ASSIGNMENT OF MASTER'S THESIS

Title:	Parallel Joint Direct and Transposed Sparse Matrix-Vector Multiplication
Student:	Bc. Claudio Kozický
Supervisor:	doc. Ing. Ivan Šimeček, Ph.D.
Study Programme:	Informatics
Study Branch:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2019/20

Instructions

1. Review existing sparse matrix storage formats.
2. Review existing approaches to parallel joint direct and transposed sparse matrix-vector multiplication (SpMMTV) for CPUs and GPUs on shared memory systems.
3. Implement a parallel multilayer approach to SpMMTV for CPUs as described in [1].4. Discuss possible optimizations of the implemented algorithm.
5. Implement some of the discussed optimizations and measure the resulting speedup.
6. Compare the performance of the implementation with similarly focused libraries.

References

[1] Šimeček I., Langr D., Kotenkov I. (2018) Multilayer Approach for Joint Direct and Transposed Sparse Matrix Vector Multiplication for Multithreaded CPUs. In: Wyrzykowski R., Dongarra J., Deelman E., Karczewski K. (eds) Parallel Processing and Applied Mathematics. PPAM 2017. Lecture Notes in Computer Science, vol 10777. Springer, Cham

doc. Ing. Jan Janoušek, Ph.D.	doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Head of Department	Dean

Prague January 7, 2019

Master's thesis

Parallel Joint Direct and Transposed Sparse Matrix-Vector Multiplication

Bc. Claudio Kozický

Department of Theoretical Computer Science
Supervisor: doc. Ing. Ivan Šimeček, Ph.D.

9th May 2019

Acknowledgements

I would like to thank my supervisor for the assistance he provided when I was working on this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 9th May 2019 …………………

Czech Technical University in Prague
Faculty of Information Technology
© 2019 Claudio Kozický. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Kozický, Claudio. *Parallel Joint Direct and Transposed Sparse Matrix-Vector Multiplication.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Tato práce se zabývá tím, jak sloučit násobení řídké matice vektorem a násobení transponované matice vektorem do jediné operace, která se jmenuje spojené násobení řídké matice vektorem (SpMM$^{\mathrm{T}}$V). Dále se zabývá paralelizací této sloučené operace. Paralelní SpMM$^{\mathrm{T}}$V lze využít ke zrychlení metody bikonjugovaných gradientů, což je iterativní algoritmus pro řešení rozsáhlých řídkých soustav lineárních rovnic. Práce zkoumá stávající formáty pro ukládání řídkých matic a stávající přístupy paralelního SpMM$^{\mathrm{T}}$V. Je vyvinuta a podrobně popsána implementace SpMM$^{\mathrm{T}}$V pro vícejádrové procesory na systémech se sdílenou pamětí. U vyvinutých implementací jsou diskutovány možnosti optimalizace. Některé z diskutovaných optimalizací jsou rovněž implementovány. Jsou porovnány výkonnosti výsledných implementací SpMM$^{\mathrm{T}}$V a srovnány s implementací založenou na knihovně Intel Math Kernel Library.

Klíčová slova metoda bikonjugovaných gradientů, násobení řídké matice vektorem, OpenMP, paralelizace, řídká matice.

Abstract

This thesis focuses on investigating approaches to combining sparse matrix-vector multiplication and transposed sparse matrix-vector multiplication into a single operation called *joint direct and transposed sparse matrix-vector multiplication* (SpMM$^{\mathrm{T}}$V). It also focuses on parallelising this joint operation. A parallel SpMM$^{\mathrm{T}}$V operation can be used to speed up the *biconjugate gradient method*, which is an iterative algorithm for solving large sparse systems of linear equations. Existing sparse matrix storage formats and existing approaches to parallel SpMM$^{\mathrm{T}}$V are examined in this thesis. Parallel SpMM$^{\mathrm{T}}$V implementations for CPUs on shared memory systems are developed and throughly described. Optimisations of these implementations are discussed and some of them are implemented. The resulting performance of developed SpMM$^{\mathrm{T}}$V implementations is compared with an implementation based on the Intel Math Kernel Library.

Keywords biconjugate gradient method, OpenMP, parallelisation, sparse matrix, sparse matrix-vector multiplication.

Contents

Introduction **1**

1 Sparse Matrix Storage Formats **3**
1.1 Sparse Matrices . . . 3
1.2 Coordinate Format . . . 4
1.3 Compressed Sparse Row Format . . . 4
1.4 Compressed Sparse Columns Format . . . 5
1.5 Hierarchical Storage Formats . . . 5
1.6 Other Storage Formats . . . 6

2 Direct and Transposed Sparse Matrix-Vector Multiplication **7**
2.1 Background . . . 7
2.2 Existing Approaches . . . 8

3 Selected Approach **11**
3.1 Sparse Matrix Storage Format . . . 11
3.2 Parallelisation . . . 12
3.3 Optimisations . . . 13
3.3.1 Keeping Track of Finished Regions . . . 13
3.3.2 Reducing Inter-Thread Communication . . . 14
3.3.3 Symmetry Related Optimisations . . . 14

4 Hardware and Software Aspects **17**
4.1 OpenMP . . . 17
4.2 SIMD . . . 17
4.3 Math Kernel Library . . . 18

5 Implementation **19**
5.1 Sparse Matrix Storage Formats . . . 19
5.1.1 Coordinate and Compressed Sparse Row Formats . . . 20

5.1.2 Two-Level Hierarchical Storage Format 20
5.2 Matrix Construction Process 21
5.3 Computational Kernels 22
5.4 Parallelisation Methods 23
5.4.1 Traversing Region Indices 24
5.4.2 SpMM^{T}V Without Planning 24
5.4.3 Disjoint SpMM^{T}V Without Planning 25
5.4.4 Dynamic Planning 25
5.5 Iterability Considerations 26
5.6 Implementation Specific Optimisations 26
5.7 Additional Utilities 27

6 Performance Evaluation **29**
6.1 Test Environment 29
6.2 Input Sparse Matrices 30
6.3 Measurement Approach 30
6.4 Results 31

7 Conclusion **39**

Bibliography **41**

A Building and Running **45**
A.1 Build Dependencies 45
A.2 Building Executables 46
A.3 Running the Main Executable 47
A.4 Running the Test Executable 48
A.5 Additional Utilities 49

B Contents of Attached Disc **51**

List of Figures

6.1 Performance of parallel disjoint SpMM^{T}V without planning 32
6.2 Performance of parallel SpMM^{T}V without planning 33
6.3 Average parallel speedups of SpMM^{T}V implementations 34
6.4 Parallel speedup of dynamically planned disjoint SpMM^{T}V 35
6.5 Parallel speedup of dynamically planned SpMM^{T}V 35
6.6 Sequential SpMM^{T}V performance comparison 36
6.7 Average parallel performance of SpMM^{T}V implementations 37

List of Listings

5.1 Kernel for SpMM^{T}V with a COO formatted region 23
5.2 Kernel for SpMM^{T}V with a CSR formatted region 24

Introduction

Joint direct and transposed sparse matrix-vector multiplication (SpMM^{T}V) is a computational kernel that combines *sparse matrix-vector multiplication* (SpMV) and *transposed sparse matrix-vector multiplication* (SpM^{T}V) into a single joint operation. An SpMM^{T}V operation's input consists of a sparse matrix and two vectors; its output consists of two vectors. Both pairs of input and output vectors correspond to input and output vectors of the underlying SpMV and SpM^{T}V operations. SpMM^{T}V is an operation which can be performed in parallel just like SpMV and SpM^{T}V.

The *biconjugate gradient method* is an iterative algorithm that can be used to solve large sparse systems of linear equations [1, 2]. A significant portion of the algorithm's run-time is spent performing SpMV and SpM^{T}V. Executing the matrix-vector products as a single joint operation (i.e. as SpMM^{T}V) can lead to significant speed gains.

This thesis intends to:

1. Review existing sparse matrix storage formats.

2. Review existing parallel SpMM^{T}V approaches for CPUs and GPUs on shared memory systems.

3. Implement the approach to parallel SpMM^{T}V presented in [3] and discuss possible optimisations.

4. Implement some of the discussed optimisations and measure the resulting speedup.

5. Compare the implementation's performance with existing libraries.

Chapter **1**

Sparse Matrix Storage Formats

This chapter introduces sparse matrices (Section 1.1) and describes relevant sparse matrix storage formats. The most common sparse matrix storage formats, namely the Coordinate format, the Compressed Sparse Row format and the Compressed Sparse Column format, are covered in Sections 1.2, 1.3 and 1.4 respectively. Hierarchical storage formats for sparse matrices, a major focus of this thesis, are described in Section 1.5. Other interesting sparse matrix storage formats are briefly covered in Section 1.6.

1.1 Sparse Matrices

In mathematics, the term *matrix* can refer to a rectangular array of numbers [4]. It can be said that the matrix's numbers are arranged in *rows* and *columns*. The numbers of a matrix can be also referred to as the matrix's *elements*. The following text refers to the matrix's *topmost row* and *leftmost column* as to its *first row* and *first column* respectively. Analogously, the matrix's *bottom-most row* and *rightmost column* are respectively referred to as the *last row* and *last column*.

A straightforward matrix storage scheme is to store all matrix elements consecutively into a one or two-dimensional array. In order to facilitate easy access to the stored matrix's elements, it is desirable to order the elements in the array in a predictable manner. One way to order elements in an array is to start with the matrix's first row and store all of the row's elements from the first to the last column. Those elements are similarly followed by elements from the second row, third row and so forth. Such an ordering of elements in an array is often referred to as *row-major order*. A complementary approach called *column-major order* is to start with the first column, order its elements from the first to last row, and analogously to continue with the second column, third column, etc. Matrices stored in either of the described ways are further called *dense matrices*.

In some cases it may be advantageous not to store *all* elements of a matrix. Such matrices are called *sparse matrices* [5]. The values of the omitted elements are usually implicitly known. This thesis deals with sparse matrices where the left out elements are assumed to be equal to zero – the sparse matrices contain only nonzero elements. Said matrices arise in many fields, for example structural engineering, computational fluid dynamics, computer vision, optimisation, economic and financial modelling or chemical process simulation [6]. The nonzero element count of sparse matrices from these fields can be orders of magnitude lower than the total number of matrix elements.

When performing mathematical operations with matrices that predominantly consist of zeros, a significant portion of computation time can be spent on adding or multiplying zeros (e.g. in the case of matrix multiplication). Taking advantage of sparse matrix representations in such cases can lead to significant time, memory and/or storage savings.

In the rest of this chapter, M is used to refer to a sparse matrix M with r rows and c columns. The matrix consists of n_{nz} nonzero elements and n elements in total. All sparse matrix formats in this chapter only store the matrix's nonzero elements.

1.2 Coordinate Format

The so-called *Coordinate* format, commonly abbreviated as COO, is a very simple sparse matrix storage format. The data structure usually consists of three arrays of length n_{nz}, namely `vals`, `row_inds` and `col_inds`. Array `vals` contains the values of a matrix's nonzero elements. Arrays `row_inds` and `col_inds` contain the nonzero elements' row and column indices respectively. The elements in the arrays are arranged so that the ith elements of all arrays contain information about the same matrix element. [1]

The format doesn't prescribe whether the order of nonzero elements in the matrix should be reflected in the format's arrays. The ith element of the three arrays doesn't necessarily have to correspond to the ith nonzero element of matrix M. [1]

1.3 Compressed Sparse Row Format

The *Compressed Sparse Row* format (abbr. as CSR) [1] is sometimes referred to as the *Compressed Row Storage* format (abbr. as CRS) [2]. The data structure usually consists of arrays `vals`, `row_ptrs` and `col_inds`. Arrays `vals` and `col_inds` correspond to the identically named arrays of the COO format described in Section 1.2. Arrays `vals` and `col_inds` contain n_{nz} elements whereas array `row_ptrs` contains $r + 1$ elements.

The CSR format specifies that a matrix's nonzero elements should be stored in arrays `vals` and `col_inds` in row-major order. The two arrays

therefore first contain values or column indices of nonzero elements in the matrix's first row, those are followed by nonzero elements in the second row and so forth. Array `row_ptrs` indicates (i.e. *points* to) the starting positions of each row in arrays `vals` and `col_inds`. The ith element of `row_ptrs` contains the starting index of the ith row in arrays `vals` and `col_inds`. The $i+1^{\text{st}}$ element of `row_ptrs` contains the starting index of the subsequent row, which corresponds to the one-after-the-last index of the previous row. Consequentially, the first element of array `row_ptrs` is equal to zero and the last (i.e. the $r+1^{\text{st}}$) element of the array is equal to n_{nz}. A sparse matrix stored in the aforementioned way may occupy significantly less space than if it had been stored in COO. [1, 2]

1.4 Compressed Sparse Columns Format

The *Compressed Sparse Columns* format (abbr. as CSC) [1] is sometimes referred to as the *Compressed Column Storage* format (abbr. as CCS) [2]. This format is very similar to the CSR format described in Section 1.3 but compresses columns in place of rows. The data structure usually consists of arrays `vals`, `row_inds` and `col_ptrs`. Array `vals` contains the values of a matrix's nonzero elements and array `row_inds` contains the nonzero elements' column indices. Both arrays consist of n_{nz} elements in column-major order. Array `col_ptrs` contains $c+1$ elements and indicates the starting positions of each column in the previous two arrays.

1.5 Hierarchical Storage Formats

The so-called *hierarchical storage formats* (HSFs) for sparse matrices are a major focus of this thesis. They were introduced in [7] and have been investigated further in [8, 9, 3], among others. HSFs focus on decreasing the amount of storage occupied by a matrix's element indices. This is achieved by hierarchically partitioning matrices into contiguous blocks and subblocks combined with sharing certain portions of bits of individual indices among blocks.

Row and column indices of all elements of a matrix can be represented with a certain amount of bits. If we partition the matrix into blocks, it can be observed that the bit representation of row and column element indices within a block can be divided into two parts: (1) the most significant bits, which are shared among all elements in a block and (2) the least significant bits, which are unique for each block's element. Storing the shared bits only once per block and using a smaller data type to store every element's unique least significant bits can substantially lower the matrix's size [7]. Several levels of blocks can be created by repeating the described process [9].

HSFs differ by their level counts and by the matrix format used in each level [9]. Blocks of a specific level can use the COO format (Section 1.2), the CSR format (Section 1.3) or the CSC format (Section 1.4) [8]. A level can also store elements as a dense matrix (i.e. store all elements to an array in a particular order) or a bitmap (i.e. combine an array of element values with a bitmap to indicate their positions). The efficiency of a specific HSF highly depends on the chosen level count and the format used in each level.

Basic HSFs include the COOCOO format, the COOCSR format, the CSR-COO format and the CSRCSR format. These are two-level formats that use either the COO format or the CSR format in each of their levels. The amount of bits which the mentioned HSFs use to store element indices on each level is given by a parameter. [7]

The COOCOO format has been generalised to more than two levels – $(\mathrm{COO}_k)^l$ denotes an l-level HSF that uses the COO format with k-bit element indices in each level. The $(\mathrm{COO}_8)^4$ format is an example of such a format. [9]

Instead of assigning a single matrix format to every level of an HSF beforehand, each block of every level can use a different format depending on the properties of the block's elements. The *Adaptive-Blocking Hierarchical Storage Format* (ABHSF) is a two-level format, which uses the COO format in its first level and selects between using a dense matrix, a bitmap, the COO format or the CSR format in its second level. [8]

1.6 Other Storage Formats

The *Block Compressed Row Storage* format (abbr. as BCRS) focuses on efficiently storing sparse matrices that contain dense blocks of nonzero elements [2]. The format can be viewed as a two-level hierarchical storage format (Section 1.5) which uses the CSR format in its first level and a dense matrix (i.e. an array with all matrix elements in a particular order) in its second level [8].

Other storage formats target matrices whose elements are concentrated in a narrow band around the diagonal. Examples of such formats are the *Compressed Diagonal Storage* format (abbr. as CDS) [2], the *Skyline Storage* format (abbr. as SKS) or the *Ellpack-Itpack* format [1].

The text-based *Matrix Market* format (abbr. as MM) was designed to be simple and extensible. It is aimed to serve as an exchange format, not as a processing format. The format supports both dense and sparse matrices. [10]

Most, if not all, matrix storage formats can decrease the memory requirements of symmetric matrices by only storing half of their elements. Many other sparse matrix formats have been proposed but listing all of them is beyond the scope of this thesis. A survey of modern sparse matrix formats is available in [11].

Chapter **2**

Direct and Transposed Sparse Matrix-Vector Multiplication

This chapter introduces and explains an operation called *joint direct and transposed sparse matrix-vector multiplication* (SpMM$^{\mathrm{T}}$V). Section 2.1 explains the rationale behind such an operation. Existing approaches to SpMM$^{\mathrm{T}}$V are reviewed in Section 2.2.

2.1 Background

A system of linear equations can be written as $Ax = b$, where A is the coefficient matrix, x is the vector of unknowns and b is the right-hand side vector. A is an $n \times n$ matrix; x and b are both column vectors with n entries. [1]

Large sparse systems of linear equations are commonly solved using iterative methods [1, 2]. One such method is the *biconjugate gradient method* (abbr. as BiCG). When calculating a solution using BiCG, a significant portion of time is spent performing sparse matrix-vector multiplication – first with the original matrix and then with its transpose, that is $p_1 = Aq_1$ and $p_2 = A^{\mathrm{T}}q_2$.

Sparse matrix-vector multiplication (SpMV) is a widely used computational kernel [12]. The performance of SpMV is limited by memory bandwidth on modern multi-core and many-core architectures [11]. *Transposed sparse matrix-vector multiplication* (SpM$^{\mathrm{T}}$V) is a very similar computational kernel and shares the same properties.

There are several straightforward approaches to computing the SpMV and SpM$^{\mathrm{T}}$V operations in BiCG. One option is to create a transposed copy of the coefficient matrix and perform two SpMV operations in succession. A drawback of this approach is that storing the coefficient matrix of a large sparse system of linear equations twice in memory may entail prohibitively high memory requirements. Another option is to store the matrix in memory only once and compute SpMV followed by SpM$^{\mathrm{T}}$V. This approach keeps another disadvant-

age – the coefficient matrix is loaded from memory twice, which is especially inefficient considering SpMV and SpM$^{\mathrm{T}}$V are memory bound operations. [3]

An efficient approach to computing SpMV and SpM$^{\mathrm{T}}$V in BiCG is to perform both operations as a single joint (or fused) operation, thus loading the coefficient matrix from memory only once. Such an approach is called *joint direct and transposed sparse matrix-vector multiplication* (SpMM$^{\mathrm{T}}$V) and is a major focus of this thesis. [3]

2.2 Existing Approaches

This section reviews existing approaches to parallel joint direct and transposed sparse matrix-vector multiplication (SpMM$^{\mathrm{T}}$V) for CPUs and GPUs on shared memory systems. Performing SpMM$^{\mathrm{T}}$V, i.e. performing SpMV and SpM$^{\mathrm{T}}$V as a single fused operation as described in [3], seems to be largely unexplored. Current research usually covers performing operations SpMV or SpM$^{\mathrm{T}}$V by themselves or performing both of them in succession.

Slightly different operations called *SpMM* and *SpMM_T* are investigated in [13]. SpMM is the product of a matrix and multiple vectors, whereas SpMM_T is the product of a transposed matrix and multiple vectors. A parallel implementation for multi-core processors of both operations is presented. The article explores performing both operations in succession, not as a single fused operation. The article makes use of the *Compressed Sparse Blocks* sparse matrix storage format (abbr. as CSB). CSB could be viewed as a hierarchical storage format (Section 1.5) that uses the COO format in its second level.

The approach presented in [13] is adapted for graphics processing units, specifically for Nvidia GPUs, in [14]. The CSB format is redesigned as the *extended CSB* format (abbr. as eCSB). Like the CSB format, eCSB can be viewed as a hierarchical storage format (Section 1.5). A heuristic is used to choose between several formats for the second level of eCSB, namely the COO format, the Ellpack-Itpack format and a hybrid format which combines the COO and Ellpack-Itpack formats. The article takes advantage of the eCSB format to achieve similar performance for SpMV and SpM$^{\mathrm{T}}$V. The possibility of performing a single fused operation is not examined.

A parallel implementation for many-core processors, specifically Intel Xeon Phi processors, is presented in [15]. The article focuses on a slightly different pair of operations, namely on $z = A^{\mathrm{T}}x$ and $y = Az$. The authors of [15] acknowledge that certain algorithms, e.g. the biconjugate gradient method, perform matrix-vector multiplication in a way that can be fused. But such a fused version, which corresponds to SpMM$^{\mathrm{T}}$V as described in [3], is not investigated in detail.

Article [3] also mentions the approach presented in [16], which uses the *Recursive Sparse Blocks* format (abbr. as RSB). RSB is a recursive tree-based sparse matrix storage format that uses either the COO or CSR format in its

leaf nodes. Article [16] focuses on efficient parallel execution of SpMV and $\text{SpM}^{\text{T}}\text{V}$; the fused operation was not examined.

Chapter 3

Selected Approach

This chapter describes the parallel approach to *joint direct and transposed sparse matrix-vector multiplication* (SpMM$^{\mathrm{T}}$V) as presented in [3]. The used sparse matrix storage format is explained in Section 3.1 and three ways of performing parallel SpMM$^{\mathrm{T}}$V are listed in Section 3.2. Section 3.3 outlines some possible optimisations of the introduced SpMM$^{\mathrm{T}}$V operation. The implementation presented in Chapter 5 of this thesis is based on the ideas described in this section.

3.1 Sparse Matrix Storage Format

The implementation in [3] supports sparse matrices with single-precision floating-point numbers. A 32-bit unsigned integer data type is used to store row and column indices of a matrix's nonzero elements. This limits the maximum dimensions of a matrix to $2^{32} \times 2^{32}$. At the time of writing, no matrices from the *SuiteSparse Matrix Collection* [6], which serves as a source of test matrices for this thesis, exceed this limit. While lowering the maximum dimensions of input matrices doesn't seem significantly limiting, using a smaller data type for row and column indices substantially reduces the size of stored matrices. As was mentioned in Section 2.1, SpMM$^{\mathrm{T}}$V is a memory bound operation which significantly benefits from decreasing the amount of transferred data.

Sparse matrices are stored in memory using a two-level hierarchical storage format (Section 1.5). The format makes use of the COO format (Section 1.2) and the CSR format (Section 1.3). The hierarchical format uses COO in its first level. Its second level uses either COO or CSR. A loaded matrix is split into regions where each region contains at most 2^{16} rows and 2^{16} columns. Row and column indices of created regions are renumbered so that the topmost, leftmost region element has indices equal to $(0, 0)$. Because of this renumbering, row and column indices within a region can be stored using a 16-bit unsigned integer data type, which further decreases the stored matrix's

size. The renumbered elements are stored in the second level of the hierarchical storage format. A region of renumbered elements is stored using the CSR format if the region's nonzero element count is greater than or equal to the region's row count. Otherwise it is stored in the COO format. The hierarchical format's first level contains row and column indices of every region's topmost, leftmost element. These row and column indices correspond to indices from the original matrix. When performing SpMM$^{\mathrm{T}}$V with a region, original row and column element indices are computed from indices stored in both levels of the hierarchical format.

The nonzero element counts of regions created in the aforementioned manner can differ significantly. Regions with more nonzero elements than a given threshold are replaced with a certain amount of smaller regions. The threshold θ is equal to $\alpha n_{nz}/\tau$, where $\alpha \in (0, 1]$ is a command-line parameter, n_{nz} is the matrix's nonzero element count and τ is the thread count used for parallel SpMM$^{\mathrm{T}}$V. After this process, which [3] calls *region normalisation*, each region contains less than θ nonzero elements.

The used hierarchical storage format contains regions in a specific order. This order is used to traverse regions when performing SpMM$^{\mathrm{T}}$V. One option is to sort the regions in row-major order, which corresponds to the order used in the input, CSR formatted matrix. This ordering is referred to as *lexicographical order* in the implementation from [3]. Sorting the regions according to the so-called Morton order or Z-order is also suggested. This ordering should improve memory access locality and maximise cache utilisation.

Every CSR region may have empty leading or trailing rows, which are traversed redundantly when performing SpMM$^{\mathrm{T}}$V. A preprocessing step is implemented to detect and remove such rows.

3.2 Parallelisation

The parallel SpMM$^{\mathrm{T}}$V operation implemented in [3] expects a square sparse matrix in the hierarchical format described in Section 3.1. It also expects a single vector stored as a dense array. This vector needs to have appropriate dimensions so it can be multiplied with the matrix. The outputs of the SpMM$^{\mathrm{T}}$V operation are two vectors stored as dense arrays. One vector is the result of performing SpMV, the other is the result of performing SpM$^{\mathrm{T}}$V.

Each region contains elements from a given subsequence of row and column indices of the input matrix. These indices determine which elements of the output vectors are updated when performing SpMM$^{\mathrm{T}}$V with a region. Parallel execution of SpMM$^{\mathrm{T}}$V operations introduces the need to ensure that the same subregions of output vectors are not written to simultaneously.

Three synchronisation approaches are proposed in [3]. The first method is called *SpMM^{T}V without planning*. In this approach an idle thread first tries to acquire the output locks of an unfinished region. If the thread fails to acquire

the locks, it continues to the next unfinished region and repeats the attempt. If the thread acquires the locks, it marks the region as finished, performs an SpMM^{T}V operation with the region, releases the locks and attempts to acquire the locks of the next unfinished region.

The second method is called *statically planned SpMM^{T}V*. The aim of this approach is to eliminate the overhead of searching for an unfinished region. The execution time of SpMM^{T}V with each region is modelled and an execution plan is built using a greedy algorithm. The finished execution plan contains a list of regions for every thread. When performing SpMM^{T}V according to the execution plan, a thread acquires a region's output locks, performs SpMM^{T}V with the region, releases the locks and continues with the next region in the list. Each thread performs SpMM^{T}V with regions in the order given by the execution plan.

The third method is called *dynamically planned SpMM^{T}V*. This approach combines the advantages of SpMM^{T}V without planning and of statically planned SpMM^{T}V. SpMM^{T}V without planning is run once and every thread's finished regions are recorded. An execution plan is created from the recorded region ordering.

3.3 Optimisations

This section outlines several possible optimisations of the SpMM^{T}V approach described in Sections 3.1 and 3.2. Section 3.3.1 describes an alternative way of keeping track of finished regions. Two parallel SpMM^{T}V execution methods, which reduce the amount of inter-thread communication, are suggested in Section 3.3.2. Section 3.3.3 describes how symmetric matrices can be taken advantage of.

3.3.1 Keeping Track of Finished Regions

A parallel execution method called *SpMM^{T}V without planning* is presented in [3] and outlined in Section 3.2. In this approach each thread continuously loops over all regions, whether finished or not, and tries to acquire a region's output locks. Threads use a shared counter, which keeps track of how many regions have been completed, to determine when to stop looking for new regions. A disadvantage of such a method is that all threads keep iterating over already finished regions.

An alternative approach is proposed – each thread can use a linked list-like data structure to keep track of finished regions. Once a thread finishes a region or discovers that a region has already been completed, the thread can remove that region from its linked list in constant time. All elements of this linked list should be stored in a single array to improve memory access locality.

3.3.2 Reducing Inter-Thread Communication

When performing SpMM$^{\mathrm{T}}$V without planning, threads need a shared data structure to keep track of which regions have or have not been completed. It can be assumed that access to such a data structure will require synchronisation. Several aspects may influence the overhead related to this data structure, e.g. the way the data structure is accessed, the data structure's implementation or the amount of used threads. An alternative approach would be to assign disjoint sets of regions to each thread before commencing SpMM$^{\mathrm{T}}$V. As a result threads would no longer need to communicate which regions have been completed. A drawback of such an approach is that it depends on each thread completing its set of regions in approximately the same time.

A parallel execution method named *dynamically planned SpMM^{T}V* is presented in [3] and outlined in Section 3.2. The optimisation introduced in this section, i.e. removing the need of inter-thread communication, can also benefit from dynamic planning. This would mean that SpMM$^{\mathrm{T}}$V without inter-thread communication is performed once, the order of every thread's completed regions is recorded and an execution plan is constructed.

3.3.3 Symmetry Related Optimisations

The input sparse matrix of the SpMM$^{\mathrm{T}}$V operation may be symmetric. The used sparse matrix hierarchical storage format can be adapted to take advantage of symmetry and store only half of the elements. When performing SpMM$^{\mathrm{T}}$V with a symmetric matrix, only half of the multiplications need to be performed. This is because the result of each multiplication can be added to two symmetrical locations of each output vector. Because SpMM$^{\mathrm{T}}$V is a memory bound operation, it substantially benefits from decreasing the amount of transferred data (Section 2.1).

For a matrix in the CSR format, performing SpMV and SpM$^{\mathrm{T}}$V exhibits different memory access patterns [16]. When performing SpMV with a matrix's row, multiple memory locations of the input vector are read and a single memory location of the output vector is overwritten. When performing SpM$^{\mathrm{T}}$V with the same matrix row, a single memory location of the input vector is read and multiple memory locations of the output vector are overwritten. Because writes are more expensive than reads, performing SpM$^{\mathrm{T}}$V with a matrix in the CSR format results in a worse memory access pattern than when performing SpMV with the same matrix [16]. But SpM$^{\mathrm{T}}$V can be performed in the same manner as SpMV when the input matrix is symmetric. Such an optimisation can be applied to an SpMM$^{\mathrm{T}}$V kernel for CSR matrices as well.

Another optimisation opportunity arises when an SpMM$^{\mathrm{T}}$V operation is performed with a symmetric matrix and two identical input vectors. In this case both resulting output vectors will be identical too. Thus the SpMM$^{\mathrm{T}}$V

operation is equivalent to performing an SpMV operation and copying the output vector.

Chapter **4**

Hardware and Software Aspects

This chapter covers miscellaneous hardware and software aspects of the implementation. Section 4.1 introduces OpenMP – an API used in parallel parts of the implementation. SIMD instructions, which are used to optimise certain parts of the implementation, are introduced in Section 4.2. Finally Section 4.3 introduces the Math Kernel Library.

4.1 OpenMP

The term *OpenMP* is commonly used to refer to the *OpenMP Application Programming Interface*. The implementation presented in Chapter 5 uses version 4.5 of the OpenMP API. The OpenMP API version 4.5 specification is available in [17].

As described in [17], the OpenMP specification consists of compiler directives, library routines and environment variables for developing parallel programs. The specification extends the C, C++ and Fortran programming languages with single program multiple data constructs, tasking constructs, synchronisation constructs, SIMD constructs, etc. A program that makes use of the OpenMP API is portable among all compilers which implement the OpenMP API specification.

One way to use the OpenMP API is through the provided compiler directives [17]. Compiler directives can be added to the program's source code to mark sections that should be executed in parallel. A compiler, which supports the OpenMP API specification, will replace the directives with code that executes the marked section in parallel.

4.2 SIMD

Modern Intel 64 processor families support single-instruction multiple-data operations [18], which are commonly referred to as SIMD instructions. Ex-

amples of SIMD instruction extensions are *Streaming SIMD Extensions* (SSE) or *Advanced Vector Extensions* (AVX). A SIMD instruction performs the same operation on multiple data elements simultaneously. The implementation presented in Chapter 5 makes use of SIMD instructions by using compiler directives from the OpenMP API specification (Section 4.1).

Instructions introduced in SSE, SSE2, SSE3, SSSE3 and SSE4 operate on 128-bit (i.e. 16-byte) XMM registers. An XMM register can hold either (a) two double-precision floating-point numbers, (b) four single-precision floating-point numbers, (c) two 64-bit integers, etc. While data can be loaded into an XMM register from any memory location, loading data from 16-byte aligned addresses is the most efficient. Data is 16-byte aligned when it is located at an address that is a multiple of 16 bytes. The same alignment restrictions apply to storing data from XMM registers to memory. [18]

Instructions introduced in AVX and AVX2 are analogous to SSE instructions but operate on 256-bit (i.e. 32-byte) registers. The most efficient AVX load and store instructions require 32-byte aligned data. [18]

4.3 Math Kernel Library

The Intel *Math Kernel Library* (MKL) as a collection of functions for software applications that solve large computational problems [19]. MKL provides BLAS and LAPACK linear algebra functions, functions for deep neural networks, vectorised math functions and so forth. Functions in MKL are highly optimised, take advantage of multi-core CPUs and utilise SIMD instructions. The library provides a C interface and a Fortran interface.

An SpMM$^{\mathrm{T}}$V implementation was developed that uses the Math Kernel Library. The implementation uses the Inspector-Executor Sparse BLAS functions [19]. This implementation is used to help assess the performance of SpMM$^{\mathrm{T}}$V implementations presented in Chapter 5.

CHAPTER 5

Implementation

The implementation has been primarily developed using the C++ programming language [20]. OpenMP, which introduced in Section 4.1, is used to implement multithreading and to implement the use of SIMD instructions. Many parts of the implementation use libraries from Boost [21].

The most time consuming part of this thesis was developing an implementation based on the ideas presented in Chapter 3. This chapter is divided into sections covering various parts of the implementation.

Section 5.1 names the implemented sparse matrix storage formats and outlines their implementations. All parallel $\text{SpMM}^\text{T}\text{V}$ implementations need to have their input matrix first converted to a two-level hierarchical storage format. The process of converting a matrix in the CSR format to a matrix in the implemented two-level hierarchical storage format is explained in Section 5.2. Computational kernels used by all parallel $\text{SpMM}^\text{T}\text{V}$ implementations are listed in Section 5.3. Section 5.4 describes all implemented parallel $\text{SpMM}^\text{T}\text{V}$ methods. Section 5.5 discusses how the implementation was adapted for usage in iterative algorithms. Extra optimisation opportunities arose during the implementation's development process and are covered in Section 5.6. Several additional utilities were developed apart from the main program, the most important ones are presented in Section 5.7.

5.1 Sparse Matrix Storage Formats

Of all the sparse matrix storage formats introduced in Chapter 1, the implementation makes use of the coordinate format, the compressed sparse row format and a two-level hierarchical format. The aim of Sections 5.1.1 and 5.1.2 is to outline key aspects of the formats' implementations, not to describe every single implementation detail. Because of this, only the most important data members are depicted. The implementation's full source code is available on the attached disc (Section B).

5.1.1 Coordinate and Compressed Sparse Row Formats

The Coordinate format was implemented as described in Section 1.2. A matrix's element values are stored as an array of single-precision floating-point numbers. Row indices and column indices are stored as two arrays of 32-bit unsigned integers. This limits the maximum dimensions of the stored matrix to $2^{32} \times 2^{32}$, but as was pointed out in Section 3.1, the benefits of this limitation should outweight the drawbacks.

The Compressed Sparse Row format was implemented as detailed in Section 1.3. A matrix's element values are stored as an array of single-precision floating-point numbers. Row pointers and column indices are stored as two arrays of 32-bit unsigned integers. Storing row pointers as 32-bit unsigned integers further constraints the stored matrix – it limits the maximum nonzero element count to 2^{32}. This limitation was deemed acceptable, because only one matrix from the *SuiteSparse Matrix Collection* [6] exceeds this limit at the time of writing.

5.1.2 Two-Level Hierarchical Storage Format

A two-level hierarchical storage format for sparse matrices (Section 1.5) was implemented as specified in Section 3.1. The format's first level uses the COO format and its second level uses the COO or CSR format. The first level stores 32-bit indices and the second level stores 16-bit indices.

All of a matrix's element values are stored as an array of single-precision floating-point numbers. Column indices are stored as an array of 16-bit unsigned integers. This array contains column indices within individual regions, not column indices in the original matrix. Finally the data structure contains an array of matrix regions.

A region contains a pointer to its element values and column indices stored in the aforementioned arrays. If it is a COO format region, it stores row indices as an array of 16-bit unsigned integers. Otherwise it is a CSR format region and stores row pointers as 32-bit unsigned integers. Row pointers use 32-bits because regions have up to 2^{16} rows and 2^{16} columns and may contain up to 2^{32} elements (Section 3.1). A region also contains a row index and a column index of its topmost, leftmost element in the original matrix. This element has row and column indices equal to $(0, 0)$ within the region. All stored row and column indices of all elements of a region are relative to this element.

Row and column indices of the above mentioned topmost, leftmost element are stored a 32-bit unsigned integers. It might seem that the indices could be stored as 16-bit unsigned integers because the maximum dimensions of a matrix are $2^{32} \times 2^{32}$ and regions start at multiples of 2^{16} rows and columns. A larger data type needs to be used because regions with too many elements are split into smaller regions.

5.2 Matrix Construction Process

Before SpMM$^{\mathrm{T}}$V can be performed with a matrix in the two-level hierarchical storage format (Section 5.1.2), it needs to be converted from its input format. Here the implementation expects the input matrix to be stored in the CSR format (Section 5.1.1).

The hierarchical matrix's construction process is based on the implementation presented in [3]. Like the implementation in [3], this implementation only supports square input matrices. Because the conversion process is fairly complicated, only a high-level overview of it is presented.

The conversion begins with an input matrix in the CSR format. Individual conversion steps are as follows:

1. Allocate the hierarchical matrix's column index and element number arrays according to the dimensions of the input matrix.

2. Determine the row and column indices of borders that partition the matrix into regions of size $2^{16} \times 2^{16}$.

3. Determine the nonzero element count of each of these regions.

4. Create regions of the appropriate types. A CSR format region is created if the region's nonzero element count is greater than its row count. A COO region is created otherwise. Empty regions, i.e. regions without any nonzero elements, are omitted.

5. Copy elements and their indices from the input matrix to the hierarchical matrix's regions. Element values are simply copied to their appropriate positions. Row and column element indices are additionally renumbered relative to the region's topmost, leftmost nonzero element. Both COO and CSR formatted region's elements are stored in row-major order.

6. Compute the region count threshold θ as $\alpha n_{nz}/\tau$, where $\alpha \in [0, 1]$ is a command-line parameter, n_{nz} is the count of nonzero elements and τ is the thread count used for parallel SpMM$^{\mathrm{T}}$V.

7. Iterate over all regions. If a region's element count exceeds threshold θ, split it into subregions. The new subregions replace the original regions and their elements.

8. Remove empty leading and trailing rows from CSR format regions.

Step 7 of the conversion process involves splitting a region into smaller subregions according to a threshold θ. This process comprises the following steps:

1. Determine the number of elements in every row and column of a region.

2. Split the region's rows into bands so that each of them consists of approximately θ elements. Perform the same with columns.

3. The borders of the created row and column bands can be viewed as a partitioning of the region into rectangular subregions. All resulting subregions will have approximately $\sqrt{n_{nz}/\theta}$ elements, where n_{nz} is equal to the nonzero element count of the region being split [3].

4. If the region begin split was not a CSR format region, determining the number of row elements in step 1 created a CSR format row pointer array. This means the region being split can be accessed as if it were a matrix in the CSR format. Steps 3 to 5 of the conversion process from the beginning of this section can therefore be reused to create the new subregions.

The hierarchical matrix resulting from the construction and region splitting processes above has regions sorted in row-major order. The implementation from [3] refers to this arrangement as *lexicographical order*. Sorting the regions according to the so-called Morton order or Z-order, as suggested in [3], was not implemented.

5.3 Computational Kernels

As described in Chapter 2, the input of an SpMM$^{\mathrm{T}}$V operation is a matrix and two vectors; the operation's output are two vectors. For simplicity, the implementation in this thesis restricts the input to square matrices and accepts a single vector (the implementation from [3] restricts the input in the same way). The two output vectors are preserved.

The input sparse matrix is first converted to a hierarchical storage format (Section 5.2). Results of the SpMM$^{\mathrm{T}}$V operation are obtained by calling an appropriate computational kernel on every region of the hierarchical matrix. All implemented parallel SpMM$^{\mathrm{T}}$V variants make use of the same pair of computational kernels, both of which are described in the remainder of this section.

The function in Listing 5.1 performs SpMM$^{\mathrm{T}}$V with a COO formatted region. It is based on the implementation from [3]. Parameter `row_indices` is the region's COO format row index array, `col_indices` is the region's COO format column index array and `values` is the region's COO format element value array. Parameter `element_count` is the region's nonzero element count. Parameter `vector` is the input vector array. Parameters `y1` and `y2` are output vector arrays. Parameters `row_offset` and `col_offset` specify the offsets of the region's indices in the original matrix. The kernel traverses all of the region's nonzero elements, computes their row and column indices in the original matrix, multiplies the elements' values with the appropriate input vector elements and adds the results to the correct output vector elements.

```
1  spmmtv_coo(row_indices[], col_indices[], values[],
2             element_count, vector[], y1[], y2[], row_offset,
3             col_offset)
4  {
5      for (i = 0; i < element_count; ++i) {
6          row_index = row_offset + row_indices[i];
7          col_index = col_offset + col_indices[i];
8          y1[row_index] += values[i] * vector[col_index];
9          y2[col_index] += values[i] * vector[row_index];
10     }
11 }
```

Listing 5.1: Pseudocode of a computational kernel for SpMM$^{\mathrm{T}}$V with a COO formatted region of a matrix stored in a hierarchical storage format

The function in Listing 5.2 performs SpMM$^{\mathrm{T}}$V with a CSR formatted region. It is based on the implementation from [3]. Parameter `row_pointers` is the region's CSR format row pointer array, `col_indices` is the region's CSR format column index array and `values` is the region's CSR format element value array. Parameter `row_count` is the region's row count. Parameter `vector` is the input vector array. Parameters `y1` and `y2` are output vector arrays. Parameters `row_offset` and `col_offset` specify the offsets of the region's indices in the original matrix. The kernel traverses the region's elements in a row-wise manner. The elements' row and column indices in the original matrix are computed as necessary and their values are multiplied with the correct input vector elements. While results are added to the appropriate elements of array `y2` immediately, the inner loop minimises writes to array `y1` by summing results into a local variable.

5.4 Parallelisation Methods

Two of the three parallel SpMM$^{\mathrm{T}}$V execution methods introduced in Section 3.2 have been implemented, namely *SpMM^{T}V without planning* and *dynamically planned SpMM^{T}V*. *Statically planned SpMM^{T}V* has not been implemented because it underperformed the other two approaches [3].

Section 3.3.2 introduces (1) an SpMM$^{\mathrm{T}}$V execution method that assigns disjoint sets of regions to individual threads and (2) a dynamically planned variant of this method. Both of these methods have been implemented. The methods are further referred to as *parallel disjoint SpMM^{T}V without planning* and *parallel dynamically planned disjoint SpMM^{T}V*.

All implemented parallel SpMM$^{\mathrm{T}}$V variants keep track of finished and unfinished regions using the approach presented in Section 3.3.1. Symmetry related optimalisations mentioned in Section 3.3.3 have not been implemented.

```
1  spmmtv_csr(row_pointers[], col_indices[], values[], row_count,
2             vector[], y1[], y2[], row_offset, col_offset)
3  {
4      for (region_row = 0; region_row < row_count; ++region_row) {
5          row_index = row_offset + region_row;
6          y1_sum = float(0);
7  
8          i_begin = row_pointers[region_row];
9          i_end = row_pointers[region_row + 1];
10 
11         for (i = i_begin; i < i_end; ++i) {
12             col_index = col_offset + col_indices[i];
13             y1_sum += values[i] * vector[col_index];
14             y2[col_index] += values[i] * vector[row_index];
15         }
16 
17         y1[row_index] += y1_sum;
18     }
19 }
```

Listing 5.2: Pseudocode of a computational kernel for SpMM^{T}V with a CSR formatted region of a matrix stored in a hierarchical storage format

5.4.1 Traversing Region Indices

All implemented parallel SpMM^{T}V variants need a way to keep track of available region indices. Variants *SpMM^{T}V without planning* and *disjoint SpMM^{T}V without planning* additionally keep track of finished and unfinished regions. The linked list-like data structure proposed in Section 3.3.1 has been implemented and is used by all implemented SpMM^{T}V variants.

The data structure is implemented as a circular, singly linked list. All of the list's elements are stored in an array to improve memory access locality as recommended in Section 3.3.1. Each element of the linked list contains a region index and a pointer to the list's next element. The data structure was designed to perform all operations used by SpMM^{T}V variants in constant time, e.g. returning the current region index, iterating to the next region index or removing the current region index from the list.

5.4.2 SpMM^{T}V Without Planning

Parallel SpMM^{T}V without planning as introduced in [3] is described in Section 3.2. The implementation additionally uses the circular linked list mentioned in Section 5.4.1 to keep track of available region indices.

Each thread starts with a list of all region indices. A thread continuously

iterates over all region indices in its list and attempts to lock the associated region. Once the thread manages to lock a region, it queries a shared array to verify that the region has not been completed yet. If the region has been completed, the thread releases the region's lock, removes the region's index from its list and continues to iterate over region indices. If the region has not been completed, the thread attempts to acquire the region's output locks. If the thread fails to do so, it releases the region's lock and resumes iteration. After the thread successfully acquires the region's output lock, it marks the region as done, releases the region's lock, performs SpMM$^{\mathrm{T}}$V with the region, releases the region's output locks, removes the current region index from its list and proceeds to iterate over region indices. A thread terminates once its list of region indices is empty.

Care has been taken to avoid the situation where all threads initially attempt to lock the same region. Lists assigned to individual threads are created so that the starting region indices of all threads are evenly spread out.

5.4.3 Disjoint SpMM$^{\mathbf{T}}$V Without Planning

Parallel disjoint SpMM^{T}V without planning performs less synchronisation than *parallel SpMM^{T}V without planning* (Section 5.4.2) but relies more on the "uniformity" of regions. As suggested in Section 3.3.2, every thread begins with a disjoint list of region indices. Specifically, a sequence of all region indices is created and contiguous, evenly sized subranges are assigned to each thread.

This variant works similarly to parallel SpMM$^{\mathrm{T}}$V without planning. Locking regions and verifying whether they have not been calculated by another thread is unnecessary. The algorithm constitutes of continuously iterating over region indices, attempting to acquire a region's output locks, performing SpMM$^{\mathrm{T}}$V with the region and removing the region's index from the threads list. Details of this approach are identical to the steps in Section 5.4.2.

5.4.4 Dynamic Planning

Section 3.2 outlines *dynamically planned SpMM^{T}V*, which was proposed in [3]. This variant begins with an initialisation step. SpMM$^{\mathrm{T}}$V without planning is performed and the order in which regions are finished is recorded. Every thread has an instance of a data structure for recording region indices, which is designed to minimise the overhead of appending a region index. The region's index is recorded after successfully performing SpMM$^{\mathrm{T}}$V with the region and releasing its output locks. All threads' recorded region indices are converted into linked lists described in Section 5.4.1.

After this initialisation step threads perform parallel SpMM$^{\mathrm{T}}$V with regions in the order given by their lists of region indices. Specifically a thread reads a region index, acquires the region's output locks, performs SpMM$^{\mathrm{T}}$V,

unlocks the output locks and switches to the next region index. Here locks are acquired in a blocking manner and the list of region indices is traversed only once.

As mentioned in Section 3.3.2, dynamic planning can be extended to parallel disjoint SpMM^{T}V (Section 5.4.3). The combined approach is called *parallel dynamically planned disjoint SpMM^{T}V*. Initialisation is implemented analogously: a single iteration of parallel disjoint SpMM^{T}V without planning is performed, region indices are recorded and threads' lists of region indices are created. Parallel SpMM^{T}V according to the order given by the created lists is performed exactly as described in the previous paragraph.

5.5 Iterability Considerations

The *biconjugate gradient method* (abbr. as BiCG) is mentioned in Section 2.1 as an iterative algorithm that can benefit from performing SpMM^{T}V. Every BiCG iteration performs SpMM^{T}V with the same matrix [1, 2]. This means if any data structures need to be reinitialised after a single SpMM^{T}V operation, it is desirable to do so as quickly as possible.

After all threads complete parallel disjoint SpMM^{T}V without planning (Section 5.4.3), it is necessary to set each thread's list of region indices (Section 5.4.1) to its original state. A shallow copy of every thread's list of region indices is created for this purpose before the first SpMM^{T}V iteration. The lists are therefore restored by copying the backed up data.

The same lists are restored in the same way after every iteration of parallel SpMM^{T}V without planning (Section 5.4.2). This execution method additionally needs to mark all of each thread's regions as unfinished after every SpMM^{T}V iteration. Because regions' completion status is indicated by the elements of a boolean array, resetting the completion status is a matter of overwriting the array with the appropriate value.

Both implemented dynamically planned methods from Section 5.4.4, i.e. parallel dynamically planned SpMM^{T}V and parallel dynamically planned disjoint SpMM^{T}V, do not require reinitialising any data after an SpMM^{T}V iteration. The planned methods' initialisation steps contain reinitialisation code because their implementations are based on methods without planning. But the initialisation steps should be considered as parts of preprocessing instead of iteration.

5.6 Implementation Specific Optimisations

Optimisations outlined in Section 3.3 can be applied to SpMM^{T}V implementations that are based on the ideas presented in [3]. Symmetry related optimisations in Section 3.3.3 should be applicable to any SpMM^{T}V approach.

This section describes optimisations which are specific to the implementation presented in Chapter 5.

Section 5.5 mentions that several SpMM$^{\mathrm{T}}$V methods need to reset each thread's list of region indices after every SpMM$^{\mathrm{T}}$V iteration. The lists are restored by copying their contents from a backup array. Because each thread has its own list, all threads can restore their lists in parallel. This is implemented by having every thread restore its list after it finds out the list is empty but before the thread terminates itself. Copying a list's underlying element array (Section 5.4.1) can be potentially sped up by making use of SIMD instructions (Section 4.2). While the usage of SIMD instructions was not implemented because of minor complications caused by the structure of list elements, it is a viable optimisation nonetheless.

Section 5.5 further mentions that every iteration of parallel SpMM$^{\mathrm{T}}$V without planning (Section 5.4.2) needs to mark all of each thread's regions as unfinished after every SpMM$^{\mathrm{T}}$V iteration. The relevant array can be overwritten only after all threads finish accessing it. In order to overwrite the array in parallel, extra thread synchronisation needs to be introduced. Such overhead may outweigh gains from parallelisation as the overwritten array is not very large and consists of boolean values. Overwriting the array is instead implemented as a single-threaded operation which uses SIMD instructions. As was mentioned in Section 4.2, SIMD instructions can only be used to their full potential on appropriately aligned data. The array, which is used to indicate the completion status of regions, is therefore aligned to enable efficient usage of up to 256-bit wide SIMD instructions. SIMD instructions are employed using the OpenMP SIMD construct [17].

5.7 Additional Utilities

Several accompanying utilities have been created in the course of developing the implementation presented in Chapter 5. This section describes the most important utilities. Most of the mentioned utilities have been implemented in the Python programming language [22].

The main program loads the input matrix from a file. Older versions of the implementation loaded the matrix from the text-based *Matrix Market* format (abbr. as MM). The MM format is introduced in Section 1.6. But loading large sparse matrices from the MM format is very slow. This turned out to be a significant hurdle when loading matrices many times in quick succession. For this reason a Python utility called `mmtobin` has been developed. It loads a matrix in the MM format and stores it as a binary file. The output binary file either contains a matrix in the COO format or in the CSR format. The files contain COO or CSR format arrays as described in Section 1.2 or 1.3 respectively. Therefore the main program can load a binary matrix by simply copying contiguous blocks of memory.

Developing and improving various SpMM^{T}V implementations resulted in the need to verify their correctness. A Python utility called `pyspmmtv` has been developed, it accepts paths to an input matrix and input vector, performs SpMM^{T}V using existing libraries and outputs the resulting pair of vectors to files. The input vector for this utility was randomly generated using a separate `random-vector` Python utility.

A test program has been developed in C++ to verify the integrity of loaded matrices and verify the correctness of SpMM^{T}V implementations. SpMM^{T}V correctness is checked by loading a matrix and its generated vector, performing spmmtv and comparing the results with the vectors created by `pyspmmtv`. The loaded matrices were created by `mmtobin` and their matching vectors were generated by `random-vector`. All SpMM^{T}V implementations developed in this thesis are tested in the aforementioned way. The test program makes extensive use of the Boost.Test library [21].

Chapter 6

Performance Evaluation

This chapter evaluates the performance of SpMM$^{\mathrm{T}}$V implementations presented in Chapter 5. Section 6.1 describes the test environment's hardware and software specifications. Section 6.2 explains the choice of input sparse matrices and lists their properties. Section 6.3 explains units used to present the results in Section 6.4. Section 6.4 contains the results of performed measurements and compares them with an existing library.

6.1 Test Environment

All performance benchmarks were performed on a node of the faculty's computer cluster called STAR. The used node consists of two six-core Intel Xeon E5-2620 v2 processors [23], i.e. it has 12 physical CPU cores in total. The node has approximately 32 GB of random-access memory.

The node runs a 64-bit GNU/Linux operating system. The implementation presented in this thesis (Chapter 5) was compiled using the C++ compiler from the GNU Compiler Collection version 8.2.1 [24]. Options passed to the compiler that affect the performance of the resulting program were:

- `-march=native`,
- `-mtune=native`,
- `-O3`,
- `-DNDEBUG`,
- `-flto` and
- `-fno-fat-lto-objects`.

The implemented program uses the OpenMP API [17] and depends on several Boost libraries [21]. All benchmarks were performed with OpenMP version 4.5 and Boost version 1.67.0. OpenMP is introduced in Section 4.1.

The implementation was also compared with functions from the Intel Math Kernel Library [19]. All benchmarks were performed with Math Kernel Library version 2017.0.0 which is a part of Intel compilers and libraries version 2017.0.098. The Math Kernel Library is introduced in Section 4.3. The library was configured to use 32-bit integers.

6.2 Input Sparse Matrices

The main input parameter of SpMM$^{\mathrm{T}}$V implementations is a sparse matrix. An implementation can be tested with randomly generated sparse matrices or with matrices from real-world applications. Both matrix types can be used to verify the correctness of the implementation, i.e. whether the calculated results are accurate. But measuring an implementation's performance with randomly generated matrices may not accurately reflect performance in real-world scenarios with domain-specific sparse matrices [6]. This is why all benchmarks have been performed with input matrices from the *SuiteSparse Matrix Collection* [6].

The selection of input sparse matrices is based (1) on the matrices used in [3] and (2) on the sample set of test matrices in [11]. Because all used test matrices are provided on the attached disc (Section B), the choice of test matrices was also shaped by the disc's capacity. Ten sparse matrices have been picked from the SuiteSparse Matrix Collection, eight of which have been used for assessing the implementations' performance. Properties of these eight matrices are listed in Table 6.1. The SuiteSparse Matrix Collection also includes the problem category or domain from which its matrices have arisen. This information is provided in Table 6.2. The two unlisted matrices, which have very few nonzero elements, were not benchmarked and were merely used for debugging purposes.

6.3 Measurement Approach

All performance results presented in Section 6.4 are based on measuring the run-time of a single iteration of SpMM$^{\mathrm{T}}$V. Any overhead in the form of preprocessing is omitted. All presented durations in Section 6.4 have been obtained as an arithmetic mean of five results.

The run-time of SpMM$^{\mathrm{T}}$V with matrices of various sizes and sparsity patterns can significantly differ. A unit called *operations per second* (OPS) is used in Section 6.4 to make comparing the run-time of different matrices easier. OPS is used as a metric of performance. If an iteration of SpMM$^{\mathrm{T}}$V with a matrix that has n_{nz} nonzero elements took d seconds, then performance

Matrix Name	n	n_{nz}	n_{nz}/n	$100 \cdot n_{nz}/n^2$
af_shell10	1,508,065	52,672,325	34.93	$2.32 \cdot 10^{-3}$
atmosmodm	1,489,752	10,319,760	6.93	$4.65 \cdot 10^{-4}$
cage14	1,505,785	27,130,349	18.02	$1.20 \cdot 10^{-3}$
FullChip	2,987,012	26,621,990	8.91	$2.98 \cdot 10^{-4}$
rajat31	4,690,002	20,316,253	4.33	$9.24 \cdot 10^{-5}$
RM07R	381,689	37,464,962	98.16	$2.57 \cdot 10^{-2}$
thermal2	1,228,045	8,580,313	6.99	$5.69 \cdot 10^{-4}$
thread	29,736	4,470,048	150.32	$5.06 \cdot 10^{-1}$

Table 6.1: Properties of matrices from the SuiteSparse Matrix Collection [6] that were used for benchmarking purposes; n is the matrix's row count and column count (i.e. all matrices are square), n_{nz} is the matrix's nonzero element count, n_{nz}/n can be interpreted as the matrix's average count of nonzero elements per row and $100 \cdot n_{nz}/n^2$ as what percentage of the matrix's elements is nonzero

Matrix Name	Matrix Category
af_shell10	structural problem
atmosmodm	computational fluid dynamics problem
cage14	directed weighted graph
FullChip	circuit simulation problem
rajat31	circuit simulation problem
RM07R	computational fluid dynamics problem
thermal2	thermal problem
thread	structural problem

Table 6.2: Problem categories or domains from which the matrices used for benchmarking have arisen as listed in the SuiteSparse Matrix Collection [6]

in OPS is equal to n_{nz}/d. This unit is used purely to facilitate the interpretation of the measured results, not to reflect the actual number of performed operations. OPS may be prefixed with M to indicate *millions of operations per second* (MOPS).

Section 6.4 also refers to a unit called *parallel speedup*. Parallel speedup is calculated from sequential run-time d_s and parallel run-time d_p as d_s/d_p.

6.4 Results

Chapter 5 introduces the developed parallel SpMM$^{\mathrm{T}}$V implementations, i.e. parallel SpMM$^{\mathrm{T}}$V without planning, parallel dynamically planned SpMM$^{\mathrm{T}}$V, parallel disjoint SpMM$^{\mathrm{T}}$V without planning and parallel dynamically planned

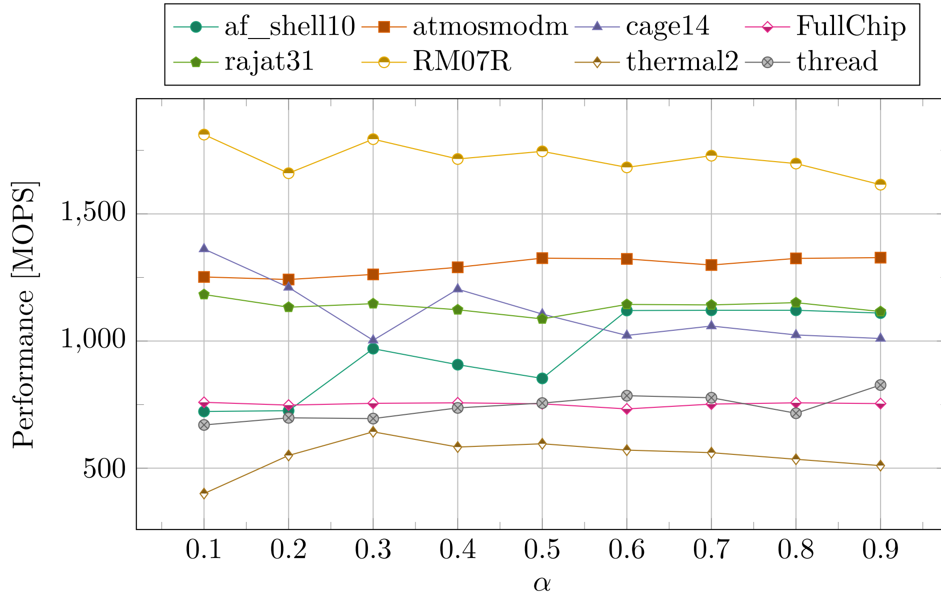

Figure 6.1: Performance of parallel disjoint SpMM$^{\mathrm{T}}$V without planning for different values of α using 12 threads

disjoint SpMM$^{\mathrm{T}}$V. The input matrix has to be first converted into a two-level hierarchical storage format (Section 5.2). This process is affected by the coefficient $\alpha \in [0, 1]$, which is passed to the implemented program as a command line argument. It has been determined in [3] that the optimal value for α is 0.1 to 0.15. Because the implementation in Chapter 5 differs from the one in [3] and because the disjoint parallel SpMM$^{\mathrm{T}}$V execution methods are not covered in [3], tests have been performed to determine the optimal value of α.

Figure 6.1 shows the performance of parallel disjoint SpMM$^{\mathrm{T}}$V without planning for different values of α, when using 12 threads. There does not seem to be an immediately obvious optimal value for α. Some matrices cause the performance of SpMM$^{\mathrm{T}}$V to be sensitive to changes to α. Other matrices make the performance of SpMM$^{\mathrm{T}}$V insensitive to changes to α. Matrices *af_shell10* and *cage14* cause the performance of SpMM$^{\mathrm{T}}$V to be affected by changes to α in opposite ways.

Performance of parallel SpMM$^{\mathrm{T}}$V without planning using 12 threads is affected very little by changes to the value of α, as can it be seen in Figure 6.2. Just like with disjoint SpMM$^{\mathrm{T}}$V, the optimal choice for α does not seem to be immediately obvious.

Further benchmarks of parallel SpMM$^{\mathrm{T}}$V execution methods had the value of α set as follows:

1. Benchmarks of parallel disjoint SpMM$^{\mathrm{T}}$V without planning and parallel

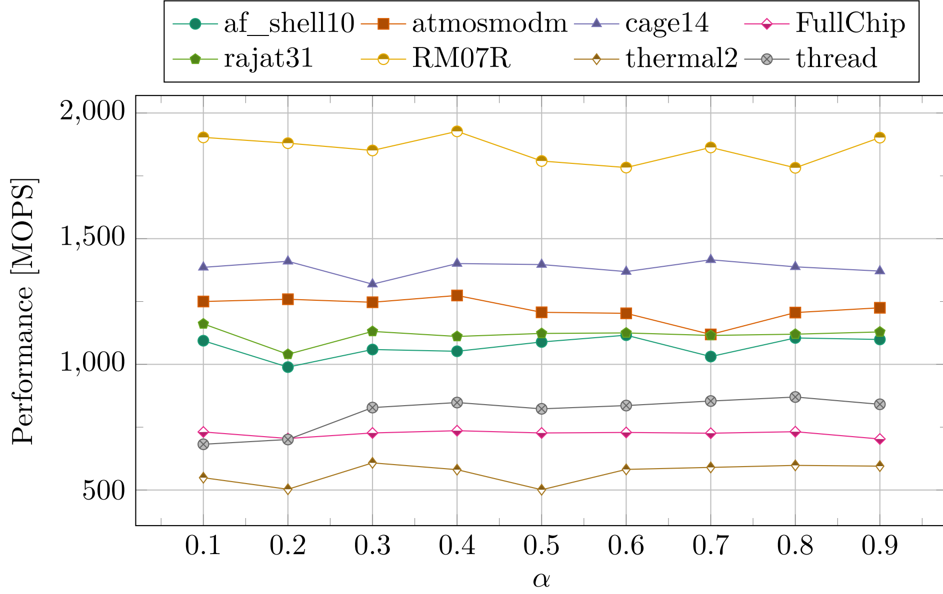

Figure 6.2: Performance of parallel SpMM$^{\mathrm{T}}$V without planning for different values of α using 12 threads

dynamically planned disjoint SpMM$^{\mathrm{T}}$V had α set to 0.6.

2. Benchmarks of parallel SpMM$^{\mathrm{T}}$V without planning and parallel dynamically planned SpMM$^{\mathrm{T}}$V had α set to 0.4.

Dynamically planned SpMM$^{\mathrm{T}}$V execution methods are largely based on their unplanned variants (Section 5.4.4). This is why dynamically planned execution methods reuse the values of α from unplanned methods.

Let m be equal to the arithmetic mean of parallel speedups of all matrices in Table 6.1 for a given execution method and a given thread count. The value of m is equal to the average parallel speedup of the given execution method and thread count. Figure 6.3 shows the average parallel speedups of implemented parallel SpMM$^{\mathrm{T}}$V execution methods for various thread counts. Parallel speedup is calculated with respect to a sequential SpMM$^{\mathrm{T}}$V execution method which uses the same two-level hierarchical storage format as parallel execution methods but performs all computations in a single thread. This sequential SpMM$^{\mathrm{T}}$V execution method is further referred to as *sequential hierarchical SpMM^{T}V*.

Average parallel speedups of SpMM$^{\mathrm{T}}$V implementations without planning were less than the average parallel speedups of their dynamically planned variants for all thread counts (Figure 6.3). Disjoint SpMM$^{\mathrm{T}}$V implementations did not turn out to be faster than their nondisjoint counterparts. It seems that

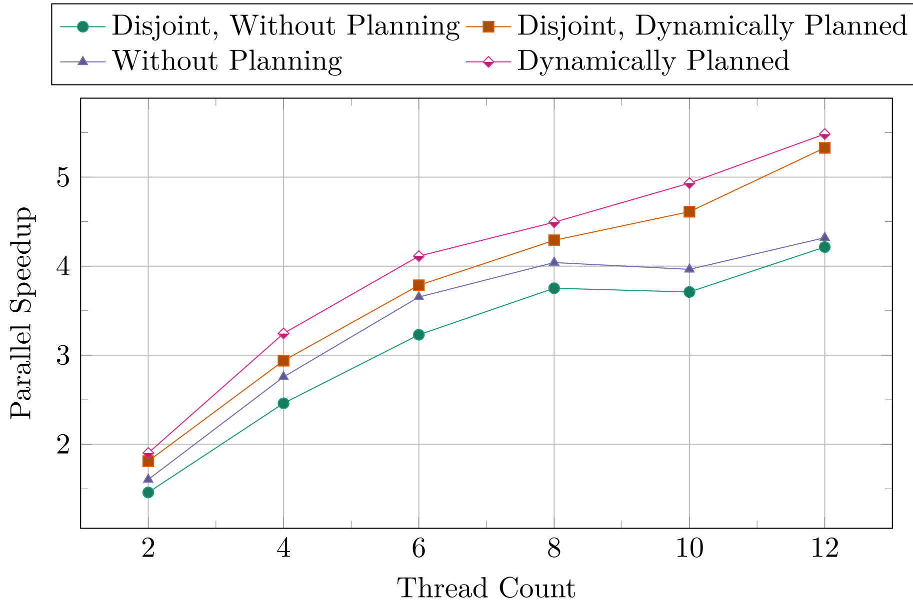

Figure 6.3: Average parallel speedups of SpMM^{T}V implementations for various thread counts

the benefits from reducing thread synchronisation overhead were outweighed by the disbalanced distribution of work among threads.

Parallel speedups of dynamically planned SpMM^{T}V implementations for various thread counts are shown in Figures 6.4 and 6.5. Instead of average parallel speedups, the figures show parallel speedups with all matrices individually. Speedups with different matrices vary significantly. The nonzero elements of matrices *af_shell10* and *atmosmodm* consist of a single narrow band along the matrix's diagonal. SpMM^{T}V with these matrices benefits the most from parallelisation. Conversely, matrix *FullChip* has nonzero elements spread out over its entire extent. Not only did increasing the thread count improve the speedup of SpMM^{T}V with this matrix very little, but the highest thread counts even decreased speedup. It can be assumed that the two-level hierarchical representation of *FullChip* contains many regions with overlapping output ranges which limits the possibility of performing SpMM^{T}V with several regions in parallel.

An SpMM^{T}V implementation was developed that uses the Intel Math Kernel Library (Section 4.3). This implementation is further referred to as *Math Kernel Library SpMM^{T}V* or *MKL SpMM^{T}V*. This implementation's performance was compared with the implementations presented in Chapter 5.

Figure 6.6 compares single-thread performance of MKL SpMM^{T}V and of sequential hierarchical SpMM^{T}V which uses the two-level hierarchical storage format from Section 5.1.2. MKL SpMM^{T}V outperforms sequential hierarch-

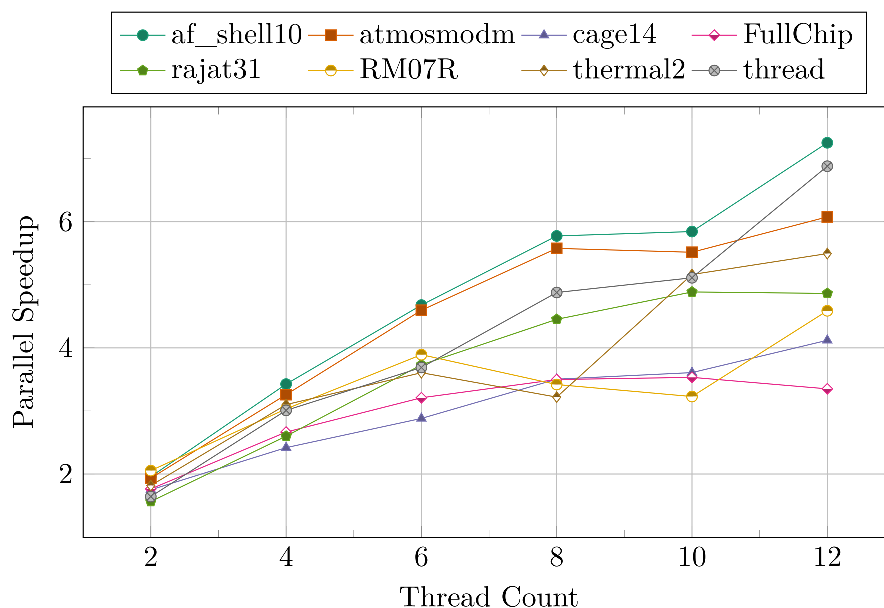

Figure 6.4: Parallel speedup of dynamically planned disjoint SpMM$^{\mathrm{T}}$V for various thread counts with all matrices

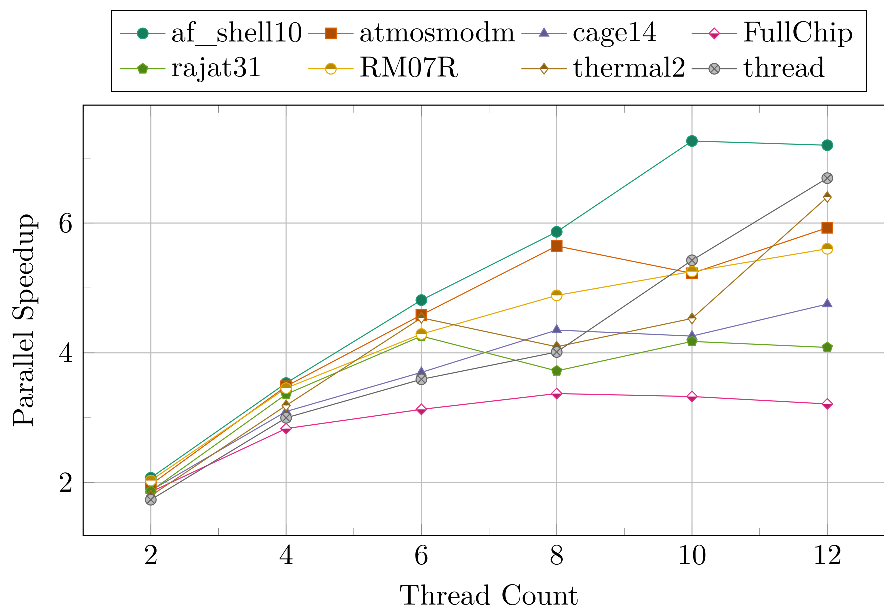

Figure 6.5: Parallel speedup of dynamically planned SpMM$^{\mathrm{T}}$V for various thread counts with all matrices

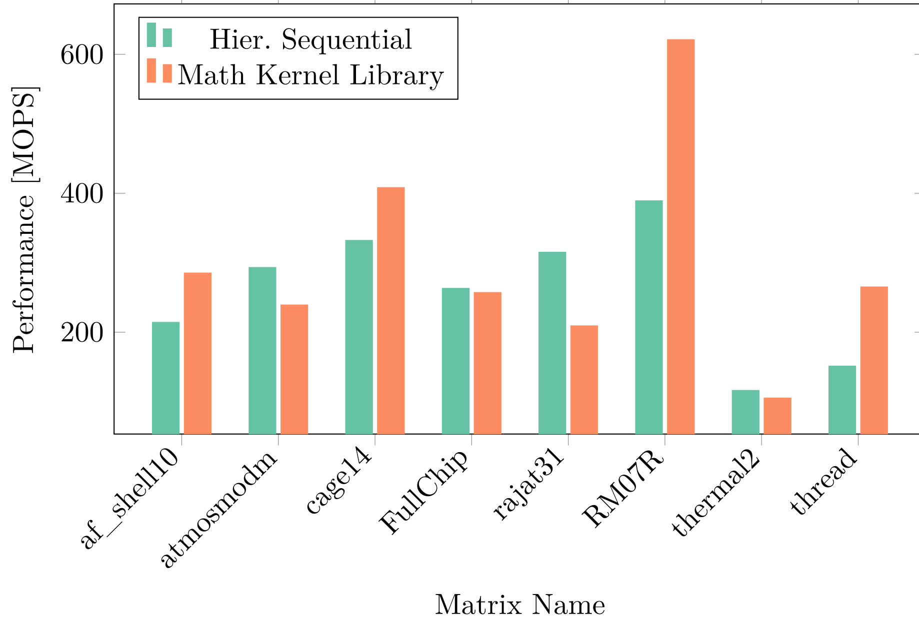

Figure 6.6: Sequential performance comparison of hierarchical SpMM$^{\mathrm{T}}$V and Math Kernel Library SpMM$^{\mathrm{T}}$V with all matrices

ical SpMM$^{\mathrm{T}}$V with the four least sparse matrices. The performance of sequential hierarchical SpMM$^{\mathrm{T}}$V matches or outperforms MKL SpMM$^{\mathrm{T}}$V with the remaining sparser matrices.

The average performance of both dynamically planned SpMM$^{\mathrm{T}}$V implementations and of MKL SpMM$^{\mathrm{T}}$V is compared in Figure 6.7 for various thread counts. Average performance is calculated analogously to average parallel speedups in Figure 6.3, i.e. average performance is an arithmetic mean of performance with all matrices. MKL SpMM$^{\mathrm{T}}$V outperforms dynamically planned disjoint SpMM$^{\mathrm{T}}$V for all thread counts but 12. The performance of dynamically planned SpMM$^{\mathrm{T}}$V is similar to performance of MKL SpMM$^{\mathrm{T}}$V. MKL SpMM$^{\mathrm{T}}$V is faster for thread counts four and six whereas dynamically planned SpMM$^{\mathrm{T}}$V is more performant for thread count equal to 12. These results suggest that both dynamically planned SpMM$^{\mathrm{T}}$V implementations scale better to higher thread counts than Math Kernel Library SpMM$^{\mathrm{T}}$V.

Table 6.3 lists durations of sequential hierarchical SpMM$^{\mathrm{T}}$V, parallel Math Kernel Library SpMM$^{\mathrm{T}}$V, parallel dynamically planned disjoint SpMM$^{\mathrm{T}}$V and parallel dynamically planned SpMM$^{\mathrm{T}}$V. Durations of parallel SpMM$^{\mathrm{T}}$V implementations are for 12 threads. Durations of parallel dynamically planned SpMM$^{\mathrm{T}}$V mostly correspond to durations in [3]. This means that optimising the traversal of region indices as described in Section 5.4.1 did not result in a significant speedup. Durations of parallel MKL SpMM$^{\mathrm{T}}$V surprisingly differ

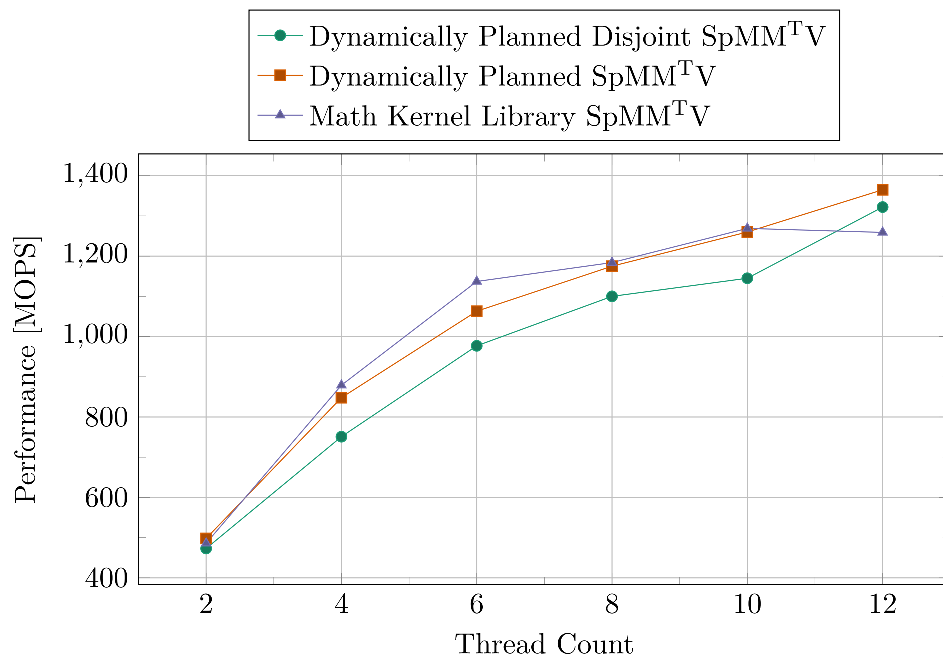

Figure 6.7: Average performance of parallel SpMM^{T}V implementations for various thread counts

quite significantly from the durations in [3]. This may be because the MKL SpMM^{T}V implementation in this thesis uses the Inspector-Executor Sparse BLAS functions [19] which the implementation in [3] might not have used.

Matrix	Sequential	Parallel MKL	Parallel Disjoint	Parallel
af_shell10	126.70	21.62	17.47	17.60
atmosmodm	35.17	7.92	5.79	5.93
cage14	81.82	16.23	19.86	17.22
FullChip	101.09	31.39	30.17	31.47
rajat31	64.46	18.00	13.25	15.78
RM07R	96.28	15.31	20.98	17.18
thermal2	42.38	8.52	7.71	6.62
thread	14.87	2.66	2.16	2.22

Table 6.3: Durations of sequential hierarchical $\text{SpMM}^\text{T}\text{V}$ (column *Sequential*), parallel Math Kernel Library $\text{SpMM}^\text{T}\text{V}$ (column *Parallel MKL*), parallel dynamically planned disjoint $\text{SpMM}^\text{T}\text{V}$ (column *Parallel Disjoint*) and parallel dynamically planned $\text{SpMM}^\text{T}\text{V}$ (column *Parallel*) in milliseconds; durations of parallel $\text{SpMM}^\text{T}\text{V}$ are for 12 threads

Chapter 7

Conclusion

This thesis investigates performing joint direct and transposed sparse matrix-vector multiplication (SpMM^{T}V) on shared memory systems. All relevant sparse matrix storage formats have been reviewed. The reviewed sparse matrix storage formats include (1) basic formats like the Coordinate format or the Compressed Sparse Row format, (2) hierarchical storage formats, which are key to implementing the SpMM^{T}V approach in [3] and (3) other storage formats, which are not as important for the implemented algorithm.

While performing SpMM^{T}V on CPUs and GPUs on shared memory systems seems to be largely unexplored, the most closely related CPU and GPU approaches have been described. A many-core implementation for Intel Xeon Phi processors was examined too.

The approach introduced in [3] is outlined and three new optimisations are proposed. Two of these optimisations are implemented. The implementation developed as a part of this thesis is explained in detail and further implementation specific optimisations are suggested.

The performance of four SpMM^{T}V implementations developed as a part of this thesis is evaluated. The performance of an SpMM^{T}V implementation based on the Intel Math Kernel Library is also evaluated. Performance differences between all versions are throughly examined and conclusions are made.

Future directions include: implementing remaining proposed optimisations, comparing the performance of SpMM^{T}V implementations with more existing libraries and investigating the possibility of a GPU implementation. Furthermore, the algorithm which converts the input matrix into the internal format before performing SpMM^{T}V should be simplified and its performance evaluated. This will facilitate assessing the tradeoff between time spent converting the input matrix and time saved by the implemented SpMM^{T}V operation.

Bibliography

[1] Y Saad. *Iterative Methods for Sparse Linear Systems*. 2nd ed. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, 2003. 520 pp. ISBN: 978-0-89871-800-3. DOI: 10.1137/1.9780898718003.

[2] R Barrett, M Berry, TF Chan, J Demmel, JM Donato, J Dongarra, V Eijkhout, R Pozo, C Romine and H van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. 2nd ed. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, 1994. 118 pp. ISBN: 978-1-61197-153-8. DOI: 10.1137/1.9781611971538.

[3] I Šimeček, D Langr and I Kotenkov. 'Multilayer Approach for Joint Direct and Transposed Sparse Matrix Vector Multiplication for Multithreaded CPUs'. In: *Parallel Processing and Applied Mathematics. Revised Selected Papers, Part I.* 12th International Conference, PPAM 2017 (Lublin, Poland, 10th–13th Sept. 2017). Ed. by R Wyrzykowski, J Dongarra, E Deelman and K Karczewski. Lecture Notes in Computer Science 10777. Cham: Springer International Publishing, 23rd Mar. 2018, pp. 47–56. ISBN: 978-3-319-78024-5. DOI: 10.1007/978-3-319-78024-5_5.

[4] H Anton and C Rorres. *Elementary Linear Algebra. Applications Version.* 10th ed. John Wiley & Sons, Inc., 12th Apr. 2010. ISBN: 978-0-470-43205-1.

[5] JH Wilkinson and C Reinsch. *Linear Algebra*. Ed. by FL Bauer. Handbook for Automatic Computation 2. Berlin, Heidelberg: Springer, 1971. 441 pp. ISBN: 978-3-662-39778-7. DOI: 10.1007/978-3-662-39778-7.

[6] TA Davis and Y Hu. 'The University of Florida Sparse Matrix Collection'. In: *ACM Trans. Math. Softw.* 38.1 (Nov. 2011). ISSN: 0098-3500. DOI: 10.1145/2049662.2049663. (Visited on 31/07/2018).

[7] I Šimeček, D Langr and P Tvrdík. ‘Space-efficient sparse matrix storage formats for massively parallel systems’. In: *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems* (Liverpool, UK, 25th–27th June 2012). IEEE, June 2012, pp. 54–60. ISBN: 978-1-4673-2164-8. DOI: `10.1109/HPCC.2012.18`.

[8] D Langr, I Šimeček, P Tvrdík, T Dytrych and JP Draayer. ‘Adaptive-Blocking Hierarchical Storage Format for Sparse Matrices’. In: *2012 Federated Conference on Computer Science and Information Systems (FedCSIS)* (Wrocław, Poland, 9th–12th Sept. 2012). IEEE, Sept. 2012, pp. 545–551. ISBN: 978-83-60810-48-4. IEEE: 6354485.

[9] I Šimeček and D Langr. ‘Space and execution efficient formats for modern processor architectures’. In: *2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)* (Timisoara, Romania, 21st–24th Sept. 2015). IEEE, Sept. 2015, pp. 98–105. ISBN: 978-1-5090-0461-4. DOI: `10.1109/SYNASC.2015.24`.

[10] RF Boisvert, R Pozo and KA Remington. *The Matrix Market Exchange Formats: Initial Design*. NIST Interagency or Internal Reports NISTIR 5935. Gaithersburg, MD: National Institute of Standards and Technology, Dec. 1996. 14 pp.

[11] D Langr and P Tvrdík. ‘Evaluation Criteria for Sparse Matrix Storage Formats’. In: *IEEE Transactions on Parallel and Distributed Systems* 27.2 (1st Feb. 2016), pp. 428–440. ISSN: 1045-9219. DOI: `10.1109/TPDS.2015.2401575`.

[12] K Asanović, R Bodik, BC Catanzaro, JJ Gebis, P Husbands, K Keutzer, DA Patterson, WL Plishker, J Shalf, SW Williams and KA Yelick. *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical Report UCB/EECS-2006-183. EECS Department, University of California, Berkeley, 18th Dec. 2006. 54 pp. URL: `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html` (visited on 25/04/2019).

[13] HM Aktulga, A Buluç, S Williams and C Yang. ‘Optimizing Sparse Matrix-Multiple Vectors Multiplication for Nuclear Configuration Interaction Calculations’. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium* (Phoenix, AZ, USA, 19th–23rd May 2014). IEEE, May 2014, pp. 1213–1222. ISBN: 978-1-4799-3800-1. DOI: `10.1109/IPDPS.2014.125`.

[14] Y Tao, Y Deng, S Mu, Z Zhang, M Zhu, L Xiao and L Ruan. ‘GPU accelerated sparse matrix-vector multiplication and sparse matrix-transpose vector multiplication’. In: *Concurrency and Computation: Practice and Experience* 27.14 (15th Sept. 2015), pp. 3771–3789. DOI: `10.1002/cpe.3415`.

[15] MO Karsavuran, K Akbudak and C Aykanat. 'Locality-Aware Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication on Many-Core Processors'. In: *IEEE Transactions on Parallel and Distributed Systems* 27.6 (8th June 2016), pp. 1713–1726. ISSN: 1045-9219. DOI: 10.1109/TPDS.2015.2453970.

[16] M Martone. 'Efficient multithreaded untransposed, transposed or symmetric sparse matrix/vector multiplication with the Recursive Sparse Blocks format'. In: *Parallel Computing* 40.7 (July 2014): *7th Workshop on Parallel Matrix Algorithms and Applications*. Ed. by C Bekas, A Grama, O Schenk and Y Saad, pp. 251–270. ISSN: 0167-8191. DOI: 10.1016/j.parco.2014.03.008. URL: https://www.sciencedirect.com/science/article/pii/S0167819114000386 (visited on 07/04/2019).

[17] OpenMP Architecture Review Board, ed. *OpenMP Application Programming Interface*. Version 4.5. Nov. 2015. 359 pp. URL: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf (visited on 05/05/2019).

[18] Intel Corporation, ed. *Intel 64 and IA-32 Architectures Software Developer's Manual. Basic Architecture*. 1. Version 253665-069US. Jan. 2019. URL: https://software.intel.com/en-us/articles/intel-sdm (visited on 05/05/2019).

[19] Intel Corporation, ed. *Intel Math Kernel Library Reference Manual. C*. Version 023. 6th Mar. 2019. URL: https://software.intel.com/en-us/mkl-developer-reference-c (visited on 05/05/2019).

[20] The Standard C++ Foundation, ed. *Standard C++*. 2019. URL: https://isocpp.org/ (visited on 04/05/2019).

[21] Boost contributors, ed. *Boost C++ Libraries*. 2019. URL: https://www.boost.org/ (visited on 04/05/2019).

[22] The Python Software Foundation, ed. *Python*. 2019. URL: https://www.python.org/ (visited on 04/05/2019).

[23] Intel Corporation, ed. *Intel Xeon Processor E5-2620 v2*. 2019. URL: https://ark.intel.com/products/75789/Intel-Xeon-Processor-E5-2620-v2-15M-Cache-2_10-GHz (visited on 08/05/2019).

[24] GCC Team, ed. *GCC, the GNU Compiler Collection*. Free Software Foundation, Inc. 20th Apr. 2017. URL: https://gcc.gnu.org/ (visited on 08/05/2019).

Appendix A

Building and Running

This chapter describes building and running the implementation presented in Chapter 5. Build dependencies are listed in Section A.1. How to build the main and test executables is detailed in Section A.2. Section A.3 explains how to perform SpMM^{T}V operations with the main executable. Section A.4 explains how to run tests with the test executable. Section A.5 describes additional utilities included on the attached disc.

Sections A.2, A.3 and A.4 include shell commands which operate on the data included on the attached disc (Chapter B). The recommended approach is to copy directory `thesis-data` from the attached disc to a writable location and change in the copied directory.

A.1 Build Dependencies

The following dependencies need to be satisfied to build the implementation presented in Chapter 5:

- CMake version 3.13 or newer.
- A version of the GNU Compiler Collection [24] with C++17 support. GNU GCC supports C++17 since version 8. While any compiler with C++17 support should work (as long as CMake detects it), the author has only tested GNU GCC.
- Installing an appropriate version of OpenMP should not be necessary as it should be provided by GNU GCC.
- A *recent* version of Boost C++ libraries [21]. The oldest version of Boost the author has verified to work is 1.67.0. The newest verified version is version 1.69.0. While other versions of Boost may work too, the author has not verified it.

- The Intel Math Kernel Library is an optional dependency. Building the implementation without MKL is possible but MKL related functionality will obviously be missing. The oldest version of MKL the author has verified to work is version 2017.0.0 from Intel compilers and libraries version 2017.0.098. The newest version the author has verified to work is MKL version 2019.0.3 from Intel compilers and libraries version 2019.3.199. Like with Boost, other versions are untested and may or may not work.

The author has only tested the build process on GNU/Linux. It should be possible to install all dependencies, possibly with the exception of MKL, using a GNU/Linux distribution's package manager. Furthermore a binary distribution of the latest version of CMake for GNU/Linux can be downloaded from the CMake website[1]. The provided archive contains a prebuilt binary which can be used immediately.

A.2 Building Executables

This section explains the steps needed to build the implementation's main and test executables. The steps assume that all dependencies listed in Section A.1 have been obtained, with the possible exception of Math Kernel Library.

1. Change to the directory that contains the implementation's source code.

   ```
   $ cd implementation
   ```

2. Create a directory for the output of the build process.

   ```
   $ mkdir -p build/release
   ```

   It may seem that the above command unnecessarily creates a nested directory. This directory structure is required if it is desired to run the implementation's tests because they contain hardcoded relative paths.

3. If Boost was installed to a nonstandard location, CMake may fail to find it. The path to Boost can be specified using an environment variable.

   ```
   $ export BOOST_ROOT=/opt/share/boost
   ```

4. Building the program with Math Kernel Library (MKL) is optional. The path to MKL needs to be specified as the library is often installed to a nonstandard location. On systems with MKL installed, environment variable `MKLROOT` may already contain the appropriate path. If unset, the environment variable can be set manually.

[1] https://cmake.org/download/

```
$ export MKLROOT=/opt/intel/mkl
```

5. The build process may be configured to optimise the created executable for the current CPU architecture if desired.

```
$ export CXXFLAGS="-march=native -mtune=native"
```

6. Change to the build directory and run CMake.

```
$ cd build/release
$ cmake -D CMAKE_BUILD_TYPE=Release ../..
```

7. Finally the executables can be built. The main executable is called `spmmtv`. The test executable is called `tests`. Execute the appropriate command depending on which executable is desired:

   - Executing `make` or `make spmmtv` builds the main executable.
   - Executing `make tests` builds the test executable.
   - Executing `make spmmtv tests` builds both executables.

   The resulting executable files are placed in the current build directory. The built main executable is available as `src/spmmtv`. The test executable is available as `tests/tests`.

A.3 Running the Main Executable

This section provides information about running the main executable of the implementation presented in Chapter 5. Instructions on how to build the executable are in Section A.2.

The executable is available in the build directory as described in Section A.2.

```
$ cd implementation/build/release
$ ./src/spmmtv -h
```

Executing the above commands prints the main executable's usage information.

The main executable expects a mode to be specified in option `-m` or `--mode`. Some of the available modes have not been introduced in this thesis yet.

- `hier-parallel-exclusive` specifies *parallel disjoint* $SpMM^TV$ *without planning*,
- `hier-parallel-exclusive-planned` specifies *par. dynamically planned disjoint* $SpMM^TV$,
- `hier-parallel-shared` specifies $SpMM^TV$ *without planning*

- `hier-parallel-shared-planned` specifies *parallel dynamically planned SpMM^{T}V*,
- `hier-sequential` specifies *sequential hierarchical SpMM^{T}V*,
- `coo-sequential` specifies sequential SpMM^{T}V with a matrix in the COO format,
- `csr-sequential` specifies sequential SpMM^{T}V with a matrix in the CSR format and
- `mkl-csr` specifies *Math Kernel Library SpMM^{T}V*.

All `hier-*` modes expect the input matrix in the CSR format. Mode `mkl-csr` expects a CSR matrix but MKL internally converts the matrix into a different format. Mode `mkl-csr` is only available when the main executable was built with Math Kernel Library.

The main executable further expects two arguments, namely a matrix path and a vector path. The specified matrix and vector will be used as the input of the performed SpMM^{T}V operation. All calls to the main executable need to specify parameters at least in the following way:

```
$ ./src/spmmtv -m $mode $matrix_path $vector_path
```

A list of input matrices can be obtained with

```
$ ls ../../../matrices-binary
```

A list of input vectors can be obtained with

```
$ ls ../../../vectors
```

Note that vectors which contain `y1` or `y2` in their file names are actually input vectors for the test executable.

Running the main executable with a valid mode, matrix path and vector path perfoms the specified SpMM^{T}V operation and prints the elapsed time. The desired thread count can be specified using option `-t`. What was referred to as α throught this thesis can be specified using option `-r`.

A.4 Running the Test Executable

This section provides information about running the test executable of the implementation presented in Chapter 5. Instructions on how to build the executable are in Section A.2.

The test executable is available in the build directory as described in Section A.2.

```
$ cd implementation/build/release
$ ./tests/tests --help
```

Executing the above commands prints the test executable's usage information. The test executable was created using the Boost.Test library.

All tests are performed by executing the following command:

```
$ ./tests/tests
```

Please note that the test executable contains hardcoded relative paths to matrix and vector directories. All previous steps in this chapter need to be performed exactly as specified for the above command to succeed. The test executable performs all implemented sequential and parallel SpMM^{T}V modes with all matrices and verifies the correctness of the results. Math Kernel Library SpMM^{T}V is the only mode that is not tested by this executable. The test executable also contains tests which verify the correctness of loaded COO and CSR format matrices.

A.5 Additional Utilities

Section 5.7 mentions that several accompanying utilities have been created in the course of developing the implementation presented in Chapter 5. Three of the most important utilities are included on the attached disc in directory `thesis-data`:

- `mmtobin` – a program to convert matrices from the Matrix Market format to the binary format accepted by the main executable (Section A.3);
- `random-vector` – a program to generate random vectors to accompany matrices created by `mmtobin` and
- `pyspmmtv` – a program to independently perform SpMM^{T}V and generate model output vectors for the test executable (Section A.4).

The installation and usage of these utilities is not explained due to time constraints. Nonetheless the source code of the utilities is still provided for reference purposes. The utilities are fully fledged Python packages with dependency information and have documented command line interfaces.

Appendix **B**

Contents of Attached Disc

```
readme.txt ........................ the description of the disc's contents
thesis-data ...................... the implementation and data directory
    implementation
    matrices-binary
    mmtobin
    pyspmmtv
    random-vector
    vectors
thesis-src ................ the LaTeX source code directory of the thesis
thesis.pdf ................................ the Master's thesis in PDF
```