Master's Thesis

Parallel Sparse Matrix-Matrix Multiplication

Luben Alexandrov

December 15, 2014

Advisers: Prof. Dr. rer. nat. Peter Sanders
Dipl.-Phys. David Kernert

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, 15.12.2014

Acknowledgments

I would like to thank my thesis advisers Prof. Dr. Peter Sanders and Dipl.-Phys. David Kernert for their commitment and active guidance throughout the work. I appreciate that they gave me the opportunity to work on this thesis and I am thankful for the many fruitful ideas they gave me.

I want also to thank my family for their support during my studies.

Abstract

The thesis investigates the BLAS-3 routine of sparse matrix-matrix multiplication (SpGEMM) based on the outer product method. Several algorithmic approaches have been implemented and empirically analyzed. The experiments have shown that an algorithm presented by Gustavson [22] outperforms other alternatives.

In this work we propose optimization techniques that improve the scalability and the cache efficiency of the Gustavson's algorithm for large matrices. Our approach succeeded to reduce the cache misses by more than a factor of five and to improve the net running time by 30% with some instances. The thesis also presents an algorithm for *flops* estimation, which can be used to determine an upper bound for the density of the result matrix.

Furthermore, the work analyzes and empirically evaluates techniques for parallelization of the multiplication in a *shared memory model* by using *Intel TBB* and *OpenMP*. We investigate the cache efficiency of the algorithm in a parallel setting and compare several approaches for load balancing of the computation.

Zusammenfassung

Die Masterarbeit untersucht die BLAS-3 Operation für Multiplikation von dünnbesetzten Matrizen durch das dyadische Produkt. Eine empirische Evaluation von unterschiedlichen Vorgehnsweisen stellte fest, dass der Algorithmus von Gustavson [22] die beste Performanz erzielt.

Diese Arbeit präsentiert Verfahren für Verbesserung der Cache-Effizienz und Skalierbarkeit des Gustavson Algorithmus. Die vorgestellte Techniken führten zu 30% Verbesserung der Laufzeit und zu Verringerung der Cachemisses mit mehr als Faktor fünf mit manchen Probleminstanzen. Weiterhin wird einen Algorithmus für Evaluation der Anzahl der benötigeten mathematischen Operationen (*flops*) vorgestellt. Dieser Wert kann dann für Bestimmung einer Obergränze für die Dichtigkeit der Ausgabematrix verwendet werden.

Zusätzlich ist die Multiplikation in *shared memory model* unter Zuhilfenahme von *Intel TBB* und *OpenMP* parallelisiert. Unterschiedliche Strategien für Lastverteilung wurden implementiert und evaluiert.

Contents

1 Introduction **1**
1.1 Contributions . . . 3
1.2 Outline . . . 4

2 Background and Related Work **5**
2.1 Related Work . . . 5
2.2 Datastructures . . . 7
2.2.1 Triples . . . 7
2.2.2 Hash Table . . . 8
2.2.3 Compressed Sparse Row (CSR) . . . 8
2.2.4 Compressed Sparse Column (CSC) . . . 10
2.2.5 Doubly Compressed Sparse Column (DCSC) . . . 10
2.3 Algorithms . . . 11
2.3.1 Inner Product . . . 11
2.3.2 Outer Product . . . 12
2.4 Limitations . . . 12

3 Problem Instances and Test System **15**
3.1 Test System and Implementation Details . . . 15

4 Sequential Algorithms **18**
4.1 Algorithms . . . 18
4.1.1 join_aggregate . . . 18
4.1.2 outerP_hash . . . 20
4.1.3 outerP_hash2 . . . 21
4.1.4 Gustavson . . . 22
4.2 Experiments . . . 24
4.3 Conclusion . . . 26

5 The Density Estimator **28**
5.1 Problem Description . . . 28
5.2 The Estimator . . . 31

6 Cache Efficiency **33**
6.1 Optimization Idea . . . 33
6.2 Initial Evaluation of the Concept . . . 34
6.3 The Custom Hash Table . . . 39
6.4 Fine-tuning of sparse_cemm . . . 44
6.5 Generalizing the Algorithm . . . 46

6.6 Conclusion 49

7 Parallelization **51**
7.1 Parallelization Framework 51
7.2 Horizontal Partitioning 53
7.3 Cache Efficiency in a Parallel Setting 57
7.4 Load Balancing 59
7.5 Conclusion 63

8 Experimental comparison with other frameworks **65**

9 Conclusion **67**

10 Future work **69**

Bibliography **71**

A Appendix **75**

List of Figures

2.1 Triple representation of a sparse matrix. 7

2.2 The compressed sparse row data structure. 9

2.3 The compressed sparse column data structure. 10

2.4 The doubly compressed sparse column data structure. 11

2.5 The outer product algorithm. 12

3.1 Cache hierarchy of Intel Xeon X5550 17

4.1 Performance of the sequential algorithms for SpGEMM 25

4.2 Cache misses of the different sequential algorithms for multiplying $matrix5_0$ 27

5.1 Connection between the densities of both input matrices and the output matrix 29

5.2 Connection between the densities of both input matrices and the output matrix, variant 2. 29

5.3 Number of flops contributing to an element 30

6.1 Execution time of `gustavson` with different hash tables; L3 cache hit ratios; matrix with random distributed elements. 36

6.2 L2 and L3 cache misses for the different hash tables. 37

6.3 Number of retired instructions. 38

6.4 Performance of different hash functions. 40

6.5 Difference of the execution time by core, deviation of the cache misses. 43

6.6 Execution times (ms) for two matrices. 49

7.1 Task scheduler with a Thread Pool, as in Intel TBB. 52

7.2 Parallelization of `gustavson` by using equidistant 1-D partitioning. . 53

7.3 Speedups factors for different matrices and variation of the number of tasks; allocation of x and xb in each task *verus* allocation once per thread 55

7.4 Execution times for `gustavson` and `sparse_cemm` when parallelized; *matrix*6_2 58
7.5 Improvement of the L3 cache efficiency; *matrix*6_2 59
7.6 Scalability of `sparse_cemm`, multiplying the road network of Europe (*matrix*7_0) 60
7.7 Example for a matrix with an unbalanced structure. 61
7.8 Performance of the parallelization with different strategies for load balancing; *matrix*6_15 62
7.9 Performance of the parallelization by increasing the granularity of the partitioning; *matrix*6_15 63
7.10 Parallelization with TBB and OpenMP; *matrix*6_15 64

8.1 Comparison with a widely used system for numerical linear algebra, sequential execution 66
8.2 Further improvement of the speedup by `sparse_cemm` 66

A.1 L3 cache misses for both algorithms for different tasks; *matrix*6_2 . . 76
A.2 Improvement of the speedup after paralleilzation on 8 physical cores . 76

List of Tables

3.1 Description of test matrices 16

3.2 Some metrics of the test matrices 16

6.1 Performance of `minimal_hash_map3` and comparison with the dense arrays. 42

6.2 Comparison between original gustavson and sparse_cemm when multiplying $matrix6_10$ 46

A.1 Sequnetial algorithms, execution times in ms 75

A.2 Execution times [ms] for the parallelized `gustavson` algorithm with different task numbers; allocation and destruction of the temporary arrays is done in every task. 75

List of Algorithms

1 The join_aggregate algorithm, matrix A has n rows. 19
2 The outerP_hash algorithm. 20
3 The `gustavson` [22] algorithm. 22
4 Pseudo code of the *flops estimator* 31
5 `sparse_cemm` algorithm. 45
6 `general_sparse_cemm` 48

1. Introduction

Matrix-matrix multiplication is a basic mathematical operation, often used as a subroutine in algorithms and numerical methods. The matrices that arise from real world problems are usually sparse, i.e. the most of the values in the matrix are zero. For example, the matrix describing the road network of Europe has dimensions $(51 * 10^6) \times (51 * 10^6)$, but consists of only $108 * 10^6$ nonzero values. The sparsity enables the development of space efficient data structures and fast algorithms that perform significantly better than the naive $O(n^3)$ multiplication.

Many real world problems lead to basic operations from the linear algebra. Therefore, the topic has been of great interest both for the academia and the industry. In 1979 [34] presents a library of basic linear algebra subprograms (BLAS). Since then the abbreviation has been commonly used by the research community to address the functionality as an interface. BLAS defines three levels of linear operations:

- BLAS-1: operations between two vectors
- BLAS-2: operations between a matrix and a vector
- BLAS-3: operations between two matrices

Part of the BLAS-3 level is the General Matrix Multiplication (GEMM), where is assumed that both input matrices have a dense structure. However, many problems lead to a multiplication of two sparse matrices with very large dimensions, where executing the multiplication on dense data structures is extremely inefficient. Because of that, a special case of Sparse General Matrix Multiplication (SpGEMM) has emerged as needed algebraic operation. A big point of interest is also the operation Sparse Matrix Vector Multiplication (SpMV). The routine is used in many linear solvers, e.g. computing eigenvalues and eigenvectors. However, this topic is not specifically discussed in this thesis. There has been done a considerable amount of work in finding good SpMV algorithms (e.g. [13] [47]), but there are not as many studies for the problem of sparse matrix-matrix multiplication. Although in the past years the topic has gained increasingly attention from the research community [3] [50], still many aspects of the problem have not been investigated.

There are many examples from different domains where SpGEMM is utilized. Methods for finding similarity (cosine similarity) between objects lead to matrix multiplication. If one object is compared to a set of objects, a matrix-vector multiplication is conducted. In case we want to examine the *many-to-many* relationships, a matrix-matrix multiplication is needed. Cosine similarity is widely used in text mining, clustering and bioinformatics.

Furthermore, there is a dualism between graphs and matrices [20]. Every topology of a graph $G = (V, E)$ can be represented as a adjacency matrix $A^{n \times n}$, where $n = |V|$ and the edges E form the nonzero elements in the matrix. If we number the vertexes from 1 to n this can be expressed as $A(i, j) \neq 0 \iff \exists \{u, v\} \in E : u = i \wedge v = j$. Thus, we can use matrix multiplication as a subroutine in graph algorithms [51]. For example, the product of multiplying the graph matrix by itself k times is a matrix that contains the information if there is path of length k between two vertices. This technique is used to implement algorithms for determining reachability between nodes [43]. Further, [51] depicts an algorithm for solving All Pairs Shortest Paths problem in a graph by using matrix multiplication.

In [19] is suggested that a graph contraction can be executed through matrix multiplication, where the adjacency matrix $A^{n \times n}$ is multiplied by a specially generated hypersparse matrix S, which contains n nonzero elements. The contracted graph is then given by the product SAS^T

As a low level mathematical operation, sparse matrix multiplication is also of importance in numerical computing. For example SpGEMM might be relevant in methods for solving Partial Differential Equations (PDEs). On its side PDEs are used in numerous areas like engineering, finance, thermodynamics, electrodynamics and applied physics in general. The Finite Elements Method (FEM) is widely used in civil and mechanical engineering and also involves PDEs.

In [50] further use cases for sparse matrix multiplication are given.

It can be concluded that there is an increasing demand for fast and scalable algorithms for sparse matrix multiplication. Therefore, the goal of this master thesis is to research the state-of-the-art methods for sparse matrix multiplication and evaluate new techniques for further improvement. To achieve this goal, we utilize the power of parallelism as nowadays multicore architectures has become common hardware [1]. This rapid development in the hardware industry has made possible that parallel computing has emerged as a powerful trend and is getting increasing attention. The new paradigm gives a whole other dimension for developing algorithms. Many old algorithms have to be rethought and adapted so that they are able to fully utilize the parallel resources.

A second development in the hardware is the adoption of larger memory hierarchies. Up to date there are processors with 20MB of Level-3 cache. Algorithms should be intelligently constructed also in this regard, so they leverage this advantage. In this thesis we analyze the problem of sparse matrix multiplication from the new perspective of parallel computing and cache-efficiency.

[1] Some speculate that the Moore's Law will soon be not valid for number of transistors on chip, but it will continue to hold for number of cores per chip.

1.1 Contributions

Throughout this work we will use A and B as input matrices and C as their product. The nonzero elements of a matrix A are denoted with $nnz(A)$. $a(i, j)$ (or a_{ij}) refers to the element in the matrix in row i and column j. $A(i, \cdot)$ and $A(\cdot, j)$ represent the whole i-th row and the whole j-th column accordingly.

The thesis concentrates on the outer product method for multiplying sparse matrices, since it has proven to be the most efficient approach [4] (more details will be elaborated in chapter *4*). In 1978 Gustavson [22] presented an algorithm (further on referred as `gustavson`) that uses the outer product approach for implementing SpGEMM. The algorithm has proven as efficient as well in time as in space and it is still the state of the art, as seen from the experiments in the thesis (see also [4]). For computing the product $A^{p\times q} \times B^{q\times n} = C^{p\times n}$ the algorithm has time complexity of $\mathcal{O}(flops + nnz(A) + nnz(C) + n)$, where flops is the minimal number of floating-point operations[2] that are needed do calculate the product and nnz(C) are the nonzero values in the result matrix. The space complexity of Gustavson's algorithm is $\mathcal{O}(2n)$, the method uses two dense arrays of size n as an accumulator. This work empirically evaluates `gustavson` by comparing it with other algorithms. The experiments have shown that the method outperforms the competitors. Outgoing from this principle, we show that one can apply further improvement by presenting a hash-based and load-balanced parallel algorithms.

Flops estimator: As a first contribution, this work presents a simple algorithm for evaluating the needed floating point operations prior to the matrix multiplication. The value can be used to determine an upper bound for the density of the rows of C. This information is of importance for algorithms presented in the thesis, regarding cache efficiency and load balancing.

Cache efficiency: The work proposes a cache-efficient sparse accumulator that improves the space complexity of the original `gustavson` algorithm from $\mathcal{O}(2n)$ to $\mathcal{O}(max_{1\leq i\leq p}(nnz(C(i, \cdot))))$. The time complexity transforms to $O(\delta flops + nnz(A) + \delta nnz(C) + n)$, where δ is expected to be a constant factor. The new algorithm (`sparse_cemm`) is based on `gustavson`, however it replaces dense arrays in the classical method with a specially tuned hash table. Our modification leads to significantly less cache misses for large scale matrices. The better cache efficiency reduces the execution time by 30% for some problem instances. Further, the influence of *NUMA* effects are investigated. The presented algorithm is more resistant to such effects. We also propose an algorithm that generalizes our approach for all problem instances.

Parallelization: The presents methods for parallelization of `gustvson` in a *shared memory model* and evaluates different load balancing approaches. The default load balancing strategies of *Intel TBB* and *OpenMP* are compared with some techniques for manual distribution of the computation among the threads. Furthermore, the effects of our cache efficiency improvement are evaluated in the parallel setting.

[2]In the literature *flops* sometimes stands also for Floating-Point Operations per second. This ratio is however more useful for hardware benchmarking. In this work the term describes the number of arithmetical operations.

`sparse_cemm` scales significantly better than the original algorithm when parallelized. The improvement in the space complexity that was introduced, makes it possible to run the algorithm on more threads. Secondly, the influence of the cache misses rises in the parallel setting, since all threads have to compete for the shared cache. The experiments have shown an interesting result: although for some instances both algorithms have equal performance in the sequential case, `sparse_cemm` achieves a 26% faster execution time in a parallel setting.

1.2 Outline

In chapter *2* the theoretical foundations of sparse matrix multiplication are presented and different data structures for storing sparse matrices are discussed. Furthermore, two basic approaches, inner product and outer product, for matrix multiplication are introduced. The next chapter presents the problem instances and the test system that was used for the evaluation. Afterwords, in chapter *4* we search for an efficient SpGEMM algorithm that solves the problem in the sequential case. In this chapter we also present the `gustvson` algorithm, which has proven to be the most efficient in the tests. Chapter *5* presents an algorithm determining an upper bound for the density of the output matrix. In chapter *6* we propose a new algorithm for multiplying sparse matrices, which is based on `gustavson`, but causes less cache misses. Chapter *7* discusses different strategies for parallel sparse matrix multiplication in a *shared memory model*. Chapter *8* compares the presented algorithms with a state of the art competitor - a widely used commercial system for scientific computation. Finally, in chapter *9* the results of this thesis are summarized and *10* elaborates possibilities for further research.

2. Background and Related Work

For simplification and without loss of generality we will assume multiplication of qudratic matrices in the analyses of the algorithms and data structures, i.e. $A^{n \times n} \times B^{n \times n} = C^{n \times n}$. All of the presented algorithms and data structures are also applicable for rectangular matrices. In this work a one-based matrix notation is used, i.e. rows and columns start with index 1.

2.1 Related Work

There has been a lot of research on the topic of general matrix multiplication. The $\mathcal{O}(n^3)$ bound of the trivial algorithm was broken and reduced to $\mathcal{O}(n^{2.81})$ by Strassen in 1969 [45]. In 1990 Coppersmith and Winograd [9] improved the lower bound further, as they presented an algorithm in $\mathcal{O}(2^{2.38})$, although with very large constant factors. An important remark is that one has to differentiate between the general matrix multiplication and the sparse matrix multiplication. These are two different problems, therefore the algorithms for dense matrices are not in the focus of the thesis. It is possible to develop algorithms with lower complexity by using the sparse property of the matrices and employing sparse data structures. The goal for the sparse algorithms is to achieve complexity that is more dependent on the nonzero *flops* and less dependent on the dimension factor n.

Duff et al. [14] proposed sparse extension to the Level-3 BLAS operations. However, their multiplication algorithms use a dense data structure for the result matrix, which reduces significantly the applicability. Sulatycke and Ghose [46] discussed the topics of cache efficiency and parallelization for sparse matrices (also using dense structures for some of the matrices). Mccourt, Smith and Zhang [35] demonstrate how the multiplication can benefit from graph coloring approaches. Their algorithms also use dense formats for the result matrix, further in some cases one of the two input matrices is dense.

An important restriction, when multiplying large scale problem instances, is to store both input matrices and their product in sparse data structures. It is crucial to retain low space complexity, storing even one matrix in a dense structure would make an algorithm not applicable for even relatively small matrices. Therefore,

in our research we concentrate on a *fully* sparse matrix multiplication, where only sparse data structures are used.

Yuster and Zwick [50] described an algorithm for sparse matrix multiplication, where they used a permutation of the columns and the rows of both input matrices A and B, so the multiplication can be decomposed in a dense section and a sparse section. Accordingly, they proposed that the dense section is multiplied with a fast algorithm for dense matrix multiplication [9] and the sparse section is computed with a sparse matrix multiplication algorithm [22]. The result matrix C is the sum of the two interim results. The complexity of their algorithm is $\mathcal{O}(\max(nnz(A), nnz(B))^{0.7} n^{1.2} + n^{2+o(1)})$. The usefulness of the approach is limited by the algorithm used for the multiplication of the dense sections, since it has large constant factors.

Gustavson [22] presented an algorithm where all of the matrices in the multiplication use sparse data structures (*CSR*, see next section). The method is suitable for practical use, it has a complexity of $\mathcal{O}(flops + nnz(C) + nnz(A) + n)$, observe that the matrix dimension is present only at a linear scale. Buluc and Gilbert [4] described an alternative algorithm, suitable for *hypersparse* matrices, where a matrix $X^{n \times n}$ is defined as *hypersparse* when $nnz(X) < n$. Their method uses the doubly compressed sparse column data structure (see next section) and has time complexity of $\mathcal{O}(nzc(A) + nzr(B) + flops * \log ni)$, where $nzc(A)$ denotes the nonzero columns of A, respectively $nzr(B)$ are the number of nonzero rows of B, ni is the number of indexes i for which $A(\cdot, i) \neq 0 \wedge B(i, \cdot) \neq 0$. The advantage is that the time complexity does not depend on n. That would be beneficial for *hypersparse* matrices, where there are columns and rows that do not contain values at all.

Another topic in the current literature is the parallelization of SpGEMM. There are significant number of publications regarding the parallelization of the dense multiplication (e.g. [6], [17]). However, introducing sparsity to the multiplication is a relatively new problem. Buluc and Gilbert [4] [3] discussed sparse algorithms in a distributed setting. They used a two-dimensional (2-D) partitioning and a hypersparse sequential algorithm as the core for a parallel method. Campagna, Kutzkov and Pagh [5] presented distributed communication-avoiding algorithm that uses hashing approaches.

Most of the current research focuses mainly on a distributed prallelization [15]. To our surprise, a parallelization in a shared memory model was not explicitly discussed in the recent literature, although it has been shown ([3] C184, [5]) that the communication costs play significant role in the execution time of distributed algorithms. In this thesis we focus on a shared memory parallelization. The main memory capacity is continuously increasing on server machines, up to date the random access memory can be ranging from several gigabytes up to terabytes. Thus, it is possible to multiply large matrices also in a shared memory environment.

Linear algebra operations have been widely used in different domains. Therefore, there are numerous libraries and systems that implement the BLAS interface, e.g. Intel Math Kernel Library (MKL) [26], Automatically Tuned Lineare Algebra Software (ATLAS) [24], Linear Algebra Package (LAPACK) [28], MATLAB [27]. However, only few support the special case of *sparse* matrix-matrix multiplication, e.g. from the previously mentioned, only MATLAB offers SpGEMM functionality.

2.2 Datastructures

Before discussing algorithms for multiplying sparse matrices, one has to address the problem of representing large scale sparse matrices in memory. As already mentioned, the graph of the European road network consists of $51 * 10^6$ nodes [12]. If the corresponding matrix is stored naive in a two dimensional dense array and each edge uses 4 Bytes the memory needed for storing the graph is $(51*10^6)^2 \times 4B =$ 10.4PB. The term "dense array" is used whenever an array data structure is referred that stores trivially all values, also these equal to zero. Thus, storing sparse matrices in two dimensional array is often not feasible, moreover there are matrices which are much bigger than the road network of Europe, e.g. Facebook has more than $1.1 * 10^9$ users, i.e. the Facebook's graph consists of more than a billion nodes. As most of these large scale matrices are extremely sparse, the only possibility to manipulate them is to store only the values not equal to zero, storing only the edges that actually exist. This chapter presents data structures for this task.

In addition, the efficiency of the data structure is also crucial for the execution time of the whole algorithm. Matrix multiplication is a low level algorithm, where the running time depends heavily on the complexity of the accessing and modifying operations of the data structures holding the matrices. Therefore, it is of big importance to select a space- and access- efficient data representation for the input and output matrices. We will discuss different structures in the following sections.

2.2.1 Triples

The triple format consists of three arrays, storing the row index, the column index and the associated value, see figure 2.1. The disadvantage of this representation is that there is no structure. For finding, if a particular value exists, a full linear scan over the list of triples is needed. Because of this, the triples are often sorted by the row index, by the column index or by both. In this way one reduces the search times, but still there is no fast (constant time) access for all elements of a specific row or column. When the list is sorted by row, *binary search* can find the starting triple of a particular row i in $\mathcal{O}(\log nnz)$, where nnz are the nonzero elements in the matrix. For finding the last element of i, a second *binary search* or onward linear scan is needed. An advantage is that the triple format is cache efficient, if the elements are accessed in a sequential order.

$$\text{A} = \begin{pmatrix} 0.12 & X & X & X & 3 \\ X & X & X & X & X \\ 2 & 1.8 & X & -1 & X \\ X & 1.9 & X & X & X \\ X & X & X & 2.72 & X \end{pmatrix}$$

row	column	value
1	1	0.12
1	5	3
3	1	2
4	2	1.9
3	2	1.8
3	4	-1
5	4	2.72

Figure 2.1: Triple representation of a sparse matrix.

2.2.2 Hash Table

One strategy to overcome the slow access time is to put the triples in a hash table. Here there are a couple of options. The coordinates pair $(row, column)$ can be used as the key and the according *value* as the value. A disadvantage of this data structure is that it is not possible to get all elements of a specific row in a constant time. An alternative is to use only *row* as the key, and a vector of $(column, value)$ tuples as the value. In this way a whole row, all nonzero values in the raw, can be retrieved in *expected* $\mathcal{O}(1)$. If we are looking for a specific value, we still have to do a linear or binary search in the row vector. The same approach can be used for fast column retrieval, where the key is the column index and value is then a vector of $(row, value)$ tuples. A third options is to combine the previous techniques and to have two-layered hash table. The top level hash table has *row* as key, value is a second hash table, which contains all values in the specific *row*. The second hash table has *column* as the key and the corresponding *value* as the value. In this way we have efficient access to specific elements as well as to a certain row. The whole row is in a hash table of its own, so retrieving of all its elements would mean iterating over it. If the hash table has many empty buckets and only few stored $(key, value)$ pairs, the iteration might be not very efficient.

A considerable disadvantage of the general randomized hash tables is that their lack of structure might lead to cache inefficiency. Furthermore, there is the overhead of calculating the hash function and collision resolution. Because of the possibility of collisions a $get(key)$ and $set(key, value)$ operations are in general not in $\mathcal{O}(1)$ but rather in *expected* $\mathcal{O}(1)$ time. It is possible that one has to iterate through other keys until the right one is found. The lack of structure has also another disadvantages. If we want to scan over the matrix (e.g. for output), we have to iterate through the hash table, which might be a costly operation. The same holds if one needs to get all elements of the matrix in general, for example for sequence of matrix multiplications (i.e. computing the product of multiple matrices, also called *chain matrix multiplication*). The obtained tuple list will be randomly distributed, so a posterior sorting of the tuples might be needed. Hash tables have also a memory overhead, since for storing n elements a table of larger size is needed, so the probability of collisions is not too high. Depending on the implementation there are also other factors that contribute to the memory overhead, i.e. additional pointers, dynamical growth, storing keys et cetera.

2.2.3 Compressed Sparse Row (CSR)

Motivated from the need of fast access to all elements in a row, this section presents the *compressed sparse row* (*CSR*) format [22]. The data structure, filled with the matrix A from fig. 2.1, is depicted in fig. 2.2.

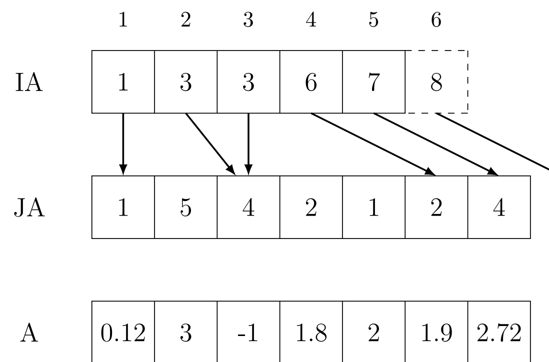

Figure 2.2: The compressed sparse row data structure.

The array JA contains the column indexes of the nonzero elements and A holds the values of the elements. The nonzero values are ordered by row, so all elements of a row are in a consecutive segment in JA and A. The IA array, also called *row pointer array*, contains pointers to the beginning of each row in JA. Each row has a pointer to the the column array, even if the row doesn't contain any values. Because of this we can access the rows of the matrix through the indexes of IA in $\mathcal{O}(1)$. In general the segment in the column array and the value array, where the row i resides, is specified by:

```
row_i_start:=IA[i]
row_i_end:=IA[i+1]-1
```

This leads to an interesting feature of the *CSR*, the size of the row i (the number of its nonzero elements) can be obtained in constant time via:

```
#nnz_in_row_i:=IA[i+1]-IA[i]
```

In order this invariant hold even for the last row of the matrix, a dummy element (or sentinel) is added at the end of the row pointer array. The value of the dummy element should be $nnz(A) + 1$. In this way, there is one branch less when iterating over IA, there is no need to check if the currently processed row is the very last one. In matrix multiplication algorithms there is a need of repeatedly iterating over the rows of the matrix, thus this one branch less plays significant role in the end execution time, as fewer branches lead also to fewer branch mispredictions. In our experiments we saw that a simple improvement as adding a dummy element might lead to 23% reduction in the running time of the whole algorithm.

It has to be also noticed that inside a row the values might be not sorted by their column index. For access to a specific value a linear scan in the row segment is needed, if the values are sorted a binary search would be possible.

The *CSR* data structure has various advantages and it s widely used in sparse linear algebra [41]. It is similar to the hash table approach, where the key is the row index, and the value is the row. In compressed sparse row the overhead of hashing doesn't exist and the row pointer array IA can be seen as an implicit hash table. Another advantage of the format is the cache efficiency of the arrays, since the rows of the matrix are chronologically ordered in consecutive segments. A disadvantage is that there is no efficient way to access a whole column of the matrix. Further consideration is that the nonzero elements of the matrix have to be sorted by row

when constructing the *CSR* format. Very often the elements of the matrices are stored in an unordered list of triples. So there will be an overhead of sorting for generating a compressed sparse row data structure from the triple list.

In graph theory the data structure is also well known and it is referred as *adjacency array* [36]. This shows once again the analogy between graphs and matrices.

2.2.4 Compressed Sparse Column (CSC)

Analog to *CSR* one can create a compressed sparse column (*CSC*) data structure (figure 2.3). The difference is that there is no row pointer array but a *column pointer* one. Thus, whole columns can be accessed in constant time and we can iterate over the matrix efficiently in column-wise manner. On the contrary, there is no efficient iteration over rows.

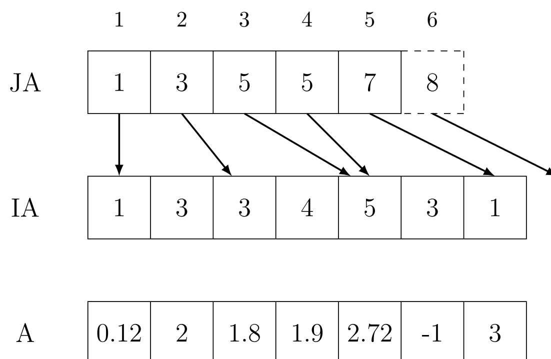

Figure 2.3: The compressed sparse column data structure.

The two formats have also another interesting characteristic. If we read a *CSC* format as if it was a *CSR*, and vice versa, we get the transposed matrix A^T. Hence, the problem of transposing a matrix can be solved by simply creating a compressed sparse row, or compressed sparse column, data structure of the matrix.

2.2.5 Doubly Compressed Sparse Column (DCSC)

The conventional *CSC* format stores pointers to all n columns, even to those, which do not contain any elements. For example in the matrix A the third column is empty, nevertheless we store a pointer for it. Further, there is no fast iteration over all non-empty columns. Buluc and Gilbert [4] present a new data structure, which aims to eliminate these drawbacks. The new format is called doubly compressed sparse column (*DCSC*) and is depicted in figure 2.4.

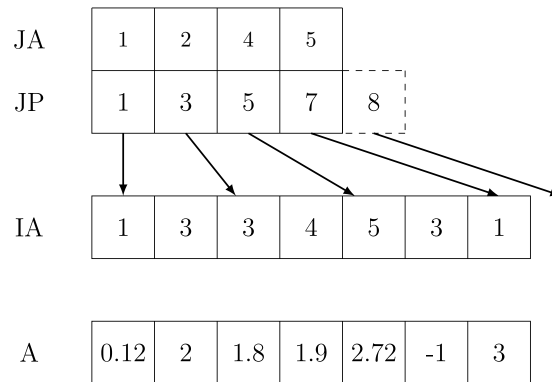

Figure 2.4: The doubly compressed sparse column data structure.

One can see from the graphic that the data structure stores only the columns that contain some elements. This feature is only relevant for*hypersparse* matrices ([4] defines hypersparse matrix as a matrix where $nnz < n$). A disadvantage of the *DCSC* is that the direct indexing of the columns is lost. If there is a need for that, one has to maintain an additional array with the original column pointers. Still, we have the advantage of fast iteration over all non-empty columns.

Analog to the doubly compressed sparse column, a doubly compressed sparse row data structure can be constructed. According to our observation, most sparse matrices, unless they are not partitioned, contain one or more elements per row and per column, which is why we do not consider the doubly compressed formats any further.

2.3 Algorithms

After some data structures suitable for storing sparse matrices have been presented, this section will discuss the two main algorithmic approaches for multiplying matrices.

2.3.1 Inner Product

Calculating the product $C = AB$ can be defined as $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$. The common algorithm for computing the matrix multiplication is the *row-column* method, also called *inner product*. An element of the result matrix is obtained through computing the dot product (inner product) between a row of the first matrix with a column of the second matrix through $c_{ij} = A(i, \cdot) \bullet B(\cdot, j)$. This approach has severe disadvantages regarding sparse matrix multiplication. In the classical algorithm we have to iterate n times (the number of columns of B) over the first row of A in order to calculate the first row of C. This can be avoided, as we will see in the next section. Another issue is the computation of the dot product. In terms of the data base jargon, this can be seen as a *join* problem, where the algorithm has to match the nonzero elements of the two vectors (the row from A and the column from B) by their indexes. This task is a costly operation. There are many possibilities to solve it, but all of these involve significant computational overheads such as sorting, additional branches, unnecessarily flops. The *join* problem is also bypassed with the *outer product*.

All in all, *inner product* is highly ineffective approach for executing SpGEMM and it is not suitable for practical use. The algorithm is in order of magnitudes worse than its counterpart (*outer product*) for sparse matrix multiplication. In our experiments, comparison between two naive implementations of the both methods, led to differences by a factor of 20000 in the execution times for particular matrices.

2.3.2 Outer Product

Another algorithmic approach for multiplying matrices is the *outer product*, called also *column-row* method. The structure of the algorithm is illustrated in figure 2.5.

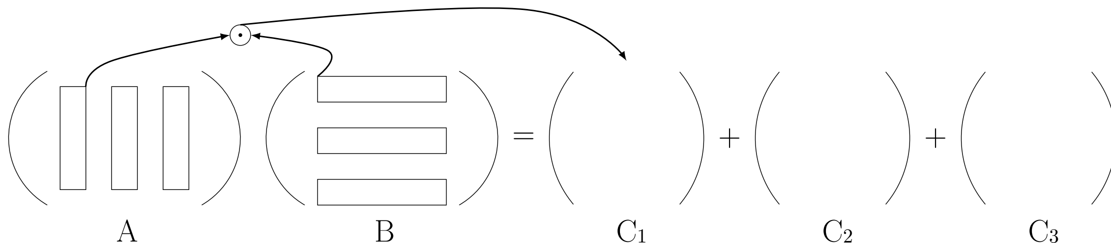

Figure 2.5: The outer product algorithm.

The result matrix C is computed by calculating the outer products (tensor products) between the columns of A with the respective rows of B. The result of the outer product between two vectors a and b is a matrix M and it is defined as $m_{ij} = a_i b_j$. Thus, we get that $C_i = A(\cdot, i) \odot B(i, \cdot)$, where $\odot$ stands for the outer product operator. At the end all matrices have to be aggregated in one solution matrix $C = \sum_{i=1}^{n} C_i$

As already mentioned, this approach have various advantages over the inner product, which is why it is widely used for multiplying sparse matrices. Through changing the order of the operations (additions are left for later time) we circumvent the *join* problem and reduce the number of iterations. There is only one iteration over the elements of the matrix A.

In the next chapters we will present some algorithms that change the order of iteration as they process the matrix A in a row-wise manner, matrix B is still processed row-wise. Nevertheless, such algorithms also employ the outer product strategy, since a column of the first matrix is multiplied with the row of the second matrix and the result is a matrix. The approach can be called *row-wise outer product.*

Since *outer product* has been proven as efficient way to multiply sparse matrices, the thesis concentrates on this approach. All of the algorithms in this work use the *column-row* method as their core, although some of them may iterate through the matrices in different fashion (row-wise outer product). The next chapters will demonstrate that although the algorithms have the same basis, there could be significant difference in their efficiency.

2.4 Limitations

Matrix multiplication is a very general mathematical operation. There are different classes of problem instances depending on multiple properties of the two input matrices, such as sparsity ratio, distribution of the nonzero elements, patterns of the

nonzero elements, dimensions et cetera. Different domains produce different type of matrices and one can make use of their specific features to further improve the general approach. The performance of the algorithms is strongly coupled to the structures and patterns in the two input matrices. Therefore, one has to keep in mind that improvement techniques are often specific for a certain class of instances (the thesis concentrates mainly on large scale sparse matrices). Further, we try to generalize them as much as possible and to derive some universal conclusions and techniques which can be used also in other domains and not only specifically for sparse matrix multiplication.

3. Problem Instances and Test System

This chapter gives an overview over the problem instances used in the experiments in this work.

The name and the kind of the matrices that were used for the evaluation can be seen in Table 3.1. The table collects all matrices used in the tests. Some of the instances were taken from the *Florida Sparse Matrix Collection* [12], others were custom generated ($matrix6_10$ - $matrix6_15$). The custom generated matrices represent graph matrices, where the nonzeros in each row are randomly distributed through the row. In our experimental environment we named (renamed) the matrices with the pattern $matrixX_Y$, where the idea is that the scale of the problem instance will grow with X, i.e. we assign problem instances with the same value for X to the same scale class. Y is used to differentiate between the problem instances in the same class.

The multiplication experiments in all upcoming chapters, consists of multiplying a particular matrix by itself.

Table 3.2 reveals more details about some features of the problem instances (dimensions, nonzeros, symmetry et cetera). The graph matrices are noted as binary because the values in these cases are 0 or 1. The non-binary matrices come from engineering problems and have *real* numbers for values (double precision was used).

Generally, there are many classes of problem instances. For example, we could separate the matrices in two types: graphs and matrices from engineering problems. The matrices from engineering problems tend to have diagonal structure (nonzeros distributed close to the diagonal). Graph matrices often have much larger dimensions and more complicated internal patterns, which are not easily recognizable.

3.1 Test System and Implementation Details

All of the experiments in this thesis, unless stated otherwise, were executed on a HP Workstation Z600 with 24GB of main memory and two Intel Xeon X5550 processors with 2.67GHz of peak frequency. Each processor has 4 physical cores and supports

Name	Florida Name	Matrix kind
matrix2_1	nemeth07	theoretical chemistry problem
matrix3_1	lhr71c	chemical process
matrix3_3	c-71	nonlinear optimization problem
matrix3_6	DIMACS10/preferentialAttachment	graph
matrix4_0	consph	FEM engineering problem
matrix4_3	DIMACS10/rgg_n_2_22_s0	graph
matrix5_0	ASIC_680ks	circuit simulation problem
matrix5_10	rajat31	circuit simulation problem
matrix5_13	DIMACS10/M6	graph
matrix6_2	DIMACS10/hugebubbles-00010	graph
matrix6_10	none (custom matrix)	random graph
matrix6_12	none (custom matrix)	random graph
matrix6_13	none (custom matrix)	random graph
matrix6_14	none (custom matrix)	random graph
matrix6_15	none (custom matrix)	random graph
matrix7_0	DIMACS10/europe_osm	road network Europe

Table 3.1: Description of test matrices

Name	Dimensions	Nnz	Nnz in result	Symmetric	Binary
matrix2_1	9.5K x 9.5K	394.8K	892.1K	true	false
matrix3_1	70.3K x 70.3K	1,528K	8.2M	true	false
matrix3_3	76.6K x 76.6K	859.5K	52.5M	true	false
matrix3_6	100K x 100K	999.9K	32.9M	true	true
matrix4_0	83.3K x 83.3K	6M	26.5M	true	false
matrix4_3	4.1M x 4.1M	60.7M	187M	true	true
matrix5_0	682.7K x 682.7K	1.6M	15.9M	false	false
matrix5_10	4.6M x 4.6M	20.3M	53.1M	false	false
matrix5_13	3.5M x 3.5M	21M	53.1M	true	true
matrix6_2	19.4M x 19.4M	58.3M	121.5M	true	true
matrix6_10	19.4M x 19.4M	58.3M	175.1M	false	true
matrix6_12	20M x 20M	69.9M	214.7M	false	true
matrix6_13	20M x 20M	69.9M	245.2M	false	true
matrix6_14	10M x 10M	30M	8.9M	false	true
matrix6_15	10M x 10M	73.9M	392.7M	false	true
matrix7_0	50.9M x 50.9M	108.1M	182.5M	true	true

Table 3.2: Some metrics of the test matrices

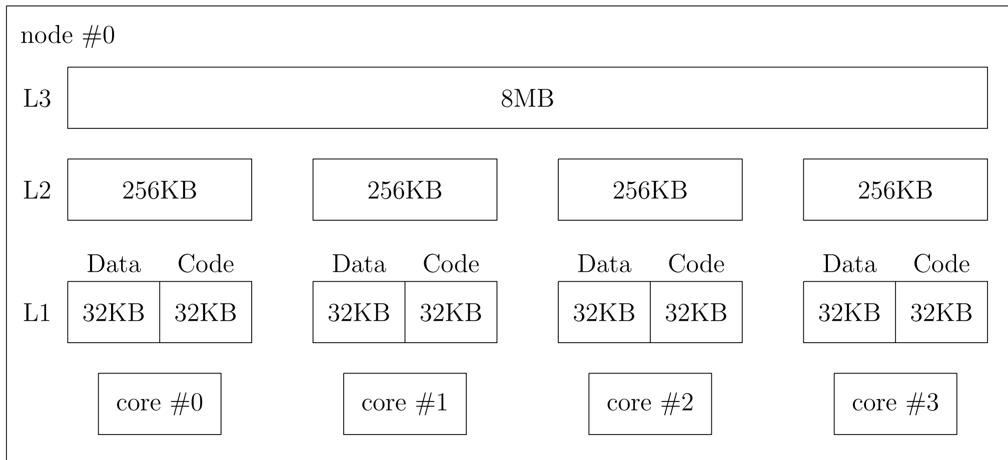

Figure 3.1: Cache hierarchy of Intel Xeon X5550

the Hyper-Threading Technology from Intel, which leads to a sum of 8 physical and 16 logical cores. Intel Xeon X5550 has a three level cache hierarchy, 256KB L1 cache, 1MB L2 cache and 8MB L3 cache. Each core has its own Level-1 and Level-2 cache, the Level-3 cache is shared between all cores. The whole architecture can be seen in Fig. 3.1. The L1 cache is divided in two equal sections, where one of them is used for data and the other is reserved for instructions.

The operating system was SUSE Linux Enterprise Server (SLES) 11. All of the algorithms were implemented in C++ 11 and were compiled with the GNU Compiler Collection (GCC) version 4.7. The code was always compiled with the compiler flag for highest level of optimization `-O3`.

Further implementation details, e.g. considering used libraries, will be discussed in the next chapters when needed.

4. Sequential Algorithms

This chapter presents and evaluates some sequential algorithms for SpGEMM. For constructing efficient parallel algorithms, we first analyze the task of sparse matrix multiplication in the sequential setting, in order to gain useful insights into the problem. At the end we will have an algorithm that can be used as the sequential core of the parallel one. Furthermore, since the goal of this thesis is to contribute new ideas to the problem of sparse matrix multiplication, it is important to evaluate different algorithms and look for the *state of the art*.

4.1 Algorithms

As already mentioned, the inner product approach is not feasible for sparse matrices, therefore all of the algorithms presented in this section implement outer product. Another criteria is that every matrix in the multiplication, both input matrices and the result matrix, must use a sparse data structure. Otherwise, the algorithm would be not scalable and not applicable for even relatively small matrices. Further major challenge is that the computation have to be more dependent on the number of nonzeros in the matrix and less on the dimensions.

In this thesis we are interested both in the theoretical and practical qualities of the presented methods. To evaluate them, there is a need of a more low level analysis of the algorithms. Therefore, when analyzing the complexities we will often consider constant factors that are usually omitted in purely theoretical algorithm analysis.

4.1.1 join_aggregate

First, we will present a simple algorithm for SpGEMM that works on a triple level (`join_agregate`). The approach is very basic and probably well known by the research community (e.g. [4] uses similar idea for the sequential algorithm).

The matrices A and B are stored in the *CSR* format. The result matrix C has triple representation. The pseudocode of the method can be seen in Algorithm 1. For the arrays of the *CSR* strucutres we use notation introduced in chapter 2, i.e. IA is the row pointer array, JA contains the column indexes and A holds the actual values.

Algorithm 1 The join_aggregate algorithm, matrix A has n rows.

```
function JOIN_AGGREGATE(matrixA:CSR, matrixB:CSR):List<Triple>
    for i ∈ 1...n do
        startRowA ← IA[i]
        endRowA ← IA[i+1]
        for j ∈ startRowA...endRowA do
            startRowB ← IB[JA[j]]
            endRowB ← IB[JA[j]+1]
            for k ∈ startRowB...endRowB do
                matrixC.insertTriple(i, JB[k], A[j] * B[k])
            end for
        end for
    end for
    aggregate(matrixC)
    return matrixC
end function
```

According to the outer product, every element a_{ij} is multiplied with all of the elements in the row $B(j, \cdot)$. The algorithm deviates from the classical column-row method as the matrix A is processed not in column-wise manner but in row-wise one. Nevertheless, the algorithm uses the basis of outer product. We are storing the result of every multiplication $a_{ij}b_{j\cdot}$ as a $(row, column, value)$ triple into $matrixC$. Potentially there will be more than one triple for the same coordinates, so an aggregation step is required, where all of the values for the same coordinate are added. There are different options for the aggregation itself. For example the list of triples could be sorted by row and by column, so the values which belong to the same coordinate are contained consecutively in a list. In this case, the aggregation can be done with one linear scan over the sorted triple list. Thus, the complexity of the aggregation in this case would be $\mathcal{O}(m \log m + m)$, where $m = |triples|$. More efficiently the aggregation can use a temporary hash table to sum up the triples. The key would be the coordinates pair $(row, column)$. Then the values can be summed with only only one linear scan over the initial triple list. A third option is to use a priority queue for the aggregation, were the key is again the combined coordinates. First we build the queue with the triple list. After that we can aggregate by retrieving the triples in ascending order with $deleMin()$ operations. The priority queue approach was used by Buluc and Gilbert [4].

In our experiments we tested the `join_aggregate` algorithm with the hash table aggregation. The two 32 Bit integers *row* and *column* are combined in one 64 Bit key with two simple bit operations:

$$key = (row << 32) | column$$

A considerable disadvantage of the hash table is the fact that after the aggregation we need to iterate through it in order to copy the final values to the originally triple matrix format.

The complexity of the algorithm is $\mathcal{O}(multiplication_flops + T_{aggregation} + n + nnz(A))$. We can see here for the first time a major problem with the sparse matrix multiplication, the computation is strongly dependent on the nonzero patterns of

both input matrices. Initially, it is not known how many nonzero elements there are going to be in the product C. The multiplication of two sparse matrices can produce a sparse matrix as well as a completely dense one. Accordingly, without any further calculations one cannot predict the needed flops, thus at first we do not have even vague estimate for the running time. The interesting topic of flops estimation, will be discussed in detail in the next chapters.

The space complexity of the algorithm $\mathcal{O}(|triples|)$, where $triples$ is the list of triples before the aggregation step. It could be specified further that $triples = multiplication_fops$, as each multiplication inserts a new triple into the list. Thus, the space complexity of `join_aggregate` is $\mathcal{O}(|multiplication_flops|)$.

4.1.2 outerP_hash

The next algorithm is called `outerP_hash` and uses hash tables for the matrices B and C, A is stored as triples (see Algorithm 2). The rows of matrix B are stored as vectors of tuples in the hash table. A tuple consists of a value and the column index of this value. A whole row can be efficiently retrieved from the hash table through its row index.

The list of triples of matrix A does not have to be sorted with this algorithm. Therefore, there will be random write access in the grid of the result matrix. This motivate the use of a second hash table for the matrix C, where the key is a tuple of $(row, column)$ coordinates.

Algorithm 2 The outerP_hash algorithm.

Input: matrixA: List<Triple>, matrixB: HashTable<int,vector<tuple> >
Output: matrixC: HashTable<coordinates, value>

```
function OUTERP-HASH(matrixA, matrixB)
    for triple ∈ matrixA do
        rowB ← matrixB[triple.column]
        for (column, value) ∈ rowB do
            row_C ← triple.row
            column_C ← column
            key ← (row_C ≪32) | column_C          ▷ combining the coordinates
            if matrixC[key] = ⊥ then
                matrixC.insert(key, triple.val∗value)
            else
                matrixC[key] += triple.val∗value
            end if
        end for
    end for
    return matrixC
end function
```

`outerP_hash` stores the calculated values directly on the right coordinates in C, thus there is the advantage that one does not need an aggregation step at the end, the aggregation is done on the fly into the hash table.

On the other hand, the use of hash tables for the matrices B and C comes with a price: there is an overhead of hashing and collisions. Another disadvantage of the algorithm is that at the end the result matrix C is stored in unstructured way, so to output the matrix, one will have to iterate over the hash table. The format of C could be also problematic if a sequence of multiplications is needed, for example when exponentiating a matrix.

One idea for improvement would be to sort the list of triples of the matrix A by their column index and than to run the algorithm. In this way the algorithm will first iterate over the first column of matrix A. Respectively every element in the column will be multiplied with the nonzero elements in the very first row of B. Thus, the whole row $B(1,\cdot)$ can be loaded into the cache. After the outer product $A(\cdot,1) \odot B(1,\cdot)$ is computed, the row $B(1,\cdot)$ will be not needed in the next steps of the multiplication. All in all, we gain locality of the data, in the sense that the matrix B is processed on row-wise manner. If the triple list is not sorted, there would be random access over the rows of B and a certain row have to be loaded multiple times into the cache. Thus, beforehand sorting will reduce the cache misses caused by the multiplication. Nevertheless, in the experiments the overhead of sorting the matrix A column-wise have proven to be dominating over the reduction of the cache misses. Moreover, the sorting causes cache misses on its own. It can be concluded that in this case the sorting is not really beneficial.

The complexity of the algorithm is $\mathcal{O}(\alpha flops + nnz(A))$, where α is the combined constant factor (more precisely expected to be a constant factor) needed for the hashing. α summarizes the time needed to retrieve value from the hash table B and to write in the hash table C. $flops$ describes all of the needed arithmetical operations, i.e. multiplications and additions:

$$flops = multiplication_flops + addition_flops \tag{4.1}$$

An advantage of the algorithm is that the execution time does not depend on n. The space complexity of the method is $\mathcal{O}(1)$, since the only additional space comes from the hash tables, which will be always with constant factor larger than the needed space for the elements stored in them.

4.1.3 outerP_hash2

After analysis of the mechanics of the previous algorithm, it can be seen that the data structure of the matrix B has the same characteristics as a compressed sparse row format, the matrix is stored row-wise and there is a fast access to each row. Hence, we can improve the algorithm by just interchanging the data structure of B with a classical CSR. In such way there will be no overhead for hashing and the cache efficiency may improve due to the arrays of the compressed sparse row. `outerP_hash2` is exactly the same as the previous method but only with matrix B in a CSR data structure.

The complexity of the algorithm doesn't change, but the constant factors have been reduced, since α depends now only on the matrix C.

4.1.4 Gustavson

Finally, this section presents an algorithm which does not have an extra aggregation step and uses the convenient CSR format for all of the matrices in the multiplication. In Algorithm 3 one can see the pseudo code of the `gustavson` algorithm [22].

Algorithm 3 The `gustavson` [22] algorithm.

Input: matrixA: CSR, matrixB: CSR
Output: matrixC: CSR

```
 1: function GUSTAVSON(matrixA, matrixB)
 2:     ip ← 0
 3:     xb:Array[1...n]
 4:     x:Array[1...n]
 5:     for i ∈ [1...n] do
 6:         xb[i] ← -1                                        ▷ Initialization
 7:     end for
 8:     for i ∈ [1 ... IA.size − 1] do
 9:         IC[i] ← ip
10:         startRowA ← IA[i]
11:         endRowA ← IA[i + 1] − 1
12:         for jp ∈ [startRowA ... endRowA] do
13:             j ← JA[jp]
14:             startRowB ← IB[j]
15:             endRowB ← IB[j + 1] − 1
16:             for kp ∈ [startRowB ... endRowB] do
17:                 k ← JB[kp]
18:                 if xb[k] ≠ i then
19:                     JC.pushBack(k)
20:                     ip++
21:                     xb[k] ← i
22:                     x[k] ← A[jp] ∗ B[kp]
23:                 else
24:                     x[k] ← x[k] + A[jp] ∗ B[kp]
25:                 end if
26:             end for
27:         end for
28:         for vp ∈ [IC[i] ... ip) do        ▷ Copy the calculated row i to C
29:             v ← JC[vp]
30:             C.pushBack(x[v])
31:         end for
32:     end for
33:     IC[IA.size] ← ip                                  ▷ Add dummy element
34:     return matrixC
35: end function
```

We used the notation from section 2.2, i.e. matrix C consists of the arrays IC, JC and C. Initially, the value $nnz(C)$ is not known. Therefore JC and C are actually unbounded arrays. The size of IC is known and it is equal to the size of IA.

The algorithm utilizes the sentinel element in the compressed sparse row data structures. After an initialization step, it starts with an iteration over the rows of he matrix A. The dummy element at the last position of the row pointer array IA, guarantees the invariant for starting and ending point of an row. This utilization of the sentinel spares two branches, which would have been needed for handling the case for the endpoint of a last row. Without a sentinel there would have been the need of a `IF` statement in line 11 and line 15 in the pseudo code.

Another major part of the algorithm is the use of the two arrays x and xb. `gustavson` processes the the data not as the classical $column - row$ method would imply. The algorithm iterates over the matrix A in a row-wise manner. Though, this order enables the computation of the result matrix to be done also row by row. Every element of the matrix $a(i,j)$ is multiplied with all of the nonzero elements in the row $B(j,\cdot)$. The results are stored in the array x. The array has the length of a row of the matrix C, so it can be seen as a temporary row (or accumulator). The product $a(i,j)b(j,k)$ is stored at the position k in x. The method uses xb to differentiate if there was already a valid entry on that position, i.e. an aggregation is needed, or if the present value is outdated, in this case the value must be overwritten. The following invariant holds:

$$xb[i] = k \Rightarrow x[i] \in C(k,\cdot) \tag{4.2}$$

For each element of x the array xb indicates to which row in the matrix C the element belongs to. At first the array x is filled with random values, so we need a simple initialization step, all values of xb are set to some invalid row number. e.g. -1. The rows of C are calculated in a strict ascending order, thus if $xb[i]$ is smaller than the currently processed row i one can be sure that the value $x[i]$ is no more relevant and it can be overwritten. The technique of using two interconnected arrays, with one storing information about the elements of the other, has proven as very beneficial. This is actually a quite generic algorithmic approach, which can be used also in other cases, e.g. similar idea is used for lazy initialization of arrays.

After executing all possible nonzero multiplications $a(i,j)b(j,k)$ for a specific row i, the result row $C(i,\cdot)$ is stored in the dense array x. Thus, a iteration over the values is needed, in order to copy them into the CSR structure of the matrix C. This is done with the loop at the 28-th line. Observe that the algorithm does not iterate over the whole dense array x, but only over the needed values. The indexes of the valid values in x are actually also the column indexes in the result matrix and the respective column indexes have been already stored in JC. With this trick `gustavson` does not use another temporary data structure for storing indexes of the calculated values in x.

Considering the pattern of iteration over the matrices A and B, it can be seen that `gustavson` is similar to the `join_aggregate` algorithm, nevertheless `gustavson` performs an *on the fly* aggregation, since it processes the result matrix row by row. Further, the algorithm does not employs any hash tables as in `outerP_hash`. By using the temporary data structure of the two dense arrays x and xb, the algorithm manages to execute the aggregation very efficiently.

The method has several advantages. There is no costly aggregation at the end. Hash tables are not used, so there is no overhead for hashing. In addition the

cache efficiency improves significantly due to the use of arrays and the chronological calculation of the matrix C, i.e. rows are being calculated in an ascending order. The result matrix is in a structured format. Moreover, all of the matrices are stored as CSR, hence the algorithm is very suitable for computing sequences of matrix multiplications. The result can be directly used as the input matrix for a consecutive multiplication.

On the other hand, CSR data structures have to be generated for the input matrices, which demands a beforehand sorting of their nonzero elements.

The complexity of the algorithm can be expressed as $\mathcal{O}(flops+nnz(A)+nnz(C)+n)$, which can be explained as follows. As already mentioned, the number of the needed arithmetical operations ($flops$) is initially unknown. We always iterate over all nonzero elements of A, even if not even one flop takes place, therefore there is the factor of $nnz(A)$. Every nonzero value of C has to be copied from the temporary array x into the CSR, hence also $nnz(C)$ is influencing the time. Lastly, in the initialization step we set all values of xb to -1 ($|xb| = n$) and the outer loop iterates over all n rows, thus n also plays role in the execution time. Usually this factor might be dominated by the first three, but by the computation of *HyperSparse* matrices this may not be the case. One of the main contributions in [4] is the developing of an algorithm suitable for *HyperSparse* matrices, which complexity does not depend on n.

The time complexity of `gustavson` depends on multiple factors. Furthermore, some of the variables in it are not known beforehand. The cost model gets even more complicated if we consider some constant factors as cache misses for example. This issue will be analyzed further in the next chapters.

The space complexity of the algorithm is on the other hand rather straight forward. The method demands only two temporary arrays x and xb as overhead, thus the needed additional space is $\mathcal{O}(2n)$.

4.2 Experiments

In this chapter we will present and evaluate empirical data for the performance of the algorithms. The description of the problem instances that were used for the evaluation can be seen in Table 3.2.

Figure 4.1 depicts the speedups for five different matrices, where the first algorithm `join_aggregate` is taken as a baseline and therefore with a speedup factor of 1. The precise execution times can be seen in table A.1. For the algorithms that need a hash table (`outerP_hash` and `outerP_hash2`), the implementation used the standard library `unordered_map` from the `tr1` namespace.

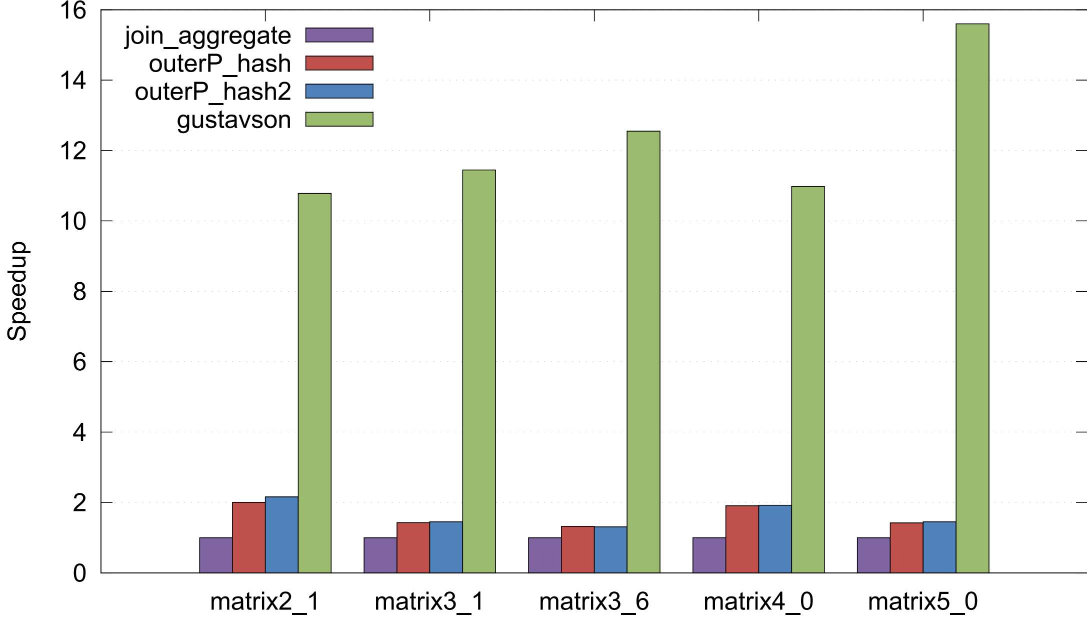

Figure 4.1: Performance of the sequential algorithms for SpGEMM

Clearly `gustavson` has remarkable advantage over the other three algorithms. It can be said that the experiments corroborate the conclusions that can be made from theoretically analyzing the complexities of the algorithms. `join_aggregate` has an additional aggregation step at the end, which is not presented in the other methods. The additional time needed for this aggregation predetermines `join_aggregate` as the slowest approach. The two algorithms that use hash table and on the fly aggregation achieve better execution times with all test matrices, where the speedup goes up to 2.16 with matrix2_1.

`outerP_hash` and `outerP_hash2` have also another crucial advantage. Very important for the applicability of an algorithm is not only its execution time but also its space complexity. In this regard the hash algorithms are significantly better than one big aggregation at the end. `join_aggregate` maintains a list of triples with size of the $multiplication_flops$. This list might be considerably larger than the nonzero elements in the result matrix. Thus, the algorithm may run out of memory although theoretically there is enough space for storing all of the nonzeros of the result matrix. On the other hand the hash algorithms do not have this issues as they have only $\mathcal{O}(1)$ space complexity. Hence, they scale much better than `join_aggregate` in regard to memory, they are in different complexity classes. Indeed, with `outerP_hash` we were able to compute the product of matrices with larger scale, for which `join_aggregate` have run out of memory.

Another phenomenon that can be observed is that although the hash algorithm are consistently better than the first method, their speedup varies strongly between different test matrices(from 2.16 by matrix2_1 to 1.31 by matrix3_6). This fact can be also explained with the size of the triple list in `join_aggregate`. This size is highly dependent on the specific structure of the matrix. The advantage of the hash algorithms will be more evident, when the triple list is larger than $nnz(C)$. For example apparently in the case with matrix2_1 there are many aggregation flops,

meaning many triples, that contribute to a single value in the result matrix, thus the triple list is quite large, so the aggregation step at the end will be costly. On the contrary, the multiplication of matrix3_6 does not need so much $addition_flops$, the ratio $\vartheta = \frac{|triples|}{nnz(C)}$ is smaller and the additional aggregation step is not so prominent, which explains the smaller speedup factors achieved by the hash algorithms. Observe that the ratio ϑ may also be equal to 1. In such case `outerP_hash` will have a minimal speedup. Noting this ratio is useful, since it can be used to bind the triples, the nonzero elements of the result and the needed additions in one equation: $(\vartheta - 1)|triples| = addition_flops$, where $\vartheta \geq 1$

There is practically no difference between the speedups of the two hash algorithms. Because the triples in matrix A are unsorted there is a random access over the rows of B. Therefore, there is no significant difference, if we use the CSR or a hash table. With matrix B in CSR we do not have the overhead of hashing, as when B in stored in a hash table the hash function will be called $nnz(A)$ times. Nevertheless, the experiment indicates that this overhead is negligible.

The remarkable winner in the executed experiments is the `gustavson` algorithm. It achieves significant speedups with all of the test matrices. The method is with more than a factor of 10 faster than the first algorithm, where the speedup goes up to 15.6 by $matrix5_0$. Figure 4.1 also indicates that the speedup of `gustavson` is tending to increase with larger problem instances. Therefore it is likely that the algorithm is not with constant factor, but rather with order of magnitude better than the other approaches. Thus, `gustavson` is not only the fastest algorithm in the test, but also the best one in regard to scalability.

There are multiple reasons for that notable performance. The algorithm uses on the fly aggregation. However, there are no hash tables and no overhead for hashing and managing keys. Further, the cache efficiency is greatly improved, for example the result matrix C is computed row by row, this leads to reduction of write cache misses. This is also evident from the chart in figure 4.2. The graphic depicts the cache misses of the different algorithms for the case of matrix5_0. One can recognize that there is a correlation between the number of cache misses and the execution times of the algorithms, respectively the speedups. The fastest algorithm causes at least cache misses. Furthermore, it is interesting that the cache misses of `gustavson` are about 16 times less than those of `join_aggregate`. This coincide almost exactly with the speedup coefficient. Nevertheless, we can also see that cache misses are not the only factor influencing the performance. Although `outerP_hash2` has less cache misses than `outerP_hash`, both algorithm have practically the same execution time. This indicates that performance is dependent on multiple parameters with cache misses being just one of them. Factors as instruction count, branch mispredictions and migrations play also crucial role in end running time. We will investigate this issue further in the chapter *Cache Efficiency*.

4.3 Conclusion

By analyzing the results of the experiments, we can conclude that `gustavson` is not only the fastest algorithm, moreover it outperforms significantly its competitors. By all test matrices the method was consistently better and it was from 5 to 11 times faster than the second best algorithm. `gustavson` also caused notably fewer cache

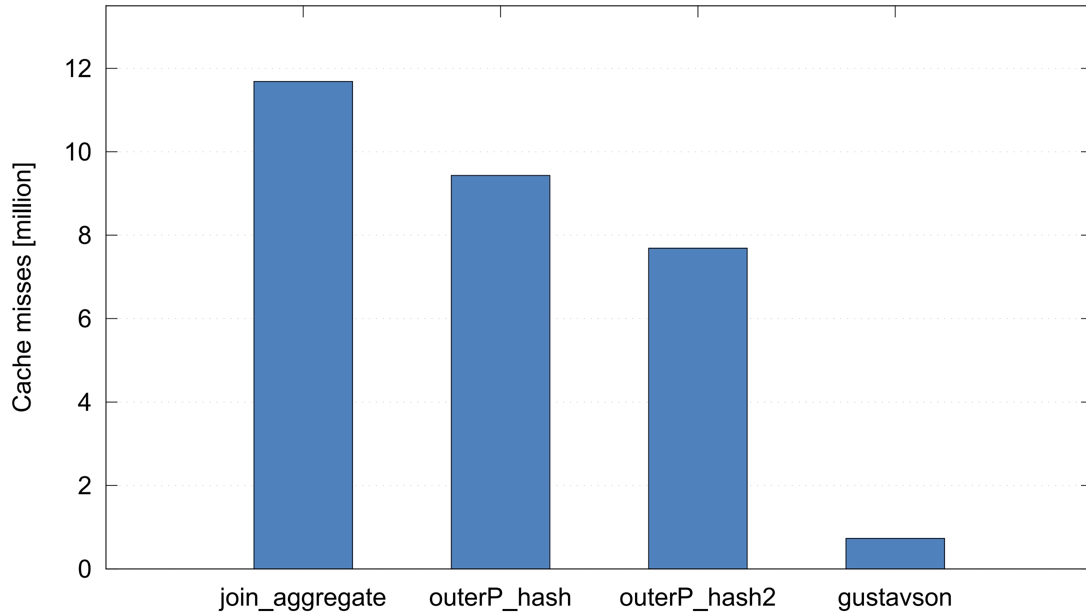

Figure 4.2: Cache misses of the different sequential algorithms for multiplying $matrix5_0$

misses. Further, the algorithm uses the practical CSR format for all of the matrices in the multiplication. Hence, the result matrix C is also stored in a structured manner, which makes the algorithm relevant in practice and also suitable for sequence of matrix multiplications.

Altogether `gustavson` has proven itself in the experiments as the *state of the art* algorithm for sparse matrix multiplication. Therefore, we will use it as a baseline in our further research. The method is apparently very efficient already, nevertheless in the next chapters we will propose new techniques for improvement of the performance. In addition to that, since `gustavson` is the best performing approach so far, it will be used as the sequential algorithmic core for a parallel SpGEMM algorithm.

5. The Density Estimator

Before starting with particular optimization proposals, we will introduce a density estimator as a component that will be used to calibrate the optimization techniques presented in the next chapters (similar idea was used by Zwick and Yuster [50]). This short chapter discusses the problem of estimating the density structure of the product matrix C.

5.1 Problem Description

The findings from the previous sections indicate that sparse matrix multiplication is a procedure with a rather complex cost model. The needed time for calculating the product of two matrices is dependent on multiple factors. Refer that the complexity of `gustavson` is $\mathcal{O}(flops + nnz(A) + nnz(C) + n)$. The dimensions and the number of nonzero elements are important, but most of the times the needed $flops$ are the dominant factor (except in the special cases of hypersparse matrices where the flops might be not dominating). Since the multiplication algorithms have only few arithmetical operations and permanent read/write accesses in the data structures, it is possible that the multiplication is bounded by the memory access times (memory bound) and not by the actual arithmetical operations (CPU bound). However, in both models the value of the *flops* is decisive for the cost model, since the number of flops correlates with the memory accesses.

Thus, for a estimation of the execution time, some knowledge about the number of flops is mandatory. Another crucial value is the size of the product matrix C. It would be beneficial if we know the number of nonzero elements in C before the multiplication. This information is important for size estimation and hence, memory allocation for the result. Without knowledge of the size, it is even not clear if it would be possible to perform the computation on the current machine (if there will be enough memory). Knowing the size of the output will also contribute to equal partitioning of the computation between the threads by parallelization. The problem is that neither the flops nor $nnz(C)$ are known beforehand. Further, there is no trivial way to determine the sparsity structure of the result matrix. This issue is illustrated in figure 5.1.

To formalize the density of a matrix this work uses the following definition.

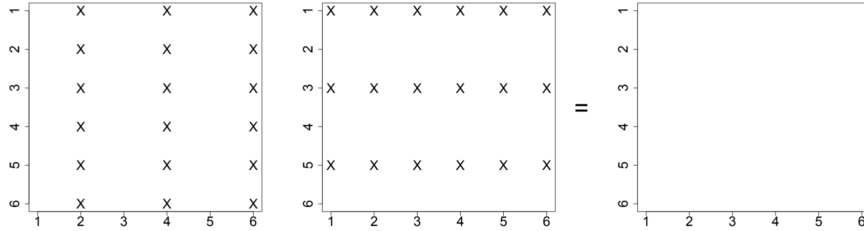

Figure 5.1: Connection between the densities of both input matrices and the output matrix

Definition 5.1. The density of matrix $A^{n \times n}$ is defined as $\xi(A) = \frac{nnz(A)}{n^2}$.

Although both input matrices in fig. 5.1 have density factor of 0.5, the patterns of their nonzero elements lead to completely empty result matrix. One has to consider that a matrix which is half full is an extremely dense sparse matrix. The sparse matrices that arise from real world problems are usually far more sparse. For example the density factor of the road network of Europe is $0.42 * 10^{-7}$.

On the other hand, very sparse matrices might multiply to a completely dense result. This kind of reaction can be observed in fig. 5.2. In general the phenomenon could be formalized through the following implication:

$$\exists j : nnz(A(\cdot, j)) = n \wedge nnz(B(j, \cdot)) = n \Rightarrow \xi(C) = 1 \tag{5.1}$$

In other words if a column $A(\cdot, j)$ and a row $B(j, \cdot)$ are full with values, then the whole output matrix will be full. In this case outer product between the two vectors will compute a complete dense matrix. Note how the input matrices are very sparse, density factor is only $\frac{1}{n}$, nevertheless the output is a dense matrix.

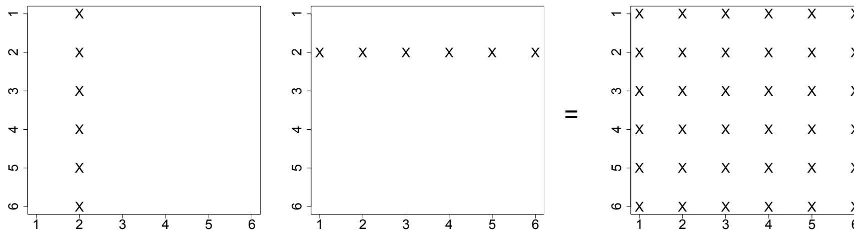

Figure 5.2: Connection between the densities of both input matrices and the output matrix, variant 2.

So far we have seen that that the size of the result is highly dependent on the patterns of both input matrices and in general case it is hard to make conclusions about the structure of the output. Parameters like the sparsity and the dimensions of the input play role in the execution time, however they can be also misleading. Matrices with huge dimensions might be multiplied relatively fast if their nonzeros follow convenient distribution. On the contrary much “smaller“ instances might cause much longer execution times, or even might be impossible to compute (due to lack of memory). This is the reason why it is hard to talk about *scale* when referring to a sparse matrix multiplication problem. The *scale* depends heavily on the needed

$flops$, however the only exact way to determine them is to execute the whole matrix multiplication.

There is a connection between $flops$ and the number of nonzeros in C. For a more precise analysis, we divided the arithmetical operations in two groups: multiplication flops ($mult_flops$) and addition flops (add_flops). For each nonzero element there must be at least one flop. From figure 5.1 can be observed that $nnz(C) = 0$ and accordingly the executed $flops$ would be 0 (we consider `gustavson` as the multiplication algorithm). It can be generalized that:

$$flops \geq nnz(C) \tag{5.2}$$

Further, it is interesting that an element of C can demand just one multiplication but also n multiplications and n additions. Thus, some nonzeros of the result matrix are much more costly to compute than others. Figure 5.3 gives a good example for that case. Observe the discrepancy between $c(1,1)$ and $c(1,6)$.

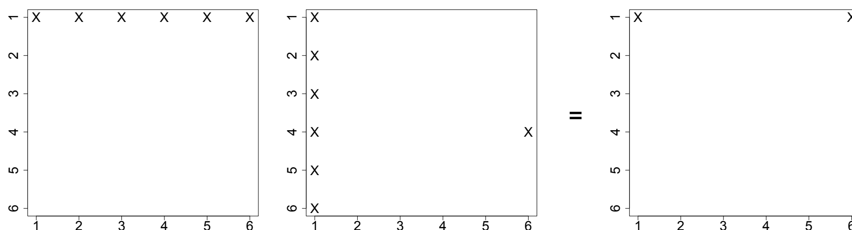

Figure 5.3: Number of flops contributing to an element

If one wants to further formalize the flop count, the multiplication flops can be described with the following equation:

$$mult_flops = \sum_{i=1}^{n} \sum\nolimits_{a(i,j) \in nnz(A(i,\cdot))} nnz(B(j,\cdot)) \tag{5.3}$$

The multiplication flops for the whole computations are the sum of the multiplication flops of each row, where the elements in the row and the nonzeros in the respective rows in B are determining how many multiplication operations will take place for that row (refer to the approach in `gustavson`). The multiplication operations are upper limit for the nonzero elements in C. As already seen in the Algorithm 1 for example, every multiplication flop creates a new triple, there will be a nonzero value at these coordinates in the result matrix. However, there could be more triples contributing to the value at the specific coordinates (Figure 5.3). At most there can be n different triples, i.e. n $mult_flops$. The addition flops on the other hand, sum the triples in order to calculate the final nonzero value. Each addition aggregates the result of two multiplication flops. This means, if there are x $mult_flops$ (triples) contributing to a single nonzero vale in C, there are also $x-1$ add_flops for summing all of the triples and calculating this value. Hence, if we assume that the probability of triples summing to zero is negligible, the following dependency can be derived:

$$mult_flops - add_flops = nnz(C) \tag{5.4}$$

The last equation illustrates the connection between the flop count and the density of the output matrix. One can conclude that one way for evaluating the density of C might be to evaluate the needed flops.

5.2 The Estimator

In the previous section was shown that estimation of the $flops$ and nonzeros of the result matrix is a major problem of sparse matrix multiplication. In the next chapters we will see that this information is needed for engineering some optimization techniques. Therefore, a *flops estimator* would be valuable for the further research.

Motivated from equation 5.3, Algorithm 4 defines the pseudo code of a *flops estimator*. Similar idea was used by Zwick and Yuster [50], however they did not compute the flops per row as we do it here.

Algorithm 4 Pseudo code of the *flops estimator*

Input: matrixA: CSR, matrixB: CSR
Output: flops:[1...n] of Number

```
1: function GET_MULT_FLOPS(matrixA, matrixB)
2:     nnzB:[1...n]
3:     flops:[1...n]
4:     for i ∈ [1...n] do                          ▷ Calculate nnz per row
5:         nnzB[i] ← IB[i + 1] − IB[i]
6:     end for
7:     for i ∈ [1...n] do
8:         startRow ← IA[i]
9:         endRow ← IA[i+1]
10:        for j ∈ [startRow...endRow] do
11:            flops[i] ← flops[i] + nnzB[JA[j]]
12:        end for
13:    end for
14:    return flops
15: end function
```

The algorithm computes the exact number of the needed $mult_flops$ for each row of the result matrix. The CSR structure contains implicit the information about the number of nonzeros per row. The *flops estimator* uses this feature of the CSR to build the array $nnzB$. After that the algorithm counts the number of multiplication operations that `gustavson` would execute per row.

The method computes only the multiplication flops, the addition flops are not regarded. Therefore, the proposed method is called an *estimator* and not an *evaluator*. The only exact way to know all of the flops, inclusive the additions, is to actually perform the whole multiplication. However, the running time of an estimator should be relatively low, so it is feasible to use it for optimizations. Otherwise, one can just execute the multiplication instead. Furthermore, we assume that the multiplication flops will be most likely the dominating factor by sparse matrices.

The complexity of the estimator is $\mathcal{O}(n + nnz(A))$, because it firstly computes the nonzeros per row of B and then there is a linear scan over the elements of A.

From flops to density

In order to make conclusion about the density of the output, we consider equation 5.4. It implies that:

$$mult_flops \geq nnz(C) \tag{5.5}$$

Thus, with the *flops estimator* we have gained an upper bound for the nonzeros of the output. In this way the *flops estimator* can be also used for density estimation (see also [50]).

Furthermore, by very sparse matrices the number of addition flops might be negligible. For example by multiplication of two matrices with dimensions $(20 * 10^6) \times (20 * 10^6)$ and with only 3 elements per row, additions would take place with very low probability, especially if the elements follow a random distribution. For such kind of matrices the assumption could be made that $add_flops \approx 0$, which leads to $mult_flops \approx nnz(C)$.

Other approaches

The presented algorithm computes deterministically the exact number of multiplication flops for each row. One proposition for improvement might be to sacrifice some precision and to use only a subset of the nonzero elements of A and from that to make a conclusion about the whole sparsity structure. However, it is questionable if this would lead to significant improvement. The deterministic method has rather low complexity already, especially when compared to the time for the multiplication.

One can use randomized data structures to estimate also the addition flops. For example a *Bloom Filter* might be used to check if there was already a value at specific coordinates. In such way, also estimation for the add_flops can be given. The execution time of the estimator would of course increase (due to additional data structures, hashing et cetera). The exacter the estimation, the closer is the estimator to the actual multiplication. It is a trade off between precision and time. There is a threshold where the execution time would be too long and although more precise, a slow estimator would not be beneficial.

An alternative approach would be to estimate the density of C only by regarding $\xi(A)$ and $\xi(B)$. If we make the assumption that the elements of the matrices follow a random distribution, the density of the result matrix can be expressed only with the help of statistics. We refer to [32] for more details.

6. Cache Efficiency

> "A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away."
> — Antoine de Saint-Exupéry

So far, the sparse accumulator-based methods like `gustavson` have shown the best performance in comparison to other approaches. Nevertheless, there is still potential for improvement. This chapter analyzes the cache efficiency of the `gustavson` algorithm and proposes new techniques for further reduction of the cache misses. At the end we will have a proven method for enhancement of the cache efficiency that can be used as a general strategy in algorithm design, even for algorithms not connected with matrix multiplication.

6.1 Optimization Idea

Our optimization idea is rooted in the two dense arrays $\boldsymbol{x}$ and $\boldsymbol{xb}$ that are used by `gustavson` for temporary storing of matrix elements. The arrays are a cache-efficient data structure already. However, in some cases they might cause also significant amount of cache misses.

The two arrays scale with the dimensions of the matrix and have a length of n. If we multiply matrices with relatively small dimensions, so that the arrays can be stored in the cache, the use of an array as a data structure is optimal. On the other hand, if we compute the product of matrices with very large dimensions the arrays will not fit in the cache entirely. For example, by the multiplication of two matrices with $n = 20 * 10^6$ and double precision for the result, the memory demand for x and xb would be $(8 + 4)Byte * 20 * 10^6 = 240$MB, which is around 30 times more than the Level-3 cache of a state of the art processor. Furthermore, by the sparse matrix multiplication there might be rather random access on the two temporary arrays. Thus, there will be a significant amount of cache misses, where at the worst case each write in the x and xb will cause a cache miss. Hence, only the writing in the temporary arrays would lead to $mult_flops$ cache misses.

Having said that, it is highly likely that the arrays will be not filled completely. They store the temporary values of an row in the result matrix, so if the output is a sparse matrix, its rows will be also sparse. Thus, `gustavson` might use too much redundant memory for x and xb. Our optimization idea is to replace the arrays with a smaller hash table, that will fit entirely in the cache. In this way all of the cache misses caused by read/write accesses will be optimized.

In this optimization technique we use the core of the `gustavson` algorithm and replace the dense arrays with a hash table. The new algorithm is called `sparse_cemm`, i.e. sparse cache-efficient matrix multiplication.

With a hash table the cache misses will be reduced, but in this case there is an overhead of additional instructions, hashing, collisions et cetera. The arrays are low level data structure. Exchanging them with a hash tables, adds significantly more instructions for each read and write. Furthermore, there has been substantial development in the hardware. CPU prefetchers has been optimized to work with arrays. The CPU can recognize patterns in the reads and writes to the main memory, so only the relevant sections are loaded to the cache. Pipelining and look-ahead techniques might also reduce the costs of cache misses. Cache technology is also improving constantly, trying to intelligently recognize needed memory blocks. Thus, although theoretically reasonable, there is a need for an empirical evaluation that would investigate, if the approach can indeed lead to a speedup.

6.2 Initial Evaluation of the Concept

This section gives details about the initial implementation of the optimization strategy. We used hash tables from different libraries to replace the two dense arrays x and xb. Intel PCM was used for the performance profiling of the algorithms.

There are a couple of challenges when implementing the optimization technique. The hash table has to remain small, so that it does not overflow out of the cache. We have seen empirically that a hash table that would mostly fit in the L2 cache is preferable than bigger one that would fit in L3. This is expected, since in this way the access times will be even smaller.

Furthermore, the hash table must not grow too often because each growing step includes rehashing of all keys and copying all of the elements to a new hash table. This process will be significant overhead over the simple arrays. The hash table has to be initialized with an appropriate size, so some growing steps are spared. As already mentioned, the whole optimization idea depends on the data structure fitting into the cache. Thus, we have to periodically empty it. This clearing process also adds to the running time. It arises also the question, when would it be optimal to empty the hash table. In this section the emptying was done naively by destruction and new initialization. If the hash table is cleared too often, the clearing process might raise the overhead. On the other hand too rare clearing, will lead to hash table with bigger fill ratio and thus more collisions or it might lead to growing.

Regarding the implementation, the column index of the result element $c(i, j)$ is used as the key in the hash table. This index coincide with the index of $c(i, j)$ in the temporary arrays x and xb. In this first implementation the value in the hash table is the pair $(c(i, j), i)$. In this way the arrays are replaced by the hash table and the index of the array becomes its key.

In order to prevent growing of the hash table, the maximum load factor was explicitly set to 1, so we have more control over when a rehash would take place. After each calculated row of the result matrix the table includes also the elements from that row. Then the algorithm checks if the number of elements in the data structure is over a certain threshold. If so, the hash table is emptied, if not, the calculation of the next row is continued without emptying. In summery, the method tries to keep the hash table at a static size and not exceed a certain fill level.

The new data structure introduces two new parameters that influence the running time. First, there is the size of the hash table, as already said, it has to fit in the lower levels of the cache. Second, there is the threshold ratio at which a clearing would be executed. The experiments suggested that a size of around 2^{14} elements is optimal. This is attributed to the fact that the data structure would then need a minimum of $(8+4+4)Byte * 2^{14} = 262144Byte$ (8 Bytes for the result, 4 Bytes for the row index and 4 Bytes for the key)[1]. The Level-2 cache of the test system has $256KB$, thus with this setting there are good chances that large portion of the data structure will be even in the L2 cache. For the second parameter, the fill ratio, the experiments have pointed that a value around 0.3 is optimal. Higher fill ratio causes the overhead of collisions happening too often.

The optimization approach was tested with a couple of hash table implementations with the described setting - table size 16834 elements and maximum fill ratio of 0.3. As input we used $matrix6_10$, described in the Appendix table 3.2. The matrix was multiplied by itself. The goal is to improve the end running time by reducing the cache misses. Therefore, in the experiment we used a matrix with randomly distributed elements. This would guarantee a random access over the temporary arrays x and xb. Furthermore, n should be sufficiently large, so the arrays will not fit into the cache. Our baseline is the classical `gustavson` algorithm, implemented with dense arrays. The execution times for the different hash tables can be seen in figure 6.1.

The experiment reveals that there is a remarkable deviation between the performance of the different hash tables. `std::map` from the C++ Standard Template Library is the slowest implementation. This makes sense, since the data structure is implemented as a tree. Thus, accessing and updating operation will be in $\mathcal{O}(\log n)$ instead the usual *expected* $\mathcal{O}(1)$. The advantage of the tree structure is by growing, however in the particular case often growing steps are not desirable. Interestingly, there is a considerable difference between the `unordered_map` from the standard library and the one from the `tr1` namespace. The one from `tr1` is about 23% better in regard to execution time.

In order not to restrict the evaluation only to the standard library, the test included also two hash tables from an external library. `google::dense_hash_map` achieved the best running time among the hash tables, noticeably better than the others. This hash table has been optimized for fast access time, but has relatively high memory overhead (78%). On the other hand `google::sparse_hash_map` is optimized for low memory consumption, it takes only about 4 additional bits for an item. The trade-off between space efficiency and time efficiency can be observed by the difference

[1]Depending on the implementation the hash table may demand more storage, for pointers, additional structures etc.

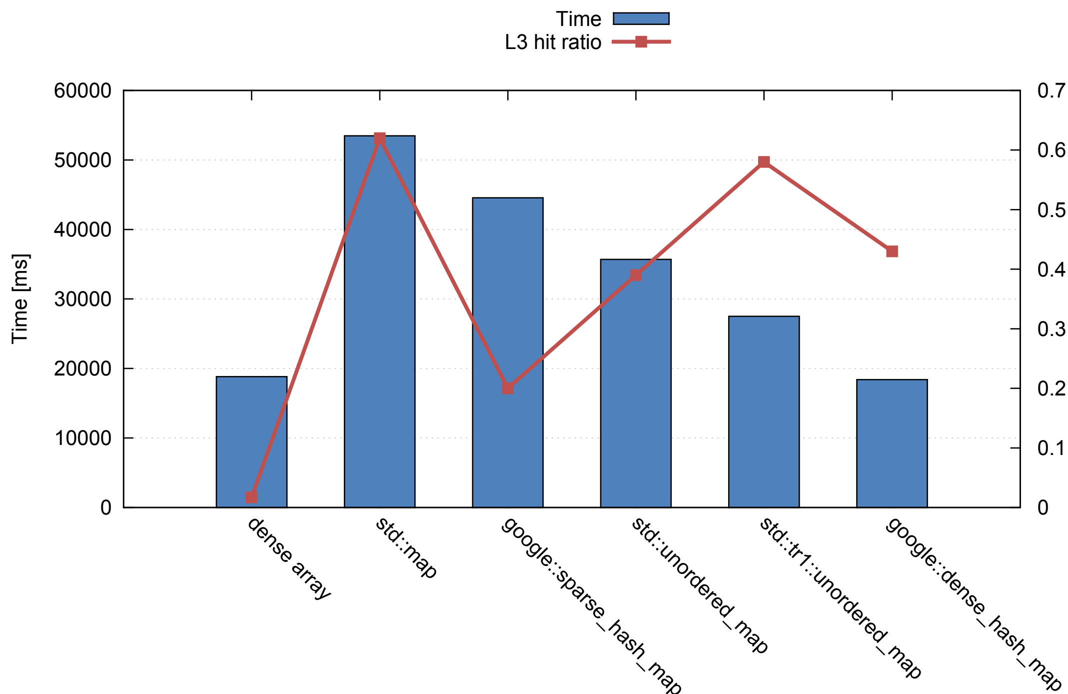

Figure 6.1: Execution time of `gustavson` with different hash tables; L3 cache hit ratios; matrix with random distributed elements.

in the performance of the two hash tables. The sparse variant, with lower memory footprint, is more than two times slower than the dense version.

Although some of the implementations with hash tables came close to the time of the original `gustavson`, none of them was able to achieve speedup against the plain arrays. To investigate the reasons, we have also depicted the hit ratio of the Level-3 cache in Figure 6.1. A cache miss on this level will be most expensive since it will cause a data retrieval from the much slower main memory. Compliant with the theory, all of the implementations with hash tables lead to a great improvement of the hit ratio. The dense array implementation has hit ratio close to 0. At first sight there is also rather illogical dependency between the time and the cache hit ratio. Although `map` has the best hit ratio it exhibits actually the slowest running time. The explanation for this phenomenon is that the cache hit ratio is misleading as a measure for the absolute performance. One should consider that the absolute number of the cache misses are actually more relevant than the ratio. It would be not beneficial if the hit ratio has improved at the cost of introducing huge amount of new cache references. The amount of cache misses might actually rise and in the same time the hit ratio can increase, if we add also enough cache references in the algorithm.

In order to be more precise, one also has to differentiate between the cache misses on the different levels. Figure 6.2 gives information about the absolute number of L2 and L3 cache misses.

The dense array has almost the same amount of L2 and L3 cache misses. This is caused by the fact that there is a random access on the array and it is much larger than the cumulative cache size. Thus, a Level-2 cache miss will cause also a Level-3

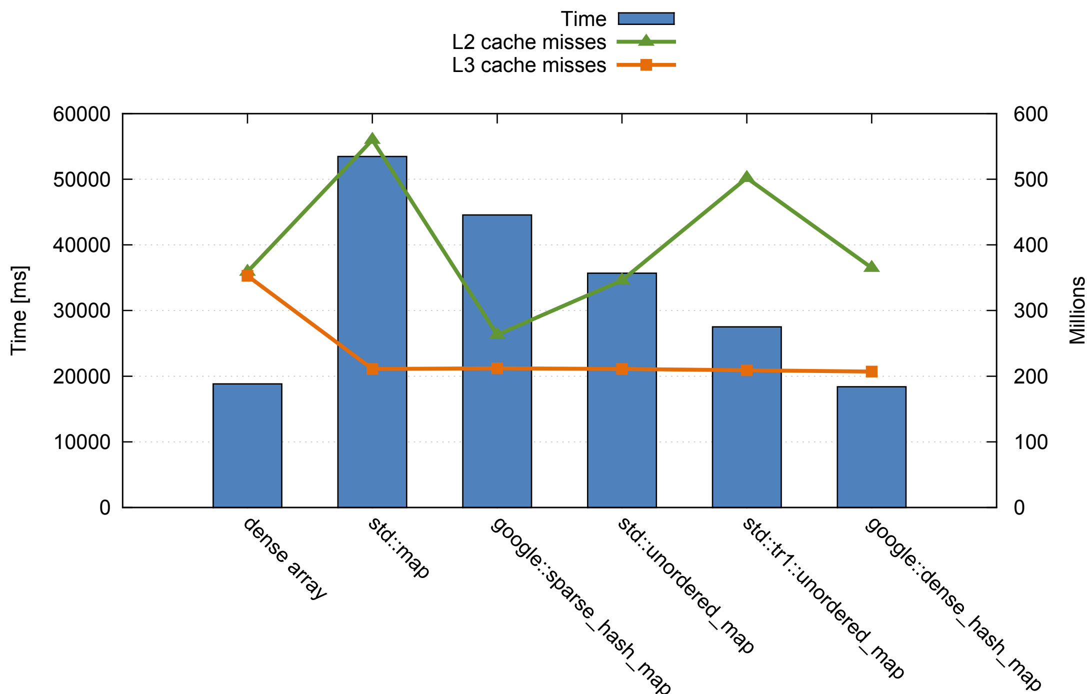

Figure 6.2: L2 and L3 cache misses for the different hash tables.

cache miss with a high probability. Further, one can observe that all hash tables led to reduction of the L3 cache misses. Moreover, all of the implementations show the exact same amount of L3 cache misses. This is due to the small size of the hash table, it is stored entirely in the cache, i. e. all levels combined, with all variants. Access to the table causes no main memory retrieval. On the other hand, there is a quite large variance between the L2 cache misses. The size of the hash table was intentionally chosen small so that theoretically big portion of the data structure may fit even into the L2 cache. The experiment reveals that there is a considerable difference between the space overheads. As already mentioned `sparse_hash_map` is extremely space efficient (only 4 additional bits per item), so it makes sense that this hash table leads to less L2 cache misses, because more portions of it fit in L2. On the contrary, `dense_hash_map` uses nearly two times the space actually needed for storing the items. Thus, a high percentage of the hash table will not fit in the Level-2 cache and so it is understandable that it has 40% more L2 cache misses than the sparse version.

From the chart one can also see the discrepancy between cache hit ratio and absolute number of cache misses. `std::map` has shown 60 times improvement in the L3 cache hit ratio. Nevertheless, this data structure has actually only 20% less cache misses, if the absolute values are regarded.

Although the absolute values give more insights in the reasons for the different running times, there is still no clear correlation between the cache efficiency and the time. For example `dense_hash_map` causes more L2 cache misses than the sparse version `sparse_hash_map`, nevertheless it is more than two times faster. Thus, there are also another factors that influence the efficiency of the algorithms.

In figure 6.3 are depicted the number of retired instructions for the different implementations. This is another key indicator, which has great effect on the execution times. Naturally, the plain array leads to an algorithm with least instructions. The hash tables inevitably add some overhead. The data depicted in the graphic indicate that this overhead is crucial for the running time. `dense_hash_map` introduces at least additional instructions when compared with the dense array. This data structure leads also to the fastest running time among the hash tables. On the other hand, `sparse_hash_map` has the biggest overhead, factor of 14 more instructions than the version with dense arrays. Accordingly, the running time is also more than 2 times slower.

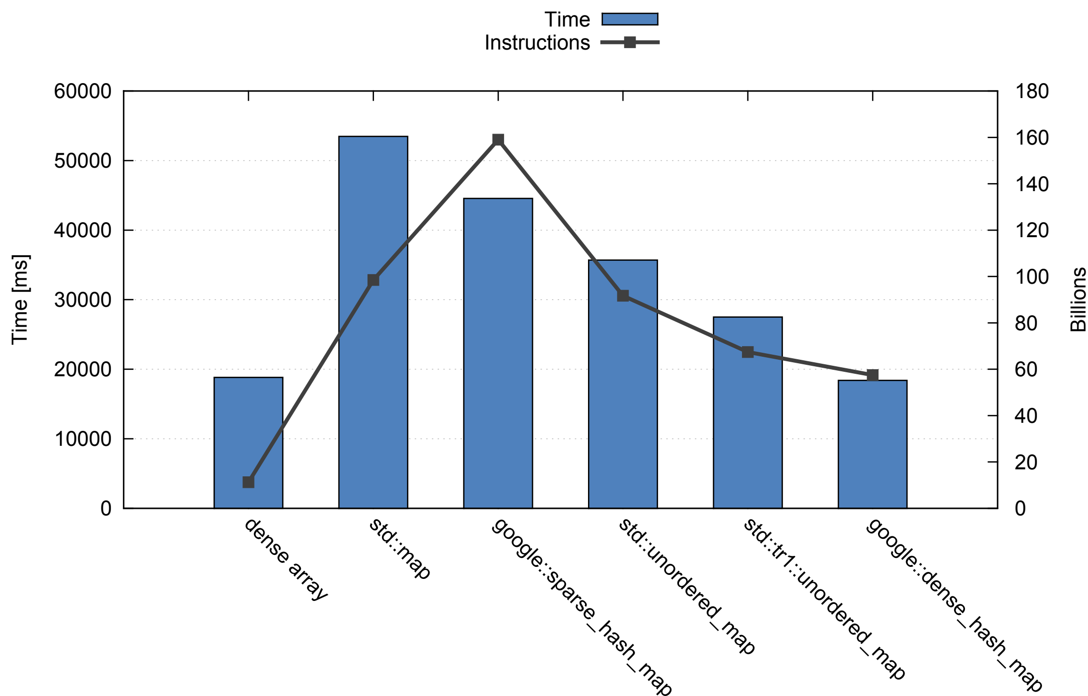

Figure 6.3: Number of retired instructions.

Overall, one can conclude that the time efficiency is influenced by multiple factors. The number of cache misses and the cache efficiency of the algorithms alone are not sufficient for making conclusions about the running times. The number of instructions should be also regarded. Optimization technique that will reduce the number of cache misses, will succeed to improve the final execution time only if it does not introduce too much instruction overhead. From the previously tested hash tables, only `dense_hash_map` came close to the time of the arrays. It reduced the number of cache misses, mainly the expensive L3 cache misses. Further, it succeeded no to add to much additional instructions, by doing that.

The first experiments show that none from the tested hash tables leads to a net improvement of the algorithm. Nevertheless, there is some potential for our optimization approach. The instructional overhead of the tested data structures was to high to beat the arrays. If we can construct a lean hash table with less instructions, it might be possible to observe a speedup.

6.3 The Custom Hash Table

In the previous section we have seen that the cache efficiency and the instructional overhead are two major variables that have to be regarded by optimization strategies. This section will introduce a custom minimal hash table, with the goal to decrease the instruction count and still implement the cache-optimization idea.

A hash table consists mainly of tree components:

- hash function
- data structure for storing the elements
- collision avoidance strategy

Each component can be changed, thus there are many different combinations possible. We have tested multiple variants but the best times were achieved by a hash table called `minimal_hash_map3` ("three" because there were a couple of optimizing iterations). The configuration of the data structure is as follows.

The hash function

The hash function is crucial for the performance of a hash table. Firstly, the function has to maintain sufficient randomness, i.e. two different keys will be hashed to the same value with very low probability. This will lead to spreading the hashed keys over the whole data structure and will reduce the number of collisions. Each collision causes additional overhead for searching the right element. Thus, a function with lower collision probability is desirable. The second factor is the instruction count that the function needs to calculate a particular hash value. In general the function will be called by every read and write operation, hence its execution time is also of great importance.

Experiments with different hash functions have been conducted (see below). Surprisingly, a simple concept of tabulation hashing ([36] [39]) led to very promising results. With classical tabulation hashing the key is tokenized in equally sized *chunks*. Additionally there are lookup tables filled with random values. If the key can be tokenized in say n words, then there must be n different lookup tables. Each lookup table contains $2^{|chunk|}$ random values, so each chunk can be used as an index in the according lookup table. The hash value is constructed by XOR operation between the random values from the lookup tables which are pointed by the chunks, their bit-pattern is interpreted as an index in the lookup table. In our application the keys were 32 bit integers, and the size of a chunk was $1Byte$. Thus, for tabulation hashing we need 4 lookup tables with 256 random values each. If the lookup table are named T_i, then the a tabulation hash function would be defined as:

$$H_1(x_1x_2x_3x_4) = T_1[x_1] \oplus T_2[x_2] \oplus T_3[x_3] \oplus T_4[x_4] \tag{6.1}$$

The different chunks are retrieved through bit shift and logical AND operations.

To reduce the number of instructions, a variance of the tabulation hashing was implemented that uses only one lookup instead of four (Peter Sanders and Kurt Mehlhorn [36]). The *key* in our use case is the column index of a matrix element, so *key* is between 1 and n. In the experiments in this thesis n is usually less than 2^{25}. In this setting only one lookup table with 2048 random elements is used. We use the first (most significant) 11 Bit as an index in the lookup table. The last 14 Bits are xor-ed to the random value from the lookup table. Hence, the hash function has only one lookup and three bit operations. If the size of the lookup table is $|key| - m$, for some $0 \leq m < |key|$, then the hash function can be defined as:

$$H_2(x) = T[x \gg m] \oplus (x \wedge (2^m - 1)) \tag{6.2}$$

The parameter m is important for the performance of the function because it determines the size of the lookup table. The lookup table should be small enough so it can fit into the lower levels of the cache, preferably L1 cache. Furthermore, m depends on the specific key domain.

In figure 6.4 are depicted the times for adding $45 * 10^6$ elements in the custom hash table when different hash functions are used. All other components of the data structure were the same with all experiments, so the only differences are the functions, i. e. collisions and execution time are causing the deviation in the time. Detailed description of the has functions Jenkins, cityHash and FNV can be seen respectively in [31], [40] and [16]. *Table hash 1* is the classical tabulation hashing and *Table hash 2* stands for the second version with less instructions. In most of the multiplications the most significant byte of the key will be zero, therefore we spare one lookup in the implementation of the classical variant, only the last three bytes are used for lookup. This one lookup and one XOR less led to 4% faster execution.

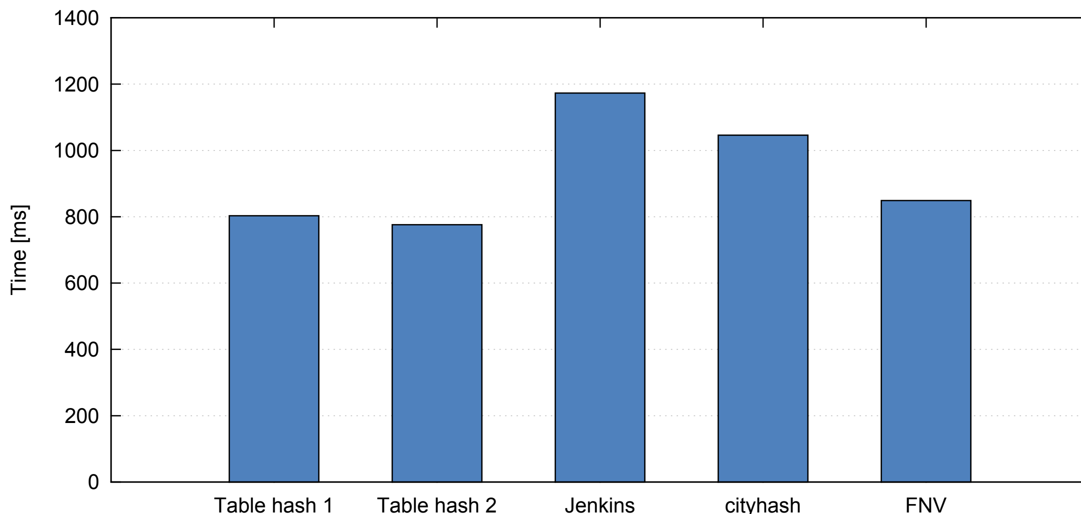

Figure 6.4: Performance of different hash functions.

The simple tabulation hashing outperforms *state of the art* hashing functions. At least in this experiment. One has to consider that multiple experiments in different

kind of settings are needed to evaluate fully a hash function. This extensive evaluation is out of the scope of the thesis. In our tests tabulation hashing achieved the best running times, therefore it was used in the custom hash table.

The data structure

As our goal is to construct a hash table as minimalistic as possible, we have chosen a plain array as the data structure for storing the items. There are no additional pointers and complications. The array stores triples containing the *key* and the values for x and xb. In order to reduce the number of instructions, the hash table has static size, thus it cannot grow. In this way instructions related to the growing logic are not present at all. The non-dynamical size of the data structure makes it faster, however it is also a significant restriction. The next sections will get back to this problem.

Collision avoidance

For collision avoidance linear probing was used. Quadratic probing was also implemented and tested but it did not lead to an improvement.

In the first experiments the data structure was cleared through new initialization, i.e. deleting the old container and allocating a new one. Further, the hash table does not have to support delete operations, so there are no gaps between the items caused by deletion. Thus, when there is a collision we can be sure that we have found the right place by finding the key or by hitting an empty cell. This helped also for sparing some instructions that otherwise would have been needed.

Initially, the size of the hash table was set to 2^{14}. The fact that it is a power of two is not a coincidence. The random values in the lookup tables are between 0 and $2^{14}-1$, in this way the hash value for every key is already in the bounds of the array used as hash table. Thus, one modulo operation is spared, because the hash can be used directly as an index.

In general the strategy in the implementation of `minimal_hash_map3` was to add as few instructions as possible. Only than `sparse_cemm` has a chance to be faster than the array implementation. Every code line less is important for the performance of the algorithm.

Table 6.1 illustrates the performance of `sparse_cemm` when `minimal_hash_map3` is used instead of the original version of `gustavson` with dense arrays. The experiment is the same as with the previously tested hash tables, so the values can be compared.

	dense array	minimal_hash_map3
L3 hit ratio	0.02	0.38
L3 cache misses [10^6]	353	205
L2 cache misses [10^6]	359	335
Instructions [10^9]	11.3	27.8
Time [ms]	18820	16740

Table 6.1: Performance of `minimal_hash_map3` and comparison with the dense arrays.

Our minimalistic implementation of a hash table succeeded to improve the running time when compared with the dense array baseline. The custom data structure managed to reduce the cache misses (L3 cache misses have been reduced by 42%) and in the same time not to add too large instructional overhead. `minimal_hash_map3` causes two times less instructions than `google::dense_hash_map` - the best from the previously tested hash tables in regard to instruction count. Overall, this leads to 11% net speedup of the multiplication when compared with the dense arrays.

Although there is an improvement in the execution time, one could expect that this has to be more evident, after all the L3 cache misses have been reduced almost two times. Interestingly, the large reduction of cache misses does not cause as large improvement in the time. One reason might be recent advancement in hardware technology. Pipelining, CPU instruction prefetchers and look ahead technologies might be reducing the cost of a cache miss on modern machines. Instead of stall waiting for data retrieval from main memory the processor could continue with other instructions. Also the time gap between cache and main memory might be closing with new hardware. Despite all this, the cache optimization strategy led to improvement.

Non Uniform Memory Access (*NUMA*)

Often there have been significant differences in the running time for the same experiment. For example, for multiplying the same matrices with the exact same algorithm there has been a deviation of more than 25% in some cases. There are multiple factors that influence the execution time of a program on a multi-core architecture. At first, our presumption was that this behavior is cased by deviation of the cache misses. On the system are running multiple processes, so some of the other programs may contaminate the shared cache with their own data. This would lead to increasing amount of cache misses of the process being tested. However, to our surprise, the number of cache misses de facto did not change. Our assumption is that *NUMA* effects are influencing the running time. In order to examine further, we pinned the algorithm thread on the different cores by using:

```
numactl --physcpubind=#core_id
```

The variance of the cache misses and the according running times are depicted in Figure 6.5. The number of cache misses stays practically constant, there is only

marginal deviation. However, there is a difference of 30% in the running times for the different cores. Considering the execution time, we can clearly observe that the cores are packed by groups of four. This fact supports also the hypothesis that *NUMA* is causing the deviation. Moreover, cores 4-7 and 12-15 are in one node (socket), respectively cores 0-3 and 8-11 are belonging to the other node. Further experiments have been made with explicitly binding the memory allocation to a specific node (`-membind=#node_id`). The deviation in the execution confirmed that *NUMA* is the reason for the time variance.

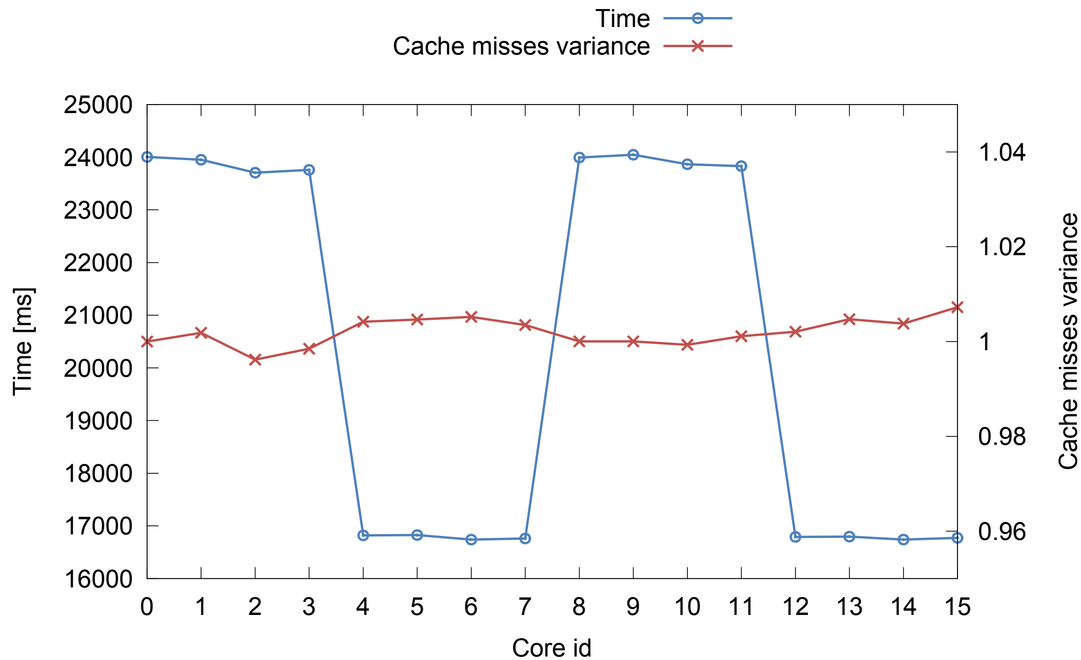

Figure 6.5: Difference of the execution time by core, deviation of the cache misses.

Hence, *NUMA* effects can cause noticeable difference. For correct evaluation of algorithms one has to carefully bind the tested program to a specific *NUMA* node. In our test setting the internal scheduler of the operating system did not succeed to automatically optimize memory allocation on its own.

Migrations

On a multi-core architecture the operating system might force migration of a thread between the different cores. This is done by the internal scheduler for load balancing purposes. The presented cache optimization technique is even more worthy if a migration takes places. A migration of the thread to different node will cause increase of the cache misses, since the cache at the new node is fresh and is not containing relevant data. Further, each write to the arrays x and xb will lead not only to a cache miss, but also the data has to be retrieved from remote memory module. Thus, *NUMA* effects will amplify the costs of a cache miss. As already seen these effects can cause severe time lags. On the other hand the hash table fits entirely in the cache. At first accessing values from it would cause also a cache miss, but gradually it will be loaded into the new local cache. This would lead to even larger speedup for `sparse_cemm`. As a conclusion, reducing the size of the temporary data structure has

positive influence on the flexibility and *NUMA* awareness of the algorithm. In real world environment, where thread migrations are possible, the cache optimization technique will be even more valuable. In the previous experiments the execution thread was pinned to a specific core, so migrations and *NUMA* were excluded. When during the experiment a migration between nodes takes place, the speedup of the hash table optimization rises to 20%.

6.4 Fine-tuning of sparse_cemm

The previous sections we have presented a cache optimization technique that led to 11% speedup when using a specially tuned hash table. It has been shown that the approach has potential. In this part we will further refine the algorithm and present its pseudocode.

Initially, the hash table has been used for processing multiple rows of the result matrix and after certain fill ratio is exceeded, it was emptied through new initialization. It is actually beneficial to empty the data structure after each processed row. This strategy leads to several advantages. Firstly, the hash table can be of smaller size, so it is more likely to fit in the lower levels of the cache. The smaller size should reduce the Level-2 cache misses when compared with the previous implementation. Secondly, if the temporary data structure is emptied after each processed row, then there is no need for storing the xb index. xb was used to differentiate if a certain value belongs to the currently processed row or not. Accordingly one should update (aggregation) or overwrite the value. By clearing the hash table after each row, we create the invariant that each value (different from zero) belongs to the currently processed row. The pseudocode of `sparse_cemm` is presented in Algorithm 5.

The size of the hash table is a tuning parameter, which also depends on the specific problem instance. Smaller size is better for the cache efficiency (less L2 and L1 cache misses). In the same time the data structure must have enough free slots so each row of the result matrix can be stored in it, without growing to be needed. Further, there has to be always some overhead of free slots, so collisions do not take place too often.

The algorithm uses additional stack for storing the positions of the stored elements in the hash table. After all values from the current row of C are calculated and stored in the temporary data structure, we can use the indexes from the stack to retrieve the values and copy them to the final CSR structure of the result matrix. Storing the indexes brings the advantage that there is one call less for the hashing function. In the implementation from the previous section the key (i.e. the column index) was hashed two times: first for storing a value into the hash table and then second time for retrieving it and copying it. In the here presented variant the key is hashed only once, namely in line 16. After that we use only the stored indexes to iterate over the nonzero elements of the hash table, in the same time the data structure is cleared by zeroing the $(key, value)$ tuple at that position.

The new algorithm is more cache-efficient than the original `gustavson`. Both algorithms have the same time complexity, since fewer cache misses is low level optimization that can be seen as constant factor in the complexity term. The space complexity on the other hand improves to $\mathcal{O}(max_{1 \leq i \leq n}(nnz(C(i, \cdot))))$.

Algorithm 5 `sparse_cemm` algorithm.

Input: matrixA: CSR, matrixB: CSR

Output: matrixC: CSR

```
 1: function SPARSE_CEMM(matrixA, matrixB)
 2:     initLookupTables()
 3:     x ← minimal_hash_map3(t)                          ▷ t being the size
 4:     indexStack:Stack<int>
 5:     ip ← 0
 6:     for i ∈ [1 ... IA.size − 1] do
 7:         IC[i] ← ip
 8:         startRowA ← IA[i]
 9:         endRowA ← IA[i + 1] − 1
10:         for jp ∈ [startRowA ... endRowA] do
11:             j ← JA[jp]
12:             startRowB ← IB[j]
13:             endRowB ← IB[j + 1] − 1
14:             for kp ∈ [startRowB ... endRowB] do
15:                 k ← JB[kp]
16:                 index ← x.getIndex(k)
17:                 if x[index] ≠ 0 then
18:                     x[index] ← x[index] + A[jp] ∗ B[kp]
19:                 else
20:                     x[index] ← A[jp] ∗ B[kp]
21:                     ip++
22:                     indexStack.push(index)
23:                 end if
24:             end for
25:         end for
26:         while indexStack not empty do
27:             z ← indexStack.pop()
28:             (key, value) ← x.popAt(z)                  ▷ get and remove
29:             JC.pushBack(key)
30:             C.pushBack(value)
31:         end while
32:     end for
33:     IC[IA.size] ← ip                              ▷ Add dummy element
34:     return matrixC                           ▷ Consists of IC, JC and C
35: end function
```

The same experiment as in the previous section was executed, this time with the lastly described optimized version of the `sparse_cemm` algorithm. The used hash table is still `minimal_hash_map3`. The hash table was initialized with a size of 256 elements, this was enough for the particular problem instance. The performance of the new algorithm can be seen in table 6.2.

	gustavson	sparse_cemm
L3 cache misses [10^6]	353	205
L2 cache misses [10^6]	359	209
Instructions [10^9]	11.3	21.4
Time [ms]	18820	15096

Table 6.2: Comparison between original gustavson and sparse_cemm when multiplying *matrix*6_10

The last optimizations reduced the execution time even more, the speedup against the `gustavson` algorithm improves from the previous 11% to 20%. If the values are compared with the previous results from Table 6.1, it can be seen that the new implementation manages to reduce further the instruction count and in the same time to improve the L2 cache efficiency. Now the temporary data structure is small enough to fit even in the Level-1 cache. Compared with the previous implementation L2 cache misses droped by 42%, the number of retired instructions have been reduced by 23%.

As already mentioned thread migrations cause *NUMA* effects that amplify the costs of a cache miss. In such cases `sparse_cemm` shows even bigger speedup. If the running thread executes on a core of *node #0* and the memory is allocated on a module close to *node #1*, the cache optimized algorithm computes the product with 28% faster than the original `gustavson`.

Furthermore, the speedups also rise with denser matrices. For example a multiplication of a uniformly distributed matrix with same dimensions as *matrix*6_10 but with 14 elements per row, instead of 3, led to a 30% improvement of the execution time (versus 20% before). With more elements there will be more memory accesses and thus more cache misses that can be optimized by `sparse_cemm`. With this instance the algorithm caused reduction of the L3 cache misses by a factor of 5 against the original `gustavson`.

6.5 Generalizing the Algorithm

The presented algorithm has also considerable disadvantages. For example, the used hash table for storing the temporary values has static size. Growing steps were intentionally not allowed, since growing will force copying of the elements and thus lead to an overhead. Further, the static hash table is simpler and has less instructions, does not need growing logic.

However, the hash table must be sufficiently large so every row of the result matrix can be stored in it. This implies that in order to be able to use `sparse_cemm`, one has to know the maximum nonzero elements for a row in C. Since the data structure has static size, the algorithm is technically not applicable for matrices where $nnz(C(i,\cdot)) > |hashtable|$.

There are cases where this is not a problem. Before the multiplication takes place, we have read the input matrices A and B, so at this point the maximum nonzeros per row is known for the both matrices. Lets denote $\kappa_A = \max nnz(A(i,\cdot))$ for

$1 \leq i \leq n$, respectively $\kappa_B = max(nnz(B(i, \cdot)))$. Knowing these values, we can easily find an upper bound for the maximum number of elements of a row in C, $\kappa_C \leq \kappa_A \kappa_B$. This estimation can be used to determine the needed size of the hash table. However, although the upper bound is correct, it is very conservative (loose) in the general case. If we multiply two matrices with constant number of elements per row, this means there is no variance between the number nonzeros per row, the bound is quite exact. On the other hand, by large variance between the number of nonzeros per row, the bound is rather inaccurate. Hence, using it for such problem instances would lead to a highly overestimation of the size of the hash table. This is a problem because the algorithm is only beneficial if the hash table is small enough to fit in the cache.

Furthermore, `sparse_cemm` has its limitations. Replacing a large array with hash table will lead to a speedup only when:

1. The array is large enough so it exceeds the cache size
2. The array is sparse, so the stored values can fit in the cache
3. There is a random access over the array

In regard to the second point: the cache optimization technique would be beneficial only if the rows of the output matrix are sparse enough. It is not beneficial to replace the arrays with hash table for dense rows. In such cases the hash table would also flow out of the cache, and it would cause the same amount of cache misses as the dense arrays. Further, it will add an instructional overhead. Thus, `sparse_cemm` is feasible only when all of the rows of C are sparse enough to fit in the cache. However, in general case there is deviation between the densities of the individual rows. It is possible that there are some rows that are even completely full and in the same time the majority of rows are sparse. In such cases it would be not advisable to use `sparse_cemm` for the whole matrix. Firstly, because it is not beneficial for the dense rows and secondly because the hash table has static size and might be too small for the rows with many elements.

Yuster and Zwick [50] proposed an algorithm for sparse matrix multiplication with a hybrid approach that partitions the multiplication in a dense and a sparse section. This strategy is useful also in our case. Since the cache optimization technique is not suitable for every kind of density, one could differentiate between dense rows and sparse rows of the result matrix. Then, for the rows that are *sparse enough* we can use `sparse_cemm` and for the dense ones - the classical `gustavson`.

Here arises the question, how will the densities of the row of C be determined. For this task the density estimator from the previous chapter 5 is utilized. The output of the estimator is an array W of size n, containing the $mult_flops$ for each row. This value can be used as an upper bound for the nonzeros in the row (refer to The Density Estimator for more details). The generalized algorithm can be described with the following five steps:

Algorithm 6 `general_sparse_cemm`

1: Density estimation per row
2: Reordering of the rows by their density
3: Find suitable density threshold line
4: Use `gustavson` for rows with density above threshold line
5: Use `sparse_cemm` for rows under (or at) the threshold line

Initially, the implementation used sorting for the second step. However, complete sorting is actually redundant. It is sufficient to partition W around the threshold line, i.e. the threshold line is used as the pivot p of the partitioning. After the reordering of the array, elements larger than p have smaller index than the one of p, respectively elements with smaller or equal values have larger index. By using a partitioning, the time needed for the sorting ($\mathcal{O}(n \log n)$) is reduced to a linear scan. The sorting of W will be however not the bottleneck of the algorithm in most cases. The partitioning has also another important advantage. Through partitioning there is a larger probability that consecutive rows will be next to each other after the reordering. This will improve the locality and thus also the cache efficiency for the reading of matrix A. We used Hoare-Partitioning, since it has been reported as efficient when compared to other methods [48].

The value for the threshold density is hardware dependent. It should be chosen in such way that the rows under the threshold line can fit entirely in the lower levels of the cache.

The reordering of the rows is not made physically. Instead a permutation array over the original CSR structure is used. After running the density estimator with input A and B we get the array with the estimated nnz per row. The row indexes are reordered according to that array. In this way a permutation of the rows is obtained. The algorithm computes the rows of the result matrix by respecting the order of that permutation.

Overall, the approach generalizes `sparse_cemm`, so the algorithm can be used with all kind of matrices. The presented method performs cache optimization only on rows where this is feasible.

One disadvantage of the algorithm is the already mentioned processing of the rows by a permuted ordering. This will lead to non consecutive reads of the rows of matrix A. Thus, there will be some overhead of cache misses related to that. Processing the matrix A with a permutation will have the effect that the result matrix is computed with the same permutation of the rows. There will be no additional cache misses regarding the storing of the result matrix because the rows are written sequentially one after other. However, C will be stored in *permuted CSR* format. This might be considerable disadvantage with some applications (e.g. sequence of matrix multiplications).

In Figure 6.6 can be seen the execution times of the described generalized algorithm (`sparse_cemm`) for multiplying two different matrices. As usual, our baseline is the classical `gustavson` algorithm. The majority of the rows in the test matrices are sparse with only 3 elements per row, which are randomly distributed over the row. Thus, the matrices are suitable for the cache optimization technique. However, in

each matrix there are also 1000 denser rows, containing 10000 elements. This rows will be more denser after the multiplication and therefore using a hash table for them is not preferable. The difference between the two test matrices is that the dense rows in $matrix6_12$ are in a one consecutive segment and in $matrix6_13$ are spread over the whole matrix.

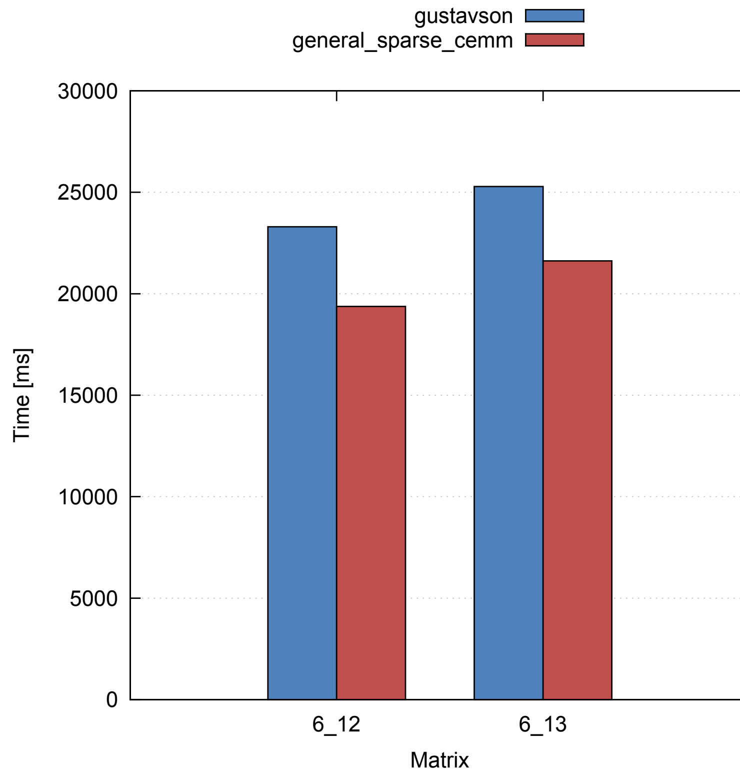

Figure 6.6: Execution times (ms) for two matrices.

The generalized algorithm outperform in both cases the traditional `gustavson`. The speedups for $matrix6_12$ and $matrix6_12$ are 17% and 16% respectively. This values are comparable with the result from the previous sections. Once again, if the memory allocation is not set to be optimal by hand, *NUMA* effects amplify the costs of the cache misses and thus lead to larger speedups. In such cases the speedup rises to 30%.

6.6 Conclusion

This chapter presented algorithms that improved even more the cache efficiency of `gustavson`. The improvement was done by exchanging the large dense arrays with a smaller hash table that would fit into the cache. Several hash table implementations have been compared. None of the already known implementations could outperform the dense arrays, although every hash table caused significant reduction of the cache misses. The experiments have shown that the instructional overhead is crucial for the performance. Better cache efficiency will not lead to a speedup at the end, if too much additional instructions are added to the algorithm. Therefore we constructed a custom minimalistic hash table specially tuned for the use case. The new data structure has considerable less instructions and succeeds to outperform the arrays with suitable problem instances. A new algorithm have been specified

(`sparse_cemm`). By employing the custom hash table, one could achieve speedup up to 30%. Further, a hybrid algorithm has been proposed, that uses the cache optimization technique only for the sparse rows of the matrix and the original arrays for the denser ones.

The tests have also indicated that *NUMA* effects have significant influence over the execution times. On a machine with only two nodes *NUMA* was responsible for a deviation of 30% when executing the same algorithm.

As a conclusion, the experiments have shown, that it is possible to improve the cache efficiency and the running time of an algorithm by replacing a large array with a smaller hash table. This holds only when specific criteria are met (see the previous sections). The presented cache optimization technique can be seen as general approach for improving the cache efficiency of an algorithm. Thus, it can be used also in other domains and not only for sparse matrix multiplication.

7. Parallelization

In order to improve the efficiency of algorithms, one has to consider possibilities for parallelized computation. This work focuses on a parallelization in a *shared memory* model.

7.1 Parallelization Framework

Intel Threading Building Blocks (Intel TBB) [42] and OpenMP [2] are parallelization frameworks, which enable parallel programming on more abstract level. They are conceptually different from low level threading libraries (pthread, boost), where each created logical thread is mapped to a thread, manged by the operating system (further on referred as physical thread). This direct mapping may lead to several disadvantages. If too many threads are created, the performance will suffer from the overhead of constantly context switching between the different threads. To the contrary, if the programmer creates too few threads underutilisation of the hardware will occur. The more sophisticated frameworks for parallel programming overcome these problems, as they introduce the concept of partitioning the computations in *tasks* or *work chunks*. This section sketches the design of a parallelization framework by discussing Intel TBB.

The framework creates a *thread pool* where the physical threads reside. The programmer can spawn tasks, which are assigned from the framework to the physical threads. Thus, there is conceptual difference between a *task* and a *thread*. The architecture is depicted in figure 7.1. Each physical thread has its own task pool, from which it fetches new tasks to execute. The TBB framework provides a higher level of abstraction, since it can determine on its own the optimal number of needed physical threads. By default, this number is about the same as the number of the logical cores presented on the machine.

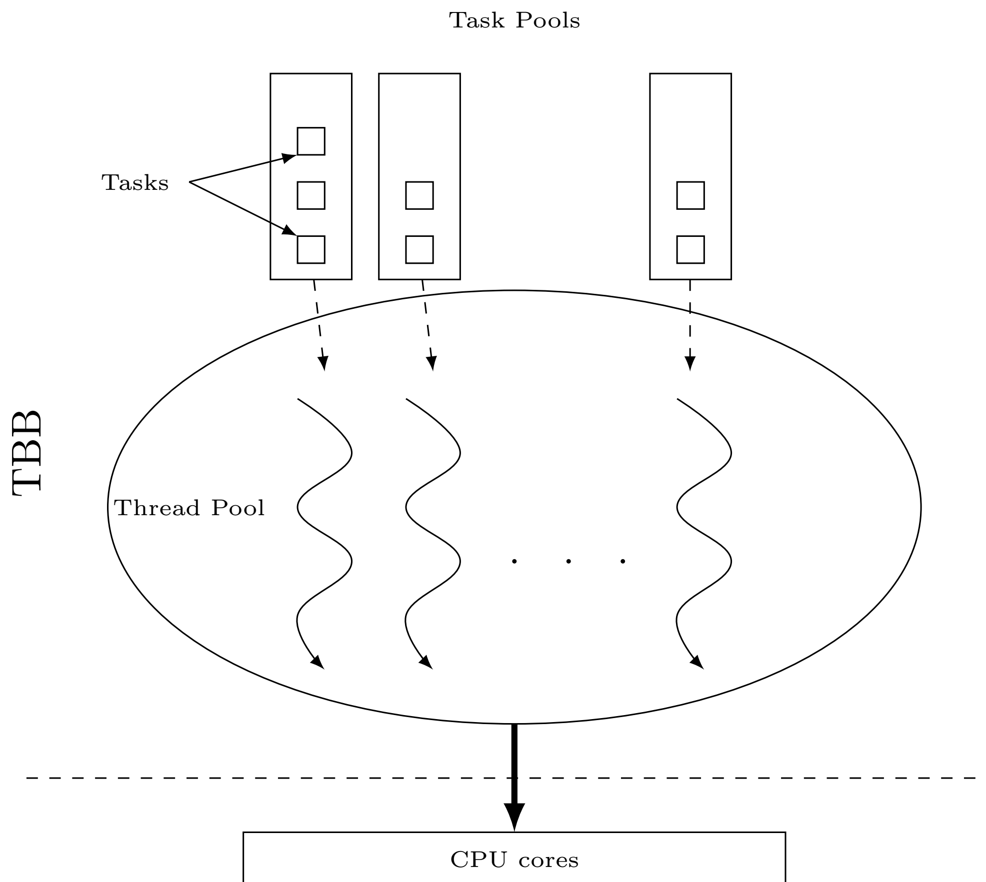

Figure 7.1: Task scheduler with a Thread Pool, as in Intel TBB.

This higher level of abstraction allows also finer granularity and better load balancing. The tasks can be seen as separate work chunks that are being executed by the physical threads. Tasks are much more light weighted than real threads. Thus, there is less overhead for spawning a task than for spawning a physical thread. This allows to partition a problem with higher resolution, i.e. dividing the problem on smaller work chunks. Although the needed time for starting and finishing a task is much smaller than these for a real thread, there is some overhead. If we have a sequential problem that cannot be partitioned, it would be always more efficient to compute it without the use of TBB. Thus, although task switching is not as expensive as thread switching, it is also a factor that have to be considered. The next sections will examine further how large this overhead is. They will present experiments that test the scalability of the task concept.

Another advantage of TBB is that it supports out of the box load balancing. The framework implements dynamical load balancing through *work-stealing*. Threads, that are ready and don't have any tasks left in their own task pool, *steal* tasks from other threads. Therefore, often it is a good strategy to spawn considerably more tasks than physical threads and leave the parallelization framework to do the load balancing.

Another important feature of the tasks is that they are not preemptive, thus there is a difference in their behavior when compared to threads. Let a thread *T1* executes a certain task *A*. The thread *T1* can be of course preempted, but *T1* will not start another task until it has not completed *A*.

Figure 7.2: Parallelization of `gustavson` by using equidistant 1-D partitioning.

7.2 Horizontal Partitioning

This section will present a parallel algorithm for sparse matrix multiplication in a *shared memory* model. As already empirically shown in the chapter *Sequential Algorithms*, the `gustavson` algorithm outperforms significantly all alternatives in the single thread case. Therefore, we concentrate our efforts in parallelizing this algorithm. The parallelization techniques will be also useful for the presented cache optimized version `sparse_cemm`, since it has the same algorithmic core.

The `gustavson` algorithm is well suited for parallelization. The result matrix is computed row by row sequentially. Further, there is no dependence between the different rows. The algorithm finishes the complete row i before it proceeds with $i + 1$. However, one must consider how to use the temporary arrays x and xb in a parallel setting. To prevent concurrent writes, we assigned a separate pair of temporary arrays to each execution thread.

Overall, the problem can be partitioned fairly easy on multiple work chunks. Each work chunk will be a consecutive segment of rows from the output matrix. This partitioning is known in the literature as horizontal partitioning or *1-D partitioning* [4], since the matrix is decomposed horizontally, in one dimension only. The partitioning pattern can be observed in figure 7.2. Notice that each task must have read access over the whole matrix B.

The matrix A is decomposed in segments and each segment is assigned to a different TBB task. For example, the task that processes the segment with row interval $[i...j]$ will compute the rows $[i...j]$ from the result matrix. One row of C is computed by only one task. Thus, there will be no overlapping between the segments and we can use different data structures for storing the smaller results. Therefore, each task writes in its own *CSR* structure. After all of the task have completed their computation, one can iterate over the segments and assemble the end result by just putting together the individual *CSRs* in the final result. There is also an optimization potential in the combining step alone. Firstly, there is no need to iterate over each single element of the segments and copy them to the final *CSR*. Since the computed segments are in *CSRs* already, we can just iterate over the whole data structures and append them to each other. After the tasks are completed the memory needed for the end result is known (the sum of all individual *CSR* sizes). Thus, we can allocate beforehand the needed space for C, there will be no costly resizing operations. As a second improvement, the joining procedure can be parallelized on its own.

Another time-consuming operation than can be optimized through parallelization is the creation of the CSR structures of both input matrices A and B. This is done by sorting the nonzero elements by their row index. The step is not directly in

the `gustavson` algorithm but it must be executed beforehand, so it would be also advantageous if we lower its time consumption. Optimization can be easily achieved by just replacing the sorting routine with a parallel one. Our experiments have shown that the creation of the *CSR* structures is substantial part of the whole time for the matrix multiplication. In some cases, the limiting operation is actually not the multiplication itself but rather the *CSR* construction. Therefore, it is highly beneficial to parallelize the sorting.

The naive approach would be to perform an equidistant partitioning, i.e. each task gets equal amount of rows assigned. The first $m - 1$ tasks process $(n \text{ div } m)$ rows and the last one computes the rest $(n \bmod m)$ rows. In the same time it is beneficial to increase the number of tasks more than the running physical threads. In this way the TBB will perform a dynamical load balancing automatically on its own.

With this approach we run the `gustavson` algorithm for each row segment (i.e. each task) independently. Thus, each task must have access to its own pair of x and xb arrays for storing its temporary results. In the first implementation each task created its own temporary structures and than freed the taken memory after completion. This does not mean that there will be as much x and xb arrays as tasks. Because the tasks are executed by the physical threads and a task is not preemptive, at each point of time there will be only as many pairs of arrays as the number of physical threads.

An alternative strategy is to extract the allocation of the x and xb arrays from the tasks and to perform it only once per thread and not in every single task. In this way the overhead of allocation is not dependent on the task count and the parallelization is more scalable.

Figure 7.3 illustrates the speedup achieved by the parallelized `gustavson` for 6 different matrices with the two different allocation strategies. The execution times for the case with allocation in each task can be seen in table A.2. As usual, each matrix is multiplied by itself. The speedups are plotted relative to a different task count and the number of running physical threads was left to be automatically optimized by the TBB framework. By default the framework spawns as many threads as the number of logical cores presented on the machine. Thus, in our case the experiments were executed with 16 threads running on 8 physical cores. It has to be noted that the thread count might be changed dynamically by the TBB framework during the execution, in order to achieve better performance.

The simple parallelization strategy leads to improvement of the execution time with all test matrices. It can be observed that the number of tasks have large influence on the achieved speedup. Further, there is a substantial discrepancy between the top speedups achieved for the different instances. The parallelization of $matrix3_3$ leads at best to a speedup factor of 2.4 against the sequential algorithm. On the other hand, the speedup rises to 5 with $matrix4_0$. Thus, we can conclude that the benefit of the parallelization is coupled to the structure of the matrix. The smallest problem instance $matrix3_3$ achieved also the smallest speedup. The parallelization framework has its overhead and the gain is not so substantial for small computational problems. This is also evident from the speedup values when we run the multiplication with only one task. For $matrix3_3$ it is slower to run `gustavson` with TBB and only one task than to execute the algorithm without the parallelization

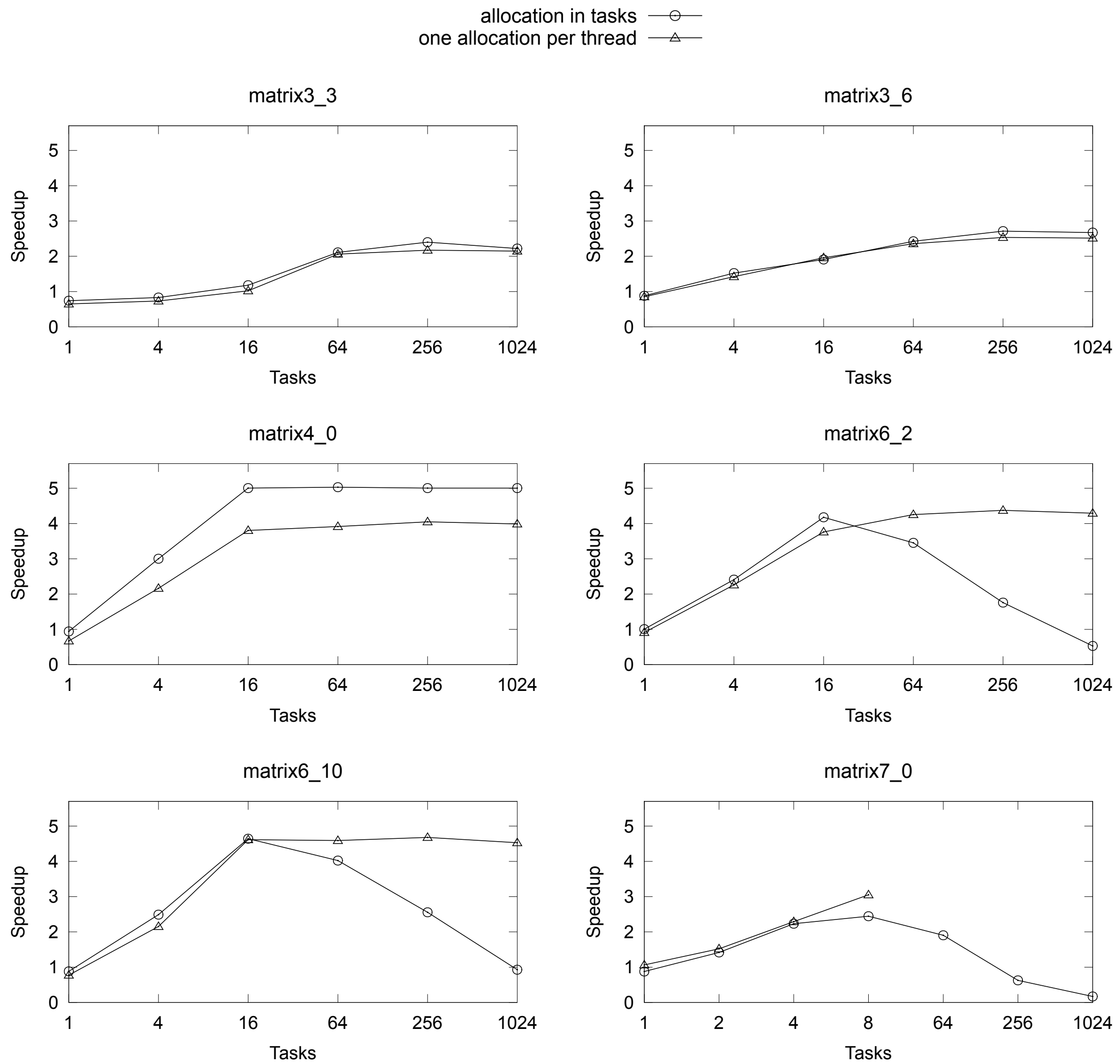

Figure 7.3: Speedups factors for different matrices and variation of the number of tasks; allocation of x and xb in each task *verus* allocation once per thread

framework. There are initialization overheads for starting and finishing a task that can be observed in this case.

First, we will focus on the performance when the allocation and deallocation of the temporary arrays is done *in each task*.

TBB employs work stealing between the threads. In this way the computation is dynamically load balanced over the processor cores. Therefore, one strategy to achieve better load balancing, and thus better execution time, is to spawn more tasks than running threads. The results from the experiment however show different behavior for some matrices. The graphs of $matrix6_2$, $matrix6_10$ and $matrix7_0$ show a rapid drop down of the speedups factors when the task count is increased beyond 16. In the same time this behavior is not present in the first three matrices, i.e. $matrix3_3$, $matrix3_6$ and $matrix4_0$. The two groups of matrices differentiate in their dimensions, the problem instances with the drop down in the speedup are significantly larger (see Table 3.2).

The rapid decrease in the speedup was caused by the non-optimal resource management of the x and xb arrays. Each task allocates the temporary structures for its execution. This allocation leads to a large overhead, as evident from the graphs. The first three matrices have smaller dimensions, so their x and xb arrays can be allocated relatively efficiently in the cache. Therefore, there was no rapid decrease in their running times when the task count was increased.

A more scalable strategy is to allocate the x and xb arrays once per thread and not once per task. In this way the overhead of allocation is not dependent on the task count. Work stealing may dispose tasks to different threads, so one pair of arrays might be used for not consecutive rows. However, this is not an obstacle since a task can use the xb array to check if a certain value belongs to the currently processed row.

The new implementation (*one allocation per thread*) uses a concurrent map, which is shared among the threads and contains a pair of x and xb arrays for each thread. Thus, the keys in the map are the *pthread ids* and the values are pair of pointers to x and xb arrays. When a task gets scheduled for an execution, firstly it gets the *pthread id* of its execution thread and then obtains the temporary structures assigned to that key in the map. The arrays for certain key are initialized when a thread with the same id accesses the map for the first time and performs a lookup for that key. Hence, there is also a write access to the map and therefore it has to be a concurrent data structure. The implementation used the `tbb::concurrent_hash_map`.

The speedups of the same experiment but with the updated implementation, where the allocation is extracted from the tasks, can be also seen in Figure 7.3.

The graphs indicate that the new implementation eliminates the allocation overhead. The trends for the smaller matrices (i.e. $matrix3_3$, $matrix3_6$) remain unchanged. However, there is no drop in the speedup by the large scale matrices $matrix6_2$ and $matrix6_10$. With the new version of the algorithm it was not possible to run the multiplication of $matrix7_0$ with more than 8 tasks (i.e. 8 threads) because of an out of memory exception. Although both variants start with the same number of threads, apparently the first version allocates less pairs of temporary arrays at a time and manages to fit the execution in the memory of the system. The memory needed for one pair of x and xb arrays is $51 * 10^6 * 12Byte = 612$MB for this instance. The second implementation allocates new pair of arrays when a new thread performs a lookup in the map. Thus, with 16 tasks and more it will allocate 16 temporary structures.

Further, it can be observed that the new implementation reduced the speedups for $matrix4_0$. Explanation for that could be that the lookups in the concurrent map might be more costly than allocating the arrays in this case, since they will have smaller size and could fit in the cache.

From the plots is also evident that the work stealing dynamical load balancing of TBB leads to improvements by some of the matrices, since the execution with more than 16 tasks, i.e. more tasks than threads, leads to increase of the speedup. The elements in $matrix6_10$ are uniformly distributed over the matrix, which leads to a well balanced problem. Thus, execution with more than 16 tasks should not increase the performance since each task will have a work chunk of the same size with the equidistant partitioning. Similarly, the elements of $matrix4_0$ follow a diagonal

pattern and are equally distributed over the matrix. Thus, this problem instance it also well balanced.

Overall, it can be concluded that the gustavson algorithm is very suitable for parallelization. The independent computation of the rows makes it possible to split the problem easily with *1-D* equidistant partitioning. Our parallelized implementations led to speedups up to a factor of 5 on a machine with 8 physical cores. However, there is also a large variance between the speedups for different matrices, where smaller problem instances achieve also smaller speedups. The experiments also indicate that the memory consumption of the parallelized algorithm could be an issue with matrices with large dimensions. For large scale matrices the allocation overhead for the temporary arrays x and xb might be a limiting factor.

7.3 Cache Efficiency in a Parallel Setting

In chapter 6 we presented an optimization technique that led to improvement of the cache efficiency and ultimately also reduced the execution time. This section will examine the performance of sparse_cemm when parallelization gets involved.

The already described strategy for parallelization with *1-D* partitioning can be used also for the parallel execution of the sparse_cemm algorithm. As already mentioned in a parallel setting each thread must have its own pair of x and xb arrays. This increases significantly the memory consumption of the algorithm when multiplying matrices with large dimensions. In the sequential case the whole capacity of the cache is used for for only one pair of arrays. With parallelization the cache has to be shared between the threads and each thread will try to fit sections of its temporary structures into the cache. They will have to compete for space in the cache. This will lead to cache contamination where some threads will cause the data relevant to others to be evicted. Thus, when gustavson is run in parallel, the possibility for a cache miss rises.

Further, in a parallel setting thread migrations between the different cores and even between the nodes are possible. This will not only cause more cache misses but will also lead to inevitable *NUMA* effects. In the sequential case we could avoid *NUMA* by forcing the execution thread to run on one specific core and simultaneously binding the memory allocation to the according node. With parallelization there is a need for fully utilizing the hardware, so the computation cannot be pinned to one core as before.

sparse_cemm managed to reduce the cache misses and to decrease the *NUMA* effects in the sequential case. The next observations examine the behavior in a parallel setting. Figure 7.4 depicts the execution time of the algorithm for multiplication of $matrix6_2$ with different number of tasks. The baseline is the parallelized gustavson algorithm. In both cases an identical *1-D* partitioning was used.

The graphic reveals an interesting result: although in the sequential case the cache optimization does not improve the running time for that problem instance, in the parallel setting sparse_cemm outperforms the classical algorithm.

Firstly, we will analyze the results in the sequential case. With one thread both algorithms have practically the same execution time. Apparently, this matrix does not fit in the class of matrices that benefit from our cache optimization (refer to the

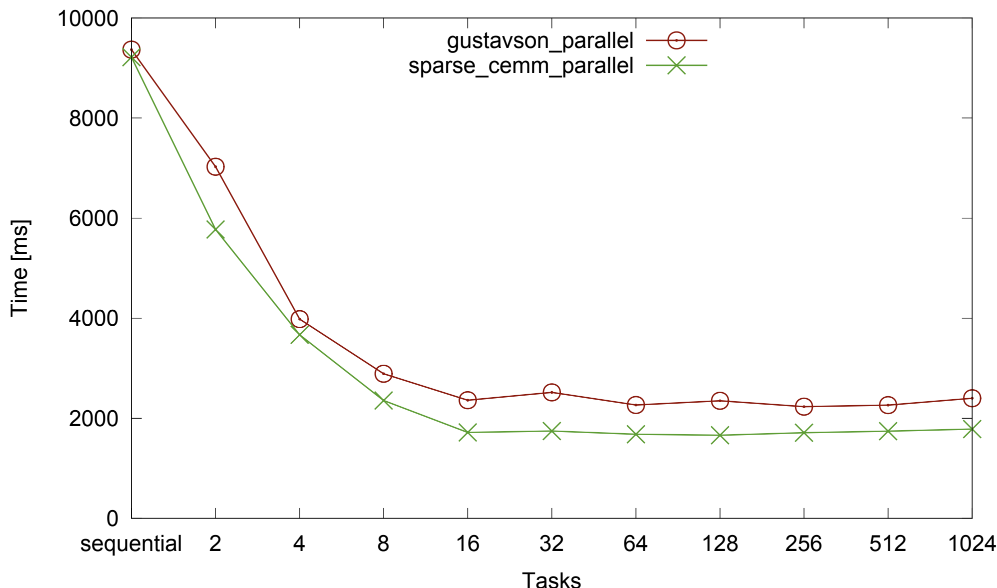

Figure 7.4: Execution times for `gustavson` and `sparse_cemm` when parallelized; $matrix6_2$

list at page 43). The dimension of the matrix is large enough so the x and xb arrays flow out of the cache. At the same time, the matrix is sparse enough, so there have to be stored only few values that can fit in the cache (the matrix has about 3 element per row on average). The problem is that the read/write access over the arrays x and xb is not random enough. There are patterns in the matrix that cause some sections of the arrays to be used more than others. This patterns are recognized by the hardware and the respective sections stay in the cache. Thus, there are not sufficiently many cache misses that the cache optimization strategy can eliminate. This is the reason for the lack of improvement in the sequential case. The particular problem instance is an example for the limitations of the cache optimization strategy. If there is a sequential access or some access patterns are presented, the arrays are still a very cache-efficient data structure.

However, when we increase the number of tasks, i.e. increasing the number of threads, `sparse_cemm` shows faster execution times than plain `gustavson`. The pair of arrays per thread cause that the memory consumption grows linearly with the number of threads with the classical algorithm. Thus, the cache inefficiency also increases with the parallelization. Hence, the presented cache optimization technique scales with the parallelization. Although, both algorithms had identical execution times when running on one thread, `sparse_cemm` led to a speedup of 26% when running on multiple threads. The speedup stays about constant after 16 tasks because from this point on the number of threads will also stay constant, i.e. there will be no more than 16 threads. As there is one pair of temporary structures per thread, we expect the speedup to grow even further if the thread count is increased.

The increase in the discrepancy by the cache efficiencies of the both algorithms can be observed in Figure 7.5. One can see there the ratio with which `sparse_cemm` has better cache efficiency when compared to `gustavson`. The absolute values of the L3 cache misses can be seen in Figure A.1. The graphic shows that the advantage of

the cache optimization technique is increasing with the task number, respectively thread number.

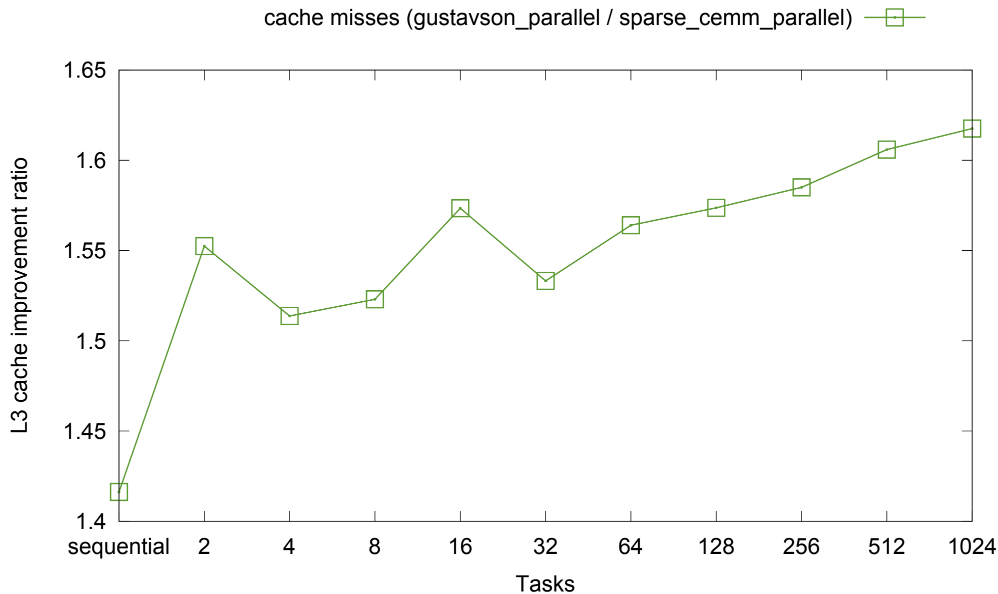

Figure 7.5: Improvement of the L3 cache efficiency; $matrix6_2$

A second significant advantage of the algorithm with hash tables is that it is more scalabe than the dense arrays. We improved the space complexity in the sequential case and now this improvement will multiply with the number of threads, because each thread must have a pair of x and xb arrays. Thus, with `sparse_cemm` it would be possible to execute a parallel multiplication of matrices with large dimensions, for which the parallelized `gustavson` will run out of memory.

An example for that behavior is $matrix7_0$ (Figure 7.6). This matrix represents the graph of the road network of Europe, the matrix was taken from the Florida Sparse Matrix Collection [12]. As seen in Figure **??** the multiplication cannot be executed with more than 8 Taks, i.e. with more than 8 Threads. However, when we use the smaller hash tables per thread instead of the large dense arrays, the computation can use the full capacity of the hardware, 16 logical cores in our case, and it will not run out of memory. With the parallel version of `sparse_cemm` was possible to achieve a multiplication time of $2020ms$ with 32 tasks, which is 40% faster than the best running time of the original algorithm.

As a conclusion, the experiments have shown that the presented in previous chapters cache optimization technique is even more worthy in a parallel setting.

7.4 Load Balancing

As seen in the previous sections, a naive partitioning of the computation in a combination with the dynamical load balancing of TBB through work stealing perform usually well for the parallelization of the multiplication. However, there are problem instances where the equidistant partitioning might be not the optimal strategy. Consider for example a problem instance where there is a very high variance between the nonzeros per row of the result matrix C as depicted in Figure 7.7. There is a

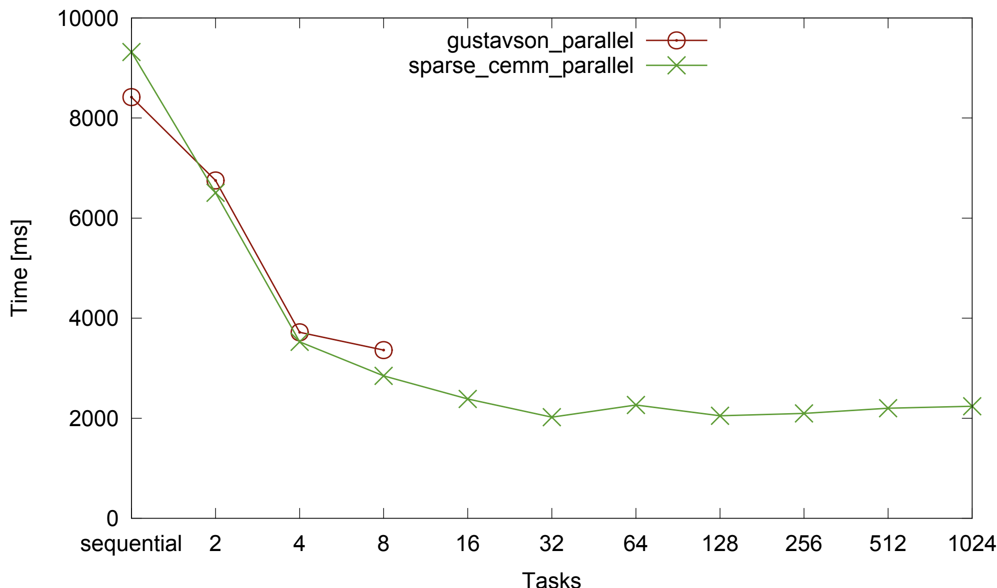

Figure 7.6: Scalability of `sparse_cemm`, multiplying the road network of Europe ($matrix7_0$)

narrow segment in the matrix with very high density (a hot spot) and in the same time the other rows are much more sparser. Respectively, the computational time for calculating the rows of the dense segment will be much higher than the time needed for the same amount of rows from different part of the matrix. Moreover, it is possible that the work needed for multiplying solely the dense segment is equal or even more than the computation power needed for the whole other part of the matrix. Thus, to make use of the parallelization and in order to achieve the optimal speedup factor one has to assign more threads to work on the dense segment. If only one thread is calculating the hot spot the algorithm will be not load balanced, since the other threads will idle and will have to wait until the thread computing the dense region finishes.

Distributing the dense segment over the threads means that the region has to be assigned to different tasks. The problem is that by large scale matrices with very high variance of the work load, i.e. high variance in the density of the rows, it is difficult to achieve that.

Consider for example the problem instance $matrix6_15$ (see table 3.2 in the Appendinx). The matrix has similar structure as the one illustrated in figure 7.7, more precisely - it has 400 consecutive rows with 110K elements per row and the rest rows of the matrix have 3 elements per row, the elements in a row are uniformly distributed. In the parallelization experiments the matrix was multiplied by itself. The indexes of the 400 dense rows were not presented in the sparse rows, thus the result matrix C will have also the same structure, i.e. 400 very dense rows and the rest very sparse.

Figure 7.8 depicts the execution times of the parallelized multiplication with three different load balancing strategies. The naive implementation (the baseline) uses the trivial equidistant *1-D* partitioning with work stealing, presented in the previous sections. As can be seen from the graphic, the increase of the task count up to 1024 does not lead to a speedup against the sequential implementation with this strategy.

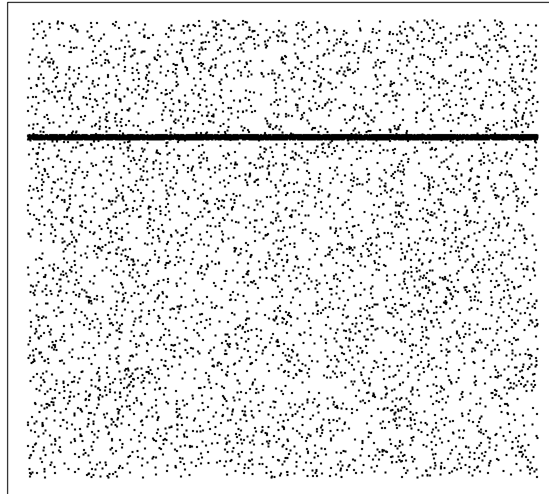

Figure 7.7: Example for a matrix with an unbalanced structure.

With 1024 equal work chunks the hot spot will be still in only one task and thus computed by a single thread. The concept of tasks offers a finer granularity than conventional threads. However, in large scale problems with high variance between the computational load of different parts, one has to spawn enough amount of tasks in order to evenly partition the problem.

On the other hand, one could employ other load balancing strategies to evenly distribute the computation among the threads. The presented density estimator (chapter 5) can be used to compute the needed multiplication flops per row of the output matrix. With this information one can perform an *1-D non-equidistant* partitioning, where each task will have a work chunk with the same amount of multiplication flops as the others. In such way, the dense region of 400 rows will be divided among multiple tasks and thus also multiple threads. Each multiplication flop leads also to a memory access to the temporary arrays x and xb. If we assume that the multiplication operations are the dominating factor of the computation than the non-equidistant partitioning with the density estimator will lead to an optimal decomposition of the problem. The overhead needed for the estimator is one scan over the nonzeros of matrix A ($\mathcal{O}(nnz(A))$).

The experiments indicate that this load balancing strategy outperforms the naive partitioning up to 1024 tasks. The increase of the task count cause a speedup even with only 2 tasks, in this case the computation will be evenly divided among two threads. After 16 tasks the execution time does not change significantly because with this tasks count the multiplication was already evenly assigned to the 16 logical threads of the test system. Nevertheless, the the approach does not exclude the out of the box work stealing of TBB. The two strategies are orthogonal and can simultaneously contribute for a better load balancing. Indeed, the work stealing increases further the speedup (see 7.8), since the execution time is reduced when more tasks are used.

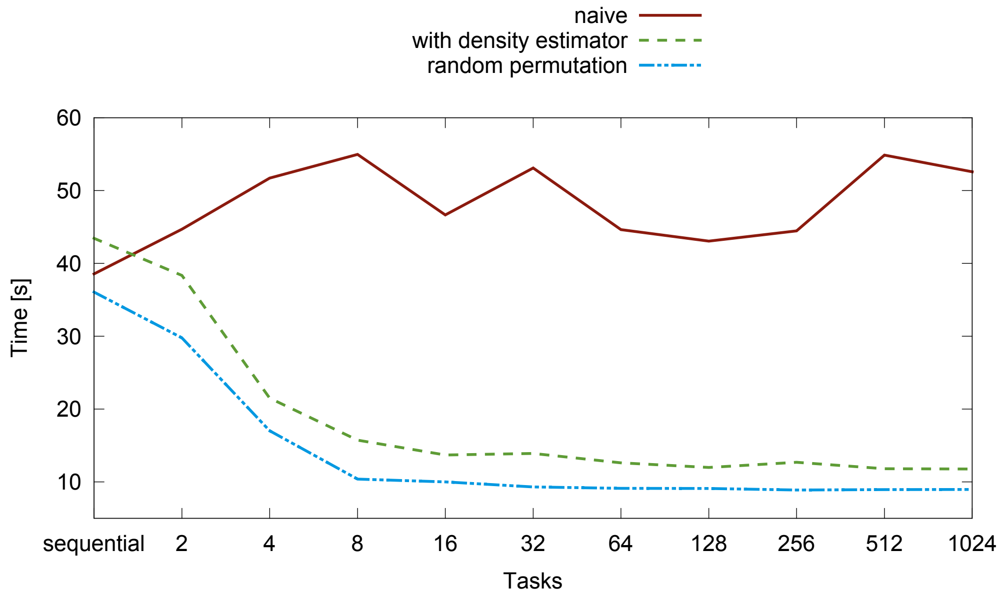

Figure 7.8: Performance of the parallelization with different strategies for load balancing; *matrix*6_15

Another option for better load balancing is to distribute each row of the result matrix to a randomly chosen thread. This will lead also to even partitioning of the dense section and as it can be see from the graphic it reduces the execution time even more than the variant with density evaluation. With this strategy there is no overhead for beforehand estimations. However, a disadvantage of the approach is that the rows of the output matrix are then stored in a randomly permuted order. Hence, depending on the application, one has to reorder the result after the multiplication.

The problem with the previous experiment is that the task count is too low so the naive partitioning cannot decompose the dense segment and it will be assigned to only one task. Therefore, the next test investigates the behavior of the load balancing techniques when the task count is further increased (see Figure 7.9). The large number of tasks leads to a finer granularity, so that the hot spot is then processed by multiple threads. Only when the dense segment can be assigned to multiple tasks, the work stealing of TBB manages to load balance the computation and the naive partitioning marks a speedup. The results show that the the TBB tasks are scalable and the overhead for spawning and finishing a task does not have heavy influence on the running time. Nevertheless, the other two approaches reached better peak speedups than the naive equidistant partitioning, where the load balancing with density estimation is with 15% faster and the random permutation with 26%.

OpenMP

Another framework for parallelization in a *shared memory model* is OpenMP. The framework is similar to TBB since it also provides mechanisms for dynamical load balancing of the computation. In order to be able to compare with the previous paralleization strategies, the implementation used the `#pragma omp parallel for` construct for the outer loop of `gustavson`. This leads also to an *1-D* partitioning of the multiplication, where each iteration of the loop (i.e. each row of the result matrix) can be independently computed. The keyword `parallel` spawns a team

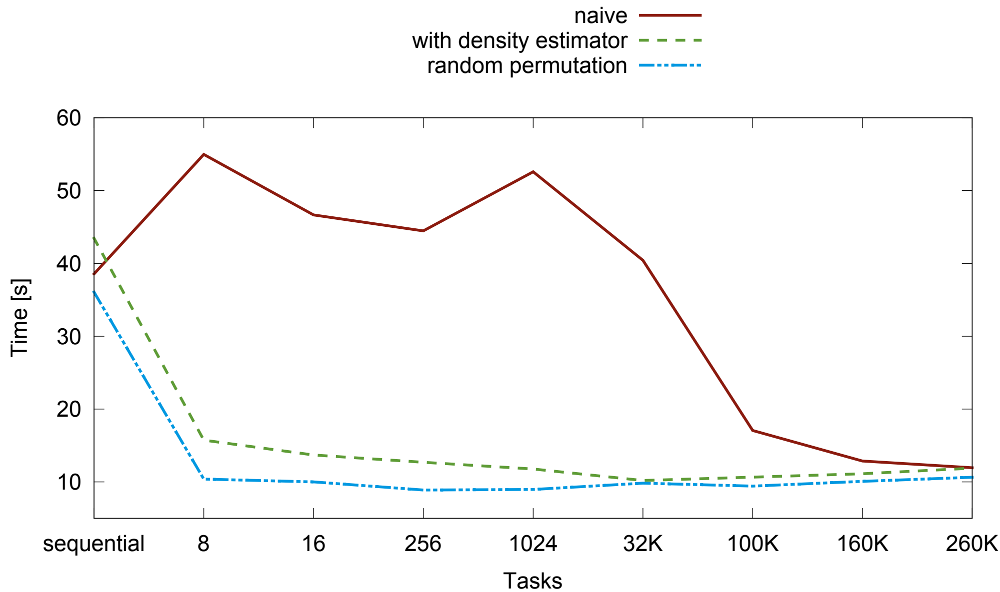

Figure 7.9: Performance of the parallelization by increasing the granularity of the partitioning; *matrix*6_15

of threads and `for` distributes the single iterations to the threads. In OpenMP one can choose from different scheduling strategies for the partitioning of the loop's iterations: *static*, *dynamic*, *guided* and *auto* [2]. Further, the granularity of the partitioning, i.e. the *chunk size*, can be specified with a second parameter. In this way it is possible to create work chunks with the same size as the TBB tasks from the previous section, thus one can compare the efficiency of the parallelization with OpenMP versus the one of TBB. Figure 7.10 presents the execution times for the paralleized multiplication of *matrix*6_15 with the *static*, *dynamic* and *guided* scheduling strategies and different number of chunks. With the *auto* setting the chunk size cannot be specified, in this case the framework determines the scheduling strategy automatically at run time. However, this setting proved less efficient than the others. The granularity was chosen in such way that the number and the sizes of the chunks correspond to these of the TBB tasks. As before, the tests have been executed with 16 threads, where each thread has its own temporary data structures and result *CSR*. At the end all *CSRs* are joined in one.

If we compare the different strategies at the point of highest speedup, i.e. at the point of finest granularity, it can be observed that the *dynamic* scheduling of OpenMP has the best performance, with 15% faster than the TBB implementation. However, it has to be noted that the *dynamic* scheduling distributes the chunks with no specific order to the execution threads. Hence, at the end the result will be permuted, on the other hand the TBB variant outputs an ordered result matrix. The *static* and *guided* scheduling strategies also lead to a permuted result.

7.5 Conclusion

This chapter showed that sparse matrix multiplication can be easily parallelized in *shared memory* through horizontal partitioning of the input matrix A and the result matrix C. The parallelization led to speedup up to a factor of 5 on a machine with 8 physical cores.

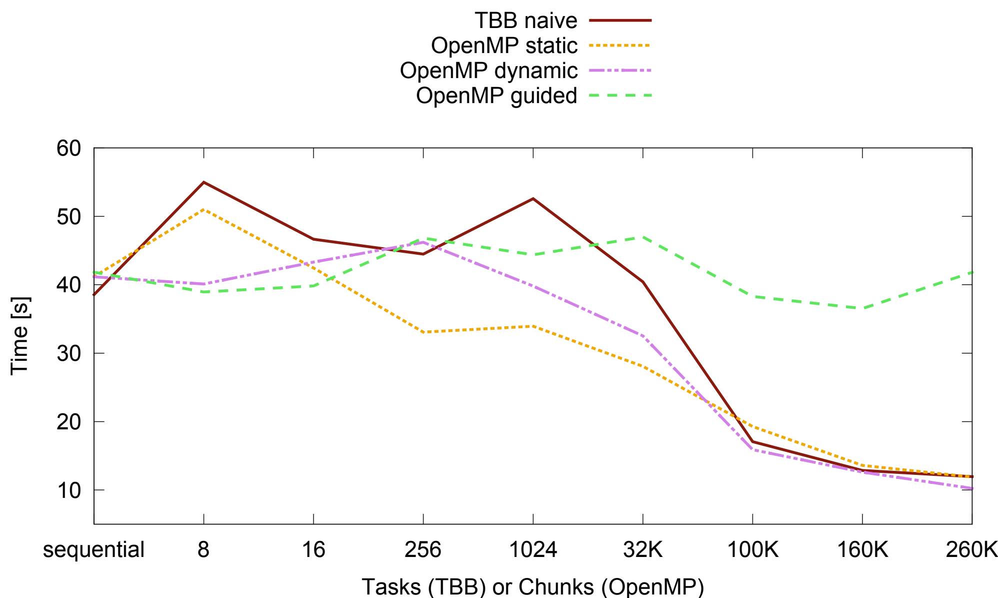

Figure 7.10: Parallelization with TBB and OpenMP; *matrix*6_15

Further, it has been shown that the parallelization of `gustavson` introduces new challenges for the cache efficiency and scalability of the algorithm. The cache optimization technique presented in `sparse_cemm` proved to be even more worthy in a parallel setting. For a problem instance where sequentially both algorithms have equal execution times, `sparse_cemm` was with 26% faster after parallelization. Another advantage is that the improvement in the space complexity multiplies with the number of threads and thus enhances the scalability. This led to 40% speedup against the parallel `gustavson` when multiplying the matrix of the road network of Europe.

The experiments concerned with the load balancing indicated that the dynamical load balancing, which TBB implements through work stealing, can achieve speedups even with highly unbalanced problems, when the computation is decomposed in small enough tasks. Nevertheless, with flops estimation or with random permutation of the rows one could achieve respectively 15% and 26% better execution times. The performance of the OpenMP implementation varied with the different scheduling strategies, where *auto* and *guided* have proven as inefficient and *static* and *dynamic* were close to the performance of TBB. However, the result with an OpenMP parallelization is a matrix with permuted rows.

8. Experimental comparison with other frameworks

Although there are many libraries for linear algebra, only few have support for sparse operations. Further, there are even fewer when it comes to sparse matrix-matrix multiplication. Many of the libraries, e.g. Intel MKL, does not provide methods for fully sparse matrix multiplication, where both input matrices and the output matrix are stored in a sparse data structure. Usually there are methods where at least one of the matrices in the multiplication is stored in a dense format. This is a problem for large scale matrices. Storing even one large scale matrix (e.g. $2^{24} \times 2^{24}$) in a dense format demands petabytes of free memory, when using 4 bytes for a matrix element. Thus, those frameworks are not applicable, or at least extremely cost ineffective, for large scale SpGEMM. In the previous chapters we have shown that matrices of such dimensions, and bigger, can be multiplied with only $24GB$ of memory in a matter of seconds. The task of this chapter is to compare the algorithms presented in this thesis with other frameworks and libraries that have support for full sparse matrix-matrix multiplication.

The comparison has been made with MATLAB - a widely used system for scientific computation that provides sparse linear algebra operations, among others also SpGEMM. The algorithm implemented in the competitor should be in the same complexity class as `gustavson`, i.e. the complexity of the computation should be proportional to the number of needed flops, the dimensions and the number of nonzero elements of B. To our surprise, we could not find library that offers parallelized version of the sparse matrix multiplication, this was also the case with the tested framework. Therefore, we compared the results with our sequential implementation of the classical `gustavson` algorithm and not with the parallelized versions. The result of the experiments with several matrices can be seen in Figure 8.1.

Our sequential implementation of `gustavson` succeeds to mark a speedup against competitor with seven out of the nine test matrices, where the speedup goes to more than a factor of 2 with $matrix3_6$ and $matrix6_10$. Analysis of the matrices where the framework was faster ($matrix4_0$ and $matrix5_10$) showed that the nonzeros in those matrices follow a diagonal pattern. This sort of matrices arise usually from

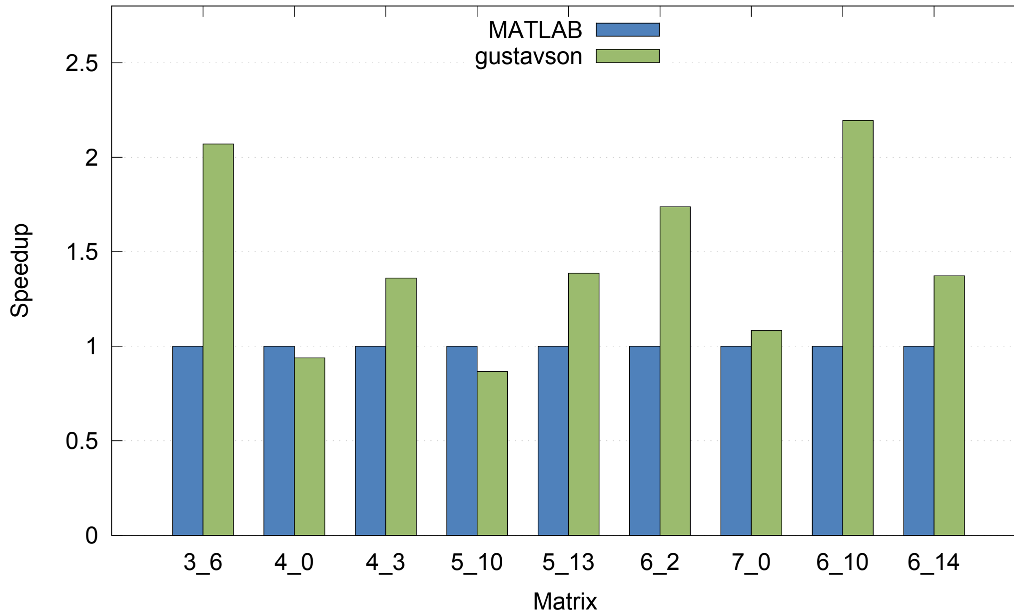

Figure 8.1: Comparison with a widely used system for numerical linear algebra, sequential execution

numerical, optimization and engineering problems. Regarding the results, one could suggest that the tested framework is performing slightly better for such kind of instances. On the other hand, for graph matrices, where there are different patterns in the distribution of the nonzeros, our implementation of `gustavson` achieves better execution times. The result prove that the algorithm is state of the art, and it is in many cases even more efficient than up to date systems for sparse linear algebra. The methods for parallelization and cache efficiency, presented in the previous chapters, enhance further the capabilities of the sequential algorithm. For example, Figure 8.2 shows how the speedup is improved even further if we use the cache optimization technique for matrices where this is feasible. Naturally, the presented parallel algorithms will increase also the speedup, this is evident from Figure A.2.

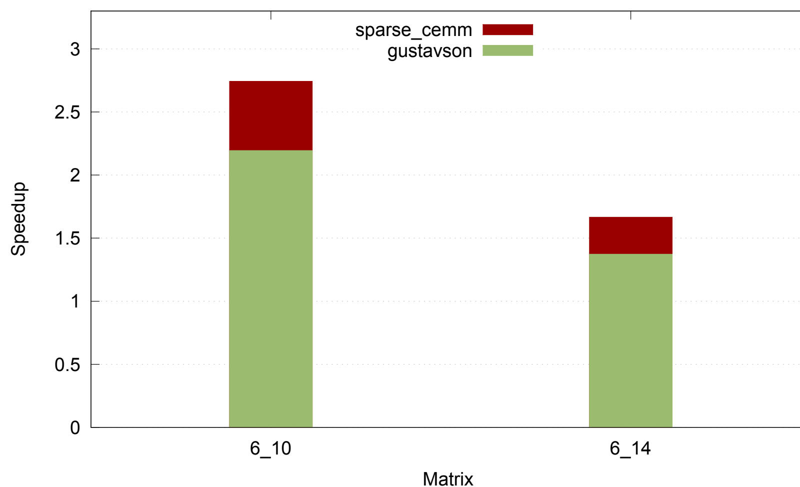

Figure 8.2: Further improvement of the speedup by `sparse_cemm`

9. Conclusion

The thesis presented multiple approaches and techniques for constructing efficient algorithms for sparse matrix-matrix multiplication. It shows that the `gustavson` algorithm is a highly efficient for executing the multiplication in the sequential case, where our implementation outperformed in many cases state of the art competitors.

The work proposes an improvement for the cache efficiency of the algorithm that in the end led to 30% reduction of the running time and improvement of the L3 cache-efficiency by a factor of 5 with some instances. The idea behind the new algorithm (`sparse_cemm`) is to replace very large arrays with few elements with smaller hash table that would fit in the cache. The experiments show that the hash table implementations from the standard library are not fast enough and thus not suitable for reduction of the running time. Although, they succeeded to reduce the cache misses, the standard library variants demonstrated large instruction overhead against the simple arrays, which leads to the slower execution. The work presents a minimalistic hash table that, by removing some functionality, managed to reduce the instructional overhead and marked a speedup against the arrays. The technique is not bounded to the problem of SpGEMM and can be used as general approach for improving cache efficiency also in other domains, where there are similar conditions.

Further, the thesis presents a simple algorithm for determining the needed multiplication flops before the actual multiplication takes place. The number of multiplication flops can be seen as an upper bound of the elements in the result matrix, thus the method is also useful for density evaluation of the output. Replacing an array with a hash table is not always beneficial. Therefore, the density estimation technique was used to construct a hybrid algorithm, which employs arrays for rows of the output that are too dense and a hash table for the sparser ones.

The test also indicated that *Non Uniform Memory Access* can have significant influence over the running times. On a test machines with only two nodes *NUMA* could lead to fluctuations of 30% by the running time.

The second part of the work shows how `gustavson` can be parallelized in a shared memory through one dimensional decomposition of the first input matrix and the result matrix. To our best knowledge, although effective and simple, such approach for a parallel SpGEMM is not present in the current literature.

Another finding was that the new algorithm `sparse_cemm` proved to be even more valuable in a parallel setting. For some matrices the technique does not lead to speedup in the sequential case, but it was with 40% faster than the classical algorithm in the parallel case. With the cache improvements and parallelization one could multiply the matrix of the road network of Europe (51 million nodes, 108 million edges) by itself on a machine with 8 physical cores and $24GB$ of memory in 2 seconds.

Also experiments with different load balancing strategies have been conducted. We used highly unbalanced problem instances, in order to investigate the properties and the scalability for the *out of the box* balancing strategies of TBB and OpenMP. Both frameworks show comparable results and succeed to achieve speedups when the granularity of the partitioning is high enough. The TBB tasks proved to be a scalable concept with minimal overhead. Nevertheless, a random permutation of the rows or partitioning through flops estimation leads to net speedup of 26%, respectively 15%, in some cases.

10. Future work

As proposal for future work, we would suggest the creation of cost model that puts into relationship the execution time of an algorithm, the number of its cache misses and the number of needed instructions. After such formula is found, one can then derive the break-even point between additional instructions and optimized cache misses. In other words we are looking for the number of additional instructions ψ that we are allowed to add as an overhead for reducing one cache miss and still not changing the running time. Knowing this value implies that each cache optimization strategy that adds less than ψ additional instructions will be worthy and will lead to a speedup at the end. The variable ψ might be dependent on the specific hardware.

It will be even more interesting, if ψ can be generalized for all kinds of algorithms and not only for matrix multiplication. If that is the case, the value can be used as a general restriction that has to be considered by algorithm engineers when trying to optimize cache misses.

Further, one could investigate more deeply the cost model of the matrix-matrix multiplication and find if the problem is memory-bound or CPU-bound. The matrix multiplication is a low level algorithm with few mathematical operations per iteration and quite a few memory accesses. Hence, there is the possibility that the time for the multiplication is bounded by the memory accesses. Therefore, it would be of interest to conduct experiments that will confirm, or deny, that the multiplication is memory-bound. Understanding the cost model will help to locate the bottleneck and thus it will provide valuable insights how to further speedup the algorithms.

Another possibility for research is to look for more locality, and thus better cache-efficiency. With the presented versions of the `gustavson` algorithm, there is a random access over the second matrix by the multiplication, i.e. B. With a *2-D* partitioning of matrix A, one could enforce better cache utilization for the B. For this purposes, a new data structure is needed - *blocked CSR*. The approach is not trivial and it will face multiple challenges, as for example need for dense structures for the output matrix.

A next proposal for future work can be to investigate external sparse matrix multiplication, i.e. using a slow non-volatile memory, which has large capacity. This would be needed for scalability of the computation.

Further, one could compare the presented here shared memory parallel algorithms to distributed methods in regard to efficiency and cost-effectiveness. Distributed approaches are more scalable, but the here presented algorithms might be more applicable for praxis relevant problems.

Another technique for improving the execution time of algorithms is to vectorize them. *Single Instruction Multiple Data* (*SIMD*) is approach that is gaining much attention and has proven as effective in many cases. However, if we assume that the running time of the algorithm is bounded by the memory accesses and not by the arithmetical operations, then *SIMD* would not lead to actual improvement. There is a need for an empirical evaluation of the matter.

Bibliography

[1] Rasmus Resen Amossen and Rasmus Pagh. “Faster Join-projects and Sparse Matrix Multiplications”. In: *Proceedings of the 12th International Conference on Database Theory*. ICDT ’09. St. Petersburg, Russia: ACM, 2009, pp. 121–126. ISBN: 978-1-60558-423-2. DOI: 10.1145/1514894.1514909.

[2] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. 2013.

[3] A. Buluc and J. Gilbert. “Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments”. In: *SIAM J. Sci. Comput.* 34.4 (2012), pp. 170–191.

[4] Aydin Buluc and John R. Gilbert. *Highly Parallel Sparse Matrix-Matrix Multiplication*. Tech. rep. Lawrence Berkeley National Laboratory and University of California, 2010.

[5] A. Campagna, K. Kutzkov, and R. Pagh. “On parallelizing matrix multiplication by the column-row method.” In: *ALENEX*. 2013, pp. 122–132.

[6] L. E. Cannon. “A Cellular Computer to Implement the Kalman Filter Algorithm”. PhD thesis. Montana State University, 1969.

[7] M. Charikar, K. Chen, and M. Farach-Colton. “Finding Frequent Items in Data Streams”. In: *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*. ICALP ’02. London, UK, UK: Springer-Verlag, 2002, pp. 693–703. ISBN: 3-540-43864-5. URL: http://dl.acm.org/citation.cfm?id=646255.684566.

[8] Edith Cohen. “Structure Prediction and Computation of Sparse Matrix Products”. In: *J. Combinatorial Optimization* 2.4 (1998), pp. 307–332. ISSN: 1382-6905. DOI: 10.1023/A:1009716300509.

[9] D. Coppersmith and S. Winograd. “Matrix Multiplication via Arithmetic Progressions”. In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC ’87. New York, NY, USA: ACM, 1987, pp. 1–6. ISBN: 0-89791-221-7. DOI: 10.1145/28395.28396. URL: http://doi.acm.org/10.1145/28395.28396.

[10] T. Cormen et al. *Introduction to Algorithms*. The MIT Press, 2009.

[11] Kernert. D., F. Köhler, and W. Lehner. “Bringing Linear Algebra Objects to Life in a Column-Oriented In-Memory Database.” In: *IMDM@VLDB*. 2013, pp. 37–49.

[12] Timothy A. Davis and Yifan Hu. “The University of Florida Sparse Matrix Collection”. In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011), 1:1–1:25. issn: 0098-3500. doi: 10.1145/2049662.2049663. url: http://doi.acm.org/10.1145/2049662.2049663.

[13] David S. Dodson, Roger G. Grimes, and John G. Lewis. “Algorithm 692: Model Implementation and Test Package for the Sparse Basic Linear Algebra Subprograms”. In: *ACM Trans. Math. Softw.* 17.2 (June 1991), pp. 264–272. issn: 0098-3500. doi: 10.1145/108556.108582.

[14] S. Duff et al. “Level 3 Basic Linear Algebra Subprograms for Sparse Matrices: A User-level Interface”. In: *ACM Trans. Math. Softw.* 23.3 (Sept. 1997), pp. 379–401. issn: 0098-3500. doi: 10.1145/275323.275327. url: http://doi.acm.org/10.1145/275323.275327.

[15] S. Filippone and M. Colajanni. “PSBLAS: A Library for Parallel Linear Algebra Computation on Sparse Matrices”. In: *ACM Trans. Math. Softw.* 26.4 (Dec. 2000), pp. 527–550. issn: 0098-3500. doi: 10.1145/365723.365732.

[16] G. Fowler et al. *The FNV Non-Cryptographic Hash Algorithm.* 2014.

[17] Robert A. van de Geijn and Jerrell Watts. *SUMMA: Scalable Universal Matrix Multiplication Algorithm.* Tech. rep. Austin, TX, USA, 1995.

[18] J. Gilbert, C. Moler, and R. Schreiber. “Sparse Matrices in Matlab: Design and Implementation”. In: *SIAM J. Matrix Anal. Appl.* 13.1 (Jan. 1992), pp. 333–356. issn: 0895-4798. doi: 10.1137/0613024. url: http://dx.doi.org/10.1137/0613024.

[19] J. Gilbert, V. Shah, and S. Reinhardt. “A Unified Framework for Numerical and Combinatorical Computing”. In: *Computing in Science and Engineering* 10.2 (2009), pp. 20–25. issn: 521-9615.

[20] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. “High-performance Graph Algorithms from Parallel Sparse Matrices”. In: *Proceedings of the 8th International Conference on Applied Parallel Computing: State of the Art in Scientific Computing.* PARA’06. Sweden: Springer-Verlag, 2007, pp. 260–269. isbn: 3-540-75754-6, 978-3-540-75754-2. url: http://dl.acm.org/citation.cfm?id=1775059.1775097.

[21] Gero Greiner and Riko Jacob. “The I/O Complexity of Sparse Matrix Dense Matrix Multiplication”. In: *Proceedings of the 9th Latin American Conference on Theoretical Informatics.* LATIN’10. Oaxaca, Mexico: Springer-Verlag, 2010, pp. 143–156. isbn: 3-642-12199-3, 978-3-642-12199-9. doi: 10.1007/978-3-642-12200-2_14.

[22] Fred G. Gustavson. “Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition”. In: *ACM Trans. Math. Softw.* 4.3 (Sept. 1978), pp. 250–269. issn: 0098-3500. doi: 10.1145/355791.355796. url: http://doi.acm.org/10.1145/355791.355796.

[23] *http://bebop.cs.berkeley.edu/oski/related.html.*

[24] *http://math-atlas.sourceforge.net/.*

[25] *http://sparsehash.googlecode.com/svn/trunk/doc/index.html.*

[26] *https://software.intel.com/en-us/intel-mkl.*

[27] *http://www.mathworks.com/products/matlab/.*

[28] *http://www.netlib.org/lapack/.*

[29] *http://www.openblas.net/.*

[30] *Intel Guide for Developing Multithreaded Application.* Intel. 2011, pp. 27–29.

[31] Bob Jenkins. *Hash functions.* Dr. Dobbs Journal. 1997.

[32] D. Kernert, F. Köhler, and W. Lehner. *SpMachO - Optimizing Sparse Linear Algebra Expressions with Probabilistic Density Estimation.* SAP.

[33] David Kernert, Frank Köhler, and Wolfgang Lehner. "SLACID - Sparse Linear Algebra in a Column-oriented In-memory Database System". In: *Proceedings of the 26th International Conference on Scientific and Statistical Database Management.* SSDBM '14. Aalborg, Denmark: ACM, 2014, 11:1–11:12. ISBN: 978-1-4503-2722-0. DOI: 10.1145/2618243.2618254. URL: http://doi.acm.org/10.1145/2618243.2618254.

[34] C. L. Lawson et al. "Basic Linear Algebra Subprograms for Fortran Usage". In: *ACM Trans. Math. Softw.* 5.3 (Sept. 1979), pp. 308–323. ISSN: 0098-3500. DOI: 10.1145/355841.355847.

[35] M. Mccourt, Smith B., and Zhang H. "Efficient Sparse Matrix-Matrix Products Using Colorings". In: (2013).

[36] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox.* ISBN-10: 3540779779. Springer, 2008.

[37] A. Metwally, D. Agrawal, and A. Abbadi. "An Integrated Efficient Solution for Computing Frequent and Top-k Elements in Data Streams". In: *ACM Trans. Database Syst.* 31.3 (Sept. 2006), pp. 1095–1133. ISSN: 0362-5915. DOI: 10.1145/1166074.1166084. URL: http://doi.acm.org/10.1145/1166074.1166084.

[38] H. Mohammadzadeh et al. "TitleFinder: Extracting the Headline of News Web Pages Based on Cosine Similarity and Overlap Scoring Similarity". In: *Proceedings of the Twelfth International Workshop on Web Information and Data Management.* New York, NY, USA: ACM, 2012, pp. 65–72. ISBN: 978-1-4503-1720-7. DOI: 10.1145/2389936.2389950.

[39] M. Patrascu and M. Thorup. "The Power of Simple Tabulation Hashing". In: *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing.* STOC '11. San Jose, California, USA: ACM, 2011, pp. 1–10. ISBN: 978-1-4503-0691-1. DOI: 10.1145/1993636.1993638. URL: http://doi.acm.org/10.1145/1993636.1993638.

[40] G Pike and J. Alakuijala. *CityHash: Fast Hash Funtions for Strings.*

[41] Sergio Pissanetsky. *Sparse Matrix Technology.* Academic Press, 1984.

[42] James Reinders. *Intel threading building blocks.* O'Reilly Media, 2007.

[43] L. Roditty and Uri Zwick. "Improved dynamic reachability algorithms for directed graphs". In: *SIAM J. on Computing* 37.5 (2008), pp. 1455–1471. DOI: 10.1137/060650271.

[44] K. Sruthi and B. Venkateshwar Reddy. "Document Clustering on Various Similarity Measures". In: *IJARCSSE* 3.8 (Aug. 2013), pp. 1269–1273. ISSN: 2277-128X.

[45] V. Strassen. “Gaussian elimination is not optimal”. In: *Numerische Mathematik* 13.4 (Dec. 1968), pp. 354–356.

[46] P.D. Sulatycke and K. Ghose. “Caching-efficient multithreaded fast multiplication of sparse matrices”. In: *Parallel Processing Symposium* (Mar. 1998), pp. 117–123. issn: 1063-7133.

[47] W. Vuduc and H. Moon. “Fast Sparse Matrix-vector Multiplication by Exploiting Variable Block Structure”. In: *Proceedings of the First International Conference on High Performance Computing and Communications*. HPCC’05. Sorrento, Italy: Springer-Verlag, 2005, pp. 807–816. isbn: 3-540-29031-1, 978-3-540-29031-5. doi: 10.1007/11557654_91.

[48] Sebastian Wild. “Java 7’s Dual Pivot Quicksort”. PhD thesis. Technische Universität Keiserslautern, 2013.

[49] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: An Insightful Visual Performance Model for Multicore Architectures”. In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. issn: 0001-0782. doi: 10.1145/1498765.1498785. url: http://doi.acm.org/10.1145/1498765.1498785.

[50] Raphael Yuster and Uri Zwick. “Fast Sparse Matrix Multiplication”. In: *ACM Trans. Algorithms* 1.1 (July 2005), pp. 2–13. issn: 1549-6325. doi: 10.1145/1077464.1077466.

[51] Uri Zwick. “All Pairs Shortest Paths Using Bridging Sets and Rectangular Matrix Multiplication”. In: *J. ACM* 49.3 (May 2002), pp. 289–317. issn: 0004-5411. doi: 10.1145/567112.567114.

A. Appendix

matrix \ algorithm	spspsp	outerP_hash	outerP_hash2	gustavson
matrix2_1	733	366	339	68
matrix3_1	2484	1740	1711	217
matrix3_6	11923	9045	9077	950
matrix4_0	20979	10993	10918	1912
matrix5_0	4734	3340	3276	303

Table A.1: Sequnetial algorithms, execution times in ms

Name	sequential	1 Task	8 Tasks	16 Tasks	64 Tasks	1024 Tasks
matrix3_3	889	1202	1038	752	422	400
matrix4_0	1907	2026	500	381	379	381
matrix3_6	1050	1193	637	550	433	393
matrix6_2	9398	9352	2756	2250	2722	17764
matrix6_10	20215	22925	4829	4355	5025	21776
matrix7_0	10303	11739	4216	3527	5411	60451

Table A.2: Execution times [ms] for the parallelized `gustavson` algorithm with different task numbers; allocation and destruction of the temporary arrays is done in every task.

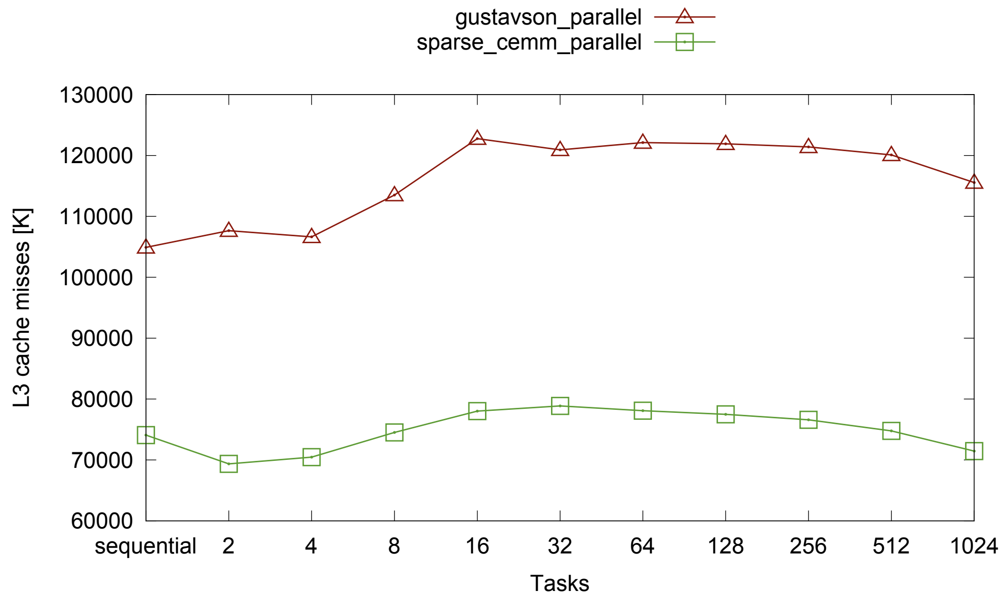

Figure A.1: L3 cache misses for both algorithms for different tasks; $matrix6_2$

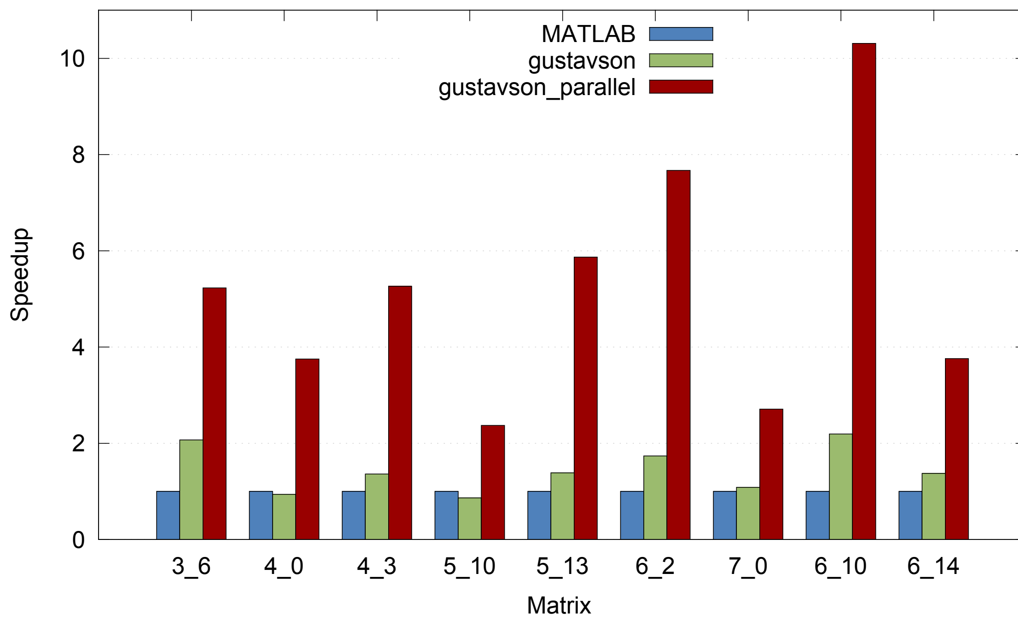

Figure A.2: Improvement of the speedup after paralleilzation on 8 physical cores