SOFTWARE ENGINEERING
ABSTRACTIONS
FOR A NUMERICAL
LINEAR ALGEBRA LIBRARY

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2012

By
Zixu Song
School of Computer Science
# Contents

Abstract 11  
Declaration 12  
Copyright 13  
Acknowledgements 14  

1 Introduction 15  
1.1 Overview 15  
1.2 Contributions 17  
1.3 Thesis Outline 18  

2 Numerical Linear Algebra 21  
2.1 Preliminary 22  
2.1.1 Matrix 22  
2.1.2 Matrix Operations 23  
2.1.3 Matrix Properties 25  
2.1.4 Storage Format 26  
2.1.5 Summary 27  
2.2 An Historical Overview on the Design of Existing NLA Libraries 28  
2.2.1 Traditional Libraries 28  
2.2.2 Object-Oriented Numerical Linear Algebra Libraries 33  
2.2.3 Summary 37  
2.3 Summary of Chapter 38  

3 Separation of Concerns 39  
3.1 Building a Complex System 40  
3.1.1 Component-Based Development 40
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Notations and definitions of some basic matrix operations.</td>
<td>24</td>
</tr>
<tr>
<td>2.2</td>
<td>Matrix properties and definitions.</td>
<td>25</td>
</tr>
<tr>
<td>2.3</td>
<td>The specification of matrix matrix multiplication [40].</td>
<td>31</td>
</tr>
<tr>
<td>2.4</td>
<td>Some specifications for solving systems of linear equations in LA-PACK.</td>
<td>32</td>
</tr>
<tr>
<td>2.5</td>
<td>Supported operations of JAMA.</td>
<td>34</td>
</tr>
<tr>
<td>3.1</td>
<td>Separation of concern metrics.</td>
<td>52</td>
</tr>
<tr>
<td>3.2</td>
<td>Coupling metrics.</td>
<td>52</td>
</tr>
<tr>
<td>3.3</td>
<td>Cohesion metrics.</td>
<td>52</td>
</tr>
<tr>
<td>3.4</td>
<td>Size metrics.</td>
<td>53</td>
</tr>
<tr>
<td>3.5</td>
<td>Three Types of Dependence.</td>
<td>56</td>
</tr>
<tr>
<td>4.1</td>
<td>Supported operations of JOOLALA.</td>
<td>76</td>
</tr>
<tr>
<td>6.1</td>
<td>Separation of concern metrics for JAMA.</td>
<td>123</td>
</tr>
<tr>
<td>6.2</td>
<td>Separation of concern metrics for JOOLALA.</td>
<td>123</td>
</tr>
<tr>
<td>6.3</td>
<td>Coupling and cohesion metrics for JAMA.</td>
<td>123</td>
</tr>
<tr>
<td>6.4</td>
<td>Coupling and cohesion metrics for JOOLALA.</td>
<td>123</td>
</tr>
<tr>
<td>6.5</td>
<td>Size metric for JAMA.</td>
<td>124</td>
</tr>
<tr>
<td>6.6</td>
<td>Size metric for JOOLALA.</td>
<td>124</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Three dimensions to discover scientific advances. ............... 16
1.2 The relationships between the different chapters. ................ 20

2.1 The algorithm for matrix-matrix multiplication $C = A \times B$ where both $A$ and $B$ are square dense matrices. ............... 26
2.2 The algorithm for matrix-matrix multiplication $C = A \times B$ where both $A$ and $B$ are upper triangular. ................ 26
2.3 Class diagram of LAPACK++ class Matrix. ....................... 35
2.4 Class diagram of OOLALA in [87]. ............................. 37

3.1 An example of code scattering from [81]. ....................... 43
3.2 The dependencies in $A$. ...................................... 60
3.3 Matrices $A$ and $B$ stored in column-major format. ........... 63
3.4 Matrix blocked by submatrices ($NB \times NB$). ............... 64
3.5 Removing an affine IF-ELSE statement by introducing additional loop bounds. ......................... 70
3.6 A Cholesky factorisation kernel from [19]. ..................... 71
3.7 Abstract syntax tree of the SCoP in Figure 3.6 from [19]. .... 73

4.1 Various method definitions in class LowerTriangularProperty. ... 77
4.2 Addition method definition in class DenseProperty. ............ 78
4.3 Addition method definition in class BandedProperty. .......... 78
4.4 Pseudo-code for implementing matrix operations with AOP. ... 78
4.5 Construct for dense property in aspect MatrixOperationConstruct-Aspect. .............................................. 80
4.6 Element assignment for matrix addition in aspect MatrixOperation-AssignmentAspect. ........................................ 80
4.7 AspectJ Version of Matrix Operation Declarations. .......... 82
4.8 F90 BLAS dsymm code fragment. .............................. 83
4.9 F90 BLAS dgemm code fragment.
4.10 Duplicate calls to checkSize in JOOLALA.
4.11 Duplicate try-catch clauses at different places in JAMA.
4.12 Implementing the checkSize concern for JOOLALA with AspectJ.
4.13 Pre-condition checks for subroutine sgemm.f.
4.14 Pre-condition checks for subroutine cgemm.f.
4.15 Pre-condition checks for subroutine ssymm.f.
4.16 Code fragment of aspect DealWithSelfOperation.

5.1 A rectangular band matrix with $m > n$.
5.2 An algorithm for matrix-matrix multiplication $C = A \times B$ where both $A$ and $B$ are dense.
5.3 An algorithm for matrix-matrix multiplication $C = A \times B$ where $A$ is upper triangular and $B$ is dense.
5.4 An algorithm for matrix-matrix multiplication $C = A \times B$ where $A$ is upper triangular and $B$ is lower triangular.
5.5 The transformed algorithm for matrix-matrix multiplication $C = A \times B$ where $A$ is upper triangular and $B$ is lower triangular.
5.6 An algorithm for matrix-matrix multiplication $C = A \times B$ where both $A$ and $B$ are upper triangular.
5.7 The start values in $A$ and $B$.
5.8 Band matrix-matrix multiplication with min/max conditions.
5.9 An algorithm for matrix-matrix multiplication on band matrices.
5.10 The iteration space of a specific band matrix-matrix multiplication.
5.11 A loop nest example.
5.12 Two nodes generated from the original loop nest.
5.13 Two nodes generated from the original loop nest after merging inequalities.
5.14 An example of a tree showing how nodes are connected.
5.15 The class structure of the source-to-source transformer.
5.16 A block example iterating a subspace of the polyhedron.
5.17 Multiplication of two lower triangular matrices.
5.18 Product of a lower triangular matrix and a column vector.

6.1 Execution time in nanoseconds of operations in JAMA and AspectJAMA.
6.2 Execution time in nanoseconds for methods transpose, scale and minor in JOOLALA and Aspect-JOOLALA. .......................... 127
6.3 Execution time in nanoseconds for methods add, sub and mult in JOOLALA and Aspect-JOOLALA. .......................... 128
6.4 Execution time in nanoseconds for methods transpose, scale and minor in JOOLALA and Aspect-JOOLALA without Grouping Matrix Operations. .......................... 130
6.5 Execution time in nanoseconds for methods add, sub and mult in JOOLALA and Aspect-JOOLALA without Grouping Matrix Operations. .......................... 131
6.6 DGEMM and DTRMM code fragments. .......................... 134
6.7 A DUOPGBMM code fragment that corresponds to the code in Figure 6.6. 135
6.8 The number of floating point operations to be performed for matrix-matrix multiplication with bandwidth 0 to 999. .......................... 138
6.9 Execution time in microseconds of matrix-matrix multiplication with bandwidth 0 to 999 for DUOPGBMM and DGBMM. .......................... 138
6.10 Execution time in microseconds of matrix-matrix multiplication with bandwidth 0 to 999 for DGBMM and AT4_DGBMM. .......................... 139
6.11 Execution time in microseconds of matrix-matrix multiplication with bandwidth 0 to 999 for AT4_DGEMM and ATLAS_DGBMM. .......................... 140
6.12 Level 2 cache misses of matrix-matrix multiplication with bandwidth 0 to 999 for DGBMM, AT4_DGEMM and ATLAS_DGBMM. .......................... 141
6.13 Comparing DGBMM with DGEMM and DTRMM for dense and triangular matrix-matrix multiplication, respectively. .......................... 142

A.1 Histograms of 100 samples for matrix addition $B \leftarrow A + B$. .......................... 170
A.2 Histograms of 1000 samples for matrix addition $B \leftarrow A + B$. .......................... 170
A.3 Histograms of 10000 samples for matrix addition $B \leftarrow A + B$. .......................... 171
A.4 Histograms of 100 samples for matrix addition $B \leftarrow A + B$. .......................... 172
A.5 Histograms of 1000 samples for matrix addition $B \leftarrow A + B$. .......................... 172
A.6 Histograms of 10000 samples for matrix addition $B \leftarrow A + B$. .......................... 173
A.7 Histograms of 100 samples for dense-dense matrix multiplication $C \leftarrow A \times B$. .......................... 173
A.8 Histograms of 1000 samples for dense-dense matrix multiplication $C \leftarrow A \times B$. .......................... 174
A.9 Histograms of 10000 samples for dense-dense matrix multiplication

\[ C \leftarrow A \times B. \]
Abstract

This thesis aims at building a numerical linear algebra library with appropriate software engineering abstractions. Three areas of knowledge, namely, Numerical Linear Algebra (NLA), Software Engineering and Compiler Optimisation Techniques, are involved. Numerical simulation is widely used in a large number of distinct disciplines to help scientists understand and discover the world. The solutions to frequently occurring numerical problems have been implemented in subroutines, which were then grouped together to form libraries for ease of use. The design, implementation and maintenance of a NLA library require a great deal of work so that the other two topics, namely, software engineering and compiler optimisation techniques have emerged. Generally speaking, these both try to divide the system into smaller and controllable concerns, and allow the programmer to deal with fewer concerns at one time.

Band matrix operation, as a new level of abstraction, is proposed for simplifying library implementation and enhancing extensibility for future functionality upgrades. Iteration Space Partitioning (ISP) is applied, in order to make the performance of this generalised implementation for band matrices comparable to that of the specialised implementations for dense and triangular matrices. The optimisation of ISP can be either programmed using the pointcut-advice model of Aspect-Oriented Programming, or integrated as part of a compiler. This naturally leads to a comparison of these two different techniques for resolving one fundamental problem.

The thesis shows that software engineering properties of a library, such as modularity and extensibility, can be improved by the use of the appropriate level of abstraction, while performance is either not sacrificed at all, or at least the loss of performance is limited. In other words, the perceived trade-off between the use of high-level abstraction and fast execution is made less significant than previously assumed.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the “Copyright”) and s/he has given The University of Manchester the right to use such Copyright for any administrative, promotional, educational and/or teaching purposes.

ii. Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulations of the John Rylands University Library of Manchester. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.

iii. The ownership of any patents, designs, trade marks and any and all other intellectual property rights except for the Copyright (the “Intellectual Property Rights”) and any reproductions of copyright works, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and exploitation of this thesis, the Copyright and any Intellectual Property Rights and/or Reproductions described in it may take place is available from the Head of School of School of Computer Science (or the Vice-President).
Acknowledgements

I would like express my deepest gratitude to my supervisor, Professor John Gurd, for his kindly supervision and guidance during my study. As an overseas student whose mother tongue is not English, he shows great patience and understanding to the difficulties I encountered. In particular, when I feel lost and was not sure about my research direction, it is him who helped me to rebuild my confidence to finish the work. Without his encouragement and inspiration, I would not reach this far.

I would also like to thank Dr. Len Freeman for his supervision when Professor Gurd is in America. During that time, he helped me to understand the mathematical discipline of this work. I really appreciate all your support in both research and life.

I gratefully thank Professor Sven Hammerling and Dr. Mikel Lujan for their advices on NLA libraries.

During more than three years of research, I have enjoyed the company of the members of Centre for Novel Computing. Thanks to Chanwit Kaewkasi, Chenchen Xi, Wuxiang Ge and Fu Chen for their ideas and thoughts during our discussions.

Many thanks to my parents who have supported me throughout my studies. I would also like to express my thanks to my wife, for her supports and encouragement in many ways.

Finally, I would like to thank Great Britain-China Educational Trust for their financial support for my writing-up period.
Chapter 1

Introduction

This thesis introduces some novel ideas for building a Numerical Linear Algebra (NLA) library with abstractions, and compares its performance and some software engineering properties with other existing libraries. It attempts to form two better bridges, where

- one is between Computational Science and Engineering (CS & E) and Software Engineering, i.e., reusing functionalities for developing computationally intensive applications; and

- the other is between Software Engineering and Compiler Optimisation, for achieving performance benefits from two completely different viewpoints.

1.1 Overview

The first area encountered in this thesis is CS & E. This discipline offers scientists and engineers a collection of tools, theories and techniques to solve mathematical models of problems on a computer. Aerospace engineers simulate the flow of air about an aircraft, which passes through the atmosphere. Meteorologists predict weather and even climate changes on earth by simulating the air and ocean environments. It is difficult to find an area of science or engineering modelling without numerical simulation on computers. Traditionally, scientific advances are discovered by both theoretical and experimental approaches. However, as the use of simulation becomes more and more widely accepted, many people believe that the discovery of scientific advances can be extended to have an additional dimension [82] (see Figure 1.1).
Almost all numerical simulations in physics, mechanics, chemistry, engineering and many other areas involve numerical linear algebra (NLA), i.e. matrix computations. Since the 1950s, the NLA community has developed a large number of subroutines and grouped them into libraries for computing NLA operations. It is clear that a library approach has the benefit of reusability: recurring matrix operations are handled by libraries. Users do not need to write code themselves, unless there are no available libraries meeting the user requirements.

The second area involved in this thesis is software engineering. Although the mathematical definitions of many matrix operations are straightforward, from the programming point of view, the attained performance and efficiency may vary considerably. A matrix can be stored in different storage formats, usually by exploiting the structure of its non-zero elements. As the result of any number plus or times zero is known, only the non-zero elements need be calculated and stored. This enables two savings, i.e., the total number of operations on elements and memory space are both reduced.

It is clear that developing a NLA library with comprehensive functionalities requires a great deal of work, in particular factors such as performance and stability have to be considered at the same time as correct functionality. Therefore,
the ideas of finding a correct abstraction and reusing functionalities have been the main concerns though the development of various NLA libraries. Traditionally, such a library has been designed by a top-down methodology and implemented in an imperative language such as Fortran 77. The low level operations in the implementation are reused by the high level operations through procedure calls. For example, *Linear Algebra PACKage* (LAPACK) [12] is implemented in a way where as much as possible of the computations are performed by calls to *Basic Linear Algebra Subprograms* (BLAS). Along with the development of *Object Oriented Programming* with *Design Patterns* [51], other reuse techniques have been proposed, including *Inheritance*, *Object Composition* and *Parameterised Types*. Moreover, the *Pointcut-Advice model of Aspect Oriented Programming* (AOP) provides a reuse mechanism defined in an abstraction called “aspect”, to universally specify what to do at many non-local places. The procedure call and pointcut-advice model are used and discussed in more detail in later chapters.

This thesis will apply some abstractions for building a NLA library with software engineering benefits. However, the use of high level abstraction must not compromise performance, which is one of the most important properties in a NLA library. To keep the performance overhead at a limited level, the third area involved is compiler optimisation, which transforms the original code into a semantically equivalent code which has improved performance. If applied properly, on the one hand, the software engineering properties such as modularity or extensibility are improved by using appropriate abstractions, but at the cost of potential performance penalties; on the other hand, the performance overheads that may be introduced by the abstractions are removed by compiler optimisation techniques.

### 1.2 Contributions

The first contribution of this thesis is that three types of crosscutting concerns that are natural in a NLA library have been identified. Each of them corresponds to one or more *crosscutting concern sorts*, as introduced in [88]. Methodologies to address these crosscutting concerns have been proposed using AOP. The power of these solutions can be shown by modularity and performance evaluation, which are measured by a suite of AOP metrics and benchmarks, respectively. To the best of the author’s knowledge, this is the first attempt to modularise crosscutting
concerns within a NLA library by using aspect-oriented techniques. The evaluation results show that the modularity of the system can be improved significantly, without compromising the performance.

Secondly, band matrix operations, in particular band matrix multiplication, are proposed for NLA libraries. Previous NLA libraries usually treat dense and triangular matrices as “general” matrices and implement “specific” matrix operations for them, respectively. However, mathematically speaking, they are special band matrices and thus the generalised algorithms for band matrix operations can be applied for solving the specialised problems. This design results in improvement of the modularity and extensibility. Moreover, by using a special compiler optimisation technique, namely iteration space partitioning, the algorithm of the multiplication of two matrices with arbitrary upper and lower bandwidths is divided into a number of blocks, each of which deals with a subspace of fixed size in a three dimensional space.\footnote{The iteration space of dense matrix multiplication is a cuboid, whereas that of band matrix multiplication is a polyhedron. Once the bandwidths of the input matrices are given, the size of each subspace is fixed.} This division leads to two benefits. On the one hand, the condition checks involving a great number of integer instructions that often come with band matrix multiplication can be removed; on the other hand, because the size of each block is fixed, i.e., the non-zero structure of each block has been fully investigated, other transformations such as loop tiling can be applied efficiently for each block.

Last, the pointcut-advice model of AOP is compared with compiler optimisation techniques for performance improvements. A review for analysing the similarities and differences between the two mechanisms from different viewpoints is provided via several case studies.

1.3 Thesis Outline

The thesis is organised in the following manner:

\textbf{Chapter 2} reviews the NLA operations and some existing NLA libraries. The characteristics of matrix operations and difficulties for developing NLA libraries are firstly described. It then introduces some existing NLA libraries, which are each representative of one-or-more different reusing mechanisms,
namely, procedure call, class inheritance, object composition and parameterised types.

**Chapter 3** introduces some key concepts for building a system. A different abstraction level is proposed, including the concepts of concern, separation of concerns and crosscutting concerns. Two methodologies, namely aspect-oriented programming and compiler transformation techniques are presented, together with examples.

**Chapter 4** presents three types of crosscutting concerns that are natural in many NLA libraries. Proposed solutions are given in order to modularise these concerns.

**Chapter 5** gives a novel view of matrix operations by the support of band matrix operations, in particular band matrix multiplication. An algorithm for band matrix multiplication is provided by introducing condition checks of array indices on the corresponding algorithm for dense matrices, and an implementation of iteration space partitioning for performing further optimisation, namely, eliminating condition checks, is introduced. Moreover, how to provide an efficient band matrix multiplication by reusing existing BLAS subroutines is revealed.

**Chapter 6** evaluates the two contributions presented in Chapters 4 and 5 from the point of view of software engineering properties and performance.

**Chapter 7** concludes the similarities and differences between two different mechanisms for the same objective, namely improving system performance. The two mechanisms are the pointcut-advice model in AOP and compiler optimisations performed by a compiler.

A graph explaining the relationships between the chapters of the thesis is given in Figure 1.2.
Figure 1.2: The relationships between the different chapters.
Chapter 2

Numerical Linear Algebra

“. . .linear algebra is everywhere in numerical simulations, often well hidden for the average user, but always crucial in terms of performance and efficiency. Almost all numerical computations in physics, mechanics, chemistry, engineering, economics, finance, etc., involve numerical linear algebra, i.e., computations involving matrices.” [10]

“Numerical linear algebra is a very important subject in numerical analysis because linear problems occur so often in applications. It has been estimated, for example, that about 75% of all scientific problems require the solution of a system of linear equations at one stage or another.” [69]

Matrix computations can be mainly divided into two groups: basic matrix operations, such as matrix multiplication, and complex matrix operations, such as solving a system of linear equations or computing eigenvalues or eigenvectors of matrices. Theoretically speaking, these areas have been well studied and are well understood, i.e. conditions for the existence/uniqueness of a solution to a linear system and criteria for diagonalizing matrices are well known. However, from the programming point of view, two problems arise when turning these theoretical solutions into practical applications on computers. One is that the implementation of such algorithms must be efficient, namely it does not take too long to get a result and the associated data do not occupy excessive memory space. Another is stability, which is the ability not to magnify approximation errors. A number of algorithms are restricted in their use due to their numerical instability, e.g. Strassen’s algorithm is less stable than the conventional technique
for matrix multiplication [65]. Note that these errors cannot be avoided due to the rounding-off approximations universally used for floating-point calculations on computers.

This chapter begins, in Section 2.1 with a review of the mathematical concepts of a matrix and its operations. How to perform matrix operations efficiently is explained by exploiting certain properties of the matrices. Section 2.2 presents a historical overview of numerical linear algebra libraries, summarising the existing solutions for providing libraries that perform matrix operations. Section 2.3 summarises the chapter.

2.1 Preliminary

This section gives the background for numerical linear algebra. Matrix and matrix operations are described in Sections 2.1.1 and 2.1.2, respectively. According to the non-zero elements structure and mathematical relations, a classification of matrix properties is introduced in Section 2.1.3. Some available storage formats for each matrix property are presented in Section 2.1.4.

2.1.1 Matrix

A matrix is a rectangular array of elements arranged in a two-dimensional container. The elements inside can either be numbers or symbolic expressions. An $m \times n$ matrix $A$ is denoted by:

\[
A = \begin{bmatrix}
    a_{11} & a_{12} & \cdots & a_{1n} \\
    a_{21} & a_{22} & \cdots & a_{2n} \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{m1} & a_{m2} & \cdots & a_{mn}
\end{bmatrix}
\]

Matrix operations are the primary concerns in *Numerical Linear Algebra* (NLA). Given a system of equations,

\[
\begin{align*}
    a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
    a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m
\end{align*}
\]  

(2.1)
these can be written in matrix form as:

\[
\begin{bmatrix}
  a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\
  a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\
  \vdots \\
  a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n
\end{bmatrix}
= 
\begin{bmatrix}
  b_1 \\
  b_2 \\
  \vdots \\
  b_m
\end{bmatrix}
\] (2.2)

Equation 2.2 can be transformed as a linear combination into:

\[
\begin{bmatrix}
  a_{11} \\
  a_{21} \\
  \vdots \\
  a_{m1}
\end{bmatrix}x_1 + 
\begin{bmatrix}
  a_{12} \\
  a_{22} \\
  \vdots \\
  a_{m2}
\end{bmatrix}x_2 + \cdots + 
\begin{bmatrix}
  a_{1n} \\
  a_{2n} \\
  \vdots \\
  a_{mn}
\end{bmatrix}x_n = 
\begin{bmatrix}
  b_1 \\
  b_2 \\
  \vdots \\
  b_m
\end{bmatrix}
\] (2.3)

Furthermore, using matrix-vector multiplication gives:

\[
\begin{bmatrix}
  a_{11} & a_{12} & \cdots & a_{1n} \\
  a_{21} & a_{22} & \cdots & a_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{m1} & a_{m2} & \cdots & a_{mn}
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  x_2 \\
  \vdots \\
  x_n
\end{bmatrix} = 
\begin{bmatrix}
  b_1 \\
  b_2 \\
  \vdots \\
  b_m
\end{bmatrix}
\] (2.4)

The solution of a system of linear equations can be described as “solve \( x \) from \( A \times x = b \)”. Therefore, matrix algebra is essential for the description of linear equations. An \( m \times n \) matrix contains \( m \) rows and \( n \) columns. An \( m \times 1 \) matrix is called a column vector and a \( 1 \times n \) matrix is called a row vector. If \( m = n \), the matrix is said to be square.

2.1.2 Matrix Operations

This section lists some typical matrix operations that are often executed using NLA libraries.

Basic Matrix Operations

Some basic matrix notations and definitions are listed in Table 2.1. Classified by the number of matrices involved, basic matrix operations can be divided into three groups:

- Binary operations: the operations generate one output matrix from two matrices or one matrix and one scalar, such as matrix multiplication or
Name | Notation | Definition | Operation Type |
--- | --- | --- | ---
Matrix Addition | $C = A + B$ | $c_{ij} = a_{ij} + b_{ij}$ | Binary
Matrix Multiplication | $C = AB$ | $c_{ij} = \sum_{k=1}^{q} a_{ik} \cdot b_{kj}$ where $k$ is the number of columns in $A$ and number of rows in $B$ | Binary
Matrix Scale | $C = \alpha A$ | $c_{ij} = \alpha a_{ij}$ | Binary
Matrix Transpose | $C = A^T$ | $c_{ij} = a_{ji}$ | Unary
Matrix Trace | $x = tr[A]$ | $x = \sum_{i=1}^{n} a_{ii}$ if $A$ is a square matrix | Reduction
Maximum Absolute Column Sum Norm | $x = ||A||_1$ | $x = \max_j \sum_{i=1}^{n} |a_{ij}|$ | Reduction
Maximum Absolute Row Sum Norm | $x = ||A||_\infty$ | $x = \max_i \sum_{j=1}^{n} |a_{ij}|$ | Reduction
Frobenius Norm | $x = ||A||_F$ | $x = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}$ | Reduction

Table 2.1: Notations and definitions of some basic matrix operations.

- Unary operations: the operations need only one matrix to generate another matrix, such as matrix transpose;
- Reduction operations: the operations take a matrix as input and generate a scalar, such as matrix norm.

Linear Systems of Equations

A linear system of equations is defined as solving $Ax = b$, where $A$ is a given non-singular square matrix $^1$ and $b$ is a given column vector. $x$ has the same dimension and element types as $b$.

Eigenvalue and Eigenvector Problems

Given an $n \times n$ matrix, find a non-zero vector $x$ and a scalar $\lambda$ that satisfy $Ax = \lambda x$.

\(^1\)A square matrix is non-singular if there exists a square matrix reverse $A^{-1}$. The system has a unique solution, i.e., $x = A^{-1}b$.
CHAPTER 2. NUMERICAL LINEAR ALGEBRA

Singular Value Problems

Given an \( m \times n \) matrix \( A \), find a \( n \times 1 \) non-zero vector \( x \) and a scalar \( \lambda \) so that \( A^T Ax = \lambda x \).

Least Squares Problems

Compute \( x \) that minimizes \( \|Ax - b\|_2 \) from \( Ax = b \), where \( A \) is an \( m \times n \) matrix (non-square), \( x \) is a \( n \times 1 \) vector and \( b \) is a \( m \times 1 \) vector. \( \|y\|_2 \) is defined as \( \sqrt{\sum_{k=1}^{n} |y_k|^2} \). In the sense of linear equations, \( m \) and \( n \) are used to denote the number of equations and the number of unknowns. If \( m > n \), \( Ax = b \) has more unknowns than equations and thus cannot be solved exactly.

2.1.3 Matrix Properties

Due to the non-zero elements structure and various mathematical relations, a number of matrix properties have been defined by the NLA community, as shown in Table 2.2.

<table>
<thead>
<tr>
<th>Property</th>
<th>Non-zero Elements Criterion and Mathematical Relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense Matrix</td>
<td>The majority of elements are non-zero.</td>
</tr>
<tr>
<td>Banded Matrix</td>
<td>The non-zero elements are located around the major diagonal.</td>
</tr>
<tr>
<td>Upper Triangular Matrix</td>
<td>The elements below the diagonal entries are zero.</td>
</tr>
<tr>
<td>Lower Triangular Matrix</td>
<td>The elements above the diagonal entries are zero.</td>
</tr>
<tr>
<td>Diagonal Matrix</td>
<td>A diagonal matrix is a square matrix in which all non-zero elements are on the major diagonal.</td>
</tr>
<tr>
<td>Symmetric Matrix</td>
<td>For all elements in a square matrix ( A ), ( a_{ij} = a_{ji} ).</td>
</tr>
<tr>
<td>Block Matrix</td>
<td>A block matrix can be defined by smaller matrices called blocks.</td>
</tr>
<tr>
<td>Sparse Matrix</td>
<td>The majority of elements are zero.</td>
</tr>
</tbody>
</table>

Table 2.2: Matrix properties and definitions.

Note that a matrix may have more than one property, e.g. a matrix can be both symmetric and banded. A non-zero elements structure criterion specifies a
matrix property by the structure and number of non-zero entries. A mathematical relation criteria means that the elements of a matrix have special relationships, e.g. in a symmetric matrix $A$, for all $i, j, a_{ij} = a_{ji}$. Both criterion enable time and memory space savings when storing and computing with elements on a computer; details are given in Section 2.1.4.

2.1.4 Storage Format

NLA not only involves linear algebra, but also computer science. The execution time of a matrix operation can be reduced if the matrix has zero elements. This is shown in Figure 2.1 and Figure 2.2.

```plaintext
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            c(i,j) = c(i,j) + a(i,k) * b(k,j)
        end for
    end for
end for
```

Figure 2.1: The algorithm for matrix-matrix multiplication $C = A \times B$ where both $A$ and $B$ are square dense matrices.

```plaintext
for i = 1 to n
    for j = i to n
        for k = i to j
            c(i,j) = c(i,j) + a(i,k) * b(k,j)
        end for
    end for
end for
```

Figure 2.2: The algorithm for matrix-matrix multiplication $C = A \times B$ where both $A$ and $B$ are upper triangular.

Moreover, the required memory space can be reduced by storing only non-zero elements. The NLA community has introduced a number of storage formats, each of which targets a matrix with certain properties. A few commonly used storage formats are described below:

- Dense Format: A matrix $A$ is stored in a two-dimensional array. Element
$a_{ij}$ is accessed by array DENSE[$i$][$j$]. This format is used for storing a matrix with dense property.

- Banded Format: The banded format is designed to store an $n \times n$ matrix $A$ with banded property. Two values $b_u$ and $b_l$ are used to indicate the upper and lower bandwidths. The array has $b_u + b_l + 1$ number of rows and $n$ number of columns. Element $a_{ij}$ is accessed by BANDED[$b_u + i - j + 1$][$j$].

- Packed Format: It is efficient to store upper triangular matrices, lower triangular matrices and symmetric matrices in packed format. Given an $n \times n$ matrix $A$, the array size of PACKED is $\frac{1}{2}(n^2 + n)$ and element $a_{ij}$ is stored in PACKED[$i + \frac{1}{2}(2n - j)(j - 1)$] or PACKED[$i + \frac{1}{2}(j - 1)j$] for lower triangular or upper triangular matrices, respectively. A matrix with symmetric property can be stored in either lower packed or upper packed format.

- Yale Sparse Matrix Format [43]: This uses three one-dimensional arrays (IA, JA and YS) to store an $m \times n$ matrix $A$ in row-wise format. Row-wise means storing elements in the form of row by row. The first array IA stores the indices to the locations in JA and YS where rows of $A$ starts. The second array JA contains the column index of each non-zero element in $A$. The third array YS stores the non-zero elements of $A$.

Note that although a matrix can be stored in many storage formats, each storage format targets one or more specific properties for efficiency reasons.

2.1.5 Summary

This section has given a general introduction to NLA. The relationships between matrix, matrix operation, matrix property and storage format are summarised as follows:

- A matrix can have one or more properties, i.e. a few properties can overlap and others cannot;

- Some properties do not have effects on the storage format, such as the positive definite property $^2$; $^2$A square real matrix $A$ is positive definite if $b^TAb > 0$ for all non-zero vectors $b$, where $b^T$ denotes the transpose of $b$. 


• No matter what the matrix properties are, a matrix should be stored in only one format at any one time;

• Certain matrix operations only apply to certain matrix properties, such as a matrix has to be square to perform solving a linear system of equations;

• Through a matrix operation, the output matrix property may be different from the input one(s) and thus may result in change of the storage format.

2.2 An Historical Overview on the Design of Existing NLA Libraries

Nowadays most basic computational problems can be solved by calling appropriate subroutines from high-quality numerical libraries. One should always use library subroutines whenever possible, rather than developing and optimising one’s own version [4].

Although high performance libraries have been developed and optimized by the vendors, some important software engineering practices, such as abstraction, have been sacrificed. This section reviews the currently available NLA libraries, which are roughly divided into two groups: traditional libraries (Section 2.2.1) and Object-Oriented Numerical Linear Algebra (OONLA) libraries (Section 2.2.2). A few typical libraries are chosen for in-depth review, focusing on software construction, in particular functionality reuse in NLA libraries.

2.2.1 Traditional Libraries

Traditional libraries are developed using a top-down methodology and implemented in an imperative language [87] such as Fortran 77. The design of these libraries primarily focuses on functionality, performance and stability. An important feature of a traditional library is the heavy use of subroutines. Examples of these libraries are NAG [86], IMSL [9], LINPACK [41], EISPACK [52], BLAS [1] and LAPACK [12]. BLAS and LAPACK are chosen to be described due to their wide usage.
BLAS

Basic Linear Algebra Subprograms (BLAS) is a specification of a set of kernel routines for linear algebra [1]. Fortran 77 is used as the language to provide a reference implementation. There are a number of matrix properties defined in BLAS: general banded, general dense, Hermitian banded, Hermitian, symmetric, symmetric banded, triangular and unstructured sparse matrices. Each matrix is stored optimized in a format that is to deliver the best performance for a particular platform. Examples of optimized implementations provided by vendors are ESSL 3 from IBM and MKL 4 from Intel. Alternatively, ATLAS 5 was developed to automatically generate optimised subroutines for different machine architectures. The functionality of BLAS is divided into three groups by the inherent computational complexities:

- Level 1: routines for scalar and vector operations, such as dot product and vector norm. These operations require $O(N)$ floating point operations on $O(N)$ data.

- Level 2: routines for matrix-vector operations, such as matrix-vector multiplication. These operations require $O(N^2)$ floating point operations on $O(N^2)$ data.

- Level 3: routines for matrix-matrix operations, such as matrix-matrix multiplication. These operations require $O(N^3)$ floating point operations on $O(N^2)$ data.

The process of using BLAS is described using the following example of matrix-matrix multiplication: for a given problem $C = A \times B$, where $A$ is an upper triangular matrix and $B$ is a dense matrix. Firstly the appropriate subroutine is found by following the naming scheme. The first letter of the subroutine name indicates the data type (REAL, DOUBLE REAL PRECISION, COMPLEX and DOUBLE COMPLEX PRECISION). The next two letters indicate the matrix properties and storage formats and the remaining letters specify the matrix operation. Supposing the precision type used is REAL, and given that the matrix $A$ is triangular, by following the name scheme, subroutine STRMM is selected. The

---

3http://www-03.ibm.com/systems/software/essl/
5http://math-atlas.sourceforge.net/
next task is to specify the parameters. The full specification of the subroutine is STRMM(SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B, LDB), which is used to perform one of the matrix operations:

\[ B \leftarrow \alpha \text{op}(A)B \]

or

\[ B \leftarrow \alpha \text{Bop}(A) \]

where \( \alpha \) is a scalar, \( B \) is an \( m \) by \( n \) matrix, \( A \) is a unit, or non-unit, upper or lower triangular matrix and \( \text{op}(A) \) is one of \( \text{op}(A) = A \) or \( \text{op}(A) = A^T \). The extension \( (\alpha) \) to the original matrix-matrix multiplication is because this scale operation is usually performed together with the matrix multiplication and can be implemented within the three nested for-loops. The specifications of matrix-matrix multiplication is given in Table 2.3. \( X^T \) and \( X^H \) represent the transpose and conjugate transpose of matrix \( X \), respectively. Four precision types, single, double, complex and double complex are supported. From the table it is easy to note that not all the combinations have been specified in BLAS. For example, if the user knows both \( A \) and \( B \) are triangular, the most appropriate subroutine, \( \text{xTRMM} \) can only make use of the property for one of them, resulting in inefficient space and calculation cost.

It is worth noting that developing a highly optimised subroutine is greatly time-consuming, as it requires careful design and may need low-level programming knowledge. In order to reduce the software engineering efforts, [72] propose a GEMM-based approach, which builds various level 3 BLAS subroutines upon the GEMM and some level 1 and level 2 subroutines. With highly optimised GEMM, other level 3 BLAS subroutines can “automatically” perform well.

**LAPACK**

LAPACK (Linear Algebra PACKage) is written in Fortran to provide routines for solving systems of linear equations, least squares problems, eigenvalue problems and singular value problems. Matrix factorisations such as LU, QR are also implemented as well as related computations such as estimating condition numbers [12]. The implementations of LAPACK make heavy use of calling the subroutines of BLAS.

Although there are only five high-level operations (linear equations, etc.) in
<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Functionality</th>
<th>Precision Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xGEMM</td>
<td>$C \leftarrow \alpha \text{op}(A)\text{op}(B) + \beta C, \text{op}(X) = X, X^T, X^H, C - m \times n$</td>
<td>S, D, C, Z</td>
</tr>
<tr>
<td>xSYMM</td>
<td>$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, A = A^T, C - m \times n$, where $A$ is a symmetric matrix</td>
<td>S, D, C, Z</td>
</tr>
<tr>
<td>xHEMM</td>
<td>$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, A = A^H, C - m \times n$, where $A$ is a hermitian matrix</td>
<td>C, Z</td>
</tr>
<tr>
<td>xSYRK</td>
<td>$C \leftarrow \alpha A A^T + \beta C, C \leftarrow \alpha A^T A + \beta C, C - n \times n$, where $C$ is a symmetric matrix</td>
<td>S, D, C, Z</td>
</tr>
<tr>
<td>xHERK</td>
<td>$C \leftarrow \alpha A A^H + \beta C, C \leftarrow \alpha A^H A + \beta C, C - n \times n$, where $C$ is a hermitian matrix</td>
<td>C, Z</td>
</tr>
<tr>
<td>xSYR2K</td>
<td>$C \leftarrow \alpha A B^T + \alpha B A^T + \beta C, C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$, where $C$ is a symmetric matrix</td>
<td>S, D, C, Z</td>
</tr>
<tr>
<td>xHER2K</td>
<td>$C \leftarrow \alpha A B^H + \alpha B A^H + \beta C, C \leftarrow \alpha A^H B + \alpha B^H A + \beta C$, where $C$ is a hermitian matrix</td>
<td>C, Z</td>
</tr>
<tr>
<td>xTRMM</td>
<td>$B \leftarrow \text{op}(A)B, B \leftarrow \alpha \text{op}(A), \text{op}(A) = A, A^T, A^H, B - m \times n$, where $A$ is a triangular matrix</td>
<td>S, D, C, Z</td>
</tr>
</tbody>
</table>

Table 2.3: The specification of matrix matrix multiplication [40].

LAPACK, due to the number of supported matrix properties, various algorithms are provided for each of these operations, e.g., xGESV solves a general system of linear equations, whereas xGBSV solves a banded system of linear equations. Moreover, more than one implementation is provided for a given algorithm, according to the storage format, e.g., xPOSV and xPPSV. Table 2.4 shows some of the implementations provided in LAPACK.

**Summary**

In conclusion, the development of a NLA program using traditional libraries involves the following steps:

- find the appropriate matrix operation(s) of the problem;
- analyse the matrices to determine the property and storage format for computing efficiently;
- find the libraries that support each matrix operation, the matrix property and the storage formats;
The traditional way of designing NLA libraries mainly focuses on performance. Some abstractions are provided by procedure calls for functionality reuse, i.e., some matrix operations can be implemented by making calls to highly optimised low level subroutines as building blocks. For example, SYMM can be implemented by calling COPY and GEMM.

To ensure that the best performance can be attained, NLA libraries are written in a combinatorial way, i.e. various aspects are combined to implement one linear algebra routine. For examples, BLAS supports many matrix properties (dense, band, triangular, symmetric etc.), several storage formats (dense, band, packed, etc) and four different element types (single precision, double precision, complex, double precision complex). Although not all the combinations are valid or efficient (for example, storing a dense matrix in packed format is invalid and storing it in band format is inefficient), nor do libraries support all the mathematically valid matrix operations efficiently (for example, BLAS does not provide a subroutine for matrix multiplication of a upper triangular and a lower triangular matrix), the total number of implementations is still large. This is called the combinatorial explosion on the number of implementations [104], which is due to the tight coupling between each algorithm and the data structures it operates on. For instance, the NIST implementation of Sparse BLAS contains more than 10,000 routines and a code generation system [100].
Moreover, from a different point of view, the interface of a traditional library is complex and thus not user-friendly [87]. The work of selecting subroutines and identifying all the parameters is non-trivial. The user has to know the matrix property and storage format of the operand(s) in order to choose an appropriate subroutine.

2.2.2 Object-Oriented Numerical Linear Algebra Libraries

Later software engineering techniques such as Object-Oriented Programming (OOP) and Design Patterns took NLA library development a significant step further. In the design of an object-oriented NLA library, special features from object-oriented programming, including class inheritance, object composition and parameterised types, enable reuse of functionalities [51] and provide a better separation between algorithms and data structures. The underlying philosophy of a NLA library with high level abstraction is to provide a simpler interface and better modularity without sacrificing delivered performance. Examples of such libraries are LAPACK++ [97], JAMA [6], SL++, Seldon, SparseLib++, BPKIT [31], MTL [105], OOLALA [87], etc. Based on the UML class diagrams of their designs and the reuse of functionality that is used, a few examples here are chosen to be explained in detail below.

JAMA

JAMA stands for Java Matrix package. It is a basic linear algebra package that provides user-level classes for constructing and manipulating dense real matrices [6]. JAMA is meant to provide sufficient functionality for handling matrix operations, packaged in a way that is natural and understandable to non-experts. JAMA is intended to serve as the standard matrix class for Java and was proposed as such to the Java Grande Forum and then to Sun. MathWorks and NIST (National Institute of Standards and Technology) have developed an implementation and have released this in the public domain.

JAMA contains six classes: one for matrix and five for factorisations. Basic matrix operations are implemented as methods of the Matrix class. The complex matrix operations, such as solving a system of linear equations, are computed by accessing the factorisations, i.e. object composition; the factorisation object

\footnote{Strictly speaking, parameterised types is not an object-oriented functionality.}
delegates operations to the matrix object. JAMA stores a matrix directly in a two
dimensional array and the classifications of different matrix properties and storage
formats are not supported; i.e., all the matrices are stored as dense matrices.
Table 2.5 expresses the capabilities of JAMA.

<table>
<thead>
<tr>
<th>Elementary Operations</th>
<th>Addition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Subtraction</td>
</tr>
<tr>
<td></td>
<td>Multiplication</td>
</tr>
<tr>
<td></td>
<td>Scalar Multiplication</td>
</tr>
<tr>
<td></td>
<td>Transpose</td>
</tr>
<tr>
<td></td>
<td>Element-Wise Multiplication</td>
</tr>
<tr>
<td></td>
<td>Element-Wise Division</td>
</tr>
<tr>
<td></td>
<td>Unary Minus</td>
</tr>
<tr>
<td></td>
<td>Norm</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Complex Operations</th>
<th>Nonsingular Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Least Squares</td>
</tr>
<tr>
<td></td>
<td>Determinant</td>
</tr>
<tr>
<td></td>
<td>Inverse</td>
</tr>
<tr>
<td></td>
<td>Condition Number</td>
</tr>
<tr>
<td></td>
<td>Rank</td>
</tr>
<tr>
<td></td>
<td>Pseudoinverse</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Factorization</th>
<th>Cholesky</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LU</td>
</tr>
<tr>
<td></td>
<td>QR</td>
</tr>
<tr>
<td></td>
<td>SVD</td>
</tr>
<tr>
<td></td>
<td>Symmetric Eigenvalue</td>
</tr>
<tr>
<td></td>
<td>Nonsymmetric Eigenvalue</td>
</tr>
</tbody>
</table>

Table 2.5: Supported operations of JAMA.

**LAPACK++ and SparseLib++**

LAPACK++ is a C++ extension to the Fortran implemented LAPACK. Various
types of matrix are supported: non-symmetric, banded, triangular, symmetric,
tridiagonal, etc. In order to model the commonality of different matrices, LA-
PACK++ maintains a class hierarchy (Figure 2.3). LAPACK++ is based on class
inheritance. The topmost level is class Matrix. In the second level, class matrix
is specialized by matrix properties. The third level combines some properties
to form a matrix with more than one property. A storage format is chosen for
each property, e.g., band matrix is stored in band format and triangular matrix
is stored in packed format.

SparseLib++ is also written in C++. As its name implies, SparseLib++ is de-
volved for sparse matrix operations, the codes of which are usually complicated,
because the underlying data stored formats are entangled within the application.
SparseLib++ attempts to address this difficulty by providing various implementa-
tions for each matrix operation. Based on class inheritance, the Matrix class of
SparseLib++ is specialised by various storage formats: sparse vector, coordinate
storage matrix, compressed row storage matrix, compressed column storage ma-
trix, compressed diagonal storage matrix, jagged diagonal storage matrix, blocked
compressed storage matrix and skyline storage matrix.

Other libraries that follow the same class inheritance design are ISIS++, BP-
KIT, IML++, etc. The implementation of such a library has a drawback in
that a full implementation is not provided. For example, SparseLib++ does
not support the symmetric and Hermitian matrix properties. And providing a
full implementation with such a design will lead to an explosion in the number
of implementations of classes, which are combinations of the supported matrix
properties and the number of storage formats available for each matrix property.
Moreover, this design is difficult to maintain; the base and derived classes are
strongly coupled and so changes to a class in a higher level of the hierarchy may
lead to changes in all its derived classes.

MTL

MTL (Matrix Template Library) [104, 105] aims to provide comprehensive func-
tionality with a small number of implementations, while at the same time achiev-
ing high performance. Four aspects of designing a NLA library are identified:
CHAPTER 2. NUMERICAL LINEAR ALGEBRA

precision element types, matrix orientation, matrix shape and storage types. To handle these aspects, MTL defines a set of component layers, each of which is a collection of classes that are templated on the lower layers. The bottommost layer contains the numerical types (real, float, etc) of an element. The next layer consist of 1D containers, which are followed by 2D containers. The matrix orientation (either row or column-wise) is wrapped to 2D containers as an adapter. The matrix shape layer defines a number of matrix properties: general, symmetric, triangular, etc, which in turn wrap matrix orientation.

The bridge pattern of the Gang of Four design patterns [51] is used in the design of MTL to reduce the number of classes. Compared with libraries such as LAPACK++, which use class inheritance, object composition and parameterised types (in C++ called templates) are used to model the library structure. The relationship between matrix property and storage formats has changed, i.e., a matrix has a storage format rather than each storage format is a kind of matrix. The benefit of this design is that the storage format can vary at run-time. Also, a unified interface is provided for the storage format, and thus the subclasses of Matrix do not need to know how an element is stored in order access an element. Another library that follows this design is PMLP [23]. Although the designs of MTL and PMLP reduce the number of combinations that need to be implemented, they do not prevent the user from creating inefficient combinations and even impossible combinations [87]. For example, given \( B \leftarrow A + B \), where \( A \) is a dense matrix and \( B \) is a upper triangular matrix, this operation is valid if both have the same order. However, \( A + B \) gives a dense matrix, but it is impossible to change an object of dense class to an object of upper triangular class because DenseMatrix is not a subclass of UpperTriangularMatrix.

OOLALA

OOLALA was designed by Lujan [87] as a basis for an Object Oriented Numerical Linear Algebra (OONLA) library in which the property and storage format of a given matrix is dynamically varied by propagating matrix properties. OOLALA provides two high level abstractions, namely, Matrix level and Iterator level abstraction, to avoid the combinatorial explosion [104] phenomenon in traditional NLA libraries.

The class diagram of OOLALA is given in Figure 2.4. Elements of a matrix are stored in arrays. Various storage formats (dense format, banded format,
packet format, etc.) build upon the array(s) with different accessing and mutating mechanisms. In class Property, a set of if-else rules are applied to choose the most appropriate storage format for a given matrix property. A matrix may change its property after performing a matrix operation. The functionalities of get and set of class Matrix are delegated by the associated methods of class Property.

![Class diagram of OOLALA in [87].](image)

OOLALA has two important modifications. One is the enabling of some matrix properties to be represented as attributes of a class rather than classes. For example, the positive definite property may be represented as an attribute of class Property. Although a matrix with or without this property affects the selection of an implementation for certain matrix operations, the storage format is not influenced. The second modification is that class Matrix is a client of class Property, which provides a unified interface for all the matrix properties. This allows a matrix property to be changed when the user creates inadvisable combinations.

### 2.2.3 Summary

Traditional NLA libraries provide a large set of subroutines, each of them dealing with one matrix operation involving various aspects, such as matrix property, storage format and element data types. JAMA does not support sparse matrix operations. OONLA libraries such as LAPACK++ and SparseLib++ reassemble the various operations into difference Matrix classes, which are maintained by a class hierarchy. Others such as MTL and PMLP decouple storage format from
matrix, allowing the change of its storage format and eliminating the combinations of storage formats and matrix properties. Moreover, the design of OOLALA moves further than MTL and PMLP: by using a bridge pattern between matrix and matrix property, the property of a matrix is decoupled from class matrix and thus a change of property is permitted.

2.3 Summary of Chapter

This chapter has introduced the natures of matrix and matrix operations, describing why this is a rich research area and the current state of NLA library design. Although current designs significantly reduce the complexity of using and designing a NLA library, further improvement can be made by introducing new methodologies, as introduced in Chapter 3.
Chapter 3

Separation of Concerns

“...one is willing to study in depth an aspect of one’s subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day... But nothing is gained — on the contrary! — by tackling these various aspects simultaneously. It is what I sometimes have called ‘the separation of concerns’... This is what I mean by ‘focussing one’s attention upon some aspect’: it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect’s point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.”[39]

The term “separation of concerns” was probably invented by Edsger W. Dijkstra in his paper “On the role of scientific thought” [38] in 1974. A concern is anything of interest to a stakeholder, which can be the project sponsor, software developer, end user, etc. Nowadays the idea of separating different concerns is becoming more acceptable and is widely used across all the programming paradigms. Examples of this include procedures in procedural programming, objects in object-oriented development, components in component-based programming, etc. Although problems from different domains are viewed and tackled differently due to their natures, they are all solved in a way where, firstly, addressing separated concerns corresponds to distinct parts of the solution, and then composing them together [91].
This chapter introduces the concept of separation of concerns (Section 3.1) and two methodologies tackling the software design problem, namely *Aspect-Oriented programming* (AOP) and *Compiler Optimisation Techniques*, in Section 3.2 and Section 3.3, respectively.

### 3.1 Building a Complex System

Nowadays, software systems are essential to both businesses and scientific laboratories. Most end users need the help of a software system to perform certain operations. By analysing the properties of a complex system, [91] proposes that the requirements for software development can be categorised as:

- **Functionality requirement.** This specifies that, when provided an input, the function should return a desired output.

- **Run-time requirement.** This requirement defines the run-time behaviour, such as performance.

- **Software artefact requirement.** This is particularly attractive to software system developers. It includes modularity, reusability, maintainability, etc.

This section describes the current method of building a complex software system. Section 3.1.1 introduces component-based development. Section 3.1.2 and 3.1.3 give the concept of a crosscutting concern and the effects of not handling crosscutting concerns properly at the source code level. A classification of crosscutting concerns is then given, in Section 3.1.4.

#### 3.1.1 Component-Based Development

Building a complex system can be extremely difficult as individual human beings cannot consider every facet of it at one time. Therefore, it is natural to break complex problems into smaller, and thus more solvable, components. Each component plays its specific role and has a well-defined interface by which it can be integrated. A component hides its internal contents from other components. From the user’s perspective, it is not necessary to know how a component is implemented in order to make it work properly. The user only needs to know how to get a desired response when the component is signalled through its interface. That is, as long as the component retains its published interface, it is possible
to modify its internal contents and even replace it completely with other code. Because this characteristic enables parallel development and software reuse, it is a useful and important technique with which to build complex systems.

The straightforward and usual approach to build a complex system [68] is:

1. Understand what the software system is supposed to do, such as what are the functional and non-functional requirements of the system;

2. According to the requirements, identify system components which individually handle a set of related requirements, i.e. mapping the world of requirements into a world of components.

3. Once each component is implemented, they can be integrated to form the required system.

### 3.1.2 Crosscutting Concerns

The purpose of a system is to meet its requirements, or more generally, its concerns. As described above, a complex system is simplified by identifying these concerns and then modularising them. Modularisation refers to the localization of artefacts, including specification, realization, and implementation of a concern or a set of concerns [68]. Due to the complexity of the software system, the developers want to explore each concern at a time and to work in isolation. However, conventional software written in a standard programming language is in the form of linear text, i.e., the program is decomposed only one way into modules, e.g. using classes as the basic units in Object-Oriented Programming (OOP). While the primary concerns are well handled in the dominant decomposition, other concerns that involve several different collaborating components cannot be encapsulated within this module. Therefore, building a complex system usually exhibits two limitations:

1. A few concerns of a system cannot be fitted into any single component [68]. These concerns are called *crosscutting concerns*.

2. Failing to predict the potential future changes in crosscutting concerns may eventually cause changes in many non-local parts of a system [81].

Suppose a software architect is asked to design a numerical weather prediction application. The primary concerns would be the scientific logic, including complicated equations which govern how the state of a fluid changes. Other concerns
would involve features such as logging, pooling and caching. These are common
to many of the primary concerns and thus cannot be separated easily or cleanly.
Moreover, later on, if a new caching method is suggested to replace the method
in a previous design, a number of side-effects may need to be taken into account,
and reimplemention of the whole system may be necessary. Basically, due to
the impossibility of separating crosscutting concerns into modules, the system to
be built can be difficult to understand and maintain.

By analysing these inabilities in conventional software development, [68] states
that:

“A successful solution to this problem involves two things: an
engineering technique to separate such concerns from requirements all
the way to code and a composition mechanism to merge the design
and implementation for each concern to result in the desired system.”

3.1.3 Code Scattering and Tangling

The above symptoms of nonmodularisation result in two effects at source-code
level, namely, code scattering and code tangling. Code scattering involves writing
code over different components, in order to fulfil one concern; whereas with code
tangling, a single component plays additional roles as it is used to handle more
than one concern of the system at a time. Figure 3.1 gives an example of code
scattering, where a logging implementation is spread over different places in the
code.

Both code scattering and code tangling lead to a breakdown in current soft-
ware development. In particular, their occurrence violates the usual need for high
cohesion and low coupling of software modules. Software cohesion and coupling
are important properties described by Larry Constantine [109] in the field of soft-
ware metrics. High cohesion defines the situation where a component is dedicated
to its own responsibility. Low coupling refers to a relationship in which a change
in one module will not result in any changes in the other. These two properties
often occur together. Modules with high cohesion and low coupling are more
likely to come with other desirable software quality factors, such as robustness,
extendibility, reliability, etc.
import java.util.logging.*;
public class Item {
    private String _id;
    private float _price;
    static Logger _logger = Logger.getLogger("trace");

    public Item(String id, float price) {
        _id = id;
        _price = price;
    }

    public String getID() {
        _logger.logp(Level.INFO, "Item", "getID", "Entering"); \ logging
        return _id;
    }

    public float getPrice() {
        _logger.logp(Level.INFO, "Item", "getPrice", "Entering"); \ logging
        return _price;
    }

    public String toString() {
        _logger.logp(Level.INFO, "Item", "toString", "Entering"); \ logging
        return "Item: " + _id;
    }
}

Figure 3.1: An example of code scattering from [81].

3.1.4 Classification of Crosscutting Concerns

Prior research has already identified a number of commonly occurring examples of crosscutting concerns, such as caching [81], logging and tracing [58]. In order to give these a consistent solution, Marin [89, 88] proposes a classification system for crosscutting concerns. Crosscutting concern sorts are used to describe the recurrent and atomic concerns. Thirteen crosscutting concern sorts [88] have been identified; the most frequently occurring of these are stated to be the following:1

- **Consistent behaviour**: consistent behaviour captured by a natural pointcut2 is applied to a number of places in a system;

- **Contract enforcement**: design by contract, such as pre- and post-condition checks;

---

1See [88] for a complete classification of crosscutting concerns.
2A pointcut is used to specify a set of certain points in the base application, where new behaviours should be merged to. See Section 3.2.1 for details.


- **Entangled roles**: a function/method with a secondary role which is tangled with the component’s primary concern;

- **Expose context**: directly expose a caller’s context to a callee, even if there are a number of components between them;

- **Role superimposition**: code implementing a secondary role is tangled with the code implementing the primary role in a component (e.g. class);

- **Policy enhancement**: a required policy affects a number of elements in the system.

### 3.2 Aspect-Oriented Programming

[77] is probably the first paper to use the term *Aspect-Oriented Programming* (AOP). Since its publication in 1997, AOP has attracted a large number of audiences from various international conferences, such as OOPSLA, ECOOP and AOSD. It has been found that AOP enables developers to write modular code for the majority of design patterns [62], logging [81] and exception detection and handling [85].

#### 3.2.1 Introduction to AOP

AOP is a new methodology for modularising crosscutting concerns. By introducing a new unit—the aspect, concerns cutting across others are encapsulated inside individual aspects and thus are not entangled with the core concern. [49] argues that the distinguishing characteristics of AOP systems are *quantification* and *obliviousness*. Quantification implies that the programmer can write unitary and separate statements that have effects at many non-local places of a system. This property is expressed as:

“In programs P, whenever condition C arises, perform action A.” [49]

This defines three semantics that an AOP language must be capable of expressing:

- **P**: The desired execution points of a program;

- **A**: The action that takes place at those execution points; and
CHAPTER 3. SEPARATION OF CONCERNS

- C: The conditions that select a set of desired execution points.

With obliviousness, the primary concern developer is not aware of the existence of aspect code; i.e. the developer does not need to prepare the code to receive the aspects. This enables concerns to be separated not only in the design of a system, but also in the developer’s head [48]. Obliviousness implies that P and A are fairly independent.

In AOP, the primary concerns are implemented in a component, such as a class; whereas the crosscutting concerns are handled by a different abstraction, namely aspect. AOP does not try to provide solutions to any new or unsolved problems, instead, it is designed for solving problems in a better way, e.g., less effort needed to develop a system or improved maintainability. The engineering techniques for handling separated concerns are presented in Section 3.2.2. These concerns, that are individually implemented in classes and aspects, are eventually merged together, in order to form an executable unit. The process of combining primary and crosscutting concerns is called weaving, which is described in Section 3.2.3.

3.2.2 Mechanisms for Supporting Modular Crosscutting Concerns

This section presents the characteristics that make a technique aspect-oriented and gives a basic introduction to some important AOP terminologies. [90] classifies the mechanisms of AOP into four categories:

- Pointcut-Advice, as in AspectJ [75, 76, 81];
- Introduction/Inter-type declaration, as in AspectJ [76, 81];
- Class composition, as in Hyper/J [93]; and
- Traversal specification, as in Demeter, DemeterJ and DJ [84].

As the work in this thesis is not relevant to class composition and traversal specification, only the first two mechanisms are described in detail in this section. AspectJ [58, 81] is chosen as the language to demonstrate examples of the two characteristics. AspectJ is an aspect-oriented extension to the Java programming language.
Pointcut-Advice Model

The pointcut-advice model allows merging of new behaviour into the execution of a program. The user is allowed to define a set of specific execution points in a program, and to execute a new behaviour that may modify or replace the original execution. Any identifiable point in the execution flow of a program is called a *join point*. In the pointcut-advice model, the program construct used to select join points and collect the context of a program is called a *pointcut*. It is a boolean predicate and can be composed with other pointcuts. For example, a pointcut called `meetConditions` can be defined as follows:

```java
pointcut meetConditions(Condition con) :
    execution(void MyClass.op(..)) &&
    args(con);
```

This pointcut targets all the execution points that match the execution of method `op` of class `MyClass`. There are two types of pointcut in AspectJ, namely, primitive and non-primitive pointcuts. A pointcut that captures a program construct is a primitive pointcut, whereas a pointcut that is used for exposing context information is called a non-primitive pointcut. For example, in the example above, `execution` selects a set of execution join points and is primitive, while `args` gives a list of argument values for the join point and is non-primitive.

The new behaviour that takes place at join points is called *advice*. The pointcut defines the rules for capturing a number of join points, for which the advice describes what new actions need to be performed. The advice given below defines that the advice calls method `check(con)` before reaching any join points captured by pointcut `meetConditions`.

```java
before(Condition con) : meetConditions(con) {
    check(con);
}
```

In AspectJ, the user can execute advice *before*, *after* or *around* the selected join points. A few examples of run-time join points of a program are listed below:

---

3The language implementation may restrict whether a join point can be captured or not. For example, loop is not treated as a join point in AspectJ, but LoopsAJ [63] provides a loop join point as an extension.
• making a call to a method;
• method execution;
• raising of an exception;
• field (read or write) access;
• initializing an object.

Inter-type Declaration

AspectJ used to have a feature called introduction, which is now called inter-type declaration. While the pointcut-advice model affects the dynamic execution flow of the program, inter-type declaration modifies the static structure of the program, such as introducing new methods/fields or declaring parents to an existing class. For instance,

```java
private void Client.reserve() {...}
```

states that a new method called `reserve` is defined as a method of class `Client`.

3.2.3 Weaving

As introduced before, the primary concerns are still implemented in the base code using an abstraction such as class; only the crosscutting concerns are developed in aspects. Therefore, it is natural to adopt a widely-used programming language and apply AOP as an extension to it.

Weaving is the key to this. Although concerns are designed and implemented separately in different abstractions for the sake of the developer, they have to be combined together to form an executable unit addressing all the desired concerns. The weaving process therefore requires high compatibility between the aspect code and the base code. For example, AspectJ is designed as an extension to Java to promote the use of AspectJ by current Java programmers with the following compatibility [76]:

• Upward compatibility — a legal Java program is also a legal AspectJ program;
• Platform compatibility — a legal AspectJ program can run on standard Java virtual machines;
CHAPTER 3. SEPARATION OF CONCERNS

- Tool compatibility — Existing tools such as IDEs, documentation tools and design tools can be extended to support AspectJ in a natural way;

- Programmer compatibility — The programmer feels that aspects are natural extensions to Java.

According to the implementation of the base language, the process of weaving may vary. Two major weaving mechanisms, namely static and dynamic weaving are introduced in the following.

Static Weaving

Static weaving modifies the static structure of the base code, such as a class, by injecting aspect code at certain join point shadows [66]. The join point shadow is the place in the source code corresponding to the join point in the program execution. For example, if logging is implemented in an aspect, it is actually identical to writing logging at different required places over many classes. In other words, the aspect code is inlined into the base code. Because of this property, highly optimised implementations can be obtained: the performance of the program written with aspects is comparable to that without aspects. However, from another point of view, it is difficult to identify which of the statements are inserted by which aspects so that the compilation time might be greatly increased as any changes made by an aspect lead to the recompilation of the whole program. An example to show this inflexibility is a tracing aspect [25]. This is used for locating a malfunction in a software system, and it is desirable that it can be woven and executed without restarting the software.

In AspectC++, a source-to-source transformation is performed based on PUMA in the weaving process [108], due to the execution restriction of the binary formats on each machine. The advantage of source code transformation is that the woven codes can be optimised by the C++ compiler. Another important transformation tool is AspectJ, which is widely used in many AOP systems, providing a bytecode transformation [66]. In contrast to source code transformation: on the one hand, the source code is not required for weaving; on the other hand, extra work is needed for optimising the bytecode as the original compiler optimisation is only applicable to the source code.
CHAPTER 3. SEPARATION OF CONCERNS

Dynamic Weaving

Dynamic weaving is a process in which the aspects can be re-defined and re-woven at runtime. In other words, aspects are reified as objects and kept in the executable program [26]. Dynamic weaving does not require join point shadows as no changes will be made to the base code. A dynamic join point based on method interception is provided to modify the execution of a program when the matched methods are called.

The purpose of dynamic weaving is for rapid prototyping and testing by avoiding re-compilation, re-deployment and re-start of the application. An example implementation of a dynamic weaver is PROSE [95].

3.2.4 Relationships with Other Programming Paradigms

“Scientist shall do science not computer hacks.” [106]

Since the first computer was built, numerous studies have been conducted on how to solve a problem in terms of the problem itself rather than learning computing working basis and limitations, i.e., more and more high level abstractions have been invented so that the program is becoming easier to be developed. In the last a few decades, different levels of abstractions, namely, binary code, assembly language, procedural programming and object-oriented programming have been invented. Today OOP is the dominant programming paradigm for software development. Although OOP has gained tremendous success, it still has shortcomings in handling crosscutting concerns properly. The emergence of AOP provides a novel abstraction, namely, concern abstraction [81], which is more related to the problem being solved. The relationships between AOP and other programming paradigms is summarised in the following:

- AOP will not replace other programming paradigms. AOP has been established precisely to modularise crosscutting concerns. It does not attempt to modify the way in which the primary concerns are handled.

- The core idea of AOP is to keep concerns separate throughout all software development. Therefore, it is effective on any programming paradigms that are inadequate in dealing with crosscutting concerns.
3.2.5 Software Metrics

“You can’t control what you can’t measure.” [37]

The biggest advantage of AOP is that it can handle crosscutting concerns in a modular way. AOP was invented to improve one important software property: modularity. However, unlike other software properties, such as performance, that can be clearly measured by the execution time or the number of floating point operations per second, Software Metrics are needed to measure properties such as modularity of one or more pieces of software.

Since the term “software engineering” was first introduced at a NATO conference in 1968 [99], significant improvements have been made in the software development process. A major part of these improvements has been bringing in quantitative methods that can describe and manage the software engineering problems more accurately. Therefore, a number of software metrics have been proposed to measure certain properties in the software development life cycle.

This section briefly traces the history of software metrics and models, giving an overview of them. The software metrics that are commonly accepted in the Aspect-Oriented Community are also presented.

Overview of Software Metrics

A large number of software metrics have been proposed during the last thirty years. However, only a few of these have proved to be useful and have thus survived. This is because most early work with software metrics lacked strict validation. The metrics were either introduced from a theoretical or an empirical standpoint [92]. For empirical validations, inadequate data and sloppy experimental methods are often deployed, thereby compromising their usefulness. These problems have been identified, and improved methodologies have been described, by [34, 17, 16] among others. Moreover, early works in software metrics are more likely to focus on specific individual properties or activities, such as counting Lines of Code (LOC), in estimating software project efforts. Grady [60] reports that most successful applications of software metrics would be organisation-wide. The guidelines to do this are also provided in [60, 59].

A few software metrics that are still in use today are listed below:

- LOC: count the lines of source code. This was the first widely used software
metric. Although using LOC with lack of care may cause significant differences in measured results, it has been shown that, with careful definition and application, LOC is a useful metric for measuring program size [71].

- Function points: measure the size of a piece of software based on the functionality the user perceives.
- Defect counts: count the number of defects per LOC.

Software metrics such as the above have been found to be useful in a number of ways [92], for example:

- guiding testing efforts;
- as design criteria;
- predicting costs and efforts for software development and maintenance;
- measuring software qualities.

**Aspect Oriented Software Metrics**

“Aspect-oriented software development (AOSD) is a promising paradigm to promote improved separation of concerns, leading to the production of software systems that are easier to maintain and reuse.” [101]

Improved separation of concerns is achieved by introducing new abstraction to software engineering. As a consequence, most existing software metrics cannot be applied directly [117, 115]. The new abstraction in AOP, namely the aspect, is a new component that can be composed in different ways with the base components. By reusing and refining the classical software metrics, such as LOC and Chidamber and Kemerer metrics [30], [101] has proposed a metrics suite for measuring modularity in AOSD. These metrics are grouped into four categories: separation of concerns (SOC), coupling, cohesion and size. These are described from Tables 3.1 to 3.4. In all cases, a lower value implies a better result.

Note that the metrics suite presented in this section assumes that aspect-oriented techniques are applied in object-oriented programming, i.e. using AOP as an extension to OOP. Although the majority of current work demonstrates the merit of AOP in the field of OOP, AOP is more broadly helpful to all programming languages that deal inadequately with crosscutting concerns.
CHAPTER 3. SEPARATION OF CONCERNS

<table>
<thead>
<tr>
<th>SOC Metric</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concern Diffusion over Components (CDC)</td>
<td>Counts the number of classes and aspects whose main purpose is to contribute to the implementation of a concern and the number of other classes and aspects that access them.</td>
</tr>
<tr>
<td>Concern Diffusion over Operations (CDO)</td>
<td>Counts the number of methods and advices whose main purpose is to contribute to the implementation of a concern and the number of other methods and advices that access them.</td>
</tr>
<tr>
<td>Concern Diffusion over Lines of Code (CDLOC)</td>
<td>Counts the number of transition points through lines of code. Transition points are the points where concern switch occurs.</td>
</tr>
</tbody>
</table>

Table 3.1: Separation of concern metrics.

<table>
<thead>
<tr>
<th>Coupling Metric</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coupling Between Components (CBC)</td>
<td>Counts the number of other classes and aspects to which a class or aspect is coupled.</td>
</tr>
<tr>
<td>Depth Inheritance Tree (DIT)</td>
<td>Counts how far down in the inheritance hierarchy a class or aspect is declared.</td>
</tr>
</tbody>
</table>

Table 3.2: Coupling metrics.

<table>
<thead>
<tr>
<th>Cohesion Metric</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lack of Cohesion in Operations (LCOO)</td>
<td>Measures the lack of cohesion of a component in terms of the number of method/advice pairs that do not access the same instance variable.</td>
</tr>
</tbody>
</table>

Table 3.3: Cohesion metrics.

The above metrics suite has already been used in many different experimental studies, including [107, 53, 54, 47, 56, 27, 61, 45, 46]. They are used later, in Chapter 6 for evaluating the modularity of two versions of NLA library packages. See [101, 55] for more information about this metrics suite.
CHAPTER 3. SEPARATION OF CONCERNS

53

<table>
<thead>
<tr>
<th>Size Metrics</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code (LOC)</td>
<td>Counts the lines of source code.</td>
</tr>
<tr>
<td>Number of Attributes (NOA)</td>
<td>Counts the number of attributes in each class or aspect.</td>
</tr>
<tr>
<td>Weighted Operations per Component (WOC)</td>
<td>Counts the number of methods or advices of each class or aspect and the number of their parameters.</td>
</tr>
</tbody>
</table>

Table 3.4: Size metrics.

3.3 Compiler Optimisation Techniques

Programming language is the bridge via which people can communicate with machines. However, on the one hand, the programming language should be fairly easy to learn and suitable for describing the real-world problem; on the other hand, the machine only understands machine language, which is composed of a sequence of 0s and 1s. This gap is narrowed by a software system called a compiler, which can translate the source code to its equivalent executable machine code.

The purpose of this section is not to explain how to build a compiler, instead, it introduces the basic optimisations that can be performed by a modern compiler (Section 3.3.2), and describes the performance and software engineering improvements promoted by the use of a special set of compiler optimisation techniques, i.e., loop optimisation (Section 3.3.3 and 3.3.4). The examples in this section are segments of Fortran-like pseudo code.

3.3.1 Introduction

The first electronic computer was invented in the 1940s and it had to be programmed in machine language, i.e., the programmer has to use sequences of 0s and 1s to tell the computer what to do. The operations allowed on this computer were very low level, such as moving data, adding data or comparing two values. It is obvious this way of programming is inefficient and error prone and the code is difficult to modify once written. One step forwards to friendly programming was the development of assembly language in the early 1950s. In assembly languages, a symbolic representation is used for the machine instruction and thus assembly language is much easier to learn and use than machine language. The instructions in assembly languages were initially only mnemonic representations.
of machine instructions. Later, macro instructions were introduced to allow the definition of shorthand for frequently used sequences of machine instructions to make it even more friendly to use. However, as different machines have different instruction sets, assembly language is dependent on the computer architecture and thus is not portable.

A significant step in the evolution of program languages was the emergence of high-level languages, with the development of Fortran for scientific computation, Cobol for business data processing and Lisp for symbolic computations, etc. Compared to assembly language, the most important feature of high-level language was the introduction of high level notations, which are close to natural language and available for machines with different architectures. Needless to say, the compiler plays an important role to promote the use of high-level languages. By acting as a translator between the source code and target machine language program, a compiler supports the following characteristics:

- High performance: not only can the execution overhead for programs written in high-level languages be minimized, but also high-performance computer architecture may be used effectively [7];
- Portability: the same program can be compiled for and executed on different machines; and
- Improved productivity: programs are the most complicated software engineering artefact so that the more advanced and sophisticated the compiler, the more attention the programmer can place on the problem they are trying to solve, rather than worrying about machines’ underlying architectures.

From the point of view of separation of concerns, the compiler tries to take more and more responsibility for the machine’s underlying specifics, leaving the programmer only the problem they want to solve. For instance, the keyword register in the C programming language allows its user to control which variables should reside in registers. However, a less efficient program may be developed if the programmer lacks knowledge on how to judge low-level problems such as register allocation [7]. Therefore, managing variables in registers may hurt the performance, in particular, if the program is running on a different machine rather than the one it was first written for. Many later high-level programming languages, such as Java, do not have this keyword, because modern compilers
normally know better about dealing with these low-level matters than programmers.

### 3.3.2 Basic Optimisations

Optimisation is performed on units of code called *basic blocks*, which are segments of code that contain no branches. The sequence of instructions inside each basic block are always executed together. If a basic block is too big to be processed, it will be divided into overlapping units called *windows*, within which the compiler repeatedly performs optimisations.

Modern compilers support a large number of optimisation techniques [7], e.g., common subexpression elimination, strength reduction, code motion, constant value propagation and evaluation, induction variable simplification, etc. In order to avoid unnecessary manual optimisations, it is important to be aware of the optimisation capability of a compiler. This section briefly introduces some standard optimizations that are referred to in later chapters.

**Strength Reduction**

Strength reduction is the transformation of replacing an arithmetic expression by an equivalent but relatively cheaper expression. For example, if variable $i$ is an integer, $2 \times i$ can be replaced by $i + i$, since integer addition is usually faster than integer multiplication. This transformation has no improvement on floating-point number operations due to the same number of cycles needed to issue floating-point number additions and multiplications. A typical example of strength reduction for floating-point number operations is the replacement of $x^2$ by $x \times x$.

**Loop-invariant Code Motion**

Code motion is an optimisation that moves a computation from inside a loop to outside the loop. This optimisation can only be performed if the computation always produces the same result each time around the loop, i.e. the computation is loop-invariant. For example, in the following loop:

```plaintext
DO 10 i = 1, 10
    v(i) = a * b * v(i)
10 END DO
```
$a \times b$ will be precalculated outside the loop by the compiler so that it will have the same effect as:

```plaintext
t = a \times b
DO 10 i = 1, 10
   v(i) = t \times v(i)
10 END DO
```

### Register Allocation and Instruction Scheduling

Because no deterministic non NP-complete algorithms could solve all the optimisation problems in an optimal way, the practical implementations are usually heuristics based on, and leading to, non-optimal solutions [4]. This topic is too rich to be covered in detail here so only a few basic principles of this optimisation are introduced next.

Table 3.5 shows the standard types of dependence. When one instruction is dependent on another, they cannot be executed simultaneously, i.e., the second instruction usually has to wait for a few cycles until the first one is finished.

<table>
<thead>
<tr>
<th>True Dependence</th>
<th>Anti-Dependence</th>
<th>Output Dependence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_2 = r_3 + r_4$</td>
<td>$r_2 = r_3 + r_4$</td>
<td>$r_2 = r_3 + r_4$</td>
</tr>
<tr>
<td>$r_5 = r_6 - r_2$</td>
<td>$r_3 = r_6 - r_7$</td>
<td>$r_2 = r_6 - r_7$</td>
</tr>
</tbody>
</table>

Table 3.5: Three Types of Dependence.

In the following example, the second line and fourth line need the result from the first line and the third line, respectively. The first line and the third can be executed by a single instruction, namely, fused multiply-add (FMA) on many RISC processors. Supposing a new instruction can start in each cycle but the result takes two cycles to be returned, the following code segment takes six cycles due to the idle cycles caused by delay of results from the first and the third lines.
These idle cycles can be easily eliminated by reordering the instructions as in the following. The values of $x$ and $z$ required are not needed immediately so that by reordering the instructions, only four cycles are needed instead of six in the original code segment.

\[
\begin{align*}
x &= 1 + 2 \times y \\
v1 &= x + v1 \\
z &= 1 + 2 \times w \\
v2 &= z - v2
\end{align*}
\]

Anti-dependence and output dependence can be eliminated by using another register. For example, in the anti-dependence of Table 3.5, the result of $r6 - r7$ can be stored in another unused register rather than $r3$.

### 3.3.3 Loop Optimisation

Loops occur commonly in the form of while statements, do-while statements or for statements in many programming languages. Many programs spend a large amount of time executing instructions in loops, in particular, for programs like NLA libraries which make heavy use of loops. Therefore, it is especially important that a compiler can generate optimised code for loops. By applying loop transformations, a loop nest can be transformed into another form to reduce different kinds of overhead, such as cache misses, while the program semantics are preserved. This section introduces some commonly used loop optimisation techniques.
Loop Interchange

Loop interchange is one of the most useful transformations for supporting parallelisation and memory hierarchy management [74]. It is achieved by exchanging the order of the two loops in a perfect loop nest. For example, the following loop nest

\[
\begin{align*}
\text{DO } & I = 1, 10 \\
\text{DO } & J = 1, 10 \\
& A(I,J) = 0 \\
\text{END DO} \\
\text{END DO}
\end{align*}
\]

is inefficient in Fortran as Fortran stores arrays by columns. Compilers that support this technique can reorder the loop nest for achieving better cache performance, as shown in the following:

\[
\begin{align*}
\text{DO } & J = 1, 10 \\
\text{DO } & I = 1, 10 \\
& A(I,J) = 0 \\
\text{END DO} \\
\text{END DO}
\end{align*}
\]

Moreover, the following transformation enables parallelisation by vectorising the inner loop:

\[
\begin{align*}
\text{DO } & J = 1, 10 \\
\text{FORALL } & (I = 1:10) \\
& A(I,J) = 0 \\
\text{END FORALL} \\
\text{END DO}
\end{align*}
\]

Note that loop interchange may not always be legal due to the violation of dependencies, as illustrated in the following example:

\[
\begin{align*}
\text{DO } & I = 1, 10 \\
\text{DO } & J = 1, 10 \\
& A(I, J+1) = A(I+1, J) \\
\text{END DO} \\
\text{END DO}
\end{align*}
\]
In this example, the first assignments before or after loop interchange are the same: \( A(1,2) \) is assigned the value of \( A(2,1) \). However, the second statement before the transformation is assigning the value of \( A(2,2) \) to \( A(1,3) \), whereas input and output of the assignment are reversed after loop interchange.

**Loop Skewing**

Loop skewing is a transformation which changes the value of both loop bound and loop index. By doing this, the iteration space is changed but the data dependencies are maintained. The purpose of this is to enable parallelism. For example, consider the following loop nest:

\[
\begin{align*}
& \text{DO } I = 2, 5 \\
& \hspace{1em} \text{DO } J = 2, 5 \\
& \hspace{2em} A(I, J) = A(I-1, J) + A(I, J-1) \\
& \hspace{1em} \text{END DO} \\
& \text{END DO}
\end{align*}
\]

Parallelisation cannot be applied directly due to the dependencies carried in both \( I \) and \( J \) loops. As shown in Figure 3.2, each element of \( A \) is dependent on two elements, the one on its left and the one beneath it. However, it is easily noticed that the elements on the diagonals can be executed in parallel, e.g., \( A(5,2), A(4,3), A(3,4) \) and \( A(2,5) \) can be calculated in parallel. By defining a new variable \( K = I + J \), the original loop nest can be transformed to:

\[
\begin{align*}
& \text{DO } I = 2, 5 \\
& \hspace{1em} \text{DO } K = 2+I, 5+I \\
& \hspace{2em} A(I, K-I) = A(I-1, K-I) + A(I, K-I-1) \\
& \hspace{1em} \text{END DO} \\
& \text{END DO}
\end{align*}
\]

Note that, although this loop still carries dependencies, by performing loop interchange, the code above can be transformed to:

\[
\begin{align*}
& \text{DO } K = 4, 10 \\
& \hspace{1em} \text{DO } I = \max(2, K-5), \min(5, K-2) \\
& \hspace{2em} A(I, K-I) = A(I-1, K-I) + A(I, K-I-1) \\
& \hspace{1em} \text{END DO} \\
& \text{END DO}
\end{align*}
\]
Figure 3.2: The dependencies in A.

This new loop nest contains no dependencies so that vectorisation can be applied:

```
DO K = 4, 10
    FORALL (I = max(2,K-5):min(5,K-2))
    END FORALL
END DO
```

This code still has two disadvantages, namely varying vector size and recomputation needed on vector bounds [74]. The solution to these disadvantages is not further discussed here as they are out of the scope of this section.

**Index Set Splitting**

Index set splitting is a technique which removes a loop variant inside the loop body by splitting the original loop into multiple adjacent loop nests, where each of them iterates over a subset of the original iteration.

For example, the following code
DO J = 1, 100
    IF (J.GT.TEMP) THEN
        A(J) = J
    ELSE
        A(J) = J - 60
    END IF
END DO

can be transformed to

DO J = max(1, TEMP+1), 100
    A(J) = J
END DO
DO J = 1, min(100, TEMP)
    A(J) = J - 60
END DO

By removing the conditional statements from the original loop, index set splitting enables other loop optimisation techniques such as loop interchange [15]. The trade-off of applying index set splitting is that it may increase the code size due to duplication of the original loop.

**Loop Unrolling**

Loop unrolling is an optimisation that simply replicates the loop body multiple times and increases the loop termination boundaries correspondingly. It was originally used for reducing the overhead for loop iterations [8], such as reducing the pointer arithmetic for moving the index. On modern computers, the primary benefit from it is improved scheduling [64, 8] so that instructions from different iterations can be scheduled together, i.e., increased instruction-level parallelism. A loop such as:

```
DO J = 1, N
    A(J) = i
END DO
```

can be transformed to:
DO  J = 1, N, 4
    A(J) = i
    A(J+1) = i
    A(J+2) = i
    A(J+3) = i
END DO

Note that loop unrolling may harm the program’s performance if it is not applied properly. For instance, excessive unrolling can cause an increase to the size of the register working set so that the unrolled loop body may exceed the limit of registers available to the user [28], or the unrolled code size may overflow the first-level instruction-cache [36]. [102] proposes a method to automatically select unroll factors for perfectly nested loops, and thereby generate compact code based on these factors.

Loop Tiling

Consider a typical code for matrix-matrix multiplication, as shown below:

DO  J = 1, N
    DO  I = 1, N
        DO  K = 1, N
            C(I,J) = C(I,J) + A(I,K) * B(K,J)
        END DO
    END DO
END DO

This involves $2N^3$ operations and $3N^2$ memory locations, which tells us that the algorithm is computationally intensive. When the first iteration of the outermost loop is executed, i.e., starting with $J = 1$, we need the first column of $B$ and all the elements of $A$, accessed row by row, as shown in Figure 3.3, where the matrix is stored in column-major format and each rectangle represents a cache line.

Normally, accessing $B$ would not burden the cache, as one column of $B$ is spread among $\frac{N}{E}$ cache lines, where $E$ is the number of elements that can be held in a cache line. Only $\frac{N}{E}$ cache misses occur for a column, and the total number of cache misses is $\frac{N^2}{E}$.

However, when only one column of $B$ is used, all the elements of $A$ are required. The elements of $A$ are accessed row by row but the matrix is stored by
columns, hence each element of $A$ causes a cache miss. Once a row of $A$ has been read, the next row may or may not cause further cache misses, depending on whether $N$ cache lines can be held in the cache and whether there is any other use of the cache that may force some of the cache lines to be expelled. In the latter case, the next few rows of $A$ can be read without cache misses until $I = E$, where another $N$ cache lines of data need to be put into cache. Therefore, the first iteration of the outermost loop with $J = 1$ takes between $\frac{N^2}{E}$ and $N^2$ caches misses. Furthermore, the remaining iterations of the outermost loop ($J = 2, 3, \text{etc.}$) may bring in additional cache misses when reading $A$. If the cache is big enough so that $\frac{N^2}{E}$ cache lines holding $A$ will not be pushed out of the cache when $J$ increases, $2\frac{N^2}{E}$ cache misses will occur: half for reading $A$ and half for reading $B$. However, if the cache cannot hold all the elements of $A$, when $J$ increases, $A$ needs to be brought back to cache again. In this case, the number of cache misses is $\frac{N^3}{E} + \frac{N^2}{E}$. Worst of all, if the cache cannot even hold a row of $A$, the number of cache misses becomes $N^3 + \frac{N^2}{E}$.

It is clear that changing the layout of the data structure would not improve the locality of this program. If $A$ is stored in row-major format, although the reuse of cache lines can be improved, it may compromise another matrix multiplication, where $A$ plays the role of $B$. A loop transformation technique called loop tiling [113] has been introduced to improve cache reuse by reordering the execution order of the instructions. In this example, the matrix can be divided into submatrices (typically square) called blocks, which have $NB$ as the submatrix
order, as shown in Figure 3.4.

\begin{figure}[h]
\centering
\includegraphics[width=0.3\textwidth]{matrix_blocked_submatrices.png}
\caption{Matrix blocked by submatrices ($NB \times NB$).}
\end{figure}

The original loop nest can be transformed to:

\begin{verbatim}
DO JJ = 1, N, NB
  DO II = 1, N, NB
    DO KK = 1, N, NB
      DO J = JJ, JJ+NB
        DO I = II, II+NB
          DO K = KK, KK+NB
            C(I,J) = C(I,J) + A(I,K) * B(K,J)
          END DO
        END DO
      END DO
    END DO
  END DO
END DO
\end{verbatim}

where $C$ is assumed to be a zero matrix and $NB$ divides $N$. When the matrices cannot fit in the cache, choosing $NB$ properly would significantly decrease the number of cache misses. A pair of blocks from $A$ and $B$ can be brought to the cache by having $\frac{2NB^2}{E}$ cache misses. Because the computational cost for the entire matrix is $2N^3$ and each block requires $2NB^3$ operations, the number of times for a pair of blocks to be brought into cache is $\left(\frac{N}{NB}\right)^3$. The total number of cache misses is calculated as $\frac{2NB^2}{E} \times \left(\frac{N}{NB}\right)^3$, i.e., $\frac{2N^3}{NB^2E}$. Although this number will not
benefit cache reuse if the matrices can be fully stored in the cache,\(^4\), cache reuse can be significantly improved when the complexity of the unblocked algorithm is \(\frac{N^3}{E} + \frac{N^2}{E}\) and \(N^3 + \frac{N^2}{E}\), by choosing a proper value for \(NB\).

### 3.3.4 Iteration Space Partitioning

By resembling other loop transformation techniques such as index set splitting, a general method, namely *Iteration Space Partitioning* (ISP), is introduced in this section to isolate a loop-body for all iterations of a loop that lie within a polyhedral set defined by a system of linear inequalities in the loop indices [22].

**Iteration Spaces**

Nested for-loops define an *iteration space*, such as occurs in band matrix multiplication. The set \(\text{IS}\) of all the iterations defined by the nested for-loop indices is called the iteration space, and it can be represented by the solution set for a system of linear inequalities \(Ax \leq b\). For example, band matrix-matrix multiplication (Figure 5.9) can be described as:

\[
\begin{bmatrix}
-1 & 0 & 0 \\
1 & 0 & 0 \\
0 & -1 & 0 \\
0 & 1 & 0 \\
-1 & 1 & 0 \\
1 & -1 & 0 \\
0 & 0 & -1 \\
1 & 0 & -1 \\
0 & 1 & -1 \\
-1 & 0 & 1 \\
0 & -1 & 1
\end{bmatrix} \begin{bmatrix}
i \\
j \\
k
\end{bmatrix} \leq \begin{bmatrix}
1 \\
m \\
nu \\
m \\
mu \\
ml \\
mnl \\
mub \\
mua \\
mlb
\end{bmatrix}
\]

In such a loop, the loop index \(I_i\) (such as \(i, j,\) or \(k\) above) iterates over all the integers between lower bound \(L_i\) and upper bound \(U_i\), which is called the *execution set* of that loop. Each individual loop bound is called *admissible* [21]

\(^4\)By choosing \(N = NB\), i.e., treating the whole matrix as a single block, \(\frac{2N^2}{NB}E\) becomes \(\frac{2N^2}{E}\), which is the same as in the unblocked algorithm.
if it satisfies one of the following conditions:

- a simple lower bound \((L_i)\) or upper bound \((U_i)\) is called admissible if it can
  be expressed in the following way:
  \[
  L_i = l_{i0} + \sum_{j=1}^{i-1} l_{ij} \cdot I_j,
  \]
  \[
  U_i = u_{i0} + \sum_{j=1}^{i-1} u_{ij} \cdot I_j,
  \]
  where all \(l_{ij}, u_{ij} \in \mathbb{Z}\) and \(l_{ii}, u_{ii} > 0\).

- a loop bound that contains the maximum or minimum of a number of single
  admissible loop bounds. This kind of loop bound is called a compound
  bound.

Obviously, the loop bounds of band matrix multiplication are admissible. Ex-
pressions such as \(i \cdot j\) are not. The reason for distinguishing these is that only
admissible bounds can be used in the process of Fourier-Motzkin elimination (see
below). See [21] and [78] for the details of admissible loop bounds.

**Fourier-Motzkin Elimination**

Fourier-Motzkin elimination [35] is an algorithm used to eliminate variables for
a given system of linear inequalities with a number of variables. A variable \(x_k\) is
eliminated by replacing each pair-wise combination of inequalities, which defines a
lower bound and an upper bound, respectively. This is illustrated in the following:

\[
\begin{align*}
L & \leq a_1 \cdot x_k \\
\frac{a_2}{a_1} \cdot x_k & \leq U
\end{align*}
\]

\( \implies \frac{a_2}{g} \cdot \frac{1}{L} \leq \frac{a_1}{g} \cdot \frac{1}{U} \)

where \(g\) is the greatest common factor of the two positive numbers \(a_1\) and \(a_2\).

The elimination of variable \(x_k\) from a system of linear inequalities creates
another system of linear inequalities without \(x_k\), such that both systems have
the same solution sets over the remaining variables. Assuming a system \(S\) with
\(n\) inequalities and \(m\) variables from \(x_1\) to \(x_m\), the system is divided into three
groups:

- the group \(A\) of inequalities in which \(x_k\) is irrelevant;
the group $B$ of inequalities involving $x_k$, where $x_k \geq L_1, \ldots, L_{nb}$, where $nb$ is the number of such inequalities;

the group $C$ of inequalities involving $x_k$, where $x_k \leq U_1, \ldots, U_{nc}$, where $nc$ is the number of such inequalities.

Each lower bandwidth $L$ or upper bandwidth $U$ may include other variables except $x_k$. The intersection of $B$ and $C$ is equivalent to $\max(L_1, \ldots, L_{nb}) \leq x_k \leq \min(U_1, \ldots, U_{nc})$. Obviously, this formula can be further simplified to $\max(L_1, \ldots, L_{nb}) \leq \min(U_1, \ldots, U_{nc})$, which contains $(nb \cdot nc)$ inequalities, and group this together with the inequalities of group $A$ to form a new system without $x_k$. The output system has $(n - nb - nc) + (nb \cdot nc)$ inequalities. The computational cost of the Fourier-Motzkin elimination algorithm is double-exponential ($n^{2^q}/4^q$ with $q$ quantifiers to eliminate, where $q \leq m$).

**Simplifying Loop Bounds**

The system of linear inequalities derived from the nested for-loops may have an overhead in computing the loop bounds. Fourier-Motzkin elimination can be used to detect and remove this overhead. Given a nested for loop that contains loop $LP$ with lower bound $L$ and upper bound $U$, the loop bounds are classified into four groups [79]:

- Empty loops: $L \geq U$, i.e., no values within the loop bounds and hence this loop including those underneath can be simply removed. An example of an empty loop is given in the following:

  \begin{verbatim}
  DO i = 1, 100
    DO j = 100, 100 - i
      S1
    END DO
  END DO
  \end{verbatim}

  Eliminating $j$ from $100 \leq j \leq 100 - i$ gives $i \leq 0$. This new inequality and $1 \leq i \leq 100$ yield $1 \leq 0$, which indicates that this system has no solution. Thus the entire loop nest may be removed.
• Redundant bounds: In loop \( LP \), \( L \) or \( U \) is a compound bound, e.g., in the form \( \max(e_1, e_2, \ldots, e_m) \), where \( e_1, e_2, \ldots, e_m \) are single bounds. \( e_k \) is redundant if it is greater than or equal to all the other bounds. A redundant bound can be removed from a compound bound without affecting the iteration space. For example:

\[
\begin{align*}
&\text{DO } i = 1, 100 \\
&\quad \text{DO } j = 1, \min(50, i + 50) \\
&\quad \quad \text{S1} \\
&\quad \text{END DO}
\end{align*}
\]

\[
\begin{align*}
&\text{DO } i = 1, 100 \\
&\quad \text{DO } j = 1, 50 \\
&\quad \quad \text{S1} \\
&\quad \text{END DO}
\end{align*}
\]

As proposed in [11], a linear inequality is redundant to a system of linear inequalities if the system derived by negating this inequality is inconsistent. In the above example, the negation of \( 50 \leq i + 50 \) is \( 50 \geq i + 51 \), i.e. \( -1 \geq i \). With \( 1 \leq i \leq 100 \), Fourier-Motzkin elimination generates \( 1 \leq -1 \) and, therefore, the upper bound of \( j \) can be simplified to 50.

• Partially empty loops: Although \( LP \) may not be empty, \( L \geq U \) for some of the values in its outer loops. For example:

\[
\begin{align*}
&\text{DO } i = 1, 100 \\
&\quad \text{DO } j = 1, i - 50 \\
&\quad \quad \text{S1} \\
&\quad \text{END DO}
\end{align*}
\]

\[
\begin{align*}
&\text{DO } i = 51, 100 \\
&\quad \text{DO } j = 1, i - 50 \\
&\quad \quad \text{S1} \\
&\quad \text{END DO}
\end{align*}
\]

The application of Fourier-Motzkin elimination to \( 1 \leq i \leq 100 \) and \( 1 \leq j \leq i - 50 \) yields \( \max(1, 51) \leq i \leq 100 \). Therefore, the lower bound of \( i \) can be replaced by 51.

• Compound bounds: Consider the case where the loop \( LP \) is compound, such as \( U = \min(e_1, e_2, \ldots, e_m) \). This can be simplified to either \( e_i \) or \( \min(e_1, e_2, \ldots, e_{i-1}, e_{i+1}, \ldots, e_m) \). By doing this, all the levels that lie above this level are duplicated and each of them has a new but less complex
compound bound that has the complexity reduced from evaluating $m$ to evaluating either one or $m - 1$ single bounds. By iterating the process described above, all the compound bounds are eventually pushed to the outermost level and can then be written as a set of nested loops without compound bounds. For instance:

\[
\begin{align*}
&\text{DO } i = 1, 50 \\
&\quad \text{DO } j = 1, i \\
&\quad \quad \text{S1} \\
&\text{DO } i = 1, 100 \\
&\quad \quad \quad \text{END DO} \\
&\quad \quad \text{DO } j = 1, \min(i, 100 - i) \\
&\quad \quad \quad \text{END DO} \\
&\quad \quad \text{S1} \\
&\quad \quad \text{END DO} \\
&\quad \quad \text{DO } i = 51, 100 \\
&\quad \quad \quad \text{S1} \\
&\quad \quad \text{END DO} \\
&\quad \quad \text{END DO} \\
&\text{END DO}
\end{align*}
\]

Although the computational complexity of Fourier-Motzkin elimination is double exponential, fast implementations exist for nested loops with small depth [98, 83]. Moreover, the computational cost of min/max functions is typically moved from run time to compile time, if this method is implemented by a compiler.

**Removing Affine IF Statements**

Suppose a loop nest contains an affine IF statement such as:

\[
\begin{align*}
&\text{DO } i = 1, m \\
&\quad \text{DO } j = 1, n \\
&\quad \quad \text{IF } (i - j \leq e) \text{ THEN} \\
&\quad \quad \quad \text{S1} \\
&\quad \quad \text{ELSE} \\
&\quad \quad \quad \text{S2} \\
&\quad \quad \text{END IF} \\
&\quad \text{END DO} \\
&\text{END DO}
\end{align*}
\]

\footnote{An IF statement is called affine IF if it contains affine condition. A condition $e \leq e'$ or $e \geq e'$ is an affine condition where both side are affine expression.}
The affine condition \((i - j <= \epsilon)\) defines a subspace of the iteration space of the loop nest. By introducing this affine IF condition into the loop nest, the original set of linear inequalities is extended by adding one more inequality, resulting in three possibilities:

1. The system becomes inconsistent, which means the statement \(S_1\) will never be executed so that the IF-ELSE statement can be replaced by the ELSE-part, i.e., executing \(S_2\) directly, without affecting the semantics;

2. The IF affine condition is redundant to the original set of linear inequalities, which means the ELSE-part will never be executed and the original IF-ELSE can be replaced by \(S_1\); or

3. Neither the loop is inconsistent nor the condition is redundant. In this case, both IF- and ELSE-parts need to be executed sometimes. They form mutually exclusive parts of the original iteration space. By using Fourier-Motzkin elimination, the original loop nest with affine IF-ELSE statement can be replaced by introducing additional loop bounds. An example of this transformation is given in Figure 3.5.

```plaintext
DO j = 1, 100
  DO i = 1, 100
    IF (i - j <= 60) S1
    ELSE -> END DO
    S2
  END DO
END DO
```

Figure 3.5: Removing an affine IF-ELSE statement by introducing additional loop bounds.

### 3.3.5 Polyhedral Model

Complex transformations, such as loop tiling and loop skewing, modify the execution order to achieve parallelisation and optimisation. Such loop transformations
are difficult to be performed automatically. The polyhedral model was proposed to address this issue. Generally speaking, it firstly transforms the program parts that fit into the model into the equivalent linear-algebraic representation, then it uses a scattering function on the linear-algebraic representation to change its execution order. Finally, the new code is generated by implementing the execution order. A notable implementation of the polyhedral model is CLooG, which is used to generate code for scanning Z-polyhedra [18].

The properties of the polyhedral model are roughly summarised in [19] as:

1. loops with affine bounds, IF conditionals with affine conditions, or using max, min, ceiling and floor operators on them [114]; and
2. the affine bounds and conditions depend only on constants or outer loop counters.

For example, the Cholesky factorisation kernel given in Figure 3.6 follows the properties of the polyhedral model. Such a set of statements is called a static control part (SCoP).

```fortran
DO i = 1, n
    x = a(i,i)
    DO j=1, i-1
        x = x - a(i,j)**2
    END DO
    p(i) = 1.0/sqrt(x)
    DO j=i+1, n
        x = a(i,j)
        DO k=1, i-1
            x = x - a(j,k)*a(i,k)
        END DO
        a(j,i) = x*p(i)
    END DO
END DO
```

Figure 3.6: A Cholesky factorisation kernel from [19].

---

6http://www.cloog.org/
7Z-polyhedra is the set of integer points in a convex polyhedron.
8A SCoP is a maximal set of consecutive statements, where the loop bounds and conditionals only depend on constants within the set of statements [20]. A while loop which may terminate due to constants declared outside of the statements, is not a SCoP.
Program execution is described using two specifications in the polyhedral model [19]:

- One specification is the iteration domain $D$, i.e., the set of values for which the statement is executed. If a statement is within a SCoP, the iteration domain can always be defined by a set of linear inequalities (a polyhedron). The polyhedron denotes a convex set of points in a lattice:

$$D = \{ x \mid x \in \mathbb{Z}, Ax \geq b \},$$

where $x$ is the iteration vector, $A$ is a constant matrix and $b$ is a constant vector. For example, the iteration domain of S2 in Figure 3.6 is:

$$\begin{bmatrix}
1 & 0 \\
-1 & 0 \\
0 & 1 \\
1 & -1
\end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \geq \begin{bmatrix} 1 \\ -n \\ 1 \\ 1 \end{bmatrix}$$

- The other specification is a scattering function $\theta(x)$. Although the set of instances for each statement is defined by the iteration domain, other factors, such as the order of the instances, are missing. [96] proposes a way to express the execution order, by giving each instance an execution date:

$$\theta(x) = Tx + t,$$

where $T$ is a constant matrix and $t$ is a constant vector. The first element in vector $x$ can be seen as the year (most significant), next as the month (less significant), etc. The sequential order of the program in Figure 3.6 can be expressed by an abstract syntax tree, as shown in Figure 3.7. The execution order can be easily captured, e.g., $\theta_{S1}(x_{S1}) = (0, i, 0)T$ and $\theta_{S2}(x_{S2}) = (0, i, 1, j, 0)T$.

Obviously, the properties of the polyhedral model summarised in this section are very similar to the admissible loop bounds and affine IF Statement in iteration space partitioning, except for the extension of using ceiling and floor operators. Moreover, the first specification of the polyhedral model, namely, the iteration domain, is described in the same way as in iteration space partitioning. However, on the one hand, iteration space partitioning does not provide a
method, such as the scattering function, for how the control flow can be managed in a SCoP. Therefore, compared with iteration space partitioning, the polyhedral model is capable of scanning multiple polyhedrons, each of which corresponds to an iteration domain for a specific statement in the nested loops. On the other hand, in the polyhedral model, although the execution domain and execution order for each statement are described by the polyhedral domain and scattering function, respectively, the maximal and minimal functions for the loop counters still have to be computed repeatedly. In iteration space partitioning, this issue is addressed by the simplification of compound bounds.

3.4 Summary of Chapter

This chapter has introduced some key concepts in building a complex system, namely, component-based development (CBD), crosscutting concerns, aspect-oriented programming (AOP) and compiler optimisation techniques.
CBD implies that complex systems are built by constructing individual components and merging these to form the required system. One big problem of current CBD is that, although the majority of concerns can be handled by each component, a few concerns cutting across others cannot be fitted into any single component.

AOP is a technique to handle these crosscutting concerns in a modular way by providing a new abstraction, namely the aspect. This can improve productivity in software development.

A high-level language compiler may also improve productivity by hiding the low-level details, particularly by providing a set of optimisation techniques to gain improved performance. In particular, the mechanism of around advice of the pointcut-advice model in AOP is very similar to that of the compiler optimisation techniques. They both capture some structures in a program then replace them with something else which does the same work but with improved features such as faster execution.

The detailed relationships between these two techniques are discussed in detail in Section 7. The next two chapters introduce how these two methodologies, namely AOP and compiler optimisation techniques can be applied to build a NLA library.
Chapter 4

Crosscutting Concerns in NLA Libraries

In this chapter, some of the libraries introduced in Chapter 2.2 are used as demonstrative examples to show how crosscutting concerns impact on source code. Sections 4.1 to 4.3 conclude that three sorts of crosscutting concerns naturally reside in many Numerical Linear Algebra libraries and show how AOP can be applied to modularise them. The three kinds of crosscutting concerns correspond to the crosscutting concern sorts as were introduced in Section 3.1.4 are summarised in Section 4.4.

4.1 Case Study: Grouping Matrix Operations

As described in Section 3.1.3, non-modularisation usually leads to code scattering and tangling at source code level. However, the representations at source code from different programming paradigms may vary. This section firstly describes the code scattering problem in an object-oriented NLA library (JOOLALA) and how AOP can be utilised to solve it, then extends the idea of grouping matrix operations to some traditional libraries, illustrating some Fortran 90 implemented BLAS subroutines as examples.

4.1.1 Modularising Matrix Operations in OONLA Library

The operations of the current JOOLALA implementation are implemented in a series of collaborating classes: `Property`, `DenseProperty`, `BandedProperty`,
### Elementary Operations

- Addition
- Subtraction
- Multiplication
- Scaling
- Transpose
- Absolute Column Sum Norm
- Absolute Row Sum Norm
- Trace
- Minor

### Complex Operations

- Solving Linear System of Equations
- Determinant
- Inverse

### Factorization

- Cholesky
- LU

Table 4.1: Supported operations of JOOLALA.

UpperTriangularProperty, LowerTriangularProperty and SymmetricProperty. Some important operations are shown in Table 4.1. Class Property provides an interface and its subclasses implement the interface using different algorithms.

Different implementations of matrix operations for a certain matrix property, namely the lower triangular property, are revealed in Figure 4.1. Figures 4.2 and 4.3 show how matrix addition is implemented with two different properties. After analysing these implementations, it is found that almost all matrix operation definitions in each concrete property class share the same two-part structure: constructing the structure to traverse all elements of a matrix (construction part) and performing certain operations on these elements to get a result (assignment part). The result of an operation can be a matrix, a vector and/or numbers, etc.

For many (but not all) elementary matrix operations, the construction part is a nested for-loop and the assignment part is a method call for assigning elements to form a new matrix. Examples of these include: matrix addition, matrix scale, and matrix multiplication. Two disciplines may be summarised, as follows:

- The nested for-loops of different matrix operations in each concrete property class share the same structure (Figure 4.1);
- The method calls in different concrete property classes implement matrix element assignment using the same algorithms (Figures 4.2 and 4.3).
public final class LowerTriangularProperty extends Property {
    ...
    public final void add(Matrix A, Matrix B) throws OolalaException {
        for (int j=1; j<=numColumns(); j++) {
            for (int i=j; i<=numRows(); i++) {
                assign(i, j, (A.element(i, j) + B.element(i, j)));
            }
        }
    }
    
    public final void scale(Matrix A, double n) throws OolalaException {
        for (int j=1; j<=numColumns(); j++) {
            for (int i=j; i<=numRows(); i++) {
                assign(i, j, A.element(i, j) * n);
            }
        }
    }
    
    public final void mult(Matrix A, Matrix B) throws OolalaException {
        for (int j=1; j<=B.numColumns(); j++) {
            for (int i=j; i<=A.numRows(); i++) {
                for (int k=j; k<=i; k++) {
                    assign(i, j, (element(i, j) + A.element(i, k) * B.element(k, j)));
                }
            }
        }
    }
    ...
}

Figure 4.1: Various method definitions in class LowerTriangularProperty.

The disciplines show that several matrix operations, such as method add of class LowerTriangularProperty, are implemented as combinations of two concerns. The first concern is to traverse all the matrix elements by nested for-loops, which is scattered over different methods of a class, such as methods add, scale and mult in class LowerTriangularProperty (Figure 4.1). The second concern, which is simply to perform a matrix element assignment, is scattered over different classes that implement a matrix operation, such as method add in class DenseProperty and BandProperty (Figure 4.2 and Figure 4.3).

When these two concerns are realised, the resulting code has parts that are...
public final class DenseProperty extends Property {
 ...
    public final void add (Matrix A, Matrix B) throws OolalaException {
        for (int i=1; i<=numRows(); i++) {
            for (int j=1; j<=numColumns(); j++) {
                assign(i, j, (A.element(i, j) + B.element(i, j)));
            }
        }
    }
...
}

Figure 4.2: Addition method definition in class DenseProperty.

public class BandedProperty extends Property {
 ...
    public void add (Matrix A, Matrix B) throws OolalaException {
        for (int i=1; i<=numRows(); i++) {
            for (int j=1; j<=numColumns(); j++) {
                if (i-j>=-upperBandwidth && i-j<=lowerBandwidth)
                    assign(i, j, (A.element(i, j) + B.element(i, j)));
            }
        }
    }
...
}

Figure 4.3: Addition method definition in class BandedProperty.

scattered and tangled over different components in two dimensions (matrix property and matrix operation). Therefore, one can modularise the implementations of matrix operations using aspect-oriented techniques to handle the two concerns separately, by applying the idea described in Figure 4.4.

while the applicable matrix operations of each matrix property are invoked execute the nested for-loop for that property to traverse elements;

while the execution flow reaches the point where the nested for-loop is invoked of a certain matrix operation over different concrete properties call the assignment method for that operation.

Figure 4.4: Pseudo-code for implementing matrix operations with AOP.

AspectJ was chosen to implement this idea due to its seamless integration with Java. However, the original source code needs to be slightly refactored,
in order to apply aspects to it. The reasons for refactoring are specific to this problem and are explained later in this section. Figures 4.5 and 4.6 illustrate how to group dense property construction and addition element assignment. A property construction is the part of code that gives the method for traversing elements of a matrix. In Figure 4.5, the pointcut `denseOperations` captures the desired methods of class `DenseProperty`, and replaces the bodies of these methods by that of the advice. Five (corresponding to the number of matrix properties supported in the library) such pointcut and advice pairs are used for property construction. An element assignment is the part of code that performs an operation on an element. Figure 4.6 shows that the `around` advice of aspect `MatrixOperationAssignmentAspect` executes the addition for a matrix element that is in the execution flow of all the addition methods of different properties inside the property construction. Six (corresponding to the number of matrix operations that can be optimised in this way) such pointcut and advice pairs are used for element assignment. Eventually, the total number of implementations is reduced from \( m \times n \) to \( m + n \), where \( m \) is the number of matrix properties supported and \( n \) is the number of operations that can be applied with this optimisation.\(^1\)

Due to a language limitation in AspectJ, the original source code is slightly refactored to expose certain join points. Two changes have been made. One is that a new method `assignAll` is declared in class `Property` and invoked in every nested for-loop. The current version of AspectJ does not treat a for-loop as a join point. As a result, a new method called `assignAll` is declared in class `Property` and invoked in each nested for-loop, in order to make the construction part capturable. An alternative way of capturing the nested for-loop is by using LoopsAJ \([63]\), which provides an implementation of a join point for loops in AspectJ. However, LoopsAJ is a project for experiments only and may lead to undesirable behaviours for the evaluation process, so it has not been used.

The other refactoring is using metadata `@` to indicate a common characteristic that is shared by many matrix operations. There are two ways of capturing a property-based pointcut: the participant pattern and annotation. Without these two methods, the developer sometimes may be able to find desired operations

\(^1\)Strictly speaking, the number of methods that need to be implemented in this instance is 28 (not 30), due to some implementations are not needed, such as method `transpose` in class `SymmetricProperty`. Therefore, the number of implementations is reduced from 28 to 11 in this instance.
public aspect MatrixOperationConstructAspect
{
    ...
    pointcut denseOperations(DenseProperty p):
        execution(@Operation final void DenseProperty.*(..))
        &&
        target(p);
    
    void around(DenseProperty p) throws OolalaException:
        DenseOperations(p)
        {
            for (int i=1; i<=p.numRows(); i++) {
                for (int j=1; j<=p.numColumns(); j++) {
                    p.assignAll(i, j);
                }
            }
        }
    ...
}

Figure 4.5: Construct for dense property in aspect MatrixOperationConstructAspect.

public aspect MatrixOperationAssignmentAspect
{
    ...
    pointcut assignPlus(Matrix m1, Matrix m2, int i, int j, Property p):
        cflow(execution(@Operation void Property+.add(Matrix, Matrix))
        &&
        args(m1, m2))
        &&
        involveAssignAll(i, j, p);
    
    void around(Matrix m1, Matrix m2, int i, int j, Property p)
        throws OolalaException:
        assignPlus(m1, m2, i, j, p)
        {
            p.assign(i, j, (m1.element(i, j) + m2.element(i, j)));}
    
    pointcut involveAssignAll(int i, int j, Property p):
        call(void Property.assignAll(int, int))
        &&
        args(i, j)
        &&
        target(p);
    ...
}

Figure 4.6: Element assignment for matrix addition in aspect MatrixOperationAssignmentAspect.

with common characteristics by looking into their method signatures. For instance, a method taking two matrices as its parameters and returning a matrix is probably performing a matrix operation. However, a few operations do not
reflect their peripheral characteristics in their signatures and hence lead to difficulties in capturing property-based join points. Moreover, the inability to tightly couple the pointcuts and the property-based join points usually make a system extremely difficult to maintain.

One solution to this problem is to use the participant pattern [81], in which classes contain pointcuts denoting certain characteristics. In this way, the class is aware of the existence of an aspect, i.e. the knowledge of the characteristics is embedded in classes, rather than in aspects. Therefore, it is easier to track changes with the participant pattern. This pattern enables its users to keep changes consistent.

Annotations are data about data. Since JDK 5.0, annotation can be used to insert arbitrary data in the source code [67]. Although an annotation does not do anything by itself, with other tools it can be processed to express metadata relating to the program members. Starting with AspectJ 5.0, matching join points based on the presence or absence of annotations is supported. Compared with the participant pattern, annotation is better supported for the separation of software development, because the participant pattern requires the class developer to have extra knowledge about aspect-oriented programming. Also, annotation needs fewer changes in the structure of class Property and its subclasses. Annotation has therefore been chosen as the method to select property-based join points in this project. Figure 4.7 shows that the property-based methods in class Property are annotated by @Operation. The shared property here is that these methods are all amenable to the optimising methodology described in this section.

4.1.2 Code Scattering in Traditional Libraries

Section 2.2.1 introduced the basis of the BLAS. From the specification of matrix-matrix multiplication shown in Table 2.3, it is easy to notice that the interfaces defined in the BLAS contain only a subset of all the mathematically valid matrix multiplications. This is because of the wide usage of those subroutines. However, after other projects, such as LAPACK, that build upon BLAS were begun, it was realised that many operations, such as matrix copy, are widely used and should be defined as separate routines. As a result, [5] shows a number of proposed routines, including some basic matrix operations:

- Matrix acc and scale: $C \leftarrow \alpha A + \beta B$;
public abstract class Property
{
    ...  
    @Operation() public abstract void scale(Matrix A, double n) throws OolalaException;
    @Operation() public abstract void minor(Matrix A, int i, int j) throws OolalaException;
    @Operation() public abstract void transpose(Matrix A) throws OolalaException;
    @Operation() public abstract void add(Matrix A, Matrix B) throws OolalaException;
    @Operation() public abstract void sub(Matrix A, Matrix B) throws OolalaException;
    @Operation() public abstract void mult(Matrix A, Matrix B) throws OolalaException;
    ...
}

Figure 4.7: AspectJ Version of Matrix Operation Declarations.

- Matrix add and scale: $B \leftarrow \alpha A + \beta B$, $B \leftarrow \alpha A^T + \beta B$;
- Matrix copy: $B \leftarrow A$, $B \leftarrow A^T$; and
- Matrix transpose: $A \leftarrow A^T$.

Because current BLASs only support a subset of the proposed functionality, the code scattering problem caused by grouping matrix operations is not that severe. However, the effects can still be observed in the source code. An example is shown in Fortran 90 BLAS for MasPar [94].

Figures 4.8 and 4.9 show the code fragments for implementing $C \leftarrow \alpha A + \beta B$, where $A$ is a symmetric and $B$ is a dense matrix in the first figure, and both $A$ and $B$ are dense matrices in the second figure. Note that the parts for calculating each element in $C$ are the same, except $A$ is assumed to be square in Figure 4.8. It is possible to handle this assignment concern by using an aspect, which injects the code for assignment to each element in $C$ for many different types of matrices.

4.1.3 Summary

In the object-oriented programming paradigm, basic matrix operations are implemented as methods of class Matrix. Generally, this is the normal form for
CHAPTER 4. CROSSCUTTING CONCERNS IN NLA LIBRARIES

\[
c(1:m,1:n) = c(1:m,1:n) \times \beta \\
\text{...}
\]
\[
\text{forall } (i = 1:m, j = 1:m) \text{ aloc}(i,j) = j-i
\]
\[
* 
\text{if (upper) then}
\]
\[
\text{aloc}(1:m,1:m) = \text{merge}( a(1:m,1:m), \text{transpose}(a(1:m,1:m)),
\]
\[
\text{aloc}(1:m,1:m) \geq \text{zero} 
\]
\[
\text{else}
\]
\[
\text{aloc}(1:m,1:m) = \text{merge}( a(1:m,1:m), \text{transpose}(a(1:m,1:m)),
\]
\[
\text{aloc}(1:m,1:m) \leq \text{zero} 
\]
\[
\text{endif}
\]
\[
* 
\text{c}(1:m,1:n) = \text{c}(1:m,1:n) + 
\]
\[
\text{alpha} \times \text{matmul}(\text{aloc}(1:m,1:m),b(1:m,1:n))
\]
\[
\text{...}
\]

Figure 4.8: F90 BLAS dsymm code fragment.

\[
c(1:m,1:n) = c(1:m,1:n) \times \beta \\
\text{...}
\]
\[
\text{if (nota) then}
\]
\[
\text{if (notb) then}
\]
\[
\text{c}(1:m,1:n) = \text{c}(1:m,1:n) + \alpha \times \text{matmul}(a(1:m,1:k),b(1:k,1:n))
\]
\[
\text{...}
\]

Figure 4.9: F90 BLAS dgemm code fragment.

representing basic matrix operations of Object-Oriented Numerical Linear Algebra (OONLA) Libraries.\(^2\) By Object-Oriented Decomposition (OOD), the NLA library is decomposed into a number of classes, which are each responsible for a certain part of the problem domain. By breaking the library into a number of kinds of matrix with different properties, related operations are grouped together within each type of matrix. However, the representation of one operation for different types of matrices becomes a crosscutting concern. In the procedural programming paradigm, it is noticeable that the concerns for traversing matrix elements and assigning elements are cutting across each other, but certain join points do not emerge clearly enough to be captured by an aspect.

The strength of this mechanism may be weakened or enforced by the number of supported matrix types and matrix operations in a library. If all the matrices are stored in one format, e.g. dense format, the implementations of matrix operations

\(^2\)Current OONLA libraries implement matrix operations in one of three forms: as methods of class Matrix, as methods of a utility class, or as classes. The second representation is not actually in object-oriented form and the third representation is not obvious as an abstraction of NLA [87].
have only one crosscutting concern, i.e. the part of the nested for-loop used to traverse all the elements of the matrix. An example of libraries with only one property stored in one format is JAMA [6]. Also, if a library, such as BLAS, does not provide a comprehensive functionality, i.e., only a few matrix operations are supported, this mechanism will be less useful.

4.2 Case Study: Pre-Condition Checks

4.2.1 Using AOP for Exception Handling

Exception handling [57] is used for improving the modularity of programs that deal with exceptional situations [32]. An abnormal event may occur at any unexpected execution point of a program. Therefore, exception handling is designed to provide reliable software that has the ability to recover from unexpected abnormal situations, such as numeric overflow.

Early theoretical work, such as [81] and [77], often introduced Aspect-Oriented Programming (AOP) as a useful technique to separate exception handling code from normal code. Lippert's study [85] found that aspects can ease the tangling of code related to exception handling and detection and thus result in a reduction in program size, based on their refactoring of a large object-oriented framework. In particular, recent works by Filho [47, 45, 46] claim that such a reduction can only be achieved when the exception code is uniform and context-independent.

It is obvious that a NLA library has much generic exception handling and detection code. There are numerous pre-condition checks, e.g., the number of rows and columns of two input matrices must agree in order to perform matrix addition. These are implemented in the form of either exception handling (try-catch clause) or exception detection (throw clause and throw statement) in an object oriented NLA library. Figure 4.10 and Figure 4.11 illustrate two kinds of pre-condition checks in JOOLALA and JAMA, respectively.

One can write aspect code to modularise these crosscutting concerns when building a NLA library. An example of such a code fragment is given in Figure 4.12. Pointcut checkSize captures the methods, namely, add and sub, that need the pre-condition check. A before advice is applied when the execution points meet.
public final class Matrix {
    ...

    public final void add(Matrix A, Matrix B) throws OolalaException {
        OolalaMatrixCorrectness.checkSize(A, B); // check if size(A)=size(B)
        ...
    }

    public final void sub(Matrix A, Matrix B) throws OolalaException {
        OolalaMatrixCorrectness.checkSize(A, B); // check if size(A)=size(B)
        ...
    }
    ...
}

Figure 4.10: Duplicate calls to checkSize in JOOLALA.

4.2.2 Aspects as Error Handling in BLAS

In the early development of traditional NLA libraries, the pre-condition checks were defined as error handling. For example, an error handler is used in dense and banded BLAS to define some of the minimal scalar input argument checking. The BLAS standard errors are described in the following [1]:

- any operator arguments whose meaning is unspecified is invalid;
- any problem dimension or matrix bandwidth less than zero;
- any vector increment equal to zero;
- any leading dimension less than one; or
- any leading dimension less than the relevant dimension of the problem.

Each BLAS error handler will print an error message describing the error and halt the execution. For instance, the Netlib BLAS implements a error handler for Level 2 and 3 BLAS routines, called xerbla.f, which simply returns the subroutine name that causes the error and an information number (INFO) stating what the error is. Some examples of subroutines calling xerbla.f are shown in Figure 4.13 to Figure 4.15.

It is easy to observe that the pre-condition checks for the same matrix operation but different data types are the same. Although the part for calculating each
CHAPTER 4. CROSSCUTTING CONCERNS IN NLA LIBRARIES

public class Matrix implements Cloneable, java.io.Serializable {
    ...
    public Matrix getMatrix (int i0, int i1, int j0, int j1) {
        ...
        try {
            for (int i = i0; i <= i1; i++) {
                for (int j = j0; j <= j1; j++) {
                    B[i-i0][j-j0] = A[i][j];
                }
            }
        } catch(ArrayIndexOutOfBoundsException e) {
            throw new ArrayIndexOutOfBoundsException("Submatrix indices");
        }
        return X;
    }
    ...
}

public Matrix getMatrix (int[] r, int[] c) {
    ...
    try {
        for (int i = 0; i < r.length; i++) {
            for (int j = 0; j < c.length; j++) {
                B[i][j] = A[r[i]][c[j]];
            }
        }
    } catch(ArrayIndexOutOfBoundsException e) {
        throw new ArrayIndexOutOfBoundsException("Submatrix indices");
    }
    return X;
}

public Matrix getMatrix (int i0, int i1, int[] c) {
    ...
    try {
        for (int i = i0; i <= i1; i++) {
            for (int j = 0; j < c.length; j++) {
                B[i-i0][j] = A[i][c[j]];
            }
        }
    } catch(ArrayIndexOutOfBoundsException e) {
        throw new ArrayIndexOutOfBoundsException("Submatrix indices");
    }
    return X;
}

Figure 4.11: Duplicate try-catch clauses at different places in JAMA.

matrix element can be different for real and complex matrices, the pre-condition checks remain the same, as shown in Figure 4.13 and Figure 4.14. Also, the
public aspect CheckCorrectnessAspect {
    ...
    pointcut checkSize(Matrix a, Matrix b) :
        (execution(final void Matrix.add(..)) ||
         execution(final void Matrix.sub(..))) &&
             args(a, b);

    before(Matrix a, Matrix b) throws IncompatibleMatrixException :
        checkSize(a, b)
    {
        if (a.numRows() != b.numRows() || a.numColumns() != b.numColumns())
            { throw new IncompatibleMatrixException
                ("Error => size(A) != size(B)." );
            }
    }
    ...
}

Figure 4.12: Implementing the checkSize concern for JOOLALA with AspectJ.

implementation of the same matrix operations for different matrix properties all
check some unified conditions, such as the matrix order cannot be negative, as
shown in Figure 4.13 and Figure 4.15.

The idea has been introduced that AOP’s pointcut-advice model requires the
language to capture certain join points for further code injection. Because the
statements for pre-condition checks are always executed before computing the
matrix operation, a possible approach may be as follows:

- Remove the pre-condition checks from all the subroutines;
- Categorise the pre-condition checks into different groups; and then
- Use AOP to provide appropriate advice, each of which handles a group of
  checks dedicated for some matrix operation subroutines.

4.2.3 Summary

The applicability of pre-condition checks is much wider than that of grouping
matrix operations. Pre-condition checks are used to ensure that certain conditions
are met before performing the matrix operation so that it does not rely on the
supported matrix properties or operations in a library. However, note that the
conditions that need to be checked may be different for different program designs.
Test the input parameters.

INFO = 0
IF ((.NOT.NOTA) .AND. (.NOT.LSAME(TRANSA,'C')) .AND. (.NOT.LSAME(TRANSA,'T'))) THEN
  INFO = 1
ELSE IF ((.NOT.NOTB) .AND. (.NOT.LSAME(TRANSB,'C')) .AND. (.NOT.LSAME(TRANSB,'T'))) THEN
  INFO = 2
ELSE IF (M.LT.0) THEN
  INFO = 3
ELSE IF (N.LT.0) THEN
  INFO = 4
ELSE IF (K.LT.0) THEN
  INFO = 5
ELSE IF (LDA.LT.MAX(1,NROWA)) THEN
  INFO = 8
ELSE IF (LDB.LT.MAX(1,NROWB)) THEN
  INFO = 10
ELSE IF (LDC.LT.MAX(1,M)) THEN
  INFO = 13
END IF
IF (INFO.NE.0) THEN
  CALL XERBLA('SGEMM ','INFO)
  RETURN
END IF

Quick return if possible.

IF ((M.EQ.0) .OR. (N.EQ.0) .OR. (((ALPHA.EQ.ZERO).OR. (K.EQ.0)).AND. (BETA.EQ.ONE))) RETURN

Figure 4.13: Pre-condition checks for subroutine sgemm.f.

For instance, the matrix dimensions are kept as attributes in class Matrix in object-oriented programming, and so it is possible that the user may pass input matrices with invalid matrix orders, such as the number of columns of $A$ does not equal the number of rows of $B$ in $A \times B$. In a traditional library, such as BLAS, a matrix is defined as an array and thus it is up to the user to tell the subroutine what the matrix orders are. The subroutine designer defines only one value to store both the number of columns of $A$ and the number of rows of $B$. Therefore, this check is unneeded in BLAS.
CONJA = LSAME(TRANSA,'C')
CONJB = LSAME(TRANSB,'C')

* Test the input parameters.
* INFO = 0
  IF ((.NOT.NOTA) .AND. (.NOT.CONJA) .AND. + (.NOT.LSAME(TRANSA,'T'))) THEN
    INFO = 1
  ELSE IF ((.NOT.NOTB) .AND. (.NOT.CONJB) .AND. + (.NOT.LSAME(TRANSB,'T'))) THEN
    INFO = 2
  ELSE IF (M.LT.0) THEN
    INFO = 3
  ELSE IF (N.LT.0) THEN
    INFO = 4
  ELSE IF (K.LT.0) THEN
    INFO = 5
  ELSE IF (LDA.LT.MAX(1,NROWA)) THEN
    INFO = 8
  ELSE IF (LDB.LT.MAX(1,NROWB)) THEN
    INFO = 10
  ELSE IF (LDC.LT.MAX(1,M)) THEN
    INFO = 13
  END IF
  IF (INFO.NE.0) THEN
    CALL XERBLA('CGEMM ',INFO)
    RETURN
  END IF
*
* Quick return if possible.
* IF ((M.EQ.0) .OR. (N.EQ.0) .OR. + (((ALPHA.EQ.ZERO).OR. (K.EQ.0)).AND. (BETA.EQ.ONE))) RETURN

Figure 4.14: Pre-condition checks for subroutine cgemm.f.

4.3 Case Study: Matrix Copying

In the design of a NLA library, a new returned matrix generated by a matrix operation is not always necessary. Therefore, for efficiency reasons, an in-place algorithm is often used, namely, assigning an input matrix to be the output matrix. However, for certain matrix operations, such as matrix multiplication, where the elements of a matrix need to be reused while performing the operation,\(^3\)

\(^3\)More precisely, when an interface of a matrix operation is designed to take an argument as both input and output, during the execution, if an element needs to be read after being written
* Test the input parameters.
* 
INFO = 0
IF ((.NOT.LSAME(SIDE,'L')) .AND. (.NOT.LSAME(SIDE,'R'))) THEN
  INFO = 1
ELSE IF ((.NOT.UPPER) .AND. (.NOT.LSAME(UPLO,'L'))) THEN
  INFO = 2
ELSE IF (M.LT.0) THEN
  INFO = 3
ELSE IF (N.LT.0) THEN
  INFO = 4
ELSE IF (LDA.LT.MAX(1,NROWA)) THEN
  INFO = 7
ELSE IF (LDB.LT.MAX(1,M)) THEN
  INFO = 9
ELSE IF (LDC.LT.MAX(1,M)) THEN
  INFO = 12
END IF
IF (INFO.NE.0) THEN
  CALL XERBLA('SSYMM ',INFO)
  RETURN
END IF
*
* Quick return if possible.
*
IF ((M.EQ.0) .OR. (N.EQ.0) .OR.
  + ((ALPHA.EQ.ZERO).AND. (BETA.EQ.ONE))) RETURN

Figure 4.15: Pre-condition checks for subroutine \texttt{ssymm.f}.

copying of the matrix must be applied first to ensure the correctness of this operation. Moreover, matrix copying is common in matrix operation optimisation. For instance, in order to make better use of the cache memory on a machine, the optimised BLAS, such as \textit{ Automatically Tuned Linear Algebra Software} (ATLAS) [112], implement matrix multiplication by firstly copying the input matrices into block-major format.\footnote{Strictly speaking, this copy is performed if the problem is large enough to tolerate copying the input matrices [111]. This is because, if the matrix is too small, the \(O(N^2)\) overhead from matrix copy may dominate the algorithm cost.}

Matrix copying is also a kind of crosscutting concern. The secondary concern of this method (copying the matrix) is entangled with the primary concern (performing the matrix operation). AOP can be utilized here to enable the two concerns to be separated, with each concern focusing on its own role. Code fragments showing how to handle matrix copy in JOOLALA with AspectJ are and changing the sequence of execution cannot avoid this, a new copy of the matrix is needed.
presented in Figure 4.16. Two pieces of advices are used here. The first is used to make a copy of the matrix before the matrix operation method is executed. The other bypasses the original operation, operating on the copy rather than the matrix itself.

```
public aspect DealWithSelfOperation {
    ...
    pointcut matrixSelfOpWithTwo(Matrix a, Matrix b, Matrix theMa) :
        execution(void !Property+.*(Matrix, Matrix, ..)) &&
        args(a, b) &&
        this(theMa);

    pointcut matrixSelfOpWithTwoPartTwo(Matrix a, Matrix b, Matrix theMa) :
        call(void Property.*(Matrix, Matrix, ..)) &&
        args(a, b, ..) &&
        this(theMa);

    before (Matrix a, Matrix b, Matrix theMa) throws OolalaException :
        matrixSelfOpWithTwo(a, b, theMa)
    {
        if (a==theMa||b==theMa) {
            theMa.cp = theMa.copy();
        }
    }

    void around(Matrix a, Matrix b, Matrix theMa) :
        matrixSelfOpWithTwoPartTwo(a, b, theMa)
    {
        if (a==theMa&&b==theMa)
            proceed(theMa.cp, theMa.cp, theMa);
        else if (a==theMa&&b!=theMa)
            proceed(theMa.cp, b, theMa);
        else if (a!=theMa&&b==theMa)
            proceed(theMa.cp, a, theMa);
        else
            proceed(a, b, theMa);
    }
    ...
}
```

Figure 4.16: Code fragment of aspect DealWithSelfOperation.

### 4.4 Summary of Chapter

In conclusion, this chapter has presented three types of crosscutting concerns in building a NLA library. Their appearances in source code and possible solutions
using AOP are given from Sections 4.1 to 4.3. In fact, the three types of crosscutting concerns described above are representative of three crosscutting concern sorts in NLA, namely, consistent behaviour, contract enforcement and entangled roles. The intent of consistent behaviour is to implement consistent behaviour in a number of methods that can be captured by a natural pointcut. Contract enforcement implies complying with design by contract rules. Entangled roles is the situation where a method has two roles, in which the secondary role is entangled with the primary role.

**Grouping Matrix Operations** A NLA library uses nested for-loops with different arguments (specifying property construction) and operations (specifying elements assignment) in many places to implement matrix operations. Using the aspect mechanism, one can use two pairs of pointcuts and associated advice to handle property construction and element operation in a uniform way. The matrix operations case study corresponds to consistent behaviour and entangled roles.

**Pre-Condition Checks** The original approach is to call a method which performs the pre-condition check for a certain matrix operation. The aspect mechanism provides a way in which a pointcut selects the operations that need the same check and a before advice performs the check. The pre-condition checks case study is representative of contract enforcement.

**Matrix Copying** Many NLA libraries implement a method/function that deals with more than one concern, e.g., a method handles both matrix copying and the matrix operation. With AOP, one can select the satisfied matrix operations, using before and around advices to handle copying of the matrix. Two crosscutting concern sorts, namely consistent behaviour and entangled roles are involved.
Chapter 5

Band-based Matrix Computation

“It often pays to reflect the structure of data in the program that processes it. When the data is an array with many zero elements, the time and space savings can be considerable.” [33]

Band matrix operation is clearly a case to which the above quotation applies. Although the mathematical definitions of many band matrix operations are straightforward, from the programming point of view, the attained performance and efficiency may vary considerably. Because any number plus or times zero is known, only the non-zero elements of a matrix need be operated on. The Numerical Linear Algebra (NLA) community has developed a large number of subroutines for performing matrix operations, usually by exploiting the structure of non-zero elements of multiple input matrices. As a result of combining the input matrices with different non-zero structures, the total number of implementations is huge. The work presented in this chapter shows that the algorithm for band matrix multiplication may replace some others, which deal with special band matrices, such as dense and triangular matrices, for software engineering benefits without damaging the performance.

Section 5.1 introduces the concept of a band matrix, explaining why there are different implementations for band matrix multiplication in terms of computational costs. Section 5.2 shows how to design an algorithm for band matrix multiplication that is optimised by iteration space partitioning, the implementation details of which are described in Sections 5.3 and 5.4. A further performance improvement is proposed in Section 5.5 by reusing the existing high performance BLAS subroutines.
5.1 Introduction

A band matrix is a matrix in which the non-zero elements are located around the major diagonal. Formally, given a \( m \times n \) matrix, for all \( i \) and \( j \), \( a_{ij} = 0 \), if \( (i - j) > ml \) or \( (j - i) > mu \). The quantities \( mu \) and \( ml \) are the upper and lower bandwidths, respectively. A rectangular general band matrix with \( m > n \) is illustrated in Figure 5.1, where \( mu = p - 1 \) and \( ml = q - 1 \).

\[
A = \begin{bmatrix}
    a_{11} & a_{12} & a_{13} & \cdots & a_{1p} & 0 & \cdots & 0 \\
    a_{21} & a_{22} & a_{23} & \cdots & 0 & \cdots & \cdots \\
    a_{31} & a_{32} & a_{33} & \cdots & 0 & \cdots & \cdots \\
    \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \cdots \\
    \vdots & \vdots & \vdots & \ddots & 0 & \ddots & \cdots \\
    0 & \cdots & \cdots & \cdots & a_{mn} \\
    \vdots & \vdots & \vdots & \cdots & \vdots & \cdots & 0 \\
    \vdots & \vdots & \vdots & \cdots & \vdots & \cdots & 0 \\
    0 & \cdots & \cdots & \cdots & 0 & \cdots & a_{ml} \\
\end{bmatrix}
\]

Figure 5.1: A rectangular band matrix with \( m > n \) [2].

There are special issues for band matrix arithmetic. For example, addition and subtraction operations on two band matrices with lower (or upper) bandwidths \( b1 \) and \( b2 \) yield a band matrix with bandwidth \( max(b1, b2) \). Multiplication of the two matrices yields a matrix with bandwidth \( (b1 + b2) \), provided that the result matrix bandwidth does not exceed the order of the corresponding result matrix. Cholesky factorisation on a symmetric positive definite band matrix with bandwidth \( b \) generates two matrices: a lower band matrix with bandwidth \( b \), and a upper band matrix which is the transpose of that lower band matrix.\(^1\)

This section reviews how matrix operations that involve band matrices are implemented and used in NLA (Section 5.1.1), explaining why different implementations were obtained for band matrix operations by giving a typical example: band matrix multiplication (Section 5.1.2).

\(^1\)The operations on these matrix operands are assumed to be mathematically valid.
5.1.1 Problem Overview

A number of widely-used matrix types, such as upper triangular matrix, dense matrix and diagonal matrix can be considered as special cases of a band matrix. For instance, an $m \times n$ upper triangular matrix is a band matrix with $mu = n - 1$ and $ml = 0$; an $m \times n$ dense matrix is also a band matrix, in this case with $mu = n - 1$ and $ml = m - 1$. Therefore, the algorithm for a band matrix operation is a general algorithm that is applicable to a range of different kinds of band matrix. However, many linear algebra libraries implement subroutines for computing each special case of band matrix, resulting in implementing a number of subroutines that are not essentially necessary.

For example, the de facto programming interface for basic linear algebra operations, the BLAS (Basic Linear Algebra Subprograms) [1], provides subroutines xGEMM and xTRMM for dense-dense and dense-triangular matrix multiplication, respectively. LAPACK [12] supports two distinct subroutines, namely, xPFTRF and xPBTRF, for Cholesky factorisation of dense and banded matrices. The number of required implementations can be significantly reduced since all the different kinds of band matrix with different bandwidths for an operation can be merged into a single implementation. Moreover, by supporting band matrix operations, the library is able to handle more kinds of matrices, i.e., any matrix that is mathematically banded is supported. This allows the libraries to become extensible to future functionality requirements.

Furthermore, [110] observes that band matrix multiplication is widely used in the development of the Invariant Subspace Decomposition Algorithm (ISDA), which is used to compute the eigenvalues and eigenvectors of matrices. ISDA makes heavy use of matrix multiplication, in order to factorise the matrix into smaller problems. [24] describes a way to perform periodic reduction to band forms, which allows most of the multiplication to involve only band matrices. However, the xGEMM subroutine in the BLAS cannot efficiently exploit all the zero-involved computations, suggesting that a subroutine for multiplying general band matrices would be advantageous.

5.1.2 Distinct Computational Costs

Although using a band matrix implementation of a matrix operation for all the special cases reduces the total number of implementations, the performance,
which is a core concern in scientific computing, must not be compromised. In other words, the performance of the generalised band matrix algorithm for a matrix operation needs to be comparable with that of the specially designed ones.

In computational complexity theory, theoretical estimates are provided for the resources (such as time, space or energy) needed by an algorithm. The time complexity calculates the number of time-critical operations as a function of the size of the input to the algorithm. This is often used as a metric to compare algorithms for a typical matrix operation.

Given \( C = A \times B \), if \( A \) and \( B \) are square dense matrices \(^2\) and all three matrices are stored in dense format, the equation may be given as:

\[
\begin{bmatrix}
    c_{11} & c_{12} & \ldots & c_{1n} \\
    c_{21} & c_{22} & \ldots & c_{2n} \\
    \vdots & \vdots & \ddots & \vdots \\
    c_{n1} & c_{n2} & \ldots & c_{nn}
\end{bmatrix} =
\begin{bmatrix}
    a_{11} & a_{12} & \ldots & a_{1n} \\
    a_{21} & a_{22} & \ldots & a_{2n} \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{n1} & a_{n2} & \ldots & a_{nn}
\end{bmatrix}
\begin{bmatrix}
    b_{11} & b_{12} & \ldots & b_{1n} \\
    b_{21} & b_{22} & \ldots & b_{2n} \\
    \vdots & \vdots & \ddots & \vdots \\
    b_{n1} & b_{n2} & \ldots & b_{nn}
\end{bmatrix}
\]

Therefore, for \( i = 1, 2, \ldots, n \) and \( j = 1, 2, \ldots, n \),

\[
c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.
\]

An algorithm for matrix-matrix multiplication for dense matrices is expressed in Figure 5.2. Because each update of \( c_{ij} \) requires two floating point operations (one addition and one multiplication), the computational cost of this algorithm is \( 2n^3 \) floating point operations.

```plaintext
DO j = 1 to n
    DO i = 1 to n
        DO k = 1 to n
            c(i,j) = c(i,j) + a(i,k) * b(k,j)
        END DO
    END DO
END DO
```

Figure 5.2: An algorithm for matrix-matrix multiplication \( C = A \times B \) where both \( A \) and \( B \) are dense.

\(^2\)The input matrices do not need to be square. Square matrix is used as an example to show the computational costs for matrix with different bandwidths.
If $A$ is an upper triangular matrix and $B$ is a dense matrix, then matrix-matrix multiplication becomes:

For $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$,

$$c_{ij} = \sum_{k=i}^{n} a_{ik} b_{kj}.$$  

In this case, the innermost loop bounds are changed as shown in Figure 5.3.\(^3\)

```
DO j = 1 to n
    DO i = 1 to n
        DO k = i to n
            c(i,j) = c(i,j) + a(i,k) * b(k,j)
        END DO
    END DO
END DO
```

Figure 5.3: An algorithm for matrix-matrix multiplication $C = A \times B$ where $A$ is upper triangular and $B$ is dense.

Accordingly, the computational cost in floating point operations is

$$2(n^2 + n(n - 1) + n(n - 2) + \ldots + n)$$

$$= 2n(1 + 2 + 3 + \ldots + n)$$

$$= n^2(n + 1)$$

$$= n^3 + n^2.$$

If $A$ is upper triangular, whereas $B$ is lower triangular, although $C$ is still dense, fewer computations need to be performed.

For $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$,

$$c_{ij} = \sum_{k=max(i,j)}^{n} a_{ik} b_{kj}.$$  

The algorithm is given in Figure 5.4.

This algorithm contains a maximum function in the lower bound of the $k$ loop.

\(^3\)Note that a matrix element such as $c(i, j)$ refers to the element at row $i$ and column $j$ of matrix $C$. It does not imply that the matrix is stored in dense format. The storage format of each matrix is not discussed in these examples.
DO j = 1 to n
  DO i = 1 to n
    DO k = max(i,j) to n
      c(i,j) = c(i,j) + a(i,k) * b(k,j)
    END DO
  END DO
END DO

Figure 5.4: An algorithm for matrix-matrix multiplication $C = A \times B$ where $A$ is upper triangular and $B$ is lower triangular.

It is easier to calculate the number of floating point operations if the original loop nest is divided into two parts, as shown in Figure 5.5.

DO j = 1 to n
  DO i = j to n
    DO k = i to n
      c(i,j) = c(i,j) + a(i,k) * b(k,j)
    END DO
  END DO
END DO

DO j = 1 to n
  DO i = 1 to j-1
    DO k = j to n
      c(i,j) = c(i,j) + a(i,k) * b(k,j)
    END DO
  END DO
END DO

Figure 5.5: The transformed algorithm for matrix-matrix multiplication $C = A \times B$ where $A$ is upper triangular and $B$ is lower triangular.

The computational cost can be calculated as the sum of the floating point operations of the two parts. The cost in the number of floating point operations of the first part is:
2((n + n − 1 + n − 2 + . . . + 1) + (n − 1 + n − 2 + . . . + 1) + . . . + 1) \\
= n(n + 1) + n(n − 1) + (n − 1)(n − 2) + . . . + 2 \times 1 \\
= 1^2 + 1 + 2^2 + 2 + 3^3 + 3 + . . . + n^2 + n \\
= \frac{n(n + 1)(n + 2)}{3} \\
= \frac{1}{3}n^3 + \frac{2}{3}n,

and the cost of the second part is

\[ 2(n − 1 + 2(n − 2) + 3(n − 3) + . . . + (n − 1)(n − (n − 1))) \]
\[ = 2(n + 2n + 3n + . . . + (n − 1)n − (1^2 + 2^2 + 3^3 + . . . + (n − 1)^2)) \]
\[ = n^2(n − 1) − \frac{n(n − 1)(2n − 1)}{3} \]
\[ = \frac{n(n − 1)(n + 1)}{3} \]
\[ = \frac{1}{3}n^3 − \frac{1}{3}n. \]

The total computational cost is therefore:

\[ \frac{1}{3}n^3 + n^2 + \frac{2}{3}n + \left(\frac{1}{3}n^3 − \frac{1}{3}n\right) \]
\[ = \frac{2}{3}n^3 + n^2 + \frac{1}{3}n. \]

If both \( A \) and \( B \) are upper triangular matrices, then the result matrix \( C \) is also upper triangular and is given by:

For \( j = 1, 2, . . . , n \) and \( i = 1, 2, . . . , j \),

\[ c_{ij} = \sum_{k=i}^{j} a_{ik}b_{kj}. \]

This algorithm for upper triangular matrix-matrix multiplication is given in
Figure 5.6.

\[
\text{DO } j = 1 \text{ to } n \\
\text{DO } i = 1 \text{ to } j \\
\text{DO } k = i \text{ to } j \\
c(i,j) = c(i,j) + a(i,k) \times b(k,j) \\
\text{END DO} \\
\text{END DO} \\
\text{END DO}
\]

Figure 5.6: An algorithm for matrix-matrix multiplication \( C = A \times B \) where both \( A \) and \( B \) are upper triangular.

The number of floating point operations is calculated as:

\[
2 \times (1 + (2 + 1) + (3 + 2 + 1) + \ldots + (n + n - 1 + \ldots + 1)) \\
= 1 \times 2 + 2 \times 3 + \ldots + n(n+1) \\
= \frac{n(n+1)(n+2)}{3} \\
= \frac{1}{3} n^3 + \frac{2}{3} n.
\]

If the bandwidths of both \( A \) and \( B \) are one, i.e., both are diagonal matrices, the result matrix \( C \) is also diagonal and can be expressed by:

For \( i = 1, 2, \ldots, n \),

\[ c_{ii} = a_{ii} b_{ii}. \]

The computational cost becomes \( n \) floating point operations in this extreme case.

Note that as well as reducing the number of floating point operations for matrices with smaller bandwidths, the above algorithms also reduce the number of necessary memory accesses.

5.1.3 Summary

This section has introduced the basis of the band property, current implementations for band matrix operations, how the current approaches can be improved and the possible difficulties. Section 5.1.1 presents the software engineering issues in traditional NLA libraries such as BLAS and LAPACK, i.e., designing
subroutines for a matrix operation for the special cases of a band matrix, and proposes generalised subroutines to replace them. Section 5.1.2 illustrates that the computational cost of a band matrix operation is significantly affected by its bandwidths, and concludes that the generalised banded implementation must run as fast as each specialised implementation for each operation in order not to compromise the performance. As a result, a straightforward solution to efficient band matrix operations would be to exclude all the zero-involved computations, i.e., the computations over matrix elements outside the boundary specified by the bandwidth. Typically, these computations can be divided into two groups:

- All the accesses to an output matrix element that is known to be zero; and
- Computations that involve calculations with zero on an element of the input matrix.

The implementations of such band matrix operations eliminate unnecessary floating point operations involving zeros, usually by inserting conditions (max/min or if) on loop indices. By doing this, the elements involving computations on zeros are not computed, by skipping the indices to those elements. How to obtain an algorithm for a band matrix operation is described in Section 5.2.

5.2 The Design of Algorithms for Band Matrix Operations

The algorithms for band matrix operations are based on the corresponding dense matrix operations. As any dense matrix itself is a special case of a band matrix, the algorithm for any dense matrix operation can be used for computing that operation for band matrices, despite the performance penalties due to the computations on zero elements.

5.2.1 Matrix-Matrix Multiplication

According to Section 5.1.3, two kinds of computation must be eliminated. For example, in matrix-matrix multiplication, for $C \ (m\text{-by-}n) = A \ (m\text{-by-}q) \times B \ (q\text{-by-}n)$, where $mua$ and $mla$ are the upper and lower bandwidths of matrix $A$; $mub$ and $mlb$ are the upper and lower bandwidths of matrix $B$; $mu$ and $ml$ are the upper and lower bandwidths of matrix $C$, then
• $c_{ij}$ is known to be zero, i.e. $c_{ij} = 0$, if $i - j > ml$ or $i - j < -mu$, and

• the non-zero elements $c_{ij}$ of $C$ are calculated by $a_{ik}$ and $b_{kj}$, some of which are known to be zero.

Using matrix-matrix multiplication as an example, the pseudo code for dense multiplication below

\[
\begin{align*}
\text{DO } & j = 1, n \\
\text{DO } & i = 1, m \\
\text{DO } & k = 1, q \\
& c(i,j) = c(i,j) + a(i,k) \times b(k,j) \\
\text{END DO} \\
\text{END DO} \\
\text{END DO}
\end{align*}
\]

is firstly transformed to:

\[
\begin{align*}
\text{DO } & j = 1, n \\
\text{DO } & i = 1, m \\
\text{IF } & c(i,j) \text{ is not zero} \\
\text{DO } & k = 1, q \\
\text{IF } & \text{neither } a(i,k) \text{ nor } b(k,j) \text{ is zero} \\
& c(i,j) = c(i,j) + a(i,k) \times b(k,j) \\
\text{END IF} \\
\text{END DO} \\
\text{END IF} \\
\text{END DO} \\
\text{END DO}
\end{align*}
\]

Since the band property specifies that any matrix element above the upper bandwidth or below the lower bandwidth is zero, the code above can be modified to:
DO j = 1, n
   DO i = 1, m
      IF -mu <= i-j <= ml
         DO k = 1, q
            IF neither a(i,k) nor b(k,j) is zero
               c(i,j) = c(i,j) + a(i,k) * b(k,j)
            END IF
         END DO
      END IF
   END DO
END DO

where, \( m_u \) and \( m_l \) are the upper and lower bandwidths of the output matrix \( C \), respectively.

![Figure 5.7: The start values in A and B.](image)

The elements in \( C \) are calculated as each row in \( A \) times each column in \( B \). Figure 5.7 shows that the start value of all the rows in \( A \) is always the larger of 1 and \( i - m_la \) and the start value of all the columns in \( B \) is the larger of 1 and \( j - m_ub \), where \( m_la \) is the lower bandwidth of \( A \) and \( m_ub \) is the upper bandwidth of \( B \). Values of \( k \) in the innermost loop below the maximum of these three values (1, \( i - m_la \) and \( j - m_ub \)) will access zero elements which do not change the value in \( c(i, j) \) and so do not need to be executed. The end values for \( k \) can be calculated in a similar way, so that the code can be further transformed, as shown in the following:
DO j = 1, n
  DO i = 1, m
    IF -mu <= i-j <= ml
      DO k = max(1,i-mla,j-mub), min(q,i+mua,j+mlb)
        c(i,j) = c(i,j) + a(i,k) * b(k,j)
      END DO
    END IF
  END DO
END DO

Note that the IF statement can be removed by adding max/min conditions in its outer loop, as shown in Figure 5.8. The iteration space will not be changed.

DO j = 1, n
  DO i = max(1,j-mu), min(m,j+ml)
    DO k = max(1,i-mla,j-mub), min(q,i+mua,j+mlb)
      c(i,j) = c(i,j) + a(i,k) * b(k,j)
    END DO
  END DO
END DO

Figure 5.8: Band matrix-matrix multiplication with min/max conditions.

Eventually, the equation for general rectangular band matrix multiplication can be written as shown in Figure 5.9.

For \( j = 1, 2, \ldots, n \) and \( i = \max(1, j-mu), \max(1, j-mu)+1, \ldots, \min(m, j+ml), \)\n\[
c_{ij} = \sum_{k=\max(1,i-mla,j-mub)}^{\min(q,i+mua,j+mlb)} a_{ik}b_{kj}
\]

Figure 5.9: An algorithm for matrix-matrix multiplication on band matrices.

The iteration space of general band matrix-matrix multiplication gives a polyhedron in three dimensions, as shown in Figure 5.10. In this figure, given \( C = A \times B \), where \( A \) is a 13 \( \times \) 12 matrix with upper bandwidth 4 and lower bandwidth 5, \( B \) is a 12 \( \times \) 11 matrix with upper bandwidth 4 and lower bandwidth 2, and \( C \) is therefore a 13 \( \times \) 11 matrix with upper bandwidth 8 and lower bandwidth 7.
5.2.2 Matrix Addition

Similarly, other matrix operations, such as matrix addition for dense matrices,\textsuperscript{4} can be utilised to design a band matrix addition. As described in Section 5.1, \(mu = \max(mua, mub)\) and \(ml = \max(mla, mlb)\). A simple algorithm for matrix addition is shown below:

\textsuperscript{4}[5] defines a updated set of BLAS, including matrix norms, diagonal scaling, matrix addition and scale, etc.
DO \ j = 1 \ to \ n \\
\ DO \ i = 1 \ to \ m \\
\ \ IF \ c(i,j) \ is \ not \ zero \\
\ \ \ \ IF \ neither \ a(i,j) \ nor \ b(i,j) \ is \ zero \\
\ \ \ \ \ c(i,j) = a(i,j) + b(i,j) \\
\ \ ELSE \ IF \ a(i,j) \ is \ zero \\
\ \ \ \ \ c(i,j) = b(i,j) \\
\ \ ELSE \ IF \ b(i,j) \ is \ zero \\
\ \ \ \ \ c(i,j) = a(i,j) \\
\ \ \ END \ IF \\
\ \ END \ IF \\
\ END \ DO \\
END \ DO \\

This can be transformed to:

DO \ j = 1 \ to \ n \\
\ DO \ i = 1 \ to \ m \\
\ \ IF \ -\mu <= i-j <= \mu \\
\ \ \ \ IF \ -\min(mua, mub) <= i-j <= \min(mla, mlb) \\
\ \ \ \ \ c(i,j) = a(i,j) + b(i,j) \\
\ \ ELSE \ IF \ -\mu <= i-j < -mua \\
\ \ \ \ \ c(i,j) = b(i,j) \\
\ \ ELSE \ IF \ -\mu <= i-j < -mub \\
\ \ \ \ \ c(i,j) = a(i,j) \\
\ \ ELSE \ IF \ mla < i-j <= ml \\
\ \ \ \ \ c(i,j) = b(i,j) \\
\ \ ELSE \ IF \ mlb < i-j <= ml \\
\ \ \ \ \ c(i,j) = a(i,j) \\
\ \ \ END \ IF \\
\ \ END \ IF \\
\ END \ DO \\
END \ DO
5.2.3 Analysis of the Algorithm for Band Matrix

On the one hand, the computational cost of a band matrix algorithm remains the same (counting the number of floating point operations only), compared with a specialised algorithm designed for other cases such as dense matrix multiplication. However, on the other hand, a large number of integer instructions are introduced in the band matrix algorithms. The minimum and maximum conditions involve integer instructions for comparing two or more integers. These integer instructions need to be executed each time in the loop nest whenever the loop index changes so that the costs of these computations may be non-trivial, or even expensive for a matrix with sufficiently small bandwidth.

To reduce the total number of instructions to be executed, it is suggested to group matrix elements that satisfy the same conditions into components and perform the computations on these components. A methodology, namely, *Iteration Space Partitioning* (ISP) [22], is introduced in the following sections for reducing the number of instructions in the implementations of band matrix operations. After this optimisation process, the number of instructions is significantly reduced from performing checks on each matrix element to performing checks on grouped components only.\(^5\)

5.3 Transformation Issues

It is evident that the technique introduced in Section 3.3.4 can be applied to simplify the loop bounds in the above algorithms. We concentrate on the statement in the innermost loop only and hence there is only one polyhedron that needs to be scanned. Two issues related to the implementation of a source-to-source transformer are discussed in this section. The first issue is about how to slightly modify the elimination of compound bounds, and the second issue describes the sequence of performing the four kinds of loop bound simplifications.

\(^5\)This approach eliminates instructions that determine whether each matrix element needs to be involved in the next computation by introducing instructions that determine the execution on each component. The latter needs a much smaller number of instructions than the former for most cases, except where the matrix size is extremely small.
5.3.1 Modification of Compound Bound Elimination

[79] describes how a compound bound, for example, upper bound $\min(e_1, e_2, \ldots, e_n)$ can be isolated into two parts: $e_i$ and $\min(e_1, e_2, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n)$. That is, the iteration space is partitioned into two parts, where the complexity of evaluating the first part becomes one simple bound and the complexity of evaluating the second part is reduced to $n - 1$ simple bounds. This transformation is based on assuming that the compound inequality $e_i \leq \min(e_1, e_2, \ldots, e_n)$ is either true or false.

However, during implementation of this transformation, it was noticed that, with two or more simple bounds, this method transforms the original code by passing a condition that contains compound bounds, making the elimination of variables in the system of inequalities more difficult. For example, Figure 5.11 gives a loop nest in which $i$, $j$ and $k$ are the variables to be eliminated.

![Figure 5.11: A loop nest example.](image)

We concentrate on the head of the loop nest only. Suppose we want to simplify the lower bound of the $i$ loop, then the original code can be transformed to two blocks of code:

```
DO j = 1 to n
  DO l = max(1, j-mub) to min(k, j+mlb)
    DO i = max(1, j-mu, l-mua) to min(m, j+ml, l+mla)
      ...
      CONTINUE
    CONTINUE
  CONTINUE
CONTINUE
```

```
//with inequality 1>=max(j-mu, l-mua)
```

*A compound inequality* is an inequality which contains compound bounds; whereas a *simple inequality* is an inequality which contains only simple bounds.
CHAPTER 5. BAND-BASED MATRIX COMPUTATION

DO j = 1 to n
  DO l = max(1, j-mub) to min(k, j+mlb)
    DO i = max(j-mu, l-mua) to min(m, j+ml, l+mla)
      ...
      CONTINUE
      CONTINUE
      CONTINUE
  //with inequality 1<=max(j-mu, l-mua)-1

A problem arises when trying to add the inequality 1 <= max(j - mu, l - mua) - 1 to the second code block. This inequality cannot be easily simplified so that the involved variables j and l cannot be further tested. The method of simplifying a compound loop bound is thus modified to assume that a simple inequality is either true or false. By doing this, the complexity of evaluating both of the two parts is reduced to n - 1. For example, by passing 1 <= j - mu is either true or false, the original code is transformed as shown in Figure 5.12.

DO j = 1 to n
  DO l = max(1, j-mub) to min(k, j+mlb)
    DO i = max(1, l-mua) to min(m, j+ml, l+mla)
      ...
      CONTINUE
      CONTINUE
  //with inequality 1<=j-mu

DO j = 1 to n
  DO l = max(1, j-mub) to min(k, j+mlb)
    DO i = max(1, l-mua) to min(m, j+ml, l+mla)
      ...
      CONTINUE
      CONTINUE
  //with inequality 1>=j-mu+1

Figure 5.12: Two nodes generated from the original loop nest.

The simple inequalities can be merged to the bound of a loop, as shown in Figure 5.13; or written as an IF condition outside the loop nest, if the simple inequality only involves constants. Further information is given in Section 5.3.2.
DO j = max(1, mu+1) to n
    DO l = max(1, j-mub) to min(k, j+mlb)
        DO i = max(j-mu, l-mua) to min(m, j+ml, l+mla)
            ...
        CONTINUE
    CONTINUE
CONTINUE

DO j = 1 to min(mu, n)
    DO l = max(1, j-mub) to min(k, j+mlb)
        DO i = max(1, l-mua) to min(m, j+ml, l+mla)
            ...
    CONTINUE
CONTINUE
CONTINUE

Figure 5.13: Two nodes generated from the original loop nest after merging inequalities.

5.3.2 Implementation of Simplifying the Four Kinds of Loop Bounds

Recall from Section 3.3.4 that there are four distinct kinds of loop bounds, namely, empty loops, redundant bounds, partially empty loops and compound bounds. After analysing the characteristics of these distinct cases, it has been found that empty loops and partially empty loops can be implemented together, whereas redundant bounds and compound bounds can be eliminated at the same time. The sequence of the simplification is described below:

1. Perform the elimination of the compound bounds from the innermost loop to the outermost loop by giving an inequality that involves only two simple bounds of a compound bound that is either true or false. The code is therefore divided into a number of blocks in this step.

2. Every time all the compound loop bounds (both upper and lower) of a loop are fully eliminated, add an inequality that represents the elimination of the current loop variable for each block.

3. An inequality is inserted by finding a matched loop variable, from the next loop (the loop that is one level out from the current working loop) to the
outermost loop. Notice that a matched loop variable may not exist; in that case, an IF conditional statement is added, for determining whether the loop nest is going to be executed or not.

For example, the code below

```plaintext
DO i = 1 to min(50, 100)
  DO j = 1 to i
    S1
  END DO
END DO

DO i = 1 to 100
  DO j = 1 to min(i, 100 - i)
    S1
  END DO
END DO

DO i = max(1, 51) to 100
  DO j = 1 to 100 - i
    S1
  END DO
END DO
```

is divided into two blocks by determining whether \( i \leq 100 - i \), i.e., \( i \leq 50 \) is true or false. If one block is empty, this tells that an empty loop is derived by eliminating the redundant bound (we do not test whether a loop nest is empty or not so that we do not otherwise know this at this step). Then, by eliminating \( j \), two inequalities, \( 1 \leq i \) and \( 1 \leq 100 - i \), are added to the two blocks, respectively.

```plaintext
DO i = 1 to min(50, 100)
  DO j = 1 to i
    S1
  END DO
END DO

DO i = max(1, 51) to 100
  DO j = 1 to 100 - i
    S1
  END DO
END DO
```

```plaintext
DO i = max(1, 51) to 100
  DO j = 1 to 100 - i
    S1
  END DO
END DO
```

```plaintext
DO i = max(1, 51) to min(100, 99)
  DO j = 1 to 100 - i
    S1
  END DO
END DO
```
In the second block, the upper bound is $\min(100, 99)$, which means it is a partially empty loop. And if the lower bound is larger than the upper bound, or the added IF condition is always false, it tells that this loop nest is empty.

5.4 Implementation of a Source-to-Source Transformation

This section describes a Java implementation which transforms a source code with max/min conditions to its equivalent source code but without conditions, by applying the technique introduced in Section 3.3.4. Note that this implementation focuses on the transformation rather than locating and replacing the matching code.

5.4.1 Structure

The input source code consists of multiple levels of loops, in which each loop has a name and two bounds: an upper bound and a lower bound. According to Section 3.3.4, an admissible loop bound is either simple or compound. In this implementation, a simple bound is described as a sequence of variables, whereas a compound bound is either the minimum or the maximum value of multiple simple bounds. A variable contains two parts, the number part and the string part. For example, Figure 5.11 is a tree with only the root (single node tree). This node has three levels of loops, namely, loops $j$, $l$ and $i$, respectively. Both the lower and upper bounds of $j$ are simple bounds, whereas the bounds in $l$ and $i$ are compound bounds. A simple bound such as $j - mub$ is composed of two variables: $j$ and $-mub$. In $-mub$, the number part is $-1$ and the string part is $mub$.

In order to track the original source code, transformed source code and even the intermediate source code, multiple levels of loops are formed into a node and stored in a binary tree. All the nodes at the same level of the tree form an expression that is equivalent (traversing the same set of points) to that at any other level. An example of a tree with three levels and seven nodes is given in Figure 5.14. The top level is the original source code, whereas the transformed source code is given at the bottom level by a number of blocks. The intermediate codes are those at the middle level, which track the transformations from each
node downwards.

![Figure 5.14: An example of a tree showing how nodes are connected.](image)

All these components are defined by classes. A number of “has a” relationships are used to form the relationship between these classes. For example, a tree has a root node; a node has a left node, a right node and many loops; a loop has two compound bounds; a compound bound consists of multiple simple bounds and a boolean to define whether it is a lower or an upper bound, and a bound contains a sequence of variables. The class dependency diagram is given in Figure 5.15.

### 5.4.2 Execution Flow of the Transformation Program

Class Tree has a method called optimise, which is used to specify the simplification of a bound (either lower or upper) of a loop level in a node. When this method is called, the node specified is first duplicated to have two copies. The two copies are connected to the original node’s left and right side. A simple inequality is then chosen in the way described in Section 5.3.1 and the corresponding bounds of the two nodes are simplified from evaluating $n$ inequalities to evaluating $n - 1$ inequalities. Afterwards, the generated inequality is passed to the left node and the inverse of that inequality is passed to the right node so that more complexity is introduced to the outer loop bounds. The merge of a node and an inequality is described in the following:

1. Because each loop has a unique name (such as $i$, $j$), the inequality to be passed is reformed by method solve in class SimpleInequality so that only the variable with the loop name is on the left part of the inequality and the variable’s number part is one. Note that the left part may be empty if the inequality contains only constant values;

2. The inequality is tested from the innermost loop to the outermost loop of a node. If the loop name is contained in this inequality, it is transformed as described above; otherwise, test the next loop. According to whether the inequality is either $<=$ or $>$, the right side of the transformed inequality
is added as an additional simple bound to either the upper or the lower compound bound of that loop. If an inequality does not contain any loop names, i.e., only constant values are in this inequality, it can be pushed outside the loop nest as an IF statement.

For example, in Figure 5.12, testing the inequality \(1 \leq j - mu\) does not match loop \(i\) but does match loop \(j\) so that it is firstly transformed to \(j \geq mu + 1\). Then the right part \(mu + 1\) is added to the lower bound of loop \(j\), as shown in Figure 5.13.

Class Tree has several methods which are based on optimise, namely, optimiseLevelUpper, optimiseLevelLower, optimiseLevel, and optimiseTree. Method optimiseLevelUpper and optimiseLevelLower are implemented by
calling \texttt{optimise} repeatedly on a bound of a loop in a node, until the bound is completely optimised (it becomes a simple bound). Method \texttt{optimiseLevel} calls \texttt{optimiseLevelUpper} and \texttt{optimiseLevelUpper} one after the other. Which gets called first does not matter. After all the compound bounds at this level are transformed to simple bounds, an additional inequality that is generated by elimination of the current loop variable is added to each block, respectively. Eventually, by providing the root of a tree, method \texttt{optimiseTree} calls method \texttt{optimiseLevel} from the innermost loop bounds upwards, i.e., the method pushes all the compound bounds from the innermost loop bounds to the outermost loop bounds, resulting in a number of code blocks which are semantically equivalent to the original code. For example, for eliminating all the max/min/if conditions, the code fragment in Figure 5.11, which is the loop nest of the band matrix multiplication algorithm,\footnote{The code fragments in Figure 5.11 has the same iteration space as in Figure 5.8, which is an algorithm for band matrix multiplication. One can be derived from the other by loop interchanging. The code fragment in Figure 5.8 is easier to understand the iteration space so that it was used for describing the newly designed algorithm, whereas the code fragment in Figures 5.11 is corresponding to the loop nests that is in the implementation of \texttt{DGEMM} in BLAS.} is divided into 384 blocks, each of which only has compound bounds that are composed of constants given as inputs at the outermost loop, i.e., it is able to determine whether or not a block is executed at an early stage of execution.

### 5.5 Optimised Band Matrix-Matrix Multiplication

It is obvious that an efficient band matrix multiplication needs to be designed carefully and requires assembly level programming, by exploiting the band property and the advanced computer architecture. This work can be costly and time-consuming. The ISP transformation described in earlier sections not only eliminates unnecessary instructions, but also prepares the code to reuse existing fully optimised subroutines, for implementing an efficient band matrix multiplication with limited effort.

If a NLA library needs to be optimised, loop tiling \cite{113}, as introduced in Section 3.3.3, needs to be applied in order to maximise cache reuse. As discussed in Section 5.1.2, the bandwidths of the matrices significantly affect the computational costs. In addition, dividing band matrices into a number of same sized
submatrices by loop tiling, the number of data entries to be accessed for each sub-
matrix can be greatly different. In this case, the same submatrix size does not
imply the same number of data entries need to be fitted into the cache. Therefore,
applying loop tiling directly to a band matrix operation is not necessarily an op-
timised solution. However, after eliminating the max/min/if functions, the band
matrix multiplication algorithm is automatically divided into a fixed number of
blocks, each of which typically handles a subspace of the three dimensional space
shown in Figure 5.10.

This section attempts to further optimise the 384 blocks, which are derived
from the algorithm of band matrix multiplication. How the calculations within a
subspace can be performed by calling the existing BLAS subroutine(s) is described
with examples in Section 5.5.1. Section 5.5.2 shows that the subspaces that are
the multiplication of two triangular matrices, the implementation of which is not
provides in BLAS, can also be handled.

5.5.1 Subspace of Dense-Dense and Dense-Triangular Ma-
trix Multiplications

```
DO 1059 J = max(-ml+M, -mlb+K), mub
   DO 1058 L = -mla+M, mua
      DO 1057 I = 1, M
         C(I,J) = C(I,J) + B(L,J)*A(I,L)
      1057 CONTINUE
   1058 CONTINUE
1059 CONTINUE
```

Figure 5.16: A block example iterating a subspace of the polyhedron.

This section makes an attempt to match the calculations within a subspace to
one or more level 3 BLAS subroutines. The most obvious match is the dense-dense

---

8 In order to avoid confusion, “submatrices” here refers to the blocks divided by loop tiling,
whereas “blocks” refers to the blocks generated by removing max/min/if functions.

9 Although the total number of blocks is not that small (384), some loop nests are always
empty as the lower bound is larger than the upper bound of the outermost loop. Many others
may also be empty, depending on the bandwidths and orders of the input matrices. The actual
number of blocks that is going to be executed for a certain case is much smaller than 384.
However, because whether to execute the statements inside a for loop or not is determined at
the outmost for loop or the outmost if condition, the performance savings would be trivial even
if we reduce the number of blocks by runtime code generation.
matrix multiplication. For example, a generated block after applying iteration space partitioning is given in Figure 5.16.

This block typically computes a dense submatrix of $C$, as the multiplication of two dense submatrices of $A$ and $B$, respectively. Other blocks involve the multiplication of two triangular matrices, dense and triangular matrices, etc.

Those blocks that are the multiplication of two dense matrices are the easiest to identify, because the loop bounds of the triply nested loop are all constants; whereas others may not be. For example, one block of code is:

\[
\text{DO 1064 } J = \max(1, -k_1+M), \min(N, ku, ku2, -k_12-1+K, ku1-k_12) \\
\text{DO 1063 } L = -k_11+M, J+k_12 \\
\text{DO 1062 } I = 1, M \\
\quad C(I,J) = C(I,J) + B(L,J)\cdot A(I,L) \\
\text{1062 CONTINUE} \\
\text{1063 CONTINUE} \\
\text{1064 CONTINUE}
\]

It is not straightforward to see what this block of code is doing and thus further transformation is needed. By interchanging the $L$ and $I$ loops, the code is transformed to:

\[
\text{DO 1064 } J = \max(1, -k_1+M), \min(N, ku, ku2, -k_12-1+K, ku1-k_12) \\
\text{DO 1063 } I = 1, M \\
\text{DO 1062 } L = -k_11+M, J+k_12 \\
\quad C(I,J) = C(I,J) + B(L,J)\cdot A(I,L) \\
\text{1062 CONTINUE} \\
\text{1063 CONTINUE} \\
\text{1064 CONTINUE}
\]

After splitting the upper bound of the $L$ loop, the code can be transformed into two parts:

\[
\text{DO 1064 } J = \max(1, -k_1+M), \min(N, ku, ku2, -k_12-1+K, ku1-k_12) \\
\text{DO 1063 } I = 1, M \\
\text{DO 1062 } L = -k_11+M, k_12 \\
\quad C(I,J) = C(I,J) + B(L,J)\cdot A(I,L) \\
\text{1062 CONTINUE} \\
\text{1063 CONTINUE} \\
\text{1064 CONTINUE}
\]
DO 1064 J = max(1, -kl+M), min(N, ku, ku2, -k12-1+K, ku1-k12)
    DO 1063 I = 1, M
        DO 1062 L = 1+kl2, J+kl2
            C(I,J) = C(I,J) + B(L,J)*A(I,L)
        1062 CONTINUE
    1063 CONTINUE
1064 CONTINUE

where the first part is the multiplication of two dense matrices (call to ATLAS_xGEMM),
and the second part is the multiplication of a dense matrix and a triangular matrix
(call to ATLAS_xTRMM), if associated the loop nest is not empty.

5.5.2 Design of Triangular-Triangular Multiplication

Similar transformation can be performed for other blocks. Note that some blocks
may involve the multiplication of two triangular matrices or an upper with a
lower triangular matrix, which are not available in the BLAS. A proposal is given
for implementing triangular matrix multiplication, based on the idea of GEMM-
based BLAS, in [72].

The multiplication of two triangular matrices can be handled by other BLAS
level 3 subroutines and some level 1 and 2 subroutines, as shown in Figure 5.17.
Firstly, we consider the case $C \leftarrow AB$, where both $A$ and $B$ are lower triangular
matrices.

Figure 5.17: Multiplication of two lower triangular matrices.

By blocking $A$, $B$ and $C$, we can obtain:

\[ C_{ij} = \sum_{k=1}^{i} A_{ik} B_{kj} \]

\[ \text{Note that the desired functionality is } C \leftarrow AB, \text{ whereas } B \leftarrow AB \text{ is provided by ATLAS_xTRMM. The block of } B \text{ may be used in other blocks in which case a matrix copy would be required on the block.} \]
\[ C_{11} = A_{11} \times B_{11} \quad \text{TRMV} \]
\[ C_{12} = A_{21} \times B_{11} + A_{22} \times B_{21} \quad \text{TRMM and COPY} \]
\[ C_{22} = A_{22} \times B_{22} \quad \text{TRMV} \]
\[ C_{31} = A_{31} \times B_{11} + A_{32} \times B_{21} + A_{33} \times B_{13} \quad \text{TRMM, COPY and GEMM} \]
\[ C_{32} = A_{32} \times B_{22} + A_{33} \times B_{32} \quad \text{TRMM and COPY} \]
\[ C_{33} = A_{33} \times B_{33} \quad \text{TRMV} \]

The diagonal block problem can be solved by repeatedly applying level 2 BLAS \text{TRMV} on the submatrix of \( A \), which performs matrix-vector products in the form of \( x \leftarrow A \times x \), where \( A \) is a triangular matrix. For example, a column of \( C_{11} \) is the product of a submatrix of \( A_{11} \) and a column of \( B_{11} \).

![Figure 5.18: Product of a lower triangular matrix and a column vector.](image)

The multiplication of a lower triangular matrix and an upper triangular matrix can be computed in a similar way, where level 2 subroutine \text{GBMV} is used to calculate the diagonal blocks.

### 5.5.3 Summary

After implementing iteration space partitioning, further loop transformations, such as loop tiling, can be enabled. Knowing that the existing optimised BLAS subroutines have been well investigated and optimised, an easy approach for implementing band matrix multiplication is that as much as possible of the computations are performed by calls to the existing high performance subroutines. This is described and explained in Section 6.2.3.

This section made an attempt to match the blocks that constitute band matrix multiplication to the existing optimised BLAS subroutines. Moreover, approaches have been proposed for some blocks that are the multiplication of two triangular matrices (either both lower/upper triangular matrix or one is upper and the
other is lower triangular matrix), which cannot be directly computed by calling the BLAS subroutines.

5.6 Summary of Chapter

This chapter has firstly introduced the band matrix, including current library approaches for implementing an operation for band matrices and the distinct computational costs affected by the matrix bandwidths. Then it is proposed that a general implementation for all the matrices that are mathematically banded should be obtained.

By analysing the difficulties, a methodology, namely iteration space partitioning, can be used to provide an optimised implementation after modification. A Java implementation is provided to perform a source-to-source transformation. How to derive an implementation of a band matrix operation can be described via two steps:

- by excluding the operations involving zeros from the dense matrix operation, defining the corresponding band matrix operation; and
- eliminating the resulting conditions by means of iteration space partitioning.

It is feasible that this capture-and-replace code for eliminating conditions can be either implemented as part of a compiler, or implemented using an AOP language that supports the pointcut-advice model and loop join points, such as LoopsAJ [63]. Note that the statements inside the loops are copied without changes. Eventually, the Java implemented transformer for band matrix multiplication generates 384 blocks of code, each of which has only compound bounds at the outermost loop, i.e., it is able to determine whether a block is executed or not at an early stage of evaluation. Moreover, if the divided blocks of code are accessible to the developer, an efficient band matrix multiplication can be implemented by reusing the existing BLAS subroutines.

\[^{11}\text{LoopsAJ is still based on Java. Due to the lack of tools (no Aspect-Fortran), the original code has been simply replaced by code generated by hand.}\]
Chapter 6

Evaluation

This chapter describes the techniques used to evaluate the novel ideas introduced in Chapters 4 and 5, together with the results of the associated evaluations. Section 6.1 and Section 6.2 analyse the new implementations in terms of software engineering properties and performance, respectively. A summary is given in Section 6.3.

All test cases are executed on a machine with:

- Intel Core 2 Duo Processor E6300 (64-bit instruction set, 2MB L2 cache, 1.86 GHz clock speed) and 4GB memory;
- Ubuntu 11.04 (64-bit) and Linux kernel version 2.6.38-11-generic.

Details of the compilers and flags used are given in Figure 6.1.2 and Figure 6.2.3, respectively. All the tests choose the minimum execution time from 100 runs. An error estimation is given in Appendix A.

6.1 Evaluation of Modularising Three Concerns

The work reported in this thesis contains two parts. The first part is the discovery of three crosscutting concerns that naturally exist in NLA libraries, proposing the use of AOP for modularising these concerns.

Two Java implemented NLA libraries, namely JAMA and JOOLALA were refactored by modularising one or more types of the crosscutting concerns encountered in building a NLA library that were described in Chapter 4. Because

\footnotetext[1]{CPU throttling was turned off to avoid random timing.}
JAMA only supports dense matrix operations, applying grouping matrix operations cannot benefit the design, as discussed in Section 4.1.3. Moreover, matrix operations, such as transpose and matrix multiplication, that may not use an in-place algorithm, have only one interface that always returns a new matrix. Therefore, only pre-condition checks are applied to JAMA, whereas JOOLALA is refactored so as to modularise all three types of crosscutting concerns. An aspect is created to handle the exceptions in JAMA, removing the scattered `try-catch` statements among a number of methods in class `JAMA`. In JOOLALA, four aspects are developed: one for pre-condition checks, one for matrix copy, two for grouping matrix operations. The first two aspects change the behaviour of the original class `Matrix`, whereas the latter two reconstruct the class `Property` and its subclasses. The refactored JAMA and JOOLALA are called Aspect-JAMA and Aspect-JOOLALA, respectively in the following. This section contains two parts: modularity evaluation (Section 6.1.1) and performance evaluation (Section 6.1.2).

### 6.1.1 Modularity Evaluation

By applying AOP for handling the three naturally existing crosscutting concerns introduced in Chapter 4, the software components of a NLA library can be better modularised. The refactored program (AspectJ version) is compared with the original program (Java version) for two libraries, namely, JOOLALA and JAMA, to show these modularity improvements, which are examined using the suite of software metrics that is presented in Section 3.2.5. Similar improvements can also be made to traditional NLA libraries, such as the BLAS. However, due to the lack of tools (no AOP language implementation for Fortran) and metrics (no metrics for AOP as an extension to a procedural programming language), the effects on traditional libraries have not been evaluated.

The following assessments are based on the application of the suite of metrics to both the original and refactored versions of the two systems, i.e. JAMA and JOOLALA. This metrics suite examines the target systems in terms of separation of concerns, system coupling and cohesion, and size. This metrics suite has been used to measure modularity in several other studies (see Section 3.2.5). Tables 6.1 to 6.6 present the results.

Tables 6.1 and 6.2 give results for the separation of concerns metrics. Except for concern diffusion over components in JAMA, all the other fields show
a significant decrease, from 67.86% to 100% for the two systems. In particular, concern diffusion over LOC in JAMA is completely removed. Tables 6.3 and 6.4 show the obtained results for the coupling and cohesion metrics. Most fields in these two versions retain the same value, except that lack of cohesion in operations decreases slightly, with less than 1% gain. Tables 6.5 and 6.6 show the
obtained results for the size metrics. The refactored (AspectJ) versions slightly increase the number of attributes and number of operations. The number of components, including both classes and aspects, grows 14.29% and 10.42% for JAMA and JOOLALA, respectively, as expected, due to the introduction of exception handling aspects. The number of lines of code reduces by 1.03% for JAMA and 4% for JOOLALA. It is worth noticing that, although the total number of lines saved in JOOLALA is a small percentage of the overall number of lines of code of the system, it accounts for nearly 15% of package Property. This was caused by applying grouping matrix operations in JOOLALA. It can be expected that, with more matrix properties and matrix operations supported by a library, a greater number of lines can be saved if grouping of matrix operations is applicable.

### 6.1.2 Performance Evaluation

The two NLA libraries are compiled by OpenJDK 1.6.0 (64-bit), whereas the aspect versions are compiled by AspectJ compiler 1.6.12.M1. The JVM was invoked with flag `-server` and `-Xmx2000m`. The performance is evaluated by counting the time in nanoseconds \(^2\) for each modified matrix operation in JAMA.

\(^2\) Method `nanoTime()` is called before and after invoking the method. The difference between the two returned values is the elapsed time.

---

**Table 6.5:** Size metric for JAMA.

<table>
<thead>
<tr>
<th>Lines of Code</th>
<th>No. of Attributes</th>
<th>No. of Operations</th>
<th>Vocabulary Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>3313</td>
<td>3257</td>
<td>29</td>
</tr>
<tr>
<td>Aspects</td>
<td>N/A</td>
<td>22</td>
<td>N/A</td>
</tr>
<tr>
<td>Total</td>
<td>3313</td>
<td>3279</td>
<td>29</td>
</tr>
<tr>
<td>Diff.</td>
<td>-1.03%</td>
<td>0.00%</td>
<td>+1.09%</td>
</tr>
</tbody>
</table>

**Table 6.6:** Size metric for JOOLALA.

<table>
<thead>
<tr>
<th>Lines of Code</th>
<th>No. of Attributes</th>
<th>No. of Operations</th>
<th>Vocabulary Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>7350</td>
<td>6749</td>
<td>97</td>
</tr>
<tr>
<td>Aspects</td>
<td>N/A</td>
<td>307</td>
<td>N/A</td>
</tr>
<tr>
<td>Total</td>
<td>7350</td>
<td>7056</td>
<td>97</td>
</tr>
<tr>
<td>Diff.</td>
<td>-4.03%</td>
<td>+1.03%</td>
<td>+2.87%</td>
</tr>
</tbody>
</table>
and JOOLALA. One or two (due to different matrix operations) $200 \times 200$ matrices with randomly assigned elements are generated for each operation. The minimal execution time is chosen from 100 runs for each operation.

**Performance Comparison of JAMA and Aspect-JAMA**

Ten operations, namely, plus, minus, arrayTimes, arrayRightDivide, arrayLeftDivide and their corresponding xEquals are affected by modularising the pre-condition checks in JAMA. Each of the operations has two implementations, for example, method plus is for $C \leftarrow A + B$ and plusEquals is for $A \leftarrow A + B$. Aspect-JAMA handles the exceptions in an aspect rather than among these operations. The performance results are given in Figure 6.1.

Figure 6.1 shows that the performance of the refactored AspectJ version is comparable with the original (Java) version for JAMA. Although the Aspect-JAMA version runs slightly faster in some cases and the JAMA version runs slightly faster in other cases, the differences are trivial and most likely to be due to error in measurement (see Appendix A).

**Performance Comparison of JOOLALA and Aspect-JOOLALA**

Aspect-JOOLALA can be applied to all the three methods described in Chapter 4. Pre-condition checks and matrix copy concerns reside within methods of class Matrix, whereas the grouping matrix operations concern takes place in methods of class Property and its subclasses. Because JOOLALA supports different matrix properties, each matrix operation is provided with five tests, i.e., the input matrix or matrices are dense, band, upper triangular, lower triangular or symmetric. The matrix dimensions used are $200 \times 200$ and both upper and lower bandwidths are 50 if a band matrix is chosen. The performance comparisons of JOOLALA and Aspect-JOOLALA for different operations are given in Figures 6.2 and 6.3.

However, the time cost of Aspect-JOOLALA is significantly slower than that of JOOLALA. In particular, matrix scale and minor show the worst results. Matrix scale and minor in Aspect-JOOLALA are about several tens of times slower than in JOOLALA. Note that the performance penalties are caused by

---

3In JAMA, the pre-condition checks handled by aspects have effects on both performing matrix operations and creating a matrix from an array. Only affected matrix operations are evaluated.
Figure 6.1: Execution time in nanoseconds of operations in JAMA and Aspect-JAMA.
Figure 6.2: Execution time in nanoseconds for methods `transpose`, `scale` and `minor` in JOOLALA and Aspect-JOOLALA.
Figure 6.3: Execution time in nanoseconds for methods `add`, `sub` and `mult` in JOOLALA and Aspect-JOOLALA.
use of arguments bound in the cflow pointcut for applying grouping of matrix operations. If only applying pre-condition checks and matrix copy in JOOLALA, the AspectJ version performs as well as the original version (Figures 6.4 and 6.5).

As described in Chapter 4, aspects group the matrix operations for distinct properties from two “aspects”: one is to construct iteration spaces for each matrix property; the other is to assign the matrix element according to the matrix operation. The former part is captured lexically, whereas the latter part has to be captured by specifying the execution flow in which it resides. Moreover, the arguments (the matrix objects) bound to that execution flow have to be exposed.

The pointcut cflow picks up all the join points that are in the execution flow of certain events. More specifically, for any pointcut p, cflow(p) applies at the point in the execution of the program when p matches some states in the call stack at that program point [14]. And, if there are arguments bound to p, these are bound to the matched values which can be found nearest the top of the call stack. For example, the pointcut

\[
cflow(\text{execution(void Property+.add(Matrix, Matrix))} \\
\&\& \text{args(m1, m2))} \&\& \\
\text{call(void Property.assignAll(*))}
\]

matches all the method calls to assignAll that occur within the dynamic scope of the execution of add in class Property and its subclass. Also, m1 and m2 are bound to the values of the arguments of the last call to add.

It is obvious that the use of cflow pointcuts requires dynamic tests to be inserted in the program to see whether or not the current state matches condition cflow(p). It also has been indicated in the literature that the performance overhead introduced may be substantial [42], both because of the update to the state and the dynamic test to be inserted. [14] proposes a few strategies for effectively handling these dynamic pointcuts, such as sharing cflow states and caching cflow stacks. These ideas were implemented in the AspectBench compiler (abc)\(^4\) and some of them were integrated into the AspectJ compiler. However, the results in Figures 6.2 and 6.3 show that the performance overhead caused by cflow with args is still substantial in the up-to-date AspectJ compiler.

---

\(^4\)After removing all the annotations (abc does not support annotations), the Aspect-JOOLALA version cannot be successfully compiled by abc as an exception arises. Therefore, it is unknown whether the ideas implemented in abc may significantly improve the performance.
Figure 6.4: Execution time in nanoseconds for methods $\text{transpose}$, $\text{scale}$ and $\text{minor}$ in JOOLALA and Aspect-JOOLALA without Grouping Matrix Operations.
Figure 6.5: Execution time in nanoseconds for methods add, sub and mult in JOOLALA and Aspect-JOOLALA without Grouping Matrix Operations.
6.1.3 Summary

This section has firstly evaluated the new implementations in terms of modularity. The refactored versions based on modularising three crosscutting concerns by aspects in a NLA library, as introduced in Chapter 4, are compared with the original versions through a suite of metrics, namely separation of concerns, coupling and cohesion and size metrics, as presented in Section 6.1.1. The results show that there is a significant improvement in the separation of concerns metrics.

The performance is then measured via a set of test cases, showing that the performances of the refactored versions are on a par with the original versions, except where there is heavy use of cflow pointcut with arguments, which is an already identified performance defect in the testing language, namely AspectJ.

We conclude that software engineering benefits have been achieved with mostly low impact on performance (and it is possible that the one instance of high impact on performance could be accelerated by future development of tools for AspectJ).

6.2 The Evaluation of Band Matrix Multiplication

The second part of the work described in this thesis gives a design of general band matrix operations, in particular, band matrix multiplication and using the BLAS as an example. The first benefit is functionality, which supports the multiplication of two general band matrices. The second benefit is in the area of software engineering, i.e., better modularity and extensibility can be obtained. By providing band matrix multiplication, the modularity is improved, due to the fact that existing subroutines which implement the matrix multiplication of two special band matrices can be replaced so that the concern of multiplication is not scattered; moreover, extensibility is gained because, if the library is required to support other special band matrix multiplication, such as multiplying two triangular matrices, a straight-forward implementation can be provided by a method call with suitable arguments. However, although some example codes are provided to illustrate that traditional libraries suffer from the code scattering problem, due to the lack of tools (no AOP language implementation for Fortran) and metrics (no metrics for AOP as an extension to procedural programming),
the modularity improvements in traditional libraries are difficult to access (Sections 6.2.1 and 6.2.2). The performance of band matrix multiplication is measured by the execution time (Section 6.2.3).

As introduced in earlier chapters, the BLAS specifies four element data types: single precision, double precision, complex, and double precision complex. Dense matrix-matrix multiplication \texttt{xGEMM} is designed as $C \leftarrow \alpha \text{op}(A)\text{op}(B) + \beta C$, where \text{op}(X) = X, X^T, X^H$. For simplifying the tests, the new subroutines for handling general band matrix-matrix multiplication made the following changes:\(^5\)

- implement real number only;
- \text{op}(X) = X;
- $\alpha = \beta = 1$.

A new subroutine with parameters (\texttt{TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC, MUA, MLA, MUB, MLB}) is designed for $C \leftarrow A \times B + C$, where both $A$ and $B$ are general band matrices. Three new subroutines, namely \texttt{DUOPGBMM}, \texttt{DGBMM} and \texttt{AT4_DGBMM} are provided. \texttt{DUOPGBMM} adopts the code shown in Figure 5.8, whereas in \texttt{DGBMM}, the code is divided into blocks to eliminate all the max/min conditionals. \texttt{AT4_DGBMM} reuses the highly optimised BLAS subroutines to build an efficient implementation for band matrix multiplication. See Section 6.2.3 for details of the three subroutines.

On the one hand, subroutines \texttt{DGEMM} and \texttt{DTRMM}\(^6\) can then be implemented by calling \texttt{DGBMM} with corresponding bandwidth parameters \texttt{MUA, MLA, MUB} and \texttt{MLB}; the performance of \texttt{DUOPGBMM} and \texttt{DGBMM} is compared with the subroutines of the Netlib’s BLAS. On the other hand, subroutine \texttt{AT4_DGBMM} still needs the existing BLAS subroutines, in order to provide an highly optimised implementation; the performance of it is compared with the subroutine in ATLAS.

\(^5\)This simplification is just for performing the tests easily and quickly; the band algorithm itself can be applied more widely.

\(^6\)Strictly speaking, although this design of \texttt{xGBMM} can replace \texttt{xGEMM}, it may not take the complete work from \texttt{xTRMM}, which differs in two ways: an in-place algorithm is used and the unit triangular matrix (a triangular matrix where the entries on the main diagonal are all one) is supported. Recall that \texttt{xTRMM} is for $B \leftarrow \alpha \text{op}(A)B$ or $B \leftarrow \alpha A\text{op}(B)$, where $A$ is a unit or non-unit triangular matrix, $B$ is a dense matrix and \text{op}(A) is $A$, $A^T$ or $A^H$. It is proposed to provide another interface, which handles the unit triangular matrix using an in-place algorithm. This operation is defined as: $B \leftarrow \alpha \text{op}(A)B$, or $B \leftarrow \alpha B\text{op}(A)$, where $A$ is an upper or lower banded matrix with ones on the main diagonal or not and $B$ is a band matrix. However, due to time constraints, this operation has not yet been implemented.
6.2.1 Modularity Evaluation

Chapter 5 proposes to implement general band matrix operations, and to build other matrix operations, that involve only mathematically banded matrices, upon them. The BLAS, the de facto programming interface for performing basic matrix operations, is chosen to show that the library is better modularised. Moreover, this idea enables the NLA library’s extensibility, by supporting more functionalities that may be needed in future updates.

As described in earlier chapters, many NLA libraries, such as the BLAS and LAPACK, define a large number of subroutines for computing the special cases of band matrix operations, resulting in an implementation explosion on the number of subroutines. In particular, the code that focuses on one concern is scattered among different subroutines. As an example, the two blocks of code shown in Figure 6.6 are chosen from subroutines \texttt{DGEMM} and \texttt{DTRMM}, respectively.\footnote{Slight modification has been made in order to expose the similarities, without affecting the semantics.}

* \texttt{DGEMM} 
\begin{verbatim}
  DO 90 J = 1,N
    ... 
    DO 80 L = 1,K 
      ... 
        DO 70 I = 1,M 
          C(I,J) = C(I,J) + B(L,J)*A(I,L)
      70 CONTINUE
    80 CONTINUE
  90 CONTINUE
\end{verbatim}

* \texttt{DTRMM} 
\begin{verbatim}
  DO 50 J = 1,N
    DO 40 L = 1,M 
      ... 
        DO 30 I = 1,L - 1 
          B(I,J) = B(I,J) + B(L,J)*A(I,L)
      30 CONTINUE 
    40 CONTINUE
  50 CONTINUE
\end{verbatim}

Figure 6.6: \texttt{DGEMM} and \texttt{DTRMM} code fragments.
By introducing band matrix multiplication as the basic operation, the NLA library can be better modularised, by writing the concern of the band matrix multiplication at one place, and handling the concern of using an in-place algorithm and the implementation of unit-diagonal matrix multiplication somewhere else. For example, the code in Figure 6.6 can be replaced by the code in Figure 6.7. If an AOP language for Fortran with a loop pointcut were to be implemented, the statement within the innermost loop for the other two concerns could be handled in an aspect. However, due to the lack of tools, the current implementation does not address the latter two concerns.

\[
\begin{align*}
\text{DO } 90 & \quad J = 1, N \\
\text{DO } 80 & \quad L = \max(1, J-ku2), \min(K, J+kl2) \\
& \quad \cdots \\
\text{DO } 70 & \quad I = \max(1, J-ku, L-ku1), \min(M, J+kl, L+kl1) \\
& \quad C(I,J) = C(I,J) + B(L,J)*A(I,L)
\end{align*}
\]

Figure 6.7: A DUOPGBMM code fragment that corresponds to the code in Figure 6.6.

In order to maintain program compatibility, it is proposed that the existing interfaces for a matrix operation remain unchanged, and one new interface for band matrix is added. The implementation of the subroutine for that band matrix operation is optimised and other implementations only need to call the band matrix operation subroutine with corresponding parameters.

6.2.2 Extensibility Evaluation

As the complexity of a software project grows in scale, the reuse of code from existing systems is essential for improving the software development lifecycle. Because of this, a system is divided into “building blocks” to reduce the production costs for it. It is important that each component can be subject to future changes. However, specific changes cannot generally be foreseen, so an important property of software reuse, called extensibility, is used to evaluate a system on whether it is able to be adapted to new tasks without accessing the source code [80].

Previous studies observe that extensibility is often achieved by using high-level abstractions, such as inheritance, overriding and late-binding in object-oriented
programming, or higher-order functions in functional programming [80], metadata in aspect-oriented programming [116] or parameterised types [80].

However, procedural programming lacks these abstractions, as procedural changes are the only approach to model changes [70] and developers have no way to hide the implementation details. New features can only be added to existing procedures by creating a new procedure, which refers all existing services to the old procedures in addition to implementing the new services it provides. As a result, extensibility of a system developed in a procedural programming language is difficult to achieve due to the limitations of the programming paradigm.

Although many traditional NLA libraries, such as the BLAS, are implemented using procedural programming, the idea described in Chapter 5 presents a system with better extensibility by supporting a generalised set of procedures, i.e., band matrix operations. For example, band matrix-matrix multiplication can be applied to any matrices that are mathematically banded. Not only are the two subroutines xGEMM (general dense matrix-matrix multiplication) and xTRMM (triangular-dense matrix multiplication) supported, but also others, such as triangular-triangular matrix multiplication and tri-diagonal matrix multiplication. As long as the band matrix operation is optimised, future functionality updates can be implemented without modifying or copying the original code.

6.2.3 Performance Evaluation

Two $1000 \times 1000$ dense matrices with equal lower and upper bandwidths (from 0 to 999, increased by 10) and randomly assigned elements are generated as input to this test. All the tests choose the minimum execution time from 100 runs. The compiler used is GCC (gfortran) 4.5.2 and the compilation flags used were `-O3 -fPIC -m64`.

Unlike in Java, where methods are provided to obtain current system time in microseconds and nanoseconds, a tool is needed to measure the system time on a small scale in Fortran. Performance Application Programming Interface (PAPI) specifies a standard application programming interface for accessing the hardware performance counters that are available on most modern microprocessors [3] and thus PAPI is chosen to measure the performance characteristics.

---

8Intel Fortran (ifort) was also used but the results from it is very similar to that generated by gfortran, except that the code generated by ifort runs slightly faster.
There are two sets of tests to be performed. One compares the performance of different implementations of band matrix multiplication, including:

- **DUOPGBMM**, an implementation directly derived from **DGEMM** in BLAS, as introduced in Section 5.2;
- **DGBMM**, which is implemented by using iteration space partitioning on **DUOPGBMM**; and
- **AT4_DGBMM**, modified from **DGBMM** by replacing 4 blocks of code that perform dense-dense matrix multiplication by calling **ATLAS DGEMM** \(^9\) from ATLAS, a tuned BLAS, giving some insights about the performance of a fully optimised band matrix multiplication subroutine.

The first set of tests show how performance is improved by applying certain optimisation techniques; and the other set compares the performance of the implementation of band matrix multiplication with specially designed subroutines for matrix multiplication in BLAS, i.e., general dense matrix multiplication and triangular-dense matrix multiplication.

### Different Implementations of Band Matrix Multiplication

The number of floating point operations performed for bandwidths from 0 to 999 is shown in Figure 6.8. Because the algorithm for multiplying two band matrices remains unchanged, the number of floating point operations for the different implementations are the same. Figure 6.8 shows that, as the bandwidth increases, the number of floating point operations to be performed increases more and more rapidly to a certain point, then increases more and more slowly.

Figure 6.9 gives the time cost of **DGBMM** and **DUOPGBMM** in microseconds. It shows that, on the one hand, the curve showing the time cost of **UOPGBMM** is similar to the curve showing the number of floating point operations, on the other hand, the curve showing the time cost of **DGBMM** does not follow the number of floating point operations. The time cost of **DGBMM** seems to have two parts: one is from bandwidth 0 to 500 and the other is from 500 to 990. The time cost of

---

\(^9\)Recall that **DGBMM** contains 384 blocks of code. Each block is a loop nest. Some of these are empty loop nests because some loop bounds are restricted by mathematical constraints, such as the sum of the two input matrix bandwidths are larger than or equal to the bandwidth of the output matrix. Each of the non-empty loop nests typically defines a subspace of the three dimensioned space shown in Figure 5.10.
Figure 6.8: The number of floating point operations to be performed for matrix-matrix multiplication with bandwidth 0 to 999.

Figure 6.9: Execution time in microseconds of matrix-matrix multiplication with bandwidth 0 to 999 for DUOPGBMM and DGBMM.
the first part increases more and more rapidly, whereas the second part increases nearly linearly with increasing bandwidth.

The two subroutines perform similarly when either the bandwidth is from 0 to 200 or close to 999 (the result matrix is dense in this case). However, for other cases, when the matrix bandwidth is neither too small nor too big, DGBMM runs up to 19.97% faster than DUOPGBMM. It is also worth noting that, when the upper and lower bandwidths of the two input matrices are larger than or equal to 500, the result matrix is dense.

![DGBMM vs AT4_DGBMM](image)

Figure 6.10: Execution time in microseconds of matrix-matrix multiplication with bandwidth 0 to 999 for DGBMM and AT4_DGBMM.

As described in Section 5.5, the blocks generated by iteration space partitioning prepare the code so that other loop transformations, such as loop tiling can be applied. Recall that each block typically handles a subspace of the three dimensional space of band matrix multiplication. Due to time constraints, only 4 out of 384 blocks (these blocks performs the multiplication of two dense matrices) are implemented by making procedure calls to the corresponding subroutines of ATLAS. Figure 6.10 shows that the performance can be significantly improved. Notice that the 4 blocks do not execute until the bandwidths are equal to or
Figure 6.11: Execution time in microseconds of matrix-matrix multiplication with bandwidth 0 to 999 for AT4_DGBMM and ATLAS_DGEMM.

larger than 500, and the affected subspaces become larger with increasing bandwidths. In the extreme case, where the bandwidth is 999, the 4 blocks dominate most of the computations and thus AT4_DGBMM runs only about 10% slower than ATLAS_DGEMM, as shown in Figure 6.11.

Recall that the implementation of ATLAS_DGEMM does not change to use a fast matrix multiplication algorithm, such as Strassen’s algorithm. As a result, the same number of floating point operations are executed and the performance improvements are merely due to the reuse of cache. Figure 6.12 shows that, by calling the optimised ATLAS subroutine in AT4_DGBMM, the level 2 data cache misses can be significantly reduced, approaching that of ATLAS_DGEMM when the 4 blocks dominate the computations.

Although ATLAS_DGEMM maximises the reuse of cache, it provides the same
Figure 6.12: Level 2 cache misses of matrix-matrix multiplication with bandwidth 0 to 999 for DGBMM, AT4_DGBMM and ATLAS_DGBMM.

time results for all the bandwidths as the structure of zero elements is not exploited. A graph computing ATLAS-based AT4_DGBMM and ATLAS_DTRMM is not provided, because the current implementation only optimises 4 blocks, none of which execute for triangular-dense matrix multiplication.

**Generalised Implementation vs Specialised Implementations**

The final test compares the performance of DGBMM and DGEMM for dense-dense matrix multiplication, and compares that of DGBMM and DTRMM for upper triangular-dense matrix multiplication. Figure 6.13 shows that all three implementations are comparable.

**6.2.4 Summary**

Code fragments from the BLAS subroutines from Netlib have been used to show the scattered code problem. By viewing band matrix multiplication as a basic
matrix operation, the modularity of existing systems can be improved, and further functionality update may be easily implemented without modifying or copying the original code.

New subroutines for handling band matrix multiplication (one with conditions eliminated and one without) show that they both run as fast as the specially designed subroutines \texttt{DGEMM} and \texttt{DTRMM} for solving their problems. However, the two new implemented subroutines differ significantly when the input matrix bandwidths are neither too large nor too small: the code with conditions eliminated runs up to 19.97% faster than the one with conditions. In addition, an efficient implementation called \texttt{AT4_DGBMM} can be built upon the subroutines of the existing tuned BLAS, such as ATLAS. The performance of \texttt{AT4_DGBMM} is close to (about 10% slower than) \texttt{ATLAS_DGEMM} from ATLAS for computing a dense matrix multiplication.

### 6.3 Summary of Chapter

This chapter has evaluated the new implementations in terms of software engineering properties and performance. Section 6.1 shows that by modularising the
three concerns via AOP, the refactored versions come with measurable modularity improvements and no performance penalties if the language pointcut cflow with arguments is not heavily used. Section 6.2 shows that the use of band matrix multiplication may resolve some code scattering problems in BLAS, and enhance the extensibility, without sacrificing the performance. Moreover, a fully optimised subroutine for multiplying two band matrices can be designed by reusing the existing tuned BLAS, though this approach does not come with any modularity improvements.\footnote{\textsuperscript{10}}

Note that the Java implementations in this chapter are not as fast as Fortran or C implementations. This is due to some of Java’s unique features, which compromise performance when compiled naively, including [13]:

\begin{itemize}
  \item exception checks for null-pointer and array access out-of-bounds;
  \item a precise exception model;
  \item the lack of support for real multi-dimensional arrays.
\end{itemize}

It would be expected that similar results could be achieved in Java implementations if these problems were resolved and higher order transformations, such as loop transformations, were enabled.

\begin{footnotesize}
\textsuperscript{10}The modularity is improved if the matrix multiplications are implemented by calls to GBMM. However, by dividing DUOPGBMM into a number of blocks of code, each block is computed by one or more existing BLAS subroutines.
\end{footnotesize}
Chapter 7
Comparison of Techniques

Chapter 3 has introduced two methodologies that can be used to separate one concern from the other for better productivity. Many early studies [77, 62, 44] claim that the use of *Aspect-Oriented Programming* (AOP) provides the usual benefits of modularity, or extensions of it, such as maintainability and reusability. However, less work has been done on how AOP can be used to improve a system’s performance, which is one of the most important properties in a *Numerical Linear Algebra* (NLA) library. Chapter 5 provided an example, illustrating that, by providing an around advice to bypass the original code with the semantically equivalent but faster code, performance can be improved. This way of achieving high speed of execution is an essentially similar mechanism as that used in compiler optimisation techniques. The similarity between the two techniques leads to some interesting and open questions: What are the relationships between bypassing code by AOP and compiler optimisation techniques provided by a compiler? Can AOP take some work from a compiler for code optimisation? If so, what would be the appropriate abstraction allowed to be captured by an aspect? Are there any problems that can only be optimised by around advice? Are there any undesired effects if further optimisations are performed by a compiler after the code has been replaced by around advice?

This chapter starts with an introduction to the science of compiler optimisation (Section 7.1), then describes some case studies, showing how AOP can be used to improve performance, in Sections 7.2 to 7.4. Comparisons between implementing optimisation techniques by around advice in AOP and in a compiler

---

1 Compiler optimisation techniques, such as strength reduction and code motion, often operate on the low-level statements, whereas the execution points defined by pointcuts in AOP are at a higher level, such as method calls or class initialisation in AspectJ.
are presented from different viewpoints in Section 7.5.

7.1 The Science of Optimisation

The term *optimisation* in compiler design refers to techniques by which a compiler produces “high performance” code from the original code. High performance usually means high speed of execution, but it can sometimes refer to other properties, such as smaller code size in embedded systems, or less power consumption in mobile devices. Four design objectives are proposed for compiler optimisations in [8]:

- The optimisation must be correct, which means the meaning of the program must be retained;
- The optimisation must improve the performance of many input programs;
- The compilation time must be kept reasonably short; and
- The engineering effort required for implementation needs to be manageable.

Implementing optimisation is a complex and difficult task so that things may easily go wrong. However, it is clear that an optimising compiler is useless if the code it generates does not give the same result as the unoptimised one, no matter how fast it performs. The first objective places supreme emphasis on the correctness of the optimising transformation.

Secondly, although the object of the optimising process is to improve the performance, there is no guarantee that the “optimised” code always runs faster for any inputs. Occasionally, performance may be impeded. Moreover, as described above, the definition of performance may vary from system requirement to system requirement. In particular, the improvement of one kind of performance may conflict with another. For example, function inlining is a widely used compiler optimisation technique for removing method call overhead, at the cost of growing code size. However, in embedded systems, this technique may not be appropriate as the embedded processor memory size is relatively small.

The third requirement is that the compilation time must be kept short, to support rapid development and to reduce the debugging cycle. Optimisation is often the last step to be processed, because the unoptimised code is easier to
debug. Turning on certain optimisations can both increase the compilation time, as it is a complex transformation, and sometimes even introduce new problems, causing the optimised code to need to be re-tested. Therefore, in a program where performance is not an critical issue, the use of optimisation may be deferred.

Last but not least, due to the complexity of a compiler design, it has to be assured that the engineering and maintenance efforts for the compiler are manageable. The number of optimisations that could be implemented in a compiler is infinite, a priority must be designed so that only those optimisations that bring about the most benefits need be implemented.

Around advice in AOP may freely replace the original code, as there is no tool preventing arbitrary code bypassing. However, the four design objectives for building a compiler can still be used as guidelines, suggesting that around advice used for improving performance should follow these requirements.

7.2 Case Study: Eliminating Virtual Method Calls

The implementations of standard and general-purpose middleware, such as J2EE, CORBA and .NET, can be remarkably complex, according to the wide range of supported domains and applications. However, developing a particular product may need only some of the supported features. Such generality and feature-richness of the middleware may bring about performance overhead for a product line. In addition, the potential for optimisation may be masked due to the generality of the middleware. Therefore, middleware specialisation has been proposed for improving performance of middleware by removing unused features.

[73] describes a novel way of using AOP for specialising middleware, i.e., only a set of features needed by the product line is selected. This work chose ACE C++ middleware as an example, showing that, by using AspectC++ to implement the specialisation of ACE middleware, performance improvement can be seen in terms of decreased latency and increased throughput.

Originally, ACE middleware had an object-oriented event-driven interface, namely the Reactor, which is an implementation of the Reactor pattern. It was developed for supporting alternative kinds of alternative concurrency models [73].

\footnote{www.dre.vanderbilt.edu} \footnote{http://www.aspectc.org}
In order to support all the alternative models, the design of the Reactor is implemented by delegation and parameterised types. For example, `ACE_Reactor_Impl` is used as an abstract class to delegate the work to its subclasses `ACE_Select_Reactor` (for single threaded reactor) and `ACE_TP_Reactor` (for thread pool reactor) by virtual method calls.

The refactored ACE middleware relies on a simple fact: once a particular kind of Reactor is selected, it will not change during the development of a product. Around advice is therefore used to eliminate all the related virtual method calls, i.e., replacing calls to all the virtual methods in class `ACE_Reactor_Impl` by the concrete methods in its subclasses. The results show that, by applying this idea, the performance of both single threaded implementation (3% decrease in latency and 2% increase in throughput) and thread pool implementation (4% decrease in latency and 3% increase in throughput) are improved.

### 7.3 Case Study: Inlining Decision as a Concern

An example of using AOP to automate a global inlining optimisation is given in [50]. In order to remove the overhead of the execution time caused by function calls/returns and function epilogue/prologue, inlining is used to execute the function body directly, rather than calling the function. Although the use of function inlining reduces the execution time, the overall code size may increase unpredictably.

Traditionally, inlining was implemented as parameterised macros in a Procedural programming language, such as C. In later languages, such as C99 or C++, an `inline` function was supported to alleviate the compilation errors caused by macros. A keyword `inline` is written alongside a function identifier to indicate that the function needs to be inlined. This is because the compiler does not have the system-wide or design-specific knowledge to decide whether inlining of a function call pays off or not. Different strategies for inlining implemented in a compiler specified by compilation flags are often at low-level and relatively simple, such as inlining all the static functions or considering inlining by the number of instructions in a function.

However, the side effects of using the `inline` keyword, such as significant code size increase, are generally difficult to be foreseen by the programmer. On the one hand, excessive inlining of the function invocations may increase code size, since
code may not be shared in the same function called by different classes; on the other hand, less inlining may cause even larger code size, as register allocation cannot be optimised so that the function has too much unnecessary function epilogue and prologue.

From the AOP point of view, the concern of the inlining decision is scattered over the whole system and should be separated from the functional requirements. This allows complex optimisation rules to be designed, by handling inlining with AOP. A global inlining strategy for each member of the system, controlled by aspects, is introduced in [50].

7.4 Case Study: Implementing an Optimisation Technique

It can be shown that the use of AOP and compiler optimisation techniques may improve the performance of a system. A compiler optimises the hand-written code via a number of techniques, such as code motion or loop tiling, as introduced in Chapter 3, whereas these transformations can also be performed via around advice to bypass code with optimised code. Chapter 5 provides an example in a NLA library, showing that a loop nest with max/min functions in its bounds can be eliminated either by AOP (if there is an AOP implementation for the language that writes the library and the loop nest can be captured by a pointcut) or by a compiler (if the specific optimisation is implemented).

The loop is probably the most widely seen component in a NLA library. Due to the computationally intensive features of NLA libraries, the loop is also the place where the most time-consuming computations occur. Numerous efforts have been directed at optimising loop structures for better performance, including loop interchange, loop skewing, loop tiling, loop unrolling, index set splitting, iteration space partitioning, etc. These optimisations are often implemented in a compiler and controlled by corresponding compilation flags. However, from the AOP point of view, under which performance can be treated as a crosscutting concern scattered over the whole system, certain loop optimisations can be implemented in an aspect. In particular, when a specific optimisation technique is not implemented in a compiler, the use of AOP may provide an optimised code by bypassing the unoptimised code.\(^4\)

\(^4\)The loop is a special language idiom which differs from many other artificial structures.
7.5 Comparison between the Two Approaches

The performance of a system can thus be optimised by either Aspect-Oriented Programming (AOP) or a compiler. A question arises whether there are any advantages or disadvantages associated with either approach. One distinct difference is that a compiler does not often have the project-wide knowledge, whereas AOP can implement the program with application-specific knowledge. A compiler is typically designed for solving general problems in a domain; therefore, the generality of the compiler may compromise specific optimisations. AOP may be used as a method to make specialised optimisations, such as middleware specialisation described in Section 7.2. Note that it is unnecessary to use AOP to excessively implement a compiler’s work, as many low-level optimisations in a compiler are common to all the programs that may be developed.

Secondly, the approaches use different mechanisms to capture the phases where optimisation should occur. According to whether the original code needs to be prepared or not, the mechanism for finding the place to apply optimisation can be either direct or indirect. In AOP, a pointcut can be defined directly to capture an execution point of a program, or the source code can be annotated to be captured, which is referred to as indirect capture. Compiler optimisation is specified via a number of compilation flags, which are predefined to directly choose the desired optimisations to be performed. Also, keywords, such as volatile in Fortran, inline, volatile and register in C, can be used to expose certain places to be optimised or unoptimised, which can be considered as indirect capture via a compiler. It is clear that the means of capture used in a compiler are simple and restricted, in contrast to AOP, where an execution point that is a complex logical composition with both lexical and dynamical scopes can be defined.

Thirdly, if a specific compiler optimisation technique is missing in a compiler, AOP might be used to implement the optimisation, providing an alternative, and easy-to-implement solution. Although this approach may give some insights for applying a new added optimisation technique to the particular problem, it must be used with extra care as the compiler optimisation sequence may not be optimal (the AOP implemented optimisation has to be applied before the others) and may

Different types of loops vary in syntax but can have the same semantics. For example, in Fortran 77, a DO loop can also be constructed by using IF and GOTO statements. [63] describes how a join point can be made for loop by using LoopsAJ, an extension to the AspectBench compiler. However, tools for handling aspects in Fortran are still missing.
lead to unpredictable effects. In particular, if the number of optimisations to be performed by AOP and by a compiler are both large, the optimisation sequence might be completely destroyed and the effect of it is uncontrollable. In the design of a compiler, two important factors must be considered, namely the set of optimisations to be selected and the order in which they are applied. It is clear that one transformation may either create or destroy opportunities for the other. Moreover, the effects of some high-impact transformations may be hidden until the original program has been transformed to a limiting factor, which can occur anywhere during the transformation process [29]. Therefore, finding the best order for a set of optimisations is difficult. Although implementing an optimisation by AOP before searching for the best order for the remaining optimisations by a compiler may lead to undesired overhead, there are two advantages of this approach:

- It is the only way to test the effects of the new added optimisation when it is missing in the compiler; and

- The aspect that handles the new added optimisation is application-specific, i.e., it only needs to find out whether the program performs better than the original one. This is not as difficult as a new compiler technique to be added in a compiler, where a number of different tests have to be performed.

7.6 Summary of Chapter

This chapter has introduced how to improve the performance of a system by AOP, which is compared with optimisations performed in a compiler in order to find the similarities and differences. Although compiler optimisation techniques have been developed for several decades, much less work has been done on analysing the equivalent AOP approach. The relationships between the two techniques from distinct viewpoints have been discussed and revealed.
Chapter 8

Conclusion

8.1 Thesis Review

This thesis involves three core areas of background knowledge: Numerical Linear Algebra (NLA), Software Engineering and Compiler Optimisation Techniques. The problem domain lies within the field of NLA libraries, the development of which is inspired and benefited by software engineering and compiler optimisation techniques.

NLA is fundamental to Computational Science and Engineering (CS&E) problems. The NLA community has developed a large number of subroutines by exploiting the structure of non-zero elements in matrices, and grouped these into libraries for computing NLA operations. It has been observed that, developing and maintaining a NLA library with comprehensive functionalities requires a great deal of work, in particular other factors, such as performance, have to be considered at the same time. Therefore, the ideas of finding a correct abstraction and reusing functionalities have been the main concerns though the development of various NLA libraries.

Two functionality reuse techniques, namely, the pointcut-advice model of aspect-oriented programming (AOP) and procedure call of procedural programming have been used, in order to implement NLA operations in the thesis.

AOP is a technique to handle crosscutting concerns in a modular way. A concern is anything of interest to a stakeholder. Separation of concerns is the process of breaking a program into distinct features with minimal overlaps in functionalities. Although the majority of concerns can be localised in traditional
programming languages, a few cannot be cleanly decomposed into a single component and result in code scattering or/and code tangling within the program. AOP provides a mechanism to overcome this limitation, by writing the code (advice) that has effects in many non-local places (defined by pointcut). From AOP’s point of view, a NLA library is a natural place for crosscutting concerns to occur. Examples of crosscutting concerns in a NLA library are pre-condition checks, the representation of matrix operations, or performance. Moreover, the idea of separating concerns enables the redesign of NLA libraries, by providing band matrix operations as an abstraction to replace the functionalities of other matrix operations.

Procedure call is used to provide a fast implementation by reusing the existing highly optimised subroutines so that the software engineering efforts can be reduced. It is based on a simple fact: the implementation is optimised if the subroutines constitute it are all optimised.

Compiler optimisation is often used to improve the performance of a system. It is applied to remove the performance overhead that may be introduced by the implementations of band matrix operations. Furthermore, the removal of unnecessary code enables the application of further optimisations, such as loop tiling. Although this loop transformation is obviously a compiler optimisation technique, due to its absence in current mainstream compilers, it is treated as a crosscutting (performance) concern and can be implemented using AOP. The relationship between AOP and compiler optimisation techniques is hereby discussed by means of several case studies.

The new implementations of three kinds of concerns (grouping matrix operations, pre-condition checks and matrix copying) and band matrix multiplication were measured in terms of performance and software engineering properties. Performance was analysed by a set of test cases, whereas the software engineering properties, such as modularity, was evaluated via a suite of software metrics and changing of models, i.e., treating band matrix operation as the generalised operation.

We conclude that this thesis comes with three contributions:

- Three kinds of crosscutting concerns that are natural in a NLA library have been identified; and approaches to address these concerns have been proposed using AOP for modularity improvements.

- Band matrix operations, in particular band matrix multiplication,
been proposed for NLA libraries. Band matrix operations use the *generalised* algorithm that can be applied to solve many *specialised* problems. Associated optimisations were performed in order to achieve comparable performance to the specialised approaches.

- The relationship between the pointcut-advice model of AOP and compiler optimisation techniques for performance improvements has been identified; and a review to analyse the pros and cons for applying the two mechanisms was presented.

### 8.2 Critique

A major omission in this thesis is that it has not provided a complete implementation for efficient band matrix multiplication. The current work only optimises 4 out of 384 blocks, through many of these are empty loop nests in any specific operation. Also, the best storage format for a band matrix has not been investigated.

Using AOP for grouping matrix operations, as introduced in Chapter 4, makes heavy use of dynamic pointcut `cflow` with arguments. This is an already identified performance defect in the implementation language, namely AspectJ. The work for optimising the compiler for the use of this has not been carried out.

The iteration space partitioning is implemented by a Java program to perform a source-to-source transformation. The effects of implementing this technique in a compiler are unknown and the sequence of performing a set of chosen optimisation techniques is unclear. Another issue is that, if the transformation is implemented via a compiler, the reuse of existing optimised BLAS subroutines may be impossible. Deciding which subroutines to call is more likely to be the work of a developer rather than a compiler.

This thesis has asserted that, with band matrix operations, the modularity of a NLA library can be improved. However, due to the lack of language support and lack of quantified measurement, a more scientific approach, such as software metrics or experiments with software developers, to justify this claim is missing.
CHAPTER 8. CONCLUSION

8.3 Future Work

The obvious immediate further work would target the omissions described in Section 8.2. Firstly, it is important to find out how each block generated by iteration space partitioning generated block corresponds to the multiplication of what kinds of matrices. With a full implementation, it is expected to see the curve of \texttt{AT4.DGBMM} (Figure 6.11) lower than that of \texttt{ATLAS.DGEMM} for all possible bandwidths. And this research has concentrated on the calculation savings for exploiting the structure of the non-zero elements, the storage format needs to be investigated in future work.

The second part of the immediate future work would be to investigate the optimisations for pointcut \texttt{cflow} with arguments. AspectJ has optimised the use of pointcut \texttt{cflow} without arguments and the AspectBench compiler (abc) is used to implement further optimisations. The abc website has recently migrated from Oxford to McGill. The effects of the optimisation of abc is unknown to the experiments described earlier on as a compilation error occurs. Further study would start from the optimisations that are claimed to be implemented in abc.

Thirdly, an alternative approach for evaluating the modularity with band matrix operations is to apply the idea to a Java implemented NLA library that supports different matrix properties. However, although this field does not lack tools or metrics, it lacks the implementation of NLA libraries. This leads naturally to the most obvious line for future work, namely the development of an Aspect-Oriented version of Fortran, together with appropriate tools and metrics. Existence of such innovations would, on the basis of the ideas presented in Chapter 4 and 5, allow more direct exploration of techniques that have the potential to revolutionise the design and development of NLA libraries.
Bibliography


Report SE-136-25, Information Processing Society of Japan (IPSJ), March
2002.
Appendix A

Error Estimate

The performance of the systems were measured by retrieving the minimal value from 100 runs, as introduced in Chapter 6. The minimal value of 100 runs was assumed to be the “true” minimal value for analysing the performance of different implementations. However, each run generates a different execution time (a sample), which is affected by a number of factors, such as cache reuse and other program getting executed by the operating system during the execution. It is important to provide an error estimate, suggesting that the minimal value from 100 runs are relatively reliable for representing the behaviours of the programs and the performance results are repeatable for future tests.

Section A.1 introduces histogram and how to choose an optimal bin-width. Section A.2 picks up three set of test cases from different working environments.

A.1 An Introduction to Histogram and Bin-width

Histogram has been used to estimate the probability distribution of an variable. The use of this method is described in the following:

1. Divide time into discrete and equally-sized bins;
2. Count the number of observations that fall into each bin;
3. Draw the histogram where the x-axis is the execution time and the y-axis is the number of observations.

Although this method is straightforward and easy, a parameter called bin-width or bandwidth that defines the smoothness of the distribution curve is missing. If the bin-width is too small, the histogram may fluctuate dramatically and
thus it is difficult to discern the underlying occurrences of data. In the other case, when the bin-width is too large, the histogram becomes flat and cannot represent the shape of the distribution curve due to its low resolution.

Shimazaki and Shinomoto [103] proposed a method for providing an optimal bin-width. Supposing we have \( N \) bins with bin-width \( w \), the number of observations in bin \( k_i \), the average number \( k \) and variance \( v \), where

\[
k = \frac{\sum_{i=1}^{N} k_i}{N}
\]

\[
v = \frac{\sum_{i=1}^{N} (k_i - k)^2}{N}
\]

The cost function \( C(w) \) is defined as:

\[
C(w) = \frac{2k - v}{w^2}
\]

where \( w \) is changed to minimise the cost function \( C(w) \). See [103] for more details.

A.2 Analysis of the Histograms

In Chapter 6, the minimal value from 100 runs was treated as the “true” minimal value for performance evaluation. This section provides the distribution curve of 100, 1000 and 10000 runs for discussing whether the evaluation might be affected by the sampling errors.\(^1\) Recall that the execution time is in nanoseconds and the input matrices are 200-by-200 for the Java and AspectJ implementations, whereas the execution time is in microseconds and the input matrices are 1000-by-1000 for the Fortran implementations. The error is calculated as the horizontal distance from the minimal value to the average value from 10000 samples.

A.2.1 JAMA and Aspect-JAMA Test

The matrix addition \( B \leftarrow A + B \) is chosen for revealing the potential errors. Figures A.1 to A.3 shows that the shapes of different probability distribution curve are similar in both implementations of JAMA and Aspect-JAMA. The minimal value and the expected value of 10000 samples of either figures are almost at the

\(^1\)Very few samples that are much larger than others are considered to be noise and thus are discarded.
same place.

(a) Method `plusEquals` in JAMA

(b) Method `plusEquals` in Aspect-JAMA

Figure A.1: Histograms of 100 samples for matrix addition $B \leftarrow A + B$.

Although by having more samples (from 100 to 10000), the minimal value decreases a bit as expected, the differences are trivial and can be ignored. The error is 9.4 and 10.2 microseconds for JAMA and Aspect-JAMA methods `plusEquals`,

(a) Method `plusEquals` in JAMA

(b) Method `plusEquals` in Aspect-JAMA

Figure A.2: Histograms of 1000 samples for matrix addition $B \leftarrow A + B$. 
APPENDIX A. ERROR ESTIMATE

171

(a) Method plusEquals in JAMA

(b) Method plusEquals in Aspect-JAMA

Figure A.3: Histograms of 10000 samples for matrix addition $B \leftarrow A + B$.

respectively. Therefore, we can confirm that the performance results in Figure 6.1 is within the bonds of experimental error.

A.2.2 JOOLALA and Aspect-JOOLALA Test

The matrix addition $B \leftarrow A + B$ is used as an example for illustrating the potential sampling errors. As introduced in Section 6.1.2, a significant performance compromise is caused by the use of arguments that is bound to the cflow pointcut. In order to provide a fair comparison, the implementations of JOOLALA and Aspect-JOOLALA without grouping matrix operations are used. The performance results are given from Figure A.4 to A.6.

The error is 128.3 and 123.1 microseconds for JOOLALA and Aspect-JOOLALA methods add, respectively. The performance of JOOLALA and Aspect-JOOLALA are quite close to each other. This is shown in both Figure 6.3 (where 100 samples are retrieved for different operations) and Figure A.6 (where 10000 samples are retrieved for matrix addition).
A.2.3 BLAS Test

The dense-dense matrix multiplication $C \leftarrow A \times B$ was chosen for testing subroutines \texttt{AT4_DGBMM} (the band-band matrix multiplication that is implemented
APPENDIX A. ERROR ESTIMATE

173

(a) Method add in JOOLALA

(b) Method add in Aspect-JOOLALA

Figure A.6: Histograms of 10000 samples for matrix addition $B \leftarrow A + B$.

by calling ATLAS subroutines) and ATLAS\_DGEMM (dense-dense matrix multiplication from ATLAS). Both input matrices are 1000-by-1000 matrices. 100, 1000 and 10000 samples for both cases are provided from Figure A.7 to Figure A.9.

(a) AT4\_DGBMM

(b) ATLAS\_DGEMM

Figure A.7: Histograms of 100 samples for dense-dense matrix multiplication $C \leftarrow A \times B$.

It is obvious that although the minimal value decreases by having increased
APPENDIX A. ERROR ESTIMATE

Figure A.8: Histograms of 1000 samples for dense-dense matrix multiplication $C \leftarrow A \times B$.

Figure A.9: Histograms of 10000 samples for dense-dense matrix multiplication $C \leftarrow A \times B$.

number of samples, the left sides of the curves in both implementations decrease more slowly and become smooth. And there is always a performance differences between the two implementations, which show that there is a performance penalty
(about 10% slower) to use the band matrix multiplication subroutine for calculating dense matrix multiplication. The differences between the last points of two curves in Figure 6.11 are not caused by sampling errors. The error is 267 and 310 microseconds for the subroutines ATLAS_DGEMM and AT4_DGBMM, respectively.

A.3 Summary

The other methods or subroutines are assumed to have the similar errors of the corresponding examples. The results show that the minimal value from 100 runs is reliable for all the three different working environments.