2018

A Computational study of sparse or structured matrix operations

Aimaiti, Nuerrennisahan (Nurgul)

Lethbridge, Alta. : Universtiy of Lethbridge, Department of Mathematics and Computer Science

A COMPUTATIONAL STUDY OF SPARSE OR STRUCTURED MATRIX OPERATIONS

NUERRENNISAHAN AIMAITI
(Nurgul Amat)
Master of Science, Umeå University, Sweden, 2015

A Thesis
Submitted to the School of Graduate Studies
of the University of Lethbridge
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

A COMPUTATIONAL STUDY OF SPARSE OR STRUCTURED MATRIX OPERATIONS

NUERRENNISAHAN AIMAITI

Date of Defense: August 21, 2018

Dr. Shahadat Hossain Supervisor	Professor	Ph.D.
Dr. Daya Gaur Committee Member	Professor	Ph.D.
Dr. Saurya Das Committee Member	Professor	Ph.D.
Dr. Howard Cheng Chair, Thesis Examination Committee	Associate Professor	Ph.D.

Dedication

" *Sometimes our light goes out but is blown into flame by another human being. Each of us owes deepest thanks to those who have rekindled this light.*" - **Albert Schweitzer**

I dedicate this thesis to my beloved parents, siblings, Dr. Dennis Will and Mrs.Marie-Jeanne Will who have been the source of my inspiration and support.

Abstract

Matrix computation is an important area in high-performance scientific computing. Major computer manufacturers and vendors typically provide architecture- aware implementation libraries such as Basic Linear Algebra Subroutines (BLAS). In this thesis, we perform an experimental study of a subset of matrix operations, where the matrices are dense, sparse, or structured in Java. We implement a subset of BLAS operations in Java and compare their performance with standard data structures Compressed Row Storage (CRS) and Java Sparse Array (JSA) for dense and sparse structured matrices. The diagonal storage format is shown to be a viable alternative for dense and structured matrices.

Acknowledgments

I would like to take the opportunity to express my gratitude and appreciation to my supervisor, Dr. Shahadat Hossain for his continuous support, encouragement, invaluable guidance and standing by me when challenges arose. I would also like to thank my committee members Dr. Daya Gaur and Dr. Saurya Das for their time, constructive comments, and suggestions.

I would like to thank Dr. Howard Cheng and Dr. Amir Akbary for their academic and administrative help. I would like to thank Administrative Support Ms. Barb Hodgson for her cordial support whenever I needed her assistance. I would also like to thank University of Lethbridge, School of Graduate Studies of Canada for funding my graduate program.

A special thanks goes to Dr. Dennis Will and his family for playing such a great role in my life. A role model is a person who has positively influenced someone in their lifes journey. I would like to thank Dr. Will for being that person for me. He is, and always will be the one person that I can turn to for advice in difficult situations, and know that he will always be there to discuss important issues, provide insight and assist me when needed.

I would like to thank my friends for their help. I am fortunate to have Hossein, Jayati, Sahar and Shamria for their excellent and insightful advises, thoughts and providing valuable suggestions.

Last but not least, I must mention my family. There are no words to express my gratitude and thanks to my beloved parents and siblings who have been with me throughout my journey. My family, including many who are several thousand miles away have always put their shoulder to the wheel in providing me with endless support, encouragement and love throughout my life. For this I will be forever grateful.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer languages suitable for writing scientific computing software are expected to have several important features such as efficiency of compiled object code, numerical precision, and the expressiveness of operations. In the past, portability was not among the must have features. However, modern applications are becoming increasingly complex. Some of the challenges are due to the enormous amount of data (exa-scale) needed to be processed and the multidisciplinary nature of applications. Also applications are increasingly required to run on heterogeneous computing platforms.

Some of the main design objectives of language Java were to facilitate platform independence and network-aware distributed application development. Traditionally, numerical scientific computing applications require language features that allow access to computing resources at lower-level such as the ability to access and manipulate heap-memory which is not directly accessible in Java. Moreover, the implementation of Java arrays (two or higher dimensional) pose performance issues for applications that manipulate large matrices and higher dimensional tensors. This is due to the fact that multi-dimensional arrays are built through object references rather than actual objects.

Matrix computation is an important area in high-performance scientific computing. Major computer manufacturers and vendors typically provide architecture- aware implementation libraries such as Basic Linear Algebra Subroutines (BLAS). In this thesis, we perform an experimental study of a subset of matrix operations, where the matrices are dense, sparse, or structured in Java. We implement existing sparse data structures including compressed

row storage and Java sparse array and provide implementations of diagonal storage that can be used for both dense and structured matrices.

1.1 Objectives of this thesis

Java has not been regarded as the next universal language for numerical computing despite its powerful features because of its poor performance in high level numerical computing including Java arrays. This research project is undertaken to identify methods to increase Java performance when using Java arrays. To do this, specific problems explored and investigated in this thesis are as follows:

- Design and implement data structures that exploit sparsity and structure of sparse matrix objects.
- Implement the linear algebraic operations needed in the solution of the linear systems in model algorithms such as Newton's method.
- Experiment and evaluate the implementations on benchmark problem instances.

1.2 Contributions

The contributions of this thesis are listed below:

1. We implement a subset of BLAS operations in Java and compare their performance with standard data structures Compressed Row Storage (CRS) and Java Sparse Array (JSA) for dense and sparse structured matrices.
2. The diagonal storage shceme is shown to be a viable alternative for dense and structured matrices.
3. We perform extensive numerical testing on large dense, sparse, and structured matrices on computing systems with multiple levels of cache memory.

1.3 Thesis Organization

The remainder of this thesis is organized as follows:

Chapter 2 presents the theoretical framework about the Java architecture and the matrix data structures.

Chapter 3 a brief introduction to the basic concepts of cache memory is presented.

Chapter 4 provides the implementation of different storage formats on some basic linear algebra routines and algorithms.

Chapter 5 contains a description of the computational models with results.

Chapter 6 concludes the thesis by summarizing the computational results and discussing possible future research needs.

Chapter 2

Theoretical Framework

2.1 Important Features of Java Architecture

Java is an object-oriented language with a very good collection of useful features. The Java language was developed by the Java group at Sun Microsystems to overcome some software engineering problems introduced by C++.

Java is known as a platform independent programming language as it is designed to operate on all operating systems. Java program functions by generating **bytecode** utilizing the high level programming language called **source code** [8]. A computer cannot execute source codes directly. A source code must be translated into machine code for execution. Figure 2.1 shows Java architecture.

In Figure 2.1, **Java compiler** (called `javac`) converts the source code into bytecode (bytecode files use the extension `.class`). However, bytecode is not an executable file. To execute a btyecode file, at the run time, **Java Virtual Machine (JVM)** interprets the bytecode and converts it into machine code which can then be directly executed on any computer with JVM installed. The JVM plays a key role in making Java portable and user friendly. This demonstrates one of the key benefits of Java which is the independence from platform, *i.e.,* Java-based apps can run on Windows, Linux or Mac operation systems [12]. The same compiled program can be run on multiple platforms without making any changes in the source code, this is one of the essential features of Java [8].

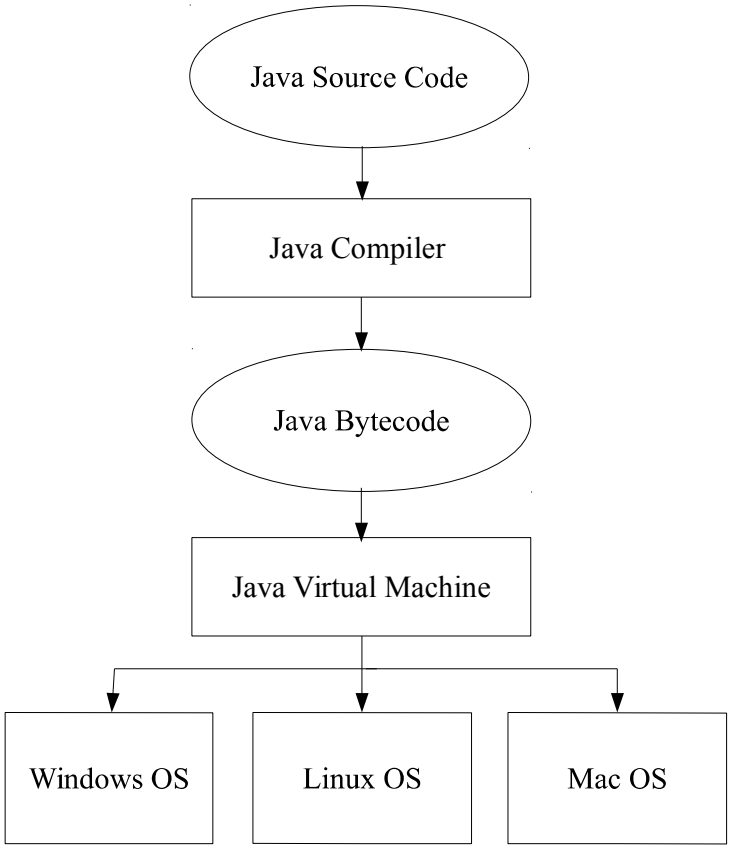

Figure 2.1: The Java Architecture

Gundersen in [12] indicates Java has been very successful due to its platform independence, the extensive libraries it possesses, a run-time system that encourages array-bounds checking, exception handling and an automated memory manager supported by a garbage collector. Java does not have features like pointers, templates and multiple inheritance which although important are more error-prone. Java has not been considered to be a high powered universal language for numerical computing in spite of its very important features because of its poor performance compare to FORTRAN, C and C++.

According to Gundersen [12], Java has poor performance including floating point arithmetic, Java arrays, the memory model and automatic garbage collection.

2.1.1 Memory Management and Garbage Collection

Memory and memory management is a complex field of computer science and there are many techniques being developed to make it more efficient [30]. Dynamic memory management is an integral characteristic of every modern programming language. There are two types of dynamic memory management: **manual** and **automatic**.

Manual memory management requires the programmer to explicitly return memory to the language when it is no longer needed. The key feature of a manual memory manager is the memory manager does not recycle any memory without such an instruction. Manual memory management is used in Fortran, C, C++, Pascal, etc [30].

Automatic memory management frees the programmer from this burden. Memory is automatically reclaimed when the run-time system can determine that it can no longer be referenced. Automatic memory managers are often referred to as garbage collectors. Most modern languages use mainly automatic memory management: BASIC, Dylan, Erlang, Haskell, Java, JavaScript, Python, etc. There are several reasons for the integration of the automatic memory management into the programming languages [30].

The advantages of automatic memory management are:

- Programmer time is freed to work on the actual problem
- Module interfaces are typically cleaner
- There are fewer memory management problems
- Memory management is often more efficient than manual memory management

The disadvantages of automatic memory management are:

- Memory is sometimes retained because it is reachable, when in fact it will not be used again
- Limited availability to certain languages

Garbage collection is an important part of the Java virtual machine's (JVM's) dynamic memory management system. It frees up occupied memory that is no longer referenced by any reachable Java object. In general, only the Java objects that are still referenced, and thus reachable, are kept. When there is no reference to an object, Java will assume that this object is not used anymore. When the garbage collection process happens, the unneeded objects are destroyed and the memory is reclaimed.

In many other programming languages, it is the programmer's responsibility to delete the garbage, but in Java, it is the responsibility of the system, not the programmer, to keep track of which objects are "garbage". The garbage collection process frees up space to enable new object allocation [8, 11].

2.1.2 Java Native Arrays

All major computer programming languages support arrays. However, Java arrays have some very unique and specific properties that other programming languages do not have making them particular useful for computing algorithms. In [31] , an *array* is defined as being a sequence of indexed components which have the following general properties:

- When an array is constructed, its length (or number of components) is fixed.
- Each component of the array has a fixed and unique index. The indices range from a lower index bound to a higher index bound.
- Any component of the array can be accessed (inspected or updated) using its index.

Other properties of arrays vary from one programming language to another. In Java, arrays are objects unlike C++. Java arrays have the following specific properties [31]:

- For an array of length n, the index range is from 0 to $n-1$.
- The type of array components are stated in the program.

- An array is itself an object. Consequently, an array is selected and allocated dynamically (by means of new) when it is required and is manipulated by reference. An array is automatically deallocated when no longer required and referred to.

- Notation **a [i]** denotes the component of array **a** with index *i*, and the length of array **a** can be noted with **a.length**.

- Java has two data types, one is **primitive data types** which is built by eight set of types named *byte, short, int, long, float, double, char*, and *boolean*. The other is **object data type**. See [31] for more details.

The array algorithms presented in this thesis are expressed in terms of the general properties of arrays and the specific properties of Java arrays.

2.1.3 Multidimensional Arrays

Java is an excellent computing language. One of the language's most significant shortcomings for effective high powered numerical computing is the lack of strong support for multidimensional arrays. The Java programming language does not support true multidimensional arrays. This has been recognised as a major deficiency in Java's applicability to numerical computing. However, when Java is supplemented with multidimensional arrays it can achieve very high-performance levels for numerical computing when used in conjunction with compiler techniques and a collection of array operations [21].

Samuel et al. [21] indicate that there are three options available to improve Java's support for multidimensional arrays. These include:

- The usage of class libraries to assist with the implementation of these structures.

- Utilising JVM to identify arrays of arrays to mimic multidimensional arrays.

- Expanding the Java language with the addition of new syntactic models for usage with multidimensional arrays. It does so by adding the new syntax directly into bytecode.

Samuel et al. [21] show that of these choices the best option for computer programmers depends on the relative importance of each of the metrics, the intended use by Java programmers, the effort required for implementation, and the overall impact on performance.

Java's two-dimensional (2D) native arrays are a blend of an object and a primitive. Each element in the outermost array in Java is an object reference. Each inner array is an array of primitive elements. An array of objects includes the reference to the objects. Similarly, when an array of primitive elements is created the array contains the referenced values for each of the elements. As references are a key component of arrays, an array element may make reference to another array thereby creating multidimensional arrays [14].

The layout of a two-dimensional matrix or rectangular array of numbers is shown in Figure 2.2. Figure 2.3 illustrates the concept of arrays of arrays, simulating a two-dimensional multiarray. In multidimensional arrays the elements in the outermost array are an object reference.

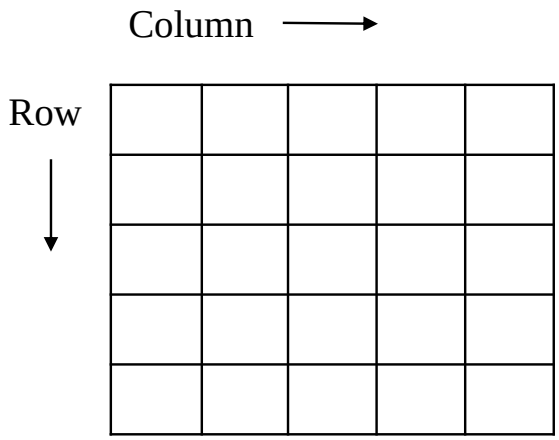

Figure 2.2: A true two-dimensional array

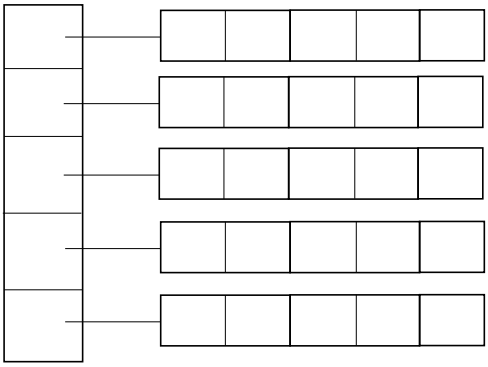

Figure 2.3: A two-dimensional Java array

Since a 2D array is a 1D array of references to 1D arrays, the arrays are not required to be rectangular. Each of these inner 1D arrays (rows) can have a different length which creates a jagged form as demonstrated in Figure 2.4. Furthermore, the structure of arrays can change during a computation. These characteristics make the job of automatically optimizing Java array code almost impossible for existing compilers [21]. Another limitation

of arrays of arrays associates with parameter passing. It is very difficult to pass general regular sections of an array of arrays between caller and callee. Referring to Figure 2.3, where it is trivial to pass row 0 of array, let's say array A to a method (just pass $A[0]$), it is not possible to pass column 0 of array A without first copying it to another one-dimensional array.

We can assume that elements of an array of primitive elements to be stored contiguously, but we cannot assume the objects of an array of objects to be stored contiguously. For a rectangular array of primitive elements, the elements of a row will be stored contiguously, but the rows may be scattered [13]. As objects are created and heap allocated they can be placed anywhere in the memory. Non-contiguous placement and access is less efficient and slower than contiguous ones due to less spatial locality. This applies to all computer languages.

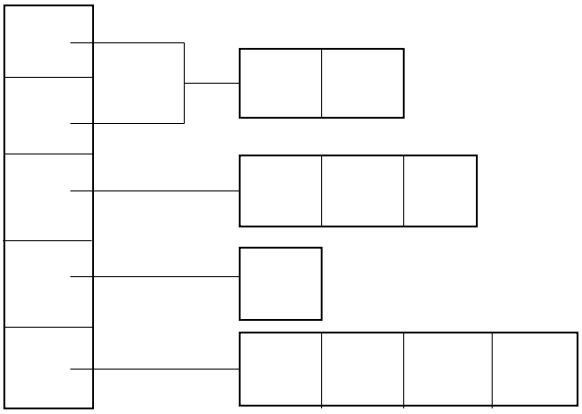

Figure 2.4: A two-dimensional Java array where it has different row length

Java matrix computation packages such as JAMA [15] and JAMPACK [29] use the native arrays as their primary storage format. Unfortunately, Java native arrays are often not considered to be sufficiently efficient for high performance computing. The most challenging aspect of the static format is related to the insertion of new elements when carrying out matrix updates and matrix factorisation. Replacing the typical native arrays with multidimensional arrays and researching different storage formats to take advantage of the flexibility of Java's arrays to create efficient algorithms would be very beneficial [21].

The paper [13] by Gunderson and Steihaug demonstrated that there was no reduction in efficiency when the dynamic and more flexible data structures in jagged arrays were used instead of the more static structure. Most importantly there was a very large gain in efficiency and therefore time saving when the more dynamic and flexible data structures were used.

Java has native support for arrays as parameters. A Java array is self-aware of its own length, unlike C/C++. A two-dimensional array is actually an array in which each element is a one-dimensional array. A two-dimensional array can be passed to a method just as a one-dimensional array, no length parameter is needed [19]. Gundersen and Steihaug[13] recently proposed the use of Java Sparse Array (JSA) storage format to take advantage of Java arrays. JSA is a new format that has more dynamic features compared to the traditionally storage formats like Compressed Row Storage (CRS), Compressed Column Storage (CCS) and Coordinate Storage (COO). JSA is a row oriented storage format similar to CRS. It uses two-dimensional array, which is formed as an array of arrays. Java sparse array format is presented in section2.7.

2.2 Matrices and Data Structures

In computer science, a matrix is known as a *dense matrix* when many, or most of the elements have a non-zero value. Every entry of a dense matrix can be stored in row-major storage, column-major storage, or diagonal storage format. The design and selection of the storage format is based on the intended use, the routines to be implemented, and the pattern of the matrix [12]. In this thesis we use both row-major storage format and diagonal storage format to perform dense matrix multiplications. These two storage formats are described in section 2.3 and section 2.5 respectively.

Wilkinson defines a sparse matrix as *any matrix with enough zeros that it pays to take advantage of them* [9]. Sparse matrices are used in specific ways in computer science, and have different data analysis and storage protocols and techniques related to their use. There

are benefits in both space and time by utilising only the nonzero elements [22]. It is more efficient to store only the non-zero elements of a sparse matrix, so that the use of storage formats for sparse matrices reduces the arithmetic operations on zeros. Sparse matrices can be useful for computing large-scale applications that dense matrices cannot handle.The objective of storage formats for sparse matrices is to reduce memory space by storing only nonzero elements of a sparse matrix and to perform efficient execution of subroutines by storing these nonzero elements in contiguous memory location [28].

Sparse matrices have two types: *structured* and *unstructured* [24]. In a structured matrix, the nonzero elements form a regular pattern, often along a small number of diagonals, while in an unstructured matrix, the nonzero elements are located irregularly. In comparison to dense matrices, sparse matrices have more complex implementation since they only store the nonzero elements and their index positions that they have in the full matrices. One of the challenges with sparse structures is that both the indices and the numerical values of the nonzero matrix entries are stored which requires more overhead. There are long standing different storage formats for unstructured large sparse matrices that are used in computer languages such as C, C++ and Fortran [14, 6, 7, 24]. Developing efficient algorithms to work with matrices is very important. Sparse algorithms also tend to be more complex than the same algorithms for dense matrices, due to the rather complex structures and algorithms for saving space and time [12].

There are a number of storage formats used for storing the sparse matrices(see for instance Saad [23]).

In this thesis, for sparse matrix we use Diagonal Storage (DIAS)format, Compressed Row Storage(CRS), and Java Sparse Array(JSA) formats to perform matrix operations. The common approach of these storage formats is to store only the non-zero elements of the sparse matrix, and employ additional indexing information about the position of these elements. These three storage formats are described in section 2.4, 2.6, and 2.7 respectively.

2.3 The Row-major Storage

A matrix $A \in \mathcal{R}^{m \times n}$ is typically stored as a two-dimensional array. Each entry in the array represents an element a_{ij} of the matrix and is accessed by the two indices, such as i for row index and j for column index. The choices of row-major or column-major indexing can have a significant impact on performances because of the way memory and cache works, and the way multiple indices are converted into a linear index. When the elements of an $m \times n$ matrix is arranged in one-dimensional array in row-major order, element a_{ij} of matrix A is stored at index $i \times n + j$, then all the linear indices $i \times n + j$ are traversed sequentially, resulting in good memory locality. However, if the column-major order is used to arrange elements of A, element a_{ij} is stored at index $j \times m + i$, then the memory access will be scattered in memory.

2.4 The Compressed Row Storage (CRS)

Compressed Row Storage (CRS), also called *Compressed Sparse Row* (CSR), is one of the most general storage formats for sparse matrices whose sparsity patterns have no known regular structure, and can be used to store any sparse matrix. There are a number of variations of the CRS format. The most obvious variation is storing the columns instead of rows. The format is known as Compressed Column Storage (CCS), and is not described in this thesis. Another common variation of sparse matrices exploits the fact that the diagonal elements of many matrices are all usually nonzero and they are accessed more often than the rest of the elements. As a result they can be stored in Diagonal Storage (DIAS) format, which will be described in Section 2.5.

The CRS format represents a sparse matrix that utilizes three one-dimensional arrays [3]:

1. *value* array stores only nonzero elements row-by-row in the matrix.

2. *col_index* (column index) array stores column indices of the corresponding elements in *value*.

3. *row_ptr* (row pointer) array stores the array index of the first non-zero element of each row in the *value* with $row\_ptr(n+1) = nnz+1$, where *nnz* is the number of non-zero elements in the matrix.

The non-zero elements in row *i* can be accessed as $value(row\_ptr(i))$. The memory storage requirement in the *CRS* format for $A \in \mathcal{R}^{n\times n}$ requires only $(2nnz+n+1)$ storage locations instead of storing n^2 elements. Hence, it provides significant storage savings.

An example of a sparse matrix *A* is as follows:

$$A = \begin{bmatrix} 0 & a_{01} & a_{02} & 0 & 0 \\ 0 & a_{11} & 0 & 0 & 0 \\ 0 & 0 & a_{22} & 0 & 0 \\ 0 & 0 & a_{32} & a_{33} & 0 \\ 0 & a_{41} & 0 & a_{43} & 0 \end{bmatrix} \tag{2.1}$$

The *CRS* format for this matrix is represented by three $1D$ arrays given in Table 2.1 below:

Table 2.1: CRS data structure

value	a_{01}	a_{02}	a_{11}	a_{22}	a_{32}	a_{33}	a_{41}	a_{43}
col_ind	1	2	1	2	2	3	1	3

row_ptr	0	2	3	4	6	8

The CRS format is rather intuitive and straightforward, and most toolkits support this format on most sparse matrix operations [27].

2.5 Diagonal Storage (DIAS)

The diagonal storage format, we call DIAS, is a standard storage format for matrices that are diagonally structured, with storage of diagonals on the matrix in consecutive memory

locations. In this section, we explain how to use the diagonal storage format to store a dense matrix and a banded matrix. Given $n \times n$ square matrix of order n. For an element a_{ij} in the matrix with row index i and column index j. Diagonal storage format uses two arrays to define the matrix storage, *diag* and *value* when the diagonals are stored not a specific diagonal order, i.e. main, super and sub diagonals are stored in a random or mixed order. When the specific storage order of diagonal elements is known, we can use only *value* array to represent the matrix storage.

To provide a clear understanding of *diag* and *value* arrays in DIAS, consider the following as an example of how a $n \times n$ general matrix A with number of diagonals $(2n-1)$ is stored in arrays *diag* and *value*. Given the following matrix A:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{2.2}$$

Elements of matrix A is stored in one-dimensional(1D) array by a diagonal order as

$$diag = \{k_0, k_1, k_2, k_3, k_{-1}, k_{-2}, k_{-3}\}, \tag{2.3}$$

$$\begin{aligned} value = \{a_{00} \quad a_{11} \quad a_{22} \quad a_{33} \mid a_{01} \quad a_{12} \quad a_{23} \mid a_{02} \quad a_{13} \mid a_{03} \mid \\ a_{10} \quad a_{21} \quad a_{32} \mid a_{20} \quad a_{31} \mid a_{30}\} \end{aligned} \tag{2.4}$$

where k_i for$(-n+1 \leq i \leq n-1)$ is a diagonal, $n = 4$ is size of matrix.

- $k_i = 0$ represents the main diagonal
- when $0 < k_i \leq n-1$, then k_i is the i^{th} super-diagonal
- when $-n+1 \leq k_i < 0$, then k_i is the i^{th} sub-diagonal

Storing matrices by diagonals has several drawbacks including the fact that the diagonals

have different lengths, and the number of diagonals is more than the number of rows or columns. The algorithm of matrix multiplication for a diagonal storage is more difficult and less intuitive to describe and understand. However, there are obvious advantages to storing or structuring a matrix by diagonals, especially for matrices that are diagonally structured [20]. If the matrix A is banded with bandwidth that is fairly constant from row to row, then it is worthwhile to take advantage of this structure in the storage format by storing the non-zero diagonals of the matrix in consecutive locations. In the following subsection, we explain how we extract each diagonal's elements which are stored in diagonal storage format, for matrix operations.

2.5.1 Formulas to identify and access diagonal elements for dense matrices

In order to find a starting index and ending index for each diagonal element stored in one-dimensional array, we ran several iterations to find an effective formula to identify starting and ending diagonal element's indices.

- Assume k is a diagonal number, $start\_index$ is an index of the first element of a diagonal array, and the end_index is an index of last element of a diagonal array. Array indices are zero-based numbering.

- In array (2.4), we ran $n = 4$ iterations, at the start of each iteration we obtained the following when $0 \leq k \leq n-1$:

k	$start\_index$	end_index
0	$0 = \mathbf{0} \times n - 0$	$3 = 0 + (n - \mathbf{0}) - 1$
1	$4 = \mathbf{1} \times n - 0$	$6 = 4 + (n - \mathbf{1}) - 1$
2	$7 = \mathbf{2} \times n - 1$	$8 = 7 + (n - \mathbf{2}) - 1$
3	$9 = \mathbf{3} \times n - 3$	$9 = 9 + (n - \mathbf{3}) - 1$
	$\Downarrow$	$\Downarrow$
	$= k \times n - \frac{k(k-1)}{2}$	$= start\_index + (n - k) - 1$

the formulas for a starting and ending index of main and super-diagonal arrays can be written as:

$$start\_index = kn - \frac{k(k-1)}{2}, \tag{2.5a}$$

$$end\_index = start\_index + (n - k) - 1 \tag{2.5b}$$

- when $-n + 1 \leq k < 0$, after three iterations it gives the following at the start of each iteration,

k	$start\_index$	end_index
-1	$10 = 10 + (\mathbf{1} - 1) \times n - 0$	$12 = 10 + (n - \mathbf{1}) - 1$
-2	$13 = 10 + (\mathbf{2} - 1) \times n - 1$	$14 = 13 + (n - \mathbf{2}) - 1$
-3	$15 = 10 + (\mathbf{3} - 1) \times n - 3$	$15 = 15 + (n - \mathbf{3}) - 1$
	$\Downarrow$	$\Downarrow$
	$= \frac{n(n+1)}{2} + (\lvert k \rvert - 1) \times n - \frac{\lvert k \rvert (\lvert k \rvert - 1)}{2}$	$= start\_index + (n - \lvert k \rvert) - 1$

therefore, the formulas for a starting and ending index of sub-diagonal arrays can be written as:

$$start\_index = \frac{n(n+1)}{2} + (|k|-1) \times n - \frac{|k|\,(|k|-1)}{2} \tag{2.6a}$$

$$end\_index = start\_index + (n-|k|) - 1 \tag{2.6b}$$

2.6 Banded Matrices and Storage Format

Banded matrices have many zero entries and all non-zero entries are located near the main diagonal. Therefore, it is no surprise that banded matrix manipulation allows for the time and storage space savings. The bandwidth of a matrix A is defined as the maximum of $|i-j|$ for which a_{ij} is nonzero. The upper bandwidth is the maximum $j-i$ for which a_{ij} is nonzero and $j > i$ [18]. For the matrices with few diagonals, the DIAS format stores all the elements of each diagonal with nonzero elements in contiguous one-dimensional array. Thus, the index array *col_index* in CRS format can be deleted [17].

Many storage formats are available, but it is most common to store all the diagonals that contains any non-zero entries. The *tridiagonal* is a special case of a banded matrix that it has nonzero elements only on the main diagonal, the first diagonal below this, and the first diagonal above the main diagonal. In this thesis we focused mainly on the general band matrix where the nonzero elements are located only along a few diagonals adjacent to the main diagonal [10].

A $n \times n$ matrix A:

$$A = \begin{bmatrix} a_{00} & a_{01} & 0 & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 & 0 \\ a_{20} & a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{31} & a_{32} & a_{33} & a_{34} & 0 \\ 0 & 0 & a_{42} & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{53} & a_{54} & a_{55} \end{bmatrix} \tag{2.7}$$

is a *band-diagonal* matrix with elements a_{ij}:

- Matrix A has *lower bandwidth* p if $a_{ij} = 0$ when $i > j + p$
- Matrix A has *upper bandwidth* q if $a_{ij} = 0$ when $j > i + q$

This 6×6 matrix A has lower bandwidth $p = 2$ and upper bandwidth $q = 1$. Instead of storing 36 elements, 16 of them are zero, we store four diagonals $(p+q+1)$, or bands, in one-dimensional array by a diagonal order as:

$$diag\_array = \{k_{-2}, k_{-1}, k_0, k_1\}. \tag{2.8}$$

$$\begin{aligned} value\_array = \{ & a_{20} \quad a_{31} \quad a_{42} \quad a_{53} \mid \quad a_{10} \quad a_{21} \quad a_{32} \quad a_{43} \quad a_{54} \mid a_{00} \quad a_{11} \quad a_{22} \\ & a_{33} \quad a_{44} \quad a_{55} \mid a_{01} \quad a_{12} \quad a_{23} \quad a_{34} \quad a_{45} \} \end{aligned} \tag{2.9}$$

where k_i for $(-2 \leq i \leq -1)$ represents the diagonal number of A.

2.6.1 Formulas to identify and access diagonal elements for banded matrices

In banded matrices,each diagonal element can be identified and accessed by the following formulas:

- $k \geq 0$, super-diagonal elements

$$start\_index = \frac{p(2n-p-1)}{2} + \frac{k(2n-k-1)}{2} + k \tag{2.10a}$$

$$end\_index = start\_index + (n-k) - 1 \tag{2.10b}$$

- $k < 0$, take absolute value of k (ie,. $|k|$), then sub-diagonal elements can be accessed

by,

$$start\_index = \frac{p(2n-p-1)}{2} - \frac{|k|\,(2n-|k|-1)}{2} \tag{2.11a}$$

$$end\_index = start\_index + (n-|k|) - 1 \tag{2.11b}$$

The formulas presented in Equations (2.10) and (2.11) and are to define and access the diagonal elements which are stored in the diagonal storage format in the order sub-diagonals, main diagonal and super-diagonals.

2.7 Java Sparse Array (JSA)

Gundersen and Steihaug [13] recently proposed the Java Sparse Array (JSA) storage format that be used to take advantage of Java arrays. *JSA* is a row oriented storage format similar to *CRS*. It uses two-dimensional arrays, which is formed as an array of arrays. A sparse matrix is represented by two arrays, one of which is the *value* array which stores arrays of the matrix elements. The other array is the *index* array which stores arrays containing the column numbers of the matrix. The data structure to store the example sparse matrix in Equation 2.1 under the *JSA* format are shown in Table 2.2.

Table 2.2: JSA data structure

value	a_{01}	a_{02}	a_{11}	a_{22}	a_{32}	a_{33}	a_{41}	a_{43}
index	1	2	1	2	2	3	1	3

or the *value* array and *index* array can be expressed in Java as

$$double[\,][\,] \quad value = \{\{a_{01},a_{02}\},\{a_{11}\},\{a_{22}\},\{a_{32},a_{33}\},\{a_{41},a_{43}\}\} \tag{2.12a}$$

$$int[\,][\,] \quad index = \{\{1,2\}\},\{1\},\{2\},\{2,3\},\{1,3\}\} \tag{2.12b}$$

In JSA format, each row in the matrix has its elements and indices in a separate array.

All the separate arrays are elements of the *value* array. An important feature of the Java Sparse Array format is that it is possible to make changes to the rows independently without making adjustments to the rest of the structure as each row is composed of a value and a corresponding index array each with its unique reference. This means a row can be removed or inserted into the *JSA* structure without creating a new large 1D array for values and indices. In *CRS* format, it is not possible to manipulate the rows independently without updating the rest of the structure.

The memory storage requirements in *JSA* for $A \in \mathcal{R}^{n \times n}$ is $(2nnz + 2n)$ storage locations compared with $(2nnz + n + 1)$ for the *CRS* format.

Chapter 3

Cache Memory

3.1 Memory Hierarchy

Actual performance of a program can be a complicated function of the architecture of that program. Slight changes in the architecture or program can change the performance significantly. Most programs have a high degree of locality in their accesses. Memory hierarchy tries to exploit locality. The matrix operations performance is effected by a memory hierarchy. In this thesis we have attempted to provide a description of the memory hierarchy system, including cache memory and locality and demonstrate the impact locality has on the functioning of the memory.

The CPU (Central Processing Unit), or simply a processor is the brain of the computer. It is an important computer component that is responsible for executing instructions stored in main memory. The CPU attaches directly to a CPU socket on the motherboard located inside the computer. The speed of a computer CPU is determined by the clock cycle, which is the amount of time between two pulses of an oscillator [16]. The clock speed of a processor is the number of instructions it can process in any given second, measured in gigahertz (GHz). For example, a CPU has a clock speed of 1 HZ if it can process one piece of instruction every second. Extrapolating this to a more real-world example: a CPU with a clock speed of 3.0 GHz can process 3 billion instructions each second.

The CPU contains one or more cache memories to provide fast access to small amounts of memory. To understand the need and use of caches, this chapter introduces some basic knowledge about cache structures. The definitions of terms use in this section are based on

those contained in text book [4] unless otherwise noted.

Figure3.1 shows a typical memory hierarchy with relationship between capacities, costs, and access times.

The CPU consists of set of **registers**, used to hold words retrieved from the cache memory. If the information required is not present in one of the registers, the CPU will request information from memory, by providing the address of the location where the required information is stored. The **cache** will first verify whether it has the requested information available or not. The cache is a relatively small, but very fast, and expensive piece of memory, between the CPU and the main memory. If the requested information is available in the cache, it can be retrieved quickly. If the information is not in the cache, the **main memory** is accessed, the main memory provides the requested information to the cache, and the cache then provides it to the CPU. If the information is not available in the main memory, **secondary memory** devices e.g. magnetic disks (like hard drives and floppy disks), optical disks (ie, CDs and CDROMs) and magnetic tape, are accessed to retrieve the information. Secondary memory is the slowest and cheapest form of memory. It offers a vast amount of storage space, but it cannot be processed directly by the CPU. So, fundamentally, the closer to the CPU a level in the memory hierarchy is located, the faster, smaller and more expensive.

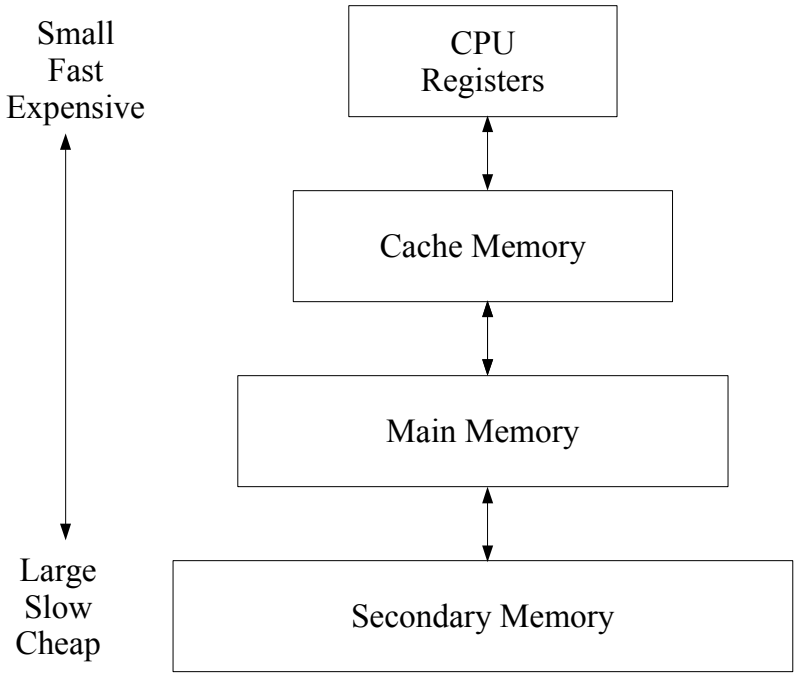

Figure 3.1: Memory Hierarchy

The **main memory** is a temporary storage device that holds both a program and the data it manipulates while the processor is executing the program. Physically, main memory consists of a collection of **dynamic random access memory** (DRAM) chips. Logically, memory is organized as a linear array of bytes, each with its unique address (array index) starting at zero.

3.2 Basic Concepts of Cache Memory

In the early computer systems, the memory hierarchy had only three levels of storage: CPU registers, main memory and disk storage. However, because of increasing the speed gap between CPU and main memory, the cache was introduced to reduce this speed gap. The cache is a very important part of the computer system. It is used to store program instructions and data that are used repeatedly in the operation of programs or information that the CPU is likely to need next. It keeps a copy of the most frequently used data from the main memory. This is to speed up the memory retrieval process. Without a cache the

computer would function very slowly. The cache can be further organized as $L1$, $L2$, $L3$, etc.

- $L1$ **cache,** is relatively small but extremely fast, can be accessed in one or two clock cycles. It is located on the CPU chip.
- $L2$ **cache,** is larger and slower than $L1$. It is located between the $L1$ cache and $L3$ cache.
- $L3$ **cache** is a specialized cache which sits between the $L2$ cache and main memory. Typically, $L3$ cache memory performance is slower compared to $L2$ cache, but is still faster than the main memory.

Basically, $L1$, $L2$, and $L3$ cache work together to improve computer performance. When a request is made to the system, the CPU checks for information it needs from $L1$ to $L3$ cache. If the required information is not found in $L1$, CPU looks to $L2$, then to $L3$ cache.

In computer science, the fundamental property of computer programs is exploited by *locality of reference*, as described in the next section.

3.3 Temporal and Spatial Locality

The ability of cache memory to improve a computer's performance relies on the concept of locality of reference. There are two different types of locality, which are *temporal locality* and *spatial locality* [4].

- **Temporal Locality** *(Locality in Time)* states that the recently referenced data or instructions are more likely to be referenced again soon. Temporal locality can be improved by using blocking technique.
- **Spatial Locality** *(Locality in Space)* states that the recently referenced data or instructions whose addresses are close by tend to be referenced soon. Spatial locality can be improved by rearranging loops. This means changing nesting of loops to access data in order stored in memory, e.g., ijk vs kij, etc.

An example of the impact of spatial locality on processing time can be demonstrated with matrix multiplication. There are several spatial arrangements for matrix multiplication. We can permute the loops (ijk) in any of six possible permutations, *i.e.,* ijk, jik, jki, kji, kij, and ikj. More details for this are described in Section 4.11. We can analyse the different permutations and predict which one will have the best performance. We executed all six versions of the loop (ijk) with different matrix dimensions on a modern system Core i7-4770, where $n \times n$ matrix is considered, where $n = 100, \ldots, 1500$ in steps of 100, and the execution time is calculated in milliseconds. See Figure 3.2.

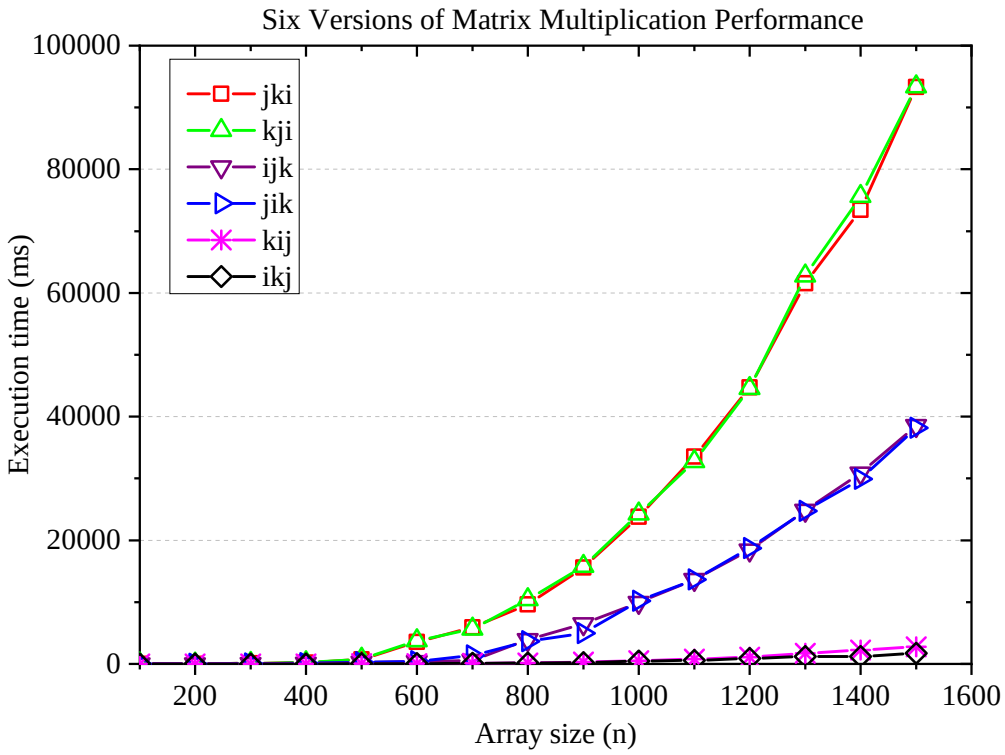

Figure 3.2: Core i7 six versions of matrix multiplication performance

This graph clearly demonstrates that permutations kij/ikj had the fewest number of misses and performed well (for cache miss analysis see section 4.11). The permutations ijk/jik were intermediate. The kji/jki versions each had 2 misses per iteration and were the least effective combinations. By simple analysis, we could actually predict what this graph would look like. When loop orders were rearranged in matrix multiplication to im-

prove spatial locality, there were no improvements to temporal locality.

Temporal locality is improved by using a technique referred to as blocking. To demonstrate this we rewrote the matrix multiplication code for ijk, kji and ikj versions to compare their unblocked matrix multiplication versions. The codes are presented in Appendix.

Figure 3.3 shows the performance of three versions which we call the $bijk$, $bkji$ and $bikj$ versions, of blocked matrix multiply on Core i7 system. We chose block size $bsize$=25. In this program, exactly the same operations are done on the same data. We can see from the results figure that blocking improves the running time. For small array sizes, the additional overhead in the blocked version causes it to run slower than the unblocked versions. There is a crossover point at $n = 500$, after which the blocked versions $bkji$ and $bijk$ run faster. There exist unblocked version ikj of matrix multiply that have the same performance as the blocked version $bikj$.

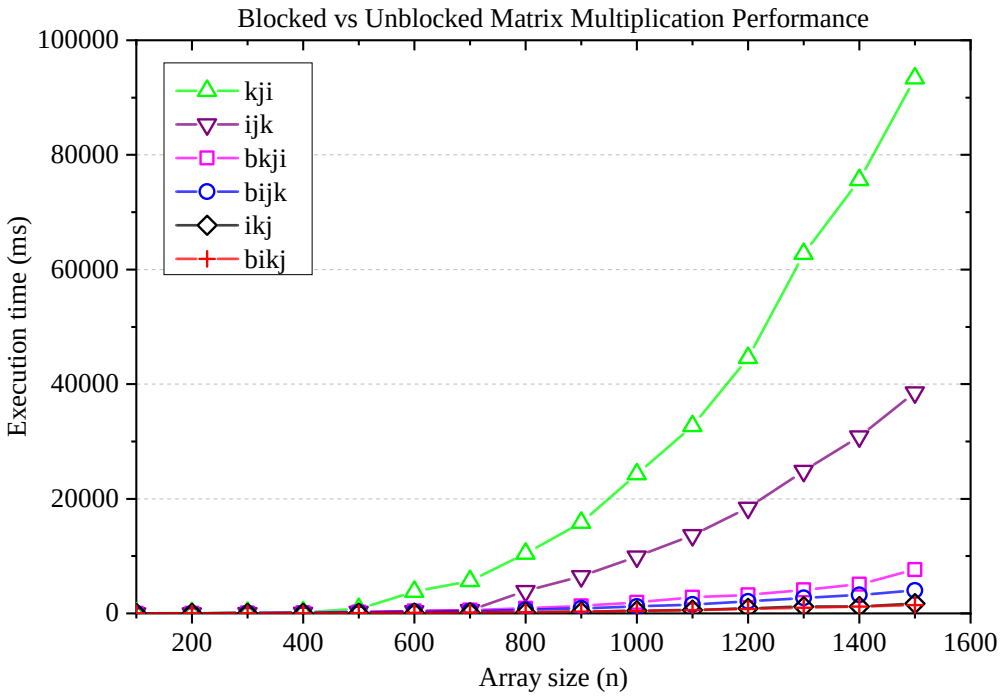

Figure 3.3: Core i7 blocked matrix multiplication performance

In Figure 3.3, legend: $bijk$, $bkji$ and $bikj$ are different versions of blocked matrix mul-

tiplication. The performance of these three unblocked versions is shown in the figure for reference.

We describe the following definitions to assess memory performance:

- **Cache hit**: It occurs if the requested data is in the cache. Hits are good, because the cache can return the data much faster than main memory.
- **Hit rate**: is probability of successful look up in the cache by CPU.
- **Cache miss**: occurs if the requested data is not in the cache. Misses cause time delays because the CPU must then wait for the slower main memory.
- **Miss rate**: is the probability of missing in the cache. It equals to (1-hit rate).
- **Miss penalty**: Any additional time required because of a miss.
- **Average memory access time** $(AMAT) = HitTime + (MissRate \times MissPenalty)$

Cache performance can be improved by using the time to hit the cache, miss rate, and miss penalty. Cache memory is built in automatic hardware storage devices. We can not really control it. If we understand the cache memory, we can take advantage of our knowledge, and exploit it and make our code run faster. We can accomplish this by focusing on the inner loops and try to do accesses with stride 1 to maximise the spatial locality.The temporal locality can be maximized by reusing the local variables which can be put into registers. To provide the background of the cache analysis, we focused on an analysis of matrix multiplication in section 4.11.

3.4 Memory Mountain

We have no control over our machine's memory organisation, but we can rewrite our programs to greatly improve performance. The performance of a memory system can be visualised by the **memory mountain** shown in Figure 3.4, which characterizes the speed at

which memory can be read based on the data access pattern. The memory mountain Figure 3.4 was generated by using the online source code from [25] and used a python script to generate the 3D plot of Core i7 cache memory performance.

The rate that a program reads data from the memory system is called the read throughput, or read bandwidth, typically expressed in MBytes per second (MB/s). If we perform the run function which is provided in [4] repeatedly using different values of size (the number of bytes) and stride (the number of words) then we are able to recover, or plot a two dimensional function of read bandwidth versus temporal and spatial locality called the memory mountain. Perpendicular to the size axis (bytes) there are 4 ridges representing the $L1, L2, L3$ cache, and the main memory. Every computer has a distinctive memory mountain composed of these ridges that indicates the capabilities of its memory system.

The layout of the memory mountain demonstrates clear differences in read throughput. The higher a point is on the ridge the faster the read throughput. The lower any point is on the ridge the slower the read throughput. Each ridge will have different high and low points in read throughput. There are different rates of change between the high and low read throughput for each of these ridges.

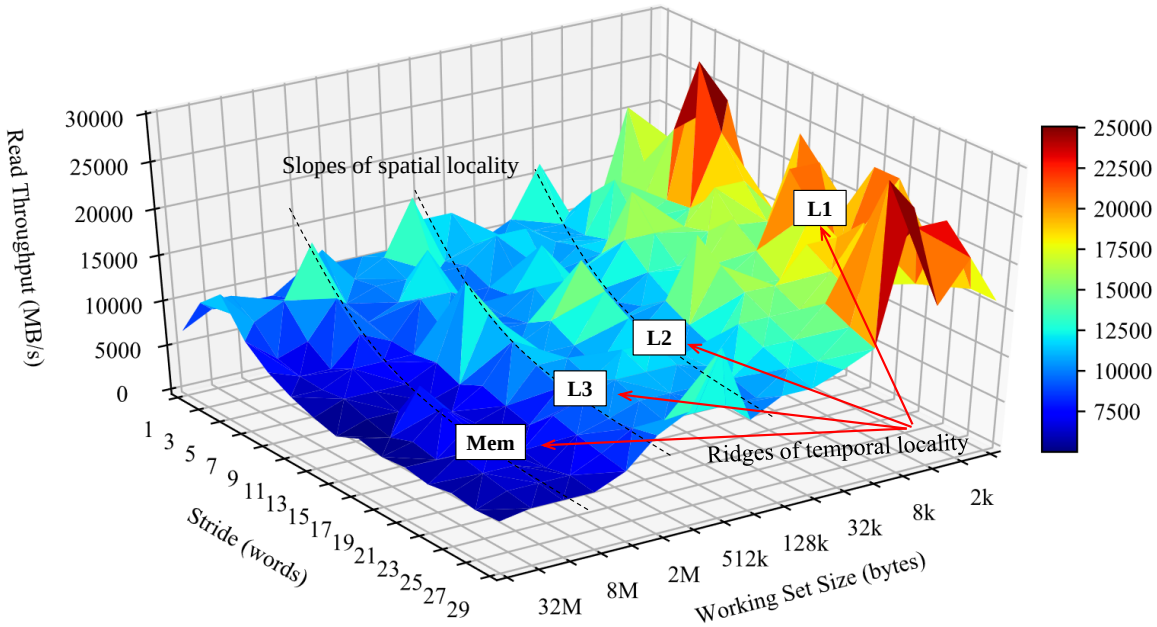

Figure 3.4: Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz - The Memory Mountain

As we increase the stride, the spatial locality is decreased. As we increase the size, the impact of temporal locality is decreased. When we increase the size, there are fewer and fewer caches in the hierarchy that can hold all that data. The speed difference between reading all of data from memory and reading from some part of the caches is enormous. This Figure 3.4 of memory mountain with four ridges each have important features, as follows:.

- Each ridge line of temporal locality corresponds to different levels in the hierarchy.
- The highest ridge line and fastest read throughput are associated with $L1$ cache, which also has the least amount of storage space.
- The second ridge, with intermediate read throughput times is that of $L2$ cache, which has an intermediate amount of storage space.
- The third ridge, with slower than $L2$, faster than main memory throughput is associated with $L3$ cache.

- The fourth ridge, with the slowest read throughput is associated with the main memory, which has the greatest amount of storage capacity.
- For each of the ridges the throughput decreases as there is an increase in the number of strides (words) and reduction in spatial locality.
- The slope is steepest with the greatest reduction in read rates on the *L2* cache memory ridge because of the large absolute miss penalty when there is a transfer blocks of data from the main memory.
- However, even in the main memory where read throughput per second are the lowest there can be a very significant increase in the read rate where there is the smallest stride.

Chapter 4

Matrix Operations on Different Storage Schemes and Algorithms

4.1 Implementation of Some Basic Linear Algebra Routines

In this chapter, we described the implementations of some basic linear algebra routines on different storage formats introduced in Chapter 2. We defined the following abbreviations used for the matrix operations in this chapter:

- Ax: multiplication of a matrix by a vector.
- $A^T x$: multiplication of the transpose of a matrix by a vector.
- AB: multiplication of two matrices.
- $A^T B$: multiplication of the transpose of a matrix by a matrix.

In some cases we used SMM as an abbreviation for Straightforward Matrix Multiplication routine, and DMM for Diagonal Matrix Multiplication routine. These above abbreviations were also used in Chapter 5 with the same meaning.

In order for matrix multiplication to be defined, the number of columns in the first matrix must be equal to the number of rows in the second matrix. For example, in $C = AB$, when $A \in \mathcal{R}^{m \times p}$, $B \in \mathcal{R}^{p \times n}$, then the resulting matrix C can be defined as $A \in \mathcal{R}^{m \times n}$. For simplicity, we considered only square matrices where $m = p = n$ in this thesis.

In the following sections, we implemented four basic linear algebra routines such as Ax, $A^T x$, AB, and $A^T B$ on the straightforward matrix multiplication (SMM), the diagonal matrix

multiplication (DMM) for dense matrix operations, and demonstrated the algorithms that take the row-wise layout into consideration. For sparse matrices, the same routines were carried out on *CRS*, *JSA* and *DIAS*, by using Java arrays for storing the nonzero elements. Matrices A and B were considered to be sparse matrices and x is a full vector. As a result there will be a full vector y in product $y = Ax$ and $y = A^T x$. The implementation of Ax and $A^T x$ on *CRS*, *JSA* and *DIAS* are straightforward. The size of the resulting vector y can be generated as x is a full vector. However, problems can arise when matrix multiplications are implemented as the structure of the output nonzero matrix is unknown before multiplication occurs [12].

4.2 The SMM on Ax and $A^T x$

In this section, we focused on the general matrix-vector multiplications. An example of this would be if we let A be an $n \times n$ matrix, x an n-length vector, and we want to compute $y = Ax$, where the y vector is the solution of the multiplication. When we perform this operation by hand we typically compute the i^{th} element of y by taking the inner product of the i^{th} row of A with vector x:

$$y_i = \sum_{j=0}^{n-1} a_{ij} x_j \quad \text{for } i = 0, \ldots, n-1 \tag{4.1}$$

The general algorithm to compute $y = Ax$ is given in Algorithm 1:

Algorithm 1: Straightforward matrix-vector multiplication

Input data: One-dimensional array *valueA*, in which the elements of matrix A stored in row-major order, and a full vector x of size n

Output data: Full vector $y = Ax$

```
1 for i = 0,...,n-1 do
2     y[i] = 0
3     for j = 0,...,n-1 do
4         y[i] += valueA[i × n + j] × x[j]
5     end
6 end
```

The innermost loop performs the inner product between the i-row of A and x and the outermost loop loops over each row of A and entry in y.

Matrix transpose is a main operation in many matrix- and vector-based computations of image, video, and image/signal processing applications. In the transpose product $y = A^Tx$, the columns of the matrix are traversed, the indices are switched in step(4) of Algorithm1 as $y[i] += valueA[j \times n + i] \times x[j]$, where cache misses may occur with each iteration as an element is accessed by columns if the data is larger than cache size.

4.3 The DMM on Ax and A^Tx

If the $n \times n$ matrix A is stored in DIAS format described in section 2.5, it is still possible to perform a matrix-vector product by using the formulas to identify and access diagonal elements for dense matrices. The algorithm is given in Algorithm 2.

Algorithm 2: Matrix-vector multiplication by diagonals

Input data: *valueA*, *diag*, *x* of size *n*

Output data: vector *y* of size *n*

```
for d=0:(diag.length-1) do
    k = diag[d]
    if k ≥ 0 then
        // A's main and super-diagonals elements multiply with x
        i = k ; j = 0
        start_index=kn − k(k−1)/2            // Equation(2.5a)
        end_index = start_index + n − k − 1   // Equation(2.5b)
        for start_index = start_index : end_index do
            y[j] += valueA[start_index] × x[i]
            i = i + 1; j = j + 1
        end
    end
    else if k < 0 then
        // A's sub-diagonals elements multiply with x
        i = 0 ; j = |k|
        start_index=n(n+1)/2 + (|k|−1)n − |k|(|k|−1)/2
            // Equation(2.6a)
        end_index= start_index + n − |k| − 1   // Equation(2.6b)
        for start_index ≤ endindex do
            y[i] += valueA[start_index] × x[i]
            i = i + 1; j = j + 1
        end
    end
end
```

The input data, output data and the multiplication processes of Algorithm 2 can be outlined as follows:

- Input data: *valueA* denotes an input array of size n^2 which contains elements of matrix A stored in diagonals (*main-super-sub diagonal*) order. *diag* denotes an array of size $(2n-1)$ which contains diagonals, and x is a full vector of size n,
- Output data: a full vector y of size n.
- Total number of elements on each diagonal can be defined by $n-|k|$, where $|k|$ is absolute value of a diagonal.
- the i^{th} element on the main or super-diagonals k, for $i = 0,\ldots,n-k$ and $k = 0,\ldots,n-1$, in matrix A is multiplied by $(i+k)^{th}$ element of vector x, the result is stored in i^{th} position in vector y.
- the i^{th} element on each sub-diagonal k, for $i = 0,\ldots,n-|k|$ and $k = -1,\ldots,-n+1$, in matrix A is multiplied by i^{th} element of vector x, the result is stored in the $(i+|k|)^{th}$ position in vector y.

For the transpose matrix-vector product $y = A^Tx$, the algorithm has the same structure as $y = Ax$ except for a minor variation in the following steps:
Step 4: where $i = k;\ j = 0$ which become $i = 0; j = k$,
Step 14: where $i = 0; j = k$ becomes $i = k; j = 0$.

4.4 AB and A^TB on Six versions Multiplication

In this section, we described six different versions of matrix multiplication. Consider the example of matrix multiplication $C = AB$, for matrix A and B, which have the size of $n \times n$, and C is an $n \times n$ matrix with entries

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik}b_{kj}. \tag{4.2}$$

From this equation, a simple algorithm can be constructed by using the three loops (i, j, k) as in Algorithm 3:

Algorithm 3: A straightforward matrix multiplication

Input data: $A \in \mathcal{R}^{n\times n}$ and $B \in \mathcal{R}^{n\times n}$

Output data: $C = C + AB$, where $C \in \mathcal{R}^{n\times n}$

```
1 for i = 0 : n − 1 do
2     for j = 0 : n − 1 do
3         for k = 0 : n − 1 do
4             C[i][j] += A[i][k] × B[k][j]
5         end
6     end
7 end
```

Interchanging the three for loops results in there being six distinct combinations of matrix multiplication [12] [13]. These can be divided into three groups based on the innermost loop as **Partial row and partial column-oriented** (V1, V2), **Pure column-oriented** (V3, V4), **Pure row-oriented** (V5, V6):

V1: Version 1, loop-order (i, j, k), i.e. V_1_ijk denotes that for each row of matrix A, all of the columns of matrix B are traversed to build up matrix C row-by-row.

V2: Version 2, loop-order (j, i, k), i.e. V_2_jik denotes that for each column of matrix B, all of the rows of matrix A are traversed to build up matrix C column-by-column.

V3: Version 3, loop-order (j, k, i), i.e. V_3_jki denotes that for each column of matrix B, all of the columns of matrix A are traversed to build up matrix C column-by-column.

V4: Version 4, loop-order (k, j, i), i.e. V_4_kji denotes that matrix C is swept through N times column-wise, accumulating on term of the inner product in each pass.

V5: Version 5, loop-order (k, i, j), i.e. V_5_kij denotes that matrix C is swept through N times row-wise, accumulating one term of the inner product in each pass of matrix C's

elements. In this case we traverse the columns of matrix A, but not in the innermost for-loop.

V6: Version 6, loop-order (i,k,j), i.e. V_6_ikj denotes that for each row of matrix A, all of the rows of matrix B are traversed to build up matrix C row-by-row.

Table 4.1 shows how the six versions of matrix multiplication extract data from the A,B and C matrices in the inner loop.

Table 4.1: Six versions innermost loop

Versions	Innermost -loop
V_1_ijk or V_2_jik	for $k = 0 : n-1$ $C[i][j] = C[i][j] + A[i][k] * B[k][j]$ end
V_3_jki or V_4_kji	for $i = 0 : n-1$ $C[i][j] = C[i][j] + A[i][k] * B[k][j]$ end
V_5_kij or V_6_ikj	for $j = 0 : n-1$ $C[i][j] = C[i][j] + A[i][k] * B[k][j]$ end

Matrix transpose is an important problem in real life applications such as image processing, signal modulation and scientific computing applications, etc [4]. It can be expressed

$$c_{ij} = \sum_{k=0}^{n-1} a_{ki} b_{kj}. \tag{4.3}$$

When the input matrices A and B are stored in a row-wise order in one-dimensional arrays as $arrayA, arrayB$, step 4 in algorithm 3 for AB and A^TB in V_1_ijk becomes:

for $C = AB$, $arrayC[i*n+j] += arrayA[i*n+k]*arrayB[k*n+j]$,

for $C = A^Tx$, $arrayC[i*n+j] += arrayA[k*n+i]*arrayB[k*n+j]$.

Where the elements of the result matrix C are also stored in one-dimensional array, called *arrayC*.

4.5 AB and A^TB on DIAS

The theorem that we implemented in the diagonal multiplication computational model is based on a 1976 paper by Niel K. MADSEN et al [20]. The fundamental difference between the algorithms for matrix multiplication $C = AB$ used in this computational model and a standard algorithm is that a diagonal of matrix C is formed, instead of the traditional rows or columns. The algorithm to compute a diagonal c_k of C can be challenging to describe mathematically. However, a very effective graphic representation of matrix multiplication by diagonals was presented in [20].

We also need the concept of a vector or 1D array "*offset* "to understand the notations in Theorem 1. Let $\mathbf{v} = (v_1, \ldots, v_n)$ be a 1D array or vector length of n. The notation $\mathbf{v}(p;q)$ denotes the vector as

$$\mathbf{v}(p;q) = (v_{p+1}, v_{p+2}, \ldots, v_{n-q}) \tag{4.4}$$

We say that $\mathbf{v}(p;q)$ has been obtained by offsetting the vector $\mathbf{v}$ by p from the left (*begin*) and q from the right (*end*) [20].

Theorem 1

Let $A = (a_{-(n-1)}, \ldots, a_0, \ldots, a_{n-1})$ and $B = (b_{-(n-1)}, \ldots, b_0, \ldots, b_{n-1})$ be $n \times n$ matrices stored by the indicated diagonals. Then $AB = C = (c_{-(n-1)}, \ldots, c_0, \ldots, c_{n-1})$ is given by the following algorithm: let $c_k = 0$ for all k, then

(i) for $k = 0 : n-1$, compute c_k by

$$c_k(i-k;) = c_k(i-k;) + a_{k-i}(;k)b_i$$

$$\text{for } i = k+1 : n-1,$$

$$c_k(;i-k) = c_k(;i-k) + a_i b_{k-i}(k;)$$

$$\text{for } i = k+1 : n-1,$$

$$c_k = c_k + a_i(;k-i)b_{k-i}(i;)$$

$$\text{for } i = 0 : k,$$

(ii) for $k = 1 : n-1$, compute c_{-k} by

$$c_{-k}(i-k;) = c_{-k}(i-k;) + a_{-i}b_{i-k}(;k)$$

$$\text{for } i = k+1 : n-1,$$

$$c_{-k}(;i-k) = c_{-k}(;i-k) + a_{i-k}(k;)b_{-i}$$

$$\text{for } i = k+1 : n-1,$$

$$c_{-k} = c_{-k} + a_{-i}(k-i;)b_{i-k}(;i)$$

$$\text{for } i = 0 : k,$$

The steps required to compute the k^{th} diagonal c_k of C are outlined below:

(1) Firstly, for $k \geq 0$ simply delete the bottom k rows of A and the top k rows of B^T.

(2) Secondly, the resulting two congruent $(n-k) \times n$ matrices are multiplied diagonal by diagonal (element by element) to form an intermediate matrix, called D_k size of $(n-k) \times n$.

(3) Finally, the diagonals of D_k are added to each other to form the final result vector c_k

whose m^{th} component is the sum of the elements in the m^{th} row of D_k.

(4) To form c_{-k} for $k \geq 0$, one starts by deleting the top k rows of A and the bottom k rows of B^T and then proceed as steps (2) and (3).

Proof of Theorem 1

Based on the definition of general matrix multiplication, a result matrix element c_{ij} for arbitrary i and j can be expressed by

$$c_{ij} = \sum_{t=0}^{n-1} a_{it} b_{tj} \tag{4.5}$$

We need to show that by the diagonal multiplication formula, the inner product expression (Equation 4.5) for c_{ij} is same as in diagonal multiplication.

Assume c_{ij} is on the k^{th} diagonal, we have the following:

- Diagonal $k = j - i$
- If $j = i$, then c_{ij} is on the main diagonal
- If $j > i$, then c_{ij} is on the superdiagonal
- If $j < i$, then c_{ij} is on the subdiagonal

Without loss of generality, let's take $j \geq i$, and k is a main or superdiagonals.

Assume c_{ij} is an arbitrary element on the main or super diagonals. Define an intermediate matrix D_k of size $(n-k) \times n$, which is formed by using the computational step (1), and step (2). Let d_{ij} be an element of D_k, it can be expressed by

$$d_{ij} = a_{ij} b_{j(i+k)}, \quad \text{for} \quad 0 \leq i \leq (n-k-1), \quad k \leq j \leq (n-1). \tag{4.6}$$

Then using step (3), adding the diagonals (or elements of each row) of D_k to get a vector C_k size of $(n-k) \times 1$. Let $C_k(i)$ denote the i^{th} element of C_k. We can use elements

of D_k to represent $C_k(i)$ as

$$C_k(i) = \sum_{t=0}^{n-1} d_{it} = \sum_{t=0}^{n-1} a_{it} b_{t(i+k)} \tag{4.7}$$

In Equation 4.7, column index of b can be replaced by $j = i+k$ as $k = j-i$ is known. Then, Equation 4.7 becomes as

$$C_k(i) = \sum_{t=0}^{n-1} d_{it} = \sum_{t=0}^{n-1} a_{it} b_{tj} \tag{4.8}$$

From Equation 4.5 and 4.8, we can get

$$C_k(i) = c_{ij} \tag{4.9}$$

Similarly, when $j < i$, for an arbitrary element c_{ij} on the subdiagonals, we can use the same methods above. The intermediate matrix will be written as D_{-k}, and d_{ij} of D_{-k} will be expressed as

$$d_{ij} = a_{(i+k)j} b_{ji}, \quad \text{for} \quad k \leq i \leq (n-1), \quad 0 \leq j \leq (n-k-1). \tag{4.10}$$

The j^{th} element of C_k will become

$$C_{-k}(j) = \sum_{t=0}^{n-1} d_{jt} = \sum_{t=0}^{n-1} a_{(j+k)t} b_{tj} \tag{4.11}$$

where k can be replaced by $i-j$, then Equation 4.12 becomes

$$C_{-k}(j) = \sum_{t=0}^{n-1} d_{jt} = \sum_{t=0}^{n-1} a_{it} b_{tj} \tag{4.12}$$

From Equation 4.5 and 4.12, we can get

$$C_{-k}(j) = c_{ij} \tag{4.13}$$

Equations 4.9 and 4.12 proved that by the diagonal multiplication formula, the inner product expression (Equation 4.5) for c_{ij} is same as in diagonal multiplication.

The computing steps above are conducted by three functions named :

1. *getDiagArrayFunction* which is used to extract the required diagonal elements using index formulas, shown in Algorithm 4.

2. *cutElementFunction* which is used to conduct step (4), shown in Algorithm 5.

3. *AB_multiplydiag* is a main function which is to correlate with the other two functions to complete the overall matrix multiplication process.

The Java codes for these functions are provided in Appendix B.

Algorithm 4: *getDiagArrayFunction*

Input data: *valueA*, k

Output data: A *getdiag_array* size of $(n-|k|))$ that stores the required diagonal elements

1 Initialize a starting index of *getdiag_array* as *index=0*

2 **if** $(k >= 0 \quad \&\& \quad k =< n-1)$ **then**

```
// Extract main or super-diagonal elements
```

3 $start\_index = kn - k(k-1)/2$ `// Equation(2.5a)`

4 $end\_index = start\_index + n - k - 1$ `// Equation(2.5b)`

5 **for** i=$start\_index : end\_index$ **do**

6 $getdiag\_array[index] = valueA[i]$

7 $index++$

8 **end**

9 **end**

10 **else if** $(k >= -n+1 \;\&\&\; k < 0)$ **then**

```
// Extract the sub-diagonal elements
```

11 $start\_index = n(n+1)/2 + (|k|-1)n - |k|(|k|-1)/2$ `// Equation(2.6a)`

12 $end\_index = start\_index + (n-|k|) - 1$ `// Equation(2.6b)`

13 **for** i=$start\_index : end\_index$ **do**

14 $getdiag\_array[index] = valueA[i]$

15 $index++$

16 **end**

17 **end**

18 **Return** $getdiag\_array$

Algorithm 5: *cutElementFunction*

Input data: $getdiag\_array$, $cutBegin$, $cutEnd$

Output data: An $output\_array$ size of $(size - cutBegin - cutEnd)$ where the required diagonal elements stored after cutting elements from beginning or ending in $getdiag\_array$

```
1 size=getdiag_array.length
2 for i = 0 : (size − cutBegin − cutEnd) − 1 do
3 |   output_array[i] = getdiag_array[i + cutBegin]
4 end
5 Return output_array
```

The input data $cutBegin$ and $cutEnd$ in Algorithm 5 denote "*offset*"of $getdiag\_array$.

4.6 Ax and A^Tx on CRS

The matrix-vector product $y = Ax$ using CRS format can be expressed in the usual way as in Equation 4.1 with the following Algorithm 6.

Algorithm 6: Matrix-vector multiplication on CRS

Input data: $valueA$, row_ptr, and col_ind arrays, vector x

Output data: vector y

```
1 for i = 0 : n − 1 do
2 |   y[i] = 0
3 |   for j = row_ptr(i) : row_ptr(i + 1) − 1 do
4 |   |   y[i] = y[i] + valueA[j] × x[col_ind[j]]
5 |   end
6 end
```

This method traverses the rows of the matrix A, and only multiplies nonzero matrix entries, the operation count is $2nnz$, which is a significant time and space savings over the dense operation requirement of $2n^2$. This method has a more favourable memory access pattern as per iteration of the outer loop it reads a row of matrix A and the input vector of x

and writes one scalar [2].

For $y = A^T x$, we switch indices to

$$\text{for all j, do for all i :} \qquad y_i = y_i + a_{ji} x_j \tag{4.14}$$

then, step 4 in Algorithm 6 is

$$y[col\_ind[i]] = y[col\_ind[i]] + valueA[i] \times x[j] \tag{4.15}$$

since this method implies traversing columns of the matrix A, an inefficient operation for matrices stored in CRS format. Because in this method, for per iteration of the outer loop it reads one row of matrix A and one element of the input vector x and both reads and writes the result vector y.

4.7 Ax and $A^T x$ on JSA

As described in section 2.7, the *JSA* format is a row oriented storage format similar to *CRS*. It uses two-dimensional arrays arranged as an array of arrays. This section shows how to perform matrix-vector multiplication routines using JSA format with Algorithm 7.

Algorithm 7: Matrix-vector multiplication on JSA

Input data: *valueA*, *indexA* arrays, and vector x

Output data: vector y

```
1 for i = 0 : n − 1 do
2     subarray = valueA[i]
3     subindex = indexA[i]
4     for j = 0 : (subarray.length − 1) do
5         y[i] += subarray[j] × x[subindex[j]]
6     end
7 end
```

In Algorithm 7, there are two multidimentional arrays, *double*[ ][ ]*valueA* and *int*[ ][ ] *indexA*. These arrays are used to store nonzero elements and the column indices in each row of matrix A.

The innermost for-loop (j) performs the multiplication of all of the rows of matrix A and vector x. The results of the innermost loop multiplication are stored in the resulting vector y.

The outermost for-loop (i) accesses all the rows of matrix A, while the innermost for-loop performing the multiplication. The index of result element in vector y is defined by the index array of a row in A. This is very similar to Ax implementation using the CRS format. However, in JSA format, a single nonzero element in the rows of matrix A cannot be manipulated independently from the rest of the nonzero structures without accessing the array of rows.

The algorithm for A^Tx on JSA format has the same structure as Algorithm 7 except step 5. It should be

$$y[subindex[j]] += subarray[subindex[j]] \times x[i] \tag{4.16}$$

In A^Tx case, accessing the elements by column-wise is very time and space consuming. For innermost for-loop (j) multiplication, each element of vector x is multiplied by each element in subarray of *valueA* array, which means for each element of x, the rows of matrix A are traversed and multiplied with x. The results are stored in vector y in the place given by the column index of matrix A.

4.8 Ax and A^Tx on DIAS

The matrix-vector multiplication of Algorithm 2 presented in section 4.3 can be conveniently applied to banded matrix-vector multiplication. The only difference is obtaining the diagonal elements. If a matrix A is a banded $n \times n$ matrix stored by sub-main-super diagonals order in a one-dimensional array, the formulas in Equation (2.10) and Equation (2.11) are used to identify and access diagonal elements in steps 5, 6 and 15, 16 of Algo-

rithm 2. The banded matrix transpose -vector multiplication is very straightforward. For the transpose matrix-vector product $y = A^T x$, the algorithm has the same structure as $y = Ax$ except for a minor variation in

step 4: where $i = k$; $j = 0$ which become $i = 0$;$j = k$, and

step 14: where $i = 0$; $j = k$ becomes $i = k$; $j = 0$.

4.9 Banded Matrix Multiplication Algorithm on CRS and JSA

In sparse matrices, the algorithm of the matrix-vector multiplication routines are rather straightforward. When it comes to matrix-matrix multiplication routines, it is much less straightforward. The reason for this is that we can not create the exact nonzero structure of the resulting matrix and perform the numerical multiplication without knowing the pattern, or the number of nonzero elements of the resulting matrix. In order to calculate the exact nonzero elements of the resulting matrix, we used the *symbolic approach* based on [22]. The numerical multiplications were performed on CRS and JSA storage formats. In [12], based on [22], the *symbolic multiplication approach* was presented in detail by performing matrix multiplications on CRS, where both input matrices were on CRS, and the resulting matrix was on CRS using Java's native arrays.

The main objective in the use of the *symbolic approach* was to find the nonzero structure of the index array and the pointer array for CRS (only the index array for JSA), then store this information in temporary arrays and expand this structure for each row created. The row pointer array can be created easily since the dimension of the resulting matrix is known. We must have an index array that holds the indexes for a new row, and to add it to the index array that holds all the rows previously created. A new index array is then created to include the original rows and the new row. We copied both the original index rows and the newly created row to the new temporary array. This operation must be carried out for each new row created.

After creating the index array and the pointer array was initialised, the numerical multi-

plication [12] was then performed successfully as the already created index array can determine the size of the value array for the resulting matrix. The large number of modifications required to be made to the index array was the most time consuming aspect of performing the symbolic multiplication approach. This updating process can be done in Java, either by a for-loop method or by using the *arraycopy( )* method. We chose the *arraycopy( )* method to update the index array for implementation. For the use of the *arraycopy( )* method in Java, see [12, 19].

The implementation of both symbolic and numerical multiplication approaches for CRS and JSA in Java (adapted from [12]), are outlined in Appendixes B.12-B.15.

4.10 Banded Matrix Multiplication Algorithm on DIAS

When the matrices have banded structures, the matrix multiplication by diagonals is really useful. The theorem that we implemented in the banded matrix multiplications by diagonals is based on a 1976 paper by Niel K. MADSEN et al [20].

Suppose $A = (a_{-L_1}, \ldots, a_0, \ldots, a_{U_1})$ and $B = (b_{-L_2}, \ldots, b_0, \ldots, b_{U_2})$ have banded structures of size n, and L_1, L_2 are lower diagonals or lower bandwidth, U_1, U_2 are upper diagonals (or upper bandwidth) of matrices A and B respectively. Then the result matrix $C = AB$ can be described by $C = (c_{-L_3}, \ldots, c_0, \ldots, c_{U_3})$. Where the lower and upper diagonals of C can be found by

$$L_3 = min(n-1, L_1 + L_2) \tag{4.17a}$$

$$U_3 = min(n-1, U_1 + U_2) \tag{4.17b}$$

For banded matrix multiplication case, theorem 1 presented in section 4.5 can be generalised to the following corollary.

Corollary 1: This is a corollary of Theorem 1

If the matrices A and B have banded structures as described above, then the result matrix $C = AB$ can be computed by the following:

Let $c_k = 0$ for all k, then

(i) For $k = 0 : U_3$ compute c_k by

$$c_k(i-k;) = c_k(i-k;) + a_{k-i}(;k)b_i$$

$$\text{for } i = k+1 : min(U_2, k+L_1),$$

$$c_k(;i-k) = c_k(;i-k) + a_i b_{k-i}(k;)$$

$$\text{for } i = k+1 : min(U_1, k+L_2),$$

$$c_k = c_k + a_i(;k-i)b_{k-i}(i;)$$

$$\text{for } i = max(0, k-U_2) : min(k, U_1),$$

(ii) for $k = 1 : L_3$ compute c_{-k} by

$$c_{-k}(i-k;) = c_{-k}(i-k;) + a_{-i}b_{i-k}(;k)$$

$$\text{for } i = k+1 : min(L_1, k+U_2),$$

$$c_{-k}(;i-k) = c_{-k}(;i-k) + a_{i-k}(k;)b_{-i}$$

$$\text{for } i = k+1 : min(L_2, k+U_1),$$

$$c_{-k} = c_{-k} + a_{-i}(k-i;)b_{i-k}(;i)$$

$$\text{for } i = max(0, k-L_2) : min(k, L_1).$$

The algorithm of matrix multiplication by diagonals is much more efficient for narrow banded matrices. In the implementation, the multiplication of two matrices A and B of size n is considered to be having the same bandwidth $L_1 = L_2 = U_1 = U_2 = p$, where p is assumed to be small relative to n so that $C = AB$ is also a banded matrix with $L_3 = U_3 = 2p$. To compute c_k of C in diagonal multiplication algorithm, it requires $2p+1-k$ vector multiplications and additions. The time complexity is $O(n(2p+1)^2)$.

The steps required to compute the k^{th} diagonal c_k of C for dense matrices multiplication by diagonals outlined in Section 4.5 can be used to compute the k^{th} diagonal c_k of C, where $L_3 \leq k \leq U_3$ in band matrix.

The computing steps for band matrices are also conducted by three functions as following:

1. *getDiagArrayFunction* is presented in Section 4.5 Algorithm 4
2. *cutElementFunction* is presented in Section 4.5 Algorithm 5
3. *BandAB_multiplydiag* is a main function which is to correlate with the other two functions to complete the overall band matrix multiplication process. See Appendix B.21 for the complete Java code.

4.11 Cache Miss Analysis for Matrix Multiply

In matrix multiplication performance, the order of three for loops (i, j, k) can have a considerable impact on practical performance due to the memory access patterns and cache use of the algorithm. The most time efficient order also depends on whether the matrices are stored in row-major order, column-major order, or a mix of both [26]. To illustrate the importance of locality, we considered six different versions of the SMM and DMM using the example of matrix multiplication $C = AB$ in Equation 4.2 and matrix transpose multiplication in Equation 4.3.

For the purpose of analysis the behaviour of the innermost loop iterations, the assumptions are as following [4]:

- $n \times n$ matrix elements are *double*
- Linesize is 8 words
- Cache size is not even big enough to hold multiple rows

In computer memory, a matrix stored in row-major order, the rows of the matrix are contiguous and the columns are discontiguous. Let's take the $V1\_ijk$ form as an example to analyse the cache miss per inner loop iteration for matrix A, B and C in the matrix multiplication Algorithm 3. Figure 4.1 shows the access pattern of matrices elements in the inner loop.

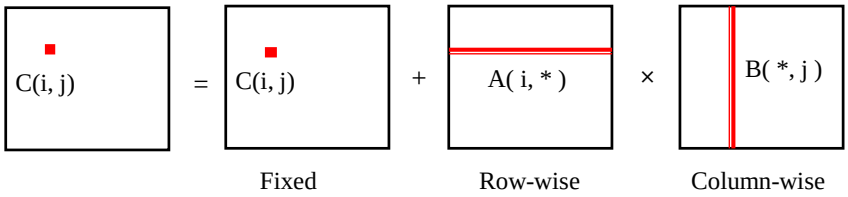

Figure 4.1: Access pattern of inner loop

Matrix A: When i and j are fixed, k is varied, row-i is accessed, there is $1/8$ misses per inner loop iteration.

Matrix B: When i and j are fixed, k is varied, elements in the columns of B are accessed. There is 1 miss per inner loop iteration.

Matrix C: There is temporal reuse for C in the inner loop iteration, therefore it has 0 miss.

In matrix multiplication by diagonals, the elements of matrix A and B are stored in the one-dimensional arrays by their diagonals order. When an element of matrix A and B is accessed with a stride 1, the miss rate for A is 1/8, and the miss rate for B is also 1/8. The miss rate for matrix C is 1/8 as its each element is formed with stride 1 by diagonal order in one-dimensional array.

Table 4.2 shows the results of rate of cache misses for six versions of SMM (based on [4]) and DMM.

Table 4.2: Cache miss analysis of matrix multiplication Equation (4.2)

Matrix multiply	Loads	Stores	A miss	B miss	C miss	Total misses
$V1\_ijk$ & $V2\_jik$	2	0	0.125	1.00	0.00	1.125
$V3\_jki$ & $V4\_kji$	2	1	1.00	0.00	1.00	2.00
$V5\_kij$ & $V6\_ikj$	2	1	0.0	0.125	0.125	0.25
diagonals	2	1	0.125	0.125	0.125	0.375

Here are the results of rate of cache misses for transpose matrix multiplications for the inner most-loop in each iteration in Table 4.3.

Table 4.3: Cache miss analysis of transpose matrix multiplication Equation (4.3)

Transpose Matrix multiply	Loads	Stores	A miss	B miss	C miss	Total misses
$V1\_ijk$ & $V2\_jik$	2	0	1.00	1.00	0.00	2.00
$V3\_jki$ & $V4\_kji$	2	1	0.125	0.00	1.00	1.125
$V5\_kij$ & $V6\_ikj$	2	1	0.00	0.125	0.125	0.25
diagonals	2	1	0.125	0.125	0.125	0.375

Important findings in tables 4.2 and 4.3,

1. A miss is the rate of cache misses when accessing an element of A.

2. When an element of array A is accessed by a row-order with a stride 1, the miss rate for A is 0.125 misses per iteration.

3. If an element of array A is accessed by a column-order with n strides, the miss rate for A is 1.00 misses per iteration.

4. The order of accessing a data in an array affects the running time of matrix multiplication.

Chapter 5

Computational Experiments

In this chapter, we present the numerical results of different storage formats on matrix multiplication routines. First, we will give the details of test data sets and this will be followed by outlining benchmarking and the test environment on which we ran our experiments.

5.1 Test Data Sets

In this thesis, we only consider the operations on real matrices. The entries for the (*full and sparse*) matrix and vector were generated randomly by using `(double) (Math.round (Math.random()*1000))/1000` to get 3 digit decimal precision in Java, where the random number generator `Math.random()` was used to obtain a random `double` value between 0.001 and 1.000 (excluding 1.000) to populate the matrix and vector [19]. All the matrices were square, with the same number of rows and columns, and non-symmetric.

- Dense matrix

The input matrix was initially generated in a two-dimensional format. The matrix entries were subsequently stored in a one-dimensional array with a row-wise order for the standard matrix multiplication. However, for diagonal multiplication, although the entries were stored in a one-dimensional array, they were arranged in a *(main_ super_ sub)* diagonal order.

- Band matrix

The input matrix was initially generated in a *(sub_ main_ super)* diagonal order in a one-dimensional array using a predetermined bandwith. Subsequent to this, the diagonal storage format was converted to CRS and JSA storage format using the format conversion algorithm in Appendix B.8. In the computational experiments, all band matrices were generated as having approximately 0.19 - 0.20% of non zero elements in the matrix.

5.2 Benchmarking and Test Environment

The execution time performance measurements for matrix multiplication routines on different storage formats were determined by benchmarking. Execution times of matrix multiplication routines were measured using `System.currentTimeMillis()` which returns the current time in milliseconds [8]. They are stored in variables of type `double`. The program was run multiple times to get the average execution time. All the benchmark results only consider the timings of the for-loop from where it starts, to where it stops. The test platform technical data information used to perform the benchmarking is shown in Table 5.1.

Table 5.1: Test platform technical data information

Model name	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
Operating system	CentOS Linux release 7.3.1611 (Core)
JDK version	OpenJDK 64-Bit Server VM (build 25.121-b13, mixed mode)
L1d and L1i cache	32KB
L2 cache	256KB
L3 cache	8192KB
CPU MHz	3400.132

5.3 Input Data Types

In Java and most other languages, a variable has a types that indicates what sort of data it can hold. As we mentioned in section 2.1.2, Java has two data types, one is **primitive data types**, the other is **object data type**. The primitive data types were chosen as an input data type in the computational experiments.

5.4 Challenges associated with running the algorithms used in this project

When the array size is increased beyond a certain size as indicated in the test result figures, the program is terminated and the following error message is given:

- `(java.lang.OutOfMemoryError: Java heap space)`. This error message arises due to JVM being unable to allocate memory in heap, and the garbage collection is unable to reclaim memory [5].

5.5 Computational Experiment

5.5.1 Introduction

In the computational experiment, we set up two computational models to demonstrate basic linear algebra routines (BLAS) on different storage formats described in Chapter 2.

BLAS are routine that provide the standard computing building blocks to perform basic matrix operations [1]. It has three levels:

- Level 1 BLAS perform scalar, vector and vector-vector operations.
- Level 2 BLAS perform matrix-vector operations.
- Level 3 BLAS perform matrix-matrix operations.

The BLAS are efficient, portable, and widely available, and they are commonly used in the development of high quality linear algebra software such as Linear Algebra Package

(LAPACK).

In the first computational model, we will implement a straightforward matrix multiplication routine that takes the row-wise layout of a two-dimensional array into consideration. Then we will focus on diagonal multiplication routine and compare it to the straightforward routine on the basis of performance speed for dense matrix.

Model 1 : Dense matrix operations

- Ax and $A^T x$ on SMM versus DMM
- AB and $A^T B$ on six versions of SMM versus DMM

In second computational model, we will implement three different sparse matrix storage formats on matrix multiplication routines and compare them each other on the basis of performance speed for banded matrix.

Model 2 : Banded matrix operation

- Ax on CRS, JSA, versus DIAS.
- $A^T x$ on CRS, JSA, versus DIAS.
- AB on CRS, JSA, versus DIAS.
- $A^T B$ on CRS, JSA, versus DIAS.

5.5.2 Model 1

The results of performing straightforward method versus diagonal multiplications on Ax and $A^T x$ is presented in Figure 5.1.

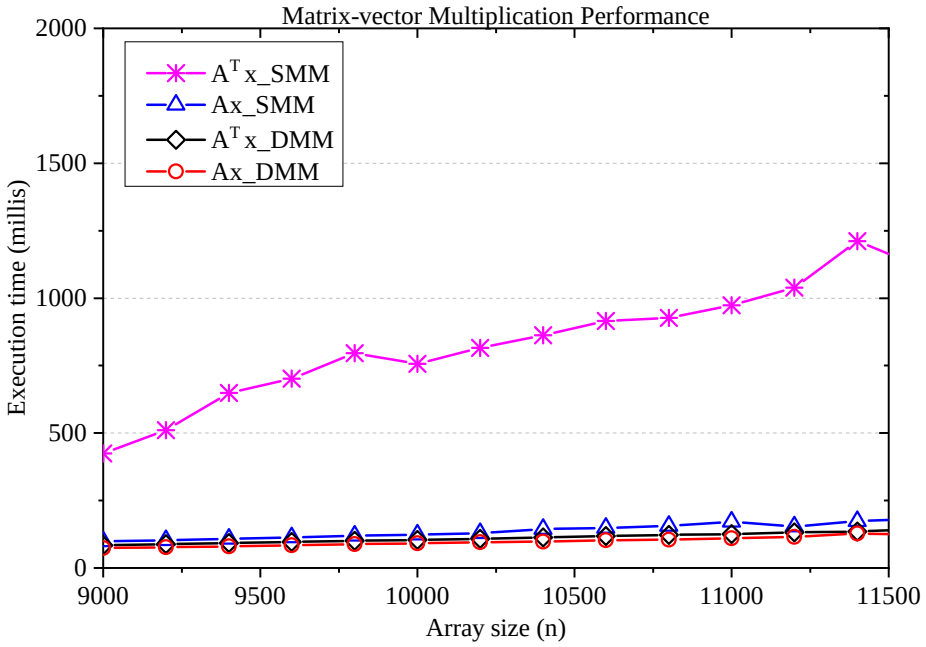

Figure 5.1: The SMM versus DMM on Ax and A^Tx

The result shows that SMM method is performing well on Ax, but its performance was not very good on A^Tx. Diagonal multiplication was more efficient than SMM.

The results of performing straightforward method versus diagonal multiplications on AB and A^TB are presented in Figure 5.2 and 5.4 respectively.

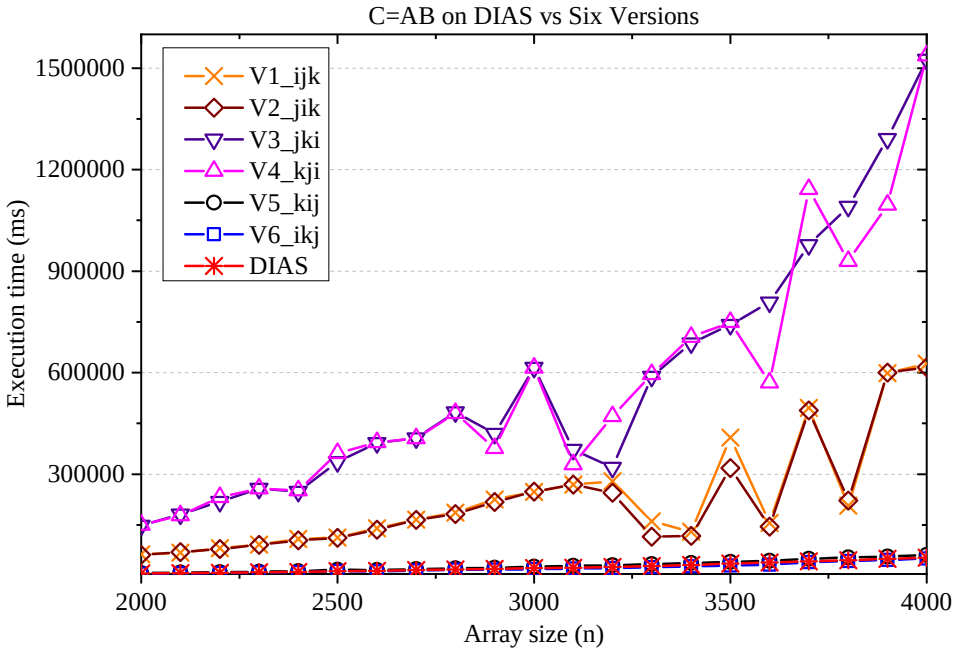

Figure 5.2: The DIAS versus six versions on AB

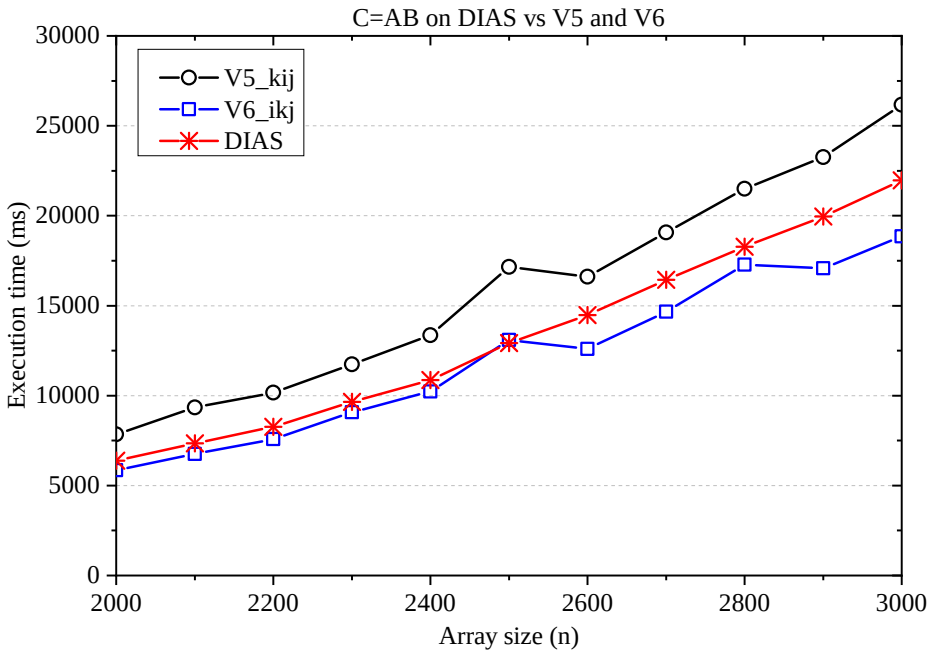

Figure 5.3: The DIAS versus $V5\_kij$ and $V6\_ikj$ on AB

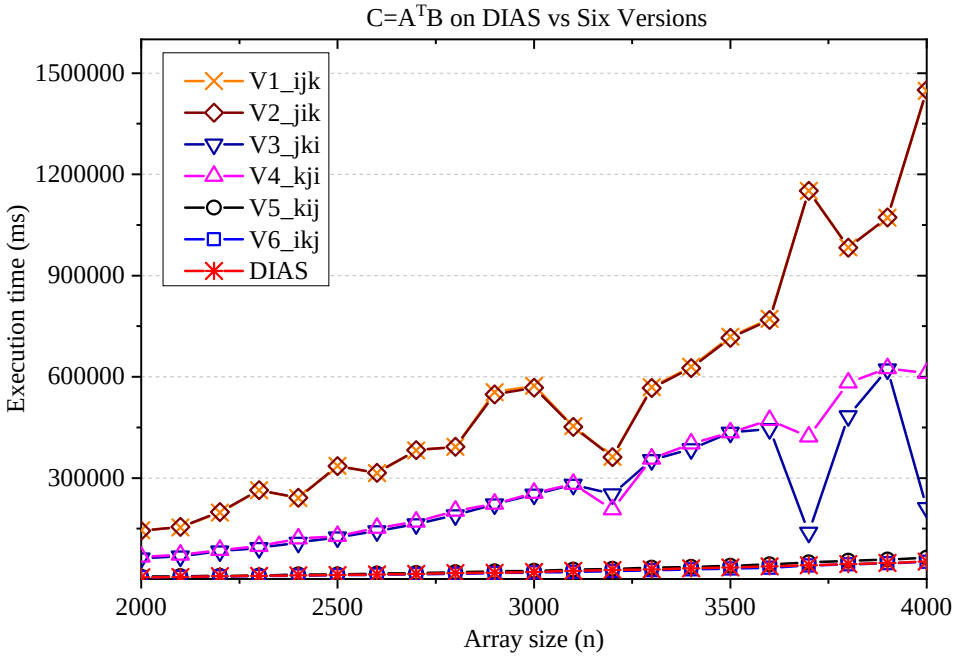

Figure 5.4: The DIAS versus six versions on $A^T B$

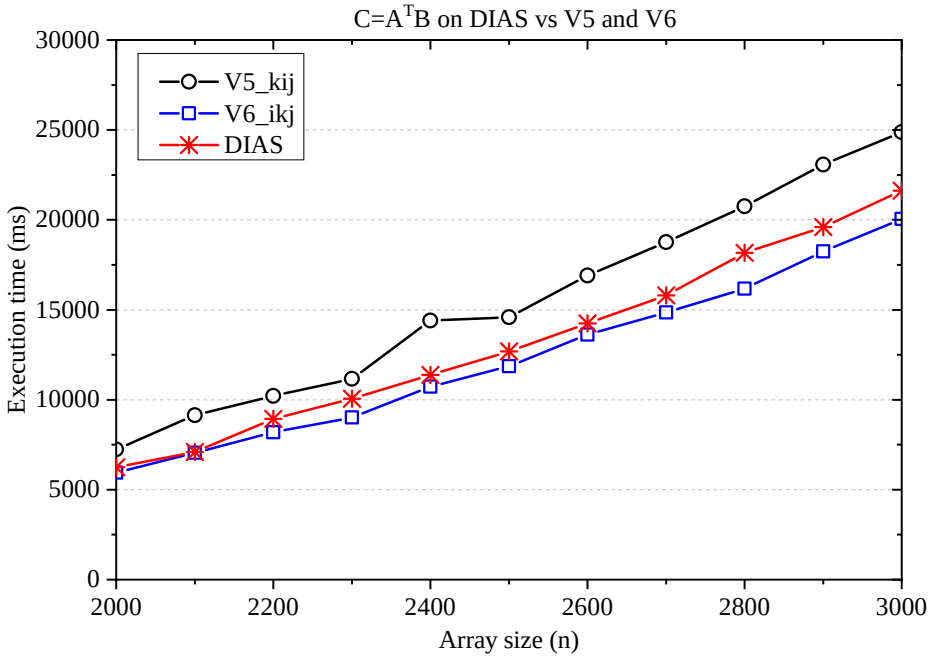

Figure 5.5: The DIAS versus $V5\_kij$ and $V6\_ikj$ on $A^T B$

From the results shown in Figures 5.2, 5.3, 5.4 and 5.5, we observed that:

- The run time changes in a cubic $O(n^3)$ order when the array size increases.
- Accessing consecutive elements in a row is faster than accessing consecutive elements in a column.
- The execution times for matrix multiplications and transpose matrix multiplications for diagonals are fairly similar.
- Data cache miss is a significant time efficiency issue when data is accessed by column order as it causes cache misses.
- It is clear to see that the *pure column-oriented* version is the least efficient, while the *pure row-oriented* version is the most efficient implementations. This is due to the need to access different object arrays when traversing columns as opposed to accessing the same object array several times. The *pure row-oriented* version does not traverse the columns of any matrices involved.
- The *pure row-oriented* version with loop- order (i,k,j) is more efficient than the version with loop-order (k,i,j), as the latter traverses the columns of matrix A in the loop-body of the second for-loop.
- Diagonal multiplication algorithm is more efficient than the *pure column-oriented* version and the *partial row/column-oriented* version in both matrix product and matrix transpose product.

5.5.3 Model 2

The results of performing CRS, JSA and DIAS on Ax and A^Tx are presented in Figure 5.6 and 5.7 respectively.

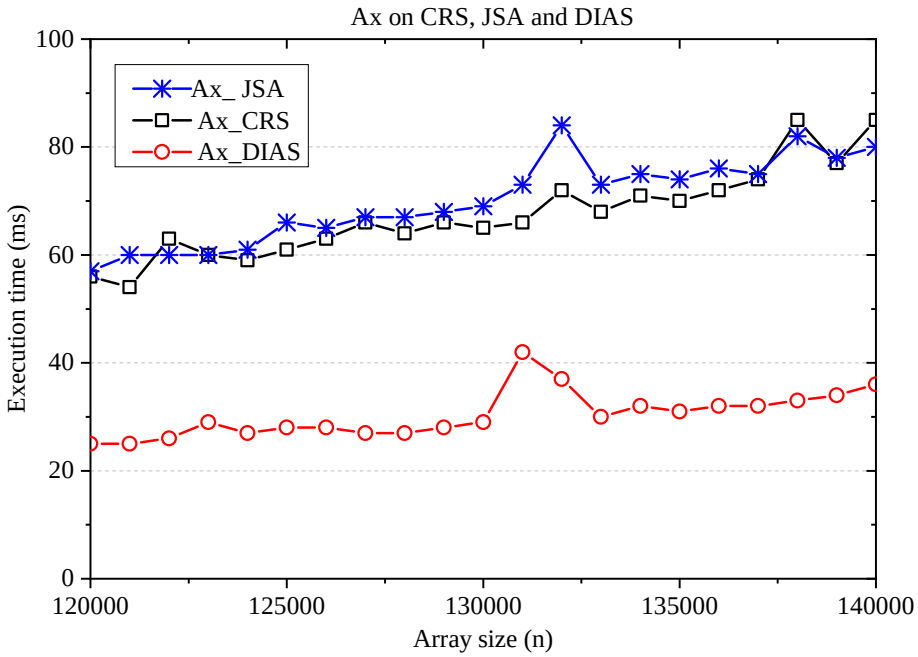

Figure 5.6: Banded matrix-vector multiplication on CRS, JSA and DIAS

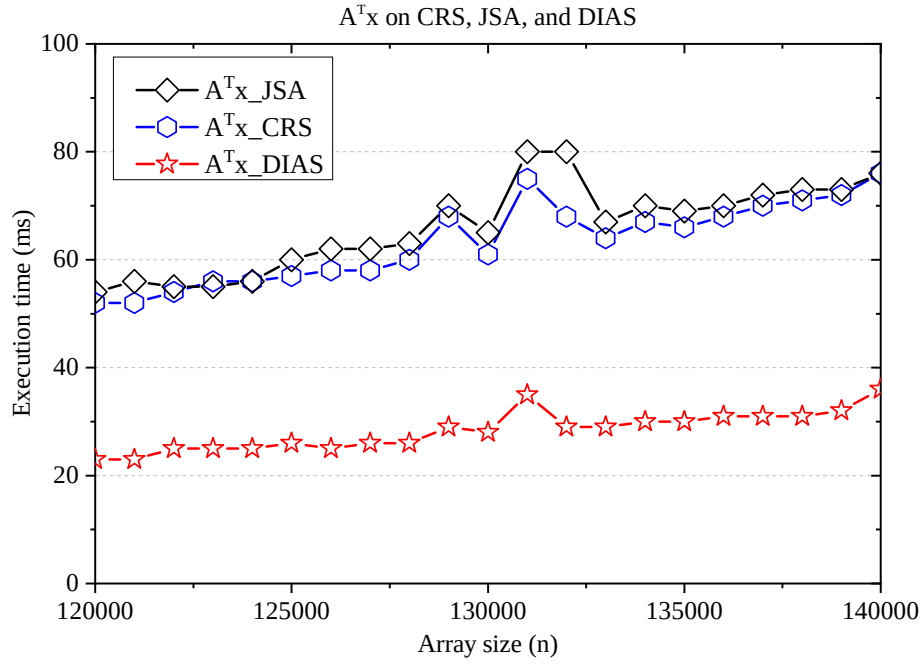

Figure 5.7: Banded matrix transpose -vector multiplication on CRS, JSA and DIAS

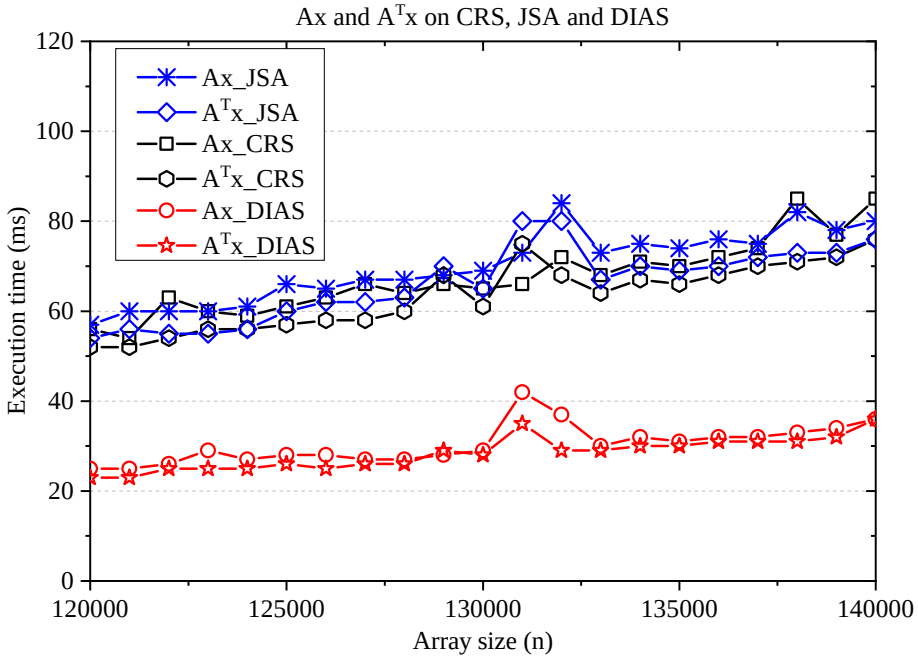

Figure 5.8: Banded matrix Ax and $A^T x$ routines on CRS, JSA and DIAS

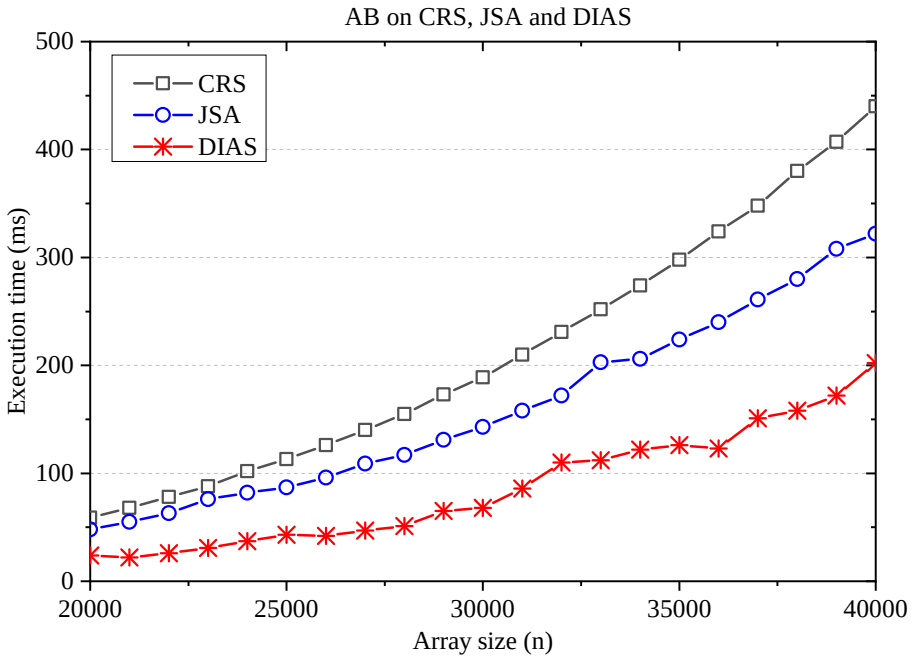

Figure 5.9: *AB* on CRS, JSA and DIAS, where numerical approach [12] was used for CRS and JSA, the total number of nonezero values are 0.19%.

The comparison between the numerical approach and algorithms (CRS), (JSA) applied

on AB is shown in Figure 5.10, where CRS' and JSA' indicate the results of Algorithms B.16 and B.19 in Appendix B. The total number of nonezero values are 0.19%.

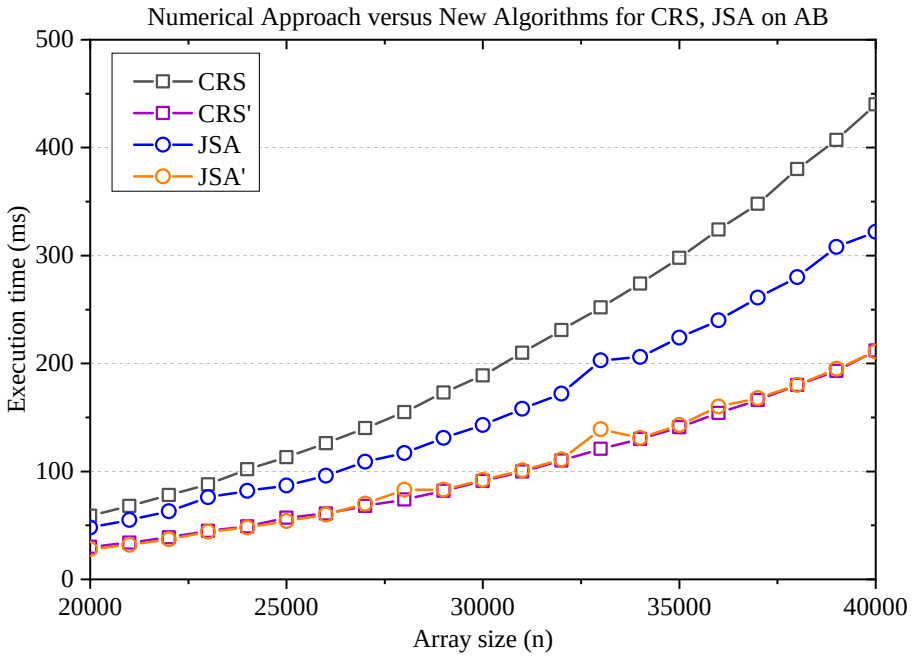

Figure 5.10: The numerical approach versus Algorithms B.16 and B.19 applied on CRS and JSA on AB

Figure 5.11 shows the matrix transpose multiplication performance on CRS, JSA and DIAS.

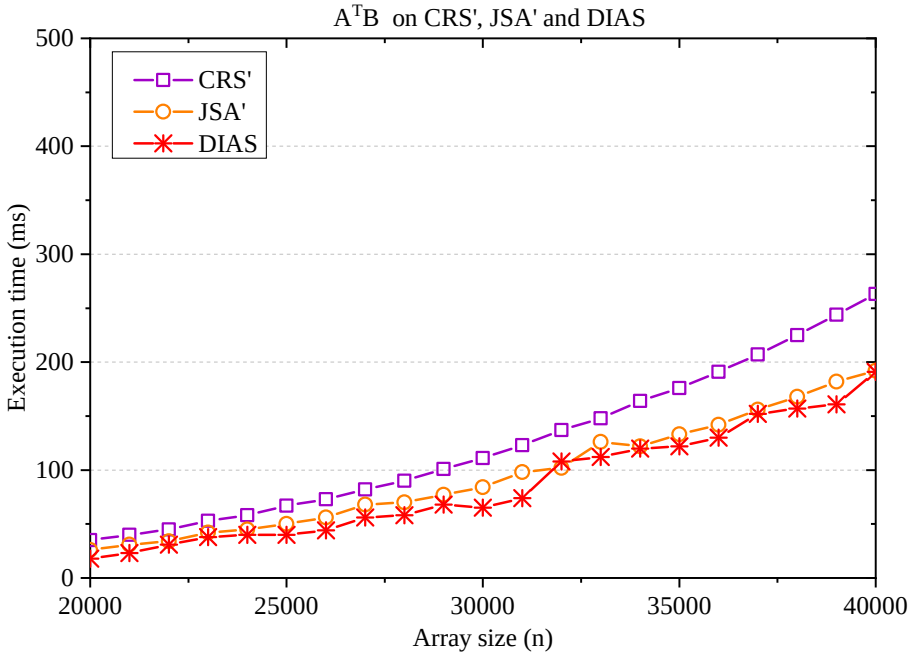

Figure 5.11: $A^T B$ on CRS', JSA' and DIAS

From the results shown in Figures 5.6 - 5.11 for the computational model 2, we observed that:

- Implementing a matrix-vector multiplication and matrix transpose -vector multiplication on CRS, JSA and DIAS format was rather straightforward since the structure or size of the resulting vector is known before calculation.

- We can see that DIAS on AB has similar efficiency as on $A^T B$. The reason for this is that multiplication starting in main-super-sub diagonal order is the same as starting in main-sub-super diagonal order. The diagonal orders do not affect the performance since we used indexing formulas to extract the diagonal elements.

- The DIAS format performed well both in banded matrix-vector operations and banded matrix-matrix operations compared to CRS and JSA formats. The time complexity of band matrices multiplication by diagonals is $O(n(2p+1)^2)$.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we performed an experimental study of a subset of BLAS operations on existing sparse matrix structures including compressed row storage, Java sparse array and provided implementations of diagonal storage that can be used for both dense and structured matrices in Java. The results of computational experiments show that:

- Data cache miss is a significant time efficiency issue when data is accessed by column order as it causes cache misses.
- The matrix transpose operations had not previously been researched in depth in Java. This project attamps to fill this void.
- The diagonal storage format performed well on the matrix operations. The execution times for matrix multiplication and transpose matrix multiplications are fairly similar. The diagonal storage format is shown to be a viable alternative for dense and structured matrices.

6.2 Future Research Needs

There are a number of interesting extensions of the work presented in this thesis.

- One possible extensions is to implement the diagonal storage formats on many and multi-core architecture. Specifically, the diagonal storage format can be implemented on many-core computing systems with Compute Unified Device Architecture (CUDA).

Currently, there does not exist an implementation of level-3 BLAS for banded matrix in CUDABLAS.

- Another interesting extension is to take general sparse diagonally structured matrices and three-dimensional tensors into account. The possibility is to study a band matrix which is diagonally structured where the diagonals are non-contiguous.

Bibliography

[1] BLAS (Basic Linear Algebra Subprograms). *Online: http://www.netlib.org/blas/old-index.html*, Accessed time: 2018-07-21.

[2] Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, volume 11. SIAM, 2000.

[3] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[4] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition, 2010.

[5] John Dean and Ray Dean. *Introduction to Programming with Java: A Problem Solving Approach*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2008.

[6] Jack Dongarra, Andrew Lumsdaine, Xinhui Niu, Roldan Pozo, and Karin Remington. Sparse matrix libraries in c++ for high performance architectures. 12 1996.

[7] Iain S Duff, Albert M Erisman, and John K Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Inc., New York, NY, USA, 1986.

[8] David J Eck. Programming: Introduction to programming using java. 2009.

[9] John R Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[10] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[11] James Gosling and Henry McGilton. The java language environment. *Sun Microsystems Computer Company*, 2550, 1995.

[12] Geir Gundersen. The use of java arrays in matrix computation. *Candidatus Scientarium (Master in Science) Thesis*, 2002.

[13] Geir Gundersen and Trond Steihaug. Data structures in java for matrix computations. *Concurrency and computation: Practice and Experience*, 16(8):799–815, 2004.

[14] Geir Gundersen and Trond Steihaug. On the use of java arrays for sparse matrix computations. In *Advances in Parallel Computing*, volume 13, pages 119–126. Elsevier, 2004.

[15] Joe Hicklin, Cleve Moler, Peter Webb, Ronald F. Boisvert, Bruce Miller, Roldan Pozo, and Karin Remington. JAMA: A Java Matrix Package. *Online: https://math.nist.gov/javanumerics/jama*, November 2012.

[16] Computer Hope. Free computer help since 1998- clock cycle. *Online: https://www.computerhope.com/jargon/c/clockcyc.html*, Accessed time: 2018-07-15.

[17] Eun-Jin Im. Optimizing the performance of sparse matrix-vector multiplication. *Computer Science*, 2000.

[18] The Matrix Market is a service of the Mathematical, Computational Sciences Division of the Information Technology Laboratory of the National Institute of Standards, and Technology. Matrix market vision 3.0, glossary. *Online: https://math.nist.gov/MatrixMarket/glossary.html*, Accessed time:2018-05-08.

[19] Y. Daniel Liang. *Introduction to Java Programming and Data Structures, Comprehensive Version, Student Value Edition Plus MyProgrammingLab with Pearson eText - Access Card Package (11th Edition)*. Pearson, 11 edition, 2017.

[20] Niel K. Madsen, Garry H. Rodrigue, and Jack I. Karush. Matrix multiplication by diagonals on a vector/parallel processor. *Information Processing Letters*, 5(2):41 – 45, 1976.

[21] Jos Moreira, Samuel Midkiff, and Manish Gupta. Supporting multidimensional arrays in java. 15, 03 2003.

[22] Sergio Pissanetzky. Introduction. In Sergio Pissanetzky, editor, *Sparse Matrix Technology*. Academic Press, 1984.

[23] Youcef Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2, 1994.

[24] Yousef Saad. *Iterative methods for sparse linear systems*, volume 82. siam, 2003.

[25] Guillaume Salagnac. Memory mountain-a script to get a 3D plot of your cache memory performance. *Online: https://github.com/guillaumesalagnac/memory-mountain*, 2015.

[26] Steven S Skiena. *The algorithm design manual: Text*, volume 1. Springer Science & Business Media, 1998.

[27] FS Smailbegovic, Georgi N Gaydadjiev, and Stamatis Vassiliadis. Sparse matrix storage format. 2005.

[28] Ivan P Stanimirović and Milan B Tasić. Performance comparison of storage formats for sparse matrices. *Facta universitatis-series: Mathematics and Informatics*, 24(1):39–51, 2009.

[29] G. W. Stewart. JAMPACK: A Package for Matrix Computations. *Online: ftp://math.nist.gov/pub/Jampack/Jampack/AboutJampack.html*, November 2012.

[30] Marta Stojanovic. Automatic Memory Management in Java. 07 2001.

[31] D.A. Watt and D.F. Brown. *Java collections: an introduction to abstract data types, data structures, and algorithms*. John Wiley, 2001.

Appendix A

Tables of Experiment Results

Most of the details of the computational results associated with Chapter 4 and Chapter 5 have been placed in this appendix. All execution times are in milliseconds. In all experiments, the matrix size was initially started at a predetermined number (ie,. n=5000) and then increased in size by steps (ie,. 200, or 1000) until we received the error message as: *java.lang.OutOfMemoryError: Java heap space*.

A.1 Performance of Dense Matrix-Vector Multiplications

Table A.1: Dense matrix vector multiplication performance in 1D (ie,. row-wise layout

Matrix size (n)	Ax	Ax_diag	A^Tx	A^Tx_diag
9000	99	74	225	84
9200	103	77	998	88
9400	108	80	261	92
9600	113	84	843	97
9800	119	88	266	101
10000	124	91	383	104
10200	129	95	848	108
10400	134	98	923	113
10600	138	103	917	118
10800	146	105	734	122
11000	151	110	438	126
11200	154	115	477	132
11400	163	128	963	135
11600	183	124	1240	144
11800	173	128	1223	146
12000	177	132	1124	155
12200	181	133	1399	151

where the matrix elements were accessed in one-dimensional arrays. We increased the matrix size from n=5000, ..., till heap the Java space *Java heap space* in steps of 200. In

the table we only presented the performance results from n=9000, . . . , and higher.

Table A.2: Dense matrix vector multiplication performance in 2D (ie,. two-dimensional layout

Matrix size (n)	Ax	Ax_diag	A^Tx	A^Tx_diag
9000	99	446	340	462
9200	104	870	784	877
9400	109	486	375	498
9600	112	486	381	504
9800	120	511	396	526
10000	123	1301	1216	1304
10200	130	888	796	899
10400	132	1065	964	1073
10600	139	981	875	994
10800	145	1557	1475	1565
11000	150	644	509	664
11200	155	673	508	697
11400	163	1614	1415	1635
11600	177	1568	1464	1580
11800	177	1806	1558	1815
12000	182	3487	3445	3490
12200	184	1769	1681	1783

where the matrix elements were accessed in two-dimensional arrays. We increased the matrix size from n=5000, . . . , till heap the Java space *Java heap space* in steps of 200. In the table we only presented the performance results from n=9000, . . . and higher.

A.2 Performance of Banded Matrix-Vector Multiplications

Table A.3: Banded matrix vector multiplication performance.

Matrix size (n)	Bandwidth	nnz %	Ax			$A^T x$		
			CRS	JSA	Diag	CRS	JSA	Diag
125000	249	0.18	61	66	28	57	60	26
126000	251	0.18	63	65	28	58	62	25
127000	253	0.18	66	67	27	58	62	26
128000	255	0.18	64	67	27	60	63	26
129000	257	0.18	66	68	28	68	70	29
130000	259	0.18	65	69	29	61	65	28
131000	261	0.18	66	73	42	75	80	35
132000	263	0.18	72	84	37	68	80	29
133000	265	0.18	68	73	30	64	67	29
134000	267	0.18	71	75	32	67	70	30
135000	269	0.18	70	74	31	66	69	30
136000	271	0.18	72	76	32	68	70	31
137000	273	0.18	74	75	32	70	72	31
138000	275	0.18	85	82	33	71	73	31
139000	277	0.18	77	78	34	72	73	32
140000	279	0.18	85	80	36	76	76	36
141000	281	0.18	80	80	34	76	76	32
142000	283	0.18	92	92	37	76	75	37
143000	285	0.18	82	85	35	77	79	33

We increased the matrix size in steps of 1000,bandwidth in steps of 2, to maintain a total number of nonzero values at approximately 0.18% in a matrix, with n=100000, ..., until we received the error message as *java.lang.OutOfMemoryError: Java heap space*. In the table we only presented the performance results from n=125000, ..., and higher.

A.3 Performance of Banded Matrix-Matrix Multiplications

Table A.4: Banded matrix-matrix multiplication performance (Numerical Approach for CRS, JSA).

Matrix size (n)	Bandwidth	nnz %	*AB*		
			CRS	JSA	Diag
25000	49	0.196	113	87	43
26000	51	0.196	126	96	42
27000	53	0.196	140	109	58
28000	55	0.196	155	117	62
29000	57	0.196	173	131	75
30000	59	0.196	189	143	76
31000	61	0.196	210	158	96
32000	63	0.196	231	172	115
33000	65	0.196	252	203	112
34000	67	0.196	274	206	122
35000	69	0.196	298	224	146
36000	71	0.197	324	240	123
37000	73	0.197	348	261	151
38000	75	0.197	380	280	181
39000	77	0.197	407	308	186
40000	79	0.197	440	322	202
41000	81	0.197	482	356	205
42000	83	0.197	505	373	244

We increased the matrix size in steps of 1000,bandwidth in steps of 2 to maintain a total number of nonzero values at approximately 0.19% in the matrix. In the table we only presented the performance results from n=25000. The numerical approach from [12] was used when implementing CRS, JSA on *AB*.

Table A.5: Banded matrix-matrix multiplication performance.

Matrix size (n)	Bandwidth	nnz %	AB			A^TB		
			CRS	JSA	Diag	CRS	JSA	Diag
25000	49	0.196	57	54	43	67	50	40
26000	51	0.196	61	60	42	73	56	44
27000	53	0.196	68	70	58	82	68	66
28000	55	0.196	74	83	62	90	70	58
29000	57	0.196	82	83	75	101	77	68
30000	59	0.197	91	92	76	111	84	65
31000	61	0.197	100	101	96	123	98	74
32000	63	0.197	110	111	115	137	102	108
33000	65	0.197	121	139	112	148	126	112
34000	67	0.197	130	131	122	164	122	120
35000	69	0.197	141	143	146	176	133	122
36000	71	0.197	154	160	123	191	142	130
37000	73	0.197	166	168	151	207	156	152
38000	75	0.197	180	180	181	225	168	157
39000	77	0.197	193	195	186	244	182	161
40000	79	0.197	212	211	202	263	192	191
41000	81	0.197	224	226	205	281	214	221
42000	83	0.198	240	242	244	301	225	230

We increased the matrix size in steps of 1000,bandwidth in steps of 2 to maintain a total number of nonzero values at approximately 0.19% in the matrix. In the table we only presented the performance results from n=25000. Algorithms B.16 and B.19 were used when implementing CRS, JSA on AB and A^TB.

Table A.6: Numerical approach versus Algorithm (B.16) for CRS and Algorithm (B.19) for JSA.

Matrix size (n)	Bandwidth	nnz %	CRS		JSA	
			Numerical	Algorithm(B.16)	Numerical	Algorithm(B.19)
25000	49	0.196	113	57	87	54
26000	51	0.196	126	61	96	60
27000	53	0.196	140	68	109	70
28000	55	0.196	155	74	117	83
29000	57	0.196	173	82	131	83
30000	59	0.197	189	91	143	92
31000	61	0.197	210	100	158	101
32000	63	0.197	231	110	172	111
33000	65	0.197	252	121	203	139
34000	67	0.197	274	130	206	131
35000	69	0.197	298	141	224	143
36000	71	0.197	324	154	240	160
37000	73	0.197	348	166	261	168
38000	75	0.197	380	180	280	180
39000	77	0.197	407	193	308	195
40000	79	0.197	440	212	322	211
41000	81	0.197	482	224	356	226
42000	83	0.198	505	240	373	242

Appendix B

Java Codes of the Implementations

In this Appendix we show the classes and methods used to implement four basic linear algebra routines using several storage formats. These classes stores only $n \times n$ square matrices. For some routines, we have provided explanatory comments pertaining to the code. If not, the routines are small and rather self explanatory. Overall the meaning of methods and the implemented routines should be rather clear for the reader who has some experiences with Java code and these routines. Java codes were written early in the work on this thesis, therefore some names of variables, routines may suffer from that on the basis of consistent notation.

Here is a list of Java packages that we used in the implementation:

```
java.util.Arrays;
java.lang.Object;
java.util.Random;
java.io.OutputStream;
java.io.FileOutputStream;
java.io.PrintStream;;
java.text.DecimalFormat;
```

Listing B.1: Java packages list

B.1 Code for Model 1: Dense Matrix Operations

B.1.1 The matrix-vector multiplication routines

```
// Method: Matrix- vector multiplication by diagonals
public static double multiply_diagAx(int n, double[] valA, int[] diag,
   double[] x ) {
   double [] y= new double[n];
   int j,i;
   double startTime = System.currentTimeMillis();
// A's main and super-diagonals elements multiply with vector x
    for (int d=0; d<diag.length; d++){
        int k=diag[d];
        if (k>=0){
            j=0;
            i=k;
            int startindex=k*n-k*(k-1)/2;
            int stopindex=startindex+n-k-1;
             for(; startindex<=stopindex; startindex++){
                y[j]+=valA[startindex]*x[i];
```

```
                    j++;
                    i++;
                }
            }
            else{  // A's sub-diagonals elements multiply with vector x
                int abs_k=Math.abs(k);
                j=abs_k;
                i=0;
                int startindex=n*(n+1)/2 + (abs_k-1)*n - abs_k*(abs_k-1)/2;
                int stopindex=startindex + n-abs_k-1;
                for(; startindex<=stopindex; startindex++){
                    y[j]+=valA[startindex]*x[i];
                    j++;
                    i++;
                }
            }
        }
        double endTime    = System.currentTimeMillis();
        double totalTime= endTime - startTime;
        return totalTime;
}
```

Listing B.2: Ax by diagonals

```
// Method: Matrix transpose- vector multiplication by diagonals
 public static double multiply_diagATx(int n, double[] valA, int[] diag,
 double[] x ) {
     double [] y= new double[n];
     int i,j;
     double startTime = System.currentTimeMillis();
     // A's main and super_diagonals elements multiply with vector x
     for (int d=0; d<diag.length; d++){
        int k=diag[d];
        if (k>=0){
            j=k;
            i=0;
            int startindex=k*n-k*(k-1)/2;
            int stopindex=startindex+n-k-1;
            for(; startindex<=stopindex; startindex++){
                y[j]+=valA[startindex]*x[i];
                j++;
                i++;
            }
        }
        else{
            int abs_k=Math.abs(k);
            j=0;
            i=abs_k;
            int startindex=n*(n+1)/2 + (abs_k-1)*n - abs_k*(abs_k-1)/2 ;
            int stopindex=startindex + n-abs_k-1;
            for(; startindex<=stopindex; startindex++){
                y[j]+=valA[startindex]*x[i];
                j++;
                i++;
             }
```

```
        }
    }
     double endTime    = System.currentTimeMillis();
     double totalTime= (endTime - startTime);
     return totalTime;
}
```

Listing B.3: $A^T x$ by diagonals

B.1.2 The matrix-matrix multiplication routines

```
//Method: Get the requested diagonal array
 public static double [] getDiagArrayFunction(double[] valueA, int k) {
    double[] getarray=new double[n-Math.abs(k)];
    if ( k >= 0 && k<=n-1){
      int startindx=k*n-k*(k-1)/2;
      int endindx=startindx + n-k-1;
      int index=0;
      for (int i=startindx; i<=endindx;i++){
           getarray[index]=valueA[i];
           index++;
      }
    }
   else if ( k <0 && k>=-(n-1)){
      int abs_k=Math.abs(k); // take the absolute value of k
      int startindx=n*(n+1)/2 + (abs_k-1)*n - abs_k*(abs_k-1)/2 ;
      int endindx=startindx + n-abs_k-1;
      int index=0;
     for (int i=startindx; i<=endindx;i++){
           getarray[index]=valueA[i];
           index++;
      }
    }
  return getarray;
}
```

Listing B.4: Get the requested diagonal array

```
//Method: Cut the elements from getDiagArray
public static double [] cutElementFunction(double[] getDiagArray, int
    cutBegin, int cutEnd) {
     int size = getDiagArray.length;
     double[] outputArray=new double [size-cutBegin-cutEnd];
     for (int i=0; i<size-cutBegin-cutEnd;i++){
        outputArray[i]=getDiagArray[i+cutBegin];
     }
 return outputArray;
 }
```

Listing B.5: Cut the elements from getDiagArray

```
//Method:Multiplying two matrices on DIAS format
public static double[][] AB_multiplydiag(int n,double[] diagStoreArrayA,
    double[] diagStoreArrayB){
double [] C = new double[n*n];
//getting main and super-diagonals of C
```

```
 for (int k=0; k<=n-1;k++){
    int inx_C = k*n-k*(k-1)/2 ;
    int endinx_C = inx_C + n-k-1;
 for (int i=k+1; i<=n-1;i++){
double[] diag_a1=cutElementFunction(getDiagArrayFunction(diagStoreArrayA
    ,k-i),0,k);
double[] diag_b1=cutElementFunction(getDiagArrayFunction(diagStoreArrayB
    ,i),0,0);
int size1=diag_b1.length;
    for (int j=0; j<=size1-1;j++){
        C[inx_C+i-k+j]= C[inx_C+i-k+j]+diag_a1[j]*diag_b1[j];
    }
double[] diag_a2=cutElementFunction(getDiagArrayFunction(diagStoreArrayA
    ,i),0,0);
double[] diag_b2=cutElementFunction(getDiagArrayFunction(diagStoreArrayB
    ,k-i),k,0);
int size2=diag_a2.length;
    for (int j=0; j<=size2-1;j++){
        C[inx_C+j]= C[inx_C+j]+diag_a2[j]*diag_b2[j];
    }
 }
 for (int i=0; i<=k; i++){
 double[] diag_a3=cutElementFunction(getDiagArrayFunction(
    diagStoreArrayA,i),0,k-i);
 double[] diag_b3=cutElementFunction(getDiagArrayFunction(
    diagStoreArrayB,k-i),i,0);
 int size3=diag_a3.length;
  for (int j=0; j<=size3-1;j++){
    C[inx_C+j]= C[inx_C+j]+diag_a3[j]*diag_b3[j];
    }
}
} //getting sub-diagonals of C
 for (int k=1; k<=n-1;k++){
  int inx_C = n*(n+1)/2 + (k-1)*n - k*(k-1)/2;
  int endinx_C = inx_C + n-k-1;
for (int i=k+1; i<=n-1;i++){
double[] diag_a1=cutElementFunction(getDiagArrayFunction(diagStoreArrayA
    ,-i),0,0);
double[] diag_b1=cutElementFunction(getDiagArrayFunction(diagStoreArrayB
    ,i-k),0,k);
 int size1=diag_b1.length;
 for (int j=0; j<=size1-1;j++){
  C[inx_C+i-k+j]= C[inx_C+i-k+j]+diag_a1[j]*diag_b1[j];
 }
double[] diag_a2=cutElementFunction(getDiagArrayFunction(diagStoreArrayA
    ,i-k),k,0);
double[] diag_b2=cutElementFunction(getDiagArrayFunction(diagStoreArrayB
    ,-i),0,0);
 int size2=diag_a2.length;
          for (int j=0; j<=size2-1;j++){
           C[inx_C+j]= C[inx_C+j]+diag_a2[j]*diag_b2[j];
          }
    }
for (int i=0; i<=k; i++){
```

```
49 double[] diag_a3=cutElementFunction(getDiagArrayFunction(diagStoreArrayA
    ,-i),k-i ,0);
50 double[] diag_b3=cutElementFunction(getDiagArrayFunction(diagStoreArrayB
    ,i-k),0,i);
51 int size3=diag_a3.length;
52   for (int j=0; j<=size3-1;j++){
53     C[inx_C+j]= C[inx_C+j]+diag_a3[j]*diag_b3[j];
54    }
55   }
56 }
57  return C;
58 }
```

Listing B.6: *AB* on DIAS

B.2 Code for Model 2: Sparse (Banded) Matrix Operations

B.2.1 The matrix-vector multiplication routines on CRS, JSA, DIAS

```
1 // Methods: Generate the random band matrix with bandwidth BW, and
    return nnz elements in 1D array called diagArray with column index (
    ie,. colind) array
2 public static Object[] RandomArray(int n, int[] diag){
3     int ii=0;
4     int nnz=0;
5     for(int i=0; i<diag.length; i++){
6       nnz+=n-Math.abs(diag[i]);
7     }
8     double[] diagArray= new double [nnz];
9     int [] colind=new int[nnz];
10 //value, column index array
11    for (int t=0; t<diag.length; t++){
12       for(int k=0; k<n-Math.abs(diag[t]); k++){
13          diagArray[ii]=(double)(Math.round(Math.random()*1000))/1000;
14          if (diag[t]<0){
15             colind[ii]=k;
16          }
17          else{
18            colind[ii]=k+Math.abs(diag[t]);
19          }
20          ii++;
21       }
22    }
23  Object[] objects=new Object[]{diagArray,colind };
24   return objects;
25 }
```

Listing B.7: Generate the random band matrix with bandwidth BW, and return nnz elements in one-dimensional array called diagArray with column index (ie,. colind) array

```
1 //Method: Convert diagonal storage format (ie,.diagArray) to CRS and JSA
     storage formats.
2  public static Object[] ConvertDiagToCRS_JSA(int n, double[]diagArray,
    int[]colind, int[] diag ){
3     int size=diagArray.length;
```

```
4       double[] CRSarray=new double[size];
5       int [] colind_CRS=new int[size];
6       int[] rowptr=new int[n+1];
7       double[][] JSAarray = new double[n][1];
8       double[] tempJSA=new double[n];
9       int[][] colind_JSA = new int[n][1];
10      int[] tempindex=new int[n];
11      int ii, jj=0;
12      int kk=0;
13      int x=0;
14      int index=0;
15      int[] newindex=new int[size];
16      int count=0;
17      rowptr[0]=0;
18    for (int r=0; r<n; r++){
19      int cnt=0;
20       for (int i=0; i<diag.length; i++){
21        int p=diag[i];
22        if(-r<=p && p<n-r){
23            if(p<0){
24              x=p;
25           }
26           if (p>=0){
27              x=0;
28           }
29           int alpha=x;
30           int sum=0;
31           for(int j=0; j<i;j++){
32            sum+=n-Math.abs(diag[j]);
33          }
34      ii=sum+r+alpha;
35      CRSarray[jj]=diagArray[ii];
36      colind_CRS[jj]=colind[ii];
37      tempJSA[cnt]=diagArray[ii];
38      tempindex[cnt]=colind[ii];
39      jj++;
40      cnt++;
41      }
42    }
43     count+=cnt;
44     rowptr[kk+1]=count;
45     kk++;
46     double [] jsa=new double[cnt];
47     int[] jsaindex = new int[cnt];
48     System.arraycopy(tempJSA, 0, jsa, 0, cnt);
49     System.arraycopy(tempindex, 0, jsaindex, 0, cnt);
50     JSAarray[r]=jsa;
51    colind_JSA[r]=jsaindex;
52  }
53    Object[] objects=new Object[]{CRSarray,rowptr,colind_CRS, JSAarray,
       colind_JSA};
54    return objects;
55  }
```

Listing B.8: Convert DIAS format to CRS storage format and JSA format

```
// Method: Multiplying a matrix with a vector on CRS
 public static double AxinCRS( double[] val, int[] col_ind, int []
    row_ptr, double [] x, int n){
 double [] y= new double[n];
 double startTime = System.currentTimeMillis();
    for (int i=0; i< n; i++) {
        for( int j=row_ptr[i]; j<=row_ptr[i+1]-1; j++){
             y[i]= y[i]+ val[j]* x[col_ind[j]];
        }
    }
    double endTime   = System.currentTimeMillis();
    double totalTime= (endTime - startTime);
    return totalTime;
}
```

Listing B.9: Banded matrix Ax on CRS

```
// Method: Multiplying a matrix with a vector on JSA
 public static double AxinJSA( double[][] val, int[][] index, double []
    x, int n){
    double [] y= new double[n];
    double startTime = System.currentTimeMillis();
      for (int i=0; i< n; i++) {
           double[] subarray=val[i];
           int []   subindex=index[i];
          for (int ii=0; ii<subarray.length; ii++){
              y[i]+=subarray[ii]*x[subindex[ii]];
           }
       }
     double endTime   = System.currentTimeMillis();
     double totalTime3= (endTime - startTime);
     return totalTime;
}
```

Listing B.10: Banded matrix Ax on JSA

```
// Method: Multiplying a matrix with a vector on DIAS
public static double AxinDiag( double[] val, int[] diag, double[] x, int
    n, int U, int L ) {
 double [] y= new double[n];
   int j,i;
    double startTime = System.currentTimeMillis();
     // A's main and super_diagonals elements multiply with vector x
    for (int d=0; d<diag.length; d++){
        int k=diag[d];
        if (k>=0){
            j=0;
            i=k;
            int startindex=L*(2*n-L-1)/2 + k*(2*n-k-1)/2 + k;
            int stopindex=startindex+n-k-1;
             for(; startindex<=stopindex; startindex++){
                y[j]+=val[startindex]*x[i];
                 j++;
                 i++;
             }
```

```
19          }
20          else{
21              int abs_k=Math.abs(k);
22              j=abs_k;
23              i=0;
24              int startindex=L*(2*n-L-1)/2 - abs_k*(2*n-abs_k-1)/2;
25              int stopindex=startindex + n-abs_k-1;
26               for(; startindex<=stopindex; startindex++){
27                  y[j]+=val[startindex]*x[i];
28                   j++;
29                   i++;
30               }
31          }
32      }
33       double endTime    = System.currentTimeMillis();
34       double totalTime= (endTime - startTime);
35       return totalTime;
36  }
```

Listing B.11: Banded matrix Ax on DIAS

B.2.2 The matrix-matrix multiplication routines on CRS, JSA, DIAS

```
1  //Method: Use the symbolic method to find column index and row pointer
       of C for CRS format.
2  public static Object[] Cindexpointer(double[] valueA, int[] indexA, int
       [] pointerA, double[] valueB, int [] indexB, int [] pointerB, int n){
3   int [] indexC = new int[n];
4   int [] pointerC=new int[n+1];
5   pointerC[0]=0;
6   int len = -1;
7   int lentemp = -1;
8   boolean test = true;
9   int temp[] = new int[n];
10  int[] indexTemp = null;
11   for(int i=0;i<n;i++){
12      temp[i]=-1;
13   }
14  for(int i = 0;i<n;i++){
15  int startrowa = pointerA[i];
16  int stoprowa =  pointerA[i+1]-1;
17     for(;startrowa<=stoprowa;startrowa++){
18        int jj = indexA[startrowa];
19          int startrowb = pointerB[jj];
20         int  stoprowb = pointerB[jj+1]-1;
21          for(;startrowb<=stoprowb;startrowb++){
22            int jcol=indexB[startrowb];
23            int jposition = temp[jcol];
24             if(jposition == -1){
25                  len++;
26                  lentemp++;
27                  indexC[lentemp]=jcol;
28                  temp[jcol]=len;
29                }
30             }
```

```
    }
for(int k=0; k<=lentemp; k++){
  temp[indexC[k]]=-1;
}
pointerC[i+1]=len+1;
 if(test){
  indexTemp=new int[len+1];
  System.arraycopy(indexC,0,indexTemp, 0, len+1);
  test=false;
 }
 else {
    if(lentemp>-1){
    int [] a= new int [len+1];
    System.arraycopy(indexTemp,0, a, 0, indexTemp.length);
    System.arraycopy(indexC, 0, a, indexTemp.length, lentemp+1);
    indexTemp=a;
     }
    }
   lentemp=-1;
   indexC=new int[n];
  }
 Object[] objectsC=new Object[]{indexTemp, pointerC};
 return objectsC;
}
```

Listing B.12: Precalculate the column indices and the row pointers matrix C for CRS format using the symbolic method, adapted from [12].

```
// Method: Multiplying two matrices on CRS format using numerical
    approach.
public static double ABinCRS_Numerical(double[] valueA, int[] indexA,
    int[] pointerA, double[] valueB, int [] indexB, int [] pointerB, int
    [] indexC, int [] pointerC, int n ){
   double[] valueC= new double[indexC.length];
   int len = -1;
   int temp[] = new int[n];
 for(int i=0;i<n;i++){
  temp[i]=-1;
 }
  double startTime1 = System.currentTimeMillis();
for(int i = 0;i<n;i++){
   int startrowa = pointerA[i];
   int stoprowa = pointerA[i+1]-1;
    for(;startrowa<=stoprowa;startrowa++){
         double scalar = valueA[startrowa];
         int jj = indexA[startrowa];
         int startrowb = pointerB[jj];
         int stoprowb = pointerB[jj+1]-1;
        for(;startrowb<=stoprowb;startrowb++){
          int jcol=indexB[startrowb];
          int jposition = temp[jcol];
          if(jposition == -1){
            len++;
            temp[jcol]=len;
```

```
24              valueC[len]=scalar*valueB[startrowb];
25            }
26            else{
27              valueC[jposition]+=scalar*valueB[startrowb];
28            }
29          }
30      }
31      for(int k=pointerC[i];k<=len;k++){
32         temp[indexC[k]]=-1;
33      }
34  }
35        double endTime1    = System.currentTimeMillis();
36        double totalTime1= endTime1 - startTime1;
37        return totalTime1;
38  }
```

Listing B.13: Numerical approach- CRS format on AB, adapted from [12].

```
1   //Method: Use the symbolic method to find column indices of C for JSA
      format.
2   public static int[][] indexCinJSA_Symbolic( int[][] indexA, int[][]
      indexB, int n){
3       int[][] indexC = new int[n][1];
4         int[] temp = new int[n];
5         int[] tempIndex = new int[n];
6         int nonzero=0;
7         int len = -1;
8       for(int i = 0;i<temp.length;i++){
9        temp[i]=-1;
10      }
11      for(int i = 0;i<indexA.length;i++){
12          int[] aindex = indexA[i];
13          for (int j=0; j<aindex.length; j++){
14             int index = aindex[j];
15             int[] bindex = indexB[index];
16            for(int k = 0;k<bindex.length;k++){
17              int jcol = bindex[k];
18              int jpos = temp[jcol];
19              if(jpos == -1){
20                len++;
21                nonzero++;
22                tempIndex[len] = jcol;
23                temp[jcol] = len;
24              }
25              else {
26            }
27           }
28        }
29          int[] cindex = new int[len+1];
30          System.arraycopy(tempIndex, 0, cindex, 0, len+1);
31          indexC[i]=cindex;
32          for(int ii = 0;ii<len+1;ii++){
33             temp[tempIndex[ii]]=-1;
34          }
35       len = -1;
```

```
    }
  return indexC;
}
```

Listing B.14: Precalculate the column indices of matrix C for JSA format using the symbolic method, adapted from [12].

```
// Method: Multiplying two matrices on JSA format using numerical
    approach.
public static double ABinJSA_Numerical(double [][] valueA, int [][]
    indexA, double [][] valueB, int [][] indexB, int n){
    double[][] valueC = new double[n][1];
    int[][] indexC = new int[n][1];
    int[] temp = new int[n];
    double[] tempValue = new double[n];
    int[] tempIndex = new int[n];
    int nonzero=0;
    int len = -1;
    for(int i = 0;i<temp.length;i++){
      temp[i]=-1;
    }
    double startTime1 = System.currentTimeMillis();
    for(int i = 0;i<valueA.length;i++){
            double[] avalue = valueA[i];
            int[] aindex = indexA[i];
        for (int j=0; j<avalue.length; j++){
              double scalar= avalue[j];
              int index = aindex[j];
              double[] bvalue = valueB[index];
              int[] bindex = indexB[index];
            for(int k = 0;k<bvalue.length;k++){
               int jcol = bindex[k];
               int jpos = temp[jcol];
               if(jpos == -1){
                  len++;
                  nonzero++;
                  tempIndex[len] = jcol;
                  temp[jcol] = len;
                  tempValue[len]=scalar*bvalue[k];
                }
                else {
                   tempValue[jpos]+=scalar*bvalue[k];
                }
            }
        }
        double [] cvalue=new double[len+1];
        int[] cindex = new int[len+1];
        System.arraycopy(tempValue, 0, cvalue, 0, len+1);
        System.arraycopy(tempIndex, 0, cindex, 0, len+1);
        valueC[i]=cvalue;
        indexC[i]=cindex;
        for(int ii = 0;ii<len+1;ii++){
           temp[tempIndex[ii]]=-1;
         }
```

```
            len = -1;
        }
          double endTime1 = System.currentTimeMillis();
          double totalTime1= endTime1 - startTime1;
          return totalTime1;
}
```

Listing B.15: Numerical approach- JSA format on AB, adapted from [12].

```
// Method: Banded matrix multiplication on CRS format.
public static double ABinCRS(double[] valA, int [] colindA, int [] ptrA,
     double[] valB, int [] colindB, int [] ptrB, int [] colindC, int []
    ptrC, int n ){
  double[] valueC= new double[colindC.length];
  double startTime1 = System.currentTimeMillis();
  for(int i =0;i<n;i++){
      int startrowa = ptrA[i];
      int stoprowa = ptrA[i+1]-1;
         for( ;startrowa<=stoprowa;startrowa++){
              double scalar = valA[startrowa];
              int ja = colindA[startrowa];
              int startrowb = ptrB[ja];
              int stoprowb = ptrB[ja+1]-1;
              for(; startrowb<=stoprowb; startrowb++){
                int jb=colindB[startrowb];
                int jc=colindC[ptrC[i]];
                 valueC[ptrC[i]-jc + jb]+=scalar*valB[startrowb];
               }
          }
   }
    double endTime1    = System.currentTimeMillis();
    double totalTime1= endTime1 - startTime1;
    return totalTime1;
}
```

Listing B.16: Banded matrix AB on CRS

```
// Method: Banded matrix-transpose multiplication on CRS format
public static double ATBinCRS(double[] valA, int [] colindA, int [] ptrA
    , double[] valB, int [] colindB, int [] ptrB, int [] colindC, int []
    ptrC, int n){
  double[] valueC= new double[colindC.length];
  double startTime2 = System.currentTimeMillis();
 for(int i =0;i<n;i++){
    int startrowa = ptrA[i];
    int stoprowa = ptrA[i+1]-1;
     for( ;startrowa<=stoprowa;startrowa++){
        double scalar = valA[startrowa];
         int ja = colindA[startrowa];
    // column index of A can be an index of rowpointerC
        int startrowb = ptrB[i];
        int stoprowb = ptrB[i+1]-1;
        for(; startrowb<=stoprowb; startrowb++){
           int jb=colindB[startrowb];
           int jc=colindC[ptrC[ja]];
```

```
17              columnindexC=jc=colindexC[ptr[ja]];
18               valueC[ptrC[ja]-jc + jb]+=scalar*valB[startrowb];
19            }
20          }
21  }
22       double endTime2  = System.currentTimeMillis();
23       double totalTime2 = (endTime2 - startTime2);
24       return totalTime2;
25  }
```

Listing B.17: Banded matrix $A^T B$ on CRS

```
1  // Method: Based on the column indices of input matrices A and B,
       precalculate the column indices of matrix C for JSA storage format
       using the symbolic method
2  public static int[][] indexCinJSA( int [][] indexA, int [][] indexB, int
       n){
3      int[][] indexC = new int[n][1];
4      int[] temp = new int[n];
5      int[] tempIndex = new int[n];
6      int nonzero=0;
7      int len = -1;
8      for(int i = 0;i<temp.length;i++){
9       temp[i]=-1;
10     }
11     for(int i = 0;i<indexA.length;i++){
12         int[] aindex = indexA[i];
13         for (int j=0; j<aindex.length; j++){
14            int index = aindex[j];
15            int[] bindex = indexB[index];
16           for(int k = 0;k<bindex.length;k++){
17             int jcol = bindex[k];
18             int jpos = temp[jcol];
19             if(jpos == -1){
20                 len++;
21                 nonzero++;
22                 tempIndex[len] = jcol;
23                 temp[jcol] = len;
24             }
25             else {
26           }
27         }
28      }
29     int[] cindex = new int[len+1];
30     System.arraycopy(tempIndex, 0, cindex, 0, len+1);
31     indexC[i]=cindex;
32     for(int ii = 0;ii<len+1;ii++){
33         temp[tempIndex[ii]]=-1;
34     }
35   len = -1;
36 }
37 return indexC;
```

```
}
```

Listing B.18: Precalculate the column indices of matrix C for JSA storage format using the symbolic method

```
// Method: Banded matrix multiplication on JSA format
public static double ABinJSA(double[][] valA, int [][] colindA, double
    [][] valB,      int [][] colindB, int [][] colindC, int n){
     double[][] valueC= new double[n][1];
     double startTime3 = System.currentTimeMillis();
 for(int i =0;i<n;i++){
   double [] rowa = valA[i];
   int [] indexa=colindA[i];
   int [] indexc=colindC[i];
   double [] rowc=new double[indexc.length];
     for( int aj=0; aj<rowa.length; aj++ ){
        double scalar = rowa[aj];
        int inda= indexa[aj];
        double[] rowb=valB[inda];
        int []   indexb=colindB[inda];
        for (int bj=0; bj<rowb.length; bj++){
            int cj= indexb[bj]-indexc[0];
            rowc[cj]+= scalar * rowb[bj];
         }
      }
     valueC[i]=rowc;
  }
    double endTime3 = System.currentTimeMillis();
    double totalTime3 = (endTime3 - startTime3);
    return totalTime3;
}
```

Listing B.19: Banded matrix AB on JSA

```
// Method: Banded matrix-transpose multiplication on JSA format
public static double ATBinJSA(double[][] valA, int [][] colindA, double
    [][] valB, int [][] colindB, int [][] colindC, int n){
   double[][] valueC= new double[n][1];
   for (int j=0; j<n; j++){
      valueC[j]=new double[colindC[j].length];
   }
      double startTime4 = System.currentTimeMillis();
    for(int i =0;i<n;i++){
          double [] rowa = valA[i];
          int [] indexa=colindA[i];
          double [] rowb=valB[i];
          int [] indexb=colindB[i];
       for( int aj=0; aj<rowa.length; aj++ ){
           double scalar = rowa[aj];
           int inda= indexa[aj];
           int [] indexc=colindC[inda];
            for (int bj=0; bj<rowb.length; bj++){
                int cj= indexb[bj]-indexc[0];
                valueC[inda][cj]+= scalar * rowb[bj];
```

```
            }
        }
    }
    double endTime4 = System.currentTimeMillis();
    double totalTime4 = (endTime4 - startTime4);
    return totalTime4;
}
```

Listing B.20: Banded matrix A^TB on JSA

```
//Method: Banded matrix multiplication Algorithm (Corollary 1), where LA
    , UA denote lower bandwidth and upper bandwith of matrix A
    respectively
public static double BandAB_multiplydiag(double[] diagStoreArrayA,
    double[] diagStoreArrayB,int LA, int UA, int LB, int UB,int LC, int
    UC,int n, int sum){
  double [] C = new double[sum];
//get main and super-diagonals of C
  double totaltime = 0.0;
for (int k=0; k<=UC;k++){
 int inx_C = LC*(2*n-LC-1)/2 + k*(2*n-k-1)/2 + k ;
 int endinx_C = inx_C + n-k-1;
 for (int i=k+1; i<=Math.min(UB, k+LA);i++){
double[] diag_a1=cutElementFunction(getDiagArrayFunction(diagStoreArrayA
    ,LA, UA, k-i,n),0,k);
double[] diag_b1=cutElementFunction(getDiagArrayFunction(diagStoreArrayB
    ,LB, UB, i,n),0,0);
        int size1=diag_b1.length;
        double start1    = System.currentTimeMillis();
        for (int j=0; j<=size1-1;j++){
          C[inx_C+i-k+j]= C[inx_C+i-k+j]+diag_a1[j]*diag_b1[j];
        }
        double end1 = (System.currentTimeMillis() - start1);
        totaltime = totaltime + end1;
double [] diag_a2=cutElementFunction(getDiagArrayFunction(
    diagStoreArrayA,LA, UA,i,n),0,0);
double [] diag_b2=cutElementFunction(getDiagArrayFunction(
    diagStoreArrayB,LB, UB,k-i,n),k,0);
         int size2=diag_a2.length;
         double start2 = System.currentTimeMillis();
         for (int j=0; j<=size2-1;j++){
               C[inx_C+j]= C[inx_C+j]+diag_a2[j]*diag_b2[j];
             }
         double end2 = (System.currentTimeMillis() - start2);
         totaltime = totaltime + end2;
       }
      for (int i=Math.max(0, k-UB); i<=Math.min(k, UA); i++){
double [] diag_a3=cutElementFunction(getDiagArrayFunction(
    diagStoreArrayA,LA, UA,i,n),0,k-i);
double [] diag_b3=cutElementFunction(getDiagArrayFunction(
    diagStoreArrayB,LB, UB,k-i,n),i,0);
         int size3=diag_a3.length;
         double start3 = System.currentTimeMillis();
         for (int j=0; j<=size3-1;j++){
              C[inx_C+j]= C[inx_C+j]+diag_a3[j]*diag_b3[j];
```

```
36            }
37          double end3 = (System.currentTimeMillis() - start3);
38          totaltime = totaltime + end3;
39       }
40 }
41 //get sub-diagonals of C
42  for (int k=1; k<=LC;k++){
43    int inx_C = LC*(2*n-LC-1)/2 - k*(2*n-k-1)/2;;
44    int endinx_C = inx_C + n-k-1;
45     for (int i=k+1; i<=Math.min(LA,k+UB);i++){
46 double [] diag_a1=cutElementFunction(getDiagArrayFunction(
    diagStoreArrayA,LA, UA,-i,n),0,0);
47 double [] diag_b1=cutElementFunction(getDiagArrayFunction(
    diagStoreArrayB,LB, UB,i-k,n),0,k);
48          int size1=diag_b1.length;
49
50            double start4 = System.currentTimeMillis();
51           for (int j=0; j<=size1-1;j++){
52              C[inx_C+i-k+j]= C[inx_C+i-k+j]+diag_a1[j]*diag_b1[j];
53           }
54           double end4 = (System.currentTimeMillis() - start4);
55           totaltime = totaltime + end4;
56
57           double [] diag_a2=cutElementFunction(getDiagArrayFunction(
    diagStoreArrayA,LA, UA,i-k,n),k,0);
58           double [] diag_b2=cutElementFunction(getDiagArrayFunction(
    diagStoreArrayB,LB, UB,-i,n),0,0);
59           int size2=diag_a2.length;
60
61            double start5 = System.currentTimeMillis();
62           for (int j=0; j<=size2-1;j++){
63            C[inx_C+j]= C[inx_C+j]+diag_a2[j]*diag_b2[j];
64            }
65            double end5 = (System.currentTimeMillis() - start5);
66            totaltime = totaltime + end5;
67     }
68       for (int i=Math.max(0,k-LB); i<=Math.min(k,LA); i++){
69         double [] diag_a3=cutElementFunction(getDiagArrayFunction(
    diagStoreArrayA,LA, UA,-i,n),k-i ,0);
70          double [] diag_b3=cutElementFunction(getDiagArrayFunction(
    diagStoreArrayB,LB, UB,i-k,n),0,i);
71          int size3=diag_a3.length;
72
73          double start6 = System.currentTimeMillis();
74          for (int j=0; j<=size3-1;j++){
75            C[inx_C+j]= C[inx_C+j]+diag_a3[j]*diag_b3[j];
76           }
77            double end6 = (System.currentTimeMillis() - start6);
78            totaltime = totaltime + end6;
79       }
80  }
81  return totaltime;
82 }
```

Listing B.21: Banded matrix AB on DIAS