ON DIAGONALLY STRUCTURED MATRIX COMPUTATION

MOHAMMAD SAKIB MAHMUD
Bachelor of Science, Chittagong University of Engineering & Technology, 2015

A thesis submitted
in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

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

ON DIAGONALLY STRUCTURED MATRIX COMPUTATION

MOHAMMAD SAKIB MAHMUD

Date of Defence: December 4, 2019

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

Dedication

To those who inspired it and will not read it - ***Anonymous***

I dedicate this thesis to my beloved parents and siblings for always being there for me.

Abstract

In this thesis, we have proposed efficient implementations of linear algebra kernels such as matrix-vector and matrix-matrix multiplications by formulating arithmetic calculations in terms of diagonals and thereby giving an orientation-neutral (column-/row-major layout) computational scheme. Matrix elements are accessed with stride-1 and no indirect referencing is involved. Access to the transposed matrix requires no additional effort. The proposed storage scheme handles dense matrices and matrices with special structures such as banded, symmetric in a uniform manner. Test results from numerical experiments with OpenMP implementation are promising. We also show that, using our diagonal framework, Java native arrays can yield superior computational performance. We present two alternative implementations for matrix-matrix multiplication operation in Java. The results from numerical testing demonstrate the advantage of our proposed methods.

Acknowledgments

At first, I want to express my sincere gratitude and appreciation towards my supervisor, Dr. Shahadat Hossain for his continuous support, encouragement and for mentoring me throughout my graduate study. I am grateful for all the opportunities I have had to learn from him in the past two years. I would also like to thank my thesis committee members Dr. Robert Bencokzi and Dr. Saurya Das for their insightful suggestions and constructive feedback.

Next, I would like to thank Administrative Support Ms. Barb Hodgson for always helping me with a smile. A big thanks goes to the University of Lethbridge, School of Graduate Studies of Canada for funding my graduate program.

As the saying goes, you don't need many friends, you only need good ones. I am blessed to have some good friends. Here, I would like to thank my friends Parijat Purohit and Ashraful Huq Suny for helping me before and after moving to Canada. It would be unfair not to mention my co-workers and friends Chowdhury Nawrin Ferdous and Wali Mohammad Abdullah for their intuitive suggestions and motivations throughout this journey.

Finally, I want to mention my family. I want to thank my beloved parents and siblings for their unconditional support and always being there for me when I needed them the most. There is no way I can repay their debt.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nowadays, researchers and engineers need access to software tools for intensive numerical computing which often demands several special features such as flexibility, numerical precision, efficient object code, and high performance. Developing these numerical software which supports extensibility and portability without sacrificing performance is often an expensive task and poses newer challenges to the developers. One of the ways to achieve this goal is to a develop generic application that can be used in a wide range of application fields and is able to run on heterogeneous computing platforms.

Resource constraints, high-performance systems, and scientific applications often require access to the low-level operations and efficient memory use. C++ is a programming language that is performance driven, supports efficient memory use and efficient lower level operations with a flexible set of facilities. This programming language was created with the mindset of using programming techniques that deal with fundamental notions such as resource management, abstraction, memory, modularity, and expression of algorithms. On top of all these, another perk that comes with C++ is its widespread use in parallel programming. Two of the most commonly used parallel programming platforms such as OpenMP and CUDA is supported by C++.

Another programming language that we used in our implementations is Java. Java is a pure object-oriented programming language that has gained rapid recognition as an excellent environment for software development due to its support for, among others, safety, portability, and platform-independence. Despite its widespread popularity in cross-platform and network-centric applications, the adoption of Java in the area of scientific computing has not been as pervasive. It is a commonly held view that Java's current specifications and its implementation framework still pose challenges to achieve high performance on important compute-intensive numerical calculations.

Matrices can represent a lot of information in an efficient and elegant way. Hence, there are many real life applications of matrix computation such as in signal analysis, Markov chain, etc. Also, numerical computation arising in solving partial differential equations often gives rise to structured e.g banded, and sparse matrices. In this thesis, we focused on performing experimental studies on a subset of matrix operations using a new storage scheme, where the matrices are dense or structured in C++ and Java arrays. Additionally, an existing sparse data structure such as Java sparse array has been used to demonstrate the advantage of the newly proposed storage scheme in Java.

1.1 Objectives of this thesis

While undertaking this research project, our aim was to develop a unique orientation-independent data structure that is algorithmically promising for the implementation of Basic Linear Algebra Subprogram (BLAS) level-3 and BLAS level-2 operations. The challenges we looked into to achieve the goal are as follows:

- Design and implementation of a data structure such that padding with redundant storage can be avoided entirely and no additional book-keeping storage is required.
- Parallel implementation of the proposed algorithm.
- Assessing the implementation of the sequential and parallel algorithms by using commonly used performance measurement metrics.

1.2 Contributions

The main contributions of this research project are:

1 We presented an orientation independent diagonal scheme for storing matrices by diagonals which proves to be a good alternative for matrix operation on dense, and structured matrices.

2 An OpenMP implementation of the proposed storage scheme in C++ to demonstrate its potential for task-based dynamic multithreaded architectures [7] [15].

3 Implementation of diagonal storage framework using 1-D Java native array and jagged array which yield superior computational performance.

4 Intense numerical computations were performed using computing systems with high processing speed and hierarchical memory.

5 Part of the results achieved in this thesis has been reported in [22] and [2]. [22] has been published in 23rd IEEE High Performance Extreme Computing Conference (HPEC 2019) and [2] has been submitted for review.

1.3 Organization of the thesis

The rest of the thesis is structured as follows:

Chapter 2 discusses the existing matrix data structures and the data structure of storing a matrix by diagonals that are proposed in this thesis.

Chapter 3 focuses on the parallel implementation approach using OpenMP.

In Chapter 4, the implementation of different storage schemes using basic linear algebra routine and algorithms are presented. In addition, the steps for multithreaded implementation of the proposed matrix data structure are presented here.

Chapter 5 begins with the experiments of numerical computation and presents the results.

The thesis is concluded in Chapter 6 by summarizing the findings from the numerical experiments with a discussion on future research.

Chapter 2

Introduction to Array-based Computing

2.1 Overview of Programming Languages Used in Implementations

C++ is an object-oriented programming language that was created by Bjarne Stroustrup during his Ph.D. thesis. It is an extension of the C programming language. C++ is recognized as an intermediate-level language as it comprises of features from the low-level and high-level programming language. It is widely used in client-server applications, embedded systems, and application software.

C++ is a language that is purely compiled. Source code is translated directly into native code or object code without converting into any intermediate form. This is one of the reasons C++ can obtain high performance. Since code in a file can refer to each other, all object files are linked together and any issue with the address is resolved at this step. The next step is to translate the object code into machine code and one executable file is generated. A C++ compiler takes a source code as an input and translates the source code into object code. After that, all the object files are sent to a linker to correct any missing addresses. The linker then converts the object codes into the machine code that is an executable file .Figure 2.1 represents the architecture of C++.

On the other hand, Java programs use both a compiler and an interpreter. Java compiler takes the source code as input and translates it into another simpler form called the bytecode (bytecode files use .class extension). Bytecode is then interpreted by the Java Virtual Machine (JVM) at run time and executed on the host system. This demonstrates one of the key advantages of using Java which is platform independence. A bytecode Java program can be executed on multiple platforms such as Windows, Linux, or Mac operating systems without altering the source code.

While the implementation architecture of C++ is known to be performance-driven because of its resource management, memory access control and object code generation, Java's implementation framework still poses challenges to gain high performance on compute-intensive numerical calculations. Even though Java has been found accomplished in software development due to its platform independence and extensive library features, some of the issues identified in [9] for numerical calculation are yet needed to be satisfactorily resolved.

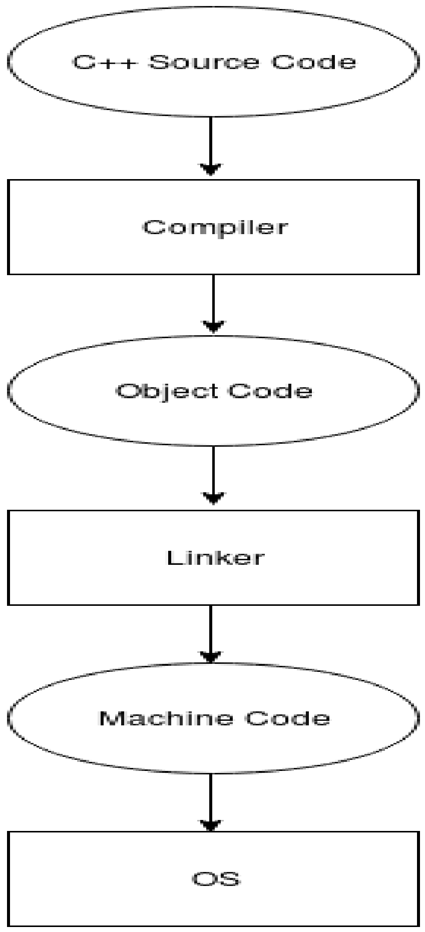

Figure 2.1: C++ Language Architecture

2.1.1 Memory Management and Memory Allocation Scheme

Memory management in computer science can be defined as the administration and coordination of the memory system. Computer applications are significantly influenced by the allocation and management of the system's memory as poor memory management can affect the robustness and speed of a program. This memory management issue is critical in computers with deep memory hierarchy. More specifically, the widening gap between processing and memory access speeds is an important performance bottleneck in numerical calculations [14]. Hence a system's overall performance can be optimized by using efficient memory storage. Memory management combines the tasks of allocation and recycling. Memory allocation can be of three types: **static memory allocation**, **automatic memory allocation** and **dynamic memory allocation**.

Static memory allocation is also known as implicit memory allocation. In this type of memory allocation, memory is allocated before program execution and the size is fixed when the program is created.

Automatic memory allocation occurs for the local variables inside a block and is usually stored on the stack. One of the pitfalls of this memory allocation is limited control over the lifetime of the memory.

Dynamic memory allocation is known as explicit memory allocation. This approach allows an application to request memory while the program is running. It comes with the advantage of controlling the size and lifetime of the memory. Memory leak may occur if memories are not freed at some point which demands efficient memory management. [4]

Memory management can be two types: **manual memory management** and **dynamic memory management**.

In manual memory management, it is the programmer's responsibility to recycle memory. This gets done usually by explicitly calling the heap management functions. No memory is recycled by the memory manager without explicit instruction from the programmer[1]. Manual memory management is used in C++, Pascal, Fortran etc. Some advantages of manual memory management are:

- Programmers can have precise control over memory usage.
- Allows the programmers to make performance optimization by choosing an allocation scheme that performs the best.

The disadvantages of manual memory management are:

- Repetitive bookkeeping of memory is required.
- Memory leak is pretty common in manual memory management. Memory leaks occur when resources are never freed.
- Double frees may happen where a resource is freed more than once.

In dynamic memory management, the run time environment automatically reclaims the memory after making sure that it can no longer be used. Objects that won't be used again are called garbage. Dynamic memory manager is also known as a garbage collector. Many high-level programming languages use dynamic programming languages such as Java, Python JavaScript etc. The advantages of dynamic memory management are:

- Programmer's involvement in freeing up memory is not needed.
- Fewer memory management bugs.
- Clean module interfaces

Some of the pitfalls of dynamic memory management are:

- Memory may be retained just because it is reachable but won't be used again. It can be an issue when small, predictable memory usage is needed.
- Not available to certain programming languages.

One of the advantages of using Java is that Java virtual machine (JVM) supports automatic memory management. The idea is to destroy the objects that are no longer needed and for this Java used a technique called ”mark and sweep”. The first step is to identify the unused objects and mark them for garbage collection. In the second step,the marked objects get deleted by the garbage collector and memory can be compacted optionally after that [24]. In case of C++, it is the programmer's burden to take care of the garbage collection. In spite of the extra work required to free up memory explicitly, many programmers prefer manual memory management over garbage collector for the sake of control and performance. In our implementation of diagonal storage scheme, all the memory usages are predictable, therefore manual memory management seems to be a better fit than dynamic memory management because in dynamic memory management unreferenced things can be left in memory for some time before they are deleted.

2.2 Brief Introduction to Matrices and Data Structures

Matrix operation is a large and important area in scientific computing. Matrix multiplication is an example of an operation that is highly dependent on the details of the data structures. In this thesis, we have experimented on matrix operations using two types of matrices: dense matrix and sparse matrix. In numerical computing and computer science, one of the frequently used matrices is dense matrix. Dense matrix is a matrix where most of the elements are non-zero values. The storage orientation of dense matrices can be row-major storage and column-major storage. The design and selection of storage format depends on the intended application, the functions to be implemented and the structure of the matrix [20]. In this thesis, dense matrix operations are performed on row-major layout and diagonal storage layout using C++ vector. Section 2.4 and Section 2.7 discuss these storage formats in details respectively.

A Sparse matrix is a matrix where most of the elements are zero. J.H Wilkinson's informally defined sparse matrix as *any matrix with enough zeroes that it pays to take advantage* *of them* [6]. The number of zero-valued elements over the total number of elements defines the sparsity of the matrix. While manipulating and storing sparse matrices for computing tasks, algorithms and data structures should be deployed that can take advantage of the sparse structure of the matrix. Operations using dense matrix can be slow and inefficient when applied to a sparse data structure because of the memory and computation power wasted on the zero values. So it is important to store only the non-zero values of sparse matrices and this type of storage can significantly reduce the amount of time spent on the arithmetic operations.

Sparse matrices can be of two types: structured and unstructured. A structured sparse matrix is a matrix where non-zero elements form a regular pattern commonly around a small number of diagonals. On the other hand, in an unstructured sparse matrix, non-zero values are irregularly located. Compared to the dense matrix, operations with sparse matrix require complex implementation because of only storing the non-zero entries and their respective index in the full matrices. Consequently, storing the non-zero entries and their respective index happens to bring more overhead. [33]

A significant number of storage scheme algorithms for sparse matrices exist [23]. In this thesis, matrix operations for sparse matrices are performed using banded diagonal Storage format (DIAS), Compressed Sparse Row (CSR), and Java Sparse Array (JSA). Section 2.6 Section 2.8, and Section 2.9 discuss these storage formats in details.

2.3 Row-major Layout

Computer memory is a linear one-dimensional structure and mapping multi-dimensional data on it can be done in several ways [28]. The selection of memory layout has significant impact on the performances of the code due to the memory and cache mechanism. One of mostly used memory layout is the row-major layout. A matrix A $\in \mathbb{R}^{m \times n}$ is usually stored in two-dimensional vector or array. Each entry in the vector or array corresponds to an element a$_{i,j}$ of the matrix and can accessed using row index i and column index j.

When a matrix is stored linearly in row-major layout, the linear representation of element a$_{i,j}$ of matrix A can be denoted by the equation $i \times n + j$ where n is the number of columns per row, $i, j < n$, and elements of matrix A are accessed in stride-1 pattern. If the matrix is stored using column-major order, element a$_{i,j}$ of matrix A can be located using the equation $j \times m + i$ where m is the number of rows per column, $i, j < n$ and elements are accessed in stride-n pattern.

2.4 Vector in C++

The Standard Template Library (STL) is a C++ template class which supports common data structures and algorithms. One of the four components of STL are containers. Containers or container classes are used to keep data or objects. Sequence container deploys data structure that can be accessed in a sequential way. Vector is a sequence container that represents array but unlike array, their size can change dynamically [23]. Some of the important properties of a vector that have been exploited in this thesis are as follows:

- Any element of the vector can be accessed using its index in constant time.
- The size of a vector can be fixed using resize() function. The indices range from lower index bound to higher index bound.
- Vector uses zero-based indexing. For a vector of fixed length n, index ranges from 0 to n-1.
- Syntax A[i] denotes the ith element of vector A and vector length can be determined using the size() function.

The storage of a vector is managed dynamically and happens to allocate memory more than what is needed. But in our algorithm, the problem size is known beforehand so memory complexity of vectors will be the same as arrays in this case.

2.5 Java Arrays

Programming languages `Fortran` and `C/C++` provide good support for the storage of mathematical objects such as vector, matrix, and higher dimensional tensor via the compound data type `array`. A d-dimensional array of type `T` is a data structure of rank or dimension $d \geq 1$ where the jth dimension, $j = 0, 1, \ldots d-1$ has *extent* n_j such that an object of type `T` stored in the structure is uniquely identified by the d-tuple $(i_0, i_1, \ldots, i_{d-1}), i_j \in \{0, 1, \ldots, n_j\}$. The dimension d as well as the extent n_j are fixed at compile time and remain the same for the duration of the program's execution.

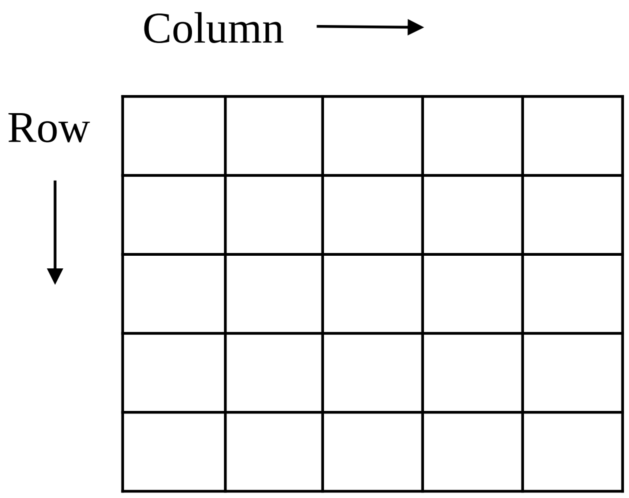

Figure 2.2: A true 2-D array of fixed rank and extent.

Figure 2.2 displays a true 2-D array (rank 2 array) where each dimension has extent 5. The (`object`) type `array` in Java is an array of element type `T` where `T` can be any primitive type or `object` type. If `T` is an `object` type `array` then we have an array of arrays (2-D array). In this case, each array element is a reference (memory address) to an object of type `array`. Storing references to element type allows for flexible shape ("jagged" structure). On the other hand, references in the data structure allow *aliasing* prohibiting the compiler to apply optimizing transformations to the code [9, 32]. The actual objects that the references point to need not be contiguous in the linear memory thus limiting spatial locality in accessing the objects. Moreover, references induce indirection or pointer chasing in accessing data. On the other hand, vector in C++ where elements are placed in contiguous storage and each vector element contains a primitive data type instead of references.

Support for true multidimensional arrays in Java have been proposed via class library (JAMA [8]), by direct translation to bytecode (Titanium [36]), and by enhancing the JVM [30]. Direct translation to bytecode requires adding new language constructs and a source-to-source translator. The class library approach can be implemented as a package for multidimensional arrays e.g. JAMA. This approach is simplest to implement and ensures portability. The third approach, an enhanced JVM, does not require a new class library or changes to the language. Instead, it performs compiler analyses to determine a suitable dense representation for multiarrays. A comprehensive discussion on the three approaches to rectangular array support can be found in [31].

Unlike the aforementioned excellent research for the support of true multidimensional arrays in Java, our proposal described in Section 3 of [2] focuses on data locality enhancements by organizing numerical calculations in linear algebra kernel operations consistent with an orientation-neutral data layout. [2]

2.6 Compressed Row Storage (CRS)

Compressed row storage format is one of the popular storage formats for storing sparse matrices when the sparsity structure of the matrix is unknown and it is introduced by Gustavson. Let assume that, we have a non-symmetric sparse matrix A, then CRS can be created using three vectors which are: [13]

- `val`: This vector stores the nonzero entries of the sparse matrix.
- `col_ind`: This vector stores the column index of the non-zero values that are in vector `val`. For example, if $val(k) = a_{ij}$, then col_ind(k)=j
- `row_ptr`: This vector stores the index of element in the `val` vector that starts a row,

that is, $val(k) = a_{ij}$, then $row\_ptr(i) \leq k \leq row\_ptr(i+1)$. By convention, the last element row_ptr(n+1) stores *nnz* where *nnz* denotes the number of non-zero elements.

An example can give us a better understanding of how CRS can be used to store a sparse matrix. Lets assume, we have a matrix A $\in \mathbb{R}^{4\times 4}$ with four non-zero elements.

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 5 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{pmatrix}$$

Figure 2.3: A Sparse matrix with dimension 4

The vectors `val, col_ind and row_ptr` are as follows for 2.3:

$\texttt{val} = [5, 8, 3, 6]$,

$\texttt{row\_ptr} = [0, 0, 2, 3, 4]$,

$\texttt{col\_ind} = [0, 1, 2, 1]$

This storage is significant in saving storage as instead of n^2 elements, we need only $2nnz + n + 1$ storage locations.

2.7 Diagonal Storage Format

Diagonal storage scheme also named DIAS is a novel storage scheme for storing matrix elements diagonally in a consecutive linear memory layout reported in [22]. Since the matrix is stored by diagonals, it can handle different kinds of matrices such as banded matrices, triangular matrices, symmetric matrices and fully dense matrices. In this section, we will describe how DIAS can be used for dense and banded matrices. The Diagonal storage format use two arrays: *diag* array and *value* array for matrix storage when any specific order such as main diagonal, superdiagonals and subdiagonals is not implied. If the specific order of diagonal storage is known, then only *value* array is needed.

For the better understanding of *diag* and *value* array we can consider a small example. Given a square matrix $A \in \mathbb{R}^{n\times n}$, an element A_{ij} of matrix A can be accessed using the row index i and column index j.

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

Figure 2.4: A square matrix with dimension n

Elements of the matrix in Fig. 2.4 are stored diagonally in 1-D array as follows:

$$\begin{pmatrix} k_0 & , & k_1 & , & k_2 & , & k_3 & , & k_{-1} & , & k_{-2} & , & k_{-3} \end{pmatrix}$$

Figure 2.5: *diag* array

where the dimension of the matrix is $n = 4$ and k_i $(-n+1 \leq i \leq n-1)$ denotes i$^{\text{th}}$ diagonal of a matrix. $k_i = 0$ represents the main diagonal(i=0). For $0 < i \leq n-1$, k_i represents i$^{\text{th}}$ superdiagonal and for $-n+1 \leq i < 0$ represents the i$^{\text{th}}$ subdiagonal.

Matrix multiplication algorithms using DIAS are hard to understand and needed a deeper level of intuition. With that being said, a useful feature of the diagonal storage scheme is that transpose of a matrix is obtained in-place without additional storage. For structured matrices such as banded and sparse matrix, diagonal storage scheme yields substantial savings in terms of storage and computation time [2]. In the next subsection, we will illustrate how the diagonal elements can be accessed in diagonal storage for matrix operations.

2.7.1 Mapping Diagonal Elements for Dense Matrices

The access to the elements of a specific diagonal requires some counting. To access the diagonal elements of a specific diagonal, we need to identify the index of the first element for that diagonal in the 1-D array. We run several iterations for that purpose.

Let's assume that, k is a diagonal number and *start_index* represents the index of first element for k$^{\text{th}}$ diagonal in 1-D array. All the arrays have zero-base indexing. In this example, we have used the array 2.4 from Section 2.7. We run 4 iterations and obtained the following result for the main diagonal and superdiagonals.

k	start_index
0	$0 = 0 \times n - 0$
1	$4 = 1 \times n - 0$
2	$7 = 2 \times n - 1$
3	$9 = 3 \times n - 3$
	$\downarrow$
	$= k \times n - \frac{k(k-1)}{2}$

The formula to identify the index of the first element of a main diagonal or superdiagonal can be written as:

$$start\_index = kn - \frac{k(k-1)}{2} \tag{2.1}$$

For the subdiagonals ($-n+1 \leq k < 0$), the following results were obtained: The formula

k	start_index
-1	$10 = 10 + (\mathbf{1} - 1) \times n - 0$
-2	$13 = 10 + (\mathbf{2} - 1) \times n - 1$
-3	$15 = 10 + (\mathbf{3} - 1) \times n - 3$
	$\downarrow$
	$= \frac{n(n+1)}{2} + (\lvert k \rvert - 1) \times n - \frac{\lvert k \rvert(\lvert k \rvert - 1)}{2}$

to identify the index of the first element of subdiagonals is as follows:

$$start\_index = \frac{n(n+1)}{2} + (|k|-1) \times n - \frac{|k|(|k|-1)}{2} \tag{2.2}$$

The formulas presented in Equation 2.1 and 2.2 were introduced by Nurgul N. Aimaiti in her master's thesis [1]. A better approach for calculating the start index of a diagonal is recently posted in our published article [22]. The implementation of diagonal matrix multiplication in C++ has also been found improved compared to the Java implementation on which k^{th} diagonal elements have been copied and returned by a function for the computation purpose. Since we know that the number of elements in any k^{th} diagonal is $n-k$, this advantage of knowing the start index and length of the diagonals has been exploited in our latest implementation to access the matrix elements directly and thus by improving performance by getting rid of the copying part.

2.8 Banded Storage Format

A Banded matrix is a special form of sparse matrix. Banded matrix is a structured matrix in which the non-zero entries are confined to a diagonal band, including the main diagonal and zero or more diagonals on either side. The sum of the distance of the non-zero diagonals from the main diagonal on the both sides of it is called the bandwidth. Bandwidth can be expressed by the form : $k_l + k_u + 1$ where k_l represents the number of non-zero subdiagonals and k_u represents the number of non-zero superdiagonals. [33]

Therefore, it seems natural to store only the non-zero entries and express computation by diagonals. For unstructured sparse matrices, storage schemes such as compressed storage by rows (CSR) and compressed storage by columns (CSC) need two additional arrays of *2n+1* to provide access to the non-zero elements. On the contrary, by exploiting the structural sparsity of non-zero elements, it is possible to access the non-zero elements along the band directly and thus avoiding the need for auxiliary data structures.

BLAS specifications provide a compact storage schemes for banded matrices that enables cache-friendly access to the matrix elements. Intel Math Kernel Library (MKL) stores banded matrices with upper bandwidth k_u and lower bandwidth k_l in an array `ab` with $(k_l + k_u + 1) \times n$ elements such that the (i, j) th element is accessed by the $\texttt{ab}(k_u + 1 + i - j, j)$ for $\max(1, j - k_u) \leq i \leq \min(n, j + kl)$. Thus, this storage scheme incurs a padding of $O({k_l}^2) + O({k_u}^2)$. Therefore, the advantage of this compact storage is realized if $k_l, k_u << n$

Madsen et. al [27] describe a matrix multiplication algorithm, hence called the MKS algorithm, in terms of diagonals which can be used to multiply banded matrices. Tsao and Turnbull [35] discuss an implementation of MKS algorithm where a variant of MKL band storage is used. Their numerical experiments show that for matrices with smaller bandwidth MKS algorithm yields substantial saving both in terms of storage and time for computation. A distributed memory ”Reduced System Conjugate Gradient” algorithm described in [18] utilizes the band structure to compute products of the form $C^{\mathrm{T}}Cp$ where p is the search direction. The computation is implemented as two matrix-vector products with matrices C and C^{T}; the matrix product $C^{\mathrm{T}}C$ is never explicitly formed. [22]

Now we will explore the storage scheme for storing banded matrices diagonally that is proposed in this thesis. The banded storage we propose here does not require any extraneous storage. Moreover, the matrix is stored by diagonals so it can handle different kinds of structured matrices such as banded matrices, triangular matrices, symmetric matrices and diagonal matrices. Let $A \in \mathbb{R}^{n\times n}$ be a banded matrix with lower bandwidth k_l and upper bandwidth k_u if

$$j > k_u + i \text{ implies } a_{ij} = 0 \text{ and } i > k_l + j \text{ implies } a_{ij} = 0.$$

. We denote by $A_k = \{a_{ij} \mid j - i = k\}$, the kth super-diagonal, and by $A_{-k} = \{a_{ij} \mid i - j = k\}$, the kth sub-diagonal of matrix A. The proposed diagonal storage use one dimensional array of size $n(k_l + k_u + 1) - \frac{k_u(k_u+1)}{2} - \frac{k_l(k_l+1)}{2}$, corresponding to the number of non-zero entries along the band.

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & 0 \\ a_{10} & a_{11} & a_{12} & a_{13} \\ 0 & a_{21} & a_{22} & a_{23} \\ 0 & 0 & a_{32} & a_{33} \end{pmatrix}$$

Figure 2.6: A banded matrix

Fig. 2.6 displays a $n \times n$ banded matrix with lower bandwidth k_l =1 and upper bandwidth k_u=2. This array represents the row-major of diagonal in the following order:sub diagonals, main diagonal, and super diagonals. Instead of storing all the elements, we store the four diagonals k_l + k_u +1.

The 1-D array in Fig. 2.7 displays the linear representation of how diagonals of a banded matrix is stored.

$$\begin{pmatrix} a_{10} & a_{21} & a_{32} & ; & a_{00} & a_{11} & a_{22} & a_{33} & ; & a_{01} & a_{12} & a_{23} & ; & a_{02} & a_{13} \end{pmatrix}$$

Figure 2.7: Diagonals representation of Banded Matrix

where, for clarity, we use a semicolon (;) to show the boundary between two consecutive diagonals.

2.8.1 Mapping Diagonal Elements for Banded Matrices

Since the number of elements in k_{th} (sub or super) is $n-k$, a mapping for the elements of a specific diagonal can be easily constructed by identifying the start index of that diagonal in 1-D array. Each diagonal element in banded matrices can be accessed by the following equations [1]:

1. The index of first element of k $^{\text{th}}$ superdiagonal can be given by the expression:

$$start\_index \leftarrow \frac{k_l(2n-k_l-1)}{2} + \frac{k(2n-k-1)}{2} + k \tag{2.3}$$

2. For subdiagonal elements, $k < 0$, take the absolute value of k, The index of first element of k th subdiagonal can be given by the expression:

$$start\_index = \frac{k_l(2n-k_l-1)}{2} - \frac{|k|(2n-|k|-1)}{2} \tag{2.4}$$

2.9 Java Sparse Array (JSA)

JSA was proposed by Gundersen and Steihaug [19] and it exploits the Java's flexible definition of multidimensional arrays. In Java, a two-dimensional array is formed using arrays of arrays. It either stores primitive types (float, double etc.) or objects. Both rectangular and jagged array can be created using this definition.

As similar to CSR, JSA is a row oriented storage format. Two arrays are used by JSA, elements of which is itself an array (object). One of these arrays, *Value*, stores arrays of matrix entries-each row of the matrix is stored in a separated array. All the separate arrays are the elements of *Value* array, thus forming an array of arrays. The second main array, *Index*, stores arrays of column indices for the matrix elements, again one array per row. Lets assume, we have a sparse matrix M with dimension 5 $\times$ 5 as follows:

$$\begin{pmatrix} * & a_{01} & * & * & * \\ * & * & a_{12} & * & a_{14} \\ * & * & a_{22} & * & a_{24} \\ a_{30} & a_{31} & * & a_{33} & * \\ * & * & * & a_{43} & * \end{pmatrix}$$

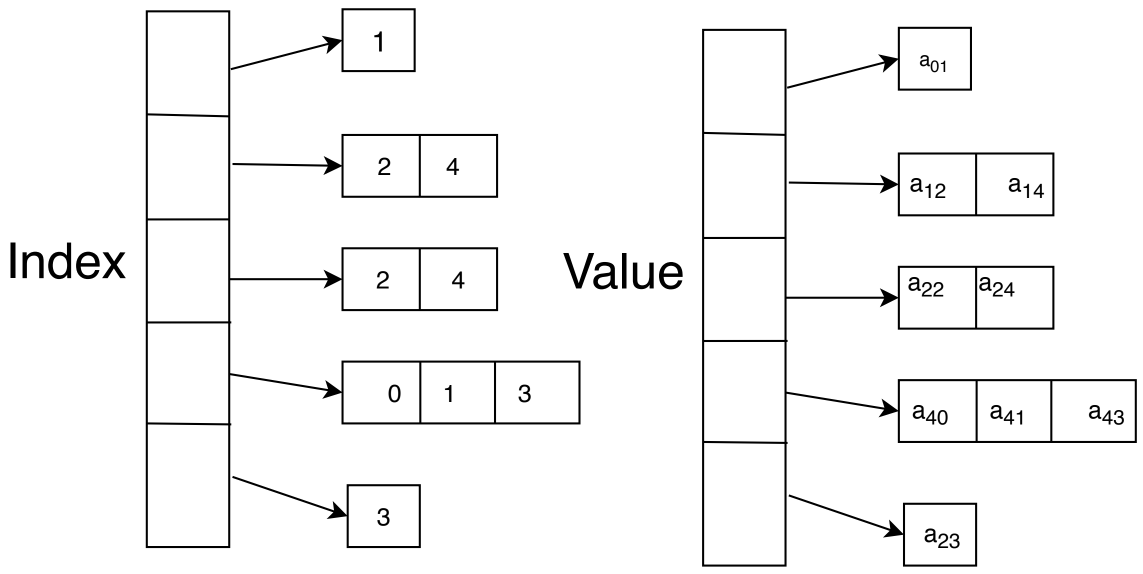

Figure 2.8: A sparse matrix stored using JSA

The $*$ symbol indicated a zero entry in the matrix. The memory requirements to store a sparse matrix in JSA is $2nnz + 2n$ array locations. [26] Fig. 2.8 shows an example sparse matrix stored using JSA.

2.10 Jagged Diagonal Storage Format

A jagged array is an array of array. Jagged array stores array instead of other data type value. A Jagged array is initialized with two [][] square brackets. The first bracket specifies the size of the array and the second bracket specifies the dimension of the array which is going to be stored as values. A reasonable alternative to a 1-D array to store the diagonals of a matrix is a Java array of arrays also referred to as Java jagged array first proposed and used in this thesis. As the diagonals in a matrix can be of different lengths, a jagged array is an appropriate data structure to store those unequal objects. The advantage of a Java array of arrays becomes evident by noting that a diagonal object can be treated independently of other diagonals.

Figure 2.9 displays the data structure for Java jagged array. In this format, each row in the matrix has its element and index in a separate array. Storing references to element

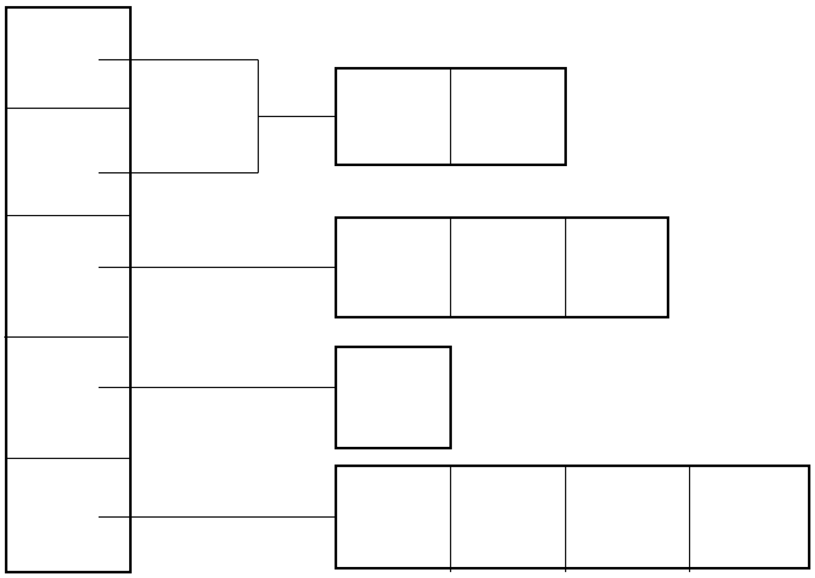

Figure 2.9: A Java jagged array

type allows for the flexible shape ("jagged" structure). On the other hand, references in the data structure allow prohibiting the compiler to apply optimization transformation to the code. The actual objects these references are pointing to need not to be contiguous in linear memory thus limiting spatial locality in accessing the objects. Additionally, references induce indirection or pointer chasing in accessing data [2]. The banded matrix of Figure 2.6 can be stored as a jagged array by the following Java declaration:

$$
\begin{aligned}
\texttt{double[][]}\ \mathit{JADiag} &= \{\{\ a_{00},\ \ a_{11},\ \ a_{22},\ \ a_{33}\ \} \\
&\quad\ \{\ a_{01},\ \ a_{12},\ \ a_{23}\ \} \\
&\quad\ \{\ a_{02},\ \ a_{13}\ \} \\
&\quad\ \{\ a_{10},\ \ a_{21},\ \ a_{32}\ \}\}
\end{aligned}
$$

Support for true multidimensional arrays in Java have been proposed via class library (JAMA [8]), by direct translation to bytecode (Titanium [36]), and by enhancing the JVM [30]. Unlike the aforementioned excellent research for support of true multidimensional arrays in Java, in this thesis we focus on data locality enhancements by organizing numerical calculations in linear algebra kernel operations consistent with an orientation-neutral data layout. [2]

All the data structures that we have discussed in this chapter will be used for implementations in Chapter 4.

Chapter 3

Parallel Implementation Approach

The importance of parallelization in accelerating numerical computation has been well received for decades. With the increasing amount of data and with the availability of parallel platforms, parallel computing is a necessity in the computing world right now. Lately, significant progress has made in parallel computation such as reducing turnaround time from the development of a microprocessor to a parallel machine based on the microprocessor and ensuring a longer life cycle for parallel applications. These progress display significant promise in the future application of parallel programming.

3.1 Serial Computing VS Parallel Computing

In general, software are written for serial computation. Serial computing is a type of computing where instructions are executed sequentially one after another on a single processor.

On the other hand, in parallel computing, a computational problem is solved by using multiple compute resources at the same time. A compute resource can be a single computer with multiple processors or a group of such computers connected by a network. The following steps can be followed to solve a problem in parallel: [5]

- The First step is to break the problem into separate parts that can be solved concurrently.
- Each part is further broken down into a series of instructions.

- Instructions from different parts can be executed in parallel on different processors.
- An overall synchronization mechanism is implemented.

3.2 Parallel Computing Platform

Parallel programming demands for the synchronization of concurrent tasks or communication of intermediate results. In shared address space architecture, memory and data are accessible to all the processors.As a result, shared address space machines focus on expressing construct for the concurrency and synchronization as well as reducing the associated overheads.The Shared address space programming model varies depending on how data is shared among the processors, concurrency model and support for synchronization. Two of the most used shared address space models are as follows:

- Directive based programming model
- Threaded programming model

In this thesis, we have experimented with directive-based programming model. Thread based programming models are popular among the system programmers rather than application programmers. This is because thread-based API such as Pthreads are considered as low-level primitives. A Large class of applications can be efficiently supported by higher-level directives and manipulation of the threads by the programmers is not required. OpenMP is such a directive-based API (Application Program Interface) that supports shared memory multiprocessing. OpenMP provides support for concurrency, data handling, and synchronization. OpenMP API can be used with C, C++, and Fortran. In the following section, we will discuss the OpenMP programming model. [3]

3.3 OpenMP Programming Model

One of the basic OpenMP directives in C and C++ is *#pragma* compiler directives. The directive consists of a directive name followed by a clause list. OpenMP programs run sequentially until the *parallel* directive is invoked. This directive creates a group of threads once invoked. The number of threads can be specified at runtime using OpenMP functions or set in the directives using an environment variable. The following code snippet shows the *parallel* directive prototype:

```
#pragma omp parallel [clause list]
/* structured block */
```

The thread that encounters *parallel* directive is known as **master** thread and is assigned thread id 0. OpenMP model hides the low-level details and allows the programmers to:

- Specify the parallel region. Each thread created by the directive executes the same parallel region.
- Specify concurrent tasks or in other words parallelizing the loop.
- Specify the scope of variables in the parallel region which is also known as data handling.
- Specify how workloads are divided among threads. This is called scheduling.
- Specify if the threads need to be synchronized

3.3.1 Data Handling in OpenMP

Manipulation of data by threads plays a vital role in program performance. OpenMP supports various data classes such as *private, shared, lastprivate and firstprivate* [3]. These classes are explained as follows:

- The clause *private (variable list)* defines that the set of variables specified local to each thread and threads can not access each other's data.
- The clause *firstprivate (variable list)* is same as *private* clause, but in this case variables can be initialized to corresponding values before the parallel directive and *first-*

private can be used to bring in that value from the outside context into the parallel region.

- The clause *lastprivate (variable list)* is special case of *private* clause. *lastprivate* clause can be used to transfer values from the parallel region to the outside context.
- The *shared (variable list)* indicates the set of variables that are shared across the threads and there will be one copy for each variable. Shared variables need to be handled with care such that data race doesn't happen. Data race happens when two threads access the same memory without proper synchronization.
- Reduction is one way of improving the performance of parallel applications by removing the synchronization point from a `for` loop. The OpenMP *reduction* clause allows threads to keep local count for a specific operation and combine the local solution into a single solution at the end of the loop.

3.3.2 Specifying Parallel Tasks in OpenMP

To specify concurrency across iteration and tasks, *parallel* directive can be associated with other directives. OpenMP supports two directives for concurrency across iteration and tasks. They are as follows:

- For directive
- sections directive

The For Directive

The `for` directive in OpenMP is used to assign the parallel iteration space across threads. The `for` directive prototype in OpenMP is as follows:

```
#pragma omp for [clause list]
/* for loop */
```

The clause list in this context comprises *private, firstprivate, lastprivate, reduction, schedule, nowait and ordered* clauses. The first four clauses are related to data handling and the semantics are the same as in the *parallel* directive. In this thesis, we have used the `for` directive to split the diagonals among the threads.

3.3.3 OpenMP Scheduling

OpenMP *scheduling* clause deals with assigning iteration space to threads. There are four scheduling classes in OpenMP: *static, dynamic, guided and runtime* [3]. Choosing the right schedule can significantly affect the performance of a parallel program.

Static Schedule

The general prototype for *static* scheduling class is `schedule(static[,chunk-size])`. *Static* scheduling breaks the iteration space equal to size *chunk-size* and maps the iteration space among the threads in a round-robin fashion. If the *chunk-size* is not specified then the iteration space is split into number of chunks equal to the number of threads available.

Dynamic Schedule

Execution time may vary depending on several reasons such as non-uniform processor load, heterogeneous computing resources, etc. OpenMP *dynamic* scheduling class can resolve this issue. The general form of this class is `schedule(dynamic[,chunk-size]`. The iteration space is split into chunks based on the *chunk-size*. Whenever a thread becomes idle it gets assigned to threads and this solves the problem of temporal imbalances in static scheduling. Since the number of diagonals in a matrix are of different sizes, we have used *dynamic* schedule in this thesis to take care of the load imbalance. If no *chunk-size* is specified, then by default the *chunk-size* is 1.

Guided Scheduling

Guided scheduling is the same as dynamic scheduling. The *chunk-size* starts large and decreases with the computation to better handle the load imbalance. The *chunk-size* is reduced at an exponential rate as each chunk is dispatched to a thread. The general prototype for guided scheduling is `schedule([, chunk-size])`. With small chunk-size, guided scheduling works better compared to the performance with bigger chunk-size.

Runtime Scheduling

If it is needed to delay scheduling decisions until runtime, then *runtime* scheduling class can be used. Since selecting a scheduling class depends on several reasons such as processor loads and computing resources, the scheduling can be set to *runtime* to pick the best scheduling class based on the impact of various scheduling classes. OpenMP environment variable **OMP_SCHEDULE** sets the scheduling class and the chunk size in this case. If no scheduling technique is specified with the *omp for* directive, then the scheduling technique is implementation-dependent.

3.4 Synchronization in OpenMP

Coordinating the execution of multiple threads in parallel programming is an important aspect. The POSIX threads also known as Pthreads API supports conditions variables and mutual exclusion flags. OpenMP provides a handful of synchronization directives in an easy to use API. We will discuss the following directives and their use in this section [3].

- The **barrier** directive
- The **critical** and **atomic** directives
- The **ordered** directive

The barrier Directive

The barrier directive works as a synchronization point. The syntax for barrier directive in OpenMP is as follows:

```
#pragma omp barrier
```

Whenever a barrier directive is reached, all the threads in a group must wait until other threads are done execution and then release. In case the `barrier` directive is used with nested *parallel* directive, then the barrier directive pairs with *parallel* directive that is followed by it. Some overheads are associated with applying barrier directive.

The critical and atomic directives

If there is a need for the serial execution of the code segment within the *parallel* region, then the critical and atomic directive can be used. The syntax for the critical section is the following:

```
#pragma omp critical [name]
        structured block
```

Whenever a **critical** directive is encountered, only one thread enters the critical region specified by a name and all the other threads must wait until it is done before entering the critical section. If name of the critical section is unspecified, then it maps to a default name that is same for all unnamed critical sections. OpenMP has another similar directive called the *atomic* directive. It is used for the small updates in memory location and the *atomic* directive ensures that the memory location update is performed as a single operation.

The ordered Directive

The ordered directive can be used for a desired execution order of a parallel loop in a way the serial version would execute it. The syntax of this directive is as follows:

```
#pragma omp ordered
        structured block
```

The **ordered** directive works as a serialization point since only one thread can enter the *ordered* block after all the threads have exited. So overhead will be higher if a large portion of a loop is enclosed within the ordered directive.

3.5 Performance Metrics for Parallel Program

Various performance metrics can be used to do the performance analysis of a parallel program. To identify the best algorithm, it is important to study the performance of a parallel program. The performance metrics used in this thesis will be discussed in the following subsections.

3.5.1 Execution Time

Execution time can be of two types: serial runtime and parallel runtime. The Serial runtime of a program refers to the time spent between the beginning and end of its execution on a single processor. Serial runtime is denoted by $\mathrm{T_s}$. The Parallel runtime of a program is the time elapsed from the moment the parallel computation starts to the moment the last processing element finishes execution [3] . Parallel runtime is denoted by $\mathrm{T_p}$

3.5.2 Speedup (S)

Speedup is a commonly used metric to analyze the performance of a parallel program since it captures how much performance gain is achieved for parallel implementation over serial implementation. Speedup is the ratio of runtime for best sequential algorithm over the runtime of a parallel algorithm executed on *p* processors. Speedup is denoted by symbol S and can be expressed mathematically as follows:

$$S = \frac{T_\mathrm{s}}{T_\mathrm{p}} \tag{3.1}$$

where T_s denotes the serial runtime of best sequential algorithm and T_p denotes the parallel runtime.

While improving the performance of a system, it usually happens that we speed up one part of a system and the overall system performance depends on how important this part was and how much it sped up. Let's assume that, we have a system and executing some applications on the system requires time T_{old}. Also, suppose that some part of the system needs a fraction α of this time and its performance is improved by a factor of k. So, the component required time $\alpha \times T_{old}$ before and now it need time $\frac{\alpha \times T_{old}}{k}$. The overall execution time would thus be

$$T_{new} = (1-\alpha)T_{old} + \frac{(\alpha T_{old}}{k})$$
$$= T_{old}[(1-\alpha) + \frac{\alpha}{k}]$$

From this we can compute speedup $S = \frac{T_{old}}{T_{new}}$ and this is known as Amdahl's law. [11] The insight from Amdahl's law is that generally, the speedup is in proportion to the number of processing elements and can not be more than the number of processing elements, p. In a special case, speedup can be more than the number of processing elements, p and this is known as super-linear speedup.

3.5.3 Efficiency (E)

Efficiency is another metric to analyze the performance of a parallel program. In an ideal scenario, speedup can be equal to the number of processing elements, p. But in real systems, this ideal behavior is not achieved since processing elements spend a fraction of time in idling and communicating. Efficiency is a measure of the part of the time for which processing element is usefully employed. Efficiency can be defined as the ratio of speedup to the number of processing elements. Efficiency is denoted by the symbol E and mathematically given by:

$$E = \frac{S}{p} \tag{3.2}$$

where S denotes speedup and p denotes the number of processing elements. Parallel efficiency can not be more than 1.

Chapter 4

BLAS2 and BLAS3 Operations on Different Storage Schemes and Algorithms

In this chapter, we are going to explore Basic Linear Algebra Subroutine (BLAS) operations using different storage schemes discussed in Chapter 2. BLAS are the routines that provide support for vector and matrix operations. Level 2 BLAS includes matrix-vector operations and Level 3 BLAS includes matrix-matrix operations. We will use the following abbreviated forms for matrix operations in this chapter:

- *Ax:* multiplying a matrix by a vector
- *A^Tx:* multiplying transpose of a matrix by a vector
- *AB:* multiplying a matrix by a matrix
- *A^TB:* multiplying transpose of a matrix by a matrix

We also denoted Standard Matrix Multiplication as SMM and Diagonal Matrix Multiplication Routine as DMM. These abbreviations carry the same meaning in the following chapter as well.

In the following sections of this chapter, we will discuss four linear algebra routines: *Ax, A^Tx, AB and A^TB* for dense matrices using algorithms that take into consideration the row-major orientation on Standard Matrix Multiplication (SMM) and diagonal storage (DIAS) on Diagonal Matrix Multiplication (DMM). For banded matrices, the same routines were applied on Diagonal Storage (DIAS) and Java Jagged Array (JJA). Since the support for banded matrix operations in BLAS specification is available at level-2 only, a naive approach would be to use the GBMV (General Band Matrix-Vector) routine n times. As a result, the performance of matrix multiplication may suffer due to difficulty in exploiting data reuse or systems with deep memory hierarchy. We demonstrated an algorithm in terms of diagonal to multiply band matrices. In the last section, we will discuss the task-based parallel implementation of these routines using OpenMP.

4.1 Matrix-Vector Multiplication (Ax *and* A^Tx) on SMM

To define the multiplication between a matrix A and vector x, we need to see vector x as a column vector. For general matrix-vector multiplication, the number of columns in A must be equal to the number of rows in vector x. So if A $\varepsilon\mathbb{R}^{m\times n}$ and x $\varepsilon\mathbb{R}^{n\times 1}$ then result vector $y = Ax$ is a column vector with dimension $m \times 1$ [10]. The general formula for matrix-vector is the following:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{pmatrix}$$

Figure 4.1: Matrix-vector multiplication

In Fig. 4.1, it is clear that matrix-vector multiplication takes the dot product of x with each row of matrix A. The general algorithm to compute $y = Ax$ is as follows:

Sometimes working with a different form of a given matrix that contains the same information is required. Transpose of a matrix is a fairly common operation in linear algebra and it has numerous applications in mathematical computations such as providing the notion of sizes and angles [29]. In transpose product, $y = A^{\mathrm{T}}x$, the rows and columns of matrix A

Algorithm 1 Standard matrix-vector multiplication

Input Data: One-dimensional array *valA* in which the elements of matrix A is stored in row-major layout and a vector x of size n.
Output Data: A column vector y of size n.

```
1: procedure MULTIPLY()
2:     for $i \leftarrow 0$ to $n-1$ do
3:         $y[i] \leftarrow 0$
4:         for $j \leftarrow 0$ to $n-1$ do
5:             $y[j] \leftarrow$ y[j] + valA[i $\times$ n + j] $\times$ x[j]
6:         end for
7:     end for
8: end procedure
```

are switched and the elements of matrix A are accessed in a stride-n pattern instead of the stride-1 pattern.

4.2 Matrix-Vector Multiplication (Ax *and* A^Tx) on Diagonal Storage Format

If a $n \times n$ matrix A is stored diagonally and a vector x of size n is taken then matrix-vector multiplication can be performed using the following algorithm:

In Algorithm 2, we are storing the elements of matrix A diagonally in n 1-D array named *valA* and to get the start index of a specific diagonal we are using the index formula in chapter 2.

4.2.1 Computing Steps of Ax and A^Tx using Diagonal Storage Format

The computing steps of Algorithm 2 are as follows:

- Total number of elements on each diagonal can be defined by $n-k$. $|k|$ denotes the absolute value of kth diagonal.
- *DiagStore() function*: This function receives a 2-D matrix of size $n \times n$ as its argument and stores the elements of the matrix into an 1-D array or vector in main-super-sub

Algorithm 2 Matrix Vector Multiplication

Input: valA, diag, vector x of size n

Output: vector y of size n

```
1: procedure MULTIPLY( )
2:     for $d \leftarrow 0$ to $diag\_index.size() - 1$ do
3:         $k \leftarrow$ diag_index[d]
4:         if $k >= 0$ then
5:             $i \leftarrow$ k
6:             $j \leftarrow 0$
7:             start_index $\leftarrow$ (k*n)-(k*(k-1))/2
8:             stop_index $\leftarrow$ start_index + n - k -1
9:             for start_index to stop_index do
10:                y[j] $\leftarrow$ y[j]+valA[start_index] $\times$ x[i]
11:                $i \leftarrow$ i+1
12:                $j \leftarrow$ j+1
13:            end for
14:        end if
15:        if $k < 0$ then
16:            $i \leftarrow 0$
17:            $j \leftarrow$ abs(k)
18:            start_index $\leftarrow$ n*(n+1)/2 + (j-1)*n-(j*(j-1))/2
19:            for start_index to stop_index do
20:                y[j] $\leftarrow$ y[j]+valA[start_index] $\times$ x[i]
21:                $i \leftarrow$ i+1
22:                $j \leftarrow$ j+1
23:            end for
24:        end if
25:    end for
26: end procedure
```

diagonal order. For a given matrix A, a matrix element is expressed by the form $a_{i,j}$ where i corresponds to row index and j corresponds to column index,then the m$^{\text{th}}$ component of k$^{\text{th}}$ main diagonal or superdiagonal can be accessed by the following equations:

$$a_k(m) = a_{m,k+m}$$

and the m$^{\text{th}}$ component of k$^{\text{th}}$ subdiagonal can be accessed by the equation:

$$a_k(m) = a_{k+m,m}$$

- *multiply() function:* This functions represents the implementation of matrix-vector multiplication using diagonal storage. It receives the dimension of matrix A that is n, the diagonally stored vectorized format of matrix A and a vector x of size n as its arguments. The key here is to access i$^{\text{th}}$ element on k$^{\text{th}}$ diagonal in a stride-1 pattern using the index formulas on step 7 and multiply that element with the i$^{\text{th}}$ element of vector x using Step 18 of Algorithm 2. The result is stored in the i$^{\text{th}}$ position of vector y. Accessing the elements in stride-1 pattern will improve cache performance.

  For the transpose matrix-vector operation, the structure of the algorithm is the same as *y=Ax* except for the following changes in the algorithm:

  Line 4: i = k, j = 0 becomes i = 0, j = k

  Line 14 : i = 0, j = k becomes i = k, j = 0

4.3 Matrix-Matrix Multiplication on SMM

Consider the numeric algebra kernel operation of matrix-matrix multiplication (MM) of the form:

$$C \leftarrow AB$$

and

$$C \leftarrow A^{\mathrm{T}}B$$

where A, B, C are square matrices of dimension n. Usually, matrix-matrix multiplication is implemented using three nested loops and the loops are identified by their indices i, j, and k. A simple algorithm for matrix multiplication is presented in Algorithm 3 :

Algorithm 3 Standard matrix-matrix multiplication

```
Input Data: Two square matrices A and B of size n
Output Data: Matrix C of size n
1:  procedure MULTIPLY()
2:      for $i \leftarrow 0$ to $n-1$ do
3:          $y[i] \leftarrow 0$
4:          for $j \leftarrow 0$ to $n-1$ do
5:              for $k \leftarrow 0$ to $n-1$ do
6:                  $C[i][j] \leftarrow C[i][j] + A[i][k] \times B[k][j]$
7:              end for
8:          end for
9:      end for
10: end procedure
```

If we permute the loop indices with some minor changes in code, we get six functionally similar version of matrix multiplication. Each version performs the same number of addition and multiplication operations. But the innermost loop iterations make the difference in the number of accesses and the locality. These versions can be grouped into three classes such as Partial row and partial column-oriented (V1, V2), Pure column-oriented (V3, V4), Pure row-oriented (V5,V6) [12].

- V1: The loop order for version one is (i, j, k). Each row of matrix A is accessed in stride-1 pattern and each column of matrix B is accessed in the stride of n.

- V2: For version two the loop order is (j,i,k). The access pattern for this version is the same as V1.

- V3: The loop order for version three is (j,k,i). Hence, each column of matrix A and Matrix C is scanned in the stride of n pattern. In this case, spatial locality will decrease compared to the V1 and V2 because of loop interchanging.

- V4: The loop order for version four is (kji). The access pattern for elements in the matrices is the same as V3.

- V5: The loop order for version five is (k,i,j). The innermost loop scans each row of matrix B and C in a stride-1 pattern which will result in good spatial locality.

- V6: For version six, the loop order is (i,k,j). The scanning pattern for the rows of matrix B and C is the same as V5.

In terms of spatial locality, V5 and V6 happen to perform better than the other versions because matrix B and C are both scanned row-wise in the stride-1 pattern. In this thesis, we have compared our proposed storage scheme with V5 and V6.

Transpose of a matrix is a new matrix where the rows and columns of the original matrix are flipped. For $C = A^{\mathrm{T}}B$, the step 4 in Algorithm 3 becomes

C[i][j] $\leftarrow$ *C[i][j]* + *A[k][i]* $\times$ *B[k][j]*.

4.4 Matrix-matrix Multiplication using Diagonal Storage Format

Materials in sections 4.4 – 4.6 discuss diagonally structured linear algebra borrowed heavily from [22][2] In this section, we will describe the algorithm matrix-matrix multiplication by diagonals for dense matrices. We will use the following figure as a running example to illustrate our algorithm [22].

$$\begin{pmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{pmatrix} \leftarrow \begin{pmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{pmatrix} + \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{pmatrix}$$

Figure 4.2: Matrix Matrix Multiplication By Diagonals

and the pseudocode for the algorithm is as follows:

Algorithm 4 Matrix Matrix Multiplication by Diagonals

```
1: procedure MULTIPLYDIAG( )
2:
3:     for $k \leftarrow 0$ to n-1 do
4:         for $i \leftarrow k+1$ to n-1 do
5:             $c_k(;i-k) \leftarrow c_k(;i-k) + a_i \times b_{k-i}(;k)$
6:             $c_k(i-k;) \leftarrow c_k(i-k;) + a_{k-i}(;k) \times b_i$
7:         end for
8:         for $i \leftarrow 0$ to k do
9:             $c_k \leftarrow c_k + a_i(;k-i) \times b_{k-i}(i;)$
10:        end for
11:    end for
12:    for $k \leftarrow 1$ to n-1 do
13:        for $i \leftarrow k+1$ to n-1 do
14:            $c_{-k}(i-k;) \leftarrow c_k(i-k;) + a_{-i} \times b_{i-k}(;k)$
15:            $c_{-k}(;i-k) \leftarrow c_{-k}(;i-k) + a_{i-k}(k;) \times b_{-i}$
16:        end for
17:        for $i \leftarrow 0$ to k do
18:            $c_{-k} \leftarrow c_{-k} + a_{-i}(k-i;) \times b_{i-k}(;i)$
19:        end for
20:    end for
21: end procedure
```

The key observations in developing the algorithm for computing *k*th super/sub diagonal of result matrix C is to identify the part of argument matrices A and B that contribute to the calculation. Specifically, the last k rows of matrix A and the first k columns of matrix B can be ignored. To illustrate our algorithm we will consider computing diagonal C_3 for the matrix multiplication operation of size 4 as shown in Fig. 4.2. Diagonal C_3 has only one element which is c_{03}. The segments of matrices A and B that take part in computing c_{03} is shown below:

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \end{pmatrix} \begin{pmatrix} . & . & . & b_{03} \\ . & . & . & b_{13} \\ . & . & . & b_{23} \\ . & . & . & b_{33} \end{pmatrix}$$

We can express the computation in terms of diagonals:

$$C_3 \leftarrow C_3 + \mathsf{A}_0 \times \mathsf{B}_3 + \mathsf{A}_1 \times \mathsf{B}_2 + \mathsf{A}_2 \times \mathsf{B}_1 + \mathsf{A}_3 \times \mathsf{B}_0$$

In the above, the diagonals for argument matrices B and C are written in ”math sans serif font” to reflect the fact that in respective diagonals the last k (k=3 here) elements of matrix A and first k (k=3 here) elements of matrix B have been removed. Therefore, each of the diagonals is a vector of length one. It is evident that the standard inner product of row 0 of matrix A and column 3 of matrix B produces the same result as multiplication by diagonals. It is important to note that, in pair of multiplying diagonals the indices of the diagonals add to the index of the diagonal of matrix C being computed. Next, we will show that multiplication of diagonals, indicated by symbol $\times$, is point-wise or Hadamard multiplication [21]. Now we will show an example for computing diagonal C_0. We denote by $A_k = \{a_{ij} | j - i = k\}$ the kth super-diagonal and $A_{-k} = a_{ij} | i - j = k$ the kth sub-diagonal. According to this observation, we identify diagonal pairs A_i, B_j such that $i + j = 0$ [22]. The computation is followed :

.

.

$$\begin{pmatrix} c_{00} & \cdot & \cdot & \cdot \\ \cdot & c_{11} & \cdot & \cdot \\ \cdot & \cdot & c_{22} & \cdot \\ \cdot & \cdot & \cdot & c_{33} \end{pmatrix} \leftarrow \begin{pmatrix} c_{00} & \cdot & \cdot & \cdot \\ \cdot & c_{11} & \cdot & \cdot \\ \cdot & \cdot & c_{22} & \cdot \\ \cdot & \cdot & \cdot & c_{33} \end{pmatrix} + \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{pmatrix}$$

$$\begin{aligned} &+ (\mathsf{A}_0 \times \mathsf{B}_0) \\ &+ (\mathsf{A}_1 \times \mathsf{B}_{-1}) + (\mathsf{A}_{-1} \times \mathsf{B}_1) \\ &+ (\mathsf{A}_{-2} \times \mathsf{B}_2) + (\mathsf{A}_2 \times \mathsf{B}_{-2}) \\ &+ (\mathsf{A}_3 \times \mathsf{B}_{-3}) + (\mathsf{A}_{-3} \times \mathsf{B}_3) \end{aligned}$$

$$\begin{aligned} \begin{pmatrix} c_{00} \\ c_{11} \\ c_{22} \\ c_{33} \end{pmatrix} \leftarrow & \begin{pmatrix} c_{00} \\ c_{11} \\ c_{22} \\ c_{33} \end{pmatrix} + \begin{pmatrix} a_{00} \\ a_{11} \\ a_{22} \\ a_{33} \end{pmatrix} \times \begin{pmatrix} b_{00} \\ b_{11} \\ b_{22} \\ b_{33} \end{pmatrix} \\ & + \begin{pmatrix} a_{01} \\ a_{12} \\ a_{23} \\ 0 \end{pmatrix} \times \begin{pmatrix} b_{10} \\ b_{21} \\ b_{32} \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ a_{10} \\ a_{21} \\ a_{32} \end{pmatrix} \times \begin{pmatrix} b_{10} \\ b_{21} \\ b_{32} \\ 0 \end{pmatrix} \\ & + \begin{pmatrix} a_{02} \\ a_{13} \\ 0 \\ 0 \end{pmatrix} \times \begin{pmatrix} b_{20} \\ b_{31} \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ a_{20} \\ a_{31} \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ b_{02} \\ b_{13} \end{pmatrix} \\ & + \begin{pmatrix} a_{03} \\ 0 \\ 0 \\ 0 \end{pmatrix} \times \begin{pmatrix} b_{30} \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ a_{30} \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ 0 \\ b_{03} \end{pmatrix} \end{aligned}$$

In the above illustration, zeros appearing in the shorter diagonal are used for clarity only.

The above discussion is captured in the following theorem as reported in [22] [2]:

Theorem 4.1. *In the matrix-matrix multiplication operation $C \leftarrow C + AB$ for $A, B, C \, \varepsilon \, \mathbb{R}^{n \times n}$ performed by diagonals, the kth diagonal for $k \geq 0$ is given by*

$$C_k = \sum_{j=0}^{n-1} A_j \times B_{k-j} + \sum_{j=k+1}^{n-1} A_{k-j} \times B_j \tag{4.1}$$

and for $k \leq 0$

$$C_k = \sum_{j=0}^{n-1} A_{k+j} \times B_{-j} + \sum_{j=k+1}^{n-1} A_{-j} \times B_{k+j} \tag{4.2}$$

The proof of this theorem is based on induction.

These above steps can be computed using the following functions:

- *getIndex() function:* This function returns the index of the start element for kth diagonal.
- *DiagStore() function:* This function receives a 2-D dense matrix as its argument and converts it into 1-D array or vector. Conversion of 2-D matrix into 1-D vector includes storing by the diagonals (main diagonal-super diagonal-sub diagonal) order. For a given matrix A, a matrix element is expressed by the form $a_{i,j}$ where i corresponds to row index and j corresponds to column index,then the m$^{\text{th}}$ component of k$^{\text{th}}$ main diagonal or superdiagonal can be accessed by the following equations:

  $$a_k(m) = a_{m,k+m}$$

  and the m$^{\text{th}}$ component of k$^{\text{th}}$ subdiagonal can be accessed by the equation:

  $$a_k(m) = a_{k+m,m}$$

- *multiplyDiag() function*: This function contains the implementation of DIAS algorithm to output C=AB. It receives array dimension n, two 1-D array or vectors (A,B) which are already stored by the diagonals using DiagStore() function.

4.5 Banded Matrix-Matrix Multiplication

In banded matrix-matrix multiplication, any of the argument matrices can be banded. Therefore, it is a special case of general matrix-matrix multiplication by diagonals.

Let, l_A, l_B, l_C and u_A, u_B, u_C denote respectively, the lower-bandwidth and the upper-bandwidth for matrices A, B and C. It can be shown that l_C = $min(l_A + l_B, n-1)$ and u_C = $min(u_A + u_B, n-1)$. So the special case of banded matrix multiplication can be done by stating that the diagonals that fall within the bandwidth of the respective matrices will be accessed only. For instance, to compute C_k for $0 \leq k \leq min(u_A + u_B, n-1)$ can be expressed using 4.1

$$C_k = \sum_{j=0}^{n-1} A_j \times B_{k-j} + \sum_{j=k+1}^{n-1} A_{k-j} \times B_j \tag{4.3}$$

For sub-expression $\sum_{j=0}^{n-1} A_j \times B_{k-j}$, it must be $j \leq u_A$ and

$$\begin{cases} k-j \leq u_B, & k \geq j \\ j-k \leq l_B, & k < j \end{cases}$$

The conditions for the other sub-expression are followed,

$$\begin{cases} j \leq u_B, j-k \leq l_A, & k < j \end{cases}$$

In a similar way, the sub-diagonals for the banded multiplication can be derived from 4.2 .

The pseudocode for banded matrix-matrix multiplication is followed:

Algorithm 5 Banded Matrix Matrix Multiplication by Diagonals

```
1: procedure MULTIPLYBAND( )
2:
3:     for $k \leftarrow 0$ to $u_C$ do
4:         for $i \leftarrow k+1$ to $min(u_B, k+l_A)$ do
5:             $c_k(;i-k) \leftarrow c_k(;i-k) + a_i \times b_{k-i}(k;)$
6:             $c_k(i-k;) \leftarrow c_k(i-k;) + a_{k-i}(;k) * b_i$
7:         end for
8:         for $i \leftarrow max(0, k-u_B)$ to $min(k, u_A)$ do
9:             $c_k \leftarrow c_k + a_i(;k-i) \times b_{k-i}(i;)$
10:        end for
11:    end for
12:    for $k \leftarrow 1$ to $l_C$ do
13:        for $i \leftarrow k+1$ to n-1 do
14:            $c_{-k}(i-k;) \leftarrow c_k(i-k;) + a_{-i} \times b_{i-k}(;k)$
15:            $c_{-k}(;i-k) \leftarrow c_{-k}(;i-k) + a_{i-k}(k;) \times b_{-i}$
16:        end for
17:        for $i \leftarrow max(0, k-l_B)$ to $min(k, l_A)$ do
18:            $c_{-k} \leftarrow c_{-k} + a_{-i}(k-i;) * b_{i-k}(;i)$
19:        end for
20:    end for
21: end procedure
```

The computing steps for banded matrix multiplication are same as the computing steps for general matrix-matrix multiplication by diagonals.

4.6 Computational Experience with Diagonally Structured Linear Algebra in Java

Even though Java is fairly popular in the developing world, it is a commonly believed that Java's current specification and its implementation framework still pose challenges to achieve high performance on important compute intensive numerical calculations. With the introduction of Just-In-Time (JIT) compiler technology, Java Virtual Machine (JVM) has increasingly been successful in closing the performance disparity with other mainstream compiled languages [9].

One of the classic bottlenecks between computer processors and memory is that computer processors process data at a much higher rate than the rate at which current memory technology can deliver data. This gap in processor and memory speed is not likely to go down in the near future [17]. Modern computer systems employ multiple levels of storage devices in a hierarchical manner, with devices that are faster and have smaller capacity are organized closer to the processor. A device at a specific level thus acts as a staging post or cache for the next faster device. Cache-friendly algorithms tend to reference data items that are located close to the ones (in storage devices) that were referenced recently (spatial locality[1]) or multiple references to the same data items within a short time period. The impact of hierarchical memory on performance is profoundly evident in compute-intensive numerical calculations e.g., matrix-matrix multiplication where cache-efficient reference to matrix data depends on the layout of data in memory for improved temporal locality and the reordering of the calculations for improved spatial locality [34].

In this thesis, we enable cache-friendly access to data by using diagonally structured storage and computation for matrix-matrix multiplication. We provide two alternative implementations using 1-D Java native array and jagged array, and use the computational framework of [22] to organize the underlying arithmetic calculations. [2]

4.7 Impact of Caching in Memory Hierarchy and Exploiting Locality

A cache can defined as a smaller, faster memory that acts as a staging area for slower, larger memory. The process of using a cache is known as caching.

The basic concept of memory hierarchy is that faster storage device at level k acts as a cache for slower storage device at level $k+1$. Storage device at level k is partitioned into a number of blocks that are the same size as the blocks in storage device at level $k+1$.

[1]Corei7 cache line can contain 8 `doubles` so that once the line is brought in, the next 7 adjacent `double` values are served from the faster cache.

Storage device at level k contains copies of a subset of blocks from level $k+1$. Data is copied between level k and $k+1$. When a program needs a data object d from level $k+1$, at first it looks for d in one of tha blocks cached at level k. If d is found at level k, then this event is called a cache hit. If d is not found at level l, then what we have is called a cache miss. When a cache miss occurs, data block that contains d is copied to level k from level $k+1$. After it has been copied from level $k+1$ to level k, it stays there in expectation of later accesses. Here, the advantage of locality comes in. In temporal locality, same data object are more likely to be referenced multiple times. Once the data object has been copied into cache on the first miss, it is expected that number of subsequent hits will occur on the same data object. As the cache is faster than the storage at next lower level, these subsequent hits will serve faster than the original miss. In spatial locality, subsequent data objects of d within a block are likely to be referenced and the cost of copying a block will be amortized by this [11]. The matrix operations in terms of diagonals exploit spatial locality by accessing the required elements of a diagonal in stride-1 pattern.

4.8 Parallelization Steps

The algorithms we discussed above were implemented for task-based dynamic multi-threaded architecture using OpenMP. To develop a algorithm in parallel, there are two basic stages:

- Developing a decomposition and mapping strategy
- Exploit this technique among the nodes

In the above discussion, decomposition refers to splitting the computations to be performed among a specific number of threads. One of the commonly used decomposition techniques is data decomposition. Data decomposition consists of two steps,

- Partitioning the data on which the computations are performed.

- Using this data partition to create partitioning of the computations into tasks.

The partitioning can be done in numerous ways. For parallel implementation of matrix operations by diagonals we have used the concept of data decomposition. Our parallel implementation of matrix operations by diagonals comprises of the following the steps:

Step 1 The algorithm for matrix multiplication by diagonals contains nested loops. Only the outer-most loop was parallelized using OpenMP. In each iteration, each thread is assigned a specific diagonal to compute.

Step 2 Each process multiplies the required elements to compute its assigned local diagonal. Since a kth diagonal can have $n-k$ elements, so each process will performing $n \times (n-k)$ computations.

Step 3 A matrix of dimension $n \times n$ has $2n-1$ diagonals. So we are partitioning $2n-1$ diagonals over p processes and we can use maximum of $2n-1$ processes. This can be expressed as follows:

$$\frac{2n-1}{p}$$

Chapter 5

Numerical Experiments

In this chapter, we will present the test results from the computational study of different storage schemes and matrix multiplication algorithms discussed in Chapter 2 and Chapter 4 respectively.. First, we will briefly discuss the test data set followed by the computing system employed for all numerical testings. Next, we will outline the benchmarking and the test results will be presented graphically in the rest of the chapter.

5.1 Data Sets for Numerical Test

We performed our numerical experiments on real matrices. The elements of the matrices (*full and sparse*) are generated randomly using a user-defined function. A random seed is passed to the function to generate a random number. We have also used C++ *rand()* function to generate random numbers for computation. The data types we experimented with are single-precision and double-precision floating-point numbers. All the matrices are square, with the same number of rows and columns, and non-symmetric.

- Dense Matrix

The input matrices are initially generated in a 2-D format. Then the matrix elements are stored in a 1-D array using a row-major layout for standard matrix multiplication. For diagonal storage scheme, the matrix entries were stored in a 1-D array with specific diagonal order (*main-super-sub)*

- Banded Matrix

In case of banded matrix, only the elements within a predefined bandwidth are stored diagonally using a 1-D array in a specific order (*(sub-main-super)*.

5.2 Test Environment and Benchmarking

The execution time performance measure for matrix operations was measured using the OpenMP function `omp_get_wtime()`. This function returns the execution time in seconds. The difference between two values returned by the two different calls to `omp_get_wtime()` function, one before the respective function call and the other at the end was used to compute the total execution time. We ran the programs multiple times to get the average execution time. To time the *for* loop for benchmarking results, we considered taking time from where it begins, to where it ends. Table 5.1 displays the computing system employed for all numerical testing. Table 5.3 and Table 5.2 displays the hardware specs for the numerical computing. All the parallel implementations of the algorithms were tested using the test environment presented in Table 5.2. Table 5.3 was used to obtain sequential computing results.

Table 5.1: System Info

Operating System:	CentOS
GCC version :	4.4.7
Linux version:	2.6.32
OpenMP Version :	3.0

Table 5.2: Test Hardware Specifications

Processor	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
CPU Cores(s)	4
L1 d and L1 i cache	32KB
L2 cache	256KB
L3 cache	8192KB
CPU MHz	3400.132

Table 5.3: Test Environment Specifications (Medusa)

Processor Model:	AMD Opteron(tm) Processor 4284
Thread(s) :	16
Thread(s) per core:	2
Core(s) per socket:	4
Socket(s):	2
CPU MHz:	1400.00
L1d cache:	16K
L1i cache:	64K
L2 cache:	2048K
L3 cache:	6144K

We have used Table 5.3 to perform experiments on the parallel implementation and Table 5.2 was used to run experiments on the sequential implementation. In Table A.2, each CPU has 4 cores and there are 2 threads per core. So we get to experiment with 16 threads on our parallel implementation of the algorithms. It also includes separate d-cache (data cache) and i-cache (instruction cache) for L1 so that the processor can read instruction and processor at the same time without making conflict between data accesses and instruction accesses.

5.3 Memory Exceptions While Implementing the Algorithms

We implemented the algorithms using C++ and Java. For dense matrix-matrix multiplication in C++, we can use row and column dimension up to 16,000 before the total memory is exhausted. In Java, after using the `-Xmx` flag to specify the maximum heap memory, the matrix dimension comes down to 8,000 before JVM runs out of heap memory. For the sparse matrices, we experimented with dimension 100000 and since we are storing the non-zero elements only, memory seems not be an issue in this case.

5.4 Numerical Experiments

In this section, we will present the experimental results for basic linear algebra routines (BLAS) discussed in Chapter 4 for different storage schemes presented in Chapter 2. Speedup and Efficiency for parallel implementation were computed respectively using equation (3.1) and equation (3.2) in Chapter 3. We have three computational models to present the results that are:

Model A: Dense Matrix Operations

- Ax and A^Tx on SMM versus DMM
- AB and A^TB on SMM versus DMM

The first computational model compares the test results for dense matrix operations using standard method that takes the linear layout(row-major) of a 2-D array into consideration versus dense matrix operations using diagonal multiplication method.

Model B: Banded Matrix Operation on Java Arrays

- Banded Matrix-Matrix Multiplication on DIAS using Java jagged array versus 1-D diag array.

- Speedup comparison of Java 1-D array diagonal, jagged array diagonal, and Java sparse array (JSA).

In the second computational model, our first numerical experiment compares the performance of banded matrix-matrix multiplication using diagonal storage schemes for Java jagged array versus 1-D diag array. 1-D array diag stores the diagonals by using Java native arrays and Java jagged array stores the diagonals using Java array of arrays. In the second numerical experiment, we present the performance comparison of 1-D array diagonal, jagged array diagonal, and JSA, relative to CSR storage (CSR clocked the longest time in our experiments) for banded matrix-matrix multiplication. JSA, in general, outperforms CSR has also been observed in [25] which supports our approach of measuring performance relative to CSR.

Model C: Parallel Benchmarking

- Speedup and Efficiency obtained by diagonal matrix-matrix multiplication method for banded matrices.

- Speedup and Efficiency obtained by diagonal matrix-matrix multiplication method for dense matrices.

In our final model, we present the performance metrics of the diagonal multiplication method for banded matrices and dense matrices using the parallel performance metrics discussed in Chapter 3.

5.4.1 Model A:

In our first numerical experiment of Model A, we present the execution time performance measure for the standard multiplication method and diagonal multiplication method on Ax. We experimented with `float` data types and the execution time was measured in `milliseconds`.

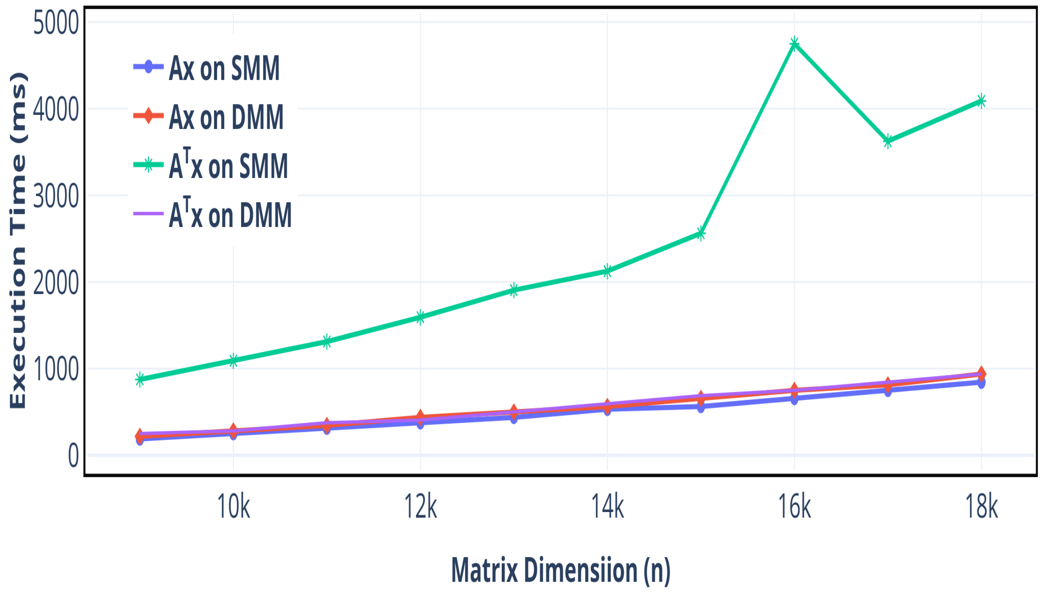

Figure 5.1: SMM versus DMM on Ax and $\mathrm{A}^{\mathrm{T}}\mathrm{x}$

As can be seen in Figure 5.1, for matrix-vector multiplication, SMM and DMM are performing nearly the same. But in case of $\mathrm{A}^{\mathrm{T}}\mathrm{x}$, the performance of SMM has degraded whereas the performance of DMM is nearly same as the performance of DMM on Ax. This is because if we use diagonal storage format, then transpose of a matrix can be obtained in-place without any extra storage and matrix elements are accessed in stride-1 pattern that results in good cache performance. This can be done by switching the superdiagonals and the subdiagonals of the matrix.

In our second numerical experiment of Model A, we compare the DIAS against the $V1$, $V5$ and $V6$ from Chapter 4 on AB and A^TB. Execution time was measured in seconds and `float` datatype was used in the experiments. As discussed in Chapter 4, out of the six loop orderings, loop $i-k-j$ and $k-i-j$ (pure row oriented) produce the best timing followed by $i-j-k$ and $j-i-k$ (partial row/column oriented). Pure column-oriented loop orderings (jki and kji) happens to perform the worst because in this case as matrix entries are accessed in the stride of n pattern which results in more cache misses than the other versions. A Cache miss is a state where the data requested for processing by a component or application is not found in the cache memory. It causes execution delays by requiring the program or application to fetch the data from other cache levels or the main memory [11].

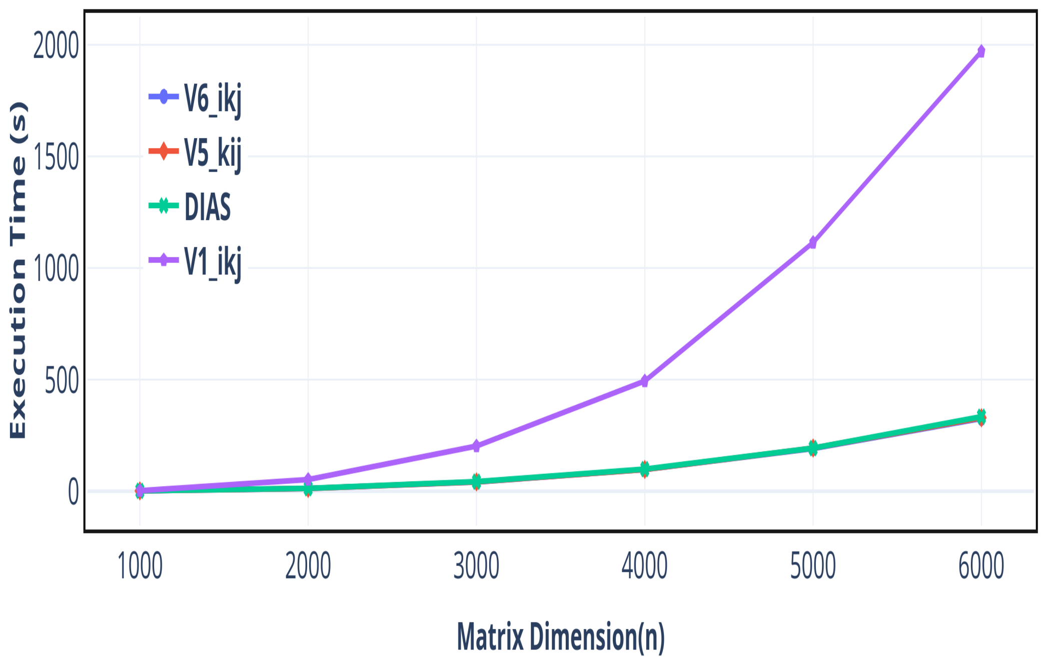

Figure 5.2: The DIAS versus $V1_ijk$, $V5_kij$ and $V6\_ikj$ on AB

From Figure 5.2, we can see that partial (row/column) oriented version $(i-j-k)$ is performing the worst. Our diagonal multiplication algorithm is performing nearly same the pure row oriented versions ($i-j-k$ and $k-i-j$) which implies that the diagonal multiplication algorithm is more efficient than the pure column-oriented and partial row/column oriented version for $C = AB$. This is because accessing elements in a row-major order is faster than accessing elements in a column-major order.

In Figure 5.3, as consistent with the result observed in Figure 5.2, partial row/column oriented version$(-i-j-k)$ is the least efficient while diagonal multiplication algorithm is performing slightly better than pure row-oriented version ($i-j-k$ and $k-i-j$). It is also noticeable that execution times for matrix multiplications and transpose matrix multiplication are similar in case of diagonal multiplication method.

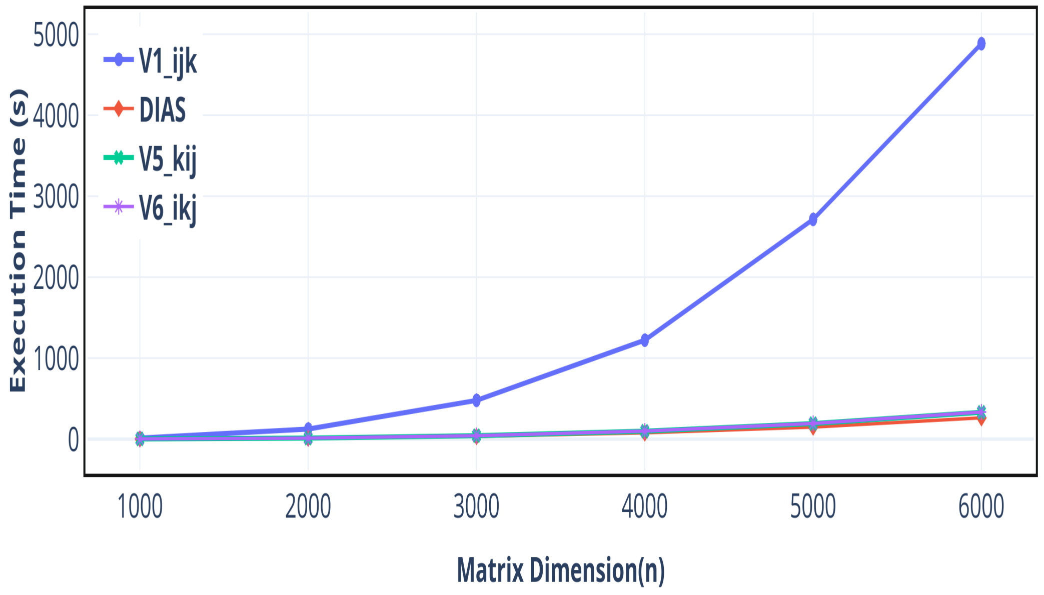

Figure 5.3: The DIAS versus $V1_ijk$, $V5_kij$ and $V6\_ikj$ on A^TB

5.4.2 Model B:

In the first numerical experiment of Model B, we present the execution time performance measure between Java jagged array versus 1-D diag array for banded matrix-matrix multiplication using diagonal storage. Execution time was measured in seconds and `double` datatype was used in the experiments.

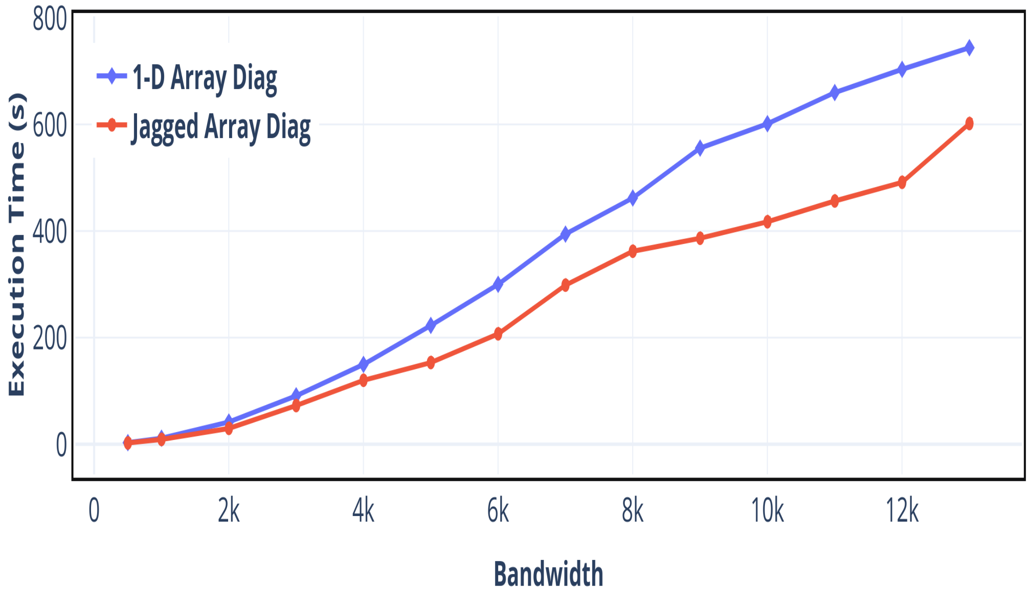

Figure 5.4: Banded matrix-matrix multiplication on DIAS

In Figure 5.4, we plot the time (in seconds) taken by matrix-matrix multiplication for 8000×8000 real (`double`) matrices with varying bandwidths ($k_b = 501, \ldots, 13001$), until JVM out-of-memory exception is encountered. As we can see in the figure, jagged array implementation performs better than the 1-D array. This is a bit unexpected as 1-D arrays are free of reference indirection costs associated with jagged arrays. The Same behavior is also observed in a recent paper [16] on a benchmark application (3-D Poisson solver using Fast Fourier Transform). On an open JVM system, the 1-D array storage is found to be almost 2 times slower than the jagged array storage. Index calculation overhead must be the reason for higher running time [2].

In our second numerical experiment of Model B, we computed speedup for three data structures: Java 1-D array diagonal, jagged array diagonal, and Java sparse array (JSA) with respect to Compressed Sparse Row (CSR) format. The speedup was computed using the following expression:

$$Speedup = \frac{T_e}{T_{CSR}} \tag{5.1}$$

where T_e denotes the execution time required by Java 1-D array diagonal, jagged array diagonal, and Java sparse array (JSA) respectively and T_{CSR} denotes the execution time required by Compressed Sparse Row (CSR) format. Execution time was measured in seconds and `double` datatype was used in the experiments.

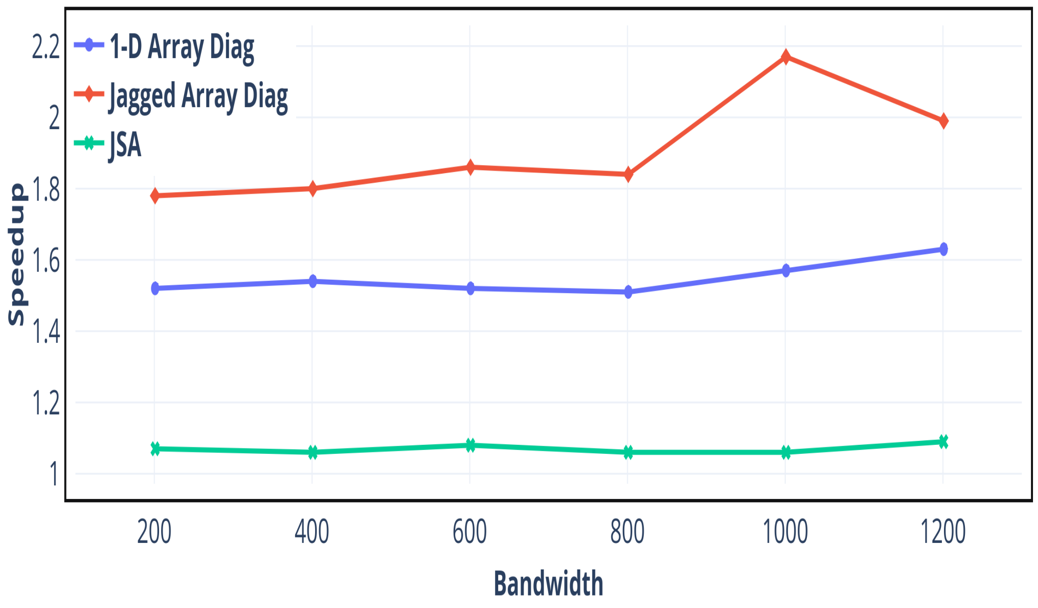

Figure 5.5: Speedup for 3 methods with respect to CRS for dimension 50000

In Figure 5.5 as can be seen, diagonal storage schemes perform the best (more than a factor of 2x) followed by the JSA. Also jagged array diag performs better than 1-D array diag and the similar result has been found in Figure 5.4.

5.4.3 Model C:

The following figures present the speedup and efficiency obtained by diagonal matrix multiplication algorithm for banded matrix with $n = 100000$ and p=16 threads for varying bandwidth. We have experimented with different OpenMP schedules discussed in Chapter 3 for different chunk size. `dynamic` schedule with chunk size 15 gives the best result that is presented here. We have used Equation 3.1 and Equation 3.2 from Chapter 3 to compute the speedup and efficiency. Execution time was measured in seconds and `float` datatype was used in the experiments.

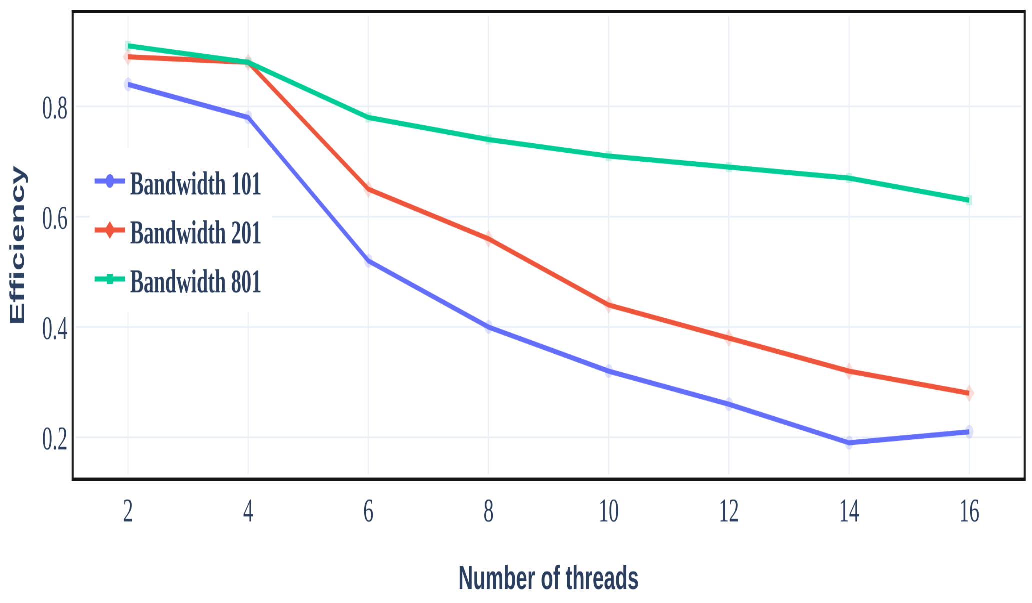

Figure 5.6: Efficiency for chunk size 15 with matrix dimension 100000

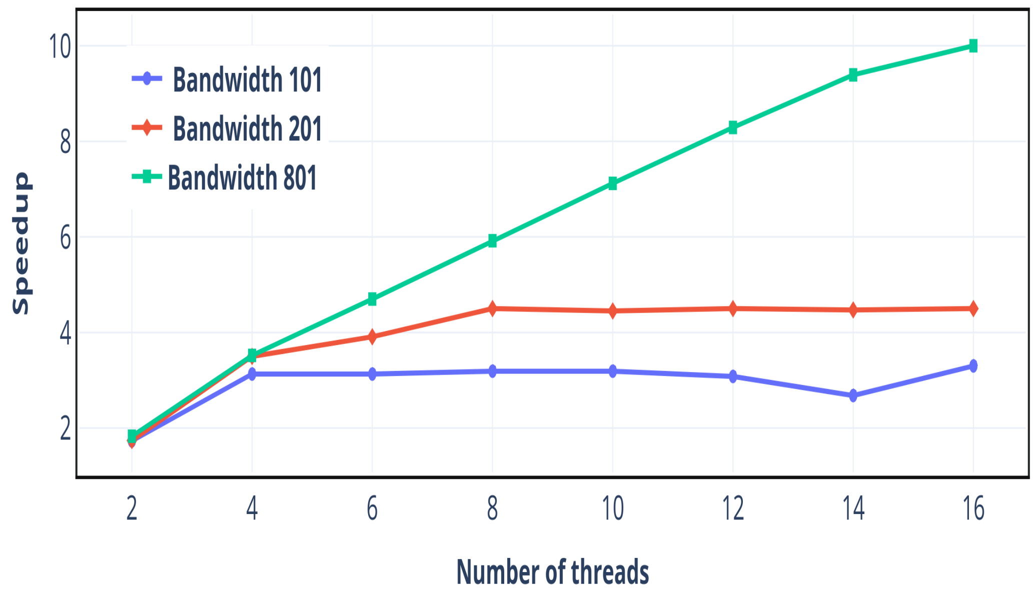

Figure 5.7: Speedup for chunk size 15 with matrix dimension 100000

In Figure 5.6 and Figure 5.7, we plot the efficiency and speedup while changing the number of threads employed and keeping the problem size fixed. As we can see from the graph, efficiency and speedup increase with the bandwidth increasing and they peak between 800 and 1600 after which the performance degrades. This is because with bandwidth increasing more work becomes available to be parallelized and parallel overhead has more chances to become amortized with the increasing number of threads [22].

Next, we plot the speedup and efficiency for dense matrices while changing the number of threads employed along with the problem size. Figure 5.8 and Figure 5.9 respectively present the speedup and efficiency for dense matrices:

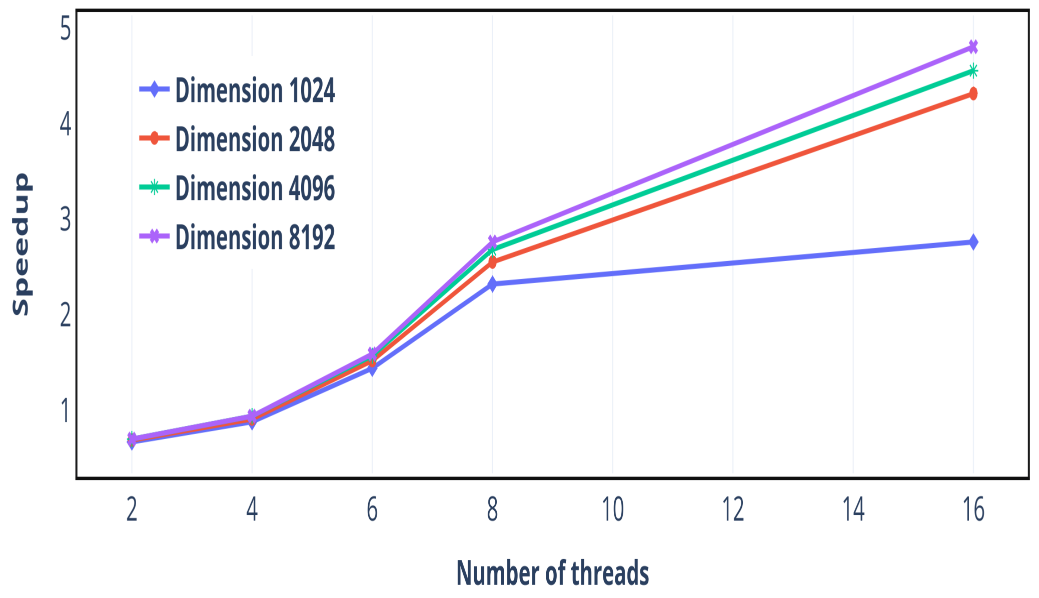

Figure 5.8: Speedup for dense matrices with varying input size

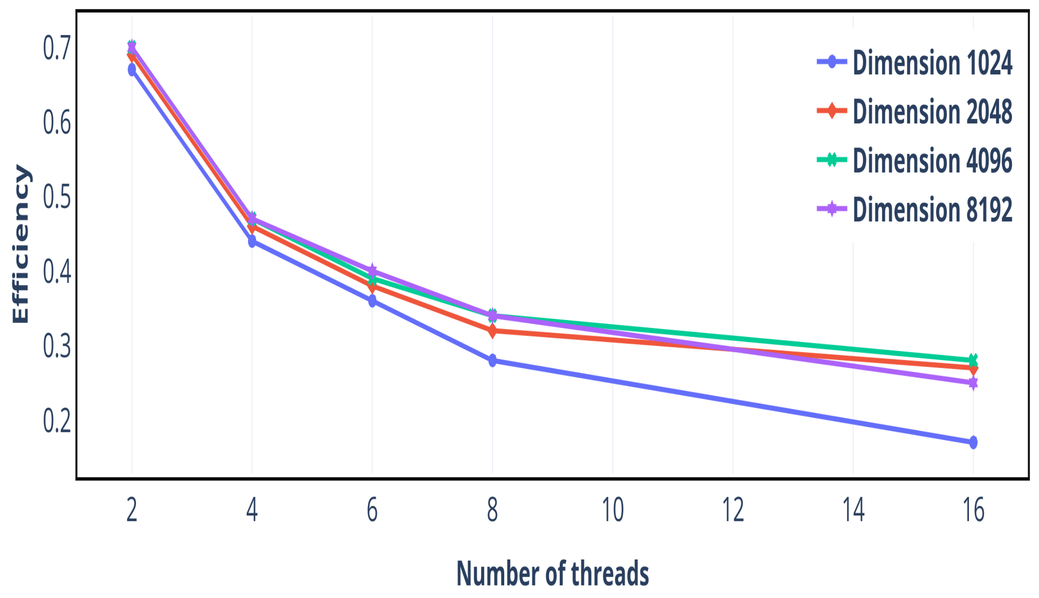

Figure 5.9: Efficiency for dense matrices with varying input size

In the above experiments, execution time was measured in seconds and `float` datatype was used in the experiments. As can be seen in the above figures, if we increase the number of threads for a fixed problem size, speedup increases and efficiency decreases. But if we increase the problem size while keeping the number of threads fixed, both speedup and efficiency increase.

Based on the above discussion, we can come to the conclusion that the implemented parallel program is scalable even though we are parallelizing the outer for-loop only.

Details of all the computational experiments in this chapter are presented in Appendix A.

Chapter 6

Conclusion and Future Work

In this thesis, we have presented a novel data structure that stores the matrix elements diagonally and matrix-matrix or matrix-vector multiplication algorithms that work by diagonals. This approach gives us an orientation independent framework. The storage scheme provides stride-1 access to the matrix elements which a key factor in achieving high performance. The transpose of a matrix requires no additional effort or data structure. Numerical experiments in Chapter 5 demonstrate that the methods presented in this thesis can be easily parallelized and show nice scalability.

Additionally, we have proposed two other implementations for the linear algebraic kernel operations of matrix-matrix multiplication and the focus was on the enhancement of locality reference using Java native arrays. This research approach is significant due to Java's lack of support for true rectangular arrays which makes it inconvenient for numerical computing. The results from the numerical testing demonstrate that diagonally structured storage and computation show great promise for efficient linear kernel operations using Java.

The methods we have presented in this thesis can be extended to further research projects such as GPU acceleration of diagonally structured linear algebra kernels and a block algorithm for dense matrix-matrix multiplication by diagonals.

Bibliography

[1] Nuerrennisahan Nurgul Aimaiti. A computational study of sparse or structured matrix operations, msc thesis, deaprtment of mathematics and computer science, university of lethbridge, alberta, canada, 2018.

[2] Nuerrennisahan (Nurgul) Aimaiti, Shahadat Hossain, and Mohammad Sakib Mahmud. Computational experience with diagonally structured linear algebra in java. Submitted in 2019.

[3] Anshul G. Ananth G. and Vipin K George K. *Introduction to Parallel Computing*. Pearson Education Limited, USA, 2nd edition, 2016.

[4] Richard John Anthony. Chapter 4 - the resource view. In *Systems Programming*, pages 203 – 276. Morgan Kaufmann, Boston, 2016.

[5] Blaise Barney. Parallel Computing. `Online:https://computing.llnl.gov/tutorials/parallel_comp/y`, (Accessed:2019-09-28).

[6] Ake Bj̈orck. *6. Direct Methods for Sparse Problems*, pages 215–268. 1996.

[7] Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, March 1996.

[8] Ronald F Boisvert, Jack J. Dongarra, Roldan Pozo, Karin A Remington, and GW Stewart. Developing numerical libraries in java. *Concurrency: Practice and Experience*, 10(11-13):1117–1129, 1998.

[9] Ronald F Boisvert, José Moreira, Michael Philippsen, and Roldan Pozo. Java and numerical computing. *Computing in Science & Engineering*, 3(2):18, 2001.

[10] Stephen Boyd and Lieven Vandenberghe. *Introduction to Applied Linear Algebra*. Cambridge University Press, New York, NY, USA, 2018.

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

[12] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 3rd edition, 2016.

[13] Aydin Bulu, John Gilbert, and Viral Shah. *Implementing Sparse Matrices for Graph Algorithms*, pages 287–314. 01 2011.

[14] Carlos Carvalho. The gap between processor and memory speeds. 01 2002.

[15] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[16] Francisco Heron de Carvalho Junior and Cenez Araújo Rezende. Performance evaluation of virtual execution environments for intensive computing on usual representations of multidimensional arrays. *Science of Computer Programming*, 132:29–49, 2016.

[17] Jack Dongarra and Aad J van der Steen. High-performance computing systems: Status and outlook. *Acta Numerica*, 21:379–474, 2012.

[18] Lori A. Freitag and James M. Ortega. The rscg algorithm on distributed memory architectures. Technical report, Charlottesville, VA, USA, 1992.

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

[20] Gier Gundersen. The use of java arrays in matrix computation, candidatus scientarium (master of science) thesis, 2002.

[21] Roger A Horn. *Topics in Matrix Analysis*. Cambridge University Press, New York, NY, USA, 1986.

[22] Shahadat Hossain and Mohammad Sakib Mahmud. On computing with diagonally structured matrices. In *Proceedings of The Twenty Third IEEE High Performance Extreme Computing Conference (HPEC 2019)*. IEEE, 2019.

[23] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley Professional, 2nd edition, 2012.

[24] Daniel Leuck and Patrick Niemeyer. *Learning Java*. O'Reilly Media, Inc., 2013.

[25] Mikel Lujan, Anila Usman, Patrick Hardie, TL Freeman, and John R Gurd. Storage formats for sparse matrices in java. In *International Conference on Computational Science*, pages 364–371. Springer, 2005.

[26] Mikel Lujn, Anila Usman, Patrick Hardie, Len Freeman, and John Gurd. Storage formats for sparse matrices in java. volume 3514, pages 364–371, 05 2005.

[27] 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.

[28] Memory layout of multi-dimensional arrays. `Online:https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays`, (Accessed:2019-09-24).

[29] Carl D. Meyer, editor. *Matrix Analysis and Applied Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[30] José E Moreira, Samuel P Midkiff, and Manish Gupta. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(2):265–295, 2000.

[31] Jose E Moreira, Samuel P Midkiff, and Manish Gupta. Supporting multidimensional arrays in java. *Concurrency and Computation: Practice and Experience*, 15(3-5):317–340, 2003.

[32] Igor Pechtchanski and Vivek Sarkar. Immutability specification and its applications. *Concurrency and Computation: Practice and Experience*, 17(5-6):639–662, 2005.

[33] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[34] Thomas Stricker and T Cross. Global address space, non-uniform bandwidth: A memory system performance characterization of parallel systems. In *Proceedings Third International Symposium on High-Performance Computer Architecture*, pages 168–179. IEEE, 1997.

[35] A. Tsao and T. Turnbull. A comparison of algorithms for banded matrix multiplication. Technical report, Supercomputing Research Centre, 1993.

[36] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, et al. Titanium: a high-performance java dialect. *Concurrency and Computation: Practice and Experience*, 10(11-13):825–836, 1998.

Appendix A

Tables of Numerical Experiments

In this appendix, we present the numerical details of computational results associated with Chapter 4 and Chapter 5. Execution time was measured in seconds and milliseconds. Matrix dimension was started at a predetermined number and was increased gradually until memory exception occurred. Speedup and Efficiency for parallel implementation of the algorithms was computed respectively using equation (3.1) and equation (3.2) in Chapter 3 and the number of threads deployed for parallel implementation varies from 2 to 16. SMM stands for standard multiplication method and DMM stands for diagonal multiplication method as referred in Chapter 4. V1 refers to the partial row/column-oriented $(i-j-k)$, V5 $(k-i-j)$ and V6 $(i-k-j)$ refers to the pure row-oriented versions of matrix-matrix multiplication already discussed in Chapter 4.

A.1 Performance of Dense Matrix-Vector Multiplication

Table A.1: Ax and A^Tx on SMM versus DMM

Matrix Dimension (n)	Execution Time (ms)			
	Ax (SMM)	Ax(DMM)	A^Tx (SMM)	A^Tx (DMM)
9000	188	218	875	250
10000	250	281	1090	281
11000	313	344	1312	375
12000	375	437	1593	406
13000	438	500	1906	500
14000	531	563	2125	594
15000	563	656	2562	688
16000	656	750	4750	750
17000	750	813	3625	844
18000	843	938	4090	938

Table A.1 represents the numerical details for matrix-vector multiplication using SMM and DMM. Execution time was measured in milliseconds (ms) and matrix dimension varies from (9000,...,18000).

A.2 Performance Measure of Dense Matrix-Matrix Multiplication on AB

Table A.2: V1 vs V5 vs V6 vs DIAS for AB

Matrix Dimension (n)	Execution Time (s)			
	AB (ijk)	AB(ikj)	AB(kij)	AB (DMM)
1000	2.38	1.5	1.5	1.63
2000	52.38	12.06	12.39	13.25
3000	202.25	40.88	41.56	42.87
4000	493.63	97	97.43	99.01
5000	1113.96	190.88	193.50	193
6000	1970	325.5	329.06	333.93

Table A.2 represents the numerical details for dense matrix-matrix multiplication (AB) using V1,V5,V6, and DMM. Execution time was measured in seconds and matrix dimension varies from (1000,...,6000).

A.3 Performance Measure of Dense Matrix-Matrix Multiplication A^TB

Table A.3: V1 vs V5 vs V6 vs DIAS on A^TB

Matrix Dimension (n)	Execution Time (s)			
	A^TB (ijk)	A^TB(ikj)	A^TB(kij)	A^TB (DMM)
1000	2.38	1.5	1.5	1.63
2000	52.38	12.06	12.39	13.25
3000	202.25	40.88	41.56	42.87
4000	493.63	97	97.43	99.01
5000	1113.96	190.88	193.50	193
6000	1970	325.5	329.06	333.93

Table A.3 represents the numerical details for A^TB using V1,V5,V6, and DMM. Execution time was measured in seconds and matrix dimension varies from (1000,...,6000).

A.4 Performance of Banded Matrix-matrix Multiplication using Diagonal Storage on Java Jagged Array and Java 1-D array

Table A.4: 1-D array diag vs Jagged array diag for dimension 8000

Bandwidth	Execution Time (s)	
	1-D Array Diag	Jagged Array Diag
501	2.89	2.30
1001	11.20	8.94
2001	41.60	29.69
3001	90.83	72.47
4001	149.62	119.92
5001	222.83	153.17
6001	299.93	207.83
7001	394.16	298.42
8001	461.954	362.03
9001	555.58	386.41
10001	601.28	417.31
11001	659.72	456.27
12001	703.32	491.35
13001	743.86	601.57
14001	Out of Memory	631.09

In Table A.4, Matrix elements were stored using Java native arrays and Java jagged arrays. Matrix dimension in this above experiment was set at 8000 $\times$ 8000 and bandwidth varies from ($k_b = 501, \ldots, 13001$), until JVM out-of-memory exception is encountered. Execution time was measured in seconds.

A.5 Speedup Comparison of Java 1-D Array Diagonal, Jagged Array Diagonal and Java Sparse Array relative to CRS

Table A.5: Speedup with respect to CRS for dimension 50000 with increasing bandwidth

Bandwidth	1-D Array Diag	Jagged Array Diag	JSA
201	1.52	1.78	1.07
401	1.54	1.80	1.06
601	1.52	1.86	1.08
801	1.51	1.84	1.06
1001	1.57	2.17	1.06
1201	1.63	1.99	1.09

Table A.5 represents the performance comparison among three data structures that are Java 1-D array diag, Java jagged array diag and Java sparse array with respect to CRS. The matrix size used in the above experiment was 50000×50000 and the bandwidth was increased in a step of 200.

A.6 Speedup obtained by diagonal matrix-matrix multiplication method for banded matrices

Table A.6: Speedup data for matrix dimension 100000 with different chunk size

B is denoted as bandwidth

Dimension	Chunk size	Number of Threads	Speedup		
			B=101	B= 201	B= 801
100000	15	2	1.73	1.74	1.83
		4	3.13	3.50	3.52
		6	3.13	3.91	4.70
		8	3.19	4.50	5.92
		10	3.19	4.45	7.12
		12	3.19	4.50	8.29
		14	2.68	4.47	9.39
		16	3.30	4.50	10.00
100000	30	2	1.81	1.80	1.84
		4	1.78	3.26	3.62
		6	1.79	3.13	4.64
		8	1.78	3.16	5.72
		10	1.78	2.84	6.78
		12	1.82	2.97	8.21
		14	1.78	2.80	9.22
		16	1.80	2.82	9.27
100000	60	2	1.04	1.67	1.81
		4	1.08	1.68	3.38
		6	1.12	1.68	4.63
		8	1.13	1.68	5.28
		10	1.04	1.68	5.36
		12	1.10	1.68	5.31
		14	1.05	1.65	5.30
		16	1.03	1.67	5.33

We experimented with matrix dimension 100000 with different chunk sizes while varying bandwidth from 101 to 801. Number of threads was increased with a step of 2. Chunk size 15 yields the best result that is what we presented in Chapter 5.

A.7 Efficiency obtained by diagonal matrix-matrix multiplication method for banded matrices

Table A.7: Efficiency data for matrix dimension 100000 with different chunk size

B is denoted as bandwidth

Dimension	Chunk size	Number of Threads	Efficiency		
			B=101	B= 201	B= 801
100000	15	2	0.84	0.89	0.91
		4	0.78	0.88	0.88
		6	0.52	0.65	0.78
		8	0.40	0.56	0.74
		10	0.32	0.44	0.71
		12	0.26	0.38	0.69
		14	0.19	0.32	0.67
		16	0.21	0.28	0.63
100000	30	2	0.90	0.90	0.92
		4	0.45	0.82	0.90
		6	0.30	0.52	0.77
		8	0.22	0.39	0.72
		10	0.18	0.31	0.68
		12	0.15	0.25	0.68
		14	0.13	0.23	0.65
		16	0.11	0.18	0.58
100000	60	2	0.52	0.84	0.91
		4	0.27	0.42	0.85
		6	0.19	0.28	0.77
		8	0.13	0.21	0.66
		10	0.10	0.17	0.54
		12	0.09	0.12	0.44
		14	0.07	0.10	0.38
		16	0.06	0.10	0.33

We experimented with matrix dimension 100000 with different chunk sizes while varying bandwidth from 101 to 801. Number of threads was increased with a step of 2. Chunk size 15 yields the best result that is what we presented in Chapter 5.

A.8 Speedup obtained by diagonal matrix-matrix multiplication method for dense matrices

Table A.8: Speedup data for various matrix sizes

Number of Threads	Dimension			
	1024	2048	4096	8192
2	0.67	0.69	0.70	0.70
4	0.88	0.91	0.94	0.94
6	1.44	1.52	1.57	1.59
8	2.32	2.55	2.68	2.76
16	2.76	4.31	4.55	4.8

Table A.8 represents the parallel speedup obtained by diagonal multiplication method for dense matrices.

A.9 Efficiency obtained by diagonal matrix-matrix multiplication method for dense matrices

Table A.9: Efficiency data for various matrix sizes

Number of Threads	Dimension			
	1024	2048	4096	8192
2	0.67	0.69	0.70	0.70
4	0.44	0.46	0.47	0.48
6	0.36	0.38	0.39	0.40
8	0.28	0.32	0.34	0.34
16	0.17	0.27	0.28	0.25

Table A.9 represents the parallel efficiency obtained by diagonal multiplication method for dense matrices.