(x_T) < m(x))}_{\neg G} \\ & \Longrightarrow & (\alpha = x_T^{\rm T} y_T + \hat{\alpha}) \wedge \underbrace{(0 \le m(x_T) = m(y_T) \le m(x)) \wedge (m(x_T) \ge m(x))}_{\Longrightarrow\ m(x_T) = m(y_T) = m(x)} \\ & \Longrightarrow & (\alpha = x^{\rm T} y + \hat{\alpha}), \end{array}$$

since x_T and y_T are subvectors of x and y and therefore $m(x_T) = m(y_T) = m(x)$ implies that $x_T = x$ and $y_T = y$. Thus we can claim that the algorithm correctly computes $\alpha := x^{\rm T} y + \alpha$.

2.4 Goal-Oriented Derivation of Algorithms

While in the previous sections we discussed how to add annotations to an existing algorithm in an effort to prove it correct, we now demonstrate how one can methodically and constructively *derive* correct algorithms. *Goal-oriented* derivation of algorithms starts with the specification of the operation for which an algorithm is to be developed. From the specification, assertions are systematically determined and inserted into the algorithm before commands are added. By then inserting commands that make the assertions *true* at the indicated points, the algorithm is developed, hand-in-hand with its proof of correctness.

We draw the attention of the reader again to Figure 2.2. The numbers in the left column, labeled **Step**, indicate in what order to fill out the annotated algorithm.

Step 1: Specifying the precondition and postcondition. The statement of the operation to be performed, $\alpha := x^{\rm T} y + \alpha$, dictates the precondition and postcondition indicated in Steps 1a and 1b. The precondition is given by

$$P_{pre} : \alpha = \hat{\alpha} \wedge 0 \le m(x) = m(y),$$

and the postcondition is

$$P_{post} : \alpha = x^{\rm T} y + \hat{\alpha}.$$

Step 2: Determining loop-invariants. As part of the computation of $\alpha := x^{\rm T} y + \alpha$ we will sweep through vectors x and y in a way that creates two different subvectors of each of those vectors: the parts that have already been used to update α and the parts that remain yet to be used in this update. This suggests a partitioning of x and y as

$$x \rightarrow \left(\frac{x_T}{x_B} \right) \quad \text{and} \quad y \rightarrow \left(\frac{y_T}{y_B} \right),$$

where $m(x_T) = m(y_T)$ since otherwise $x_T^T y_T$ is not well-defined.

We take these partitioned vectors and substitute them into the postcondition to find that

$$\alpha = \left(\frac{x_T}{x_B}\right)^T \left(\frac{y_T}{y_B}\right) + \hat{\alpha} = \left(\, x_T^T \; \big| \; x_B^T \, \right) \left(\frac{y_T}{y_B}\right) + \hat{\alpha},$$

or,

$$\alpha = x_T^T y_T + x_B^T y_B + \hat{\alpha}.$$

This *partitioned matrix expression* (*PME*) expresses the *final* value of α in terms of its original value and the partitioned vectors.

Remark 2.9 *The partitioned matrix expression (PME) is obtained by substitution of the partitioned operands into the postcondition.*

Now, at an intermediate iteration of the loop, α does not contain its final value. Rather, it contains some *partial* result towards that final result. This partial result should be reflected in the loop-invariant. One such intermediate state is given by

$$P_{inv} : (\alpha = x_T^T y_T + \hat{\alpha}) \wedge (0 \leq m(x_T) = m(y_T) \leq m(x)),$$

which we note is exactly the loop-invariant that we used to prove the algorithm correct in Figure 2.2.

Remark 2.10 *Once it is decided how to partition vectors and matrices into regions that have been updated and/or used in a consistent fashion, loop-invariants can be systematically determined* a priori.

Step 3: Choosing a loop-guard. The condition

$$P_{inv} \wedge \neg G \quad \equiv \quad ((\alpha = x_T^T y_T + \hat{\alpha}) \wedge (0 \leq m(x_T) = m(y_T) \leq m(x))) \wedge \neg G$$

must imply that "$P_{post} : \alpha = x^T y + \hat{\alpha}$" holds. *If* x_T and y_T equal all of x and y, respectively, then the loop-invariant implies the postcondition: The choice '$G : m(x_T) < m(x)$" satisfies the desired condition that $P_{inv} \wedge \neg G$ implies that $m(x_T) = m(x)$, as x_T must be a subvector of x, and

$$\underbrace{((\alpha = x_T^T y_T + \hat{\alpha}) \wedge (0 \leq m(x_T) = m(y_T) \leq m(x)))}_{P_{inv}} \;\wedge\; \underbrace{(m(x_T) \geq m(x))}_{\neg G}$$

$$\Longrightarrow \alpha = x^T y + \hat{\alpha},$$

as was already argued in the previous section. This loop-guard is entered in Step 3 in Figure 2.2.

Remark 2.11 *The loop-invariant and the postcondition together* prescribe *a (non-unique) loop-guard G.*

Step 4: Initialization. If we partition

$$x \rightarrow \left(\frac{x_T}{x_B}\right) \quad \text{and} \quad y \rightarrow \left(\frac{y_T}{y_B}\right),$$

where x_T and y_T have *no* elements, then we place variables α, x_T, x_B, y_T, and y_B in a state where the loop-invariant is satisfied. This initialization appears in Step 4 in Figure 2.2.

Remark 2.12 *The loop-invariant and the precondition together* prescribe *the initialization.*

Step 5: Progressing through the vectors. We now note that, as part of the computation, x_T and y_T start by containing no elements and must ultimately equal all of x and y, respectively. Thus, as part of the loop, elements must be taken from x_B and y_B and must be added to x_T and y_T, respectively. This is denoted in Figure 2.2 by the statements

Repartition

$$\left(\frac{x_T}{x_B}\right) \rightarrow \left(\frac{x_0}{\frac{\chi_1}{x_2}}\right), \left(\frac{y_T}{y_B}\right) \rightarrow \left(\frac{y_0}{\frac{\psi_1}{y_2}}\right)$$

where χ_1 and ψ_1 are scalars,

and

Continue with

$$\left(\frac{x_T}{x_B}\right) \leftarrow \left(\frac{\frac{x_0}{\chi_1}}{x_2}\right), \left(\frac{y_T}{y_B}\right) \leftarrow \left(\frac{\frac{y_0}{\psi_1}}{y_2}\right).$$

This notation simply captures the movement of χ_1, the top element of x_B, from x_B to x_T. Similarly ψ_1 moves from y_B to y_T. The movement through the vectors guarantees that the loop eventually terminates, which is one condition required for the Fundamental Invariance Theorem to apply.

Remark 2.13 *The initialization and the loop-guard together prescribe the movement through the vectors.*

Step 6: Determining the state after repartitioning. The repartitionings in Step 5a do not change the contents of α: it is an "indexing" operation. We can thus ask ourselves the question of what the contents of

α are in terms of the exposed parts of x and y. We can derive this state, P_{before}, via *textual substitution*: The repartitionings in Step 5a imply that

$$\frac{x_T = x_0}{x_B = \left(\frac{\chi_1}{x_2}\right)} \quad \text{and} \quad \frac{y_T = y_0}{y_B = \left(\frac{\psi_1}{y_2}\right)} .$$

If we substitute the expressions on the right of the equalities into the loop-invariant we find that

$$\alpha = x_T^{\mathrm{T}} y_T + \hat{\alpha}$$

implies that

$$\alpha = \underbrace{x_0}_{x_T}{}^T \underbrace{y_0}_{y_T} + \hat{\alpha},$$

which is entered in Step 6 in Figure 2.2.

Remark 2.14 *The state in Step 6 is determined via textual substitution.*

Step 7: Determining the state after moving the thick lines. The movement of the thick lines in Step 5b means that now

$$\frac{x_T = \left(\frac{x_0}{\chi_1}\right)}{x_B = x_2} \quad \text{and} \quad \frac{y_T = \left(\frac{y_0}{\psi_1}\right)}{y_B = y_2} ,$$

so that

$$\alpha = x_T^{\mathrm{T}} y_T + \hat{\alpha}$$

implies that

$$\alpha = \underbrace{\left(\frac{x_0}{\chi_1}\right)}_{x_T}{}^T \underbrace{\left(\frac{y_0}{\psi_1}\right)}_{y_T} + \hat{\alpha} = x_0^{\mathrm{T}} y_0 + \chi_1 \psi_1 + \hat{\alpha},$$

which is then entered as state P_{after} in Step 7 in Figure 2.2.

Remark 2.15 *The state in Step 7 is determined via textual substitution and the application of the rules of linear algebra.*

Step 8: Determining the update. Comparing the contents in Step 6 and Step 7 now tells us that the state of α must change from

$$P_{before} : \alpha = x_0^{\mathrm{T}} y_0 + \hat{\alpha}$$

to

$$P_{after} : \alpha = \underbrace{x_0^{\mathrm{T}} y_0 + \hat{\alpha}}_{\text{already in } \alpha} \quad + \chi_1 \psi_1,$$

which can be accomplished by updating α as

$$\alpha := \chi_1 \psi_1 + \alpha.$$

This is then entered in Step 8 in Figure 2.2.

Remark 2.16 *It is* not *the case that $\hat{\alpha}$ (the original contents of α) must be saved, and that the update $\alpha = x_0^{\mathrm{T}} y_0 + \chi_1 \psi_1 + \hat{\alpha}$ must be performed. Since α already contains $x_0^{\mathrm{T}} y_0 + \hat{\alpha}$, only $\chi_1 \psi_1$ needs to be added. Thus, $\hat{\alpha}$ is only needed to be able to reason about the correctness of the algorithm.*

Final algorithm. Finally, we note that all the annotations (in the grey boxes) in Figure 2.2 were only introduced to derive the statements of the algorithm. Deleting these produces the algorithm already stated in Figure 2.1.

Exercise 2.17 *Reproduce Figure 2.2 by visiting* `$BASE/Chapter2/` *and following the directions associated with this exercise.*

To assist in the typesetting, some LaTeX macros, from the FLAME-LaTeX (FLaTeX) API, are collected in Figure 2.3.

2.5 Cost Analysis

As part of deriving an algorithm, one should obtain a formula for its cost. In this section we discuss how to incorporate an analysis of the cost into the annotated algorithm.

Let us again consider the algorithm for computing $\alpha := x^{\mathrm{T}} y + \alpha$ in Figure 2.2 and assume that α and the elements of x and y are all real numbers. Each execution of the loop requires two flops: a multiply and an add. The cost of computing $\alpha := x^{\mathrm{T}} y + \alpha$ is thus given by

$$\sum_{k=0}^{m-1} 2 \text{ flops} = 2m \text{ flops}, \tag{2.1}$$

where $m = m(x)$.

Let us examine how one would prove the equality in (2.1). There are two approaches: one is to say "well, that is obvious" while the other proves it rigorously via mathematical induction:

FLaTeX command	Result
`\FlaTwoByOne{x_T}` `{x_B}`	$\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$

`\FlaThreeByOneB{x_0}`
`{x_1}`
`{x_2}`
	$\left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \end{array}\right)$

`\FlaThreeByOneT{x_0}`
`{x_1}`
`{x_2}`
	$\left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \end{array}\right)$

`\FlaOneByTwo{x_L}{x_R}`	$\left(\begin{array}{c\|c} x_L & x_R \end{array}\right)$
`\FlaOneByThreeR{x_0}{x_1}{x_2}`	$\left(\begin{array}{c\|c\|c} x_0 & x_1 & x_2 \end{array}\right)$
`\FlaOneByThreeL{x_0}{x_1}{x_2}`	$\left(\begin{array}{c\|c\|c} x_0 & x_1 & x_2 \end{array}\right)$
`\FlaTwoByTwo{A_{TL}}{A_{TR}}` `{A_{BL}}{A_{BR}}`	$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$

`\FlaThreeByThreeBR{A_{00}}{A_{01}}{A_{02}}`
`{A_{10}}{A_{11}}{A_{12}}`
`{A_{20}}{A_{21}}{A_{22}}`
	$\left(\begin{array}{c\|c\|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$

`\FlaThreeByThreeBL{A_{00}}{A_{01}}{A_{02}}`
`{A_{10}}{A_{11}}{A_{12}}`
`{A_{20}}{A_{21}}{A_{22}}`
	$\left(\begin{array}{c\|c\|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$

`\FlaThreeByThreeTR{A_{00}}{A_{01}}{A_{02}}`
`{A_{10}}{A_{11}}{A_{12}}`
`{A_{20}}{A_{21}}{A_{22}}`
	$\left(\begin{array}{c\|c\|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$

`\FlaThreeByThreeTL{A_{00}}{A_{01}}{A_{02}}`
`{A_{10}}{A_{11}}{A_{12}}`
`{A_{20}}{A_{21}}{A_{22}}`
	$\left(\begin{array}{c\|c\|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$

Figure 2.3: Various FLaTeX commands for typesetting partitioned matrices.

Step	**Cost Analysis:** $\alpha := \text{SAPDOT}(x, y, \alpha)$
1a	$C_{sf} = 0$ flops
4	**Partition** $x \rightarrow \left(\frac{x_T}{x_B}\right)$, $y \rightarrow \left(\frac{y_T}{y_B}\right)$ **where** x_T and y_T have 0 elements
2	$C_{sf} = 2m(x_T)$ flops
3	**while** $m(x_T) < m(x)$ **do**
2,3	$C_{sf} = 2m(x_T)$ flops
5a	**Repartition** $\left(\frac{x_T}{x_B}\right) \rightarrow \begin{pmatrix} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{pmatrix}$, $\left(\frac{y_T}{y_B}\right) \rightarrow \begin{pmatrix} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{pmatrix}$ **where** χ_1 and ψ_1 are scalars
6	$C_{sf} = 2m(x_0)$ flops
8	$\alpha := \chi_1 \psi_1 + \alpha$ **Cost:** 2 flops
5b	**Continue with** $\left(\frac{x_T}{x_B}\right) \leftarrow \begin{pmatrix} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{pmatrix}$, $\left(\frac{y_T}{y_B}\right) \leftarrow \begin{pmatrix} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{pmatrix}$
7	$C_{sf} = 2m(x_0) + 2$ flops
2	$C_{sf} = 2m(x_T)$ flops
	endwhile
2,3	$C_{sf} = 2m(x_T)$ flops
1b	**Total Cost:** $2m(x)$ flops

Figure 2.4: Cost analysis for the algorithm in Fig. 2.2.

- Base case. For $m = 0$:

$$\sum_{k=0}^{0-1} 2 = \sum_{k=0}^{-1} 2 = 0 = 2(0) = 2m.$$

- Assume $\sum_{k=0}^{m-1} 2 = 2m$. Show that $\sum_{k=0}^{(m+1)-1} 2 = 2(m+1)$:

$$\sum_{k=0}^{(m+1)-1} 2 = \left(\sum_{k=0}^{m-1} 2\right) + 2 = 2m + 2 = 2(m+1).$$

We conclude by the Principle of Mathematical Induction that $\sum_{k=0}^{m-1} 2 = 2m$.

This inductive proof can be incorporated into the worksheet as illustrated in Figure 2.4, yielding the *cost worksheet*. In that figure, we introduce C_{sf} which stands for *"Cost-so-far"*. Assertions are added to the worksheet indicating the computation cost incurred so far at the specified point in the algorithm. In Step 1a, the cost is given by $C_{sf} = 0$. At Step 2, just before the loop, this translates to $C_{sf} = 2m(x_T)$ since $m(x_T) = 0$ and the operation in Step 4 is merely an indexing operation, which does not represent useful computation and is therefore not counted. This is analogous to the base case in our inductive proof. The assertion that $C_{sf} = 2m(x_T)$ is true at the top of the loop-body is equivalent to the induction hypothesis. We will refer to this cost as the *cost-invariant* of the loop. We need to show that it is again true at the bottom of the loop-body, where $m(x_T)$ is one greater than $m(x_T)$ at the top of the loop. We do so by inserting $C_{sf} = 2m(x_0)$ in Step 6, which follows by textual substitution and the fact that the operations in Step 5a are indexing operations and do not count towards C_{sf}. The fact that two flops are performed in Step 8 and the operations in Step 5b are indexing operations means that $C_{sf} = 2m(x_0)+2$ at Step 7. Upon completion $m(x_T) = m(x_0)+1$ in Step 2, due to the fact that one element has been added to x_T, shows that $C_{sf} = 2m(x_T)$ at the bottom of the loop-body. Thus, as was true for the loop-invariant, $C_{sf} = 2m(x_T)$ upon leaving the loop. Since there $m(x_T) = m(x)$, so that the total cost of the algorithm is $2m(x)$ flops.

The above analysis demonstrates the link between a loop-invariant, a cost-invariant, and an inductive hypothesis. The proof of the Fundamental Invariance Theorem employs mathematical induction [15].

2.6 Summary

In this chapter, we have introduced the fundamentals of the FLAME approach in the setting of a simple example, the APDOT. Let us recap the highlights so far.

- In our notation algorithms are expressed without detailed indexing. By partitioning vectors into subvectors, the boundary between those subvectors can be used to indicate how far into the vector indexing has reached. Elements near that boundary are of interest since they may move across the boundary as they are updated and/or used in the current iteration. It is this insight that allows us to restrict indexing only to the sets $\{T, B\}$ and $\{0, 1, 2\}$ when tracking vectors.

- Assertions naturally express the state in which variables should be at a given point in an algorithm.

- Loop-invariants are the key to proving loops correct.

- Loop-invariants are systematically identified *a priori* from the postcondition, which is the specification of the computation to be performed. This makes the approach goal-oriented.

- Given a precondition, postcondition, and a specific loop-invariant, all other steps of the derivation are prescribed. The systematic method for deriving all these parts is embodied in Figure 2.5, which we will refer to as *the worksheet* from here on.

Step	**Annotated Algorithm:** $[D, E, F, \ldots] := \text{op}(A, B, C, D, \ldots)$
1a	$\{P_{pre}\}$
4	**Partition** **where**
2	$\{P_{inv}\}$
3	**while** G **do**
2,3	$\{(P_{inv}) \wedge (G)\}$
5a	**Repartition** **where**
6	$\{P_{before}\}$
8	S_U
5b	**Continue with**
7	$\{P_{after}\}$
2	$\{P_{inv}\}$
	endwhile
2,3	$\{(P_{inv}) \wedge \neg (G)\}$
1b	$\{P_{post}\}$

Figure 2.5: Worksheet for deriving an algorithm.

- An expression for the cost of an algorithm can be determined by summing the cost of the operations in the loop-body. A closed-form expression for this summation can then be proven correct by annotating the worksheet with a cost-invariant.

Remark 2.18 *A blank worksheet, to be used in subsequent exercises, can be obtained by visiting* `$BASE/Chapter2/`.

2.7 Other Vector-Vector Operations

A number of other commonly encountered operations involving vectors are tabulated in Figure 2.6.

2.8 Further Exercises

For additional exercises, visit `$BASE/Chapter2/`.

Name	Abbreviation	**Operation**	**Cost** (flops)
Scaling	SCAL	$x := \alpha x$	m
Inverse scaling	INVSCAL	$x := x/\alpha$	m
Addition	ADD	$y := x + y$	m
Dot (inner) product	DOT	$\alpha := x^T y$	$2m - 1$
Alpha plus DOT product	APDOT	$\alpha := \alpha + x^T y$	$2m$
Alpha x Plus y	AXPY	$y := \alpha x + y$	$2m$

Figure 2.6: Vector-vector operations. Here, α is a scalar while x and y are vectors of length m.

Chapter 3

Matrix-Vector Operations

The previous chapter introduced the FLAME approach to deriving and implementing linear algebra algorithms. The primary example chosen for that chapter was an operation that involved scalars and vectors only, as did the exercises in that chapter. Such operations are referred to as *vector-vector operations*. In this chapter, we move on to simple operations that combine matrices and vectors and that are thus referred to as *matrix-vector* operations.

Matrices differ from vectors in that their two-dimensional shape permits systematic traversal in multiple directions: While vectors are naturally accessed from top to bottom or vice-versa, matrices can be accessed row-wise, column-wise, and by quadrants, as we will see in this chapter. This multitude of ways in which matrices can be partitioned leads to a much richer set of algorithms.

We focus on three common matrix-vector operations, namely, the *matrix-vector product*, the *rank-1 update*, and the solution of a *triangular linear system of equations*. The latter will also be used to illustrate the derivation of a blocked variant of an algorithm, a technique that supports performance and modularity. We will see that these operations build on the vector-vector operations encountered in the previous chapter and become themselves building-blocks for blocked algorithms for matrix-vector operations, and more complex operations in later chapters.

3.1 Notation

A vector $x \in \mathbb{R}^m$ is an ordered tuple of m real numbers[1]. It is written as a column of elements, with parenthesis around it:

$$x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{m-1} \end{pmatrix}.$$

The parenthesis are there only for visual effect. In some cases, for clarity, we will include bars to separate the elements:

$$x = \left(\begin{array}{c} \chi_0 \\ \hline \chi_1 \\ \hline \vdots \\ \hline \chi_{m-1} \end{array}\right).$$

We adopt the convention that lowercase Roman letters are used for vector variables and lowercase Greek letters for scalars. Also, the elements of a vector are denoted by the Greek letter that corresponds to the Roman letter used to denote the vector. A table of corresponding letters is given in Appendix A.

If x is a column vector, then the row vector with identical elements organized as a row is denoted by x^{T}:

$$x^{\mathrm{T}} = (\chi_0, \chi_1, \ldots, \chi_{m-1}).$$

The "T" superscript there stands for *transposition*. Sometimes we will leave out the commas that separate the elements, replacing them with a blank instead, or we will use separation bars:

$$x^{\mathrm{T}} = \left(\begin{array}{cccc} \chi_0 & \chi_1 & \cdots & \chi_{m-1} \end{array} \right) = \left(\begin{array}{c|c|c|c} \chi_0 & \chi_1 & \cdots & \chi_{m-1} \end{array} \right).$$

Often it will be space-consuming to have a column vector in a sentence written as a column of its elements. Thus, rather than writing $x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{m-1} \end{pmatrix}$ we will then write $x = (\chi_0, \chi_1, \ldots, \chi_{m-1})^{\mathrm{T}}$.

A matrix $A \in \mathbb{R}^{m \times n}$ is a two-dimensional array of elements where its (i, j) element is given by α_{ij}:

$$A = \begin{pmatrix} \alpha_{00} & \alpha_{01} & \cdots & \alpha_{0,n-1} \\ \alpha_{10} & \alpha_{11} & \cdots & \alpha_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,n-1} \end{pmatrix}.$$

[1] Thorough the book we will take scalars, vectors, and matrices to be real valued. Most of the results also apply to the case where they are complex valued.

We adopt the convention that matrices are denoted by uppercase Roman letters. Frequently, we will partition A by columns or rows:

$$A = (a_0, a_1, \ldots, a_{n-1}) = \begin{pmatrix} \check{a}_0^{\mathrm{T}} \\ \check{a}_1^{\mathrm{T}} \\ \vdots \\ \check{a}_{m-1}^{\mathrm{T}} \end{pmatrix},$$

where a_j and $\check{a}_i^{\mathrm{T}}$ stand, respectively, for the jth column and the ith row of A. Both a_j and $\check{a}_i$ are vectors. The "ˇ" superscript is used to distinguish a_j from $\check{a}_j$. Following our convention, letters for the columns, rows, and elements are picked to correspond to the letter used for the matrix, as indicated in Appendix A.

Remark 3.1 *Lowercase Greek letters and Roman letters will be used to denote scalars and vectors, respectively. Uppercase Roman letters will be used for matrices.*
Exceptions to this rule are variables that denote the (integer) dimensions of the vectors and matrices which are denoted by Roman lowercase letters to follow the traditional convention.

During an algorithm one or more variables (scalars, vectors, or matrices) will be modified so that they no longer contain their *original values* (contents). Whenever we need to refer to the original contents of a variable we will put a "ˆ" symbol on top of it. For example, $\hat{A}$, $\hat{a}$, and $\hat{\alpha}$ will refer to the original contents (those before the algorithm commences) of A, a, and α, respectively.

Remark 3.2 *A variable name with a "ˆ" symbol on top of it refers to the original contents of that variable. This will be used for scalars, vectors, matrices, and also for parts (elements, subvectors, submatrices) of these variables.*

3.2 Linear Transformations and Matrices

While the reader has likely been exposed to the definition of the *matrix-vector product* before, we believe it to be a good idea to review *why* it is defined as it is.

Definition 3.3 *Let $\mathcal{F} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ be a* function *that maps a vector from $\mathbb{R}^n$ to a vector in $\mathbb{R}^m$. Then $\mathcal{F}$ is said to be a* linear transformation *if $\mathcal{F}(\alpha x + y) = \alpha \mathcal{F}(x) + \mathcal{F}(y)$ for all $\alpha \in \mathbb{R}$ and $x, y \in \mathbb{R}^n$.*

Consider the *unit basis vectors*, $e_j \in \mathbb{R}^n$, $0 \le j < n$, which are defined by the vectors of all zeroes except for the jth element, which equals 1:

$$e_j = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \begin{matrix} \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} j \text{ zeroes (elements } 0,1,\ldots,j-1) \\ \leftarrow \text{element } j \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} n-j-1 \text{ zeroes (elements } j+1,j+2,\ldots,n-1) \end{matrix}$$

Any vector $x = (\chi_0, \chi_1, \ldots, \chi_{n-1})^{\mathrm{T}} \in \mathbb{R}^n$ can then be written as a *linear combination* of these vectors:

$$x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{n-1} \end{pmatrix} = \chi_0 \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \chi_1 \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix} + \cdots + \chi_{n-1} \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix} = \chi_0 e_0 + \chi_1 e_1 + \cdots + \chi_{n-1} e_{n-1}.$$

If $\mathcal{F} : \mathbb{R}^n \to \mathbb{R}^m$ is a linear transformation, then

$$\begin{aligned} \mathcal{F}(x) &= \mathcal{F}(\chi_0 e_0 + \chi_1 e_1 + \cdots + \chi_{n-1} e_{n-1}) = \chi_0 \mathcal{F}(e_0) + \chi_1 \mathcal{F}(e_1) + \cdots + \chi_{n-1} \mathcal{F}(e_{n-1}) \\ &= \chi_0 a_0 + \chi_1 a_1 + \cdots + \chi_{n-1} a_{n-1}, \end{aligned}$$

where $a_j = \mathcal{F}(e_j) \in \mathbb{R}^m$, $0 \le j < n$. Thus, we conclude that if we know how the linear transformation $\mathcal{F}$ acts on the unit basis vectors, we can evaluate $\mathcal{F}(x)$ as a linear combination of the vectors a_j, $0 \le j < n$, with the coefficients given by the elements of x. The matrix $A \in \mathbb{R}^{m \times n}$ that has a_j, $0 \le j < n$, as its jth column thus *represents* the linear transformation $\mathcal{F}$, and the matrix-vector product Ax is defined as

$$\begin{aligned} Ax &\equiv \mathcal{F}(x) = \chi_0 a_0 + \chi_1 a_1 + \cdots + \chi_{n-1} a_{n-1} \\ &= \chi_0 \underbrace{\begin{pmatrix} \alpha_{00} \\ \alpha_{10} \\ \vdots \\ \alpha_{m-1,0} \end{pmatrix}}_{a_0} + \chi_1 \underbrace{\begin{pmatrix} \alpha_{01} \\ \alpha_{11} \\ \vdots \\ \alpha_{m-1,1} \end{pmatrix}}_{a_1} + \cdots + \chi_{n-1} \underbrace{\begin{pmatrix} \alpha_{0,n-1} \\ \alpha_{1,n-1} \\ \vdots \\ \alpha_{m-1,n-1} \end{pmatrix}}_{a_{n-1}} \\ &= \begin{pmatrix} \alpha_{00}\chi_0 & + & \alpha_{01}\chi_1 & + & \cdots & + & \alpha_{0,n-1}\chi_{n-1} \\ \alpha_{10}\chi_0 & + & \alpha_{11}\chi_1 & + & \cdots & + & \alpha_{1,n-1}\chi_{n-1} \\ \vdots & & \vdots & & & & \vdots \\ \alpha_{m-1,0}\chi_0 & + & \alpha_{m-1,1}\chi_1 & + & \cdots & + & \alpha_{m-1,n-1}\chi_{n-1} \end{pmatrix}. \end{aligned} \tag{3.1}$$

Exercise 3.4 *Let $A = (a_0, a_1, \ldots, a_{n-1}) \in \mathbb{R}^{m\times n}$ be a partitioning of A by columns. Show that $Ae_j = a_j$, $0 \le j < n$, from the definition of the matrix-vector product in (3.1).*

Exercise 3.5 *Let $\mathcal{F} : \mathbb{R}^n \to \mathbb{R}^n$ have the property that $\mathcal{F}(x) = x$ for all $x \in \mathbb{R}^n$. Show that*

(a) $\mathcal{F}$ is a linear transformation.

(b) $\mathcal{F}(x) = I_n x$, where I_n is the identity matrix *defined as $I_n = (e_0, e_1, \ldots, e_{n-1})$.*

The following two exercises relate to the distributive property of the matrix-vector product.

Exercise 3.6 *Let $\mathcal{F} : \mathbb{R}^n \to \mathbb{R}^m$ and $\mathcal{G} : \mathbb{R}^n \to \mathbb{R}^m$ both be linear transformations. Show that $\mathcal{H} : \mathbb{R}^n \to \mathbb{R}^m$ defined by $\mathcal{H}(x) = \mathcal{F}(x) + \mathcal{G}(x)$, is also a linear transformation. Next, let A, B, and C equal the matrices that represent $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{H}$, respectively. Explain why $C = A + B$ should be defined as the matrix that results from adding corresponding elements of A and B.*

Exercise 3.7 *Let $A \in \mathbb{R}^{m\times n}$ and $x, y \in \mathbb{R}^n$. Show that $A(x + y) = Ax + Ay$; that is, the matrix-vector product is distributive with respect to vector addition.*

Two important definitions, which will be used later in the book, are the following.

Definition 3.8 *A set of vectors $v_1, v_2, \ldots, v_n \in \mathbb{R}^m$ is said to be linearly independent if*

$$\nu_1 v_1 + \nu_2 v_2 + \cdots + \nu_n v_n = 0,$$

with $\nu_1, \nu_2, \ldots, \nu_n \in \mathbb{R}$, implies $\nu_1 = \nu_2 = \ldots = \nu_n = 0$.

Definition 3.9 *The column (row) rank of a matrix A is the maximal number of linearly independent column (row) vectors of A.*

Note that the row and column rank of a matrix are always equal.

Let us consider the operation $y := \alpha Ax + \beta y$, with $x \in \mathbb{R}^n$, $y =\in \mathbb{R}^m$ partitioned into elements, and $A \in \mathbb{R}^{m\times n}$ partitioned into elements, columns, and rows as discussed in Section 3.1. This is a more general form of the matrix-vector product and will be referred to as GEMV from here on. For simplicity, we consider $\alpha = \beta = 1$ in this section.

From

$$y := Ax + y = \chi_0 a_0 + \chi_1 a_1 + \cdots + \chi_{n-1} a_{n-1} + y \quad = \quad [[[[\chi_0 a_0 + \chi_1 a_1] + \cdots] + \chi_{n-1} a_{n-1}] + y],$$

we note that GEMV can be computed by repeatedly performing AXPY operations. Because of the commutative property of vector addition, the AXPYs in this expression can be performed in any order.

Next, we show that GEMV can be equally well computed as a series of APDOTs involving the rows of matrix A, vector x, and the elements of y:

$$y := Ax + y = \begin{pmatrix} \alpha_{00}\chi_0 + \alpha_{01}\chi_1 + & \cdots & +\alpha_{0,n-1}\chi_{n-1} \\ \alpha_{10}\chi_0 + \alpha_{11}\chi_1 + & \cdots & +\alpha_{1,n-1}\chi_{n-1} \\ \vdots & \ddots & \vdots \\ \alpha_{m-1,0}\chi_0 + \alpha_{m-1,1}\chi_1 + & \cdots & +\alpha_{m-1,n-1}\chi_{n-1} \end{pmatrix} + \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{m-1} \end{pmatrix} = \begin{pmatrix} \check{a}_0^T x + \psi_0 \\ \hline \check{a}_1^T x + \psi_1 \\ \hline \vdots \\ \hline \check{a}_{m-1}^T x + \psi_{m-1} \end{pmatrix}.$$

Example 3.10 *Consider*

$$x = \begin{pmatrix} 3 \\ 0 \\ 1 \end{pmatrix}, \quad y = \begin{pmatrix} 2 \\ 5 \\ 1 \\ 8 \end{pmatrix}, \quad \textit{and} \quad A = \begin{pmatrix} 3 & 1 & 2 \\ 1 & 2 & 2 \\ 9 & 0 & 2 \\ 7 & 1 & 0 \end{pmatrix}.$$

The matrix-vector product $y := Ax + y$ can be computed as several AXPY*s:*

$$y := \left[\left[\left[3 \begin{pmatrix} 3 \\ 1 \\ 9 \\ 7 \end{pmatrix} + 0 \begin{pmatrix} 1 \\ 2 \\ 0 \\ 1 \end{pmatrix}\right] + 1 \begin{pmatrix} 2 \\ 2 \\ 2 \\ 0 \end{pmatrix}\right] + \begin{pmatrix} 2 \\ 5 \\ 1 \\ 8 \end{pmatrix}\right] = \begin{pmatrix} 13 \\ 10 \\ 30 \\ 29 \end{pmatrix},$$

as well as via repeated APDOT*s:*

$$y := \begin{pmatrix} (3\ 1\ 2) \begin{pmatrix} 3 \\ 0 \\ 1 \end{pmatrix} + 2 \\ \hline (1\ 2\ 2) \begin{pmatrix} 3 \\ 0 \\ 1 \end{pmatrix} + 5 \\ \hline (9\ 0\ 2) \begin{pmatrix} 3 \\ 0 \\ 1 \end{pmatrix} + 1 \\ \hline (7\ 1\ 0) \begin{pmatrix} 3 \\ 0 \\ 1 \end{pmatrix} + 8 \end{pmatrix} = \begin{pmatrix} 3 \cdot 3 + 1 \cdot 0 + 2 \cdot 1 + 2 \\ 1 \cdot 3 + 2 \cdot 0 + 2 \cdot 1 + 5 \\ 9 \cdot 3 + 0 \cdot 0 + 2 \cdot 1 + 1 \\ 7 \cdot 3 + 1 \cdot 0 + 0 \cdot 1 + 9 \end{pmatrix} = \begin{pmatrix} 13 \\ 10 \\ 30 \\ 29 \end{pmatrix}.$$

Exercise 3.11 *For the data given in Example 3.10, show that the order in which the* AXPY*s are carried out does not affect the result.*

The following theorem is a generalization of the above observations.

Theorem 3.12 *Partition*

$$A \rightarrow \begin{pmatrix} A_{00} & A_{01} & \cdots & A_{0,\nu-1} \\ A_{10} & A_{11} & \cdots & A_{1,\nu-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{\mu-1,0} & A_{\mu-1,1} & \cdots & A_{\mu-1,\nu-1} \end{pmatrix}, \quad x \rightarrow \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{\mu-1} \end{pmatrix}, \quad \textit{and} \quad y \rightarrow \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{\nu-1} \end{pmatrix},$$

where $A_{i,j} \in \mathbb{R}^{m_i \times n_p}$, $x_j \in \mathbb{R}^{n_j}$, and $y_i \in \mathbb{R}^{m_i}$. Then, the ith subvector of $y = Ax$ is given by

$$y_i = \sum_{j=0}^{\nu-1} A_{i,j} x_j. \tag{3.2}$$

The proof of this theorem is tedious and is therefore skipped.

Two corollaries follow immediately from the above theorem. There corollaries provide the PMEs for two sets of loop-invariants from which algorithms for computing GEMV can be derived.

Corollary 3.13 *Partition matrix A and vector x as*

$$A \rightarrow \left(A_L \mid A_R \right) \quad \text{and} \quad x \rightarrow \left(\frac{x_T}{x_B} \right),$$

with $n(A_L) = m(x_T)$ (and, therefore, $n(A_R) = m(x_B)$). Here $m(X)$ and $n(X)$ equal the row and column size of matrix X, respectively. Then

$$Ax + y = \left(A_L \mid A_R \right) \left(\frac{x_T}{x_B} \right) + y = A_L x_T + A_R x_B + y.$$

Corollary 3.14 *Partition matrix A and vector y as*

$$A \rightarrow \left(\frac{A_T}{A_B} \right) \quad \text{and} \quad y \rightarrow \left(\frac{y_T}{y_B} \right),$$

so that $m(A_T) = m(y_T)$ (and, therefore, $m(A_B) = m(y_B)$). Then

$$Ax + y = \left(\frac{A_T}{A_B} \right) x + \left(\frac{y_T}{y_B} \right) = \left(\frac{A_T x + y_T}{A_B x + y_B} \right).$$

Remark 3.15 *Subscripts "L" and "R" will serve to specify the Left and Right submatrices/subvectors of a matrix/vector, respectively. Similarly, subscripts "T" and "B" will be used to specify the Top and Bottom submatrices/subvectors.*

Exercise 3.16 *Prove Corollary 3.13.*
Exercise 3.17 *Prove Corollary 3.14.*

Remark 3.18 *Corollaries 3.13 and 3.14 pose certain restrictions on the dimensions of the partitioned matrices/vectors so that the matrix-vector product is "consistently" defined for these partitioned elements.*
Let us share a few hints on what a conformal *partitioning is for the type of expressions encountered in* GEMV*.*
Consider a matrix A, two vectors x, y, and an operation which relates them as:

1. *$x + y$ or $x := y$; then, $m(x) = m(y)$, and any partitioning by elements in one of the two operands must be done conformally (with the same dimensions) in the other operand.*

2. *Ax; then, $n(A) = m(x)$ and any partitioning by columns in A must be conformally performed by elements in x.*

3.3 Algorithms for the Matrix-Vector Product

In this section we derive algorithms for computing the matrix-vector product as well as the cost of these algorithms.

Variant 1	Variant 2
$\left(\frac{y_T}{y_B}\right) = \left(\frac{A_T x + \hat{y}_T}{\hat{y}_B}\right) \wedge P_{cons}$	$\left(\frac{y_T}{y_B}\right) = \left(\frac{\hat{y}_T}{A_B x + \hat{y}_B}\right) \wedge P_{cons}$
Variant 3	Variant 4
$y = A_L x_T + \hat{y} \wedge P_{cons}$	$y = A_L x_T + \hat{y} \wedge P_{cons}$

Figure 3.1: Four loop-invariants for GEMV.

3.3.1 Derivation

We now derive algorithms for GEMV using the eight steps in the worksheet (Figure 2.5).

Remark 3.19 *In order to derive the following algorithms, we do not assume the reader is* a priori *aware of any method for computing* GEMV*. Rather, we apply systematically the steps in the worksheet to derive two different algorithms, which correspond to the computation of* GEMV *via a series of* AXPY*s or* APDOT*s.*

Step 1: Specifying the precondition and postcondition. The precondition for the algorithm is given by

$$P_{pre} : (A \in \mathbb{R}^{m\times n}) \wedge (y \in \mathbb{R}^m) \wedge (x \in \mathbb{R}^n),$$

while the postcondition is

$$P_{post} : y = Ax + \hat{y}.$$

Recall the use of $\hat{y}$ in the postcondition denoting the original contents of y.

In the second step we need to choose a partitioning for the operands involved in the computation. There are three ways of partitioning matrices: by rows, by columns, or into quadrants. While the first two were already possible for vectors, the third one is new. The next two subsections show that the partitionings by rows/columns naturally lead to the algorithms composed of APDOTs/AXPYs. The presentation of the third type of partitioning, by quadrants, is delayed until we deal with triangular matrices, later in this chapter.

Step 2: Determining loop-invariants. Corollaries 3.13 and 3.14 provide us with two PMEs from which loop-invariants can be determined:

$$\left(\frac{y_T}{y_B}\right) = \left(\frac{A_T x + \hat{y}_T}{A_B x + \hat{y}_B}\right) \wedge P_{cons} \quad \text{and} \quad y = A_L x_T + A_R x_B + \hat{y} P_{cons}.$$

Here $P_{cons} : m(y_T) = m(A_T)$ for the first PME and $P_{cons} : m(x_T) = n(A_L)$ for the second PME.

Remark 3.20 *We will often use the consistency predicate "P_{cons}" to establish conditions on the partitionings of the operands that ensure that operations are well-defined.*

A loop-invariant inherently describes an intermediate result towards the final result computed by a loop. The observation that only part of the computations have been performed before each iteration yields the four different loop-invariants in Figure 3.1.

Let us focus on Invariant 1:

$$P_{inv-1} : \left(\left(\frac{y_T}{y_B} \right) = \left(\frac{A_T x + \hat{y}_T}{\hat{y}_B} \right) \right) \wedge P_{cons}, \tag{3.3}$$

which reflects a state where elements of y_T have already been updated with the final result while elements of y_B have not. We will see next how the partitioning of A by rows together with this loop-invariant fixes all remaining steps in the worksheet and leads us to the algorithm identified as Variant 1 in Figure 3.2.

Step 3: Choosing a loop-guard. Upon completion of the loop, the loop-invariant is true, the loop-guard G is false, and the condition

$$P_{inv} \wedge \neg G \equiv \left(\left(\left(\frac{y_T}{y_B} \right) = \left(\frac{A_T x + \hat{y}_T}{\hat{y}_B} \right) \right) \wedge P_{cons} \right) \wedge \neg G \tag{3.4}$$

must imply that $y = Ax + \hat{y}$. Now, if y_T equals all of y then, by consistency, A_T equals all of A, and (3.4) implies that the postcondition is true. Therefore, we adopt "$G : m(y_T) < m(y)$" as the required loop-guard G for the worksheet.

Step 4: Initialization. Next, we must find an initialization that, ideally with a minimum amount of computations, sets the variables of the algorithm in a state where the loop-invariant (including the consistency condition) holds.

We note that the partitioning

$$A \rightarrow \left(\frac{A_T}{A_B} \right), \quad y \rightarrow \left(\frac{y_T}{y_B} \right),$$

where A_T has 0 rows and y_T has no elements, sets the variables A_T, A_B, y_T, and y_B in a state where the loop-invariant is satisfied. This initialization, which only involves indexing operations, appears in Step 4 in Figure 3.2.

Step 5: Progressing through the operands. As part of the computation, A_T and y_T, start by having no elements, but must ultimately equal all of A and y, respectively. Thus, as part of the loop, rows must be taken from A_B and added to A_T while elements must be moved from y_B to y_T. This is denoted in Figure 3.2 by the

Step	**Annotated Algorithm:** $y := Ax + y$
1a	$\{(A \in \mathbb{R}^{m \times n}) \wedge (y \in \mathbb{R}^m) \wedge (x \in \mathbb{R}^n)\}$
4	**Partition** $A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$, $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$ **where** A_T has 0 rows and y_T has 0 elements
2	$\left\{\left(\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} A_T x + \hat{y}_T \\ \hline \hat{y}_B \end{array}\right)\right) \wedge P_{cons}\right\}$
3	**while** $m(y_T) < m(y)$ **do**
2,3	$\left\{\left(\left(\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} A_T x + \hat{y}_T \\ \hline \hat{y}_B \end{array}\right)\right) \wedge P_{cons}\right) \wedge (m(y_T) < m(y))\right\}$
5a	**Repartition** $\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline a_1^{\mathrm{T}} \\ A_2 \end{array}\right)$, $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right)$ **where** a_1^{T} is a row and ψ_1 is a scalar
6	$\left\{\left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right) = \left(\begin{array}{c} A_0 x + \hat{y}_0 \\ \hat{\psi}_1 \\ \hline \hat{y}_2 \end{array}\right)\right\}$
8	$\psi_1 = a_1^{\mathrm{T}} x + \psi_1$
5b	**Continue with** $\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ a_1^{\mathrm{T}} \\ \hline A_2 \end{array}\right)$, $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right)$
7	$\left\{\left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right) = \left(\begin{array}{c} A_0 x + \hat{y}_0 \\ \hline a_1^{\mathrm{T}} x + \hat{\psi}_1 \\ \hat{y}_2 \end{array}\right)\right\}$
2	$\left\{\left(\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} A_T x + \hat{y}_T \\ \hline \hat{y}_B \end{array}\right)\right) \wedge P_{cons}\right\}$
	endwhile
2,3	$\left\{\left(\left(\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) = \left(\begin{array}{c} A_T x + \hat{y}_T \\ \hline \hat{y}_B \end{array}\right)\right) \wedge P_{cons}\right) \wedge \neg (m(y_T) < m(y))\right\}$
1b	$\{y = Ax + \hat{y}\}$

Figure 3.2: Annotated algorithm for computing $y := Ax + y$ (Variant 1).

repartitioning statements[2]

$$\left(\frac{A_T}{A_B}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline a_1^T \\ \hline A_2 \end{array}\right) \quad \text{and} \quad \left(\frac{y_T}{y_B}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right),$$

and the redefinition of the original partitionings in

$$\left(\frac{A_T}{A_B}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline a_1^T \\ \hline A_2 \end{array}\right) \quad \text{and} \quad \left(\frac{y_T}{y_B}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right).$$

This manner of moving the elements ensures that P_{cons} holds and that the loop terminates.

Step 6: Determining the state after repartitioning. The contents of y in terms of the partitioned matrix and vectors, P_{before} in the worksheet in Figure 2.5, is determined via textual substitution as follows. From the partitionings in Step 5a,

$$\begin{array}{c} A_T = A_0 \\ \hline A_B = \left(\begin{array}{c} a_1^T \\ \hline A_2 \end{array}\right) \end{array} \quad \text{and} \quad \begin{array}{c} y_T = y_0 \\ \hline y_B = \left(\begin{array}{c} \psi_1 \\ \hline y_2 \end{array}\right) \end{array},$$

if we substitute the quantities on the right of the equalities into the loop-invariant,

$$\left(\left(\frac{y_T}{y_B}\right) = \left(\frac{A_T x + \hat{y}_T}{\hat{y}_B}\right)\right) \wedge P_{cons},$$

we find that

$$\left(\begin{array}{c} y_0 \\ \hline \left(\begin{array}{c} \psi_1 \\ \hline y_2 \end{array}\right) \end{array}\right) = \left(\begin{array}{c} A_0 x + \hat{y}_0 \\ \hline \left(\begin{array}{c} \hat{\psi}_1 \\ \hline \hat{y}_2 \end{array}\right) \end{array}\right), \quad \text{or,} \quad \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right) = \left(\begin{array}{c} A_0 x + \hat{y}_0 \\ \hline \hat{\psi}_1 \\ \hline \hat{y}_2 \end{array}\right),$$

as entered in Step 6 in Figure 3.2.

Step 7: Determining the state after moving the thick lines. After moving the thick lines, in Step 5b

$$\left(\frac{y_T}{y_B}\right) = \left(\frac{A_T x + \hat{y}_T}{\hat{y}_B}\right)$$

[2] In the partitionings we do not use the superscript "ˇ" for the row a_1^T as, in this case, there is no possible confusion with a column of the matrix.

implies that

$$\left(\begin{array}{c}\left(\begin{array}{c} y_0 \\ \hline \psi_1 \end{array}\right) \\ \hline y_2 \end{array}\right) = \left(\begin{array}{c}\left(\begin{array}{c} A_0 \\ \hline a_1^T \end{array}\right) x + \left(\begin{array}{c} \hat{y}_0 \\ \hline \hat{\psi}_1 \end{array}\right) \\ \hline \hat{y}_2 \end{array}\right), \quad \text{or,} \quad \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right) = \left(\begin{array}{c} A_0 x + \hat{y}_0 \\ \hline a_1^T x + \hat{\psi}_1 \\ \hline \hat{y}_2 \end{array}\right).$$

This is entered as the state P_{after} in the worksheet in Figure 2.5, as shown in Step 7 in Figure 3.2.

Step 8: Determining the update. Comparing the contents in Step 6 and Step 7 now tells us that the contents of y must be updated from

$$\left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right) = \left(\begin{array}{c} A_0 x + \hat{y}_0 \\ \hline \hat{\psi}_1 \\ \hline \hat{y}_2 \end{array}\right) \quad \text{to} \quad \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right) = \left(\begin{array}{c} A_0 x + \hat{y}_0 \\ \hline a_1^T x + \hat{\psi}_1 \\ \hline \hat{y}_2 \end{array}\right).$$

Therefore, we conclude that y must be updated by adding $a_1^T x$ to ψ_1:

$$\psi_1 := a_1^T x + \psi_1,$$

which is entered as the corresponding update in Figure 3.2.

Final algorithm. By deleting the annotations (assertions) we finally obtain the algorithm for GEMV (Variant 1) given in Figure 3.3. All the arithmetic operations in this algorithm are performed in terms of APDOT.

Remark 3.21 *The partitionings together with the loop-invariant prescribe steps 3–8 of the worksheet.*

Exercise 3.22 *Derive an algorithm for computing $y := Ax + y$ using the Invariant 2 in Figure 3.1.*

Exercise 3.23 *Consider Invariant 3 in Figure 3.1. Provide all steps that justify the worksheet in Figure 3.4. State the algorithm without assertions.*

Exercise 3.24 *Derive an algorithm for computing $y := Ax + y$ using the Invariant 4 in Figure 3.1.*

3.3.2 Cost Analysis of GEMV

We next prove that the cost of the algorithm in Figure 3.3 is $2mn$ flops.

Each execution of the loop of the algorithm in Figure 3.3 is a APDOT which requires $2n$ flops, with $n = m(x) = n(A)$ (see Table 2.6). The cost of computing $y := Ax + y$ is thus given by

$$\sum_{k=0}^{m-1} 2n \text{ flops} = 2mn \text{ flops}, \tag{3.5}$$

Algorithm: $y := \text{MATVEC_VAR1}(A, x, y)$

Partition $A \rightarrow \left(\frac{A_T}{A_B}\right)$, $y \rightarrow \left(\frac{y_T}{y_B}\right)$
  where A_T has 0 rows and y_T has 0 elements
while $m(y_T) < m(y)$ **do**
  Repartition
  $$\left(\frac{A_T}{A_B}\right) \rightarrow \begin{pmatrix} A_0 \\ \hline a_1^T \\ \hline A_2 \end{pmatrix}, \left(\frac{y_T}{y_B}\right) \rightarrow \begin{pmatrix} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{pmatrix}$$
  where a_1^T is a row and ψ_1 is a scalar

  $\psi_1 = a_1^T x + \psi_1$

  Continue with
  $$\left(\frac{A_T}{A_B}\right) \leftarrow \begin{pmatrix} A_0 \\ \hline a_1^T \\ \hline A_2 \end{pmatrix}, \left(\frac{y_T}{y_B}\right) \leftarrow \begin{pmatrix} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{pmatrix}$$
endwhile

Figure 3.3: Algorithm for computing $y := Ax + y$ (Variant 1).

where $m = m(y) = m(A)$.

Consider now Figure 3.5. where assertions are added indicating the computation cost incurred so far at the specified points in the algorithm. In Step 1a, the cost is given by $C_{sf} = 0$. At Step 2, just before the loop, this translates to the cost-invariant $C_{sf} = 2m(y_T)n$ since $m(y_T) = 0$. We need to show that the cost-invariant, which is true at the top of the loop, is again true at the bottom of the loop-body, where $m(y_T)$ is one greater than $m(y_T)$ at the top of the loop. We do so by inserting $C_{sf} = 2m(y_0)n$ in Step 6, which follows by textual substitution and the fact that Step 5a is composed of indexing operations with no cost. As $2n$ flops are performed in Step 8 and the operations in Step 5b are indexing operations, $C_{sf} = 2(m(y_0) + 1)n$ at Step 7. Since $m(y_T) = m(y_0) + 1$ in Step 2, due to the fact that one element has been added to y_T, shows that $C_{sf} = 2m(y_T)n$ at the bottom of the loop-body. Thus, as was true for the loop-invariant, $C_{sf} = 2m(y_T)n$ upon leaving the loop. Since there $m(y_T) = m(y)$, we establish that the total cost of the algorithm is $2mn$ flops.

Exercise 3.25 *Prove that the costs of the algorithms corresponding to Variant 2–4 are also $2mn$ flops.*

Step	**Annotated Algorithm:** $y := Ax + y$
1a	$\{(A \in \mathbb{R}^{m\times n}) \wedge (y \in \mathbb{R}^m) \wedge (x \in \mathbb{R}^n)\}$
4	**Partition** $A \rightarrow \left(A_L \,\middle\|\, A_R \right)$, $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$ **where** A_L has 0 columns and x_T has 0 elements
2	$\{(y = A_L x_T + \hat{y}) \wedge P_{cons}\}$
3	**while** $m(x_T) < m(x)$ **do**
2,3	$\{((y = A_L x_T + \hat{y}) \wedge P_{cons}) \wedge (m(x_T) < m(x))\}$
5a	**Repartition** $\left(A_L \,\middle\|\, A_R \right) \rightarrow \left(A_0 \,\middle\|\, a_1 \,\middle\|\, A_2 \right)$, $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{array}\right)$ **where** a_1 is a column and χ_1 is a scalar
6	$\{y = A_0 x_0 + \hat{y}\}$
8	$y := y + \chi_1 a_1$
5b	**Continue with** $\left(A_L \,\middle\|\, A_R \right) \leftarrow \left(A_0 \,\middle\|\, a_1 \,\middle\|\, A_2 \right)$, $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{array}\right)$
7	$\{y = A_0 x_0 + \chi_1 a_1 + \hat{y}\}$
2	$\{(y = A_L x_T + \hat{y}) \wedge P_{cons}\}$
	endwhile
2,3	$\{((y = A_L x_T + \hat{y}) \wedge P_{cons}) \wedge \neg (m(x_T) < m(x))\}$
1b	$\{y = Ax + \hat{y}\}$

Figure 3.4: Annotated algorithm for computing $y := Ax + y$ (Variant 3).

3.4 Rank-1 Update

Consider the vectors $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$, and the matrix $A \in \mathbb{R}^{m\times n}$ partitioned as in Section 3.1. A second operation that plays a critical role in linear algebra is the *rank-1 update* (GER), defined as

$$A := A + \alpha y x^{\mathrm{T}}. \tag{3.6}$$

For simplicity in this section we consider $\alpha = 1$. In this operation the (i, j) element of A is updated as $\alpha_{i,j} := \alpha_{i,j} + \psi_i \chi_j$, $0 \leq i < m$, $0 \leq j < n$.

Step	**Cost Analysis:** $y := \text{MATVEC_VAR1}(A, x, y)$
1a	$C_{sf} = 0$ flops
4	**Partition** $A \rightarrow \left(\frac{A_T}{A_B}\right)$, $y \rightarrow \left(\frac{y_T}{y_B}\right)$, $\hat{y} \rightarrow \left(\frac{\hat{y}_T}{\hat{y}_B}\right)$ **where** A_T has 0 rows, and y_T, $\hat{y}_T$ have 0 elements
2	$C_{sf} = 2m(y_T)n$ flops
3	**while** $m(y_T) < m(y)$ **do**
2,3	$C_{sf} = 2m(y_T)n$ flops
5a	**Repartition** $\left(\frac{A_T}{A_B}\right) \rightarrow \begin{pmatrix} A_0 \\ \hline a_1^T \\ \hline A_2 \end{pmatrix}$, $\left(\frac{y_T}{y_B}\right) \rightarrow \begin{pmatrix} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{pmatrix}$, $\left(\frac{\hat{y}_T}{\hat{y}_B}\right) \rightarrow \begin{pmatrix} \hat{y}_0 \\ \hline \hat{\psi}_1 \\ \hline \hat{y}_2 \end{pmatrix}$ **where** a_1^T is a row, and ψ_1, $\hat{\psi}_1$ are scalars
6	$C_{sf} = 2m(y_0)n$ flops
8	$\psi_1 = a_1^T x + \psi_1$ **Cost:** $2n$ flops
5b	**Continue with** $\left(\frac{A_T}{A_B}\right) \leftarrow \begin{pmatrix} A_0 \\ \hline a_1^T \\ \hline A_2 \end{pmatrix}$, $\left(\frac{y_T}{y_B}\right) \leftarrow \begin{pmatrix} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{pmatrix}$, $\left(\frac{\hat{y}_T}{\hat{y}_B}\right) \leftarrow \begin{pmatrix} \hat{y}_0 \\ \hline \hat{\psi}_1 \\ \hline \hat{y}_2 \end{pmatrix}$
7	$C_{sf} = 2(m(y_0) + 1)n$ flops
2	$C_{sf} = 2m(y_T)n$ flops
	endwhile
2,3	$C_{sf} = 2m(y_T)n$ flops
1b	**Total Cost:** $2mn$ flops

Figure 3.5: Cost analysis for the algorithm in Fig. 3.3.

Example 3.26 *Consider*

$$x = \begin{pmatrix} 3 \\ 0 \\ 1 \end{pmatrix}, \quad y = \begin{pmatrix} 2 \\ 5 \\ 1 \\ 8 \end{pmatrix}, \quad \textit{and} \quad A = \begin{pmatrix} 3 & 1 & 2 \\ 1 & 2 & 2 \\ 9 & 0 & 2 \\ 7 & 1 & 0 \end{pmatrix}.$$

The result of performing a GER *on A involving vectors x and y is given by:*

$$A := \begin{pmatrix} 3 & 1 & 2 \\ 1 & 2 & 2 \\ 9 & 0 & 2 \\ 7 & 1 & 0 \end{pmatrix} + \begin{pmatrix} 2 \\ 5 \\ 1 \\ 8 \end{pmatrix}(3, 0, 1) = \begin{pmatrix} 3+2\cdot3 & 1+2\cdot0 & 2+2\cdot1 \\ 1+5\cdot3 & 2+5\cdot0 & 2+5\cdot1 \\ 9+1\cdot3 & 0+1\cdot0 & 2+1\cdot1 \\ 7+8\cdot3 & 1+8\cdot0 & 0+8\cdot1 \end{pmatrix} = \begin{pmatrix} 9 & 1 & 4 \\ 16 & 2 & 7 \\ 12 & 0 & 3 \\ 31 & 1 & 8 \end{pmatrix}.$$

The term *rank-1 update* comes from the fact that the rank of the matrix yx^{T} is at most one. Indeed,

$$yx^{\mathrm{T}} = (\chi_0 y, \chi_1 y, \ldots, \chi_{n-1} y)$$

clearly shows that all columns of this matrix are multiples of the same vector y, and thus there can be at most one linearly independent column.

Now, we note that

$$\begin{aligned}
A &:= A + yx^{\mathrm{T}} \\
&= \begin{pmatrix} \alpha_{00} & \alpha_{01} & \cdots & \alpha_{0,n-1} \\ \alpha_{10} & \alpha_{11} & \cdots & \alpha_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,n-1} \end{pmatrix} + \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{m-1} \end{pmatrix} (\chi_0, \chi_1, \ldots, \chi_{n-1}) \\
&= \begin{pmatrix} \alpha_{00} + \psi_0\chi_0 & \alpha_{01} + \psi_0\chi_1 & \cdots & \alpha_{0,n-1} + \psi_0\chi_{n-1} \\ \alpha_{10} + \psi_1\chi_0 & \alpha_{11} + \psi\chi & \cdots & \alpha_{1,n-1} + \psi_1\chi_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{m-1,0} + \psi_{m-1}\chi_0 & \alpha_{m-1,1} + \psi\chi & \cdots & \alpha_{m-1,n-1} + \psi_{m-1}\chi_{n-1} \end{pmatrix} \\
&= (a_0 + \chi_0 y, a_1 + \chi_1 y, \ldots, a_{n-1} + \chi_{n-1} y) = \begin{pmatrix} \check{a}_0^{\mathrm{T}} + \psi_0 x^{\mathrm{T}} \\ \check{a}_1^{\mathrm{T}} + \psi_1 x^{\mathrm{T}} \\ \vdots \\ \check{a}_{m-1}^{\mathrm{T}} + \psi_{m-1} x^{\mathrm{T}} \end{pmatrix},
\end{aligned}$$

which shows that, in the computation of $A + yx^{\mathrm{T}}$, column a_j, $0 \le j < n$, is replaced by $a_j + \chi_j y$ while row $\check{a}_i^{\mathrm{T}}$, $0 \le i < m$, is replaced by $\check{a}_i^{\mathrm{T}} + \psi_i x^{\mathrm{T}}$.

Based on the above observations the next two corollaries give the PMEs that can be used to derive the algorithms for computing GER.

Corollary 3.27 *Partition matrix A and vector x as*

$$A \rightarrow \left(A_L \,\middle|\, A_R \right) \quad \textit{and} \quad x \rightarrow \left(\frac{x_T}{x_B} \right),$$

with $n(A_L) = m(x_T)$. Then

$$A + yx^{\mathrm{T}} = \left(A_L \,\middle|\, A_R \right) + y \left(\frac{x_T}{x_B} \right)^{\mathrm{T}} = \left(A_L + yx_T^{\mathrm{T}} \,\middle|\, A_R + yx_B^{\mathrm{T}} \right).$$

Corollary 3.28 *Partition matrix A and vector y as*

$$A \rightarrow \left(\frac{A_T}{A_B} \right) \quad \textit{and} \quad y \rightarrow \left(\frac{y_T}{y_B} \right),$$

such that $m(A_T) = m(y_T)$. *Then*

$$A + yx^{\mathrm{T}} = \left(\frac{A_T}{A_B}\right) + \left(\frac{y_T}{y_B}\right) x^{\mathrm{T}} = \left(\frac{A_T + y_T x^{\mathrm{T}}}{A_B + y_B x^{\mathrm{T}}}\right).$$

Remark 3.29 *Corollaries 3.27 and 3.28 again pose restrictions on the dimensions of the partitioned matrices/vectors so that an operation is "consistently" defined for these partitioned elements.*
We now give a few rules that apply to the partitionings performed on the operands that arise in GER. *Consider two matrices* A, B, *and an operation which relates them as:*

1. $A + B$ *or* $A := B$; *then,* $m(A) = m(B)$, $n(A) = n(B)$ *and any partitioning by rows/columns in one of the two operands must be done conformally (with the same dimensions) in the other operand.*

Consider now a matrix A, *two vectors* x, y *and the* GER *operation*

1. $A + yx^{\mathrm{T}}$; *then,* $m(A) = m(y)$, $n(A) = m(x)$, *and any partitioning by rows/columns in* A *must be conformally performed by elements in* y/x *(and vice versa).*

Exercise 3.30 *Prove Corollary 3.27.*

Exercise 3.31 *Prove Corollary 3.28.*

Exercise 3.32 *Derive two different algorithms for* GER *using the partitionings*

$$A \rightarrow \left(A_L \,\middle|\, A_R \right), \quad x \rightarrow \left(\frac{x_T}{x_B}\right).$$

Exercise 3.33 *Derive two different algorithms for* GER *using the partitionings*

$$A \rightarrow \left(\frac{A_T}{A_B}\right), \quad y \rightarrow \left(\frac{y_T}{y_B}\right).$$

Exercise 3.34 *Prove that all of the previous four algorithms for* GER *incur in* $2mn$ *flops.*

3.5 Solving Triangular Linear Systems of Equations

Consider the *linear system* with n unknowns and m equations defined by

$$\begin{array}{cccc} \alpha_{00}\chi_0 + \alpha_{01}\chi_1 + & \dots & +\alpha_{0,n}\chi_n & = & \beta_0, \\ \alpha_{10}\chi_0 + \alpha_{11}\chi_1 + & \dots & +\alpha_{1,n}\chi_n & = & \beta_1, \\ \vdots & \ddots & \vdots & = & \vdots \\ \alpha_{m-1,0}\chi_0 + \alpha_{m-1,1}\chi_1 + & \dots & +\alpha_{m-1,n}\chi_n & = & \beta_m - 1. \end{array}$$

This system of equations can also be expressed in matrix form as:

$$\begin{pmatrix} \alpha_{00} & \alpha_{01} & \cdots & \alpha_{0,n-1} \\ \alpha_{10} & \alpha_{11} & \cdots & \alpha_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,n-1} \end{pmatrix} \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{n-1} \end{pmatrix} = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_{m-1} \end{pmatrix} \equiv Ax = b.$$

Here, $A \in \mathbb{R}^{m \times n}$ is the *coefficient matrix*, $b \in \mathbb{R}^m$ is the *right-hand side vector*, and $x \in \mathbb{R}^n$ is the *vector of unknowns*.

Let us now define the *diagonal* elements of the matrix A as those elements of the form $\alpha_{i,i}$, $0 \leq i < \min(m, n)$. In this section we study a simple case of a linear system which appears when the coefficient matrix is square and has zeros in all its elements above the diagonal; we then say that the coefficient matrix is *lower triangular* and we prefer to denote it using L instead of A, where L stands for Lower:

$$\begin{pmatrix} \lambda_{00} & 0 & \cdots & 0 \\ \lambda_{10} & \lambda_{11} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \lambda_{n-1,0} & \lambda_{n-1,0} & \cdots & \lambda_{n-1,n-1} \end{pmatrix} \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{n-1} \end{pmatrix} = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_{n-1} \end{pmatrix} \equiv Lx = b.$$

This *triangular linear system of equations* has a solution if $\lambda_{i,i} \neq 0$, $0 \leq i < n$.

An analogous case occurs when the coefficient matrix is upper triangular, that is, all the elements below the diagonal of the matrix are zero.

Remark 3.35 *Lower/upper triangular matrices will be denoted by letters such as L/U for Lower/Upper. Lower/upper triangular matrices are square.*

We next proceed to derive algorithms for computing this operation (hereafter, TRSV) by filling out the worksheet in Figure 2.5. During the presentation one should think of x as the vector that represents the final solution, which ultimately will overwrite b upon completion of the loop.

Remark 3.36 *In order to emphasize that the methodology allows one to derive algorithms for a given linear algebra operation without an* a priori *knowledge of a method, we directly proceed with the derivation of an algorithm for the solution of triangular linear systems, and delay the presentation of a concrete example until the end of this section.*

Step 1: Specifying the precondition and postcondition. The precondition for the algorithm is given by

$$P_{pre} : (L \in \mathbb{R}^{n \times n}) \wedge \mathrm{TrLw}(L) \wedge (x, b \in \mathbb{R}^n).$$

Step	**Annotated Algorithm:** $(b := x) \wedge (Lx = b)$
1a	$\{(L \in \mathbb{R}^{n \times n}) \wedge \mathrm{TrLw}(L) \wedge (x, b \in \mathbb{R}^n)\}$
4	**Partition** $L \rightarrow \left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)$, $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$, $b \rightarrow \left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right)$ **where** L_{TL} is 0×0, and x_T, b_T have 0 elements
2	$\left\{\left(\left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \hat{b}_B - L_{BL} x_T \end{array}\right)\right) \wedge (L_{TL} x_T = \hat{b}_T)\right\}$
3	**while** $m(b_T) < m(b)$ **do**
2,3	$\left\{\left(\left(\left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \hat{b}_B - L_{BL} x_T \end{array}\right)\right) \wedge (L_{TL} x_T = \hat{b}_T)\right) \wedge (m(b_T) < m(b))\right\}$
5a	**Repartition** $\left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c\|c\|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right)$, $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{array}\right)$, $\left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) \rightarrow \left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline b_2 \end{array}\right)$ **where** λ_{11}, χ_1, and β_1 are scalars
6	$\left\{\left(\left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline b_2 \end{array}\right) = \left(\begin{array}{c} x_0 \\ \hline \hat{\beta}_1 - l_{10}^T x_0 \\ \hline \hat{b}_2 - L_{20} x_0 \end{array}\right)\right) \wedge (L_{00} x_0 = \hat{b}_0)\right\}$
8	$\chi_1 := \beta_1 / \lambda_{11}$ $b_2 := b_2 - \chi_1 l_{21}$ (AXPY)
5b	**Continue with** $\left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c\|c\|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right)$, $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{array}\right)$, $\left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) \leftarrow \left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline b_2 \end{array}\right)$
7	$\left\{\left(\left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline b_2 \end{array}\right) = \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline \hat{b}_2 - L_{20} x_0 - \chi_1 l_{21} \end{array}\right)\right) \wedge \left(\begin{array}{c} L_{00} x_0 = \hat{b}_0 \\ \hline l_{10}^T x_0 + \lambda_{11} \chi_1 = \hat{\beta}_1 \end{array}\right)\right\}$
2	$\left\{\left(\left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \hat{b}_B - L_{BL} x_T \end{array}\right)\right) \wedge (L_{TL} x_T = \hat{b}_T)\right\}$
	endwhile
2,3	$\left\{\left(\left(\left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \hat{b}_B - L_{BL} x_T \end{array}\right)\right) \wedge (L_{TL} x_T = \hat{b}_T)\right) \wedge \neg (m(b_T) < m(b))\right\}$
1b	$\left\{(b = x) \wedge (Lx = \hat{b})\right\}$

Figure 3.6: Annotated algorithm for solving $Lx = b$ (unblocked Variant 2).

Here, the predicate $\mathrm{TrLw}(L)$ is true if L is a lower triangular matrix. (A similar predicate, $\mathrm{TrUp}(U)$, will play an analogous role for upper triangular matrices.) The postcondition is that

$$P_{post} : (b = x) \wedge (Lx = \hat{b});$$

in other words, upon completion the contents of b equal those of x, where x is the solution of the lower triangular linear system $Lx = \hat{b}$. This is indicated in Steps 1a and 1b in Figure 3.6.

Next, let us use L to introduce a new type of partitioning, into quadrants:

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

Since upon termination $Lx = \hat{b}$, vectors x and b must be partitioned consistently as

$$x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array} \right), \quad b \rightarrow \left(\begin{array}{c} b_T \\ \hline b_B \end{array} \right),$$

where "$P_{cons} : n(L_{TL}) = m(x_T) = m(b_T)$" holds. Furthermore, we will require that both L_{TL} and L_{BR} are themselves lower triangular matrices, that is,

$$P_{struct} : \mathrm{TrLw}(L_{TL}) \wedge (m(L_{TL}) = n(L_{TL})) \wedge \mathrm{TrLw}(L_{BR}) \wedge (m(L_{BR}) = n(L_{BR}))$$

is true. Now, as L is square lower triangular, it is actually sufficient to require that

$$P_{struct} : m(L_{TL}) = n(L_{TL}).$$

holds.

Remark 3.37 *We will often use the structural predicate "P_{struct}" to establish conditions on the structure of the exposed blocks.*

Remark 3.38 *When dealing with triangular matrices, in order for the diagonal blocks (submatrices) that are exposed to themselves be triangular, we always partition this type of matrices into* quadrants*, with square blocks on the diagonal.*

Although we employ predicates P_{cons} and P_{struct} during the derivation of the algorithm, in order to condense the assertions for this algorithm, we do not include these two predicates as part of the invariant in the presentation of the corresponding worksheet.

Variant 1
$\left(\left(\left(\frac{b_T}{b_B}\right) = \left(\frac{x_T}{\hat{b}_B}\right),\right) \wedge (L_{TL}x_T = \hat{b}_T)\right) \wedge P_{cons} \wedge P_{struct}$
Variant 2
$\left(\left(\left(\frac{b_T}{b_B}\right) = \left(\frac{x_T}{\hat{b}_B - L_{BL}x_T}\right),\right) \wedge (L_{TL}x_T = \hat{b}_T)\right) \wedge P_{cons} \wedge P_{struct}$

Figure 3.7: Two loop-invariants for solving $Lx = b$, overwriting b with x.

Step 2: Determining loop-invariants. The PME is given by

$$\left(\left(\frac{b_T}{b_B}\right) = \left(\frac{x_T}{x_B}\right)\right) \wedge \left(\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)\left(\frac{x_T}{x_B}\right) = \left(\frac{\hat{b}_T}{\hat{b}_B}\right)\right)$$

or, by Theorem 3.12,

$$\left(\left(\frac{b_T}{b_B}\right) = \left(\frac{x_T}{x_B}\right)\right) \wedge \left(\frac{L_{TL}x_T = \hat{b}_T}{L_{BL}x_T + L_{BR}x_B = \hat{b}_B}\right),$$

which is finally equivalent to

$$\left(\left(\frac{b_T}{b_B}\right) = \left(\frac{x_T}{x_B}\right)\right) \wedge \left(\frac{L_{TL}x_T = \hat{b}_T}{L_{BR}x_B = \hat{b}_B - L_{BL}x_T}\right).$$

This shows that x_T can be computed from the first equality (the one at the top), after which $\hat{b}_B$ must be updated by subtracting $L_{BL}x_T$ from it, before x_B can be computed using the second equality. This constraint on the order in which subresults must be computed yields the two loop-invariants in Figure 3.7.

Step 3: Choosing a Loop-guard. For either of the two loop-invariants, the loop-guard "$G : m(b_T) < m(b)$" has the property that $(P_{inv} \wedge \neg G) \Rightarrow P_{post}$.

Step 4: Initialization. For either of the two loop-invariants, the initialization

$$L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right), \quad x \rightarrow \left(\frac{x_T}{x_B}\right), \quad \text{and} \quad b \rightarrow \left(\frac{b_T}{b_B}\right),$$

where L_{TL} is 0×0, and x_T, b_T have 0 elements, has the property that it sets the variables in a state where the loop-invariant holds.

Step 5: Progressing through the operands. For either of the two loop-invariants, the repartitioning shown in Step 5a in Figure 3.6[3], followed by moving the thick lines as in Step 5b in the same figure denote that progress is made through the operands so that the loop eventually terminates. It also ensures that P_{cons} and P_{struct} hold.

Only now does the derivation become dependent on the loop-invariant that we choose. Let us choose Invariant 2, which will produce the algorithm identified as Variant 2 for this operation.

Step 6: Determining the state after repartitioning. Invariant 2 and the repartitioning of the partitioned matrix and vectors imply that

$$\left(\left(\begin{array}{c} b_0 \\ \hline \left(\begin{array}{c} \beta_1 \\ \hline b_2 \end{array}\right)\end{array}\right) = \left(\begin{array}{c} x_0 \\ \hline \left(\begin{array}{c} \hat{\beta}_1 \\ \hline \hat{b}_2 \end{array}\right) - \left(\begin{array}{c} l_{10}^T \\ \hline L_{20} \end{array}\right) x_0 \end{array}\right)\right) \wedge (L_{00}x_0 = \hat{b}_0)$$

$$\equiv \left(\left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline b_2 \end{array}\right) = \left(\begin{array}{c} x_0 \\ \hline \hat{\beta}_1 - l_{10}^T x_0 \\ \hline \hat{b}_2 - L_{20}x_0 \end{array}\right)\right) \wedge (L_{00}x_0 = \hat{b}_0),$$

which is entered in Step 6 as in Figure 3.6.

Step 7: Determining the state after moving the thick lines. In Step 5b, Invariant 2 and the moving of the thick lines imply that

$$\left(\left(\begin{array}{c} \left(\begin{array}{c} b_0 \\ \hline \beta_1 \end{array}\right) \\ \hline b_2 \end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c} x_0 \\ \hline \chi_1 \end{array}\right) \\ \hline \hat{b}_2 - \left(\begin{array}{c|c} L_{20} & l_{21} \end{array}\right)\left(\begin{array}{c} x_0 \\ \hline \chi_1 \end{array}\right) \end{array}\right)\right) \wedge \left(\left(\begin{array}{c|c} L_{00} & 0 \\ \hline l_{10}^T & \lambda_{11} \end{array}\right)\left(\begin{array}{c} x_0 \\ \hline \chi_1 \end{array}\right) = \left(\begin{array}{c} \hat{b}_0 \\ \hline \hat{\beta}_1 \end{array}\right)\right)$$

$$\equiv \left(\left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline b_2 \end{array}\right) = \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline \hat{b}_2 - L_{20}x_0 - \chi_1 l_{21} \end{array}\right)\right) \wedge \left(\left(\begin{array}{c} L_{00}x_0 = \hat{b}_0 \\ \hline l_{10}^T x_0 + \lambda_{11}\chi_1 = \hat{\beta}_1 \end{array}\right)\right),$$

which is entered in the corresponding step as in Figure 3.6.

Step 8. Determining the update. Comparing the contents in Step 6 and Step 7 now tells us that the contents of b must be updated from

$$\left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline b_2 \end{array}\right) = \left(\begin{array}{c} x_0 \\ \hline \hat{\beta}_1 - l_{10}^T x_0 \\ \hline \hat{b}_2 - L_{20}x_0 \end{array}\right) \quad \text{to} \quad \left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline b_2 \end{array}\right) = \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline \hat{b}_2 - L_{20}x_0 - \chi_1 l_{21} \end{array}\right),$$

[3] In the repartitioning of L the superscript "T" denotes that l_{01}^T is a row vector as corresponds to λ_{11} being a scalar.

where

$$L_{00}x_0 = \hat{b}_0 \quad \text{and}$$
$$l_{10}^{\mathrm{T}}x_0 + \lambda_{11}\chi_1 = \hat{\beta}_1.$$

Manipulating the last equation yields that $\chi_1 = (\hat{\beta}_1 - l_{10}^{\mathrm{T}}x_0)/\lambda_{11}$. Since β_1 already contains $(\hat{\beta}_1 - l_{10}^{\mathrm{T}}x_0)$ we conclude that the update to be performed is given by

$$\begin{array}{rcl} \chi_1 & := & \beta_1/\lambda_{11} \quad \text{and} \\ b_2 & := & b_2 - \chi_1 l_{21}, \end{array}$$

which is entered in the corresponding step as in Figure 3.6.

Final algorithm. By deleting the temporary variable x, which is only used for the purpose of proving the algorithm correct while it is constructed, we arrive at the algorithm in Figure 3.8. In Section 4.2, we discuss an API for representing algorithms in Matlab M-script code, FLAME@LAB. The FLAME@LAB code for the algorithm in Figure 3.8 is given in Figure 3.9.

Example 3.39 *Let us now illustrate how this algorithm proceeds. Consider a triangular linear system defined by*

$$L = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 2 & 1 & 2 & 0 \\ 0 & 2 & 1 & 3 \end{pmatrix}, \quad b = \begin{pmatrix} 2 \\ 3 \\ 10 \\ 19 \end{pmatrix}.$$

From a little manipulation we can see that the solution to this system is given by

$$\begin{array}{rclcl} \chi_0 & := & (\ 2 \ )/2 & = & 1, \\ \chi_1 & := & (\ 3 - 1 \cdot 1 \ )/1 & = & 2, \\ \chi_2 & := & (\ 10 - 2 \cdot 1 - 1 \cdot 2 \ )/2 & = & 3, \\ \chi_3 & := & (\ 19 - 0 \cdot 1 - 2 \cdot 2 - 1 \cdot 3 \ )/3 & = & 4. \end{array}$$

In Figure 3.10 we show the initial contents of each quadrant (iteration labeled as 0) as well as the contents as computation proceeds from the first to the fourth (and final) iteration. In the figure, faces of normal size indicate data and operations/results that have already been performed/computed, while the small faces indicate operations that have yet to be performed.

The way the solver classified as Variant 2 works, corresponds to what is called an "eager" algorithm, in the sense that once an unknown is computed, it is immediately "eliminated" from the remaining equations. Sometimes this algorithm is also classified as the "column-oriented" algorithm of forward substitution as, at each iteration, it utilizes a column of L in the update of the remaining independent terms by using a saxpy operation. It is sometimes called *forward substitution* for reasons that will become clear in Chapter 6.

Exercise 3.40 *Prove that the cost of the triangular linear system solver formulated in Figure 3.8 is* $n^2 + n \approx n^2$ *flops.* **Hint:** *Use* $C_{sf} = m(x_0) + \sum_{k=0}^{m(x_0)-1} 2(n-k-1)$ *flops.*

Algorithm: $b := \text{TRSV_VAR2}(L, b)$

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

  where L_{TL} is 0×0 and b_T has 0 elements

while $m(b_T) < m(b)$ **do**

  Repartition

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right), \left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) \rightarrow \left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline b_2 \end{array}\right)$$

  where λ_{11} and β_1 are scalars

  $\beta_1 := \beta_1 / \lambda_{11}$

  $b_2 := b_2 - \beta_1 l_{21}$ (AXPY)

  Continue with

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right), \left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) \leftarrow \left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline b_2 \end{array}\right)$$

endwhile

Figure 3.8: Algorithm for solving $Lx = b$ (unblocked Variant 2).

Remark 3.41 *When dealing with cost expression we will generally neglect lower order terms.*

Exercise 3.42 *Derive an algorithm for solving $Lx = b$ by choosing the Invariant 1 in Figure 3.7. The solution to this exercise corresponds to an algorithm that is "lazy" (for each equation, it does not eliminate previous unknown until it becomes necessary) or row-oriented (accesses to L are by rows, in the form of* APDOT*s).*

Exercise 3.43 *Prove that the cost of the triangular linear system solver for the lazy algorithm obtained as the solution to Exercise 3.42 is n^2 flops.*

Exercise 3.44 *Derive algorithms for the solution of the following triangular linear systems:*

1. $Ux = b$.
2. $L^T x = b$.
3. $U^T x = b$.

Here $L, U \in \mathbb{R}^{n \times n}$ are lower and upper triangular, respectively, and $x, b \in \mathbb{R}^n$.

```
 1  function [ b_out ] = Trsv_lower_unb_var2( L, b )
 2
 3    [ LTL, LTR, ...
 4      LBL, LBR ] = FLA_Part_2x2( L,      0, 0, 'FLA_TL' );
 5    [ bT, ...
 6      bB ] = FLA_Part_2x1( b,      0, 'FLA_TOP' );
 7
 8    while ( size( LTL, 1 ) < size( L, 1 ) )
 9      [ L00,  l01,      L02,  ...
10        l10t, lambda11, l12t, ...
11        L20,  l21,      L22 ] = FLA_Repart_2x2_to_3x3( LTL, LTR, ...
12                                                        LBL, LBR, 1, 1, 'FLA_BR' );
13      [ b0, ...
14        beta1, ...
15        b2 ] = FLA_Repart_2x1_to_3x1( bT, ...
16                                      bB,      1, 'FLA_BOTTOM' );
17      %------------------------------------------------------------%
18      beta1 = beta1 / lambda1;
19      b2    = b2 - beta1 * l21;
20      %------------------------------------------------------------%
21      [ LTL, LTR, ...
22        LBL, LBR ] = FLA_Cont_with_3x3_to_2x2( L00,  l01,      L02,  ...
23                                               l10t, lambda11, l12t, ...
24                                               L20,  l21,      L22,      'FLA_TL' );
25      [ bT, ...
26        bB ] = FLA_Cont_with_3x1_to_2x1( b0, ...
27                                         beta1, ...
28                                         b2,     'FLA_TOP' );
29    end
30    b_out = [ bT
31              bB ];
32  return
```

Figure 3.9: FLAME@LAB code for solving $Lx = b$, overwriting b with x (unblocked Variant 2).

3.6 Blocked Algorithms

Key objectives when designing and implementing linear algebra libraries are modularity and performance. In this section we show how both can be accommodated by casting algorithms in terms of GEMV. The idea is to derive so-called *blocked algorithms* which differ from the algorithms derived so far in that they move the thick lines more than one element, row, and/or column at a time. We illustrate this technique by revisiting the TRSV operation.

#Iter.	$\left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)$	$\left(\begin{array}{c} L_{TL}^{-1} b_T \\ \hline b_B - L_{BL}(L_{TL}^{-1} b_T) \end{array}\right)$	$=$	$\left(\begin{array}{c} x_T \\ \hline b_B - L_{BL} x_T \end{array}\right)$
0	2 0 0 0	(2)/2		2
	1 1 0 0	(3 − 1 · 1)/2	=	3
	2 1 2 0	(10 − 2 · 1 − 1 · 2)/2		10
	0 2 1 3	(19 − 0 · 1 − 2 · 2 − 1 · 3)/3		19
1	2 0 0 0	(2)/2		1
	1 1 0 0	(3 - 1 · 1)/1	=	2
	2 1 2 0	(10 - 2 · 1 − 1 · 2)/2		8
	0 2 1 3	(19 - 0 · 1 − 2 · 2 − 1 · 3)/3		19
2	2 0 0 0	(2)/2		1
	1 1 0 0	(3 - 1 · 1)/1	=	2
	2 1 2 0	(10 - 2 · 1 - 1 · 2)/2		6
	0 2 1 3	(19 - 0 · 1 - 2 · 2 − 1 · 3)/3		15
3	2 0 0 0	(2)/2		1
	1 1 0 0	(3 - 1 · 1)/1	=	2
	2 1 2 0	(10 - 2 · 1 - 1 · 2)/2		3
	0 2 1 3	(19 - 0 · 1 - 2 · 2 - 1 · 3)/3		12
4	2 0 0 0	(2)/2		1
	1 1 0 0	(3 - 1 · 1)/1	=	2
	2 1 2 0	(10 - 2 · 1 - 1 · 2)/2		3
	0 2 1 3	(19 - 0 · 1 - 2 · 2 - 1 · 3)/3		4

Figure 3.10: Example of the computation of $(b := x) \wedge (Lx = b)$ (Variant 2). Computations yet to be performed are in tiny font.

Remark 3.45 *The derivation of blocked algorithms is identical to that of unblocked algorithm up to and including Step 4.*

Let us choose Invariant 2, which will produce now the worksheet of a blocked algorithm identified as Variant 2 in Figure 3.11.

Step 5: Progressing through the operands. We now choose to move through vectors x and b by n_b elements per iteration. Here n_b is the (ideal) *block size* of the algorithm. In other words, at each iteration of the loop, n_b elements are taken from x_B and b_B and moved to x_T, b_T, respectively. For consistency then, a block of dimension $n_b \times n_b$ must be also moved from L_{BR} to L_{TL}. We can proceed in this manner by first repartitioning

<table>
<tr><th>Step</th><th>Annotated Algorithm: $(b := x) \wedge (Lx = b)$</th></tr>
<tr><td>1a</td><td>$\{(L \in \mathbb{R}^{n \times n}) \wedge \mathrm{TrLw}(L) \wedge (x, b \in \mathbb{R}^n)\}$</td></tr>
<tr><td>4</td><td>Partition $L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)$, $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$, $b \rightarrow \left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right)$
where L_{TL} is 0×0, and x_T, b_T have 0 elements</td></tr>
<tr><td>2</td><td>$\left\{\left(\left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \hat{b}_B - L_{BL}x_T \end{array}\right)\right) \wedge (L_{TL}x_T = \hat{b}_T)\right\}$</td></tr>
<tr><td>3</td><td>while $m(b_T) < m(b)$ do</td></tr>
<tr><td>2,3</td><td>$\left\{\left(\left(\left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \hat{b}_B - L_{BL}x_T \end{array}\right)\right) \wedge (L_{TL}x_T = \hat{b}_T)\right) \wedge (m(b_T) < m(b))\right\}$</td></tr>
<tr><td>5a</td><td>Determine block size n_b
Repartition
$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$, $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \end{array}\right)$, $\left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) \rightarrow \left(\begin{array}{c} b_0 \\ \hline b_1 \\ \hline b_2 \end{array}\right)$
where L_{11} is $n_b \times n_b$, and x_1, b_1 have n_b elements</td></tr>
<tr><td>6</td><td>$\left\{\left(\left(\begin{array}{c} b_0 \\ \hline b_1 \\ \hline b_2 \end{array}\right) = \left(\begin{array}{c} x_0 \\ \hline \hat{b}_1 - L_{10}x_0 \\ \hline \hat{b}_2 - L_{20}x_0 \end{array}\right)\right) \wedge (L_{00}x_0 = \hat{b}_0)\right\}$</td></tr>
<tr><td>8</td><td>$x_1 := \textsc{trsv}(L_1, b_1)$
$b_2 := b_2 - L_{21}x_1$ ($\textsc{gemv}$)</td></tr>
<tr><td>5b</td><td>Continue with
$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$, $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \end{array}\right)$, $\left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) \leftarrow \left(\begin{array}{c} b_0 \\ \hline b_1 \\ \hline b_2 \end{array}\right)$</td></tr>
<tr><td>7</td><td>$\left\{\left(\left(\begin{array}{c} b_0 \\ \hline b_1 \\ \hline b_2 \end{array}\right) = \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline \hat{b}_2 - L_{20}x_0 - L_{21}x_1 \end{array}\right)\right) \wedge \left(\begin{array}{c} L_{00}x_0 = \hat{b}_0 \\ \hline L_{10}x_0 + L_{11}x_1 = \hat{b}_1 \end{array}\right)\right\}$</td></tr>
<tr><td>2</td><td>$\left\{\left(\left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \hat{b}_B - L_{BL}x_T \end{array}\right)\right) \wedge (L_{TL}x_T = \hat{b}_T)\right\}$</td></tr>
<tr><td></td><td>endwhile</td></tr>
<tr><td>2,3</td><td>$\left\{\left(\left(\left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) = \left(\begin{array}{c} x_T \\ \hline \hat{b}_B - L_{BL}x_T \end{array}\right)\right) \wedge (L_{TL}x_T = \hat{b}_T)\right) \wedge \neg(m(b_T) < m(b))\right\}$</td></tr>
<tr><td>1b</td><td>$\{(b = x) \wedge (Lx = \hat{b})\}$</td></tr>
</table>

Figure 3.11: Annotated algorithm for solving $Lx = b$ (blocked Variant 2).

the matrix and the vectors as:

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \quad \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \end{array}\right), \quad \left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) \rightarrow \left(\begin{array}{c} b_0 \\ \hline b_1 \\ \hline b_2 \end{array}\right),$$

where L_{11} is a block of dimension $n_b \times n_b$, and x_1, b_1 have n_b elements each. These block/elements are then moved to the corresponding parts of the matrix/vectors as indicated by

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \end{array}\right), \left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) \leftarrow \left(\begin{array}{c} b_0 \\ \hline b_1 \\ \hline b_2 \end{array}\right).$$

This movement ensures that the loop eventually terminates and that both P_{cons} and P_{struct} hold.

Remark 3.46 *In practice, the block size is adjusted at each iteration as the minimum between the* algorithmic (or optimal) block size *and the number of remaining elements.*

Step 6: Determining the state after repartitioning. Invariant 2 and the definition of the repartitioning for the blocked algorithm imply that

$$\left(\left(\begin{array}{c} b_0 \\ \hline \left(\begin{array}{c} b_1 \\ \hline b_2 \end{array}\right) \end{array}\right) = \left(\begin{array}{c} x_0 \\ \hline \left(\begin{array}{c} \hat{b}_1 \\ \hline \hat{b}_2 \end{array}\right) - \left(\begin{array}{c} L_{10} \\ \hline L_{20} \end{array}\right) x_0 \end{array}\right)\right) \wedge (L_{00}x_0 = \hat{b}_0),$$

or

$$\left(\left(\begin{array}{c} b_0 \\ \hline b_1 \\ \hline b_2 \end{array}\right) = \left(\begin{array}{c} x_0 \\ \hline \hat{b}_1 - L_{10}x_0 \\ \hline \hat{b}_2 - L_{20}x_0 \end{array}\right)\right) \wedge (L_{00}x_0 = \hat{b}_0),$$

which is entered in Step 6 as in Figure 3.11.

Step 7: Determining the state after moving the thick lines. In Step 5b the Invariant 2 implies that

$$\left(\left(\begin{array}{c} \left(\begin{array}{c} b_0 \\ \hline b_1 \end{array}\right) \\ \hline b_2 \end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{c} x_0 \\ \hline x_1 \end{array}\right) \\ \hline \hat{b}_2 - \left(\begin{array}{c|c} L_{20} & L_{21} \end{array}\right) \left(\begin{array}{c} x_0 \\ \hline x_1 \end{array}\right) \end{array}\right)\right) \wedge \left(\left(\begin{array}{c|c} L_{00} & 0 \\ \hline L_{10} & L_{11} \end{array}\right) \left(\begin{array}{c} x_0 \\ \hline x_1 \end{array}\right) = \left(\begin{array}{c} \hat{b}_0 \\ \hline \hat{b}_1 \end{array}\right)\right),$$

or

$$\left(\left(\begin{array}{c} b_0 \\ \hline b_1 \\ \hline b_2 \end{array} \right) = \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline \hat{b}_2 - L_{20}x_0 - L_{21}x_1 \end{array} \right) \right) \wedge \left(\left(\begin{array}{c} L_{00}x_0 = \hat{b}_0 \\ \hline L_{10}x_0 + L_{11}x_1 = \hat{b}_1 \end{array} \right) \right),$$

which is entered in the corresponding step as in Figure 3.11.

Step 8. Determining the update. Comparing the contents in Step 6 and Step 7 now tells us that the contents of b must be updated from

$$\left(\begin{array}{c} b_0 \\ \hline b_1 \\ \hline b_2 \end{array} \right) = \left(\begin{array}{c} x_0 \\ \hline \hat{b}_1 - L_{10}x_0 \\ \hline \hat{b}_2 - L_{20}x_0 \end{array} \right) \quad \text{to} \quad \left(\begin{array}{c} b_0 \\ \hline b_1 \\ \hline b_2 \end{array} \right) = \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline \hat{b}_2 - L_{20}x_0 - L_{21}x_1 \end{array} \right),$$

where

$$\begin{array}{rcl} L_{00}x_0 & = & \hat{b}_0 \quad \text{and} \\ L_{10}x_0 + L_{11}x_1 & = & \hat{b}_1. \end{array}$$

From the the last equation we find that $L_{11}x_1 = (\hat{b}_1 - L_{10}x_0)$. Since b_1 already contains $(\hat{b}_1 - L_{10}x_0)$ we conclude that in the update we first need to solve the triangular linear system

$$L_{11}x_1 = b_1, \tag{3.7}$$

and then perform the update

$$b_2 := b_2 - L_{21}x_1. \tag{3.8}$$

This is entered in the corresponding step as in Figure 3.11. Now, solving the triangular linear system in (3.7) can be accomplished by using any unblocked algorithm or by a *recursive* call to the same routine, but with a smaller block size. We note that the operation that appears in (3.8) is a GEMV.

Final algorithm. By deleting the assertions and the temporary variable x, we obtain the blocked algorithm in Figure 3.12. If n_b is set to 1 in this algorithm, then it performs exactly the same operations and in the same order as the corresponding unblocked algorithm.

3.6.1 Cost analysis

Rather than proving a given cost expression for an algorithm, one is usually confronted with the problem of having to obtain such a formulae. We therefore shift the purpose of our cost analysis to match the more common exercise of having to obtain the cost of a (blocked) algorithm. In order to do so, we will generally assume that the blocksize is an exact multiple of the problem size.

Algorithm: $b := \text{TRSV_BLK_VAR2}(L, b)$

Partition $L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)$, $b \rightarrow \left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right)$
  where L_{TL} is 0×0 and b_T has 0 elements

while $m(b_T) < m(b)$ **do**
  Determine block size n_b
  Repartition

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) \rightarrow \left(\begin{array}{c} b_0 \\ \hline b_1 \\ \hline b_2 \end{array}\right)$$

  where L_{11} is $n_b \times n_b$ and b_1 has n_b elements

  $b_1 := \text{TRSV}(L_{11}, b_1)$
  $b_2 := b_2 - L_{21} b_1$ (GEMV)

  Continue with

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) \leftarrow \left(\begin{array}{c} b_0 \\ \hline b_1 \\ \hline b_2 \end{array}\right)$$

endwhile

Figure 3.12: Algorithm for solving $Lx = b$ (blocked Variant 2).

For the blocked algorithm for TRSV in Figure 3.12, we consider that $n = \nu n_b$ with $n_b \ll n$. The algorithm thus iterates ν times, with a triangular linear system of fixed order n_b ($L_{11}^{-1} b_1$) being solved and a GEMV operation of decreasing size ($b_1 := b_1 - L_{21} b_1$) being performed at each iteration. As a matter of fact, the row dimension of the matrix involved in the GEMV operation decreases by n_b rows per iteration so that, at iteration k, L_{21} is of dimension $(\nu - k - 1)n_b \times n_b$. Thus, the cost of solving the triangular linear system using the blocked algorithm is approximately

$$\sum_{k=0}^{\nu-1} (n_b^2 + 2(\nu - k - 1)n_b^2) \approx 2n_b^2 \left(\nu^2 - \frac{\nu^2}{2}\right) = n^2 \text{ flops.}$$

The cost of blocked variant of TRSV is equal to that of the unblocked version. This is true for most of the blocked algorithms that we will derive in this book. Nevertheless, be aware that there exist a class of blocked algorithms, related to the computation of orthogonal factorizations, which do not satisfy this property.

Exercise 3.47 *Show that all unblocked and blocked algorithms for computing the solution to $Lx = B$ have exactly the same operation count by performing an exact analysis of the operation count.*

```
1    #include "FLAME.h"
2
3   int Trsv_lower_blk_var2( FLA_Obj L, FLA_Obj b, int nb_alg )
4   {
5     FLA_Obj LTL,   LTR,      L00, L01, L02,
6             LBL,   LBR,      L10, L11, L12,
7                              L20, L21, L22;
8     FLA_Obj bT,              b0,
9             bB,              b1,
10                             b2;
11    int b;
12
13    FLA_Part_2x2( L,    &LTL, &LTR,
14                        &LBL, &LBR,     0, 0, FLA_TL );
15    FLA_Part_2x1( b,    &bT,
16                        &bB,            0, FLA_TOP );
17
18    while ( FLA_Obj_length( LTL ) < FLA_Obj_length( L ) ){
19      b = min( FLA_Obj_length( LBR ), nb_alg );
20
21      FLA_Repart_2x2_to_3x3( LTL, /**/ LTR,       &L00, /**/ &L01, &L02,
22                          /* ************* */   /* ******************** */
23                                                 &L10, /**/ &L11, &L12,
24                             LBL, /**/ LBR,       &L20, /**/ &L21, &L22,
25                             b, b, FLA_BR );
26      FLA_Repart_2x1_to_3x1( bT,                &b0,
27                          /* ** */            /* ** */
28                                                &b1,
29                             bB,                &b2,        b, FLA_BOTTOM );
30      /*------------------------------------------------------------*/
31      Trsv_lower_unb_var2( L11, b1 );       /* b1 := inv( L11 ) * b1 */
32      FLA_Gemv( FLA_NO_TRANSPOSE,           /* b2 := b2 - L21 * b1   */
33                ONE, L21, b1, ONE, b2 )
34      /*------------------------------------------------------------*/
35      FLA_Cont_with_3x3_to_2x2( &LTL, /**/ &LTR,       L00, L01, /**/ L02,
36                                                       L10, L11, /**/ L12,
37                              /* ************** */  /* ****************** */
38                                &LBL, /**/ &LBR,       L20, L21, /**/ L22,
39                                FLA_TL );
40      FLA_Cont_with_3x1_to_2x1( &bT,                b0,
41                                                    b1,
42                              /* ** */            /* ** */
43                                &bB,                b2,     FLA_TOP );
44    }
45    return FLA_SUCCESS;
46  }
```

Figure 3.13: FLAME/C code for solving $Lx = b$, overwriting b with x (Blocked Variant 2).

3.7 Summary

Let us recap the highlights of this chapter.

- Most computations in unblocked algorithms for matrix-vector operations are expressed as AXPYs and APDOTs.
- Operations involving matrices typically yield more algorithmic variants than those involving only vectors due to the fact that matrices can be traversed in multiple directions.
- Blocked algorithms for matrix-vector operations can typically be cast in terms of GEMV and/or GER. High-performance can be achieved in a modular manner by optimizing these two operations, and casting other operations in terms of them.
- The derivation of blocked algorithms is no more complex than that of unblocked algorithms.
- Algorithms for all matrix-vector operations that are discussed can be derived using the methodology presented in Chapter 2.
- Again, we note that the derivation of loop-invariants is systematic, and that the algorithm is prescribed once a loop-invariant is chosen although now a remaining choice is whether to derive an unblocked algorithm or a blocked one.

3.8 Other Matrix-Vector Operations

A number of other commonly encountered matrix-vector operations tabulated in Figure 3.14.

3.9 Further Exercises

For additional exercises, visit `$BASE/Chapter3/`.

Name	Abbrev.	Operation	Cost (flops)	Comment
GEneral Matrix Vector multiplication	GEMV	$y := \alpha Ax + \beta y$ $y := \alpha A^T x + \beta y$	$2mn$	$A \in \mathbb{R}^{m \times n}$
SYmmetric Matrix Vector multiplication	SYMV	$y := \alpha Ax + \beta y$	$2n^2$	$A \in \mathbb{R}^{n \times n}$ is symmetric and stored in lower/upper triangular part of matrix.
TRiangular Matrix Vector multiplication	TRMV	$x := Lx$ $x := L^T x$ $x := Ux$ $x := U^T x$	n^2	$L \in \mathbb{R}^{n \times n}$ is lower triangular. $U \in \mathbb{R}^{n \times n}$ is upper triangular.
GEneral Rank-1 update	GER	$A := A + \alpha yx^{\mathrm{T}}$	$2mn$	$A \in \mathbb{R}^{m \times n}$
SYmmetric Rank-1 update	SYR	$A := A + \alpha xx^{\mathrm{T}}$	n^2	$A \in \mathbb{R}^{n \times n}$ is symmetric and stored in lower/upper triangular part of matrix.
SYmmetric Rank-2 update	SYR2	$A :=$ $A + \alpha(xy^{\mathrm{T}} + yx^{\mathrm{T}})$	$2n^2$	$A \in \mathbb{R}^{n \times n}$ is symmetric and stored in lower/upper triangular part of matrix.
TRiangular Solve	TRSV	$x := L^{-1}x$ $x := L^{-T}x$ $x := U^{-1}x$ $x := U^{-T}x$	n^2	$L \in \mathbb{R}^{n \times n}$ is lower triangular. $U \in \mathbb{R}^{n \times n}$ is upper triangular. $X^{-T} = (X^{-1})^T = (X^T)^{-1}$

Figure 3.14: Basic operations combining matrices and vectors. Cost is approximate.

Chapter 4

The FLAME Application Programming Interfaces

In this chapter we present two *Application Programming Interfaces* (APIs) for coding linear algebra algorithms. While these APIs are almost trivial extensions of the M-script language and the C programming language, they greatly simplify the task of typesetting, programming, and maintaining families of algorithms for a broad spectrum of linear algebra operations. In combination with the FLAME methodology for deriving algorithms, these APIs facilitate the rapid derivation, verification, documentation, and implementation of a family of algorithms for a single linear algebra operation. Since the algorithms are expressed in code much like they are explained in a classroom setting, the APIs become not just a tool for implementing libraries, but also a valuable resource for teaching the algorithms that are incorporated in the libraries.

4.1 Example: GEMV Revisited

In order to present the FLAME APIs, we consider in this chapter again the matrix-vector product (GEMV), $y := y + Ax$, where $A \in \mathbb{R}^{m \times n}$, $y \in \mathbb{R}^m$, and $x \in \mathbb{R}^n$.

In Chapter 3 we derived unblocked algorithms for the computation of this operation. In Figure 4.1 and 4.2 we show the two blocked algorithms for computing GEMV. The first of these algorithms proceeds by blocks of rows of A, computing a new subvector of entries of y at each iteration by means of a matrix-vector product of dimension $m_b \times n$. To expose the necessary operands this algorithm requires the matrix to be partitioned by rows.

The second algorithm accesses the elements of A by blocks of columns and, at each iteration, updates y

Algorithm: $y := \text{MATVEC_BLK_VAR1}(A, x, y)$

Partition $A \rightarrow \left(\frac{A_T}{A_B}\right)$, $y \rightarrow \left(\frac{y_T}{y_B}\right)$
  where A_T has 0 rows and y_T has 0 elements
while $m(y_T) < m(y)$ **do**
  Determine block size m_b
  Repartition

$$\left(\frac{A_T}{A_B}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right), \left(\frac{y_T}{y_B}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline y_2 \end{array}\right)$$

  where A_1 has m_b rows and y_1 has m_b elements

  $y_1 = y_1 + A_1 x$

  Continue with

$$\left(\frac{A_T}{A_B}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right), \left(\frac{y_T}{y_B}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline y_2 \end{array}\right)$$

endwhile

Figure 4.1: Algorithm for computing $y := Ax + y$ (blocked Variant 1).

completely by performing a matrix-vector product of dimension $m \times n_b$. In this case, it is necessary to partition the matrix A by columns.

4.2 The FLAME@lab Interface for M-script

A popular tool for numerical computations is Matlab [22]. This application encompasses both a high-level programming language, M-script, and an interactive interface for algorithm development and numerical computations. Tools that use the same scripting language include Octaveand LabView MathScript.

In this section we describe the FLAME API for M-script: FLAME@lab, an extension consisting of nine functions that allow one to code the algorithms derived via the FLAME approach using the M-script language.

4.2.1 Horizontal partitioning

As illustrated in Figure 4.1, in stating a linear algebra algorithm one may wish to produce a 2×1 partitioning of a matrix (or vector). Given a Matlab matrix A, the following M-script function in FLAME@lab creates

Algorithm: $y := \text{MATVEC_BLK_VAR3}(A, x, y)$

Partition $A \rightarrow \left(A_L \,\|\, A_R \right)$, $x \rightarrow \left(\frac{x_T}{x_B} \right)$
  where A_L has 0 columns and x_T has 0 elements
while $m(x_T) < m(x)$ **do**
  Determine block size n_b
  Repartition

$$\left(A_L \,\|\, A_R \right) \rightarrow \left(A_0 \,\|\, A_1 \,|\, A_2 \right), \left(\frac{x_T}{x_B} \right) \rightarrow \begin{pmatrix} x_0 \\ \hline x_1 \\ \hline x_2 \end{pmatrix}$$

  where A_1 has n_b columns and x_1 has n_b elements

$y := y + A_1 x_1$

  Continue with

$$\left(A_L \,\|\, A_R \right) \leftarrow \left(A_0 \,|\, A_1 \,\|\, A_2 \right), \left(\frac{x_T}{x_B} \right) \leftarrow \begin{pmatrix} x_0 \\ \hline x_1 \\ \hline x_2 \end{pmatrix}$$

endwhile

Figure 4.2: Algorithm for computing $y := Ax + y$ (blocked Variant 3).

two submatrices:

```
[ AT,...
  AB     ] = FLA_Part_2x1( A, mb, side )
```

Purpose: *Partition matrix* `A` *into a top and a bottom side where the side indicated by* `side` *has* `mb` *rows.*

`A`	– matrix to be partitioned
`mb`	– row dimension of side indicated by `side`
`side`	– side for which row dimension is given
`AT, AB`	– matrices for Top and Bottom parts

Here `side` can take on the values (character strings) `'FLA_TOP'` or `'FLA_BOTTOM'` to indicate that `mb` is the row dimension of the Top matrix `AT`, or the Bottom matrix `AB`, respectively. The routine can also be used to partition a (column) vector.

Remark 4.1 *Invocation of the M-script function*

```
[ AT,...
  AB     ] = FLA_Part_2x1( A,      mb, side )
```

in MATLAB *creates two new submatrices. Subsequent modifications of the contents of these submatrices therefore do not affect the original contents of the matrix. This is an important difference to consider with respect to the FLAME/C API, to be introduced in Section 4.3, where the submatrices are views (references) into the original matrix, not copies of it!*

As an example of the use of this routine, the translation of the algorithm fragment from Figure 4.1 on the left results in the code on the right:

Partition $A \rightarrow \left(\frac{A_T}{A_B}\right)$
where A_T has 0 rows

```
[ AT,...
  AB, ] = FLA_Part_2x1( A,...
                        0, 'FLA_TOP' )
```

Remark 4.2 *The above example stresses the fact that the formatting of the code can be used to help represent the algorithm in code. Clearly, some of the benefit of the API would be lost if in the example the code appeared as*

```
[ AT, AB ] = FLA_Part_2x1( A, 0, 'FLA_TOP' )
```

For some of the subsequent calls this becomes even more dramatic.

Also from Figure 4.1, we notice that it is necessary to be able to take a 2×1 partitioning of a given matrix A (or vector y) and repartition that into a 3×1 partitioning so that the submatrices that need to be updated and/or used for computation can be identified. To support this, we introduce the M-script function

```
[ A0,...
  A1,...
  A2     ] =  FLA_Repart_2x1_to_3x1( AT,...
                                     AB,      mb, side )
```

Purpose: *Repartition a 2×1 partitioning of a matrix into a 3×1 partitioning where submatrix* `A1` *with* `mb` *rows is split from from the side indicated by* `side`*.*

`AT, AB`	– matrices for Top and Bottom parts
`mb`	– row dimension of A_1
`side`	– side from which A_1 is partitioned
`A0, A1, A2`	– matrices for A_0, A_1, A_2

Here `side` can take on the values `'FLA_TOP'` or `'FLA_BOTTOM'` to indicate that submatrix `A1`, with `mb` rows, is partitioned from `AT` or `AB`, respectively.

Thus, for example, the translation of the algorithm fragment from Figure 4.1 on the left results in the code on the right:

Repartition

$$\left(\frac{A_T}{A_B}\right) \rightarrow \left(\frac{A_0}{\frac{A_1}{A_2}}\right)$$

where A_1 has m_b rows

```
[ A0,...
  A1,...
  A2 ] = FLA_Repart_2x1_to_3x1( AT,...
                                AB,      mb, 'FLA_BOTTOM' )
```

where parameter mb has the value m_b.

Remark 4.3 *Similarly to what is expressed in Remark 4.1, the invocation of the M-script function*

```
[ A0,...
  A1,...
  A2     ] =  FLA_Repart_2x1_to_3x1( AT,...
                                     AB,      mb, side )
```

creates three new matrices and any modification of the contents of A0, A1, A2 *does not affect the original matrix* A *nor the two submatrices* AT, AB*. Readability is greatly reduced if it were typeset like*

```
[ A0, A1, A2 ] =  FLA_Repart_2x1_to_3x1( AT, AB, mb, side )
```

Remark 4.4 *Choosing variable names can further relate the code to the algorithm, as is illustrated by comparing*

$$\left(\frac{A_0}{\frac{A_1}{A_2}}\right) \text{ and } \begin{array}{l}\texttt{A0}\\\texttt{A1}\\\texttt{A2}\end{array} \text{ ; and } \left(\frac{y_0}{\frac{\psi_1}{y_2}}\right) \text{ and } \begin{array}{l}\texttt{y0}\\\texttt{psi1}\\\texttt{y2}\end{array} .$$

Once the contents of the so-identified submatrices have been updated, AT and AB must be updated to reflect that progress is being made, in terms of the regions indicated by the thick lines. This movement of the thick lines is accomplished by a call to the M-script function

```
[ AT,...
  AB     ] = FLA_Cont_with_3x1_to_2x1( A0,...
                                       A1,...
                                       A2, side )
```

Purpose: *Update the* 2×1 *partitioning of a matrix by moving the boundaries so that* A1 *is joined to the side indicated by* side.

A0, A1, A2	– matrices for A_0, A_1, A_2
side	– side to which A_1 is joined
AT, AB	– matrices for Top and Bottom parts

For example, the algorithm fragment from Figure 4.1 on the left results in the code on the right:

Continue with

$$\left(\frac{A_T}{A_B}\right) \leftarrow \left(\frac{A_0}{\frac{A_1}{A_2}}\right)$$

```
[ AT,...
  AB ] = FLA_Cont_with_3x1_to_2x1( A0,...
                                   A1,...
                                   A2, 'FLA_TOP' )
```

The translation of the algorithm in Figure 4.1 to M-script code is now given in Figure 4.3. In the implementation, the parameter `mb_alg` holds the *algorithmic block size* (the number of elements of y that will be computed at each iteration). As, in general, $m(y)(= m(A))$ will not be a multiple of this block size, at each iteration m_b elements are computed, with m_b determined as $\min(m(y_B), \texttt{mb_alg})(= \min(m(A_B), \texttt{mb_alg}))$. Also, we use there a different variable for the input vector, `y`, and the output vector (result), `y_out`. The reason for this is that it will allow the FLAME@LAB code to be more easily translated to FLAME/C, the C API.

In M-script, `size( A, 1 )` and `size( A, 2 )` return the row and column dimension of array `A`, respectively. Placing a ";" at the end of a statement suppresses the printing of the value computed in the statement. The final statement

```
y_out = [ yT
          yB ];
```

sets the output variable `y_out` to the vector that results from concatenating `yT` and `yB`.

Exercise 4.5 *Visit* `$BASE/Chapter4/` *and follow the directions to reproduce and execute the code in Fig. 4.1.*

4.2.2 Vertical partitioning

Alternatively, as illustrated in Figure 4.2, in stating a linear algebra algorithm one may wish to initially partition a matrix by columns. For this we introduce the M-script function

```
[ AL, AR ] = FLA_Part_1x2( A,      nb, side )
```

Purpose: *Partition matrix* `A` *into a left and a right side where the side indicated by* `side` *has* `nb` *columns.*

`A`	– matrix to be partitioned
`nb`	– column dimension of side indicated by `side`
`side`	– side for which column dimension is given
`AL, AR`	– matrices for Left and Right parts

Here `side` can take on the values `'FLA_LEFT'` or `'FLA_RIGHT'` to indicate whether the partitioning is performed on the Left matrix or on the Right matrix, respectively. This routine can also be used to partition a row vector.

Matrices that have already been vertically partitioned can be further partitioned using the M-script function

```
1   function [ y_out ] = MATVEC_BLK_VAR1( A, x, y, mb_alg )
2     [ AT, ...
3       AB ] = FLA_Part_2x1( A,      0, 'FLA_TOP' );
4     [ yT, ...
5       yB ] = FLA_Part_2x1( y,      0, 'FLA_TOP' );
6
7     while ( size( yT, 1 ) < size( y, 1 ) )
8       mb = min( size( AB, 1 ), mb_alg );
9       [ A0, ...
10        A1, ...
11        A2 ] = FLA_Repart_2x1_to_3x1( AT, ...
12                                      AB,      mb, 'FLA_BOTTOM' );
13      [ y0, ...
14        y1, ...
15        y2 ] = FLA_Repart_2x1_to_3x1( yT, ...
16                                      yB,      mb, 'FLA_BOTTOM' );
17      %------------------------------------------------------------%
18        y1 = y1 + A1 * x;
19      %------------------------------------------------------------%
20      [ AT, ...
21        AB ] = FLA_Cont_with_3x1_to_2x1( A0, ...
22                                         A1, ...
23                                         A2,      'FLA_TOP' );
24      [ yT, ...
25        yB ] = FLA_Cont_with_3x1_to_2x1( y0, ...
26                                         y1, ...
27                                         y2,      'FLA_TOP' );
28    end
29    y_out = [ yT
30              yB ];
31  return
```

Figure 4.3: M-script code for the blocked algorithm for computing $y := Ax + y$ (Variant 1).

```
[ A0, A1, A2 ] = FLA_Repart_1x2_to_1x3( AL, AR,      nb, side )
```

Purpose: *Repartition a 1×2 partitioning of a matrix into a 1×3 partitioning where submatrix* `A1` *with* `nb` *columns is split from the right of* `AL` *or the left of* `AR`*, as indicated by* `side`*.*

`AL, AR`	– matrices for Left and Right parts
`nb`	– column dimension of A_1
`side`	– side from which A_1 is partitioned
`A0, A1, A2`	– matrices for A_0, A_1, A_2

Adding the middle submatrix to the first or last submatrix is now accomplished by a call to the M-script function

```
[ AL, AR ] = FLA_Cont_with_1x3_to_1x2( A0, A1, A2,      side )
```

Purpose: *Update the 1×2 partitioning of a matrix by moving the boundaries so that* `A1` *is joined to the side indicated by* `side`*.*

`A0, A1, A2`	– matrices for A_0, A_1, A_2
`side`	– side to which A_1 is joined
`AL, AR`	– matrices for views of Left and Right parts

Armed with these three functions, it is straight-forward to code the algorithm in Figure 4.2 in M-script language, as shown in Figure 4.4.

Exercise 4.6 *Visit* `$BASE/Chapter4/` *and follow the directions to reproduce and execute the code in Fig. 4.2.*

4.2.3 Bidimensional partitioning

Similar to the horizontal and vertical partitionings into submatrices discussed above, in stating a linear algebra algorithm one may wish to partition a matrix into four *quadrants* (as, for example, was necessary for the coefficient matrix L in the solution of triangular linear systems of equations; see Section 3.5). In FLAME@LAB, the following M-script function creates one matrix for each of the four quadrants:

```
[ ATL, ATR,...
  ABL, ABR     ] = FLA_Part_2x2( A,      mb, nb, quadrant )
```

Purpose: *Partition matrix* `A` *into four quadrants where the quadrant indicated by* `quadrant` *is* `mb` $\times$ `nb`*.*

`A`	– matrix to be partitioned
`mb, nb`	– row and column dimensions of quadrant indicated by `quadrant`
`quadrant`	– quadrant for which dimensions are given
`ATL, ATR, ABL, ABR`	– matrices for TL, TR, BL, and BR quadrants

Here `quadrant` is a MATLAB string that can take on the values `'FLA_TL'`, `'FLA_TR'`, `'FLA_BL'`, and `'FLA_BR'` to indicate that `mb` and `nb` are the dimensions of `ATL`, `ATR`, `ABL`, and `ATR`, respectively.

Given that a matrix is already partitioned into a 2×2 partitioning, it can be further repartitioned into 3×3 partitioning with the M-script function

```
1   function [ y_out ] = MATVEC_BLK_VAR3( A, x, y, nb_alg )
2     [ AL, AR ] = FLA_Part_1x2( A,      0, 'FLA_LEFT' );
3     [ xT, ...
4       xB ] = FLA_Part_2x1( x,          0, 'FLA_TOP' );
5
6     while ( size( xT, 1 ) < size( x, 1 ) )
7       nb = min( size( AR, 2 ), nb_alg );
8       [ A0, A1, A2 ]= FLA_Repart_1x2_to_1x3( AL, AR,      nb, 'FLA_RIGHT' );
9       [ x0, ...
10        x1, ...
11        x2 ] = FLA_Repart_2x1_to_3x1( xT, ...
12                                      xB,      nb, 'FLA_BOTTOM' );
13      %------------------------------------------------------------%
14        y = y + A1 * x1;
15      %------------------------------------------------------------%
16      [ AL, AR ] = FLA_Cont_with_1x3_to_1x2( A0, A1, A2,      'FLA_LEFT' );
17      [ xT, ...
18        xB ] = FLA_Cont_with_3x1_to_2x1( x0, ...
19                                         x1, ...
20                                         x2,      'FLA_TOP' );
21    end
22    y_out = y;
23  return
```

Figure 4.4: M-script code for the blocked algorithm for computing $y := Ax + y$ (Variant 3).

```
[ A00, A01, A02,...
  A10, A11, A12,...
  A20, A21, A22     ] =  FLA_Repart_2x2_to_3x3( ATL, ATR,...
                                                ABL, ABR,      mb, nb, quadrant )
```

Purpose: *Repartition a* 2×2 *partitioning of a matrix into a* 3×3 *partitioning where the* `mb` $\times$ `nb` *submatrix* `A11` *is split from the quadrant indicated by* `quadrant`.

`ATL, ATR, ABL, ABR`	– Matrices for TL, TR, BL, and BR quadrants
`mb, nb`	– row and column dimensions of A_{11}
`quadrant`	– quadrant from which A_{11} is partitioned
`A00-A22`	– matrices for A_{00}–A_{22}

Given a 3×3 partitioning, the middle submatrix can be appended to one of the four quadrants, `ATL`, `ATR`, `ABL`, and `ABR`, of the corresponding 2×2 partitioning with the M-script function

```
[ ATL, ATR,...
  ABL, ABR     ] = FLA_Cont_with_3x3_to_2x2( A00, A01, A02,...
                                              A10, A11, A12,...
                                              A20, A21, A22,      quadrant )
```

Purpose: *Update the* 2×2 *partitioning of a matrix by moving the boundaries so that* `A11` *is joined to the quadrant indicated by* `quadrant`*.*

`A00-A22` – matrices for A_{00}–A_{22}
`quadrant` – quadrant to which A_{11} is to be joined
`ATL, ATR, ABL, ABR` – matrices for TL, TR, BL, and BR quadrants

Remark 4.7 *The routines described in this section for the* MATLAB *M-script language suffice to implement a broad range of algorithms encountered in dense linear algebra.*

Exercise 4.8 *Visit* `$BASE/Chapter4/` *and follow the directions to download* FLAME@LAB *and to code and execute the algorithm in Figure 3.8 for solving the lower triangular linear system* $Lx = b$*.*

4.2.4 A few other useful routines

We have already encountered the situation where an algorithm computes with a lower triangular part of a matrix, or a matrix is symmetric and only one of the triangular parts is stored. The following M-script routines are frequently useful:

Routine	Function	Comment
TRIL $(\cdot)$	Returns lower triangular part of matrix	M-script native
TRIU$(\cdot)$	Returns upper triangular part of matrix	M-script native
TRILU$(\cdot)$	Returns lower triangular part of matrix and sets diagonal element to one	FLAME@LAB
TRIUU$(\cdot)$	Returns upper triangular part of matrix and sets diagonal element to one	FLAME@LAB
SYMML$(\cdot)$	$B =$ SYMML(A) set B to the symmetric matrix with TRIL $(B) =$ TRIL (A)	FLAME@LAB
SYMMU$(\cdot)$	$B =$ SYMMU(A) set B to the symmetric matrix with TRIU$(B) =$ TRIU(A)	FLAME@LAB

Routines identified as "native" are part of M-script while the routines identified as FLAME@LAB must be downloaded.

4.3 The FLAME/C Interface for the C Programming Language

It is easily recognized that the FLAME@LAB codes will likely fall short of attaining peak performance. (In particular, the copying that inherently occurs when submatrices are created and manipulated represents pure overhead.) In order to attain high performance, one tends to code in more traditional programming languages, like Fortran or C, and to link to high-performance implementations of libraries such as the Basic Linear Algebra

Subprograms (BLAS) [21, 10, 9]. In this section we introduce a set of library routines that allow us to capture linear algebra algorithms presented in the format used in FLAME in C code.

Readers familiar with MPI [23], PETSc [1], or PLAPACK [28] will recognize the programming style, *object-based programming*, as being very similar to that used by those (and other) interfaces. It is this style of programming that allows us to hide the indexing details much like FLAME@LAB does. We will see that a more substantial infrastructure must be provided in addition to the routines that partition and repartition matrix objects.

4.3.1 Initializing and finalizing FLAME/C

Before using the FLAME/C environment one must initialize it with a call to

```
FLA_Error FLA_Init( )
```

Purpose: *Initialize* FLAME/C.

If no more FLAME/C calls are to be made, the environment is exited by calling

```
FLA_Error FLA_Finalize( )
```

Purpose: *Finalize* FLAME/C.

4.3.2 Linear algebra objects

The following attributes describe a matrix as it is stored in the memory of a computer:

- the datatype of the entries in the matrix, e.g., `double` or `float`,
- m and n, the row and column dimensions of the matrix,
- the address where the data is stored, and
- the mapping that describes how the two-dimensional array of elements in a matrix is mapped to memory, which is inherently one-dimensional.

The following call creates an object (*descriptor* or *handle*) of type `FLA_Obj` for a matrix and creates space to store the entries in the matrix:

```
FLA_Error FLA_Obj_create( FLA_Datatype datatype, int m, int n, FLA_Obj *matrix )
```

Purpose: *Create an object that describes an* `m` × `n` *matrix and create the associated storage array.*

`datatype`	– datatype of matrix
`m, n`	– row dimensions of matrix
`matrix`	– descriptor for the matrix

Valid datatype values include

FLA_INT, FLA_DOUBLE, FLA_FLOAT, FLA_DOUBLE_COMPLEX, FLA_COMPLEX

for the obvious datatypes that are commonly encountered. The leading dimension of the array that is used to store the matrix in column-major order is itself determined inside of this call.

Remark 4.9 *For simplicity, we chose to limit the storage of matrices to column-major storage. The* leading dimension *of a matrix can be thought of as the dimension of the array in which the matrix is embedded (which is often larger than the row-dimension of the matrix) or as the increment (in elements) required to address consecutive elements in a row of the matrix. Column-major storage is chosen to be consistent with Fortran, which is often still the choice of language for linear algebra applications.*

FLAME/C treats vectors as special cases of matrices: an $n \times 1$ matrix or a $1 \times n$ matrix. Thus, to create an object for a vector x of n double-precision real numbers either of the following calls suffices:

```
FLA_Obj_create( FLA_DOUBLE, n, 1, &x );
FLA_Obj_create( FLA_DOUBLE, 1, n, &x );
```

Here `n` is an integer variable with value n and `x` is an object of type `FLA_Obj`.

Similarly, FLAME/C treats a scalar as a 1×1 matrix. Thus, to create an object for a scalar α the following call is made:

```
FLA_Obj_create( FLA_DOUBLE, 1, 1, &alpha );
```

where `alpha` is an object of type `FLA_Obj`. A number of scalars occur frequently and are therefore predefined by FLAME/C: `FLA_MINUS_ONE`, `FLA_ZERO`, and `FLA_ONE`.

If an object is created with `FLA_Obj_create`, a call to `FLA_Obj_free` is required to ensure that all space associated with the object is properly released:

```
FLA_Error FLA_Obj_free( FLA_Obj *matrix )
```

Purpose: *Free all space allocated to store data associated with* `matrix`.

`matrix` – descriptor for the object

4.3.3 Inquiry routines

In order to be able to work with the raw data, a number of inquiry routines can be used to access information about a matrix (or vector or scalar). The datatype and row and column dimensions of the matrix can be extracted by calling

```
FLA_Datatype FLA_Obj_datatype( FLA_Obj matrix )
int          FLA_Obj_length  ( FLA_Obj matrix )
int          FLA_Obj_width   ( FLA_Obj matrix )
```

Purpose: *Extract datatype, row, or column dimension of* `matrix`*, respectively.*

`matrix`	– object that describes the matrix
return value	– datatype, row, or column dimension of matrix, respectively

The address of the array that stores the matrix and its leading dimension can be retrieved by calling

```
void *FLA_Obj_buffer( FLA_Obj matrix )
int   FLA_Obj_ldim    ( FLA_Obj matrix )
```

Purpose: *Extract address and leading dimension of* `matrix`*, respectively.*

`matrix`	– object that describes the matrix
return value	– address and leading dimension of matrix, respectively

4.3.4 Filling the entries of an object

A sample routine for filling a matrix (or a vector) with data is given in Figure 4.5. The macro definition in line 3 is used to access the matrix A stored in array `A` using column-major ordering.

4.3.5 Attaching an existing buffer of elements

Sometimes a user already has a buffer for the data and an object that references this data needs to be created. For this FLAME/C provides the call

```
FLA_Error FLA_Obj_create_without_buffer( FLA_Datatype datatype, int m, int n, FLA_Obj *matrix )
```

Purpose: *Create an object that describes an* `m` $\times$ `n` *matrix without creating the associated storage array.*

`datatype`	– datatype of matrix
`m, n`	– row dimensions of matrix
`matrix`	– descriptor for the matrix

A buffer with data, assumed to be stored in column major order with a know leading dimension, can then be attached to the object with the call

```
FLA_Error FLA_Obj_attach_buffer( void *buff, int ldim, FLA_Obj *matrix )
```

Purpose: *Create an object that describes an* `m` $\times$ `n` *matrix without creating the associated storage array.*

`buff`	– address of buffer
`ldim`	– leading dimension of matrix
`matrix`	– descriptor for the matrix

```
1  #include "FLAME.h"
2
3  #define BUFFER( i, j ) buff[ (j)*lda + (i) ]
4
5  void fill_matrix( FLA_Obj A )
6  {
7    FLA_Datatype
8      datatype;
9    int
10     m, n, lda;
11
12   datatype = FLA_Obj_datatype( A );
13   m        = FLA_Obj_length( A );
14   n        = FLA_Obj_width ( A );
15   lda      = FLA_Obj_ldim  ( A );
16
17   if ( datatype == FLA_DOUBLE ){
18     double *buff;
19     int    i, j;
20
21     buff = ( double * ) FLA_Obj_buffer( A );
22
23     for ( j=0; j<n; j++ )
24       for ( i=0; i<m; i++ )
25         BUFFER( i, j ) = i+j*0.01;
26   }
27   else
28     FLA_Check_error_code( FLA_NOT_YET_IMPLEMENTED );
29 }
```

Figure 4.5: A sample routine for filling a matrix.

Freeing an object that was created without a buffer is accomplished by calling

```
FLA_Error FLA_Obj_free_without_buffer( FLA_Obj *matrix )
```

Purpose: *Free the space allocated for the object* `matrix` *without freeing the buffer that stores elements of* `matrix`.

`matrix`	– descriptor for the object

4.3.6 A most useful utility routine

We single out one of the more useful routines in the FLAME/C library, which is particularly helpful for simple debugging:

```
FLA_Error FLA_Obj_show( char *string1, FLA_Obj A, char *format, char *string2 )
```

Purpose: *Print the contents of* `A`.

`string1`	– string to be printed before contents
`A`	– descriptor for A
`format`	– format to be used to print each individual element
`string2`	– string to be printed after contents

In particular, the result of

```
FLA_Obj_show( "A =[", A, "%lf", "];" );
```

is

```
A = [
< entries_of_A >
];
```

which can then be cut and pasted into a MATLAB or OCTAVE session. This becomes useful when checking results against a FLAME@LAB implementation of an operation.

4.3.7 Writing a driver routine

We now give an example of how to use the calls introduced so far to write a sample driver routine that calls a routine that performs the matrix-vector multiplication $y := y + Ax$.

In Figure 4.6 we present the contents of the driver routine:

- **line 1:** FLAME/C program files start by including the `FLAME.h` header file.
- **lines 5–6:** FLAME/C objects `A`, `x`, and `y`, which hold matrix A and vectors x and y, respectively, are declared to be of type `FLA_Obj`.
- **line 10:** Before any calls to FLAME/C routines can be made, the environment must be initialized by a call to `FLA_Init`.

```
1  #include "FLAME.h"
2
3  void main()
4  {
5    FLA_Obj
6      A, x, y;
7    int
8      m, n;
9
10   FLA_Init( );
11
12   printf( "enter matrix dimensions m and n:" );
13   scanf( "%d%d", &m, &n );
14
15   FLA_Obj_create( FLA_DOUBLE, m, n, &A );
16   FLA_Obj_create( FLA_DOUBLE, n, 1, &x );
17   FLA_Obj_create( FLA_DOUBLE, m, 1, &y );
18
19   fill_matrix( A );
20   fill_matrix( x );
21   fill_matrix( y );
22
23   FLA_Obj_show( "y = [", y, "%lf", "]" );
24
25   matvec_blk_var1( A, x, y );
26
27   FLA_Obj_show( "A = [", A, "%lf", "]" );
28   FLA_Obj_show( "x = [", x, "%lf", "]" );
29   FLA_Obj_show( "y = [", y, "%lf", "]" );
30
31   FLA_Obj_free( &A );
32   FLA_Obj_free( &y );
33   FLA_Obj_free( &x );
34
35   FLA_Finalize( );
36 }
```

Figure 4.6: A sample C driver routine for matrix-vector multiplication.

- **lines 12–13:** In our example, the user inputs the row and column dimension of matrix A.
- **lines 15–17:** Descriptors are created for A, x, and y.
- **lines 19–21:** The routine in Figure 4.5 is used to fill A, x, and y with values.
- **line 25:** Compute $y := y + Ax$ using the routine for performing that operation (to be discussed later).
- **lines 23, 27–29:** Print out the contents of A, x, and (both the initial and final) y.
- **lines 31–33:** Free the objects.
- **line 35:** Finalize FLAME/C.

Exercise 4.10 *Visit* `$BASE/Chapter4/` *and follow the directions on how to download the libFLAME library. Then compile and execute the sample driver as directed.*

4.3.8 Horizontal partitioning

Having introduced the basics of the C API for FLAME, let us turn now to those routines that allow to objects to be partitioned and repartitioned.

Figure 4.1 illustrates the need for a 2×1 partitioning of a matrix (or a vector). In C we avoid complicated indexing by introducing the notion of a *view*, which is a **reference** into an existing matrix or vector.

Given a descriptor `A` of a matrix A, the following routine creates descriptors for two submatrices:

```
FLA_Error FLA_Part_2x1( FLA_Obj A, FLA_Obj *AT,
                                   FLA_Obj *AB,      int mb, FLA_Side side )
```

Purpose: *Partition matrix* `A` *into a top and bottom side where the side indicated by* `side` *has* `mb` *rows.*

`A`	– matrix to be partitioned
`mb`	– row dimension of side indicated by `side`
`side`	– side for which row dimension is given
`AT, AB`	– views of Top and Bottom parts

Here `side` can take on the values `FLA_TOP` or `FLA_BOTTOM` to indicate that `mb` indicates the row dimension of `AT` or `AB`, respectively.

For example, the algorithm fragment from Figure 4.1 on the left is translated into the code on the right

Partition $A \rightarrow \left(\frac{A_T}{A_B}\right)$ **where** A_T has 0 rows	`FLA_Part_2x1( A, &AT,` `&AB, 0, FLA_TOP );`

Remark 4.11 *As was the case for* FLAME@lab, *we stress that the formatting of the code is important: Clearly, some of the benefit of the API would be lost if in the example the code appeared as*

```
FLA_Part_2x1( A, &AT, &AB, 0, FLA_TOP );
```

Remark 4.12 *Invocation of the C routine*

```
FLA_Part_2x1( A,    &AT,
                    &AB,      0, FLA_TOP );
```

creates two views (referrences), one for each submatrix. Subsequent modifications of the contents of a view **do** *affect the original contents of the matrix. This is an important difference to consider with respect to the* FLAME@LAB *API introduced in the previous section, where the submatrices were copies of the original matrix!*

From Figure 4.1, we also realize the need for an operation that takes a 2×1 partitioning of a given matrix A and repartitions this into a 3×1 partitioning so that submatrices that need to be updated and/or used for computation can be identified. To support this, we introduce the routine:

```
FLA_Error FLA_Repart_from_2x1_to_3x1( FLA_Obj AT,   FLA_Obj *A0,
                                                    FLA_Obj *A1,
                                      FLA_Obj AB,   FLA_Obj *A2, int mb, FLA_Side side )
```

Purpose: *Repartition a* 2×1 *partitioning of matrix* `A` *into a* 3×1 *partitioning where submatrix* `A1` *with* `mb` *rows is split from the side indicated by* `side`*.*

`AT, AB`	– views of Top and Bottom sides
`mb`	– row dimension of A_1
`side`	– side from which A_1 is partitioned
`A0, A1, A2`	– views of A_0, A_1, A_2

Here `side` can take on the values `FLA_TOP` or `FLA_BOTTOM` to indicate that `mb` submatrix `A1` is partitioned from `AT` or `AB`, respectively.

Thus, the following fragment of algorithm from Figure 4.1 on the left translates to the code on the right:

Repartition

$$\left(\frac{A_T}{A_B}\right) \rightarrow \left(\frac{A_0}{\frac{A_1}{A_2}}\right)$$

where A_1 has m_b rows

```
FLA_Repart_2x1_to_3x1( AT,        &A0,
                     /* ** */   /* ** */
                                  &A1,
                          AB,     &A2,  mb, FLA_BOTTOM );
```

Remark 4.13 *Choosing variable names can further relate the code to the algorithm, as is illustrated by comparing*

$$\left(\frac{A_0}{\frac{A_1}{A_2}}\right) \textit{ and } \begin{matrix}\texttt{A0}\\ \texttt{A1}\\ \texttt{A2}\end{matrix} \text{ ; and } \left(\frac{y_0}{\frac{\psi_1}{y_2}}\right) \textit{ and } \begin{matrix}\texttt{y0}\\ \texttt{psi1}\\ \texttt{y2}\end{matrix} .$$

Once the contents of the so-identified submatrices have been updated, the contents of `AT` and `AB` must be updated to reflect that progress is being made, in terms of the regions indicated by the thick lines. This

movement of the thick lines is accomplished by a call to the C routine:

```
FLA_Error FLA_Cont_with_3x1_to_2x1( FLA_Obj *AT,  FLA_Obj A0,
                                                  FLA_Obj A1,
                                    FLA_Obj *AB,  FLA_Obj A2,      FLA_Side side )
```

Purpose: *Update the* 2×1 *partitioning of matrix* `A` *by moving the boundaries so that* `A1` *is joined to the side indicated by* `side`.

`A0, A1, A2`	– views of A_0, A_1, A_2
`side`	– side to which A_1 is joined
`AT, AB`	– views of Top and Bottom sides

Thus, the algorithm fragment from Figure 4.1 on the left results in the code on the right:

Continue with

$$\left(\frac{A_T}{A_B}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

```
FLA_Cont_with_3x1_to_2x1( &AT,               A0,
                                             A1,
                         /* ** */         /* ** */
                          &AB,               A2,      FLA_TOP );
```

Using the three routines for horizontal partitioning, the algorithm in Figure 4.1 is translated into the C code in Figure 4.7.

4.3.9 Vertical partitioning

Figure 4.2 illustrates that, when stating a linear algebra algorithm one may in wish to proceed by columns. Therefore, we introduce the following pair of C routines for partitioning and repartitioning a matrix (or vector) vertically:

```
FLA_Error FLA_Part_1x2( FLA_Obj A, FLA_Obj *AL, FLA_Obj *AR,      int nb, FLA_Side  side )
```

Purpose: *Partition matrix* `A` *into a left and right side where the side indicated by* `side` *has* `nb` *columns.*

`A`	– matrix to be partitioned
`nb`	– column dimension of side indicated by `side`
`side`	– side for which column dimension is given
`AL, AR`	– views of Left and Right parts

and

```
 1  #include "FLAME.h"
 2
 3  void MATVEC_BLK_VAR1( FLA_Obj A, FLA_Obj x, FLA_Obj y, int mb_alg )
 4  {
 5    FLA_Obj AT,              A0,              yT,              y0,
 6            AB,              A1,              yB,              y1,
 7                             A2,                               y2;
 8
 9    int mb;
10
11    FLA_Part_2x1( A,    &AT,
12                        &AB,      0,    FLA_TOP );
13
14    FLA_Part_2x1( y,    &yT,
15                        &yB,      0,    FLA_TOP );
16
17    while ( FLA_Obj_length( yT ) < FLA_Obj_length( y ) ){
18      mb = min( FLA_Obj_length( AB ), mb_alg );
19
20      FLA_Repart_2x1_to_3x1( AT,                &A0,
21                           /* ** */            /* ** */
22                                                &A1,
23                             AB,                &A2,      mb, FLA_BOTTOM );
24
25      FLA_Repart_2x1_to_3x1( yT,                &y0,
26                           /* ** */            /* ** */
27                                                &y1,
28                             yB,                &y2,      mb, FLA_BOTTOM );
29      /*------------------------------------------------------------*/
30        MATVEC_VAR1( A1, x, y1 );
31      /*------------------------------------------------------------*/
32      FLA_Cont_with_3x1_to_2x1( &AT,                A0,
33                                                    A1,
34                               /* ** */           /* ** */
35                                 &AB,               A2,   FLA_TOP );
36
37      FLA_Cont_with_3x1_to_2x1( &yT,                y0,
38                                                    y1,
39                               /* ** */           /* ** */
40                                 &yB,               y2,   FLA_TOP );
41    }
42  }
```

Figure 4.7: FLAME/C code for computing $y := Ax + y$ (Blocked Variant 1).

```
FLA_Error FLA_Repart_from_1x2_to_1x3( FLA_Obj AL,                FLA_Obj AR,
                                      FLA_Obj *A0, FLA_Obj *A1, FLA_Obj *A2,
                                      int nb, FLA_Side side )
```

Purpose: *Repartition a 1×2 partitioning of matrix* `A` *into a 1×3 partitioning where submatrix* `A1` *with* `nb` *columns is split from the side indicated by* `side`.

`AL, AR`	– views of Left and Right sides
`nb`	– column dimension of A_1
`side`	– side from which A_1 is partitioned
`A0, A1, A2`	– views of A_0, A_1, A_2

Here `side` can take on the values `FLA_LEFT` or `FLA_RIGHT`.

Adding the middle submatrix to the first or last submatrix is now accomplished by a call to the C routine:

```
FLA_Error FLA_Cont_with_1x3_to_1x2( FLA_Obj *AL,            FLA_Obj *AR,
                                    FLA_Obj A0, FLA_Obj A1, FLA_Obj A2,   FLA_Side side )
```

Purpose: *Update the 1×2 partitioning of matrix* `A` *by moving the boundaries so that* `A1` *is joined to the side indicated by* `side`.

`A0, A1, A2`	– views of A_0, A_1, A_2
`side`	– side to which A_1 is joined
`AL, AR`	– views of Left and Right sides

Using the three routines just introduced, the algorithm in Figure 4.1 is translated into the C code in Figure 4.8.

4.3.10 Bidimensional partitioning

Similar to the horizontal and vertical partitionings into submatrices discussed above, in stating a linear algebra algorithm one may wish to partition a matrix into four quadrants. The following C routine creates one matrix for each of the four quadrants:

```
FLA_Error FLA_Part_2x2( FLA_Obj A, FLA_Obj *ATL, FLA_Obj *ATR,
                                   FLA_Obj *ABL, FLA_Obj *ABR,
                        int mb, int nb, FLA_Quadrant quadrant )
```

Purpose: *Partition matrix* `A` *into four quadrants where the quadrant indicated by* `quadrant` *is* `mb` $\times$ `nb`.

`A`	– matrix to be partitioned
`mb, nb`	– row and column dimensions of quadrant indicated by `quadrant`
`quadrant`	– quadrant for which dimensions are given in `mb` and `nb`
`ATL-ABR`	– views of TL, TR, BL, and BR quadrants

```
 1  #include "FLAME.h"
 2
 3  void MATVEC_BLK_VAR3( FLA_Obj A, FLA_Obj x, FLA_Obj y, int nb_alg )
 4  {
 5    FLA_Obj AL,    AR,       A0,  A1,  A2;
 6
 7    FLA_Obj xT,              x0,
 8            xB,              x1,
 9                             x2;
10
11    int nb;
12
13    FLA_Part_1x2( A,    &AL,  &AR,
14                  0, FLA_LEFT );
15
16    FLA_Part_2x1( x,    &xT,
17                        &xB,
18                  0, FLA_TOP );
19
20    while ( FLA_Obj_length( xT ) < FLA_Obj_length( x ) ){
21      b = min( FLA_Obj_width( AR ), nb_alg );
22
23      FLA_Repart_1x2_to_1x3( AL,  /**/ AR,        &A0, /**/ &A1, &A2,
24                             nb, FLA_RIGHT );
25
26      FLA_Repart_2x1_to_3x1( xT,                &x0,
27                          /* ** */            /* ** */
28                                                &x1,
29                             xB,                &x2,
30                             nb, FLA_BOTTOM );
31      /*------------------------------------------------------------*/
32        MATVEC_VAR2( A1, x1, y );
33      /*------------------------------------------------------------*/
34      FLA_Cont_with_1x3_to_1x2( &AL,  /**/ &AR,        A0, A1, /**/ A2,
35                                FLA_LEFT );
36
37      FLA_Cont_with_3x1_to_2x1( &xT,                x0,
38                                                    x1,
39                              /* ** */           /* ** */
40                                &xB,                x2,
41                                FLA_TOP );
42    }
43  }
```

Figure 4.8: FLAME/C code for computing $y := Ax + y$ (Blocked Variant 3).

Here quadrant can take on the values FLA_TL, FLA_TR, FLA_BL, and FLA_BR (defined in FLAME.h) to indicate that mb and nb specify the dimensions of the Top-Left, Top-Right, Bottom-Left, or Bottom-Right quadrant, respectively.

Given that a matrix is already partitioned into a 2×2 partitioning, it can be further repartitioned into 3×3 partitioning with the C routine:

```
FLA_Error FLA_Repart_from_2x2_to_3x3
      ( FLA_Obj ATL, FLA_Obj ATR,   FLA_Obj *A00, FLA_Obj *A01, FLA_Obj *A02,
                                    FLA_Obj *A10, FLA_Obj *A11, FLA_Obj *A12,
        FLA_Obj ABL, FLA_Obj ABR,   FLA_Obj *A20, FLA_Obj *A21, FLA_Obj *A22,
        int mb, int nb, FLA_Quadrant quadrant )
```

Purpose: *Repartition a* 2×2 *partitioning of matrix* A *into a* 3×3 *partitioning where* mb $\times$ nb *submatrix* A11 *is split from the quadrant indicated by* quadrant*.*

ATL, ATR, ABL, ABR	– views of TL, TR, BL, and BR quadrants
mb, nb	– row and column dimensions of A_{11}
quadrant	– quadrant from which A_{11} is partitioned
A00-A22	– views of A_{00}–A_{22}

Here quadrant can again take on the values FLA_TL, FLA_TR, FLA_BL, and FLA_BR to indicate that mb $\times$ nb submatrix A11 is split from submatrix ATL, ATR, ABL, or ABR, respectively.

Given a 3×3 partitioning, the middle submatrix can be appended to either of the four quadrants, ATL, ATR, ABL, and ABR, of the corresponding 2×2 partitioning with the C routine

```
FLA_Error FLA_Cont_with_3x3_to_2x2
      ( FLA_Obj *ATL, FLA_Obj *ATR,   FLA_Obj A00, FLA_Obj A01, FLA_Obj A02,
                                      FLA_Obj A10, FLA_Obj A11, FLA_Obj A12,
        FLA_Obj *ABL, FLA_Obj *ABR,   FLA_Obj A20, FLA_Obj A21, FLA_Obj A22,
        FLA_Quadrant quadrant )
```

Purpose: *Update the* 2×2 *partitioning of matrix* A *by moving the boundaries so that* A11 *is joined to the quadrant indicated by* quadrant*.*

ATL, ATR, ABL, ABR	– views of TL, TR, BL, and BR quadrants
A00-A22	– views of A_{00}–A_{22}
quadrant	– quadrant to which A_{11} is to be joined

Here the value of quadrant (FLA_TL, FLA_TR, FLA_BL, or FLA_BR) specifies the quadrant submatrix A11 is to be joined.

4.3.11 Merging views

Sometimes it becomes convenient to merge multiple views into a single view. For this FLAME/C provides the routines

```
FLA_Error FLA_Merge_2x1( FLA_Obj AT,
                         FLA_Obj AB,   FLA_Obj *A )
```

Purpose: *Merge a 2×1 partitioned matrix.*

`AT, AB`	– views of Top and Bottom sides
`A`	– matrix of which `AT` and `AB` are views

```
FLA_Error FLA_Merge_1x2( FLA_Obj AL, FLA_Obj AR,      FLA_Obj *A )
```

Purpose: *Merge a 1×2 partitioned matrix.*

`AL, AR`	– views of Left and Right sides
`A`	– matrix of which `AT` and `AB` are views

```
FLA_Error FLA_Merge_2x2( FLA_Obj ATL, FLA_Obj ATR,
                         FLA_Obj ABL, FLA_Obj ABR,    FLA_Obj *A )
```

Purpose: *Merge a 2×2 partitioned matrix.*

`ATL-ABR`	– views of quadrants
`A`	– matrix of which `ATL-ABR` are views

4.3.12 Computational kernels

All operations described in the last subsection hide the details of indexing in the linear algebra objects. To compute with and/or update data associated with a linear algebra object, one calls subroutines that perform the desired operations.

Such subroutines typically take one of three forms:

- Subroutines coded using the FLAME/C interface (including, possibly, a recursive call);
- Subroutines coded using a more traditional coding style; or
- Wrappers to highly optimized kernels.

Naturally these are actually three points on a spectrum of possibilities, since one can mix these techniques.

A number of matrix and/or vector operations have been identified to be frequently used by the linear algebra community. Many of these are part of the BLAS. Since highly optimized implementations of these operations are supported by widely available library implementations, it makes sense to provide a set of subroutines that are simply wrappers to the BLAS. An example of a wrapper routine to the level 2 BLAS routine `cblas_dgemv`, a commonly available kernel for computing a matrix-vector multiplication, is given in Figure 4.9.

For additional information on supported functionality see Appendix B or visit the webpage

`http://www.cs.utexas.edu/users/flame/`

```
1  #include "FLAME.h"
2  #include "cblas.h"
3
4  void matvec_wrapper( FLA_Obj A, FLA_Obj x, FLA_Obj y )
5  {
6    FLA_Datatype
7      datatype_A;
8    int
9      m_A, n_A, ldim_A, m_x, n_y, inc_x, m_y, n_y, inc_y;
10
11   datatype_A = FLA_Obj_datatype( A );
12   m_A        = FLA_Obj_length( A );
13   n_A        = FLA_Obj_width ( A );
14   ldim_A     = FLA_Obj_ldim  ( A );
15
16   m_x        = FLA_Obj_length( x );
17   n_x        = FLA_Obj_width ( x );
18
19   m_y        = FLA_Obj_length( y );
20   n_y        = FLA_Obj_width ( y );
21
22   if ( m_x == 1 ) {
23     m_x  = n_x;
24     inc_x = FLA_Obj_ldim( x );
25   }
26   else inc_x = 1;
27
28   if ( m_y == 1 ) {
29     m_y  = n_y;
30     inc_y = FLA_Obj_ldim( y );
31   }
32   else inc_y = 1;
33
34   if ( datatype_A == FLA_DOUBLE ){
35     double *buff_A, *buff_x, *buff_y;
36
37     buff_A = ( double * ) FLA_Obj_buffer( A );
38     buff_x = ( double * ) FLA_Obj_buffer( x );
39     buff_y = ( double * ) FLA_Obj_buffer( y );
40
41     cblas_dgemv( CblasColMaj, CblasNoTrans,
42                  1.0, buff_A, ldim_A, buff_x, inc_x,
43                  1.0, buff_y, inc_y );
44   }
45   else FLA_Abort( "Datatype not yet supported", __LINE__, __FILE__ );
46 }
```

Figure 4.9: A sample matrix-vector multiplication routine. This routine is implemented as a wrapper to the CBLAS routine `cblas_dgemv` for matrix-vector multiplications.

Exercise 4.14 *Use the routines in the* FLAME/C *API to implement the algorithm in Figure 3.8 for computing the solution of a lower triangular system* $b := x$*, where* $Lx = b$*.*

4.4 Summary

The FLAME@lab and FLAMEC APIs illustrate how by raising the level of abstraction at which one codes, intricate indexing can be avoided in the code, therefore reducing the opportunity for the introduction of errors and raising the confidence in correctness of the code. Thus, the proven correctness of those algorithms derived using the FLAME methodology translates to a high degree of confidence in the implementation.

The two APIs that we presented are simple ones and serve to illustrate the issues. Similar interfaces to more elaborate programming languages (e.g., C++, Java, and LabView's G graphical programming language) can be easily defined allowing special features of those languages to be used to even further raise the level of abstraction at which one codes.

4.5 Further Exercises

For additional exercises, visit `$BASE/Chapter4/`.

Chapter 5

High Performance Algorithms

Dense linear algebra operations are often at the heart of scientific computations that stress even the fastest computers available. As a result, it is important that routines that compute these operations attain high performance in the sense that they perform near the minimal number of operations and achieve near the highest possible rate of execution. In this chapter we show that high performance can be achieved by casting computation as much as possible in terms of the matrix-matrix product operation (GEMM). We also expose that for many matrix-matrix operations in linear algebra the derivation techniques discussed so far yield algorithms that are rich in GEMM.

Remark 5.1 *Starting from this chapter, we adopt a more concise manner of presenting the derivation of the algorithms where we only specify the partitioning of the operands and the loop-invariant. Recall that these two elements prescribe the remaining derivation procedure of the worksheet and, therefore, the algorithm.*

5.1 Architectural Considerations

We start by introducing a simple model of a processor and its memory hierarchy. Next, we argue that the vector-vector and matrix-vector operations discussed so far can inherently not achieve high performance, while the matrix-matrix product potentially can.

5.1.1 A simple model

A processor, current as of this writing, includes a *Central Processing Unit* (*CPU*), in which computation occurs, and a (main) *memory*, usually a Random Access Memory (RAM). In order to perform an operation in the CPU,

Operation		flops	memops	$\frac{\text{flops}}{\text{memops}}$
Vector-vector operations				
SCAL	$x := \alpha x$	n	n	1/1
ADD	$x := x + y$	n	n	1/1
DOT	$\alpha := x^{\mathrm{T}} y$	$2n$	$2n$	1/1
APDOT	$\alpha := \alpha + x^{\mathrm{T}} y$	$2n$	$2n$	1/1
AXPY	$y := \alpha x + y$	$2n$	$3n$	2/3
Matrix-vector operations				
GEMV	$y := \alpha A x + \beta y$	$2n^2$	n^2	2/1
GER	$A := \alpha y x^{\mathrm{T}} + A$	$2n^2$	$2n^2$	1/1
TRSV	$x := T^{-1} b$	n^2	$n^2/2$	2/1
Matrix-matrix operations				
GEMM	$C := \alpha AB + \beta C$	$2n^3$	$4n^2$	$n/2$

Figure 5.1: Analysis of the cost of various operations. Here $\alpha \in \mathbb{R}$, $x, y, b \in \mathbb{R}^n$, and $A, B, C, T \in \mathbb{R}^{n \times n}$, with T being triangular.

data must be fetched from the memory to the *registers* in the CPU, and results must eventually be returned to the memory. The fundamental obstacle to high performance (executing useful computations at the rate at which the CPU can process) is the speed of memory: fetching and/or storing a data item from/to the memory requires more time than it takes to perform a flop with it. This is known as the *memory bandwidth bottleneck*.

The solution has been to introduce a small *cache* memory, which is fast enough to keep up with the CPU, but small enough to be economical (e.g., in terms of space required inside the processor). The pyramid in Figure 5.2 depicts the resulting model of the memory architecture. The model is greatly simplified in that currently most architectures have several layers of cache and often also present additional, even slower, levels of memory. The model is sufficient to explain the main issues behind achieving high performance.

5.1.2 Inherent limitations

Vector-vector and matrix-vector operations can inherently benefit little from cache memories. As an example, consider the AXPY operation, $y := \alpha x + y$, where $\alpha \in \mathbb{R}$ and $x, y \in \mathbb{R}^n$ initially and ultimately reside in the main memory. Define a *memop* as a memory fetch or store operation. An efficient implementation of AXPY will load α from memory to a register and will then compute with x and y, which must be fetched from memory as well. The updated result, y, must also be stored, for a total of about $3n$ memops for the $2n$ flops that are executed. Thus, three memops are required for every two flops. If memops are more expensive than flops (as usually is the case), it is the memops that limit the performance that can be attained for AXPY. In Figure 5.1 we summarize the results of a similar analysis for other vector-vector operations.

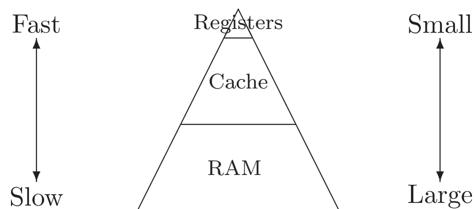

Figure 5.2: Simplified model of the memory architecture used to illustrate the high-performance implementation of GEMM.

Next, consider the GEMV operation $y := Ax + y$, where $x, y \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n\times n}$. This operation involves roughly n^2 data (for the matrix), initially stored in memory, and $2n^2$ flops. Thus, an optimal implementation will fetch every element of A exactly once, yielding a ratio of one memop for every two flops. Although this is better than the ratio for the AXPY, memops still dominate the cost of the algorithm if they are much slower than flops. Figure 5.1 summarizes the analysis for other matrix-vector operations.

It is by casting linear algebra algorithms in terms of the matrix-matrix product, GEMM, that there is the opportunity to overcome this memory bottleneck. Consider the product $C := AB + C$ where all three matrices are square of order n. This operation involves $4n^2$ memops (A and B must be fetched from memory while C must be both fetched and stored) and requires $2n^3$ flops[1] for a ratio of $4n^2/2n^3 = 2/n$ memops/flops. Thus, if n is large enough, the cost of performing memops is small relative to that of performing useful computations with the data, and there is an opportunity to amortize the cost of fetching data into the cache over many computations.

5.2 Matrix-Matrix Product: Background

While it is the matrix-vector product, discussed in Chapter 3, that is the fundamental matrix operation that links matrices to linear operators, the matrix-matrix product is the operation that is fundamental to high-performance implementation of linear algebra operations. In this section we review the matrix-matrix product, its properties, and some nomenclature.

The general form of the GEMM operation is given by

$$C := \alpha\, \mathrm{op}(A)\, \mathrm{op}(B) + \beta C,$$

where $\alpha, \beta \in \mathbb{R}$, $\mathrm{op}(A) \in \mathbb{R}^{m\times k}$, $\mathrm{op}(B) \in \mathbb{R}^{k\times n}$, $C \in \mathbb{R}^{m\times n}$, and $\mathrm{op}(X)$ is one of X or X^{T}. That is, the GEMM

[1] The cost of GEMM will be show to be cubic later in this chapter.

operation can take one of the following four forms

$$\begin{array}{l} C := \alpha A \;\; B + \beta C, \\ C := \alpha A^{\mathrm{T}} B + \beta C, \\ C := \alpha A \;\; B^{\mathrm{T}} + \beta C, \quad \text{or} \\ C := \alpha A^{\mathrm{T}} B^{\mathrm{T}} + \beta C. \end{array}$$

In the remainder of this chapter we will focus on the special case where $\alpha = \beta = 1$ and matrices A and B are not transposed. All insights can be easily extended to the other cases.

Throughout this chapter, unless otherwise stated, we will assume that $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$, and $C \in \mathbb{R}^{m \times n}$. These matrices will be partitioned into rows, columns, and elements using the conventions discussed in Section 3.1.

5.2.1 Definition

The reader is likely familiar with the matrix-matrix product operation. Nonetheless, it is our experience that it is useful to review *why* the matrix-matrix product is defined as it is.

Like the matrix-vector product, the matrix-matrix product is related to the properties of linear transformations. In particular, the matrix-matrix product AB equals the matrix that corresponds to the composition of the transformations represented by A and B.

We start by reviewing the definition of the composition of two transformations.

Definition 5.2 *Consider two linear transformations $\mathcal{F} : \mathbb{R}^n \rightarrow \mathbb{R}^k$ and $\mathcal{G} : \mathbb{R}^k \rightarrow \mathbb{R}^m$. Then the composition of these transformations $(\mathcal{G} \circ \mathcal{F}) : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is defined by $(\mathcal{G} \circ \mathcal{F})(x) = \mathcal{G}(\mathcal{F}(x))$ for all $x \in \mathbb{R}^n$.*

The next theorem shows that if both $\mathcal{G}$ and $\mathcal{F}$ are linear transformations then so is their composition.

Theorem 5.3 *Consider two linear transformations $\mathcal{F} : \mathbb{R}^n \rightarrow \mathbb{R}^k$ and $\mathcal{G} : \mathbb{R}^k \rightarrow \mathbb{R}^m$. Then $(\mathcal{G} \circ \mathcal{F})$ is also a linear transformation.*

Proof: Let $\alpha \in \mathbb{R}$ and $x, y \in \mathbb{R}^n$. Then

$$(\mathcal{G} \circ \mathcal{F})(\alpha x + y) = \mathcal{G}(\mathcal{F}(\alpha x + y)) = \mathcal{G}(\alpha \mathcal{F}(x) + \mathcal{F}(y)) = \alpha \mathcal{G}(\mathcal{F}(x)) + \mathcal{G}(\mathcal{F}(y)) = \alpha (\mathcal{G} \circ \mathcal{F})(x) + (\mathcal{G} \circ \mathcal{F})(y).$$

!

With these observations we are ready to relate the composition of linear transformations to the matrix-matrix product.

Assume A and B equal the matrices that correspond to the linear transformations $\mathcal{G}$ and $\mathcal{F}$, respectively. Since $(\mathcal{G} \circ \mathcal{F})$ is also a linear transformation, there exists a matrix C so that $Cx = (\mathcal{G} \circ \mathcal{F})(x)$ for all $x \in \mathbb{R}^n$. The question now becomes how C relates to A and B. The key is the observation that $Cx = (\mathcal{G} \circ \mathcal{F})(x) = \mathcal{G}(\mathcal{F}(x)) = A(Bx)$, by the definition of composition and the relation between the matrix-vector product and linear transformations.

Let $e_i, e_j \in \mathbb{R}^n$ denote, respectively, the ith, jth unit basis vector. This observation defines the jth column of C as follows

$$c_j = Ce_j = (\mathcal{G} \circ \mathcal{F})(e_j) = A(Be_j) = Ab_j = \begin{pmatrix} \check{a}_0^T \\ \check{a}_1^T \\ \vdots \\ \check{a}_{m-1}^T \end{pmatrix} b_j = \begin{pmatrix} \check{a}_0^T b_j \\ \check{a}_1^T b_j \\ \vdots \\ \check{a}_{m-1}^T b_j \end{pmatrix}, \tag{5.1}$$

so that the (i,j) element of C is given by

$$\gamma_{i,j} = e_i^T c_j = \check{a}_i^T b_j = \sum_{p=0}^{k-1} \alpha_{i,p}\beta_{p,j}. \tag{5.2}$$

Remark 5.4 *Given $x \in \mathbb{R}^n$, by definition $ABx = (AB)x = A(Bx)$.*

Exercise 5.5 *Show that the cost of computing the matrix-matrix product is $2mnk$ flops.*

Exercise 5.6 *Show that the ith row of C is given by $\check{c}_i^T = \check{a}_i^T B$.*

Exercise 5.7 *Show that $A(BC) = (AB)C$. (This motivates the fact that no parenthesis are needed when more than two matrices are multiplied together: $ABC = A(BC) = (AB)C$.)*

5.2.2 Multiplying partitioned matrices

The following theorem will become a central result in the derivation of blocked algorithms for the matrix-matrix product.

Theorem 5.8 *Partition*

$$A \rightarrow \begin{pmatrix} A_{00} & A_{01} & \dots & A_{0,\kappa-1} \\ A_{10} & A_{11} & \dots & A_{1,\kappa-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{\mu-1,0} & A_{\mu-1,1} & \dots & A_{\mu-1,\kappa-1} \end{pmatrix}, B \rightarrow \begin{pmatrix} B_{00} & B_{01} & \dots & B_{0,\nu-1} \\ B_{10} & B_{11} & \dots & B_{1,\nu-1} \\ \vdots & \vdots & \ddots & \vdots \\ B_{\kappa-1,0} & B_{\kappa-1,1} & \dots & B_{\kappa-1,\nu-1} \end{pmatrix}, \quad \textit{and}$$

$$C \rightarrow \begin{pmatrix} C_{00} & C_{01} & \dots & C_{0,\nu-1} \\ C_{10} & C_{11} & \dots & C_{1,\nu-1} \\ \vdots & \vdots & \ddots & \vdots \\ C_{\mu-1,0} & C_{\mu-1,1} & \dots & C_{\mu-1,\nu-1} \end{pmatrix},$$

where $A_{i,p} \in \mathbb{R}^{m_i \times k_p}$, $B_{p,j} \in \mathbb{R}^{k_p \times n_j}$, and $C_{i,j} \in \mathbb{R}^{m_i \times n_j}$. Then, the (i,j) block of $C = AB$ is given by

$$C_{i,j} = \sum_{p=0}^{\kappa-1} A_{i,p}B_{p,j}. \tag{5.3}$$

The proof of this theorem is tedious. We therefore resort to Exercise 5.9 to demonstrate why it is true without giving a rigorous proof in this text.

Exercise 5.9 *Show that*

$$\left(\begin{array}{cc|c} 1 & -1 & 3 \\ 2 & 0 & -1 \\ \hline -1 & 2 & 1 \\ 0 & 1 & 2 \end{array}\right)\left(\begin{array}{cc|c} -1 & 0 & 2 \\ 1 & -1 & 1 \\ \hline 2 & 1 & -1 \end{array}\right) = \left(\begin{array}{c|c} \begin{pmatrix} 1 & -1 \\ 2 & 0 \end{pmatrix}\begin{pmatrix} -1 & 0 \\ 1 & -1 \end{pmatrix} + \begin{pmatrix} 3 \\ -1 \end{pmatrix}\begin{pmatrix} 2 & 1 \end{pmatrix} & \begin{pmatrix} 1 & -1 \\ 2 & 0 \end{pmatrix}\begin{pmatrix} 2 \\ 1 \end{pmatrix} + \begin{pmatrix} 3 \\ -1 \end{pmatrix}\begin{pmatrix} -1 \end{pmatrix} \\ \hline \begin{pmatrix} -1 & 2 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} -1 & 0 \\ 1 & -1 \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \end{pmatrix}\begin{pmatrix} 2 & 1 \end{pmatrix} & \begin{pmatrix} -1 & 2 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 2 \\ 1 \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \end{pmatrix}\begin{pmatrix} -1 \end{pmatrix} \end{array}\right)$$

Remark 5.10 *Multiplying two partitioned matrices is exactly like multiplying two matrices with scalar elements, but with the individual elements replaced by submatrices. However, since the product of matrices does not commute, the order of the submatrices of A and B in the product is important: While $\alpha_{i,p}\beta_{p,j} = \beta_{p,j}\alpha_{i,p}$, $A_{i,p}B_{p,j}$ is generally not the same as $B_{p,j}A_{i,p}$. Also, the partitioning of A, B, and C must be conformal: $m(A_{i,p}) = m(C_{i,j})$, $n(B_{p,j}) = n(C_{i,j})$, and $n(A_{i,p}) = m(B_{p,j})$, for $0 \le i < \mu$, $0 \le j < \nu$, $0 \le p < \kappa$.*

5.2.3 Shapes of GEMM

It is convenient to view GEMM as a collection of special cases, based on the dimensions (shape) of the operands A, B, and C. In Figure 5.3 we show the different shapes that will be encountered in our algorithms, and label them for future reference. Special cases are labeled as GEXY, where the first two letters, "GE", stand for *general* and refer to the fact that matrices A, B, and C have no special structure. The next pair of letters, "X" and "Y", stand for the shape (determined by the dimensions m, n, and k) of matrices A and B, respectively, and follow the convention in Figure 5.4. The exception to this naming convention is GEPDOT, for which both dimensions of C are small so that the operation resembles a dot product with matrices, and GER which has already appeared.

Remark 5.11 *In the next section we will see that "small" is linked to a dimension a block of a matrix that fits in the cache of the target architecture.*

5.3 Algorithms for GEMM

Equations (5.1), (5.2), and Exercise 5.6 show that there are a number of different ways of computing GEMM. In this section we formulate three unblocked algorithms for computing GEMM and their corresponding blocked counterparts by performing different partitionings of the matrices. The algorithms are obtained by only employing horizontal/vertical partitionings of two of the three matrices of the problem in each case.

The precondition for all the algorithms is

$$P_{pre} : (A \in \mathbb{R}^{m\times k}) \wedge (B \in \mathbb{R}^{k\times n}) \wedge (C \in \mathbb{R}^{m\times n}),$$

while the postcondition is given by

$$P_{post} : C = AB + \hat{C}.$$

m	n	k	Illustration	Label
large	large	large	:= +	GEMM
large	large	1	:= +	GER
large	large	small	:= +	GEPP
large	1	large	:= +	GEMV
large	small	large	:= +	GEMP
1	large	large	:= +	GEVM
small	large	large	:= +	GEPM
large	small	small	:= +	GEPB
small	large	small	:= +	GEBP
small	small	large	:= +	GEPDOT

Figure 5.3: Special shapes of GEMM.

Letter	**Shape**	**Description**
M	Matrix	Two dimensions are large or unknown.
P	Panel	One of the dimensions is small.
B	Block	Two dimensions are small.
V	Vector	One of the dimensions equals one.

Figure 5.4: Naming convention for the shape of matrices involved in GEMM.

5.3.1 Implementing GEMM with GER or GEPP

We start by noting that

$$C := AB + C = (a_0, a_1, \ldots, a_{k-1}) \begin{pmatrix} \check{b}_0^{\mathrm{T}} \\ \check{b}_1^{\mathrm{T}} \\ \vdots \\ \check{b}_{k-1}^{\mathrm{T}} \end{pmatrix} + C = a_0\check{b}_0^{\mathrm{T}} + a_1\check{b}_1^{\mathrm{T}} + \cdots + a_{k-1}\check{b}_{k-1}^{\mathrm{T}} + C, \tag{5.4}$$

which shows that GEMM can be computed as a series of GERs. In line with Remark 5.10, this can be viewed as partitioning A by columns, B by rows, and updating the matrix C *as if* A and B were row and column vectors (of symbols), respectively, and AB the dot product of these two vectors.

Exercise 5.12 *Show that*

$$\left(\begin{array}{r|r|r} 1 & -1 & 3 \\ 2 & 0 & -1 \\ -1 & 2 & 1 \\ 0 & 1 & 2 \end{array}\right) \left(\begin{array}{rrr} -1 & 0 & 2 \\ \hline 1 & -1 & 1 \\ \hline 2 & 1 & -1 \end{array}\right) = \begin{pmatrix} 1 \\ 2 \\ -1 \\ 0 \end{pmatrix} \begin{pmatrix} -1 & 0 & 2 \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \\ 2 \\ 1 \end{pmatrix} \begin{pmatrix} 1 & -1 & 1 \end{pmatrix} + \begin{pmatrix} 3 \\ -1 \\ 1 \\ 2 \end{pmatrix} \begin{pmatrix} 2 & 1 & -1 \end{pmatrix}.$$

The following corollary of Theorem 5.8 yields a PME for deriving the first unblocked and blocked variants for GEMM.

Corollary 5.13 *Partition*

$$A \rightarrow \left(A_L \,\middle|\, A_R \right) \quad \textit{and} \quad B \rightarrow \left(\frac{B_T}{B_B}\right),$$

where $n(A_L) = m(B_T)$. *Then,*

$$AB + C = \left(A_L \,\middle|\, A_R \right) \left(\frac{B_T}{B_B}\right) + C = A_L B_T + A_R B_B + C.$$

Using the partitionings defined in the corollary and the loop-invariant

$$P_{inv} : (C = A_L B_T + \hat{C}) \wedge P_{cons},$$

with "$P_{cons} : n(A_L) = m(B_T)$", we obtain the two algorithms for GEMM in Figure 5.5. The unblocked variant there performs all its computations in terms of GER, while the corresponding blocked variant operates on A_1 of k_b columns and B_1 of k_b rows, with $k_b \ll m, n$ in general, and therefore utilizes a generalization of GER, namely GEPP (see Table 5.3).

5.3.2 Implementing GEMM with GEMV or GEMP

Alternatively, $C := AB + \hat{C}$ can be computed by columns as

$$(c_0, c_1, \ldots, c_{n-1}) := \underbrace{A\,(b_0, b_1, \ldots, b_{n-1})}_{\substack{\text{"scalar product"} \\ \text{with symbols}}} + (\hat{c}_0, \hat{c}_1, \ldots, \hat{c}_{n-1}) = (Ab_0 + \hat{c}_0, Ab_1 + \hat{c}_1, \ldots, Ab_{n-1} + \hat{c}_{n-1}). \tag{5.5}$$

Algorithm: $C :=$ GEMM_UNB_VAR1(A,B,C)	**Algorithm:** $C :=$ GEMM_BLK_VAR1(A,B,C)
Partition $A \rightarrow \left(A_L \,\|\, A_R \right)$, $B \rightarrow \left(\frac{B_T}{B_B} \right)$ **where** A_L has 0 columns and B_T has 0 rows	**Partition** $A \rightarrow \left(A_L \,\|\, A_R \right)$, $B \rightarrow \left(\frac{B_T}{B_B} \right)$ **where** A_L has 0 columns and B_T has 0 rows
while $n(A_L) < n(A)$ **do**	**while** $n(A_L) < n(A)$ **do**
	Determine block size k_b
Repartition $\left(A_L \,\|\, A_R \right) \rightarrow \left(A_0 \,\|\, a_1 \,\|\, A_2 \right)$, $\left(\frac{B_T}{B_B} \right) \rightarrow \begin{pmatrix} B_0 \\ b_1^T \\ B_2 \end{pmatrix}$ **where** a_1 is a column and b_1^T is a row	**Repartition** $\left(A_L \,\|\, A_R \right) \rightarrow \left(A_0 \,\|\, A_1 \,\|\, A_2 \right)$, $\left(\frac{B_T}{B_B} \right) \rightarrow \begin{pmatrix} B_0 \\ B_1 \\ B_2 \end{pmatrix}$ **where** A_1 has k_b columns and B_1 has k_b rows
$C := a_1 b_1^T + C$ (GER)	$C := A_1 B_1 + C$ (GEPP)
Continue with $\left(A_L \,\|\, A_R \right) \leftarrow \left(A_0 \,\|\, a_1 \,\|\, A_2 \right)$, $\left(\frac{B_T}{B_B} \right) \leftarrow \begin{pmatrix} B_0 \\ b_1^T \\ B_2 \end{pmatrix}$	**Continue with** $\left(A_L \,\|\, A_R \right) \leftarrow \left(A_0 \,\|\, A_1 \,\|\, A_2 \right)$, $\left(\frac{B_T}{B_B} \right) \leftarrow \begin{pmatrix} B_0 \\ B_1 \\ B_2 \end{pmatrix}$
endwhile	**endwhile**

Figure 5.5: Left: GEMM implemented as a sequence of GER operations (unblocked Variant 1). Right: GEMM implemented as a sequence of GEPP operations (blocked Variant 1).

That is, each column of C is obtained from a GEMV of A and the corresponding column of B. In line with Remark 5.10, this can be viewed as partitioning B and C by columns and updating C *as if* A were a scalar and B and C were row vectors.

Exercise 5.14 *Show that*

$$\begin{pmatrix} 1 & -1 & 3 \\ 2 & 0 & -1 \\ -1 & 2 & 1 \\ 0 & 1 & 2 \end{pmatrix} \left(\begin{array}{r|r|r} -1 & 0 & 2 \\ 1 & -1 & 1 \\ 2 & 1 & -1 \end{array} \right) = \left(\begin{pmatrix} 1 & -1 & 3 \\ 2 & 0 & -1 \\ -1 & 2 & 1 \\ 0 & 1 & 2 \end{pmatrix} \begin{pmatrix} -1 \\ 1 \\ 2 \end{pmatrix} \;\middle|\; \begin{pmatrix} 1 & -1 & 3 \\ 2 & 0 & -1 \\ -1 & 2 & 1 \\ 0 & 1 & 2 \end{pmatrix} \begin{pmatrix} 0 \\ -1 \\ 1 \end{pmatrix} \;\middle|\; \begin{pmatrix} 1 & -1 & 3 \\ 2 & 0 & -1 \\ -1 & 2 & 1 \\ 0 & 1 & 2 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix} \right).$$

The following corollary of Theorem 5.8 yields a second PME from which another unblocked and a blocked variant for GEMM can be derived.

Corollary 5.15 *Partition*

$$B \rightarrow \left(B_L \,\|\, B_R \right) \quad \textit{and} \quad C \rightarrow \left(C_L \,\|\, C_R \right),$$

where $n(B_L) = n(C_L)$. *Then*

$$AB + C = A \left(B_L \,\|\, B_R \right) + \left(C_L \,\|\, C_R \right) = \left(AB_L + C_L \,\|\, AB_R + C_R \right).$$

Algorithm: $C := \text{GEMM_UNB_VAR2}(A, B, C)$	**Algorithm:** $C := \text{GEMM_BLK_VAR2}(A, B, C)$
Partition $B \rightarrow \left(B_L \,\|\, B_R \right)$, $C \rightarrow \left(C_L \,\|\, C_R \right)$ **where** B_L and C_L have 0 columns	**Partition** $B \rightarrow \left(B_L \,\|\, B_R \right)$, $C \rightarrow \left(C_L \,\|\, C_R \right)$ **where** B_L and C_L have 0 columns
while $n(B_L) < n(B)$ **do**	**while** $n(B_L) < n(B)$ **do**
	Determine block size n_b
Repartition $\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, b_1 \,\|\, B_2 \right)$, $\left(C_L \,\|\, C_R \right) \rightarrow \left(C_0 \,\|\, c_1 \,\|\, C_2 \right)$ **where** b_1 and c_1 are columns	**Repartition** $\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, B_1 \,\|\, B_2 \right)$, $\left(C_L \,\|\, C_R \right) \rightarrow \left(C_0 \,\|\, C_1 \,\|\, C_2 \right)$ **where** B_1 and C_1 have n_b columns
$c_1 := Ab_1 + c_1$ (GEMV)	$C_1 := AB_1 + C_1$ (GEMP)
Continue with $\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,\|\, b_1 \,\|\, B_2 \right)$, $\left(C_L \,\|\, C_R \right) \leftarrow \left(C_0 \,\|\, c_1 \,\|\, C_2 \right)$	**Continue with** $\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,\|\, B_1 \,\|\, B_2 \right)$, $\left(C_L \,\|\, C_R \right) \leftarrow \left(C_0 \,\|\, C_1 \,\|\, C_2 \right)$
endwhile	**endwhile**

Figure 5.6: Left: GEMM implemented as a sequence of GEMV operations (unblocked Variant 2). Right: GEMM implemented as a sequence of GEMP operations (blocked Variant 2).

Using the partitionings in the corollary and the loop-invariant

$$P_{inv} : (\left(C_L \;\middle|\; C_R \right) = \left(AB_L + \hat{C}_L \;\middle|\; \hat{C}_R \right)) \wedge P_{cons}$$

with "$P_{cons} : n(B_L) = n(C_L)$, we arrive at the algorithms for GEMM in Figure 5.6. The unblocked variant in this case is composed of GEMV operations, and the blocked variant of GEMP as, in this case, (see Table 5.3) both B_1 and C_1 are "narrow" blocks of columns (panels) with only n_b columns, $n_b \ll m, k$.

5.3.3 Implementing GEMM with GEVM or GEPM

Finally, we recall that the ith row of AB is given by

$$e_i^T(AB) = (e_i^T A)B = \check{a}_i^T B.$$

Let us denote the original contents of the rows of C by $\check{c}_i^{\mathrm{T}}$[2]. We can formulate GEMM as partitioning A and C by rows and computing $C := AB + \hat{C}$ *as if* B were a scalar and A and C were column vectors (of symbols):

$$\left(\begin{array}{c} \check{c}_0^{\mathrm{T}} \\ \check{c}_1^{\mathrm{T}} \\ \vdots \\ \check{c}_{m-1}^{\mathrm{T}} \end{array}\right) := \underbrace{\left(\begin{array}{c} \check{a}_0^{\mathrm{T}} \\ \check{a}_1^{\mathrm{T}} \\ \vdots \\ \check{a}_{m-1}^{\mathrm{T}} \end{array}\right) B}_{\substack{\text{“scalar product”} \\ \text{with symbols}}} + \left(\begin{array}{c} \check{c}_0^{\mathrm{T}} \\ \check{c}_1^{\mathrm{T}} \\ \vdots \\ \check{c}_{m-1}^{\mathrm{T}} \end{array}\right) = \left(\begin{array}{c} \check{a}_0^{\mathrm{T}} B + \check{c}_0^{\mathrm{T}} \\ \check{a}_1^{\mathrm{T}} B + \check{c}_1^{\mathrm{T}} \\ \vdots \\ \check{a}_{m-1}^{\mathrm{T}} + \check{c}_{m-1}^{\mathrm{T}} \end{array}\right). \tag{5.6}$$

In this case, the product is composed of GEVM operations.

Exercise 5.16 *Show that*

$$\left(\begin{array}{ccc} 1 & -1 & 3 \\ \hline 2 & 0 & -1 \\ \hline -1 & 2 & 1 \\ \hline 0 & 1 & 2 \end{array}\right) \left(\begin{array}{ccc} -1 & 0 & 2 \\ 1 & -1 & 1 \\ 2 & 1 & -1 \end{array}\right) = \left(\begin{array}{c} \left(\begin{array}{ccc} 1 & -1 & 3 \end{array}\right) \left(\begin{array}{ccc} -1 & 0 & 2 \\ 1 & -1 & 1 \\ 2 & 1 & -1 \end{array}\right) \\ \hline \left(\begin{array}{ccc} 2 & 0 & -1 \end{array}\right) \left(\begin{array}{ccc} -1 & 0 & 2 \\ 1 & -1 & 1 \\ 2 & 1 & -1 \end{array}\right) \\ \hline \left(\begin{array}{ccc} -1 & 2 & 1 \end{array}\right) \left(\begin{array}{ccc} -1 & 0 & 2 \\ 1 & -1 & 1 \\ 2 & 1 & -1 \end{array}\right) \\ \hline \left(\begin{array}{ccc} 0 & 1 & 2 \end{array}\right) \left(\begin{array}{ccc} -1 & 0 & 2 \\ 1 & -1 & 1 \\ 2 & 1 & -1 \end{array}\right) \end{array}\right).$$

Finally, the following corollary of Theorem 5.8 yields a third PME from which another unblocked and a blocked variant for GEMMcan be derived.

Corollary 5.17 *Partition*

$$A \rightarrow \left(\frac{A_T}{A_B}\right) \quad \textit{and} \quad C \rightarrow \left(\frac{C_T}{C_B}\right),$$

where $m(A_T) = m(C_T)$. *Then*

$$AB + C = \left(\frac{A_T}{A_B}\right) B + \left(\frac{C_T}{C_B}\right) = \left(\frac{A_T B + C_T}{A_B B + C_B}\right).$$

Using the partitionings in the corollary and the loop-invariant

$$P_{inv} : \left(\left(\frac{C_T}{C_B}\right) = \left(\frac{A_T B + \hat{C}_T}{\hat{C}_B}\right)\right) \wedge P_{cons},$$

[2] We deviate here from our usual notation as, in this case, we would have had to use a double superscript to denote the original contents of $\check{c}_i^{\mathrm{T}}$.

Algorithm: $C := \text{GEMM_UNB_VAR3}(A, B, C)$

Partition $A \rightarrow \left(\frac{A_T}{A_B}\right)$, $C \rightarrow \left(\frac{C_T}{C_B}\right)$
  where A_T and C_T have 0 rows
while $m(A_T) < m(A)$ **do**

  Repartition
  $$\left(\frac{A_T}{A_B}\right) \rightarrow \begin{pmatrix} A_0 \\ \hline a_1^T \\ \hline A_2 \end{pmatrix}, \left(\frac{C_T}{C_B}\right) \rightarrow \begin{pmatrix} C_0 \\ \hline c_1^T \\ \hline C_2 \end{pmatrix}$$
  where a_1^T and c_1^T are rows

  $c_1^T := a_1^T B + c_1^T$ (GEVM)

  Continue with
  $$\left(\frac{A_T}{A_B}\right) \leftarrow \begin{pmatrix} A_0 \\ \hline a_1^T \\ \hline A_2 \end{pmatrix}, \left(\frac{C_T}{C_B}\right) \leftarrow \begin{pmatrix} C_0 \\ \hline c_1^T \\ \hline C_2 \end{pmatrix}$$
endwhile

Algorithm: $C := \text{GEMM_BLK_VAR3}(A, B, C)$

Partition $A \rightarrow \left(\frac{A_T}{A_B}\right)$, $C \rightarrow \left(\frac{C_T}{C_B}\right)$
  where A_T and C_T have 0 rows
while $m(A_T) < m(A)$ **do**
  Determine block size m_b
  Repartition
  $$\left(\frac{A_T}{A_B}\right) \rightarrow \begin{pmatrix} A_0 \\ \hline A_1 \\ \hline A_2 \end{pmatrix}, \left(\frac{C_T}{C_B}\right) \rightarrow \begin{pmatrix} C_0 \\ \hline C_1 \\ \hline C_2 \end{pmatrix}$$
  where A_1 and C_1 have m_b rows

  $C_1 := A_1 B + C_1$ (GEPM)

  Continue with
  $$\left(\frac{A_T}{A_B}\right) \leftarrow \begin{pmatrix} A_0 \\ \hline A_1 \\ \hline A_2 \end{pmatrix}, \left(\frac{C_T}{C_B}\right) \leftarrow \begin{pmatrix} C_0 \\ \hline C_1 \\ \hline C_2 \end{pmatrix}$$
endwhile

Figure 5.7: Left: GEMM implemented as a sequence of GEVM operations (unblocked Variant 3). Right: GEMM implemented as a sequence of GEPM operations (blocked Variant 3).

with "$P_{cons} : m(A_T) = m(C_T)$", we obtain the algorithms for GEMM in Figure 5.7. The unblocked algorithm there consists of GEVM operations, and the blocked variant of the corresponding generalization in the form of GEPM as now A_1 and C_1 are blocks of m_b rows and $m_b \ll n, k$ (see Table 5.3).

Remark 5.18 *The blocked variants in Figures 5.5, 5.6, and 5.7 compute* GEMM *in terms of* GEPP, GEMP, *and* GEPM *operations, respectively. In Section 5.4 it will be shown how these three operations can be implemented to achieve high performance. As a consequence, the blocked algorithms for* GEMM *based on such operations also achieve high performance.*

5.3.4 Performance

Performance of the algorithms presented in Figures 5.5–5.7 on an Intel Pentium 4@2.8GHz is given in Figure 5.8. (See Section 1.5 for details on the architecture, compiler, etc.) The line labeled as "Simple" refers to a traditional implementation of GEMM, consisting of three nested loops fully optimized by the compiler. Highly optimized implementations of the GER, GEMV, and GEVM operations were called by the GEMM_UNB_VAR1, GEMM_UNB_VAR2, and GEMM_UNB_VAR3 implementations. Similarly, highly optimized implementations for the GEPP, GEMP, and GEPM operations were called by implementations of the blocked algorithms, which used a

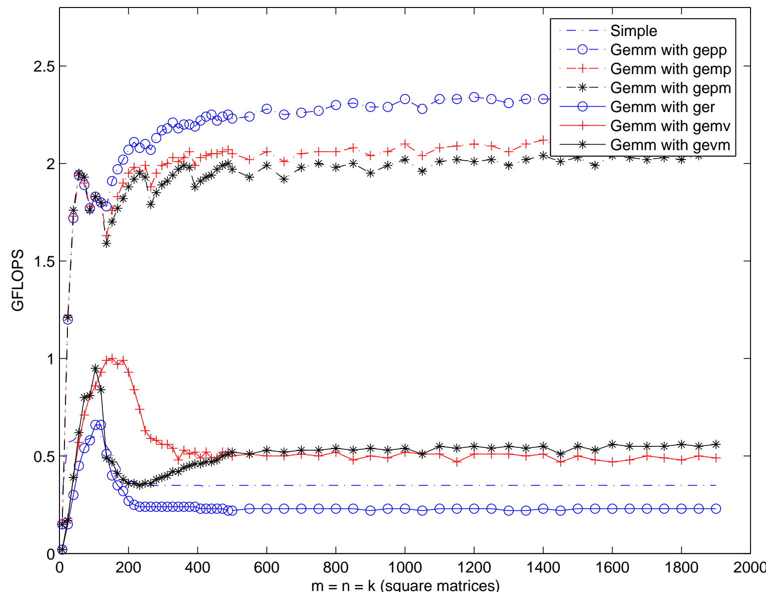

Figure 5.8: Performance of the implementations of the algorithms in Figures 5.5–5.7.

block size of $k_b = n_b = m_b = 128$. All three matrices were taken to be square.

Some observations:

- Among the unblocked algorithms, the one that utilizes GER is the slowest. This is not surprising: it generates twice as much data traffic between main memory and the registers as either of the other two.
- The "hump" in the performance of the unblocked algorithms coincides with problem sizes that fit in the cache. For illustration, consider the GEMV-based implementation in GEMM_UNB_VAR2. Performance of the code is reasonable when the dimensions are relatively small since from one GEMV to the next matrix A then remains in the cache.
- The blocked algorithms attain a substantial percentage of peak performance (2.8 GFLOPS), that ranges between 70 and 85%.
- Among the blocked algorithms, the variant that utilizes GEPP is the fastest. On almost all architectures at this writing high performance implementations of GEPP outperform those of GEMP and GEPM. The reasons for this go beyond the scope of this text.

- On different architectures, the relative performance of the algorithms may be quite different. However, blocked algorithms invariably outperform unblocked ones.

5.4 High-Performance Implementation of GEPP, GEMP, and GEPM

In the previous section, we saw that *if* fast implementations of GEPP, GEMP, and GEPM are available, then high performance implementations of GEMM can be built upon these. In this section we show *how* fast implementations of these operations can be obtained.

5.4.1 The basic building blocks: GEBP, GEPB, and GEPDOT

Consider the matrix-matrix product operation with $A \in \mathbb{R}^{m_c \times k_c}$, $B \in \mathbb{R}^{k_c \times n}$, and $C \in \mathbb{R}^{m_c \times n}$. The product then has the GEBP shape:

$$\boxed{\quad C \quad} := \boxed{A}\ \boxed{\quad B \quad} + \boxed{\quad C \quad}$$

Furthermore, assume that

- The dimensions m_c, k_c are small enough so that A, a column from B, and a column from C together fit in the cache.
- If A and the two columns are in the cache then GEMV can be computed at the peak rate of the CPU.
- If A is in the cache it remains there until no longer needed.

Under these assumptions, the approach to implementing algorithm GEBP in Figure 5.9 amortizes the cost of moving data between the main memory and the cache as follows. The total cost of updating C is $m_c k_c + (2m_c + k_c)n$ memops for $2m_c k_c n$ flops. Now, let $c = m_c \approx k_c$. Then, the ratio between computation and data movement is

$$\frac{2c^2 n}{c^2 + 3cn} \frac{\text{flops}}{\text{memops}} \approx \frac{2c}{3} \frac{\text{flops}}{\text{memops}} \text{ when } c \ll n.$$

If $c \approx n/100$ then even if memops are 10 times slower than flops, the memops add only about 10% overhead to the computation.

We note the similarity between algorithm GEBP_UNB_VAR2 in Figure 5.9 and the unblocked Variant 2 for GEMM in Figure 5.6 (right).

> **Remark 5.19** *In the highest-performance implementations of* GEBP*, both A and B are typically copied into a contiguous buffer and/or transposed. For complete details on this observation, see [13].*

Algorithm: $C :=$ GEBP_UNB_VAR2(A, B, C)

Partition $B \rightarrow \left(B_L \,\middle|\, B_R \right)$, $C \rightarrow \left(C_L \,\middle|\, C_R \right)$
  where B_L and C_L have 0 columns

while $n(B_L) < n(B)$ **do**

  Repartition
    $\left(B_L \,\middle|\, B_R \right) \rightarrow \left(B_0 \,\middle|\, b_1 \,\middle|\, B_2 \right)$, $\left(C_L \,\middle|\, C_R \right) \rightarrow \left(C_0 \,\middle|\, c_1 \,\middle|\, C_2 \right)$
    where b_1 and c_1 are columns

  Only 1st iteration: Load A into the cache (cost: $m_c k_c$ memops)
  Load b_1 into the cache (cost: k_c memops)
  Load c_1 into the cache (cost: m_c memops)
  $c_1 := Ab_1 + c_1$(cost: $2m_c k_c$ memops)
  Store c_1 into the memory (cost: m_c memops)

  Continue with
    $\left(B_L \,\middle|\, B_R \right) \leftarrow \left(B_0 \,\middle|\, b_1 \,\middle|\, B_2 \right)$, $\left(C_L \,\middle|\, C_R \right) \leftarrow \left(C_0 \,\middle|\, c_1 \,\middle|\, C_2 \right)$

endwhile

Figure 5.9: GEBP implemented as a sequence of GEMV, with indication of the memops and flops costs. Note that a program typically has no explicit control over the loading of a cache. Instead it is in using the data that the cache is loaded, and by carefully ordering the computation that the architecture is encouraged to keep data in the cache.

Algorithm: $C := \text{GEPP_BLK_VAR3}(A, B, C)$

Partition $A \rightarrow \left(\frac{A_T}{A_B}\right)$, $C \rightarrow \left(\frac{C_T}{C_B}\right)$
  where A_T and C_T have 0 rows
while $m(A_T) < m(A)$ **do**
  Determine block size m_b
  Repartition
$$\left(\frac{A_T}{A_B}\right) \rightarrow \begin{pmatrix} A_0 \\ \hline A_1 \\ \hline A_2 \end{pmatrix}, \left(\frac{C_T}{C_B}\right) \rightarrow \begin{pmatrix} C_0 \\ \hline C_1 \\ \hline C_2 \end{pmatrix}$$
  where A_1 and C_1 have b rows

  $C_1 := A_1 B + C_1$ (GEBP)

  Continue with
$$\left(\frac{A_T}{A_B}\right) \leftarrow \begin{pmatrix} A_0 \\ \hline A_1 \\ \hline A_2 \end{pmatrix}, \left(\frac{C_T}{C_B}\right) \leftarrow \begin{pmatrix} C_0 \\ \hline C_1 \\ \hline C_2 \end{pmatrix}$$
endwhile

Algorithm: $C := \text{GEMM_BLK_VAR3}(A, B, C)$

Partition $A \rightarrow \left(\frac{A_T}{A_B}\right)$, $C \rightarrow \left(\frac{C_T}{C_B}\right)$
  where A_T and C_T have 0 rows
while $m(A_T) < m(A)$ **do**
  Determine block size m_b
  Repartition
$$\left(\frac{A_T}{A_B}\right) \rightarrow \begin{pmatrix} A_0 \\ \hline A_1 \\ \hline A_2 \end{pmatrix}, \left(\frac{C_T}{C_B}\right) \rightarrow \begin{pmatrix} C_0 \\ \hline C_1 \\ \hline C_2 \end{pmatrix}$$
  where A_1 and C_1 have m_b rows

  $C_1 := A_1 B + C_1$ (GEPM)

  Continue with
$$\left(\frac{A_T}{A_B}\right) \leftarrow \begin{pmatrix} A_0 \\ \hline A_1 \\ \hline A_2 \end{pmatrix}, \left(\frac{C_T}{C_B}\right) \leftarrow \begin{pmatrix} C_0 \\ \hline C_1 \\ \hline C_2 \end{pmatrix}$$
endwhile

Figure 5.10: Left: GEPP implemented as a sequence of GEBP operations. Right: GEMM implemented as a sequence of GEPM operations.

Exercise 5.20 *Propose a similar scheme for the* GEPB *operation, where* $A \in \mathbb{R}^{m \times k_c}$, $B \in \mathbb{R}^{k_c \times n_c}$, *and* $C \in \mathbb{R}^{m \times n_c}$. *State your assumptions carefully. Analyze the ratio of flops to memops.*

Exercise 5.21 *Propose a similar scheme for the* GEPDOT *operation, where* $A \in \mathbb{R}^{m_c \times k}$, $B \in \mathbb{R}^{k \times n_c}$, *and* $C \in \mathbb{R}^{m_c \times n_c}$. *State your assumptions carefully. Analyze the ratio of flops to memops.*

5.4.2 Panel-panel multiply (GEPP)

Consider the GEPP operation $C := AB + C$, where $A \in \mathbb{R}^{m \times k_b}$, $B \in \mathbb{R}^{k_b \times n}$, and $C \in \mathbb{R}^{m \times n}$. By partitioning the matrices into two different directions, we will obtain two algorithms for this operation, based on GEBP or GEPB. We will review here the first variant while the second one is proposed as an exercise.

Assume m is an exact multiple of m_b and partition matrices A and C into blocks of m_b rows so that the product takes the form

$$\begin{pmatrix} C_0 \\ C_1 \\ \vdots \end{pmatrix} := \begin{pmatrix} A_0 \\ A_1 \\ \vdots \end{pmatrix} B + \begin{pmatrix} C_0 \\ C_1 \\ \vdots \end{pmatrix}$$

Then, each C_i can be computed as a GEBP of the form $C_i := A_i B + C_i$. Since it was argued that GEBP can attain high performance, provided $m_b = m_c$ and $k_b = k_c$, so can the GEPP.

Remark 5.22 *In the implementation of* GEPP *based on* GEBP *there is complete freedom to chose $m_b = m_c$. Also, k_b is usually set by the routine that invokes* GEPP *(e.g.,* GEMM_BLK_VAR1*), so that it can be chosen there as $k_b = k_c$. An analogous situation will occur for the alternative implementation of* GEPP *based on* GEPB*, and for all implementations of* GEMP *and* GEPM*.*

The algorithm for this is given in Figure 5.10 (left). For comparison, we repeat Variant 3 for computing GEMM in the same figure (right). The two algorithms are identical except that the constraint on the row dimension of A and column dimension of B changes the update from a GEPM to a GEBP operation.

Exercise 5.23 *For the* GEPP *operation assume n is an exact multiple of n_b, and partition B and C into blocks of n_b columns so that*

$$\begin{pmatrix} C_0 & C_1 & \cdots \end{pmatrix} := A \begin{pmatrix} B_0 & B_1 & \cdots \end{pmatrix} + \begin{pmatrix} C_0 & C_1 & \cdots \end{pmatrix}$$

Propose an alternative high-performance algorithm for computing GEPP *based on* GEPB*. Compare the resulting algorithm to the three variants for computing* GEMM*. Which variant does it match?*

5.4.3 Matrix-panel multiply (GEMP)

Consider again the GEMP computation $C := AB + C$, where now $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n_b}$, and $C \in \mathbb{R}^{m \times n_b}$. The two algorithms that are obtained in this case are based on GEPB and GEPDOT.

Assume k is an exact multiple of k_b, and partition A into blocks of k_b columns and B into blocks of k_b rows so that

$$C := \left(A_0 \mid A_1 \mid \cdots \right) \begin{pmatrix} B_0 \\ B_1 \\ \vdots \end{pmatrix} + C$$

Then, C can be computed as repeated updates of C with GEPB operations, $C := A_p B_p + C$. The algorithm is identical to GEMM_BLK_VAR1 in Figure 5.5 except that the update changes from a GEPP to a GEPB operation. If GEPB attains high performance, if $n_b = n_c$ and $k_b = k_c$, so will this algorithm for computing GEMP.

Exercise 5.24 *For the* GEMP *operation assume m is an exact multiple of m_b, and partition A and C by blocks of m_b rows as*

$$\begin{pmatrix} C_0 \\ C_1 \\ \vdots \end{pmatrix} := \begin{pmatrix} A_0 \\ A_1 \\ \vdots \end{pmatrix} B + \begin{pmatrix} C_0 \\ C_1 \\ \vdots \end{pmatrix}$$

Propose an alternative high-performance algorithm based on GEPDOT.

5.4.4 Panel-matrix multiply (GEPM)

Finally, consider once more the computation $C := AB + C$, except that now $A \in \mathbb{R}^{m_b \times k}$, $B \in \mathbb{R}^{k \times n}$, and $C \in \mathbb{R}^{m_b \times n}$. Again two algorithms are obtained in this case, based on GEPDOT and GEBP.

Assume that n is an exact multiple of n_b, so that a partitioning of B and C into blocks of n_b columns takes the form

$$\left(C_0 \mid C_1 \mid \cdots \right) := A \left(B_0 \mid B_1 \mid \cdots \right) + \left(C_0 \mid C_1 \mid \cdots \right)$$

Then each block of C can be computed as $C_j := AB_j + C$ using the GEPDOT operation. The algorithm is identical to GEMM_BLK_VAR2 in Figure 5.6 except that the update changes from a GEMP to a GEPDOT operation. If GEPDOT attains high performance, provided $m_b \approx m_c$ and $n_b = n_c$, so will this algorithm for computing GEPM.

Exercise 5.25 *For the* GEPM *operation assume k is an exact multiple of k_b, and partition A and B by blocks of k_b rows and columns, respectively, so that*

$$C := \left(\begin{array}{c|c|c} A_0 & A_1 & \cdots \end{array} \right) \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline \vdots \end{array} \right) + C$$

Propose an alternative high-performance algorithm based on GEBP.

5.4.5 Putting it all together

Figure 5.11 summarizes the insights discussed in Sections 5.3 and 5.4 regarding the implementation of GEMM. A GEMM can be implemented in terms of high-performance implementations of GEPP, GEMP, or GEPM. Each of these in turn can be implemented in terms of GEBP, GEPB, and/or GEPDOT.

Remark 5.26 *If one variant of matrix-matrix multiplication is used at one level, that same variant does* not *occur at the next level. There are theoretical reasons for this that go beyond the scope of this text. For details, see [17].*

5.5 Modularity and Performance via GEMM: Implementing SYMM

A key concept introduced in the section is how to achieve high performance by casting computation in terms of GEMM.

We start by reviewing an important structural property of matrices: symmetry.

Definition 5.27 *A square matrix is symmetric, denoted as $A = A^{\mathrm{T}}$, if its (i, j) element equals its (j, i) element.*

In this section we consider the special case of the matrix-matrix product

$$C := AB + C,$$

where $A \in \mathbb{R}^{m \times m}$ is symmetric and $C,\ B \in \mathbb{R}^{m \times n}$. (Thus, $k = m$.)

Remark 5.28 *Unless otherwise stated, we assume hereafter that it is the lower part of the symmetric matrix A, including the diagonal, that contains the relevant entries of the matrix. In our notation this is denoted as* $\mathrm{SyLw}(A)$*. The algorithms that are derived will not make any reference to the contents of the strictly upper triangular part (superdiagonals) of symmetric matrices.*

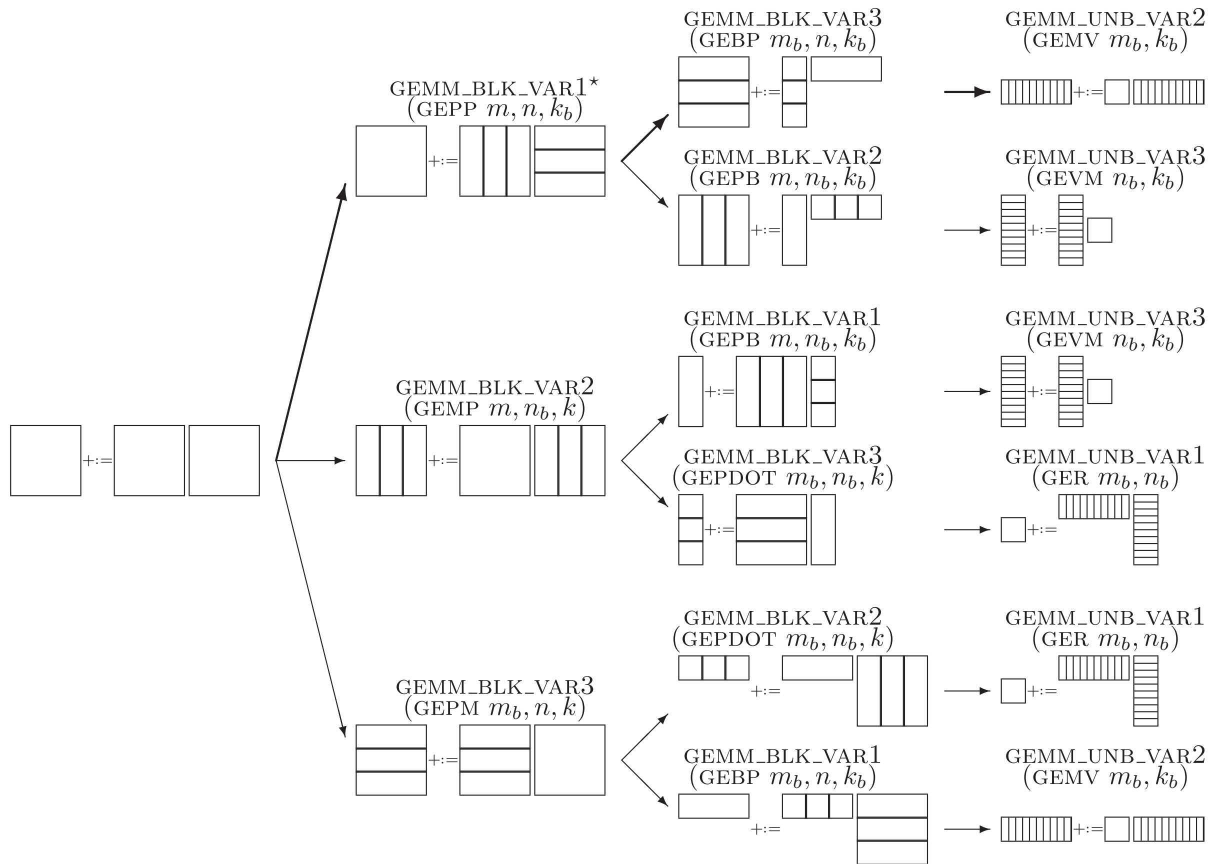

Figure 5.11: Implementations of GEMM. The legend on top of each figure indicates the algorithm that is invoked in that case, and (between parenthesis) the shape and dimensions of the subproblems the case is decomposed into. For instance, in the case marked with "⋆", the product is performed via algorithm GEMM_BLK_VAR1, which is then decomposed into matrix-matrix products of shape GEPP and dimensions m, n, and k_b.

When dealing with symmetric matrices, in general only the upper or lower part of the matrix is actually stored. One option is to copy the stored part of A into both the upper and lower triangular part of a temporary matrix and to the use GEMM. This is undesirable if A is large, since it requires temporary space.

The precondition for the symmetric matrix-matrix product, SYMM, is given by

$$P_{pre} : (A \in \mathbb{R}^{m \times m}) \wedge \text{SyLw}(A) \wedge (B, C \in \mathbb{R}^{m \times n}),$$

while the postcondition is that

$$P_{post} : C = AB + \hat{C}.$$

We next formulate a partitioning and a collection of loop-invariants that potentially yield algorithms for SYMM. Let us partition the symmetric matrix A into quadrants as

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{BL}^{\mathrm{T}} \\ \hline A_{BL} & A_{BR} \end{array} \right).$$

Then, from the postcondition, $C = AB + \hat{C}$, a consistent partitioning of matrices B and C is given by

$$B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array} \right), \quad C \rightarrow \left(\begin{array}{c} C_T \\ \hline C_B \end{array} \right),$$

where "$P_{cons} : n(A_{TL}) = m(B_T) \wedge m(A_{TL}) = m(C_T)$" holds. (A different possibility would be to also partition B and C into quadrants, a case that is proposed as an exercise at the end of this section.) Very much as what we do for triangular matrices, for symmetric matrices we also require the blocks in the diagonal to be square (and therefore) symmetric. Thus, in the previous partitioning of A we want that

$$P_{struct} : \text{SyLw}(A_{TL}) \wedge (m(A_{TL}) = n(A_{TL})) \wedge \text{SyLw}(A_{BR}) \wedge (m(A_{BR}) = n(A_{BR}))$$

holds. Indeed, because SyLw(A), it is sufficient to define

$$P_{struct} : \text{SyLw}(A_{TL}).$$

Remark 5.29 *When dealing with symmetric matrices, in order for the diagonal blocks that are exposed to be themselves symmetric, we always partition this type of matrices into* quadrants*, with square blocks in the diagonal.*

The PME is given by

$$\left(\begin{array}{c} C_T \\ \hline C_B \end{array} \right) = \left(\begin{array}{c|c} A_{TL} & A_{BL}^{\mathrm{T}} \\ \hline A_{BL} & A_{BR} \end{array} \right) \left(\begin{array}{c} B_T \\ \hline B_B \end{array} \right) + \left(\begin{array}{c} \hat{C}_T \\ \hline \hat{C}_B \end{array} \right),$$

which is equivalent to

$$\left(\begin{array}{c} C_T = A_{TL} B_T + A_{BL}^{\mathrm{T}} B_B + \hat{C}_T \\ \hline C_B = A_{BL} B_T + A_{BR} B_B + \hat{C}_B \end{array} \right).$$

Recall that loop-invariants result by assuming that some computation is yet to be performed.

A systematic enumeration of subresults, each of which is a *potential* loop-invariants, is given in Table 5.12. We are only interested in *feasible* loop-invariants:

Definition 5.30 *A feasible loop-invariant is a loop-invariant that yields a correct algorithm when the derivation methodology is applied. If a loop-invariant is not feasible, it is* infeasible.

In the column marked by "Comment" reasons are given why a loop-invariant is not feasible.

Among the feasible loop-invariants in Figure 5.12, we now choose

$$\left(\left(\frac{C_T}{C_B} \right) = \left(\frac{A_{TL} B_T + \hat{C}_T}{\hat{C}_B} \right) \right) \wedge P_{cons} \wedge P_{struct},$$

for the remainder of this section. This invariant yields the blocked algorithm in Figure 5.13. As part of the update, in this algorithm the symmetric matrix-matrix multiplication $A_{11}B_1$ needs to be computed (being a square block in the diagonal of A, $A_{11} = A_{11}^T$). In order to do so, we can apply an unblocked version of the algorithm which, of course, would not reference the strictly upper triangular part of A_{11}. The remaining two updates require the computation of two GEMMs, $A_{10}^T B_1$ and $A_{10}B_0$, and do not reference any block in the strictly upper triangular part of A either.

Exercise 5.31 *Show that the cost of the algorithm for* SYMM *in Figure 5.13 is* $2m^2n$ *flops.*

Exercise 5.32 *Derive a pair of blocked algorithms for computing* $C := AB + \hat{C}$*, with* SyLw(A)*, by partitioning all three matrices into quadrants and choosing two feasible loop-invariants found for this case.*

Exercise 5.33 *Derive a blocked algorithm for computing* $C := BA + \hat{C}$*, with* SyLw(A)*,* $A \in \mathbb{R}^{m\times m}$*,* $C, B \in \mathbb{R}^{n\times m}$*, by partitioning both* B *and* C *in a single dimension and choosing a feasible loop-invariant found for this case.*

5.5.1 Performance

Consider now m to be an exact multiple of m_b, $m = \mu m_b$. The algorithm in Figure 5.13 requires μ iterations, with $2m_b^2 n$ flops being performed as a symmetric matrix multiplication ($A_{11}B_1$) at each iteration, while the rest of the computations is in terms of two GEMMs ($A_{10}^T B_1$ and $A_{10}B_0$). The amount of computations carried out as symmetric matrix multiplications, $2m\, m_b\, n$ flops, is only a minor part of the total cost of the algorithm, $2m^2 n$ flops (provided $m_b \ll m$). Thus, given an efficient implementation of GEMM, high performance can be expected from this algorithm.

5.6 Summary

The highlights of this Chapter are:

- A high level description of the architectural features of a computer that affect high-performance implementation of the matrix-matrix multiply.

Computed?				$P_{inv} : \left(\frac{C_T}{C_B}\right) =$	Comment
$A_{TL}B_T$	$A_{BL}^T B_B$	$A_{BL}B_T$	$A_{BR}B_B$		
N	N	N	N	$\left(\frac{\hat{C}_T}{\hat{C}_B}\right)$	No loop-guard exists so that $P_{inv} \wedge \neg G \Rightarrow P_{post}$
Y	N	N	N	$\left(\frac{A_{TL}B_T + \hat{C}_T}{\hat{C}_B}\right)$	Variant 1 (Fig. 5.13)
N	Y	N	N	$\left(\frac{A_{BL}^T B_B + \hat{C}_T}{\hat{C}_B}\right)$	No loop-guard exists so that $P_{inv} \wedge \neg G \Rightarrow P_{post}$.
Y	Y	N	N	$\left(\frac{A_{TL}B_T + A_{BL}^T B_B + \hat{C}_T}{\hat{C}_B}\right)$	Variant 2
N	N	Y	N	$\left(\frac{\hat{C}_T}{A_{BL}B_T + \hat{C}_B}\right)$	No loop-guard exists so that $P_{inv} \wedge \neg G \Rightarrow P_{post}$.
Y	N	Y	N	$\left(\frac{A_{TL}B_T + \hat{C}_T}{A_{BL}B_T + \hat{C}_B}\right)$	Leads to an alternative algorithm. Variant 3
N	Y	Y	N	$\left(\frac{A_{BL}^T B_B + \hat{C}_T}{A_{BL}B_T + \hat{C}_B}\right)$	No loop-guard exists so that $P_{inv} \wedge \neg G \Rightarrow P_{post}$.
Y	Y	Y	N	$\left(\frac{A_{TL}B_T + A_{BL}^T B_B + \hat{C}_T}{A_{BL}B_T + \hat{C}_B}\right)$	Variant 4
N	N	N	Y	$\left(\frac{\hat{C}_T}{A_{BR}B_B + \hat{C}_B}\right)$	Variant 5
Y	N	N	Y	$\left(\frac{A_{TL}B_T + \hat{C}_T}{A_{BR}B_B + \hat{C}_B}\right)$	No simple initialization exists to achieve this state.
N	Y	N	Y	$\left(\frac{A_{BL}^T B_B + \hat{C}_T}{A_{BR}B_B + \hat{C}_B}\right)$	Variant 6
Y	Y	N	Y	$\left(\frac{A_{TL}B_T + A_{BL}^T B_B + \hat{C}_T}{A_{BR}B_B + \hat{C}_B}\right)$	No simple initialization exists to achieve this state.
N	N	Y	Y	$\left(\frac{\hat{C}_T}{A_{BL}B_T + A_{BR}B_B + \hat{C}_B}\right)$	Variant 7
Y	N	Y	Y	$\left(\frac{A_{TL}B_T + \hat{C}_T}{A_{BL}B_T + A_{BR}B_B + \hat{C}_B}\right)$	No simple initialization exists to achieve this state.
N	Y	Y	Y	$\left(\frac{A_{BL}^T B_B + \hat{C}_T}{A_{BL}B_T + A_{BR}B_B + \hat{C}_B}\right)$	Variant 8
Y	Y	Y	Y	$\left(\frac{A_{TL}B_T + A_{BL}^T B_B + \hat{C}_T}{A_{BL}B_T + A_{BR}B_B + \hat{C}_B}\right)$	No simple initialization exists to achieve this state.

Figure 5.12: Potential loop-invariants for $C := AB + C$, with SyLw(A), using the partitioning in Section 5.5. Potential invariants are derived from the PME by systematically including/excluding (Y/N) a term.

Algorithm: $C := \text{SYMM_BLK_VAR1}(A, B, C)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right)$, and $C \rightarrow \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right)$

 where A_{TL} is 0×0, and B_T, C_T have 0 rows

while $m(A_{TL}) < m(A)$ **do**

 Determine block size m_b

 Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right), \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline C_2 \end{array}\right)$$

 where A_{11} is $m_b \times m_b$, and B_1, C_1 have m_b rows

---

$C_0 := A_{10}^{\mathrm{T}} B_1 + C_0$

$C_1 := A_{10} B_0 + A_{11} B_1 + C_1$

 Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right), \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) \leftarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline C_2 \end{array}\right)$$

endwhile

Figure 5.13: Algorithm for computing $C := AB + C$, with SyLw(A) (blocked Variant 1).

- The hierarchical anatomy of the implementation of this operation that exploits the hierarchical organization of multilevel memories of current architectures.
- The very high performance that is attained by this particular operation.
- How to cast algorithms for linear algebra operations in terms matrix-matrix multiply.
- The modular high-performance that results.

A recurrent theme of this and subsequent chapters will be that blocked algorithms for all major linear algebra operations can be derived that cast most computations in terms of GEPP, GEMP, and GEPM. The block size for these is tied to the size of cache memory.

5.7 Other Matrix-Matrix Operations

A number of other commonly encountered matrix-matrix operations tabulated in Figure 5.14.

Name	Abbrev.	Operation	Cost (flops)	Comment
GEneral Matrix-Matrix multiplication	GEMM	$C := \alpha AB + \beta C$	$2mnk$	$C \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$
		$C := \alpha A^T B + \beta C$	$2mnk$	$C \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{k \times m}$, $B \in \mathbb{R}^{k \times n}$
		$C := \alpha AB^T + \beta C$	$2mnk$	$C \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{n \times k}$
		$C := \alpha A^T B^T + \beta C$	$2mnk$	$C \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{k \times m}$, $B \in \mathbb{R}^{n \times k}$
SYmmetric Matrix Matrix multiplication	SYMM	$C := \alpha AB + \beta C$	$2mn^2$	$C \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{m \times m}$ symmetric, stored in lower/upper triangular part
		$C := \alpha BA + \beta C$	$2mn^2$	$C \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{n \times n}$ symmetric, stored in lower/upper triangular part
TRiangular Matrix Matrix multiplication	TRMM	$B := LB, B := L^T B$ $B := UB, B := U^T B$	$m^2 n$	$B \in \mathbb{R}^{m \times n}$, $L \in \mathbb{R}^{m \times m}$ is lower triangular, $U \in \mathbb{R}^{m \times m}$ is upper triangular
		$B := BL, B := BL^T$ $B := BU, B := BU^T$	mn^2	$B \in \mathbb{R}^{m \times n}$, $L \in \mathbb{R}^{n \times n}$ is lower triangular, $U \in \mathbb{R}^{n \times n}$ is upper triangular
SYmmetric Rank-K update	SYRK	$C := \alpha AA^T + \beta C$	$n^2 k$	$A \in \mathbb{R}^{n \times k}$, $C \in \mathbb{R}^{n \times n}$ is symmetric, stored in lower/upper triangular part
		$C := \alpha A^T A + \beta C$	$n^2 k$	$A \in \mathbb{R}^{k \times n}$, $C \in \mathbb{R}^{n \times n}$ symmetric, stored in lower/upper triangular part
SYmmetric Rank-2K update	SYR2K	$C := \alpha(AB^T + BA^T) + \beta C$	$2n^2 k$	$A \in \mathbb{R}^{n \times k}$, $C \in \mathbb{R}^{n \times n}$ symmetric, stored in lower/upper triangular part
		$C := \alpha(A^T B + B^T A) + \beta C$	$2n^2 k$	$A \in \mathbb{R}^{k \times n}$, $C \in \mathbb{R}^{n \times n}$ symmetric, stored in lower/upper triangular part
TRiangular Solve with Multiple right-hand sides	TRSM	$B := L^{-1}B, B := L^{-T}B$ $B := U^{-1}B, B := U^{-T}B$	$m^2 n$	$B \in \mathbb{R}^{m \times n}$, $L \in \mathbb{R}^{m \times m}$ is lower triangular, $U \in \mathbb{R}^{m \times m}$ is upper triangular
		$B := BL, B := BL^T$ $B := BU, B := BU^T$	mn^2	$B \in \mathbb{R}^{m \times n}$, $L \in \mathbb{R}^{n \times n}$ is lower triangular, $U \in \mathbb{R}^{n \times n}$ is upper triangular

Figure 5.14: Basic matrix-matrix operations. Cost is approximate.

5.8 Further Exercises

For additional exercises, visit `$BASE/Chapter5/`.

Chapter 6

The LU and Cholesky Factorizations

A commonly employed strategy for solving (dense) linear systems starts with the factorization of the coefficient matrix of the system into the product of two triangular matrices, followed by the solves with the resulting triangular systems. In this chapter we review two such factorizations, the LU and Cholesky factorizations. The LU factorization (combined with pivoting) is the most commonly used method for solving general linear systems. The Cholesky factorization plays an analogous role for systems with a *symmetric positive definite* (SPD) coefficient matrix.

Throughout this chapter, and unless otherwise stated explicitly, we assume that the coefficient matrix (and therefore the triangular matrices resulting from the factorization) to be nonsingular with n rows and columns.

6.1 Gaussian Elimination

Recall that *Gaussian elimination* is the process used to transform a set of linear equations into an upper triangular system, which is then solved via *back substitution*.

Assume that we are given the linear system with n equations and n unknowns defined by[1]

$$\begin{pmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1,n} \\ \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{n,1} & \alpha_{n,2} & \cdots & \alpha_{n,n} \end{pmatrix} \begin{pmatrix} \chi_1 \\ \chi_2 \\ \vdots \\ \chi_n \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{pmatrix}, \quad \text{or,} \quad Ax = b.$$

[1]For the first time we use indices starting at 1. We do so to better illustrate the relation between Gaussian elimination and the LU factorization in this section. In particular, the symbol α_{11} will denote the same element in the first step of both methods.

Gaussian elimination starts by subtracting a multiple of the first row of the matrix from the second row so as to annihilate element α_{21}. To do so, the *multiplier* $\lambda_{21} = \alpha_{21}/\alpha_{11}$ is first computed, after which the first row times the multiplier λ_{21} is subtracted from the second row of A, resulting in

$$\left(\begin{array}{cccc} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1,n} \\ 0 & \alpha_{22}-\lambda_{21}\alpha_{12} & \cdots & \alpha_{2,n}-\lambda_{21}\alpha_{1,n} \\ \alpha_{31} & \alpha_{32} & \cdots & \alpha_{3,n} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{n,1} & \alpha_{n,2} & \cdots & \alpha_{n,n} \end{array}\right).$$

Next, the multiplier to eliminate the element α_{31} is computed as $\lambda_{31} = \alpha_{31}/\alpha_{11}$, and the first row times λ_{31} is subtracted from the third row of A to obtain

$$\left(\begin{array}{cccc} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1,n} \\ 0 & \alpha_{22}-\lambda_{21}\alpha_{12} & \cdots & \alpha_{2,n}-\lambda_{21}\alpha_{1,n} \\ 0 & \alpha_{32}-\lambda_{31}\alpha_{12} & \cdots & \alpha_{3,n}-\lambda_{31}\alpha_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{n,1} & \alpha_{n,2} & \cdots & \alpha_{n,n} \end{array}\right).$$

Repeating this for the remaining rows yields

$$\left(\begin{array}{c|ccc} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1,n} \\ \hline 0 & \alpha_{22}-\lambda_{21}\alpha_{12} & \cdots & \alpha_{2,n}-\lambda_{21}\alpha_{1,n} \\ 0 & \alpha_{32}-\lambda_{31}\alpha_{12} & \cdots & \alpha_{3,n}-\lambda_{31}\alpha_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \alpha_{n,2}-\lambda_{n,1}\alpha_{12} & \cdots & \alpha_{n,n}-\lambda_{n,1}\alpha_{1,n} \end{array}\right). \tag{6.1}$$

Typically, the multipliers $\lambda_{21}, \lambda_{31}, \ldots, \lambda_{n,1}$ are stored over the zeroes that are introduced. After this, the process continues with the bottom right $n-1 \times n-1$ quadrant of the matrix until eventually A becomes an upper triangular matrix.

6.2 The LU Factorization

Given a linear system $Ax = b$ where the coefficient matrix has no special structure or properties (often referred to as a *general* structure), computing the LU factorization of this matrix is usually the first step for towards solving the system. For the system to have a unique solution it is a necessary and sufficient condition that A is *nonsingular*.

Definition 6.1 *Given a square matrix A with linearly independent columns is said to be nonsingular (or invertible).*

We remind the reader of the following theorem, found in any standard linear algebra text, gives a number of equivalent characterizations of a nonsingular matrix:

Theorem 6.2 *Given a sqaure matrix A, the following equivalent:*

- *A is nonsingular.*
- *$Ax = 0$ if and only if $x = 0$.*
- *$Ax = b$ has a unique solution.*
- *There exists a an matrix, denoted as A^{-1}, which satisfies $AA^{-1} = A^{-1}A = I_n$).*
- *The determinant of A is nonzero.*

Definition 6.3 *Partition A as $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BL} \end{array}\right)$, where $A_{TL} \in \mathbb{R}^{k \times k}$. Matrices A_{TL}, $1 \leq k \leq n$, are called the* leading principle submatrices *of A.*

The LU factorization decomposes a matrix A into a unit lower triangular matrix L and an upper triangular matrix U such that $A = LU$. The following theorem states conditions for the existence of such a factorization.

Theorem 6.4 *Assume all leading principle submatrices of A are nonsingular. Then there exist a unit lower triangular matrix L and a nonsingular upper triangular matrix U such that $A = LU$.*

Proof: We delay the proof of this theorem until after algorithms for computing the LU factorization have been given.

6.2.1 The LU factorization is Gaussian elimination

We show now that the LU factorization is just Gaussian elimination in disguise.

Partition

$$A \rightarrow \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right), \quad L \rightarrow \left(\begin{array}{c|c} 1 & 0 \\ \hline l_{21} & L_{22} \end{array}\right), \quad \text{and} \quad U \rightarrow \left(\begin{array}{c|c} \mu_{11} & u_{12}^T \\ \hline 0 & U_{22} \end{array}\right),$$

where α_{11} and μ_{11} are both scalars. From $A = LU$ we find that

$$\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} 1 & 0 \\ \hline l_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c} \mu_{11} & u_{12}^T \\ \hline 0 & U_{22} \end{array}\right) = \left(\begin{array}{c|c} \mu_{11} & u_{12}^T \\ \hline \mu_{11} l_{21} & l_{21} u_{12}^T + L_{22} U_{22} \end{array}\right).$$

Equating corresponding submatrices on the left and the right of this equation yields the following insights:

- The first row of U is given by

$$\mu_{11} := \alpha_{11} \quad \text{and} \quad u_{12}^T := a_{12}^T. \tag{6.2}$$

- The first column of L, below the diagonal, is computed by

$$l_{21} := a_{21}/\mu_{11} = a_{21}/\alpha_{11}. \tag{6.3}$$

- Since $A_{22} = l_{21}u_{12}^{\mathrm{T}} + L_{22}U_{22}$, L_{22} and U_{22} can be computed from an LU factorization of $A_{22} - l_{21}u_{12}^{\mathrm{T}}$; that is,

$$(A_{22} - l_{21}u_{12}^{\mathrm{T}}) = L_{22}U_{22}. \tag{6.4}$$

Compare (6.2)–(6.4) with the steps in Gaussian elimination:

- According to (6.2), the first row of U equals the first row of A, just like the first row of A is left untouched in Gaussian elimination.
- In (6.3) we can recognize *the computation of the multipliers in Gaussian elimination:*

$$l_{21} := a_{21}/\alpha_{11} = \begin{pmatrix} \alpha_{21} \\ \alpha_{31} \\ \vdots \\ \alpha_{n,1} \end{pmatrix} /\alpha_{11} = \begin{pmatrix} \alpha_{21}/\alpha_{11} \\ \alpha_{31}/\alpha_{11} \\ \vdots \\ \alpha_{n,1}/\alpha_{11} \end{pmatrix} = \begin{pmatrix} \lambda_{21} \\ \lambda_{31} \\ \vdots \\ \lambda_{n,1} \end{pmatrix}.$$

- The update $A_{22} - l_{21}u_{12}^{\mathrm{T}}$ in (6.4) corresponds to

$$\begin{pmatrix} \alpha_{22} & \alpha_{23} & \cdots & \alpha_{2,n} \\ \alpha_{32} & \alpha_{33} & \cdots & \alpha_{3,n} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{n,2} & \alpha_{n,3} & \cdots & \alpha_{n,n} \end{pmatrix} - \begin{pmatrix} \lambda_{21} \\ \lambda_{31} \\ \vdots \\ \lambda_{n,1} \end{pmatrix} \begin{pmatrix} \alpha_{12} & \alpha_{13} & \cdots & \alpha_{1,n} \end{pmatrix}$$
$$= \begin{pmatrix} \alpha_{22} - \lambda_{21}\alpha_{12} & \alpha_{23} - \lambda_{21}\alpha_{13} & \cdots & \alpha_{2,n} - \lambda_{21}\alpha_{1,n} \\ \alpha_{32} - \lambda_{31}\alpha_{12} & \alpha_{33} - \lambda_{31}\alpha_{13} & \cdots & \alpha_{3,n} - \lambda_{31}\alpha_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{n,2} - \lambda_{n,1}\alpha_{12} & \alpha_{n,3} - \lambda_{n,1}\alpha_{13} & \cdots & \alpha_{n,n} - \lambda_{n,1}\alpha_{1,n} \end{pmatrix},$$

  which are *the same operations that are performed during Gaussian elimination on the* $n-1 \times n-1$ *bottom right submatrix of* A*.*

- After these computations have been performed, both the LU factorization and Gaussian elimination proceed (recursively) with the $n-1 \times n-1$ bottom right quadrant of A.

We conclude that Gaussian elimination and and the described algorithm for computing LU factorization perform exactly the same computations.

Exercise 6.5 *Gaussian elimination is usually applied to the augmented matrix* $\left(A \mid b \right)$ *so that, upon completion, A is overwritten by U, b is overwritten by an intermediate result y, and the solution of the linear system $Ax = b$ is obtained from $Ux = y$. Use the system defined by*

$$A = \begin{pmatrix} 3 & -1 & 2 \\ -3 & 3 & -1 \\ 6 & 0 & 4 \end{pmatrix}, \quad b = \begin{pmatrix} 7 \\ 0 \\ 18 \end{pmatrix},$$

to show that, given the LU factorization

$$A = LU = \begin{pmatrix} 1 & & \\ -1 & 1 & \\ 2 & 1 & 1 \end{pmatrix} \begin{pmatrix} 3 & -1 & 2 \\ & 2 & 1 \\ & & -1 \end{pmatrix},$$

y satisfies $Ly = b$.

The previous exercise illustrates that in applying Gaussian elimination to the augmented system, both the LU factorization of the matrix and the solution of the unit lower triangular system are performed simultaneously (in the augmented matrix, b is overwritten with the solution of $Ly = b$). On the other hand, when solving a linear system via the LU factorization, the matrix is first decomposed into the triangular matrices L and U, and then two linear systems are solved: y is computed from $Ly = b$, and then the solution x is obtained from $Ux = y$.

6.2.2 Variants

Let us examine next how to derive different variants for computing the LU factorization. The precondition[2] and postcondition for this operation are given, respectively, by

$$\begin{array}{lcl} P_{pre} & : & A = \hat{A} \\ P_{post} & : & (A = \{L \backslash U\}) \wedge (LU = \hat{A}), \end{array}$$

where the notation $A = \{L \backslash U\}$ in the postcondition indicates that L overwrites the elements of A below the diagonal while U overwrites those on and above the diagonal. (The unit elements on the diagonal entries of L will not be stored since they are implicitly known.) The requirement that L and U overwrite specific parts of A implicitly defines the dimensions and triangular structure of these factors.

In order to determine a collection of feasible loop-invariants, we start by choosing a partitioning of the matrices involved in the factorization. The triangular form of L and U requires them to be partitioned into quadrants with square diagonal blocks so that the off-diagonal block of zeroes can be cleanly identified., This then requires A to be conformally partitioned into quadrants as well. Thus,

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right), \quad L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array} \right), \quad U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array} \right),$$

[2] Strictly speaking, the precondition should also assert that A is square and has nonsingular leading principle submatrices (see Theorem 6.4).

where

$$P_{cons} \quad : \quad m(A_{TL}) = n(A_{TL}) = m(L_{TL}) = n(L_{TL}) = m(U_{TL}) = n(U_{TL})$$

holds. Substituting these into the postcondition yields

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} \{L\backslash U\}_{TL} & U_{TR} \\ \hline L_{BL} & \{L\backslash U\}_{BR} \end{array}\right) \wedge \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) = \left(\begin{array}{c|c} \hat{A}_{TL} & \hat{A}_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right),$$

from which, multiplying out the second expression, we obtain the PME for LU factorization:

$$\begin{array}{c|c} L_{TL}U_{TL} = \hat{A}_{TL} & L_{TL}U_{TR} = \hat{A}_{TR} \\ \hline L_{BL}U_{TL} = \hat{A}_{BL} & L_{BR}U_{BR} = \hat{A}_{BR} - L_{BL}U_{TR} \end{array}.$$

These equations exhibit data dependences which dictate an order for the computations: $\hat{A}_{TL}$ must be factored into $L_{TL}U_{TL}$ before $U_{TR} := L_{TL}^{-1}\hat{A}_{TR}$ and $L_{BL} := \hat{A}_{BL}U_{TL}^{-1}$ can be computed, and these two triangular systems need to be solved before the update $\hat{A}_{BR} - L_{BL}U_{TR}$ can be carried out. By taking into account these dependencies, the PME yields the five feasible loop-invariants for the LU factorization in Figure 6.1.

Exercise 6.6 *Derive unblocked and blocked algorithms corresponding to each of the five loop-invariants in Figure 6.1.*

Note that the resulting algorithms are exactly those given in Figure 1.3.

The loop-invariants in Figure 1.3 yield all algorithms depicted on the cover, and discussed in, G.W. Stewart's book on matrix factorization [25]. All these algorithms perform the same computatons but in different order.

6.2.3 Cost analysis

Let $C_{\mathrm{LU}\nu}(k)$ equal the number of flops that have been performed by Variant ν in Figure 1.3 when the algorithm has proceeded to the point where, before the loop guard is evaluated, A_{TL} is $k \times k$. Concentrating on Variant 5, we can recursively define the cost as

$$\begin{array}{rcl} C_{\mathrm{LU5}}(0) & = & 0 \text{ flops}, \\ C_{\mathrm{LU5}}(k+1) & = & C_{\mathrm{LU5}}(k) + \left((n-k-1) + 2(n-k-1)^2\right) \text{ flops}, \ 0 \le k < n, \end{array}$$

where $n = m(A) = n(A)$. The base case comes from the fact that for a 0×0 matrix no computation needs to be performed. The recurrence results from the the cost of the updates in the loop-body: $l_{21} := a_{21}/\mu_{11}$ costs $n-k-1$ flops and $A_{22} := A_{22} - a_{21}a_{12}^{\mathrm{T}}$ costs $2(n-k-1)^2$ flops. Thus, the total cost for Variant 5 is given by[3]

$$C_{\mathrm{LU5}}(n) \quad = \quad \sum_{k=0}^{n-1}\left((n-k-1) + 2(n-k-1)^2\right) = \sum_{k=0}^{n-1}\left(k + 2k^2\right)$$

[3] When A_{TL} equals all the matrix, the loop-guard is evaluated false and no update is performed so that $C_{\mathrm{LU}}(n) = C_{\mathrm{LU}}(n-1)$.

Variant 1
$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \{L\backslash U\}_{TL} & \hat{A}_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right)$
Variant 2
$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \{L\backslash U\}_{TL} & U_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right)$
Variant 3
$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \{L\backslash U\}_{TL} & \hat{A}_{TR} \\ \hline L_{BL} & \hat{A}_{BR} \end{array}\right)$
Variant 4
$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \{L\backslash U\}_{TL} & U_{TR} \\ \hline L_{BL} & \hat{A}_{BR} \end{array}\right)$
Variant 5
$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \{L\backslash U\}_{TL} & U_{TR} \\ \hline L_{BL} & \hat{A}_{BR} - L_{BL}U_{TR} \end{array}\right)$

Figure 6.1: Five loop-invariants for the LU factorization.

$$\begin{aligned}
&= 2\sum_{k=0}^{n-1} k^2 + \sum_{k=0}^{n-1} k = 2\left(\sum_{k=1}^{n} k^2 - n^2\right) + \sum_{k=1}^{n} k - n \\
&= 2\left(\frac{n^3}{3} - \frac{3}{2}n^2 + \frac{n}{6}\right) + \left(\frac{n^2}{2} - \frac{n}{2}\right) = \frac{2}{3}n^3 - \frac{5}{2}n^2 - \frac{n}{6} \\
&\approx \frac{2}{3}n^3 \text{ flops.}
\end{aligned}$$

Note: The relation

$$\sum_{k=1}^{n} k^2 \approx \int_1^n x^2 dx = \frac{n^3}{3} - \frac{1}{3} \approx n^3/3$$

is a convenient way for approximating

$$\sum_{k=1}^{n} k^2 = \frac{n^3}{3} - \frac{n^2}{2} + \frac{n}{6}.$$

Exercise 6.7 *Use mathematical induction to prove that* $C_{\mathrm{LU5}}(n) = \frac{2}{3}n^3 - \frac{5}{2}n^2 - \frac{n}{6}$.

Exercise 6.8 *Show that the cost of the other four variants is identical to that of Variant 5.*

Exercise 6.9 *Show that the costs of the blocked algorithms for the LU factorization are identical to those of the nonblocked algorithms. For simplicity, assume that n is an integer multiple of n_b. What fraction of flops are cast in terms of* GEMM*?*

6.2.4 Performance

The performance of the LU factorization was already discussed in Section 1.5.

6.2.5 Gauss transforms and the LU factorization

The LU factorization is often defined in terms of the application of a series of Gauss transformations. In this section we characterize these transformation matrices and list some of their relevant properties. Gauss transforms will reappear in the next sections, when pivoting is introduced in the LU factorization.

Definition 6.10 *A* Gauss transform *is a matrix of the form*

$$L_k = \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & l_{21} & I_{n-k-1} \end{array}\right), \quad 0 \leq k < n. \tag{6.5}$$

Exercise 6.11 *Consider a set of Gauss transforms L_k, $0 \leq k < n$, defined as in (6.5). Show that*

1. *The inverse is given by*

$$L_k^{-1} = \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & -l_{21} & I_{n-k-1} \end{array}\right).$$

2. *Gauss transformations can be easily accumulated as*

$$\left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^{\mathrm{T}} & 1 & 0 \\ \hline L_{20} & 0 & I_{n-k-1} \end{array}\right) L_k = \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^{\mathrm{T}} & 1 & 0 \\ \hline L_{20} & l_{21} & I_{n-k-1} \end{array}\right),$$

where $L_{00} \in \mathbb{R}^{k \times k}$.

3. $L_0 L_1 \cdots L_{n-1} e_k = L_k e_k$, $0 \leq k < n$.
   Hint: *Use Result 2.*

Definition 6.12 *We will refer to*

$$L_{ac,k} = \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & I_{n-k} \end{array}\right), \tag{6.6}$$

where $L_{TL} \in \mathbb{R}^{k\times k}$ is unit lower triangular, as an accumulated Gauss transform.

> **Remark 6.13** *In subsequent discussion, often we will not explicitly define the dimensions of Gauss transforms and accumulated Gauss transforms, since they can be deduced from the dimension of the matrix or vector to which the transformation is applied.*

The name of this transform signifies that the product $L_{ac,k}B$ is equivalent to computing $L_0 L_1 \cdots L_{k-1} B$, where the jth column of the Gauss transform L_j, $0 \leq j < k$, equals that of $L_{ac,k}$.

Exercise 6.14 *Show that the accumulated Gauss transform $L_{ac,k} = L_0 L_1 \cdots L_{k-1}$, defined as in (6.6), satisfies*

$$L_{ac,k}^{-1} = \left(\begin{array}{c|c} L_{TL}^{-1} & 0 \\ \hline -L_{BL}L_{TL}^{-1} & I_{n-k} \end{array}\right) \quad \text{and} \quad L_{ac,k}^{-1}A = L_{k-1}^{-1} \cdots L_1^{-1} L_0^{-1} A.$$

We are now ready to describe the LU factorization in terms of the application of Gauss transforms. Partition A and the first Gauss transform L_0:

$$A \rightarrow \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right), \quad L_0 \rightarrow \left(\begin{array}{c|c} 1 & 0 \\ \hline l_{21} & I_{n-1} \end{array}\right),$$

where α_{11} is a scalar. Next, observe the result of applying the inverse of L_0 to A:

$$L_0^{-1}A = \left(\begin{array}{c|c} 1 & 0 \\ \hline -l_{21} & I_{n-1} \end{array}\right) \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} - \alpha_{11} l_{21} & A_{22} - l_{21} a_{12}^T \end{array}\right).$$

By choosing $l_{21} := a_{21}/\alpha_{11}$, we obtain $a_{21} - \alpha_{11} l_{21} = 0$ and A is updated exactly as in the first iteration of the unblocked algorithm for overwriting the matrix with its LU factorization via Variant 5.

Next, assume that after k steps A has been overwritten by $L_{k-1}^{-1} \cdots L_1^{-1} L_0^{-1} \hat{A}$ so that, by careful selection of the Gauss transforms,

$$A := L_{k-1}^{-1} \cdots L_1^{-1} L_0^{-1} \hat{A} = \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & A_{BR} \end{array}\right),$$

where $U_{TL} \in \mathbb{R}^{k\times k}$ is upper triangular. Repartition

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & u_{12} & U_{02} \\ \hline 0 & \alpha_{11} & a_{12}^T \\ \hline 0 & a_{21} & A_{22} \end{array}\right) \quad \text{and} \quad L_k \rightarrow \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & l_{21} & I_{n-k-1} \end{array}\right),$$

and choose $l_{21} := a_{21}/\alpha_{11}$. Then,

$$L_k^{-1}A = L_k^{-1}(L_{k-1}^{-1}\cdots L_1^{-1}L_0^{-1})\hat{A} = \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & -l_{21} & I_{n-k-1} \end{array}\right)\left(\begin{array}{c|c|c} U_{00} & u_{12} & U_{02} \\ \hline 0 & \alpha_{11} & a_{12}^{\mathrm{T}} \\ \hline 0 & a_{21} & A_{22} \end{array}\right)$$
$$= \left(\begin{array}{c|c|c} U_{00} & u_{12} & U_{02} \\ \hline 0 & \mu_{11} & u_{12}^{\mathrm{T}} \\ \hline 0 & 0 & A_{22} - l_{21}u_{12}^{\mathrm{T}} \end{array}\right).$$

An inductive argument can thus be used to confirm that reducing a matrix to upper triangular form by successive application of inverses of carefully chosen Gauss transforms is equivalent to performing an LU factorization. Indeed, if $L_{n-1}^{-1}\cdots L_1^{-1}L_0^{-1}A = U$ then $A = LU$, where $L = L_0L_1\cdots L_{n-1}$ is trivially constructed from the columns of the corresponding Gauss transforms.

6.3 The Basics of Partial Pivoting

The linear system $Ax = b$ has a unique solution provided A is nonsingular. However, the LU factorization described so far is only computable under the conditions stated in Theorem 6.4, which are much more restrictive. For example, the LU factorization will not complete when $A = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$. In practice, even if these conditions are satisfied, the use of finite precision arithmetic yields the computation of the LU factorization inadvisable, as we argue in this section.

Let us review the update to matrix A in (6.1):

$$\left(\begin{array}{c|ccc} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1,n} \\ \hline 0 & \alpha_{22} - \lambda_{21}\alpha_{12} & \cdots & \alpha_{2,n} - \lambda_{21}\alpha_{1,n} \\ 0 & \alpha_{32} - \lambda_{31}\alpha_{12} & \cdots & \alpha_{3,n} - \lambda_{31}\alpha_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \alpha_{n,2} - \lambda_{n,1}\alpha_{12} & \cdots & \alpha_{n,n} - \lambda_{n,1}\alpha_{1,n} \end{array}\right).$$

where $\lambda_{i,1} = \alpha_{i,1}/\alpha_{11}$, $2 \le i \le n$. The algorithm clearly fails if $\alpha_{11} = 0$, as corresponds to the 1×1 principal submatrix of A being singular. Assume now that $\alpha_{11} \ne 0$ and denote the absolute value (magnitude) of a scalar α by $|\alpha|$. If $|\alpha_{i,1}| \gg |\alpha_{11}|$, then $\lambda_{i,1}$ will be large and it can happen that $|\alpha_{i,j} - \lambda_{i,1}\alpha_{i,j}| \gg |\alpha_{i,j}|$, $2 \le j \le n$; that is, the update greatly increases the magnitude of $\alpha_{i,j}$. This is a phenomenon known as *element growth* and, as the following example borrowed from [29] shows, in the presence of limited precision has a catastrophic impact on the accuracy of the results.

Example 6.15 *Consider the linear system $Ax = b$ defined by*

$$\begin{pmatrix} 0.002 & 1.231 & 2.471 \\ 1.196 & 3.165 & 2.543 \\ 1.475 & 4.271 & 2.142 \end{pmatrix}\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 3.704 \\ 6.904 \\ 7.888 \end{pmatrix}.$$

A simple calculation shows that $x = (1, 1, 1)^{\mathrm{T}}$ *is the exact solution of the system.* $(Ax - b = 0)$.

Now, assume we use a computer where all the operations are done in four-digit decimal floating-point arithmetic. Computing the LU factorization of A in this machine then yields

$$L = \begin{pmatrix} 1.000 & & \\ 598.0 & 1.000 & \\ 737.5 & 1.233 & 1.000 \end{pmatrix} \quad \text{and} \quad U = \begin{pmatrix} 0.002 & 1.231 & 2.471 \\ & -732.9 & -1475 \\ & & -1820 \end{pmatrix},$$

which shows two large multipliers in L and the consequent element growth in U.

If we next employ these factors to solve for the system, applying forward substitution to $Ly = b$, *we obtain*

$$y = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 3.704 \\ -2208 \\ -2000 \end{pmatrix},$$

and from that, applying backward substitution to $Ux = y$,

$$x = \begin{pmatrix} 4.000 \\ -1.012 \\ 2.000 \end{pmatrix},$$

which is a completely erroneous solution!

Let us rearrange now the equations of the system (rows of A and b) in a different order

$$\bar{A}x = \bar{b} \equiv \begin{pmatrix} 1.475 & 4.271 & 2.142 \\ 1.196 & 3.165 & 2.543 \\ 0.002 & 1.231 & 2.471 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 7.888 \\ 6.904 \\ 3.704 \end{pmatrix}.$$

Then, on the same machine, we obtain the triangular factors

$$L = \begin{pmatrix} 1.000 & & \\ 1.356 \times 10^{-3} & 1.000 & \\ 0.8108 & -0.2433 & 1.000 \end{pmatrix} \quad \text{and} \quad U = \begin{pmatrix} 1.475 & 4.271 & 2.142 \\ & 1.225 & 2.468 \\ & & 1.407 \end{pmatrix}$$

which present multipliers in L of smaller magnitude less than one, and no dramatic element growth in U.

Using these factors, from $Ly = \bar{b}$ *and* $Ux = y$, *we obtain, respectively,*

$$y = \begin{pmatrix} 7.888 \\ 3.693 \\ 1.407 \end{pmatrix} \quad \text{and} \quad x = \begin{pmatrix} 1.000 \\ 1.000 \\ 1.000 \end{pmatrix}.$$

The system is now solved for the exact x.

The second part of the previous example illustrated that the problem of element growth can be solved by rearranging the rows of the matrix. Specifically, the first column of matrix A is searched for the largest element in magnitude. The row that contains such element, the *pivot row*, is swapped with the first row, after which the current step of the LU factorization proceeds. The net effect is that $|\lambda_{i,1}| \leq 1$ so that $|\alpha_{i,j} - \lambda_{i,1}\alpha_{1,j}|$ is of the same magnitude as the largest of $|\alpha_{i,j}|$ and $|\alpha_{1,j}|$, thus keeping element growth bounded. This is known as the *LU factorization with partial pivoting.*

6.3.1 Permutation matrices

To formally include row swapping in the LU factorization we introduce permutation matrices, which have the effect of rearranging the elements of vectors and entire rows or columns of matrices.

Definition 6.16 *A matrix $P \in \mathbb{R}^{n\times n}$ is said to be a* permutation *matrix (or permutation) if, when applied to the vector $x = (\chi_0, \chi_1, \ldots, \chi_{n-1})^{\mathrm{T}}$, it merely rearranges the order of the elements in that vector. Such a permutation can be represented by the vector of integers $p = (\pi_0, \pi_1, \ldots, \pi_{n-1})^{\mathrm{T}}$, where $\{\pi_0, \pi_1, \ldots, \pi_{n-1}\}$ is a permutation of $\{0, 1, \ldots, n-1\}$, and the scalars π_is indicate that the permuted vector is given by $Px = \left(\chi_{\pi_0}, \chi_{\pi_1}, \ldots, \chi_{\pi_{n-1}}\right)^{\mathrm{T}}$.*

A permutation matrix is equal to the identity matrix with permuted rows, as the next exercise states.

Exercise 6.17 *Given $p = (\pi_0, \pi_1, \ldots, \pi_{n-1})^{\mathrm{T}}$, a permutation of $\{0, 1, \ldots, n-1\}$, show that*

$$P = \begin{pmatrix} e_{\pi_0}^{\mathrm{T}} \\ e_{\pi_1}^{\mathrm{T}} \\ \vdots \\ e_{\pi_{n-1}}^{\mathrm{T}} \end{pmatrix} \in \mathbb{R}^{n\times n} \tag{6.7}$$

is the permutation matrix that, when applied to vector $x = (\chi_0, \chi_1, \ldots, \chi_{n-1})^{\mathrm{T}}$, yields $Px = \left(\chi_{\pi_0}, \chi_{\pi_1}, \ldots, \chi_{\pi_{n-1}}\right)^{\mathrm{T}}$.

The following exercise recalls a few essential properties of permutation matrices.

Exercise 6.18 *Consider A, $x \in \mathbb{R}^n$, and let $P \in \mathbb{R}^{n\times n}$ be a permutation matrix. Show that*

1. *The inverse is given by $P^{-1} = P^{\mathrm{T}}$.* **Hint:** *use (6.7).*

2. *PA rearranges the rows of A exactly in the same order as the elements of x are rearranged by Px.* **Hint:** *Partition P as in (6.7) and recall that row π of A is given by $e_\pi^{\mathrm{T}} A$.*

3. *AP^{T} rearranges the columns of A exactly in the same order as the elements of x are rearranged by Px.* **Hint:** *Consider $(PA^{\mathrm{T}})^{\mathrm{T}}$.*

We will frequently employ permutation matrices that swap the first element of a vector with element π of that vector:

Definition 6.19 *The permutation that, when applied to a vector, swaps the first element with element π is defined as*

$$P(\pi) = \begin{cases} I_n & \text{if } \pi = 0, \\ \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & I_{\pi-1} & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & I_{n-\pi-1} \end{pmatrix} & \text{otherwise.} \end{cases}$$

Definition 6.20 *Given* $p = (\pi_0, \pi_1, \ldots, \pi_{k-1})^{\mathrm{T}}$*, a permutation of* $\{0, 1, \ldots, k-1\}$*,* $P(p)$ *denotes the permutation*

$$P(p) = \begin{pmatrix} I_{k-1} & 0 \\ 0 & P(\pi_{k-1}) \end{pmatrix} \begin{pmatrix} I_{k-2} & 0 \\ 0 & P(\pi_{k-2}) \end{pmatrix} \cdots \begin{pmatrix} 1 & 0 \\ 0 & P(\pi_1) \end{pmatrix} P(\pi_0).$$

Remark 6.21 *In the previous definition, and from here on, we will typically not explicitly denote the dimension of a permutation matrix, since it can be deduced from the dimension of the matrix or the vector the permutation is applied to.*

6.3.2 An algorithm for LU factorization with partial pivoting

An algorithm that incorporates pivoting into Variant 5 (unblocked) of the LU factorization is given in Figure 6.2 (left). In that algorithm, the function PivIndex(x) returns the index of the element of largest magnitude in vector x. The matrix $P(\pi_1)$ is never formed: the appropriate rows of the matrix to which $P(\pi_1)$ is applied are merely swapped. The algorithm computes

$$L_{n-1}^{-1} P_{n-1} \cdots L_1^{-1} P_1 L_0^{-1} P_0 A = U, \tag{6.8}$$

where

$$P_k = \begin{pmatrix} I_k & 0 \\ 0 & P(\pi_k) \end{pmatrix}, \quad 0 \le k < n,$$

overwriting the upper triangular part of A with U, and the strictly lower triangular part of the kth column of A with the multipliers in L_k, $0 \le k < n$. The vector of pivot indices is stored in integer valued vector p.

6.4 Partial Pivoting and High Performance

The basic approach for partial pivoting described in Section 6.3 suffers from at least three shortcomings:

1. Solving the linear system once the factorization has been computed is inherently inefficient. The reason for this is as follows: If $Ax = b$ is the linear system to be solved, it is not hard to see that x can be obtained from
$$Ux = (L_{n-1}^{-1} P_{n-1} \cdots L_1^{-1} P_1 L_0^{-1} P_0 b) = y. \tag{6.9}$$
The problem is that vector b is repeatedly updated by applying a Gauss transform to it, an operation which is rich in AXPY (vector-vector) operations.
2. This style of pivoting is difficult to incorporate in the other variants of the LU factorization.
3. Finally, it is hard to develop an equivalent blocked algorithm.

We will show that these drawbacks can be overcome by instead computing L, U, and a permutation p of $\{0, 1, \ldots, n-1\}$ to satisfy

$$P(p)A = LU. \tag{6.10}$$

Algorithm: $[A, p] := \text{LUP_unb_var5_B}(A)$

Partition
$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), \; p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$$
where A_{TL} is 0×0 and p_T has 0 elements

while $n(A_{TL}) < n(A)$ **do**

Repartition
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \; \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$$
where α_{11} and π_1 are scalars

$$\pi_1 := \text{PivIndex}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$$
$$\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) := P(\pi_1)\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right)$$
$$a_{21} := a_{21}/\alpha_{11}$$
$$A_{22} := A_{22} - a_{21}a_{12}^T$$

Continue with
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \; \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$$

endwhile

Algorithm: $[A, p] := \text{LUP_unb_var5}(A)$

Partition
$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), \; p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$$
where A_{TL} is 0×0 and p_T has 0 elements

while $n(A_{TL}) < n(A)$ **do**

Repartition
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \; \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$$
where α_{11} and π_1 are scalars

$$\pi_1 := \text{PivIndex}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$$
$$\left(\begin{array}{c|c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) := P(\pi_1)\left(\begin{array}{c|c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$
$$a_{21} := a_{21}/\alpha_{11}$$
$$A_{22} := A_{22} - a_{21}a_{12}^T$$

Continue with
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \; \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$$

endwhile

Figure 6.2: Unblocked algorithms for the LU factorization with partial pivoting (Variant 5). Left: basic algorithm. Right: High-performance algorithm.

6.4.1 Swapping the order of permutation matrices and Gauss transforms

We start by reviewing the theory that relates (6.10) to (6.9). Critical to justifying (6.10) is an understanding of how the order of permutations and Gauss transforms can be swapped:

Lemma 6.22 *Let*

$$P = \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & Q \end{array}\right) \quad \textit{and} \quad L_k = \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & l_{21} & I_{n-k-1} \end{array}\right), \tag{6.11}$$

where $Q \in \mathbb{R}^{n-k-1 \times n-k-1}$ *is a permutation matrix. Then,*

$$\begin{aligned} PL_k &= \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & Q \end{array}\right) \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & l_{21} & I_{n-k-1} \end{array}\right) \\ &= \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & Ql_{21} & I_{n-k-1} \end{array}\right) \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & Q \end{array}\right) = \bar{L}_k P. \end{aligned}$$

In words, there exists a Gauss transform $\bar{L}_k$*, of the same dimension and structure as* L_k*, such that* $PL_k = \bar{L}_k P$.

Exercise 6.23 *Prove Lemma 6.22.*

The above lemma supports the following observation: According to (6.8), the basic LU factorization with partial pivoting yields

$$L_{n-1}^{-1} P_{n-1} \cdots L_1^{-1} P_1 L_0^{-1} P_0 A = U$$

or

$$A = P_0 L_0 P_1 L_1 \cdots P_{n-1} L_{n-1} U.$$

From the lemma, there exist Gauss transforms $L_k^{(j)}$, $0 \leq k \leq j < n$, such that

$$\begin{aligned} A &= P_0 \underbrace{L_0 P_1} L_1 P_2 L_2 \cdots P_{n-1} L_{n-1} U \\ &= P_0 P_1 L_0^{(1)} \underbrace{L_1 P_2} L_2 \cdots P_{n-1} L_{n-1} U \\ &= P_0 P_1 \underbrace{L_0^{(1)} P_2} L_1^{(1)} L_2 \cdots P_{n-1} L_{n-1} U \\ &= P_0 P_1 P_2 L_0^{(2)} L_1^{(2)} \cdots P_{n-1} L_{n-1} U \\ &= \quad \cdots \\ &= P_0 P_1 P_2 \cdots P_{n-1} L_0^{(n-1)} L_1^{(n-1)} \cdots L_{n-1}^{(n-1)} U. \end{aligned}$$

This culminates into the following result:

Theorem 6.24 *Let*

$$P_k = \begin{pmatrix} I_k & 0 \\ 0 & P(\pi_k) \end{pmatrix}$$

and L_k*,* $0 \le k < n$*, be as computed by the basic LU factorization with partial pivoting so that*

$$L_{n-1}^{-1} P_{n-1} \cdots L_1^{-1} P_1 L_0^{-1} P_0 A = U.$$

Then there exists a lower triangular matrix L *such that* $P(p)A = LU$ *with* $p = (\pi_0, \pi_1, \ldots, \pi_{n-1})^{\mathrm{T}}$.

Proof: L is given by $L = L_0^{(n-1)} L_1^{(n-1)} \cdots L_{n-1}^{(n-1)}$.

If L, U, and p satisfy $P(p)A = LU$, then $Ax = b$ can be solved by applying all permutations to b followed by two (clean) triangular solves:

$$Ax = b \quad \Longrightarrow \quad \underbrace{PA}_{LU}\, x = \underbrace{Pb}_{\bar{b}} \quad \Longrightarrow \quad LUx = \bar{b}.$$

6.4.2 Deriving multiple algorithms

The observations in Section 6.4.1 can be turned into the algorithm in Figure 6.2 (right). It differs from the algorithm on the left only in that entire rows of A are pivoted, as corresponds to the permutations being also applied to the part of L computed so far. This has the effect of reordering the permutations and Gauss transforms so that the permutations are all shifted to the left.

In this section we show that instead that algorithm, and several other variants, can be systematically derived from the postcondition

$$P_{post} : (A = \{L \backslash U\}) \wedge (LU = P(p)\hat{A}) \wedge (|L| \le 1).$$

Here $|L| \le 1$ indicates that all entries in L must be of magnitude less or equal than one.

Remark 6.25 *It will become obvious that the family of algorithms for the LU factorization with pivoting can be derived without the introduction of Gauss transforms or knowledge about how Gauss transforms and permutation matrices can be reordered. They result from systematic application of the derivation techniques to the operation that computes* p*,* L*, and* U *that satisfy the postcondition.*

As usual, we start by deriving a PME for this operation: Partition A, L, and U into quadrants,

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), \quad L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right), \quad U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right),$$

and partition p conformally as

$$p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$$

where the predicate

$$P_{cons} \quad : \quad m(A_{TL}) = n(A_{TL}) = m(L_{TL}) = n(L_{TL}) = m(U_{TL}) = n(U_{TL}) = m(p_T)$$

holds. Substitution into the postcondition then yields

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} \{L\backslash U\}_{TL} & U_{TR} \\ \hline L_{BL} & \{L\backslash U\}_{BR} \end{array}\right) \tag{6.12}$$

$$\wedge \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) = P\left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \left(\begin{array}{c|c} \hat{A}_{TL} & \hat{A}_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right) \tag{6.13}$$

$$\wedge \left|\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)\right| \leq 1. \tag{6.14}$$

Theorem 6.26 *The expressions in (6.13) and (6.14) are equivalent to the simultaneous equations*

$$\left(\begin{array}{c|c} \bar{A}_{TL} & \bar{A}_{TR} \\ \hline \bar{A}_{BL} & \bar{A}_{BR} \end{array}\right) = P(p_T)\left(\begin{array}{c|c} \hat{A}_{TL} & \hat{A}_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right), \tag{6.15}$$

$$\bar{L}_{BL} = P(p_B)^T L_{BL}, \tag{6.16}$$

$$\left(\begin{array}{c} L_{TL} \\ \hline \bar{L}_{BL} \end{array}\right) U_{TL} = \left(\begin{array}{c} \bar{A}_{TL} \\ \hline \bar{A}_{BL} \end{array}\right) \wedge \left|\left(\begin{array}{c} L_{TL} \\ \hline \bar{L}_{BL} \end{array}\right)\right| \leq 1, \tag{6.17}$$

$$L_{TL}U_{TR} = \bar{A}_{TR}, \tag{6.18}$$

$$L_{BR}U_{BR} = P(p_B)(\bar{A}_{BR} - \bar{L}_{BL}U_{TR}) \wedge |L_{BR}| \leq 1, \tag{6.19}$$

which together represent the PME for LU factorization with partial pivoting.

Exercise 6.27 *Prove Theorem 6.26.*

Equations (6.15)–(6.19) have the following interpretation:

- Equations (6.15) and (6.16) are included for notational convenience. Equation (6.16) states that $\bar{L}_{BL}$ equals the final L_{BL} except that its rows have not yet been permuted according to future computation.
- Equation (6.17) denotes an LU factorization with partial pivoting of the submatrices to the left of the thick line in (6.13).
- Equations (6.18) and (6.19) indicate that U_{TR} and $\{L\backslash U\}_{BR}$ result from permuting the submatrices to the right of the thick line in (6.14), after which U_{TR} is computed by solving the triangular system $L_{TL}^{-1}\bar{A}_{TR}$, and $\{L\backslash U\}_{BR}$ result from updating $\bar{A}_{BR}$ and performing an LU factorization with partial pivoting of that quadrant. Equation (6.16) resurfaces here, since the permutations p_B must also be applied to $\bar{L}_{BL}$ to yield L_{BL}.

Variant 3a

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{TR} \end{array}\right) = \left(\begin{array}{c|c} \{L\backslash U\}_{TL} & \hat{A}_{TR} \\ \hline \bar{L}_{BL} & \hat{A}_{BR} \end{array}\right)$$
$$\wedge \quad \left(\begin{array}{c} \bar{A}_{TL} \\ \hline \bar{A}_{BL} \end{array}\right) = P(p_T) \left(\begin{array}{c} \hat{A}_{TL} \\ \hline \hat{A}_{BL} \end{array}\right)$$
$$\wedge \quad \left(\begin{array}{c} L_{TL} \\ \hline \bar{L}_{BL} \end{array}\right) U_{TL} = \left(\begin{array}{c} \bar{A}_{TL} \\ \hline \bar{A}_{BL} \end{array}\right) \quad \wedge \quad \left|\left(\begin{array}{c} L_{TL} \\ \hline \bar{L}_{BL} \end{array}\right)\right| \leq 1$$

Variant 3b

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{TR} \end{array}\right) = \left(\begin{array}{c|c} \{L\backslash U\}_{TL} & \bar{A}_{TR} \\ \hline \bar{L}_{BL} & \bar{A}_{BR} \end{array}\right)$$
$$\wedge \quad \left(\begin{array}{c|c} \bar{A}_{TL} & \bar{A}_{TR} \\ \hline \bar{A}_{BL} & \bar{A}_{BR} \end{array}\right) = P(p_T) \left(\begin{array}{c|c} \hat{A}_{TL} & \hat{A}_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right)$$
$$\wedge \quad \left(\begin{array}{c} L_{TL} \\ \hline \bar{L}_{BL} \end{array}\right) U_{TL} = \left(\begin{array}{c} \bar{A}_{TL} \\ \hline \bar{A}_{BL} \end{array}\right) \quad \wedge \quad \left|\left(\begin{array}{c} L_{TL} \\ \hline \bar{L}_{BL} \end{array}\right)\right| \leq 1$$

Variant 4

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{TR} \end{array}\right) = \left(\begin{array}{c|c} \{L\backslash U\}_{TL} & U_{TR} \\ \hline \bar{L}_{BL} & \bar{A}_{BR} \end{array}\right)$$
$$\wedge \quad \left(\begin{array}{c|c} \bar{A}_{TL} & \bar{A}_{TR} \\ \hline \bar{A}_{BL} & \bar{A}_{BR} \end{array}\right) = P(p_T) \left(\begin{array}{c|c} \hat{A}_{TL} & \hat{A}_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right)$$
$$\wedge \quad \left(\begin{array}{c} L_{TL} \\ \hline \bar{L}_{BL} \end{array}\right) U_{TL} = \left(\begin{array}{c} \bar{A}_{TL} \\ \hline \bar{A}_{BL} \end{array}\right) \ \wedge \ L_{TL}U_{TR} = \bar{A}_{TR} \ \wedge \ \left|\left(\begin{array}{c} L_{TL} \\ \hline \bar{L}_{BL} \end{array}\right)\right| \leq 1$$

Variant 5

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{TR} \end{array}\right) = \left(\begin{array}{c|c} \{L\backslash U\}_{TL} & U_{TR} \\ \hline \bar{L}_{BL} & \bar{A}_{BR} - \bar{L}_{BL}U_{TR} \end{array}\right)$$
$$\wedge \quad \left(\begin{array}{c|c} \bar{A}_{TL} & \bar{A}_{TR} \\ \hline \bar{A}_{BL} & \bar{A}_{BR} \end{array}\right) = P(p_T) \left(\begin{array}{c|c} \hat{A}_{TL} & \hat{A}_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right)$$
$$\wedge \quad \left(\begin{array}{c} L_{TL} \\ \hline \bar{L}_{BL} \end{array}\right) U_{TL} = \left(\begin{array}{c} \bar{A}_{TL} \\ \hline \bar{A}_{BL} \end{array}\right) \ \wedge \ L_{TL}U_{TR} = \bar{A}_{TR} \ \wedge \ \left|\left(\begin{array}{c} L_{TL} \\ \hline \bar{L}_{BL} \end{array}\right)\right| \leq 1$$

Figure 6.3: Four loop-invariants for the LU factorization with partial pivoting.

Equations (6.15)–(6.19) dictate an inherent order in which the computations must proceed:

- If one of p_T, L_{TL}, U_{TL}, or $\bar{L}_{BL}$ has been computed, so have the others. In other words, any loop invariant must include the computation of all four of these results.
- In the loop invariant, $A_{TR} = \bar{A}_{TR}$ implies $A_{BR} = \bar{A}_{BR}$ and viceversa.
- In the loop invariant, $A_{TR} = U_{TR}$ only if $A_{BR} = \bar{A}_{BR}$, since U_{TR} requires $\bar{A}_{TR}$.
- In the loop invariant $A_{BR} = \bar{A}_{BR}$ only if $A_{TR} = U_{TR}$, since $\bar{A}_{BR}$ requires U_{BR}.

These constraints yield the four feasible loop invariants given in Figure 6.3. In that figure, the variant number reflects the variant for the LU factorization *without* pivoting (Figure 6.1) that is most closely related. Variants 3a and 3b only differ in that for Variant 3b the pivots computed so far have also been applied to the columns to the right of the thick line in (6.14). Variants 1 and 2 from Figure 6.1 have no correspondence here as pivoting affects the entire rows of $\left(\begin{array}{c} A_{TL} \\ \hline A_{BL} \end{array}\right)$. In other words, in these two variants L_{TL} and U_{TL} have been computed but $\bar{L}_{BL}$ has not which can be argued is not possible for a feasible loop invariant.

Let us fully elaborate the case labeled as Variant 5. Partitioning to expose the next rows and columns of A, L, U and the next element of p, as usual, and substituting into the loop invariant yields

$$\left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c|c} \{L\backslash U\}_{00} & u_{01} & U_{02} \\ \hline l_{10}^T & \bar{\alpha}_{11} - l_{10}^T u_{01} & \bar{a}_{12}^T - l_{10}^T U_{02} \\ \hline \bar{L}_{20} & \bar{a}_{21} - \bar{L}_{20} u_{01} & \bar{A}_{22} - \bar{L}_{20} U_{02} \end{array}\right) \tag{6.20}$$

$$\wedge \left(\begin{array}{c|c|c} \bar{A}_{00} & \bar{a}_{01} & \bar{A}_{02} \\ \hline \bar{a}_{10}^T & \bar{\alpha}_{11} & \bar{a}_{12}^T \\ \hline \bar{A}_{20} & \bar{a}_{21} & \bar{A}_{22} \end{array}\right) = P(p_0) \left(\begin{array}{c|c|c} \hat{A}_{00} & \hat{a}_{01} & \hat{A}_{02} \\ \hline \hat{a}_{10}^T & \hat{\alpha}_{11} & \hat{a}_{12}^T \\ \hline \hat{A}_{20} & \hat{a}_{21} & \hat{A}_{22} \end{array}\right) \wedge \left(\begin{array}{c} L_{00} \\ \hline l_{10}^T \\ \hline \bar{L}_{20} \end{array}\right) U_{00} = \left(\begin{array}{c} \bar{A}_{00} \\ \hline \bar{a}_{10}^T \\ \hline \bar{A}_{20} \end{array}\right) \tag{6.21}$$

$$\wedge L_{00} \left(\begin{array}{c|c} u_{01} & U_{02} \end{array}\right) = \left(\begin{array}{c|c} \bar{a}_{01} & \bar{A}_{02} \end{array}\right) \wedge \left| \left(\begin{array}{c} L_{00} \\ \hline l_{10}^T \\ \hline \bar{L}_{20} \end{array}\right) \right| \leq 1. \tag{6.22}$$

Similarly, after moving the thick lines substitution into the loop invariant yields

$$\left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c|c} \{L\backslash U\}_{00} & u_{01} & U_{02} \\ \hline l_{10}^T & \mu_{11} & u_{12}^T \\ \hline \bar{\bar{L}}_{20} & \bar{l}_{21} & \bar{A}_{22} - \bar{\bar{L}}_{20} U_{02} - \bar{l}_{21} u_{12}^T \end{array}\right) \tag{6.23}$$

$$\wedge \left(\begin{array}{c|c|c} \bar{\bar{A}}_{00} & \bar{\bar{a}}_{01} & \bar{\bar{A}}_{02} \\ \hline \bar{\bar{a}}_{10}^T & \bar{\bar{\alpha}}_{11} & \bar{\bar{a}}_{12}^T \\ \hline \bar{A}_{20} & \bar{\bar{a}}_{21} & \bar{A}_{22} \end{array}\right) = P(\left(\begin{array}{c} p_0 \\ \hline \pi_1 \end{array}\right)) \left(\begin{array}{c|c|c} \hat{A}_{00} & \hat{a}_{01} & \hat{A}_{02} \\ \hline \hat{a}_{10}^T & \hat{\alpha}_{11} & \hat{a}_{12}^T \\ \hline \hat{A}_{20} & \hat{a}_{21} & \hat{A}_{22} \end{array}\right) \tag{6.24}$$

$$\wedge \left(\begin{array}{c|c} L_{00} & 0 \\ \hline l_{10}^{\mathrm{T}} & 1 \\ \hline \bar{L}_{20} & \bar{l}_{21} \end{array}\right) \left(\begin{array}{c|c} U_{00} & u_{01} \\ \hline 0 & \mu_{11} \end{array}\right) = \left(\begin{array}{c|c} \bar{\bar{A}}_{00} & \bar{\bar{a}}_{01} \\ \hline \bar{\bar{a}}_{10}^{\mathrm{T}} & \bar{\bar{\alpha}}_{11} \\ \hline \bar{A}_{20} & \bar{\bar{a}}_{21} \end{array}\right) \wedge \left(\begin{array}{c|c} L_{00} & 0 \\ \hline l_{10}^{\mathrm{T}} & 1 \end{array}\right) \left(\begin{array}{c} U_{02} \\ \hline u_{12}^{\mathrm{T}} \end{array}\right) = \left(\begin{array}{c} \bar{\bar{A}}_{02} \\ \hline \bar{\bar{a}}_{12}^{\mathrm{T}} \end{array}\right) \tag{6.25}$$

$$\wedge \left| \left(\begin{array}{c|c} L_{00} & 0 \\ \hline l_{10}^{\mathrm{T}} & 1 \\ \hline \bar{L}_{20} & \bar{l}_{21} \end{array}\right) \right| \leq 1. \tag{6.26}$$

A careful manipulation of the conditions after repartitioning, in (6.20)–(6.22), and the conditions after moving the thick line, in (6.23)–(6.26), shows that the current contents of A must be updated by the steps

1. $\pi_1 := \mathrm{PivIndex}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$.
2. $\left(\begin{array}{c|c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) := P(\pi_1) \left(\begin{array}{c|c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$.
3. $a_{21} := a_{21}/\alpha_{11}$.
4. $A_{22} := A_{22} - a_{21}a_{12}^T$.

The algorithms corresponding to Variants 3a, 4, and 5 are given in Figure 6.4. There, $\textsc{trilu}(A_{i,i})$ stands for the unit lower triangular matrix stored in $A_{i,i}$, $i = 0, 1$.

6.5 The Cholesky Factorization

In this section we review a specialized factorization for a *symmetric positive definite matrix*. The symmetric structure yields a reduction of the cost of the factorization algorithm while the combination of that property and positive definiteness ensures the existence of the factorization and eliminates the need for pivoting.

Definition 6.28 *A symmetric matrix $A \in \mathbb{R}^{n\times n}$ is said to be* symmetric positive definite (SPD) *if and only if $x^{\mathrm{T}}Ax > 0$ for all $x \in \mathbb{R}^n$ such that $x \neq 0$.*

6.5.1 Variants for computing the Cholesky factorization

Given an SPD matrix $A \in \mathbb{R}^{n\times n}$, the Cholesky factorization computes a lower triangular matrix $L \in \mathbb{R}^{n\times n}$ such that $A = LL^{\mathrm{T}}$. (As an alternative we could compute an upper triangular matrix $U \in \mathbb{R}^{n\times n}$ such that $A = U^{\mathrm{T}}U$.) As usual we will assume that the lower triangular part of A contains the relevant entries of the matrix, that is, $\mathrm{SyLw}(A)$. On completion, the triangular factor L will overwrite this part of the matrix while the strictly upper triangular part of A will not be referenced or modified.

Algorithm: $[A,p] := \text{LU_PIV_UNB}(A)$

Partition
$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$
where A_{TL} is 0×0 and p_T has 0 elements
while $n(A_{TL}) < n(A)$ **do**

Repartition
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$$
where α_{11} and π_1 are scalars

Variant 3a:
$\left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array}\right) := P(p_0) \left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array}\right)$
$a_{01} := \text{TRILU}(A_{00})^{-1} a_{01}$
$\alpha_{11} := \alpha_{11} - a_{10}^T a_{01}$
$a_{21} := a_{21} - A_{20} a_{01}$
$\pi_1 := \text{PivIndex}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$
$\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right) := P(\pi_1) \left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$
$a_{21} := a_{21}/\alpha_{11}$

Variant 4:
$\alpha_{11} := \alpha_{11} - a_{10}^T a_{01}$
$a_{21} := a_{21} - A_{20} a_{01}$
$a_{12}^T := a_{12}^T - a_{10}^T A_{02}$
$\pi_1 := \text{PivIndex}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$
$\left(\begin{array}{c|c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) := P(\pi_1) \left(\begin{array}{c|c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$
$a_{21} := a_{21}/\alpha_{11}$

Variant 5:
$\pi_1 := \text{PivIndex}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$
$\left(\begin{array}{c|c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) := P(\pi_1) \left(\begin{array}{c|c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$
$a_{21} := a_{21}/\alpha_{11}$
$A_{22} := A_{22} - a_{21} a_{12}^T$

Continue with
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$$
endwhile

Algorithm: $[A,p] := \text{LU_PIV_BLK}(A)$

Partition
$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$
where A_{TL} is 0×0 has p_T has 0 elements
while $n(A_{TL}) < n(A)$ **do**
Determine block size n_b
Repartition
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$$
where A_{11} is $n_b \times n_b$ and p_1 has n_b elements

Variant 3a:
$\left(\begin{array}{c} A_{01} \\ \hline A_{11} \\ \hline A_{21} \end{array}\right) := P(p_0) \left(\begin{array}{c} A_{01} \\ \hline A_{11} \\ \hline A_{21} \end{array}\right)$
$A_{01} := \text{TRILU}(A_{00}) A_{01}$
$A_{11} := A_{11} - A_{10} A_{01}$
$A_{21} := A_{21} - A_{20} A_{01}$
$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), p_1\right] := \text{LUP_UNB}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$

Variant 4:
$A_{11} := A_{11} - A_{10} A_{01}$
$A_{21} := A_{21} - A_{20} A_{01}$
$A_{12} := A_{12} - A_{10} A_{02}$
$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), p_1\right] := \text{LUP_UNB}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$
$\left(\begin{array}{c|c|c} A_{10} & & A_{12} \\ \hline A_{20} & & A_{22} \end{array}\right) := P(p_1) \left(\begin{array}{c|c|c} A_{10} & & A_{12} \\ \hline A_{20} & & A_{22} \end{array}\right)$
$A_{12} := \text{TRILU}(A_{11})^{-1} A_{12}$

Variant 5:
$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), p_1\right] := \text{LUP_UNB}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$
$\left(\begin{array}{c|c|c} A_{10} & & A_{12} \\ \hline A_{20} & & A_{22} \end{array}\right) := P(p_1) \left(\begin{array}{c|c|c} A_{10} & & A_{12} \\ \hline A_{20} & & A_{22} \end{array}\right)$
$A_{12} := \text{TRILU}(A_{11})^{-1} A_{12}$
$A_{22} := A_{22} - A_{21} A_{12}$

Continue with
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$$
endwhile

Figure 6.4: Unblocked and blocked algorithms for the LU factorization with partial pivoting.

Let us examine how to derive different variants for computing this factorization. The precondition[4] and postcondition of the operation are expressed, respectively, as

$$\begin{array}{lcl} P_{pre} & : & A \in \mathbb{R}^{n\times n} \wedge \mathrm{SyLw}(A) \quad \text{and} \\ P_{post} & : & (\textsc{tril}\,(A) = L) \wedge (LL^{\mathrm{T}} = \hat{A}), \end{array}$$

where, as usual, $\textsc{tril}\,(A)$ denotes the lower triangular part of A. The postcondition implicitly specifies the dimensions and lower triangular structure of L.

The triangular structure of L requires it to be partitioned into quadrants with square diagonal blocks, and this requires A to be conformally partitioned into quadrants as well:

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

where

$$P_{cons} \quad : \quad m(A_{TL}) = n(A_{TL}) = m(L_{TL}) = n(L_{TL}).$$

holds. Substituting these into the postcondition yields

$$\begin{array}{c} \textsc{tril}\left(\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right)\right) = \left(\begin{array}{c|c} \textsc{tril}\,(A_{TL}) & 0 \\ \hline A_{BL} & \textsc{tril}\,(A_{BR}) \end{array}\right) = \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \\ \wedge \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \left(\begin{array}{c|c} L_{TL}^{\mathrm{T}} & L_{BL}^{\mathrm{T}} \\ \hline 0 & L_{BR}^{\mathrm{T}} \end{array}\right) = \left(\begin{array}{c|c} \hat{A}_{TL} & \star \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right). \end{array}$$

The "$\star$" symbol is used in this expression and from now on to indicate a part of a symmetric matrix that is not referenced. The second part of the postcondition can then be rewritten as the PME

$$\begin{array}{c|c} L_{TL}L_{TL}^{\mathrm{T}} = \hat{A}_{TL} & \star \\ \hline L_{BL}L_{TL}^{\mathrm{T}} = \hat{A}_{BL} & L_{BR}L_{BR} = \hat{A}_{BR} - L_{BL}L_{BL}^{\mathrm{T}} \end{array},$$

showing that $\hat{A}_{TL}$ must be factored before $L_{BL} := \hat{A}_{BL}L_{TL}^{-\mathrm{T}}$ can be solved, and L_{BL} itself is needed in order to compute the update $\hat{A}_{BR} - L_{BL}L_{BL}^{\mathrm{T}}$. These dependences result in the three feasible loop-invariants for Cholesky factorization in Figure 6.5. We present the unblocked and blocked algorithms that result from these three invariants in Figure 6.6.

Exercise 6.29 *Using the worksheet, show that the unblocked and blocked algorithms corresponding to the three loop-invariants in Figure 6.5 are those given in Figure 6.6.*

Exercise 6.30 *Identify the type of operations that are performed in the blocked algorithms for the Cholesky factorization in Figure 6.6 (right) as one of these types:* TRSM, GEMM, CHOL, *or* SYRK.

[4] A complete precondition would also assert that A is positive definite in order to guarantee existence of the factorization.

Variant 1	
$\left(\begin{array}{c\|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L_{TL} & \star \\ \hline L_{BL} & \hat{A}_{BR} - L_{BL}L_{BL}^{\mathrm{T}} \end{array}\right)$	
Variant 2	**Variant 3**
$\left(\begin{array}{c\|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L_{TL} & \star \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c\|c} L_{TL} & \star \\ \hline L_{BL} & \hat{A}_{BR} \end{array}\right)$

Figure 6.5: Three loop-invariants for the Cholesky factorization.

Exercise 6.31 *Prove that the cost of the Cholesky factorization is*

$$C_{\mathrm{CHOL}} = \frac{n^3}{3} \text{ flops.}$$

Exercise 6.32 *Show that the cost of the blocked algorithms for the Cholesky factorization is the same as that of the nonblocked algorithms.*
Considering that n is an exact multiple of n_b with $n_b \ll n$, what is the amount of flops that are performed in terms of GEMM*?*

6.5.2 Performance

The performance of the Cholesky factorization is similar to that of the LU factorization, which was studied in Section 1.5.

6.6 Summary

In this chapter it was demonstrated that

- The FLAME techniques for deriving algorithms extend to more complex linear algebra operations.
- Algorithms for factorization operations can be cast in terms of matrix-matrix multiply, and its special cases, so that high performance can be attained.
- Complex operations, like the LU factorization with partial pivoting, fit the mold.

This chapter completes the discussion of the basic techniques that underlie the FLAME methodology.

Algorithm: $A := \textsc{Chol_unb}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
where A_{TL} is 0×0
while $m(A_{TL}) < m(A)$ **do**

Repartition
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$
where α_{11} is 1×1

Variant 1:
$\alpha_{11} := \sqrt{\alpha_{11}}$
$a_{21} := a_{21}/\alpha_{11}$
$A_{22} := A_{22} - a_{21}a_{21}^T$

Variant 2:
$a_{10}^T := a_{10}^T \text{TRIL}(A)_{00}^{-T}$
$\alpha_{11} := \alpha_{11} - a_{10}^T a_{10}$
$\alpha_{11} := \sqrt{\alpha_{11}}$

Variant 3:
$\alpha_{11} := \alpha_{11} - a_{10}^T a_{10}$
$\alpha_{11} := \sqrt{\alpha_{11}}$
$a_{21} := a_{21} - A_{20}a_{10}$
$a_{21} := a_{21}/\alpha_{11}$

Continue with
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$
endwhile

Algorithm: $A := \textsc{Chol_blk}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
where A_{TL} is 0×0
while $m(A_{TL}) < m(A)$ **do**
Determine block size n_b
Repartition
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$
where A_{11} is $n_b \times n_b$

Variant 1:
$A_{11} := \textsc{Chol_unb}(A_{11})$
$A_{21} := A_{21} \text{TRIL}(A_{11})^{-T}$
$A_{22} := A_{22} - A_{21}A_{21}^T$

Variant 2:
$A_{10} := A_{10} \text{TRIL}(A)_{00}^{-T}$
$A_{11} := A_{11} - A_{10}A_{10}^T$
$A_{11} := \textsc{Chol_unb}(A_{11})$

Variant 3:
$A_{11} := A_{11} - A_{10}A_{10}^T$
$A_{11} := \textsc{Chol_unb}(A_{11})$
$A_{21} := A_{21} - A_{20}A_{10}^T$
$A_{21} := A_{21} \text{TRIL}(A)_{11}^{-T}$

Continue with
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$
endwhile

Figure 6.6: Algorithms for computing the Cholesky factorization.

6.7 Further Exercises

For additional exercises, visit `$BASE/Chapter6/`.

Appendix A

The Use of Letters

We attempt to be very consistent with our notation in this book as well as in FLAME related papers, the FLAME website, and the linear algebra wiki.

As mentioned in Remark 3.1, Lowercase Greek letters and Roman letters will be used to denote scalars and vectors, respectively. Uppercase Roman letters will be used for matrices. Exceptions to this rule are variables that denote the (integer) dimensions of the vectors and matrices which are denoted by Roman lowercase letters to follow the traditional convention.

The letters used for a matrix, vectors that appear as submatrices of that matrix (e.g., its columns), and elements of that matrix are chosen in a consistent fashion Similarly, letters used for a vector and elements of that vector are chosen to correspond. This consistent choice is indicated in Figure A.1. In that table we do not claim that Greek letters used are the Greek letters that correspond to the inidicated Roman letters. We are merely indicating what letters we chose.

Matrix	Vector	Scalar			Note
		Symbol	LaTeX	Code	
A	a	α	`\alpha`	`alpha`	
B	b	β	`\beta`	`beta`	
C	c	γ	`\gamma`	`gamma`	
D	d	δ	`\delta`	`delta`	
E	e	ϵ	`\epsilon`	`epsilon`	$e_j = j$th unit basis vector.
F	f	ϕ	`\phi`	`phi`	
G	g	ξ	`\xi`	`xi`	
H	h	η	`\eta`	`eta`	
I	i	ι	`\iota`	`iota`	I is used for the identity matrix.
K	k	κ	`\kappa`	`kappa`	
L	l	λ	`\lambda`	`lambda`	
M	m	μ	`\mu`	`mu`	$m(X)$ = row dimension of X.
N	n	ν	`\nu`	`nu`	$n(X)$ = column dimension of X. Shared with V.
P	p	π	`\pi`	`pi`	
Q	q	θ	`\theta`	`theta`	
R	r	ρ	`\rho`	`rho`	
S	s	σ	`\sigma`	`sigma`	
T	t	τ	`\tau`	`tau`	
U	u	υ	`\upsilon`	`upsilon`	
V	v	ν	`\nu`	`nu`	Shared with N.
W	w	ω	`\omega`	`omega`	
X	x	χ	`\chi`	`chi`	
Y	y	ψ	`\psi`	`psi`	
Z	z	ζ	`\zeta`	`zeta`	

Figure A.1: Correspondence between letters used for matrices (uppercase Roman), vectors (lowercase Roman) and the symbols used to denote their scalar entries (lowercase Greek letters).

Appendix B

Summary of FLAME/C Routines

In this appendix, we list a number of routines supported as part of the current implementation of the FLAME library.

Additional Information

Information on the library, libFLAME, that uses the APIs and techniques discussed in this book, and the functionality supported by the library, visit

`http://www.cs.utexas.edu/users/flame/`

A Quick Reference guide can be downloaded from

`http://www.cs.utexas.edu/users/flame/Publications/`

B.1 Parameters

A number of parameters can be passed in that indicate how FLAME objects are to be used. These are summarized in Fig. B.1.

Parameter	Datatype	Permitted values	Meaning

`datatype`	`FLA_Datatype`	`FLA_INT`
		`FLA_FLOAT`
		`FLA_DOUBLE`
		`FLA_COMPLEX`
		`FLA_DOUBLE_COMPLEX`
			Elements of object are to be of indicated numerical datatype.

`side`	`FLA_Side`	`FLA_LEFT` `FLA_RIGHT`	Indicates whether the matrix with special structure appears on the left or on the right.
`uplo`	`FLA_Uplo`	`FLA_LOWER_TRIANGULAR` `FLA_UPPER_TRIANGULAR`	Indicates whether lower or upper triangular part of array stores the matrix.

`trans`
`transA`
`transB`
	`FLA_Trans`	`FLA_NO_TRANSPOSE`	Do not transpose: $op_{\texttt{trans}}(X) = X$.

		`FLA_TRANSPOSE`	Transpose: $op_{\texttt{trans}}(X) = X^T$.
		`FLA_CONJ_TRANSPOSE`	Conjugate transpose: $op_{\texttt{trans}}(X) = X^H = X^T$.
		`FLA_CONJ_NO_TRANSPOSE`	Conjugate no transpose: $op_{\texttt{trans}}(X) = X$.
`diag`	`FLA_Diag`	`FLA_NONUNIT_DIAG`	Use values stored on the diagonal.
		`FLA_UNIT_DIAG`	Compute as if diagonal elements equal one.
		`FLA_UNIT_DIAG`	Compute as if diagonal elements equal zero.
`conj`	`FLA_Conj`	`FLA_NO_CONJUGATE` `FLA_CONJUGATE`	Indicates whether to conjugate.

`quadrant`	`FLA_Quadrant`	`FLA_TL`
		`FLA_TR`
		`FLA_BL`
		`FLA_BR`
			Indicates from which quadrant the center submatrix is partitioned.

Figure B.1: Table of parameters and permitted values.

B.2 Initializing and Finalizing FLAME/C

`FLA_Init( )` Initialize FLAME.
`FLA_Finalize( )` Finalize FLAME.

B.3 Manipulating Linear Algebra Objects

Creators, destructors, and inquiry routines

`FLA_Obj_create( FLA_Datatype datatype, int m, int n, FLA_Obj *matrix )` Create an object that describes an $m \times n$ matrix and create the associated storage array.
`FLA_Obj_create_without_buffer( FLA_Datatype datatype, int m, int n, FLA_Obj *matrix )` Create an object that describes an $m \times n$ matrix without creating the associated storage array.
`FLA_Obj_attach_buffer( void *buff, int ldim, FLA_Obj *matrix )` Attach a buffer that holds a matrix stored in column-major order with leading dimension `ldim` to the object `matrix`.
`FLA_Obj_create_conf_to( FLA_Trans trans, FLA_Obj old, FLA_Obj *matrix )` Like `FLA_Obj_create` except that it creates an object with same datatype and dimensions as `old`, transposing if desired.
`FLA_Obj_free( FLA_Obj *obj )` Free all space allocated to store data associated with `obj`.
`FLA_Obj_free_without_buffer( FLA_Obj *obj )` Free the space allocated to for `obj` without freeing the buffer.
`FLA_Datatype FLA_Obj_datatype( FLA_Obj matrix )` Extract datatype of matrix.
`int FLA_Obj_length( FLA_Obj matrix )` Extract row dimension of matrix.
`int FLA_Obj_width( FLA_Obj matrix )` Extract column dimension of matrix.
`void *FLA_Obj_buffer( FLA_Obj matrix )` Extract the address where the matrix is stored.
`int FLA_Obj_ldim( FLA_Obj matrix )` Extract the leading dimension for the array in which the matrix is stored.

Partitioning, etc.

```
FLA_Part_2x2( FLA_Obj A, FLA_Obj *ATL, FLA_Obj *ATR,
                         FLA_Obj *ABL, FLA_Obj *ABR, int mb, int nb, FLA_Quadrant quadrant )
```
Partition matrix A into four quadrants where the quadrant indicated by `quadrant` is `mb` $\times$ `nb`.

```
FLA_Merge_2x2( FLA_Obj ATL, FLA_Obj TR,
               FLA_Obj ABL, FLA_Obj BR,   FLA_Obj *A  )
```
Merge a 2×2 partitioning of a matrix into a single view.

```
FLA_Repart_from_2x2_to_3x3 ( FLA_Obj ATL, FLA_Obj ATR,   FLA_Obj *A00, FLA_Obj *A01, FLA_Obj *A02,
                                                         FLA_Obj *A10, FLA_Obj *A11, FLA_Obj *A12,
                             FLA_Obj ABL, FLA_Obj ABR,   FLA_Obj *A20, FLA_Obj *A21, FLA_Obj *A22,
                             int mb, int nb, FLA_Quadrant quadrant )
```
Repartition a 2×2 partitioning of matrix A into a 3×3 partitioning where `mb` $\times$ `nb` submatrix A_{11} is split from the quadrant indicated by `quadrant`.

```
FLA_Cont_with_3x3_to_2x2( FLA_Obj *ATL, FLA_Obj *ATR,   FLA_Obj A00, FLA_Obj A01, FLA_Obj A02,
                                                        FLA_Obj A10, FLA_Obj A11, FLA_Obj A12,
                          FLA_Obj *ABL, FLA_Obj *ABR,   FLA_Obj A20, FLA_Obj A21, FLA_Obj A22,
                          FLA_Quadrant quadrant )
```
Update the 2×2 partitioning of matrix A by moving the boundaries so that A_{11} is added to the quadrant indicated by `quadrant`.

```
FLA_Part_2x1( FLA_Obj A, FLA_Obj *AT,
                         FLA_Obj *AB, int mb, FLA_Side side )
```
Partition matrix A into a top and bottom side where the side indicated by `side` has `mb` rows.

```
FLA_Merge_2x1( FLA_Obj AT,
               FLA_Obj AB,    FLA_Obj *A  )
```
Merge a 2×1 partitioning of a matrix into a single view.

```
FLA_Repart_from_2x1_to_3x1( FLA_Obj AT,   FLA_Obj *A0,
                                          FLA_Obj *A1,
                            FLA_Obj AB,   FLA_Obj *A2, int mb, FLA_Side side )
```
Repartition a 2×1 partitioning of matrix A into a 3×1 partitioning where submatrix A_1 with `mb` rows is split from the side indicated by `side`.

```
FLA_Cont_with_3x1_to_2x1( FLA_Obj *AT,  FLA_Obj A0,
                                        FLA_Obj A1,
                          FLA_Obj *AB,  FLA_Obj A2, FLA_Side side )
```
Update the 2×1 partitioning of matrix A by moving the boundaries so that A_1 is added to the side indicated by `side`.

```
FLA_Part_1x2( FLA_Obj A, FLA_Obj *AL, FLA_Obj *AR, int nb, FLA_Side side )
```
Partition matrix A into a left and right side where the side indicated by `side` has `nb` columns

```
FLA_Merge_1x2( FLA_Obj AL, FLA_Obj AR,    FLA_Obj *A  )
```
Merge a 1×2 partitioning of a matrix into a single view.

```
FLA_Repart_from_1x2_to_1x3( FLA_Obj  AL,             FLA_Obj  AR,
                            FLA_Obj *A0, FLA_Obj *A1, FLA_Obj *A2, int nb, FLA_Side side )
```
Repartition a 1×2 partitioning of matrix A into a 1×3 partitioning where submatrix A_1 with `nb` columns is split from the side indicated by `side`.

```
FLA_Cont_with_1x3_to_1x2( FLA_Obj *AL,            FLA_Obj *AR,
                          FLA_Obj A0, FLA_Obj A1, FLA_Obj A2, FLA_Side side )
```
Update the 1×2 partitioning of matrix A by moving the boundaries so that A_1 is added to the side indicated by `side`.

B.4 Printing the Contents of an Object

```
FLA_Obj_show( char *string1, FLA_Obj A, char *format, char *string2 )
```
Print the contents of A.

B.5 A Subset of Supported Operations

For information on additional operations supported by the libFLAME library, such as Cholesky, LU, and QR factorization and related solvers, visit `http://www.cs.utexas.edu/users/flame/` . Most of the operations are those supported by the Basic Linear Algebra Subprograms (BLAS) [21, 10, 9]. For this reason, we adopt a naming convention that is very familiar to those who have used traditional BLAS routines.

General operations

Note: the name of the FLA_Axpy routine comes from the BLAS routine axpy which stands for double precision alpha times vector x plus vector y. We have generalized this routine to also work with matrices.

Routine / Operation
`FLA_Axpy( FLA_Obj alpha, FLA_Obj A, FLA_Obj B )`
$B := \alpha A + B$.
`FLA_Axpy_x( FLA_Trans trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj B )`
$B := \alpha \text{op}_{\texttt{trans}}(A) + B$.
`FLA_Copy( FLA_Obj A, FLA_Obj B )`
$B := A$.
`FLA_Copy_x( FLA_Trans trans, FLA_Obj A, FLA_Obj B )`
$B := \text{op}_{\texttt{trans}}(A)$.
`FLA_Inv_scal( FLA_Obj alpha, FLA_Obj A )`
$A := \frac{1}{\alpha} A$.
`FLA_Negate( FLA_Obj A )`
$A := -A$.
`FLA_Obj_set_to_one( FLA_Obj A )`
Set all elements of A to one.
`FLA_Obj_set_to_zero( FLA_Obj A )`
Set all elements of A to zero.
`FLA_Random_matrix( FLA_Obj A )`
Fill A with random values in the range $(-1, 1)$.
`FLA_Scal( FLA_Obj alpha, FLA_Obj A )`
$A := \alpha A$.
`FLA_Set_diagonal( FLA_Obj sigma, FLA_Obj A )`
Set the diagonal of A to σI. All other values in A are unaffected.
`FLA_Shift_spectrum( FLA_Obj alpha, FLA_Obj sigma, FLA_Obj A )`
$A := A + \alpha\sigma I$.
`FLA_Swap( FLA_Obj A, FLA_Obj B )`
$A, B := B, A$.
`FLA_Swap_x( FLA_Trans trans, FLA_Obj A, FLA_Obj B )`
$A, B := \text{op}_{trans}(A), \text{op}_{trans}(B)$.
`FLA_Symmetrize( FLA_Uplo uplo, FLA_Conj conj, FLA_Obj A )`
$A := \text{symm}(A)$ or $A := \text{herm}(A)$, where `uplo` indicates whether A is originally stored only in the upper or lower triangular part of `A`.
`FLA_Triangularize( FLA_Uplo uplo, FLA_Diag diag, FLA_Obj A )`
$A := \text{lower}(A)$ or $A := \text{upper}(A)$.

Scalar operations

`FLA_Invert( FLA_Obj alpha )`
$\alpha := 1/\alpha$.
`FLA_Sqrt( FLA_Obj alpha )`
$\alpha := \sqrt{\alpha}$. **Note:** `A` must describe a scalar.

Vector-vector operations

Note: some of the below operations also appear above under "General operations". Traditional users of the BLAS would expect them to appear under the heading "Vector-vector operations," which is why we repeat them.

`FLA_Axpy( FLA_Obj alpha, FLA_Obj x, FLA_Obj y )`
$y := \alpha x + y$. (alpha x plus y.)
`FLA_Copy( FLA_Obj x, FLA_Obj y )`
$y := x$.
`FLA_Dot( FLA_Obj x, FLA_Obj y, FLA_Obj rho )`
$\rho := x^T y$.
`FLA_Dot_x( FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj beta, FLA_Obj rho )`
$\rho := \alpha x^T y + \beta \rho$.
`FLA_Iamax( FLA_Obj x, FLA_Obj k )`
Compute index k such that $\|\chi_k\| = \|x\|_\infty$.
`FLA_Inv_scal( FLA_Obj alpha, FLA_Obj x )`
$x := \frac{1}{\alpha} x$.
`FLA_Nrm1( FLA_Obj x, FLA_Obj alpha )`
$\alpha := \|x\|_1$.
`FLA_Nrm2( FLA_Obj x, FLA_Obj alpha )`
$\alpha := \|x\|_2$.
`FLA_Nrm_inf( FLA_Obj x, FLA_Obj alpha )`
$\alpha := \|x\|_\infty$.
`FLA_Scal( FLA_Obj alpha, FLA_Obj x )`
$x := \alpha x$.
`FLA_Swap( FLA_Obj x, FLA_Obj y )`
$x, y := y, x$.

Matrix-vector operations

As for the vector-vector operations, we adopt a naming convention that is very familiar to those who have used traditional level-2 BLAS routines. The name `FLA_XXYY` encodes the following information:

XX	Meaning
Ge	General rectangular matrix.
Tr	One of the operands is a triangular matrix.
Sy	One of the operands is a symmetric matrix.

YY	Meaning
mv	Matrix-vector multiplication.
sv	Solution of a linear system.
r	Rank-1 update.
r2	Rank-2 update.

```
FLA_Gemv( FLA_Trans trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj x, FLA_Obj beta, FLA_Obj y )
```
$y := \alpha \mathrm{op}_{\mathtt{trans}}(A)x + \beta y$.

```
FLA_Ger( FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj A )
```
$A := \alpha x y^T + A$.

```
FLA_Symv( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj A, FLA_Obj x, FLA_Obj beta, FLA_Obj y )
```
$y := \alpha A x + \beta y$, where A is symmetric and stored in the upper or lower triangular part of `A`, as indicated by `uplo`.

```
FLA_Syr( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj A )
```
$A := \alpha x x^T + A$, where A is symmetric and stored in the upper or lower triangular part of `A`, as indicated by `uplo`.

```
FLA_Syr2( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj A )
```
$A := \alpha x y^T + \alpha y x^T + A$, where A is symmetric and stored in the upper or lower triangular part of `A`, as indicated by `uplo`.

```
FLA_Syr2k( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C )
```
$C := \alpha(\mathrm{op}_{\mathtt{trans}}(A)\mathrm{op}_{\mathtt{trans}}(B)^T + \mathrm{op}_{\mathtt{trans}}(B)\mathrm{op}_{\mathtt{trans}}(A)^T + \beta C$, where C is symmetric and stored in the upper or lower triangular part of `C`, as indicated by `uplo`.

```
FLA_Trmv( FLA_Uplo uplo, FLA_Trans trans, FLA_Diag diag, FLA_Obj A, FLA_Obj x )
```
$x := \mathrm{op}_{\mathtt{trans}}(A)x$, where A is upper or lower triangular, as indicated by `uplo`.

```
FLA_Trmv_x( FLA_Uplo uplo, FLA_Trans trans, FLA_Diag diag,
            FLA_Obj alpha, FLA_Obj A, FLA_Obj x, FLA_Obj beta, FLA_Obj y )
```
Update $y := \alpha \mathrm{op}_{\mathtt{trans}}(A)x + \beta y$, where A is upper or lower triangular, as indicated by `uplo`.

```
FLA_Trsv( FLA_Uplo uplo, FLA_Trans trans, FLA_Diag diag, FLA_Obj A, FLA_Obj x )
```
$x := \mathrm{op}_{\mathtt{trans}}(A)^{-1}x$, where A is upper or lower triangular, as indicated by `uplo`.

```
FLA_Trsv_x( FLA_Uplo uplo, FLA_Trans trans, FLA_Diag diag,
            FLA_Obj alpha, FLA_Obj A, FLA_Obj x, FLA_Obj beta, FLA_Obj y )
```
$y := \alpha \mathrm{op}_{\mathtt{trans}}(A)^{-1}x + \beta y$, where A is upper or lower triangular, as indicated by `uplo`.

Matrix-matrix operations

As for the vector-vector and matrix-vector operations, we adopt a naming convention that is very familiar to those who have used traditional level-3 BLAS routines. `FLA_XXYY` in the name encodes

XX	Meaning
`Ge`	General rectangular matrix.
`Tr`	One of the operands is a triangular matrix.
`Sy`	One of the operands is a symmetric matrix.

YY	Meaning
`mm`	Matrix-matrix multiplication.
`sm`	Solution of a linear system with multiple right-hand sides.
`rk`	Rank-k update.
`r2k`	Rank-2k update.

```
FLA_Gemm( FLA_Trans transA, FLA_Trans transB, FLA_Obj alpha, FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C )
```
$C := \alpha \text{op}_{\texttt{transA}}(A)\text{op}_{\texttt{transB}}(B) + \beta C$.

```
FLA_Symm( FLA_Side side, FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C )
```
$C := \alpha AB + \beta C$ or $C := \alpha BA + \beta C$, where A is symmetric, `side` indicates the side from which A multiplies B, `uplo` indicates whether A is stored in the upper or lower triangular part of `A`.

```
FLA_Syr2k( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C )
```
$C := \alpha(\text{op}_{\texttt{trans}}(A)\text{op}_{\texttt{trans}}(B)^T + \text{op}_{\texttt{trans}}(B)\text{op}_{\texttt{trans}}(A)^T + \beta C$, where C is symmetric and stored in the upper or lower triangular part of `C`, as indicated by `uplo`.

```
FLA_Syrk( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj beta, FLA_Obj C )
```
$C := \alpha \text{op}_{\texttt{trans}}(A)\text{op}_{\texttt{trans}}(A)^T + \beta C$, where C is symmetric and stored in the upper or lower triangular part of `C`, as indicated by `uplo`.

```
FLA_Trmm( FLA_Side side, FLA_Uplo uplo, FLA_Trans trans, FLA_Diag diag, FLA_Obj alpha, FLA_Obj A, FLA_Obj B )
```
$B := \alpha \text{op}_{\texttt{trans}}(A)B$ (`side == FLA_LEFT`) or $B := \alpha B \text{op}_{\texttt{trans}}(A)$ (`side == FLA_RIGHT`). where A is upper or lower triangular, as indicated by `uplo`.

```
FLA_Trmm_x( FLA_Side side, FLA_Uplo uplo, FLA_Trans transA, FLA_Trans transB, FLA_Diag diag,
            FLA_Obj alpha, FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C )
```
$C := \alpha \text{op}_{\texttt{transA}}(A)\text{op}_{\texttt{transB}}(B) + \beta C$ (`side == FLA_LEFT`) or $C := \alpha \text{op}_{\texttt{transB}}(B)\text{op}_{\texttt{transA}}(A) + \beta C$ (`side == FLA_RIGHT`) where A is upper or lower triangular, as indicated by `uplo`.

```
FLA_Trsm( FLA_Side side, FLA_Uplo uplo, FLA_Trans trans, FLA_Diag diag, FLA_Obj alpha, FLA_Obj A, FLA_Obj B )
```
$B := \alpha \text{op}_{\texttt{trans}}(A)^{-1}B$ (`SIDE == FLA_LEFT`) or $B := \alpha B \text{op}_{\texttt{trans}}(A)^{-1}$ (`SIDE == FLA_RIGHT`) where A is upper or lower triangular, as indicated by `uplo`.

```
FLA_Trsm_x( FLA_Side side, FLA_Uplo uplo, FLA_Trans transA, FLA_Trans transB, FLA_Diag diag,
            FLA_Obj alpha, FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C )
```
$C := \alpha \text{op}_{\texttt{transA}}(A)^{-1}\text{op}_{\texttt{transB}}(B) + \beta C$ (`SIDE == FLA_LEFT`) or $C := \alpha \text{op}_{\texttt{transB}}(B)\text{op}_{\texttt{transA}}(A)^{-1} + \beta C$ (`SIDE == FLA_RIGHT`) where A is upper or lower triangular, as indicated by `uplo`.

Bibliography

[1] Satish Balay, William Gropp, Lois Curfman McInnes, and Barry Smith. PETSc 2.0 Users Manual. Technical Report ANL-95/11, Argonne National Laboratory, Oct. 1996. 4.3

[2] Paolo Bientinesi. *Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms*. PhD thesis, 2006. 1.6

[3] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft.*, 31(1):1–26, 2005. submitted. 1.2

[4] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: the FLAME application program interfaces. *ACM Trans. Math. Soft.*, 31(1):27–59, 2005. 1.2

[5] P. D. Crout. A short method for evaluating determinants and solving systmes of linear equations with real or complex coefficients. *AIEE Trans.*, 60:1235–1240, 1941. 1.3

[6] James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[7] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, 8:174–186, 1968. 1.3

[8] E. W. Dijkstra. *A discipline of programming*. Prentice-Hall, 1976. 1.3

[9] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990. 4.3, B.5

[10] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988. 4.3, B.5

[11] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Symposium on Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967. 1.3

[12] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 3rd edition, 1996.

[13] Kazushige Goto and Robert A. van de Geijn. On reducing TLB misses in matrix multiplication. *ACM Trans. Math. Soft.*, 2006. To appear. 5.19

[14] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[15] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science. Springer-Verlag, 1992. 2.3, 2.5

[16] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001. 1.2

[17] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. A family of high-performance matrix multiplication algorithms. In Vassil N. Alexandrov, Jack J. Dongarra, Benjoe A. Juliano, René S. Renner, and C.J. Kenneth Tan, editors, *Computational Science - ICCS 2001, Part I*, Lecture Notes in Computer Science 2073, pages 51–60. Springer-Verlag, 2001. 5.26

[18] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, second edition, 2002.

[19] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pages 576–580, October 1969. 1.3

[20] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, MA, 2nd edition, 1994.

[21] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, Sept. 1979. 4.3, B.5

[22] C. Moler, J. Little, and S. Bangert. *Pro-Matlab, User's Guide*. The Mathworks, Inc., 1987. 4.2

[23] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996. 4.3

[24] G. W. Stewart. *Introduction to Matrix Computations*. Academic Press, Orlando, Florida, 1973.

[25] G. W. Stewart. *Matrix Algorithms Volume 1: Basic Decompositions*. SIAM, 1998. 6.2.2

[26] Gilbert Strang. *Linear Algebra and its Application, Third Edition*. Academic Press, 1988.

[27] Lloyd N. Trefethen and III David Bau. *Numerical Linear Algebra*. SIAM, 1997.

[28] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package.* The MIT Press, 1997. 4.3

[29] David S. Watkins. *Fundamentals of Matrix Computations.* John Wiley & Sons, Inc., New York, 2nd edition, 2002. 6.3

Index

$\backslash$, 117
:=, 9
ˆ, 15, 29
$\Longrightarrow$, 15
$\neg$, 15
$\star$, 134
$\wedge$, 14
$\cdot$, 10
:=, 10
=, 10
{ }, 13
$BASE/, ix

absolute value, 122
access
    by quadrants, 27
    column-wise, 27
    row-wise, 27
ADD, 25
    cost, 88
algorithm
    annotated, 15, 16
    blocked, 6, 7, 51
    column-oriented, 49
    correct, 5
    correctness, 13
    correctness of, 13
    derivation, 9
    eager, 49
    expressing, 9
    GEMV
        blocked Variant 1, 62
        blocked Variant 3, 63
    goal-oriented derivation of, 16
    high-performance, 6, 87
    lazy, 50
    loop-based, 5
    recursive, 8
    stability, 8
    TRSV
        blocked Variant 2, 56
        unblocked Variant 1, 50
        unblocked Variant 2, 50
    typesetting, 2
algorithmic block size, 54, 66
algorithmic variant, 5
annotated algorithm, 15
APDOT, 10, 11, 22, 25
    cost, 22, 88
API, 5, 61
    FLAME@lab, 62
    FLAME/C, 70

Appendices, 136
architecture, 5, 87
  cache-based, 7
arithmetic product, 10
assertion, 13
assignment, 10
AXPY, 25, 144
  cost, 88

back substitution, 113
becomes, 10
bidimensional partitioning
  FLAME@LAB, 68
  FLAME/C, 81
blank worksheet, 24
block size, 52
  algorithmic, 54
Boolean expression, 13
Bottom, 11

C, 5, 61
cache, 7
cache memory, 88
casting in terms of
  GEMM, 87, 105
  GEMV, 51
Cholesky factorization, 113
  algorithms, 136
  invariants, 135
clock
  cycle, 7
  rate, 7
  speed, 8
coefficient matrix, 44
column, 28
column-oriented algorithm, 49
command, 13
  sequence, 13
complex valued, 28

computer architecture, 5
conformal, 33
Continue with ..., 12
correctness, 13
  proof, 15
cost
  ADD, 88
  analysis, 20
  APDOT, 20, 88
  AXPY, 88
  GEMM, 88
  GEMV, 38, 41, 88
  invariant, 23
  GER, 88
  SCAL, 88
  DOT, 88
  SYMM, 108
  TRSV, 88
    unblocked Variant 1, 50
    unblocked Variant 2, 49
  worksheet, 23
cost-invariant, 23
Cost-so-far, 23
CPU, 87
Crout, 5
C_{sf}, 23

datatype, 71
derivation, 9
  formal, 1
  goal-oriented, 9
  systematic, 9
descriptor, 71
diagonal, 2, 44
Dijkstra, 5
DOT, 25
download
  lab, 70
  FLAMEC, 77

e_j, 30
element, 28
element growth, 122
equality, 10
equations, 43

factorization
    Cholesky, 113
    LU, 113
fetch, 88
final result, 17
FLA_Cont_with_1x3_to_1x2
    FLAME@LAB, 68
    FLAME/C, 81
FLA_Cont_with_3x1_to_2x1
    FLAME@LAB, 65
    FLAME/C, 79
FLA_Cont_with_3x3_to_2x2
    FLAME@LAB, 69
    FLAME/C, 83
FLA_Finalize, 71
FLA_Init, 71, 141
lab
    download, 70
FLAME project, 2
FLAME@LAB, 62
    bidimensional partitioning, 68
    FLA_Cont_with_1x3_to_1x2, 68
    FLA_Cont_with_3x1_to_2x1, 65
    FLA_Cont_with_3x3_to_2x2, 69
    FLA_Part_1x2, 66
    FLA_Part_2x1, 63
    FLA_Part_2x2, 68
    FLA_Repart_1x2_to_1x3, 66
    FLA_Repart_2x1_to_3x1, 64
    FLA_Repart_2x2_to_3x3, 68
    horizontal partitioning, 62
    vertical partitioning, 66
FLAMEC
    download, 77
FLAME/C, 139
    bidimensional partitioning, 81
    creators, 141
    destructors, 141
    finalizing, 71, 141
    FLA_Cont_with_1x3_to_1x2, 81
    FLA_Cont_with_3x1_to_2x1, 79
    FLA_Cont_with_3x3_to_2x2, 83
    FLA_Finalize, 71
    FLA_Init, 71, 141
    FLA_Merge_1x2, 84
    FLA_Merge_2x1, 84
    FLA_Merge_2x2, 84
    FLA_Obj_attach_buffer, 73
    FLA_Obj_buffer, 73
    FLA_Obj_create, 71
    FLA_Obj_create_without_buffer, 73
    FLA_Obj_datatype, 73
    FLA_Obj_free, 72
    FLA_Obj_free_without_buffer, 75
    FLA_Obj_ldim, 73
    FLA_Obj_length, 73
    FLA_Obj_show, 75
    FLA_Obj_width, 73
    FLA_Part_1x2, 79
    FLA_Part_2x1, 77
    FLA_Part_2x2, 81
    FLA_Repart_1x2_to_1x3, 79
    FLA_Repart_2x1_to_3x1, 78
    FLA_Repart_2x2_to_3x3, 83
    horizontal partitioning, 77
    initializing, 71, 141
    inquiry routines, 141
    manipulating objects, 71, 141
    object
      print contents, 75, 143
    operations
      Cholesky factorization, 143

general, 144
LU factorization, 143
matrix-matrix, 147
matrix-vector, 146
QR factorization, 143
scalar, 145
solvers, 143
vector-vector, 145
parameters, 139
table, 140
reference, 139
TRSV
blocked Variant 2, 57
vertical partitioning, 79
FLA_Merge_1x2, 84
FLA_Merge_2x1, 84
FLA_Merge_2x2, 84
MINUS_ONE, 72
FLA_Obj_attach_buffer, 73
FLA_Obj_buffer, 73
FLA_Obj_create, 71
FLA_Obj_create_without_buffer, 73
FLA_Obj_datatype, 73
FLA_Obj_free_without_buffer, 75
FLA_Obj_free, 72
FLA_Obj_ldim, 73
FLA_Obj_length, 73
FLA_Obj_show, 75
FLA_Obj_width, 73
FLA_ONE, 72
FLA_Part_1x2
FLAME@LAB, 66
FLAME/C, 79
FLA_Part_2x1
FLAME@LAB, 63
FLAME/C, 77
FLA_Part_2x2
FLAME@LAB, 68
FLAME/C, 81
FLA_Repart_1x2_to_1x3
FLAME@LAB, 66
FLAME/C, 79
FLA_Repart_2x1_to_3x1
FLAME@LAB, 64
FLAME/C, 78
FLA_Repart_2x2_to_3x3
FLAME@LAB, 68
FLAME/C, 83
FLATEX, 5, 20, 21
\FlaOneByThree..., 21
\FlaOneByTwo, 21
\FlaThreeByOne..., 21
\FlaThreeByThree..., 21
\FlaTwoByOne, 21
\FlaTwoByTwo, 21
FLA_ZERO, 72
floating-point arithmetic, 123
flop, 7, 20, 88
Floyd, 5
formal derivation, 1, 5
Formal Linear Algebra Methods Environment, 2
Fortran, 5
forward substitution, 49
function, 29
Fundamental Invariance Theorem, 14

Gauss, 5
Gauss transform, 120
accumulated, 121
Gaussian elimination, 1, 115--117
GEBP, 100
algorithm, 101
shape, 93
GEMM, 7, 111
blocked Variant 1, 95
blocked Variant 2, 96
blocked Variant 3, 98
cost, 88, 91

definitions, 90
naming, 93
partitioned
product, 91
performance, 98
PME 1, 94
PME 2, 95
PME 3, 97
properties, 89
shape, 92, 93
GEBP, 93
GEMM, 93
GEMP, 93
GEPB, 93
GEPDOT, 93
GEPM, 93
GEPP, 93
naming, 93
unblocked Variant 1, 95
unblocked Variant 2, 96
unblocked Variant 3, 98
GEMP
shape, 93
GEMV, 31, 39, 41, 59
algorithm
blocked Variant 1, 62
blocked Variant 3, 63
casting in terms of, 51
cost, 88
derivation, 34
loop-invariants, 34
PMEs, 34
shape, 93
Variant 1
algorithm, 39
cost, 41
worksheet, 36
Variant 3
worksheet, 40
GEPB, 100
shape, 93
GEPDOT, 100
shape, 93
GEPM
shape, 93
GEPP
algorithm, 102
shape, 93
GER
shape, 93
GER, 59
GFLOPS, 7
GHz, 8
gigaflops, 7
goal-oriented derivation, 16

handle, 71
Haskell, 6
high performance, 6
high-performance
GEMM, 7
implementation, 100
Hoare, 5
triple, 13
horizontal partitioning
FLAME@LAB, 62
FLAME/C, 77

identity matrix, 31, 138
implies, 15
induction hypothesis, 23
initialization, 18
inner product, 9
invariant
cost, 23
invariants
TRSV, 47
inverse, 115

invertible, 114
INVSCAL, 25
iteration, 13

LABVIEW G, 6
LABVIEW MATHSCRIPT, 5, 62
  M-script, 5
$L_{ac,k}$, 121
LaTeX, 5, 20
lazy, 50
leading dimension, 72
leading principle submatrix, 115
letters
  lowercase Greek, 28, 29, 137
  lowercase Roman, 28, 29, 137
  uppercase Greek, 29, 137
linear algebra operations, 1
linear combination, 30
linear independent, 31
linear system, 43
  matrix form, 44
  triangular, 1, 44
linear transformation, 29, 90
  composition, 90
lines
  thick, 2
logical
  and, 14
  negation, 15
loop, 10, 13, 14
  verifying, 13
loop-based, 5
loop-body, 13
loop-guard, 13
  choosing, 17
loop-invariant, 15, 16
  determining, 16
  feasible, 108
  infeasible, 108
  potential, 108
lower triangular matrix, 44
  unit, 1, 132
$L\backslash U$, 117
LU factorization, 1, 113
  algorithms
    LU_BLK (all), 4
    LU_UNB (all), 4
    LU_UNB_VAR5, 2
  cost, 118, 120
  invariants, 119
  loop-invariants, 118
  performance, 7
  PME, 118, 129
  with partial pivoting, 123
    algorithms, 133
    invariants, 130
LU_BLK, 4
LU_UNB, 4
LU_UNB_VAR5, 2, 6

$m(\cdot)$, 11, 33, 138
M-script, 5, 61, 62
  LABVIEW MATHSCRIPT, 5, 62
  MATLAB, 5, 62
  OCTAVE, 5, 62
magnitude, 122
map, 29
MATHEMATICA, 6
mathematical induction
  principle of, 22
MATLAB, 5, 62
  M-script, 5
matrix, 27--29
  augmented, 117
  diagonal, 2, 44
  element, 28
  permutation, 124
  rank, 31

SPD, 132
symmetric, 105
unit lower triangular, 1, 115, 132
upper triangular, 1
matrix-matrix operations, 110
matrix-vector operation, 27
matrix-vector operations, 27, 58
matrix-vector product, 27, 33
rank-1 update, 27, 40
solution of triangular system, 27, 43
table of, 59, 111
triangular solve, 43
matrix-vector product, 27, 29, 33
definition, 30
distributive property, 31
GEMV
PMEs, 32
via APDOTs, 31
via AXPYs, 32
memop, 88
memory, 7, 87
bandwidth, 7
bottleneck, 88
cache, 88
hierarchy, 87
model, 87, 89
multiplier, 114

$n(\cdot)$, 33, 138
n_b, 52
nonsingular, 114
notation, 2
numerical stability, 8

object, 71
buffer, 73
datatype, 73
ldim, 73
leading dimension, 73
length, 73
manipulating, 71, 141
width, 73
OCTAVE, 5, 62
M-script, 5
operation
linear algebra, 1
matrix-matrix, 88
matrix-vector, 27, 88
specification, 16
vector-vector, 27, 88
original contents, 15

P, 13
P_{after}, 19
P_{before}, 19
P_{cons}, 34
P_{inv}, 14, 15
P_{post}, 13, 15
P_{pre}, 13, 15
P_{struct}, 46
$P(\cdot)$, 124, 125
partial pivoting, 122, 123, 125
partial result, 17
Partition ..., 11
partitioning
bidimensional
FLAME@LAB, 68
FLAME/C, 81
conformal, 33
horizontal
FLAME@LAB, 62
FLAME/C, 77
vertical
FLAME@LAB, 66
FLAME/C, 79
peak performance, 8
performance, 6
LU factorization, 7

peak, 8
permutation, 124
PivIndex(·), 125
pivot row, 123
PME, 17
postcondition, 13, 15, 16
precondition, 13, 15, 16
predicate, 13
Preface, v
Principle of Mathematical Induction, 22
processor
model, 87
programming language
C, 5, 61
Fortran, 5
Haskell, 6
LabView G, 6
M-script, 5, 61
Mathematica, 6
proof of correctness, 15

quadrant, 68
Quick Reference Guide, 139

RAM, 87
rank, 31
rank-1 update, 27, 40
GER, 40
cost, 88
definition, 40
PMEs, 42
real number, 20, 28
recursive, 8, 55
reference, 77
registers, 88
Repartition ..., 12
right-hand side vector, 44

SCAL, 25
cost, 88
scalar, 28
DOT
cost, 88
shape
GEBP, 93
GEMM, 93
GEMP, 93
GEMV, 93
GEPB, 93
GEPDOT, 93
GEPM, 93
GEPP, 93
GER, 93
side, 63
small, 92
SPD, 113
S, 13
stability, 8
analysis, 8
state, 13
after moving the thick lines, 19
determining, 19
after repartitioning, 18
determining, 18
store, 88
SyLw(·), 105
SyLw(·), 132
SYMM, 111
blocked Variant 1, 110
cost, 108
loop-invariants, 109
performance, 108
PME 1, 107
symmetric positive definite, 113
SYMV, 59
SYR, 59
SYR2, 59
SYR2K, 111
SYRK, 111

T (transposition), 28
textual substitution, 19
thick lines, 2, 11
Top, 11
transform
  Gauss, 120
transposition, 28
triangular linear system, 1, 27, 44
  solution, 27
triangular matrix
  lower, 44
  unit lower, 115
  upper, 44
triangular solve, 43
TRIL (A), 134
TRILU$(\cdot)$, 132
TrLw$(\cdot)$, 46
TRMM, 111
TRMV, 59
TRSV, 44, 59
  algorithm
    blocked Variant 2, 56
    unblocked Variant 1, 50
    unblocked Variant 2, 50
  blocked, 51
  cost, 88
    unblocked Variant 1, 50
    unblocked Variant 2, 49
  FLAME/C
    blocked Variant 2, 57
  loop-invariants, 47
  unblocked, 43
  worksheet
    blocked Variant 2, 53
    unblocked Variant 2, 45
TrUp$(\cdot)$, 46
TRSM, 111
TSoPMC, ix
typesetting, 2

unit basis vector, 30, 91, 138
unit lower triangular matrix, 1, 132
unknowns, 43
update, 20
  determining, 20
upper triangular
  strictly, 105
upper triangular matrix, 1, 44

variable
  scalar, 28
  state of, 15
  vector, 28
variant
  algorithmic, 5
  blocked, 27
vector, 9, 28
  addition, 25
  element, 10, 28
  inverse scaling, 25
  length, 11
  scaling, 25
  unit basis, 30
vector of unknowns, 44
vector-vector operation, 25, 27
  APDOT, 10, 22
vector-vector operations, 24
  table of, 25
vertical partitioning
  FLAME@LAB, 66
  FLAME/C, 79
view, 77

webpages, viii
while ..., 13
wiki, ix
worksheet, 23, 24
  blank, 24
  TRSV

blocked Variant 2, 53
unblocked Variant 2, 45
www.linearalgebrawiki.org, ix