CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Engineering an Efficient Implementation of Pagh's Compressed Matrix Multiplication Algorithm

Master's thesis in Computer science and engineering

LUKAS SCHIAVONE

FILIP TORPHAGE

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Master's thesis 2024

Engineering an Efficient Implementation of Pagh's Compressed Matrix Multiplication Algorithm

LUKAS SCHIAVONE

FILIP TORPHAGE

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Engineering an Efficient Implementation of Pagh's Compressed Matrix Multiplication Algorithm
LUKAS SCHIAVONE
FILIP TORPHAGE

Supervisor: Matti Karppa, Department of Computer Science and Engineering
Examiner: Peter Damaschke, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Engineering an Efficient Implementation of Pagh's Compressed Matrix Multiplication Algorithm
LUKAS SCHIAVONE
FILIP TORPHAGE
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Matrix multiplication is a foundational operation in many fields of science such as machine learning, scientific computing and statistics. Simultaneously, matrix multiplication is an expensive operation to perform, with the mathematical definition requiring $\mathcal{O}(n^3)$ arithmetic operations to compute all the elements in a product of two matrices.

In this thesis we implement the algorithm for approximating matrix multiplication described in Rasmus Pagh's paper *Compressed Matrix Multiplication* (2013) in C++. The performance of the implementation of the algorithm is evaluated for different sizes of input matrices as well as varying input parameters and using different classes of hash functions. The results show that due to having a lower asymptotic complexity than traditional matrix multiplication methods, our implementation will regardless of the input parameters eventually be faster. The input parameters used will determine at which point this will occur.

We also measure the degree of parallelism available in Pagh's algorithm in this thesis and present the results in graphs to show the performance gained from increasing the number of utilized cores.

In addition, we have applied our implementation to the problem of calculating sample covariance matrices and studied which values of the input parameters are suitable for this problem. The accuracy of the approximations of Pagh's algorithm for computing sample covariance matrices are measured for varying parameters to determine this.

Keywords: Algorithms, randomized algorithms, sketching, matrix multiplication, C++.

Acknowledgements

To begin with, we would like to thank our supervisor Matti Karppa for providing us with invaluable advice throughout the course of the project. This field of research is a relatively unfamiliar one for the both of us, and without his guidance this project would not have been possible. We would also like to thank our fellow students Samuel Bach and Albin Karlsson for providing valuable feedback on our report.

Lukas Schiavone & Filip Torphage, Gothenburg, 2024-06-18

Contents

List of Figures

List of Algorithms

List of Listings

1

Introduction

Matrix multiplication is a fundamental operation in many areas of science, and consequently any improvement to the run time of matrix multiplication algorithms is useful. The definition of the element $c_{i,j}$ of the $m \times p$ matrix $C = AB$, where A is a $m \times n$ matrix and B is a $n \times p$ matrix, is the following:

$$c_{ij} = \sum_{k \in [n]} a_{ik} b_{kj},$$

where $[n]$ denotes $\{1, \ldots, n\}$, $i \in [m]$ and $j \in [p]$. The elementary algorithm for matrix multiplication applies this definition to calculate each element in C. Iterating over each element in C requires two nested for loops, and calculating the sum for an element requires an additional for loop, thus resulting in a triple nested for loops. As a result $\Theta(n^3)$ arithmetic operations are performed by the elementary algorithm.

Alternatively, a matrix product can be calculated by summing the pairwise outer products of the column vectors of A with the row vectors of B. The formula for the outer product method is defined as follows:

$$C = \sum_{k \in [n]} a_k \otimes b_k \tag{1.1}$$

where a_k denotes the kth column vector of A, b_k denotes the kth row vector of B and $\otimes$ denotes the outer product operator. Calculating one outer product requires n^2 multiplications and n outer products are calculated, hence this method also results in $\Theta(n^3)$ arithmetic operations being performed.

There are many approaches to reducing the number of arithmetic operations performed in a matrix multiplication algorithm, and among these approaches there are algorithms that make additional assumptions on the structure of the problem. Restricting the scope of an algorithm to only work for a subset of all possible problems allows it to make use of certain properties that are not guaranteed to be present in the general case. Examples of such problems include the multiplication of sparse matrices (matrices with few non-zero elements), multiplication of dense matrices and the multiplication of square $(n \times n)$ matrices.

Another area of research within the field of matrix multiplication algorithms is the study of the use of randomized algorithms for matrix multiplication. Randomized

algorithms are algorithms that employ some form of randomization in their procedures. There are two classes of randomized algorithms, Las Vegas and Monte Carlo algorithms. The former is the class of algorithms that compute exact results but the running time is a random variable and the latter is the class of randomized algorithms that compute an approximate solution to a problem. For the latter class of algorithms the error of the output must be bounded.

The algorithm proposed by Pagh in 2013 for multiplication of $n \times n$ matrices [1], hereinafter referred to as Pagh's algorithm, is one such randomized algorithm that computes an approximate solution. The matrix product is approximated by Pagh's algorithm by a sequence of steps involving compressing the matrix into polynomials with the use of 2-wise independent hash functions and then computing the Fourier transform of the polynomials to efficiently compute polynomial multiplication. The inverse Fourier transform is then applied and afterwards an estimation of each element in $C = AB$ is calculated. These steps only require one pass over the input matrices. If the product AB belongs to the class of "compressible" matrices, then calculating the approximate product C requires fewer arithmetic operations than traditional matrix multiplication methods, since AB is not explicitly constructed.

In this thesis we have implemented an efficient version of Pagh's algorithm in the C++ programming language. The implementation has been extensively benchmarked and analyzed across a range of conditions.

1.1 Aim

The main tasks of this thesis were the following:

1. Implement a prototype of Pagh's algorithm in the high-level language Python.
2. Engineer an efficient implementation of the algorithm in C++.
3. Empirically evaluate the low-level implementation of Pagh's algorithm and compare it to other matrix multiplication implementations.

1.2 Limitations

To ensure that the project could be completed in the given time frame a number of limitations were been imposed on the scope of the project. To begin with, the algorithm was implemented in C++ code meant for running on CPUs. A significant performance increase can be achieved by running code on a GPU, and thus there are more real world applications, but writing GPU code introduces further challenges that we have decided to be outside the scope of the project.

In addition, we have not focused on writing a program that makes use of the algorithm, except the benchmarking program. We have only written a small program that computes a covariance matrix given a set of samples. The thesis instead lays focus on evaluating the performance of the algorithm itself and examining potential situations in which the algorithm is applicable.

The scale at which we could evaluate our implementation was also limited by the hardware of the computational cluster that we had access to. The cluster does not represent what performance that state-of-the-art computational clusters might achieve, nor the theoretical result with an infinite number of cores.

2

Background

This chapter presents some of the important literature and concepts in the field, including papers leading up to Pagh's paper as well as papers published afterwards. Work related to the work done in this thesis is also presented in this chapter.

2.1 Matrix multiplication algorithms

The field of fast matrix multiplication algorithms, or more specifically: matrix multiplication algorithms with a complexity of $o(n^3)$, has been a contested field for a long period of time. Perhaps the most famous fast matrix multiplication algorithm is Strassen's algorithm, which achieves a complexity of $\mathcal{O}(n^{2.8074})$ [2]. Strassen showed in 1969 that a $\mathcal{O}(n^3)$ complexity for matrix multiplication was in fact not optimal by introducing a new divide and conquer algorithm for matrix multiplication. The algorithm recursively divides the input matrices into four submatrices in each recursive step and performs performs seven matrix multiplications of linear combinations of the different submatrices, which is less than the eight performed by the elementary algorithm. Although Strassen's algorithm does achieve a lower complexity, it is only practical for sufficiently large matrices because of sub-optimal memory access patterns and the constant factors that are abstracted away in the $\mathcal{O}$-notation.

After Strassen published his work there has been constant improvements to the complexity of matrix multiplication algorithms. The improvements are measured by the corresponding algorithms upper bound on the matrix multiplication exponent ω. The matrix multiplication exponent ω is defined as the least constant such that two $n \times n$ matrices can be multiplied using $\mathcal{O}(n^{\omega+\epsilon})$ arithmetic operations for all $\epsilon > 0$.

The first improvement after Strassen of the lower bound on ω was presented by Pan in 1978 [3]. The algorithm presented in the paper achieves an upper bound on ω of 2.795. Strassen later introduced another method for matrix multiplication named the laser method [4]. Using the laser method the following bound can be achieved: $\omega < 2.48$. Afterwards, Coppersmith and Winograd further improved on Strassen's ideas and brought the upper bound on ω down to 2.376 [5]. Every time the upper bound has been further decreased since, the method used has been a variation of the Coppersmith and Winograd method.

The lowest upper bound on ω achieved as of today is 2.371552. The bound was achieved by Williams et al. in 2023 [6]. Most algorithms for matrix multiplication

that fall under the category of "faster" algorithms with lower complexities than Strassen's algorithm, are unusable in practice due to large constant factors [7].

2.2 Sparse matrix multiplication algorithms

To improve the efficiency of matrix multiplication algorithms, one might limit them to operate on sparse matrices, thereby enabling further optimizations and reducing the number of arithmetic operations necessary to compute a product. This type of problem is often referred to as sparse matrix-matrix multiplication (SpGEMM), the term originates as a variant to the general matrix multiplication (GEMM) [1] which is a part of Binary Linear Algebra Subroutine (BLAS) [8]. The development of methods for solving SpGEMM problems is a prominent field. Although there is no consensus on the formal definition of the concept of sparse matrices, a matrix is usually considered sparse when the number of non-zero element $b \ll n^2$. If a matrix is sparse, then the most efficient method of representing the matrix is by simply enumerating the non-zero elements. Intuitively, the number of arithmetic operations required for multiplying sparse matrices will depend on the number of non-zero elements in the input matrices.If A and B are sparse, then most elements in $C = AB$ are likely to be zero, and in this case only the non-zero elements in C need to be explicitly constructed.

One might also consider the cases where prior knowledge about the inputs indicates that their product will be sparse. In these cases further reductions in the number of arithmetic operations may be possible. For example, when calculating a sample covariance matrix given samples of a multivariate random variable X, if most pairs of random variables (x_i, x_j) in X are independent then most elements of the covariance matrix are expected to be close to zero.

Some earlier works in the field include Yuster and Zwick's paper from 2005 [9] that details a fast SpGEMM algorithm and Gustavsson's algorithm first presented in 1978 [10]. Yuster and Zwick's algorithm was later improved upon by making it more efficient in the cases where the output matrix was also sparse by Amossen and Pagh in 2009 [11].

A lot of recent work in the field [12] is focused on accelerators, hardware-based solutions specifically optimized for SpGEMM. Accelerators are designed to maximize memory reuse and the number of computations done in parallel. An example of an accelerator for SpGEMM is MatRaptor [13].

2.3 Randomized algorithms

Randomized algorithms are algorithms that utilize elements of randomness in their procedures. Randomizing an algorithm may be beneficial for a multitude of reasons. For example, an algorithm that performs a random selection of some object may

[1] GEMM is defined as an operation of the form: $C \leftarrow \alpha AB + \beta C$, where A, B and C are matrices and α and β are scalars.

avoid calculating what the best choice is by choosing an object that is expected to be good, thus reducing the number of operations required compared to a deterministic algorithm. Another potentially useful aspect of certain randomized algorithms is their resistance to worst case inputs, i.e inputs that would cause a deterministic algorithm to perform as many operations as possible. For randomized algorithms this occurrence is instead tied to a low probability rather than specific inputs.

Randomized algorithms are divided into two categories, Monte Carlo algorithms and Las Vegas algorithms. Monte Carlo algorithms are algorithms that are guaranteed to terminate in a certain time but allow the output to be incorrect with a certain probability. An example of a Monte Carlo algorithm is a fast algorithm for approximating matrix products in introduced by Drineas et al. [14]. Las Vegas algorithms are guaranteed to produce the correct result, but the running time itself is a random variable. An example of a Las Vegas algorithm is the randomized incremental construction for convex hull [15], an incremental algorithm that incrementally constructs a convex hull in expected $\mathcal{O}(n \log n)$ time.

2.4 Related work

Pagh and Pham further expanded upon the work done in [1] by introducing the technique known as Tensor Sketch [16]. Tensor Sketching is a sketching technique (see Section 3.3) that is very similar to the one used in Pagh's algorithm, with the main difference being the use of hash functions with some key differences. It has proven useful for many problems within the fields of numerical linear algebra and machine learning, such as tensor decomposition [17] and approximation of deep neural networks [18].

As of now there does not seem to exist an implementation of Pagh's algorithm in a low-level language, nor any language for that matter. There are, however, multiple implementations of the application of Tensor Sketch described in Pagh and Pham's paper in Python libraries [19][20].

3

Theory

This chapter presents some of the key theory behind the algorithm that this thesis concerns, Pagh's algorithm. Then, the details of the algorithm as well as its theoretical guarantees are presented. The specific details of the hash functions implemented in this thesis are also presented in this chapter.

3.1 Fourier transform

The Fourier transform is a transform of a function to a complex-valued representation of the function describing the different frequencies present in the original function. The discrete Fourier transform (DFT) is a Fourier transform of a finite sequence of values to a finite sequence of complex values. The equations for computing the DFT and the inverse DFT (IDFT) respectively of an element in a sequence of length n are defined as the following:

$$X_k = \sum_{j=0}^{n-1} x_j \cdot e^{-\frac{i2\pi}{n}kj}, \; x_j = \frac{1}{n}\sum_{k=0}^{n-1} X_k \cdot e^{\frac{i2\pi}{n}kj} \tag{3.1}$$

The equations in Equation (3.1) are applied for each element in the corresponding (complex or real-valued) sequence to compute the DFT or IDFT of the sequence.

One can use the DFT to compute the product of two polynomials $p_1(x)$ and $p_2(x)$ in $\mathcal{O}(n \log n)$ operations instead of the $\mathcal{O}(n^2)$ operations required by other methods. The polynomials $p_1(x)$ and $p_2(x)$ must be stored must be stored by their coefficients for this method. If $p_1(x) = a_0x^0 + a_1x^1 + \cdots + a_nx^n$ then the coefficient representation of $p_1(x)$ is $\{a_0, a_1, \ldots, a_n\}$. The first step is to compute the DFTs of the coefficient representations of p_1 and p_2 to get $P_1 = \text{DFT}(p_1)$ and $P_2 = \text{DFT}(p_2)$. Then, an element-wise multiplication of P_1 and P_2 is performed to get the resulting product $P = P_1 \circ P_2$. Lastly, the inverse DFT (IDFT) is applied to P to get the polynomial $p(x)$ which is the product of $p_1(x)$ and $p_2(x)$.

Applying the DFT to coefficient representations of the polynomials $p_1(x)$ and $p_2(x)$ transforms them to pointwise representations, where each point is the polynomial evaluated at a complex root of unity. The pointwise representations of two polynomials can be multiplied element-wise to get the pointwise representation of their

product. To multiply the coefficient representations of two polynomials a convolution must be performed instead. The convolution operation has a higher asymptotic complexity than applying the DFT, which is why the method of multiplying polynomials using the DFT requires fewer operations than the traditional method past a certain size of $p_1(x)$ and $p_2(x)$.

3.1.1 Fast Fourier Transform

A FFT algorithm is an algorithm for computing the DFT of a sequence of values of length n using $\mathcal{O}(n \log n)$ operations. The first FFT algorithm can be attributed to Gauss who invented an recurisve algorithm for computing the DFT in 1805 [21]. The most well known FFT algorithm, the Cooley and Tukey FFT algorithm, was invented in 1965 [22]. It is a recursive divide-and-conquer algorithm that for an input of composite size $n = m \cdot p$, computes m transforms of size p recursively. The algorithm works the best for values of n that are powers of two, in these cases additional optimizations can be made. In the cases where n is a power of two, the input array of length $n = 2^r$ can be divided into two arrays of size $\frac{n}{2}$, with the elements with even indices in one array and the elements with odd indices in the other. The DFT is of each half is then computed recursively by dividing the input array by the same principle in each recursive call. The input is halved in each successive recursive call until the input is of size $n = 1$. The results of each recursive call are combined to compute the DFT of the entire sequence.

Due to the importance of n being a "highly composite number" as stated in the original publication, one might choose to use other FFT algorithms in the cases where n is not such a number, and is instead a number such as a prime number or a multiple of a prime number.

Since the formulas for computing the DFT and the IDFT are nearly identical as can be seen in Equation (3.1), an FFT algorithm can also be applied for computing the inverse FFT (IFFT) with minor modifications.

3.2 Hash functions

Hash functions are functions that map input keys of arbitrary sizes to output values, also known as hashes, of a fixed size. Naturally, the size of the input domain is usually significantly larger than the size of the output domain, and thus there is a non-zero probability that two inputs map to the same output. One of the key properties of hash functions is the avoidance of such events, this property is known as collision resistance.

Another desirable of property of hash functions is that the output of a hash function should behave randomly and be distributed somewhat uniformly among possible output values. This is commonly achieved by not using a fixed hash function and instead randomly selecting the hash function to be used, h, from a class of hash functions $\mathcal{H}$ with certain properties. The hash of a key x, $h(x)$, is then a random variable and depending on which class h belongs to, $h(x)$ will behave differently in

regards to how it is distributed and the probabilities of certain events such as two keys mapping to the same value occurring.

In the following subsections A will refer to the input domain and B will refer to the output domain of a hash function.

3.2.1 Universal hash functions

Universality is a property of classes of hash functions that provides certain theoretical guarantees for hash functions randomly selected from them. A class of hash functions $\mathcal{H}$ from A to B is said to be universal if for all $x \neq y$, $|\{h \in H \mid h(x) = h(y)\}| \leq \frac{|H|}{|B|}$ [23]. Alternatively, H is universal if for any unique keys $x, y \in A$ and hash function $h \in H$, $\Pr[h(x) = h(y)] \leq \frac{1}{m}$.

3.2.2 k-wise independent hash functions

A concept related to universality is the concept of k-wise independence. The property of k-wise independence provides stronger guarantees than the property of universality. If a class of hash functions is k-wise independent, then it is also universal. This relation does not hold in the other direction.

A class $\mathcal{H}$ of hash functions from A to B is said to be k-wise independent, or strongly universal$_k$ as the concept was originally introduced [24], if for any $h \in \mathcal{H}$ and k distinct elements $x_1, \ldots, x_k$, the hashes $h(x_1), \ldots, h(x_k)$ are independent and uniformly distributed in B. The property can also be formulated with the following equality:

$$\Pr[h(x_1) = q_1 \ \wedge \ \cdots \ \wedge \ h(x_k) = q_k] = |\,B\,|^{-k}. \tag{3.2}$$

One of the simplest classes of k-wise independent hash functions is the class of polynomials of degree $k-1$ in a finite field (both A and B are the field) [24]. Given k unique points in A and k corresponding points in B, there is only one polynomial $p(x)$ of degree k such that $p(x_i) = y_i$ for $i = 1, \ldots, k$. Only one out of $|B|^k$ polynomials of degree k passes through each of the pairs, thus Equation (3.2) holds for this class of hash functions.

The property of k-wise independence is necessary for the theoretical guarantees of many randomized algorithms, although small values of k are usually required. It is important for these algorithms that the hashes of keys $x_1, \ldots, x_k$ act as independent and uniformly distributed random variables. The hashes being uniformly distributed lowers the probability of collisions occurring, and the number of collisions impact the runtime of many algorithms. For example, a 5-wise independent class of hash functions can be used such that search, insertion and deletion operations can be performed in expected constant time in a linear probing scheme for hash tables [25]. The expected runtime depends on the expected number of collisions, and thus a hash function which minimizes collisions is essential. The k hashes being independent is

important for the theoretical properties and analysis of randomized algorithms that utilize k-wise independent hash functions.

One can relax the property of k-wise independence such that it is only required that the probability of keys $x_1, \ldots, x_k$ mapping to $q_1, \ldots, q_k$ is less than or equal to $\mu \cdot |B|^{-k}$, where $\mu \geq 1$, rather than $|B|^{-k}$. A class of hash functions that fulfils this property is instead said to be (k, μ)-wise independent [26].

3.2.3 Fully random hashing

The simplest implementation of a k-wise independent class of hash functions is the class of fully random hash functions. A fully random hash function maps each key to an independent uniformly distributed random variable. Implementing a fully random hash function for positive integers is simple, to hash a key one can index an array initialized with samples of independent and uniformly distributed random variables with the value of the key.

Fully random hashing is often impossible to apply in practice due to the space necessary to store one member of the class. The space required to store one fully random hash function is equal to the space required to store all elements in the input domain, since one random variable is assigned to each key. For example, hashing 64-bit unsigned integers to 32-bit unsigned integers would require storing 2^{64} values for one hash function. This would require storing a total of $2^{64} \cdot 32 = 2^{69}$ bits which is equivalent to 64 exbibytes (EiB). An additional issue with fully random hashing is the difficulty of obtaining truly random numbers.

3.2.4 Multiply-shift hashing

Multiply-shift hashing is a fast universal hashing scheme proposed by Dietzfelbinger et al. [27]. The hash of a key is computed by multiplying the key with a odd number a modulo a power of two and then dividing the product with another power of two. The formula for computing the hash of a key using multiply-shift hashing is the following:

$$h_a(x) = (ax \bmod 2^k) \cdot \frac{1}{2^{k-\ell}},$$

where $|A| = 2^k$, a is a odd number belonging to $\{0, 2^k\}$ and $\ell \in \{1, \ldots, k\}$. The main benefit of multiply-shift hashing is how efficient it is to compute the hash of a key. The division can be implemented with a fast bit-wise right-shift since dividing by a power of two is equivalent to right-shifting with the exponent. If implemented in a programming language with access to instructions for unsigned integers of a specified size and on a CPU that supports said instructions, the slow modulo operation can be removed since multiplication of unsigned integers will result in the overflow being discarded. An additional benefit of multiply-shift hashing is that it is only necessary to store k bits for one hash function. With the above mentioned optimizations in mind the multiply-shift can be implemented in the programming language C as seen in Listing 3.1 with parameters $k = 64$ and $\ell = 32$.

```
uint32_t hash(uint64_t x, uint64_t a) {
    return (a * x) >> 32;
}
```

Listing 3.1: Multiply-shift in C

Multiply-shift can be altered to be $(2, \frac{5}{4})$-wise independent with some modifications as shown by Dietzfelbinger [28]. The formula for this version of multiply-shift is the following:

$$h_{a,b}(x) = ((ax + b) \bmod km) \cdot \frac{1}{k},$$

where $k \geq |A|$, m is the number of bins and $0 \leq a, b < km$. If m, $|A|$, k and km are all powers of two then this class of hash functions is 2-wise independent.

The version of multiply-shift tested in the algorithm implementation is one proposed by Thorup [29]. This version of multiply-shift hashes to m bins, where m can be any arbitrary number, by computing two shift operations rather than one. Two random numbers a and b are used as coefficients. This version can be implemented in C with the code seen in Listing 3.2.

```
uint32_t hash(uint32_t x, uint32_t m, uint64_t a, uint64_t b) {
    return (((a * x + b) >> 32) * m) >> 32;
}
```

Listing 3.2: Thorup's multiply-shift in C

3.2.5 Tabulation hashing

Tabulation hashing is a hashing scheme that computes the hash of a key by treating the key as a string and dividing it into c blocks and hashing each block with a unique table that maps blocks to hashes. Then, the hashes of each block are combined using bit-wise XOR operations to form the hash of the key [30]. The formula for computing the hash of a key using tabulation hashing is the following:

$$h(x) = h_1(x_1) \oplus \cdots \oplus h_c(x_c),$$

where $\oplus$ denotes the bit-wise XOR operation. This method of hashing avoids the need of computing expensive integer multiplications by instead performing bit-wise XOR operations and table lookups. Tabulation hashing was shown to possess multiple useful qualities such as 3-wise independence by Pătraşcu and Thorup [31]. The pseudocode of an example tabulation hashing scheme can be seen in Algorithm 1.

Algorithm 1 Tabulation hashing

```
function GenerateTable(p, r)
    t ← ⌈p/r⌉
    for i ← 0 to t − 1 do
        for j ← 0 to 2^r − 1 do
            T[i][j] ← randomNumber()
        end for
    end for
    return T
end function

function Hash(T, p, r, x)
    t ← ⌈p/r⌉
    mask ← (1 ≪ r) − 1
    hash ← 0
    for i ← 0 to t − 1 do
        hash ← hash ⊕ T[i][(x ≫ (r · i)) & mask]
    end for
    return hash
end function
```

p is the number of bits of the key to be hashed, r decides the block size, T is a matrix where each row is a hash table for a block and & is a bit-wise AND operator.

3.3 Sketching

Sketching is a technique used to approximate different data structures, such as vectors, matrices or streams in a compressed, smaller format. The purpose of sketching techniques are to reduce dimensionality whilst preserving specific key properties of the original data, thus improving computational and analytical efficiency whilst working with the compressed object but at the cost of reduced accuracy. To illustrate the areas in which sketching techniques can be used, the algorithm that thesis concerns, Pagh's algorithm, utilizes the sketching technique known as Count Sketch, further detailed in Section 3.3.1, to compute sketches of row and column vectors of its input matrices. The sizes of the sketches computed are determined by a parameter b, whilst the row and column 2vectors contain n elements.

Sketching techniques are commonly applied in the field of numerical linear algebra and in nearly all cases utilize some form of randomization to form a compressed representation of the input. The randomization can be done in one of many ways, and one of which is to compress a matrix by multiplying it with a random matrix known as a sketching matrix sampled from a distribution of matrices [32]. To illustrate further, given the input matrix A, multiply it by the sketching matrix S sampled from some distribution of matrices to form a compressed matrix $C = S \cdot A$ on which further computations are performed.

Other methods of sketching matrices rely on randomly sampling from the input matrix to build a sketch of the matrix. There are multiple ways of sampling from the input matrix, including element-wise, row-wise and column-wise sampling. Row-

and column-wise sampling are preferred to element-wise sampling due to providing better worst-case bounds on errors [33].

3.3.1 Count Sketch

Count Sketch is a sketching technique introduced by Charikar et al. for estimating the most frequent elements in data streams using only one pass over the input data [34]. Although Count Sketches were originally applied to the problem of estimating the most frequent items in data streams, they have later proven to be applicable for other problems in the field of numerical linear algebra. For example, objects such as vectors can be treated as streams of numbers for the purposes of the Count Sketch algorithm.

The Count Sketch of a data stream $Q = \{q_1, \ldots, q_n\} \subseteq U$ is calculated using two sets of hash functions $\{s_1, ..., s_t\} \subset \{s \mid s : Q \to \{-1, 1\}\}$ and $\{h_1, ..., h_t\} \subset \{h \mid h : Q \to \{1, \ldots, b\}\}$ from 2-wise independent classes of hash functions as well as a matrix C of size $t \times b$ of counters. Each row i of the matrix C corresponds to the hash functions (h_i, s_i) and contains b different counters $c_{i,1}, \ldots, c_{i,b}$ that are incremented/decremented by the Count Sketch algorithm.

The algorithm for computing the Count Sketch of a stream involves updating a single counter in each row i in C using the corresponding hash functions (h_i, s_i) for each element $q_k \in Q$ in the stream. The value of $h_i(q_k)$ is used to determine which counter in the row i is to be updated and the value of $s_i(q_k)$ is added to the counter. Since s_i maps to $\{-1, 1\}$ this corresponds to incrementing/decrementing one counter in each row for each element in the stream. These steps lead to the algorithm making t estimates of the count of q_k.

To track the a most frequent elements the algorithm keeps a heap of size a. After an element q_k in the stream is processed with the above steps, the count of q_k is estimated by calculating the median of the t estimates. If the median is greater than the smallest estimated count on the heap then q_k is added to the heap and the element with the lowest estimated count on the heap is removed. If q_k is already on the heap then the estimated count is incremented by one. The pseudocode for the Count Sketch algorithm can be been in Algorithm 2.

Count Sketch has the useful property of linearity, and thus the Count Sketch of an object is equal to the sum of the Count Sketches of each component. This can be extended to multiplication as well, the Count Sketch of a product is equal to the product of the Count Sketches of each operand. Count Sketch refers to a row vector c_k in the counter matrix C in the previous statements and will do so for the rest of this thesis.

Algorithm 2 Count Sketch algorithm

```
function Add(C, q_k)
    for i ← 1 to t do
        C[i, h_i(q_k)] ← C[i, h_i(q_k)] + s_i[q_k]
    end for
end function

function Estimate(C, q_k)
    return median(C[1, h_1(q_k)] · s_1(q_k), ..., C[t, h_t(q_k)] · s_t(q_k))
end function

function CountSketch(C, Q, Heap)
    for j ← 0 to n − 1 do
        Add(C, q_j)
        if q_j ∈ Heap then
            Increment estimated count of q_j
        else if Heap not full then
            Add q_k to heap
        else if Estimate(C, q_j) > min(Heap) then
            Remove min(Heap)
            Add q_j to heap
        end if
    end for
end function
```

3.4 Pagh's algorithm

Pagh's algorithm is a Monte Carlo algorithm that calculates a sketch $\tilde{C}$ of a matrix product $C = AB$ without explicitly constructing C. There are two parameters, b and d that determine the accuracy of the algorithm. The parameter d determines how many times the algorithm is repeated and b is a parameter determining the time/accuracy trade-off. The sketch $\tilde{C}$ will be equal to C with high probability if C has at most b non-zero elements. The algorithm computes $\tilde{C}$ by computing an unbiased estimator of each element c_{ij} in C. The variance of the estimator of c_{ij} is bounded by $\|C\|_F^2/b$. The algorithm computes $\tilde{C}$ in $\tilde{\mathcal{O}}(N + nb)$ time[1], where $N \leq 2n^2$ is the number of non-zero elements in the input matrices A and B.

The two key ideas of Pagh's algorithm is to calculate the sketch of C as the sum of the sketches of each outer product as in Equation 1.1 and to use the FFT to efficiently calculate the multiplication of two Count Sketches. The sketch of each outer product is calculated as the product of the Count Sketches of the corresponding row and column vectors.

The algorithm is composed of two algorithms, the compression algorithm and the decompression algorithm. The first step of the compression algorithm is to randomly select h_1, h_2 and s_1, s_2 from 2-wise the independent classes of hash functions $\mathcal{H}$ and $\mathcal{S}$

[1]The $\tilde{\mathcal{O}}$-notation suppresses polylogarithmic terms.

respectively. Using these hash functions, the algorithm computes the Count Sketches of the column vector a_k and the row vector b_k, applies the FFT to a_k and b_k and then performs element-wise multiplication of the transformed sketches of a_k and b_k. The transformed sketches of the outer products $a_k b_k$ for $k \in [n]$ are then summed, and as as result the Fourier transform of a Count Sketch of C has been computed in the form of a row vector of length b. The above steps are repeated d times, with d different sets of hash functions to compute d transformed Count Sketches of C. To get d Count Sketches of C the IFFT is applied to each transformed Count Sketch of C.

To estimate the value of an element c_{ij} in C, the decompression algorithm is given the indices i and j as well as the d sketches computed by the compression algorithm. The decompression algorithm computes the median of d different approximations of the element c_{ij} that are calculated according to a simple formula using the same hash functions used to compress the polynomials. The pseudocode for Pagh's algorithm can be seen in Algorithm 3.

Algorithm 3 Pagh's algorithm

function CompressedProduct(A, B, b, d)
  $p \leftarrow \underline{\mathbf{0}}$
  for $t \leftarrow 0$ **to** $d-1$ **do**
    $h_1[t], h_2[t] \in_R \mathcal{H}$
    $s_1[t], s_2[t] \in_R \mathcal{S}$
    for $k \leftarrow 0$ **to** $n-1$ **do**
      $(pa, pb) \leftarrow (\underline{\mathbf{0}}, \underline{\mathbf{0}})$
      for $i \leftarrow 0$ **to** $n-1$ **do**
        $pa[h_1[t](i)] \leftarrow pa[h_1[t](i)] + s_1[t](i) \cdot A_{ik}$
        $pb[h_2[t](i)] \leftarrow pb[h_2[t](i)] + s_2[t](i) \cdot B_{ki}$
      end for
      $(pa, pb) \leftarrow (\mathrm{FFT}(pa), \mathrm{FFT}(pb))$
      for $z \leftarrow 0$ **to** $b-1$ **do**
        $p[t][z] \leftarrow p[t][z] + pa[z]pb[z]$
      end for
    end for
  end for
  for $t \leftarrow 0$ **to** $d-1$ **do** $p[t] \leftarrow \mathrm{FFT}^{-1}(p[t])$ **end for**
  return (p, h_1, h_2, s_1, s_2)
end function

function Decompress(i, j)
  for $t \leftarrow 0$ **to** $d-1$ **do**
    $X_t \leftarrow s_1[t](i) \cdot s_2[t](j) \cdot p[t][(h_1[t](i) + h_2[t](j)) \bmod b]$
  end for
  return Median($X_0, ..., X_{d-1}$)
end function

The $\in_R$ notation denotes an independent uniformly random choice and $\underline{\mathbf{0}}$ denotes a zero vector or matrix.

4

Methods

This chapter presents the most important libraries, methods and techniques used during the development of the algorithm implementation.

4.1 Prototyping

Before implementing Pagh's algorithm in the main programming language of the thesis, C++, we implemented it in the high-level language Python. The purpose of implementing the algorithm in Python was twofold. The first of which was to familiarize ourselves with and implement the algorithm without needing to focus much on language specific details. Python served this purpose well due to its simple syntax and large assortment of available libraries. The Python implementation later served as the baseline for the subsequent C++ implementation. We were also able to compare the results produced by the C++ implementation with the results of the Python implementation to verify the correctness of the C++ implementation in the early stages.

4.2 Testing

When we implemented the algorithm it was important to ensure that it worked as expected. Tests were used both when implementing new features, but also to ensure that new optimizations did not alter the already functioning properties of the program.

The library that was used for testing was Catch2 [35]. Catch2 is mainly a unit testing framework for C++ that offers a simple way to write tests. By using this, we could test all functionality of every method, including the hashing methods, in a clear and concise way.

4.2.1 Statistical testing

To be able to verify the correctness of our implementation during the thesis work we wrote a test that verifies the variance bound of each estimator of the elements in the output matrix, presented in section 3.4.

4.3 Tools

In order to gain more insight into how the program behaves during its execution, there are numerous tools that were used and many of them have different use cases. The tools that were most suitable for this project were those that could analyze the execution of the program.

With this in mind, Valgrind [36] was used for debugging in the early stages of our C++ implementation on our local computers. Valgrind is a suite of debugging and profiling tools that can be used for various purposes. We ended up using one of the more popular ones, namely Memcheck [37]. Memcheck is a tool for detecting memory-management problems. This tool allows us to quickly find memory leaks in the program, giving the stack-trace with each memory leak found.

Callgrind [38] is another tool distributed by Valgrind that was used on our local computers. It is a profiler that accurately computes the number of times each instruction is executed. As cycle count is not equivalent to execution time, it was important to take this into account when using Callgrind. However, cycle count could still be used to find external function calls that take a significant amount of time to run.

A third tool that was used, and the only one used on the cluster, was the performance analysing tool *perf* [39] in Linux. This tool was used to analyze cache usage. This includes information such as L1 and TLB cache-misses and branch-misses.

4.4 Benchmarking

As the result was empirically evaluated, it was crucial to ensure accurate performance measurements of the implementation. Our implementation was compared to state-of-the-art matrix multiplication libraries, therefore one certain precondition were put in place. It is common practise in the field of high-performance computing to preallocate everything beforehand in order to get the best performance. Therefore, the benchmarks were performed with only a few dynamic memory allocations, more on why some dynamic memory allocation were used at all can be read in section 7.2. By minimizing the dynamic allocation to a much lower number, better performance could be attained. With this precondition in place, a fitting benchmarking option was later chosen.

Benchmarking a program is not as straight forward as it might seem at first. In theory it is just measuring before and after the execution of a program and reporting the resulting time, but there is more to it than that. It is not always clear on what to measure when reporting the data. It's always good to have multiple samples, but from these samples multiple values can be retrieved, some common ones are any of the Pythagorean means [40], median or even the minimum measured time. There are advantages and disadvantages for each of these, and it is not agreed upon in the field of computer science which is the better choice. Some of these can be used in combination with each other to further strengthen their advantages. An example

of this is by combining calculating the median and the minimum executed time. If these two values are near equivalent, then that means the program is consistent and that you know that any outliers are not taken into account. Further more, it may also depend on what you are actually measuring and what you are trying to get out of the result. Additionally, retrieving the standard deviation for each benchmark would give us good indication of when the output is consistent or not. All of these metrics were measured for every test, and which one that was displayed in the result was chosen once all the results had been given.

When benchmarking an algorithm there are some general ideas that should be used. It may be beneficial to warm-up the CPU by running a few iterations on it. When modern CPUs goes idle their clock speed gets lowered to a lower clock frequency. When the CPU suddenly gets an order to execute and instructions it may take some time for the CPU to get its clock frequency back up again. Running too long benchmarks may also cause some uncertainties, which may be caused by running too many warm-ups, as the CPU can begin to throttle once it reaches a certain temperature. This will drastically lower performance and is something that was avoided as much as possible.

When benchmarking there are also other factors that need to be factored in. As the evaluation will be based on results from a cluster, the nature of a computational cluster that multiple people use constantly will have affects on the results.

4.4.1 Benchmarking tool

As our implementation was evaluated, it was also be benchmarked. There were several options available and that we ended up using in different stages of the project.

While Catch2 is first and foremost a testing library, it also offers benchmarking capabilities with a neat integration with existing tests. This ended up being the first benchmarking tool that were used for the C++ implementation. However, not much control over how the benchmarks were run or what data they provided was given to the user. While this may not be a problem for functions that have lower running times, it did not suffice here, as our running times could take long period of times, they could also vary by a lot. One of the missing features that we needed were allowing benchmark specific parameters such as number of samples. Catch2 only provided global variables for such parameters. Additionally, you could not get the raw data from each iteration or sample; instead, you just got a summary of them such as mean or median.

The second benchmarking library that was used were Google Benchmark [41]. Google Benchmark is a highly regarded open source benchmarking library written by Google. Google Benchmark offered lots of tools that fits many needs. Every benchmark could be individually configured and every repetition of a benchmark could be stored, this way we can locate if ever the running time suddenly changes. While we used this for some time, it became difficult to use for our purpose. Google Benchmark did not support templates as arguments in the way our program was built, making the library difficult to work with. There were also several statistics that was not

measured here that we wanted.

Eventually the benchmarking tool was rewritten by ourselves. That way we could make sure that we got everything we wanted out of benchmarking. This benchmarking tool makes use of the *chrono* standard C++-library. The *chrono* standard library is an library that can be used to accurately measure the wall clock time. This has the advantage that it can be built to fit our exact needs and potentially avoid any overhead caused by other benchmarking libraries. While our benchmarking implementation was initially rather limited, its functionality quickly expanded to support everything that was needed from it and more.

4.4.2 Optimization flags

When compiling our code there were several optimization flags that we could choose from, that each have different effects on the runtime of the program. The three candidates, from the `-O` flag category, that we had to choose from were `-O2`, `-O3` and `-Ofast`.

- `-O2` This flags optimizes the code to a large degree but has some constraints. It is conservative when optimizing the code order, it does not unroll loops and it restricts the amount of memory the compiler can use.
- `-O3` This flag is more aggressive than `-O2`, it inherits everything that the `-O2` flag optimizes and lifts the restrictions such as reordering the code and can use as much memory as needed for the best performance.
- `-Ofast` This flag even more aggressive than `-O3` and inherits everything it does as well. It also enables some flags that would make our program become invalid. These flags include `-ffast-math` which introduces possible wrong floating point calculations and `-fallow-store-data-races` which enables the optimization where stores are allowed to have data races. Neither of these are an option for writing a matrix multiplication algorithm, as it is important that the output is exact.

With these optimization flags in mind we ended up choosing `-O3` as our optimization flag for several reasons. `-Ofast` was quickly removed as an option as it could possibly change the result of our implementation. As the benchmarking was run on a cluster, which will be introduced in section 6.2, memory was not an issue. Further more, `-O3` also enables loop unrolling, which has great potential in the performance of our implementation, making `-O3` even more favorable than `-O2`.

A second flag that was used was `-DNDEBUG`, which defined `NDEBUG`. This flag simply disabled every `assert` within a program. It also disables every Eigen assertions as well. Having assertions in a program were good for debugging purposes, but once everything worked correctly these only slowed down the program and were therefore removed when benchmarking.

Lastly, the third notable flag which was used was the `-march` flag. The `-march` flag specifies to the compiler which system's processor architecture it should compile

to. This allows the compiler to use architecture specific features, such as unique instruction sets, to generate faster code.

4.5 Evaluation method

When it comes to the evaluation of the results, an empirical evaluation was used. The evaluation was performed by comparing the benchmarks of our implementation with varying parameters to the benchmarks of state-of-the-art matrix multiplication libraries. The option to use a few different matrix multiplication libraries is supported, such as Intel's Math Kernel Library (MKL) [42] and AMD's AMD Optimizing CPU Libraries (AOCL) [43]. Due to some difficulties, neither of these two libraries ended up being used on the cluster. Another option would have been to use OpenBLAS [44] which is also supported, but this had similar issues as AOCL and were in the end not used for the benchmarks, which will be further detailed in section 7.3. As such, when benchmarking on the cluster, we instead compared our implementation to Eigen's own matrix multiplication implementation. The evaluation method were performed by running benchmarks on the functions that were implemented in this project, but also benchmarking the matrix multiplication implementation that Eigen provides. Further details of what the specific experiments that were run can be found in chapter 6. The results along with the standard deviation will be reported, as this will give a good idea of the running time consistency of the program. In places where it gives good information, both the execution time as well as how well it scales with different parameters will be looked into.

5

Implementation

This chapter details how the different components of Pagh's algorithm have been implemented. The most important libraries used by the implementation are also briefly described in this chapter.

5.1 Libraries

The header-only linear algebra library Eigen [45] was used for its `Array` type to store and load data and to perform element-wise operations of rows of arrays. Eigen also provides implementations of matrix multiplication as well as bindings to BLAS implementations that can be compared to when benchmarking.

The FFTW library [46] was used to compute the FFT and IFFT of the rows of doubles and complex doubles respectively. FFTW is the default option to compute the FFT and IFFT, but there are other options as well. While the following two libraries were not used for benchmarking, they were implemented for systems that can run them. On systems with AMD processors, AOCLs implementation of FFTW can be used instead for potential increases to performance. On systems with Intel processors, Intel's MKL library [42] is likely to be faster than FFTW and thus can be used to compute the FFT and IFFT on these systems.

OpenMP [47] was used to parallelize the compression and decompression algorithms.

5.2 Algorithm implementation

Pagh's algorithm is composed of two separate algorithms, `compressed_product` and `decompress_matrix`. For both these algorithms we have implemented one sequential and one parallel function. Thus there are four functions total: `compressed_product_seq` and `decompress_matrix_seq` as well as `compressed_product_par` and `decompress_matrix_par`. All four functions operate on different row-major Eigen arrays of dynamic size, storing doubles and complex doubles separately. The row length of the Eigen arrays storing complex doubles are $\frac{b}{2} + 1$ as opposed to the size b as described in section 3.4. The reason as to why this is will be explained in section 5.4. All of these functions have been written as template functions, allowing the use of hash functions with different types.

A link to the full implementations of the functions as well as the rest of the source code can be found on GitHub: https://github.com/Torphage/Master-Thesis .

5.2.1 Compress

The compression functions `compressed_product_seq` and `compressed_product_par` computes d sketches of the product of the input matrices M_1 and M_2 given the size n, the parameters b and d as well as some preallocated variables.

The compression functions are divided into two parts. The first part computes d Fourier transformed sketches of the product of the matrices M_1 and M_2. The second part computes the inverse Fourier transform of each of the d transformed sketches.

The first part of the function consists of two outer for loops, the outermost of which ranges over $t \in [d]$. In each iteration of the outermost loop, a sketch of $M_1 \cdot M_2$ is computed by the inner loop using the corresponding hash functions $h_1[t], h_2[t], s_1[t], s_2[t]$ that are stored in an object called `hash`. All the hash functions stored in the `hash` object are randomly generated before the execution of the function and the implementation and generation of the hash functions will be further detailed in section 5.3.

In each iteration k of the inner for loop, there are two for loops ranging over $i \in [n]$ that compute sketches of the column vector k of M_1 and the row vector k of M_2 respectively. A deviation from the algorithm presented in Algorithm 3 is the indexing of the matrix M_1. This deviation was performed in order to achieve a sequential memory access pattern. In Algorithm 3, M_1 is indexed column-wise since the algorithm computes a sketch of a column vector of M_1. Accessing the elements of M_1 column-wise will incur many cache misses since the matrix is stored in a row-major format. Instead, the matrix M_1 is transposed before executing the function and indexed row-wise inside the function.

To reduce memory usage, the array `pa` is used for the computations of both the sketches of the row vector of M_1 and the sketches of the column vectors of M_2. After a sketch of a row vector of M_1 is computed, the FFT of `pa` is computed and stored in another array. Afterwards, the elements of `pa` are set to zero and then the above mentioned steps are repeated for the corresponding column vector of M_2, with the result of the second FFT being stored a row below the result of the first FFT. The product of the two transformed sketches is then computed and added to the corresponding row of the array `p`. The code for the first part of the function can be seen in Listing 5.1.

Parallelizing the first part of the compression function was a non-trivial task. There are multiple for loops that can be parallelized with OpenMP in the code seen in Listing 5.1. The loops that calculate the sketches of the row and column vectors respectively can be parallelized, although doing so would not likely lead to a noticeable speedup since the granularity of each task would be too small. Doing this would also leave most of the code sequential which would limit the maximum theoretical speedup. This leaves the two outer for loops as candidates for parallelization

```
for (int t = 0; t < d; t++) {
    for (int k = 0; k < n; k++) {
        pa.setZero();
        for (int i = 0; i < n; i++) {
            pa(hash(hash.h1, t, i, b)) += (2 * hash(hash.s1, t, i, 2) - 1)
                * m1(k, i);
        }
        fft(fft1, 0, 0);

        pa.setZero();
        for (int i = 0; i < n; i++) {
            pa(hash(hash.h2, t, i, b)) += (2 * hash(hash.s2, t, i, 2) - 1)
                * m2(k, i);
        }
        fft(fft1, 0, b / 2 + 1);

        p.row(t) += out.row(0) * out.row(1);
    }
}
```

Listing 5.1: Computing d transformed sketches of $C = \mathrm{AB}$

The outermost loop ranges over the values $t \in [d]$, with d being a user-specified parameter. Every iteration of the outer loop is independent to all other iterations of the loop. Parallelizing the outer loop is simple, however, since the value of d is independent of the inputs and specified by the user, all the threads available on the system might not be utilized depending on the value of d. Thus the inner loop ranging over values $k \in [n]$ was parallelized instead. As mentioned above, each iteration of the k-loop calculates a sketch of an outer product. These calculations are done independently of each other, but the sum of the sketches must be computed to compute a sketch of $M_1 \cdot M_2$. To compute the sum of the sketches an OpenMP reduction clause was added. Instead of letting every parallel thread write to the same matrix at the same time, the OpenMP reduction would create one local variable for each thread that each thread could write to.

Adding an OpenMP reduction for a variable is equivalent to adding a critical section around the line of code where the variable is updated, thus causing there to be a section of code which cannot be parallelized. This can be avoided entirely by allocating significantly more memory to the function and having each thread write each intermediate result to a new location in memory. We tested this but found it to be significantly slower than the already existing solution. This however is something that we would have liked to experiment with more given the time.

The code for computing the IFFT is simple. It consists of calculating the IFFT for each individual row in the results from the first part of the function. Parallelizing this part of the function was trivial, the iterations of the loop were completely independent of each other.

5.2.2 Decompress

The functions `decompress_matrix_seq` and `decompress_matrix_par` closely follow the definition of the decompression algorithm provided in Pagh's paper which can be seen in Algorithm 3.

While it is possible to decompress each element in the compressed product independently from each other, the two `decompress_matrix` functions decompressed every element in the compressed product and return the complete matrix multiplication. Every element in `decompress_matrix` performs two steps. The first step is to compute d estimations of the values of the resulting element. From these estimations, the median value is chosen as the estimation of the resulting element in the output matrix. It was initially expected that Eigen provided their own implementation of median, but as this was not the case an own median funciton was written. For calculating the median in expected linear time a simple algorithm using the `nth_element` function from the C++ standard library is used. The function `nth_element` partially sorts the an array such that the element at the specified location is equal to the element that would be there if the array was fully sorted. If the array is of odd length it is enough to call `nth_element` and indexing to the middle element. Computing the median of an array of even length requires additional computations, and thus the decompression function runs substantially slower for even values of the parameter d. Information about possible improvement to this function can be read in section 7.5. The code used to calculate the median can be seen in Listing 5.2.

```
std::nth_element(xt.begin(), xt.begin() + d / 2, xt.end());
median1 = xt[d / 2];

if (d % 2 != 0) {
    c(i, j) = median1;
} else {
    std::nth_element(xt.begin(), xt.begin() + (d - 1) / 2, xt.end());
    median2 = xt[d / 2 - 1];
    c(i, j) = (median1 + median2) / 2.0;
}
```

Listing 5.2: Median computation

Parallelizing the `decompress_matrix_seq` function was simple. It was enough the parallelize the outer most loop of `decompress_matrix_par`, because it looped over n there was not a similar problem to when looping over the outer most loop of `compressed_product_par`. There were two options that ended up performing very similar to each other. The first option was to store the vector of d estimators in a two dimensional matrix, allowing each iterations of the outer loop to store their own values at the same time. The second option was to dynamically allocate the vector of d estimators for each thread, this did not cause a problem because d will always be tiny. The dynamically allocated option had the advantage that each thread worked with the exact same vector for every iterations, and it performed a few less arithmetic operations than the first option. In the end `decompress_matrix_par`

ended up dynamically allocating the d estimations of the result.

5.3 Hashing

In total three hash functions were implemented. The hash functions are all implemented as different classes but they share a specific structure. These classes made use of template parameters, allowing each hash functions to input and output different types. The coefficients for a set of hash functions, e.g. $h_1[1], \ldots, h_1[d]$, are stored in an object of a hash class as a two-dimensional Eigen array. The type of the Eigen array are specified by one of the template parameters. The coefficients in row t are used to hash keys with the hash function $h_1[t]$. Whenever a hash object is created, all of the arrays of coefficients are initialized with values.

To call the hash function, all of the hash classes implemented the same function, in this case the `operator()` function. Writing the classes in this way allowed `compressed_product_par` and `decompress_matrix_par` to make use of any of the three hash functions without any modification for any specific hash function.

5.4 FFT and IFFT

There were two separate implementations that were created for the FFT and IFFT routines, one built for using MKL, while the other used FFTW which itself supported other libraries like AOCL. By smart usage of preprocessor directives, only one of the MKL and FFTW implementations will be compiled depending on the compiler flags. The implementations share all the functionality with the other and can be used in the same way not matter which one is compiled.

As briefly mentioned above, the row-size of the Eigen arrays storing complex doubles is $\frac{b}{2} + 1$. The reason for this is that the transforms that are computed in the compression functions are the real FFT and real IFFT. When the input array of size b to the FFT contains only real numbers, the value of the output at index i will be the conjugate of the value at index $b - i$. Thus, only the non-redundant values need to be computed and stored, resulting in less computations being performed and $\frac{b}{2} + 1$ elements in the output.

In both of the implementations a form of plan for executing the FFT and IFFT calculations need to be created. These plans can be reused for multiple calculations, as the input and output arrays can be specified in the calculations step. As such it is enough to create at most two different plans. For FFTW, one plan need to be created for FFT and one for IFFT. For MKL it is enough to only create one plan for both forward and backward FFT, as the function to perform the FFT are different for forward and backwards FFT.

6

Experiments

This chapter details the experiments that were performed on the algorithm as well as provides their results. As this algorithm has some unique properties, some experiments were be performed in order to see the practicality of the algorithm. As such, several benchmarks were performed on a different types of inputs to see how well our implementation performed.

6.1 Experimental setup

The experiments were computed by a combination of many smaller parts. Most of the experiments that were performed were benchmarks, therefore benchmarks are the first of the experiments that will be introduced.

6.1.1 Benchmarking suite

The benchmarking suite were written in C++ using a variety of libraries. Every benchmark that is reported in this paper were run using this benchmarking suite. Every benchmark is initially started by reading a user-written comma-separated values (CSV) file, where all the configurations on what benchmarks that are being run, as well as all their parameters, are specified. Every matrix that were used was generated before every benchmark were started. Each function that can be benchmarked has a helper function that will preallocate the necessary storage variables and then run the benchmark.

The actual function that benchmarked runs several unique loops when benchmarking. The first loop only runs for a few iterations, these results are not used in any resulting calculations, as they act as warm-up iterations. After these warm-up iterations the function is benchmarked and measured. Various calculations such as mean, median and standard deviation are performed on these measured samples before they together with the raw measurement data are returned to the user.

In addition to the benchmarking suite, a python script was created that limited one process per CPU core. This was an important part of benchmarking, as otherwise multiple threads would have to share the same hardware which would result in a slow-down. Limiting to what threads to use was performed using *taskset*, a Linux command that allows the user to specify exact what threads on the CPU that should be run.

6.1.2 Computational cluster

The benchmarks were ran on a cluster that made use of the resource management system Simple Linux Utility Resource Management (SLURM) [48]. SLURM allows users to submit jobs to the cluster by requesting some of its partitions' resources. On the cluster, a parameter in SLURM was set that limited how many threads of the CPU that you could allocate per job. While the option to allocate the whole CPU was available, it was not a feasible option as it was regularly used by other users and it would have blocked everyone for a significantly long period of time. Therefore, we allocated as much of the CPU that were possible without using all of it. The reason that much of the CPU was allocated was that it decreased the possibility that another user got allocated a thread that shares a core with you. From these allocated threads, only the threads that had unique cores were chosen to benchmark on.

6.2 Hardware

The benchmarks were performed on a computational cluster from Chalmers that we got access to. The cluster, which used SLURM, has two partition where it was possible to run 80 CPU threads in parallel, allowing us to make great use of parallelism. The CPU that was used was an AMD EPYC 7451 24-Core Processor, there were also 512 GB of shared RAM available to use. The CPU is multithreaded, allowing two threads per core, effectively allowing the use of 48 CPU threads. Additionally, this CPU is set up in a symmetric multiprocessing configuration, allowing two processors to link up, again doubling the effective number of available threads to 96 threads. As the cluster was a shared resource, it was prone to varying running times, as there were no guarantee of what workload the CPU was executing beforehand. Additionally, other parts of the CPU may be used by other users whilst you are using it, putting even more load on the CPU. Therefore, it was important to take enough samples when benchmarking in order to gain more consistent results. This also meant that it was mostly used when few users were using it.

6.3 Experimental details

Several types of experiments were benchmarked. In Pagh's paper [1] he discusses a certain type of application for his algorithm. It is possible to estimate the covariance of a certain type of randomized matrix, where the input matrices are dense and the resulting output matrix is sparse. As this is the type of matrices that Pagh's algorithm excels at compared to many other matrix multiplication implementations, its practicality will be measured and investigated. However, that experiment will not run faster than executing our implementation with two dense matrix with the same parameters of n, b and d. Estimating the covariance will however give a more accurate result compared to multiplying two dense matrices, and thus allows b and d to be smaller while still achieving high accuracy. Pagh also mentioned another property in his paper, that it is possible to decompress specific values of the total

matrix product from the compressed product. It may not always be desired to achieve the whole matrix product, but only specific values. Since Pagh provided an unbiased estimator for each element as stated in section 3.4, the bound also holds for each element. However, this will not be experimented on since it is sufficient to only look at the experiments on `compressed_product_par` as it will dominate the running time. The theoretical running time for extracting only a few elements includes the whole running time of `compressed_product_par` and a proportion of `decompress_matrix_par` that is equal to the proportion of how many out of all elements that will be extracted.

For benchmarking the algorithm, the two input matrices were instead two uniform matrices with uniform sparsity. These two matrices' result might not give an accurate result, but they are only meant to serve the purpose of benchmarking our implementation. These matrices are completely random and may not represent real world applications. However, as the execution time is generally not dependent on the sparsity of the output, this will not affect the results. The case where the execution time can depend on the output sparsity is when the output matrix only contains zeros. Many calculations within the these benchmarks may not give accurate results, as it is the performance that is the key takeaway.

There are several benchmarks that will be performed. The benchmarks will generally be performed on five different functions, two of these are simply the sequential versions of two parallel versions. The first function that will be benchmarked is Eigen's built-in matrix multiplication implementation. The second and third function to be evaluated will be `compressed_product_par` and `decompress_matrix_par`. The last two functions to be evaluated are the sequential versions of the last two functions, namely `compressed_product_seq` and `decompress_matrix_seq`. These five functions will be benchmarked with varying parameters such as number of cores as well as values of n, b and d. Eigen does not make use of either b or d, as such Eigen will be excluded from benchmarks where any of these are incremented. Additionally, neither `decompress_matrix_par` nor `decompress_matrix_seq` will be benchmarked against any change of b, as those functions' execution time are not affected by b. Each of these parameters will be benchmarked separately, with the additional benchmark where `compressed_product_par` will be benchmarked with increasing values of both n and b at the same time. The reasoning for this lies in what b represent. Due to b representing a time/accuracy parameter it, some tests where both n and b increased simultaneously would represent the real usage in some capacity. The choice of having b increase an equal amount as n was arbitrary, but by combining the values from increasing n and b separately and together, some form of behavior of the algorithm can be extracted. Even though d also needs to be increased as both n and b increases, it does so at a much slower rate and therefore d will not be increasing as along n and b in a benchmark. Further more, while all benchmarks have used the Multiply-shift hashing function unless specified, both fully random hash and tabulation hash were also tested for `compressed_product_par` and `decompress_matrix_par`.

When measuring the accuracy between two matrices there are many options for this operation. It was decided to calculate the Frobenius norm of the difference of two matrices, which gives the distance between the two matrices. To get the accuracy

of the implementation, the experiment will calculate the distance the approximated covariance matrix and the real covariance matrix.

6.4 Results

The results have been presented with multiple graphs, with each graph modifying one, or possibly multiple, variables each. For each graph, the input variables of the benchmarks are all listed, such that the benchmarks can easily be replicated. All benchmarks are measured by the arithmetic mean, and each graph that measures time also includes a standard deviation error bar. For the times where the standard deviation can not be seen, it is simply because it is small enough to not render. In every graph except the ones measuring the different hashing functions, the multiply-shift hash function were used.

The first benchmarks tested the speed-ups of different functions when using different amount of cores, the results can be viewed in Figure 6.1. The execution time of these benchmarks can be seen in Figure 6.2. When benchmarking `compressed_product_par` and `decompress_matrix_par` with one core, `compressed_product_seq` and `decompress_matrix_seq` were used respectively instead. The functionality of these are equivalent, but both `compressed_product_seq` and `decompress_matrix_seq` are specifically optimized with one core in mind. The graphs took a dive at 40 cores, this seemingly strange behavior will be further discussed in section 7.4. The following remarks on the results from running multiple cores are using the data from 30 cores and less, as they behave as expected. In the figures it can be seen that `decompress_matrix_par` seem to behave quite linearly. The other function, `compressed_product_par` performed much worse than its counterpart with an increasing number of cores. Eigen performed better than `compress_product_par`, but is not near-linear as `decompress_matrix_par` seems to be from these few samples. Additionally, from the graph with the execution time it was observed that even the combined values of `compressed_product_par` and `decompress_matrix_par` were significantly faster than Eigen for the specified parameters.

The next following benchmarks demonstrates how these three functions behave when incrementing n, b and d independently. These tests were not be benchmarked with the either of the sequential versions, `compressed_product_seq` and `decompress_matrix_seq`.

The second benchmark measures the remaining three functions when incrementing n. The results from these benchmarks can be seen in two figures. In Figure 6.3 the total execution time for each benchmark is measured. The same data were also used to create Figure 6.4, but here the vertical axis represents how much the execution time grows when compared to the leftmost benchmark, where $n = 10\,000$. These two functions also includes two additional functions that were benchmarked. As brought up in section 6.3, one of these represent when b is increasing alongside n, whilst the last one represent the total execution of our implementation, including both `compressed_product_par` and `decompress_matrix_par`, when $n = b$.

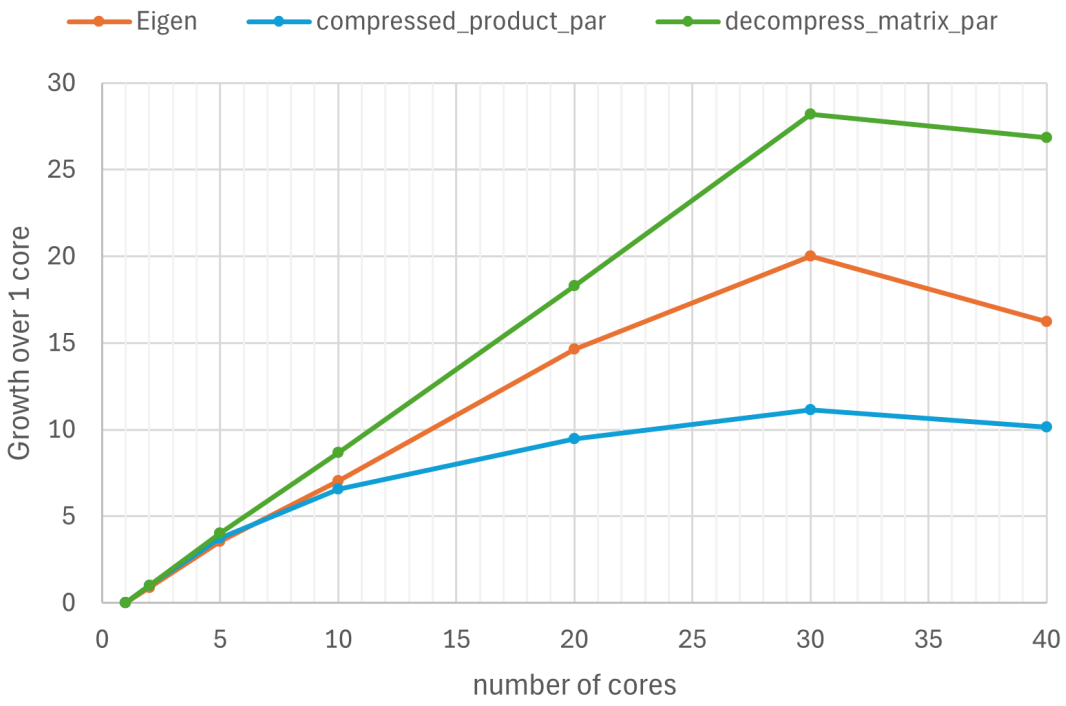

Figure 6.1: Benchmark of different functions with varying number of cores. The parameters here are $n = 20\,000$, $b = 20\,000$, $d = 11$.

In these results with varying n, it can be seen that Eigen has both a slower execution time and an execution time that grows worse when increasing n, even with the experiment where `compressed_product_par` were benchmarked when both n and b were increasing.

The third set of benchmarks that were performed was with varying values of b. In these benchmarks, a larger range of values for b than the normal 10 000 to 30 000 range with an increment of 5 000 was need. The results ended up using values of b that ranged from 30 000 to 100 000 with an increment of 10 000. The other parameters were $n = 20\,000$, $d = 11$ and the number of cores used where 40. The execution time of these benchmarks can be seen in Figure 6.5 whilst how well it scales for values of b can be seen in Figure 6.6. It can be seen that `compressed_product_par` grows slowly, and that it is not linear.

The forth benchmark that were performed were when d was the varying parameter. The parameter d was also tested for for a wide range of inputs just like b. For these tests the parameters n, b as well as the number of cores used were constant. The parameters that were used were $n = 20\,000$, $b = 20\,000$ and the number of cores used were 40. For these tests, only choices of d that were odd numbers were chosen. More on why this was the case will be further discussed in section 7.5.

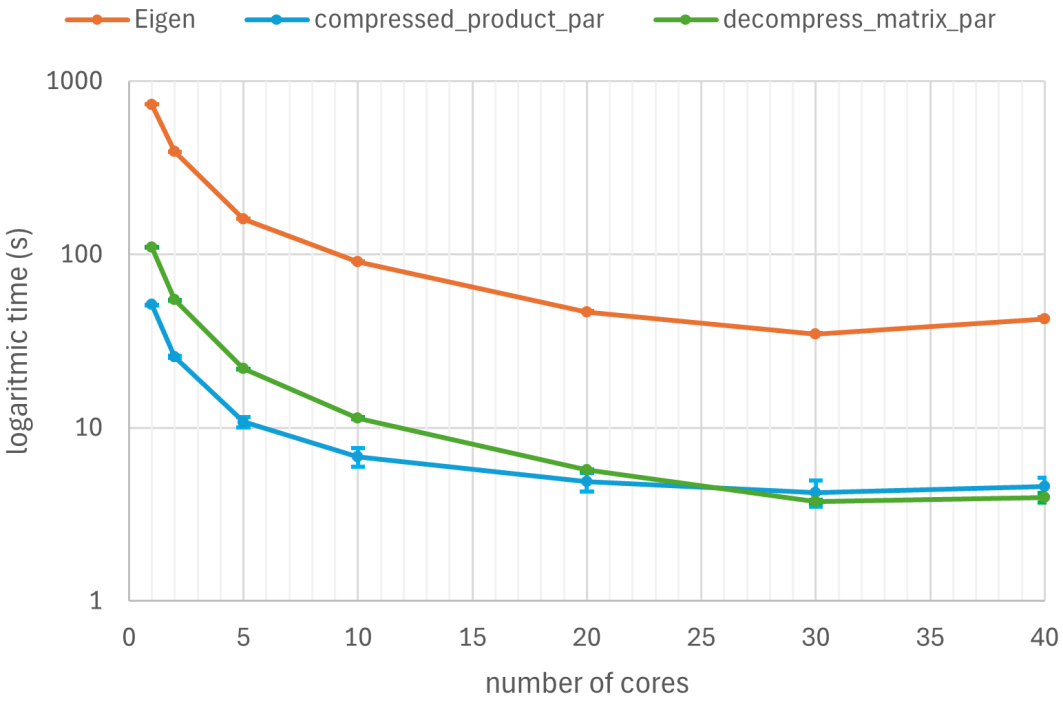

Figure 6.2: Execution time of the different functions with the parameters $n = 20\,000$, $b = 20\,000$, $d = 11$.

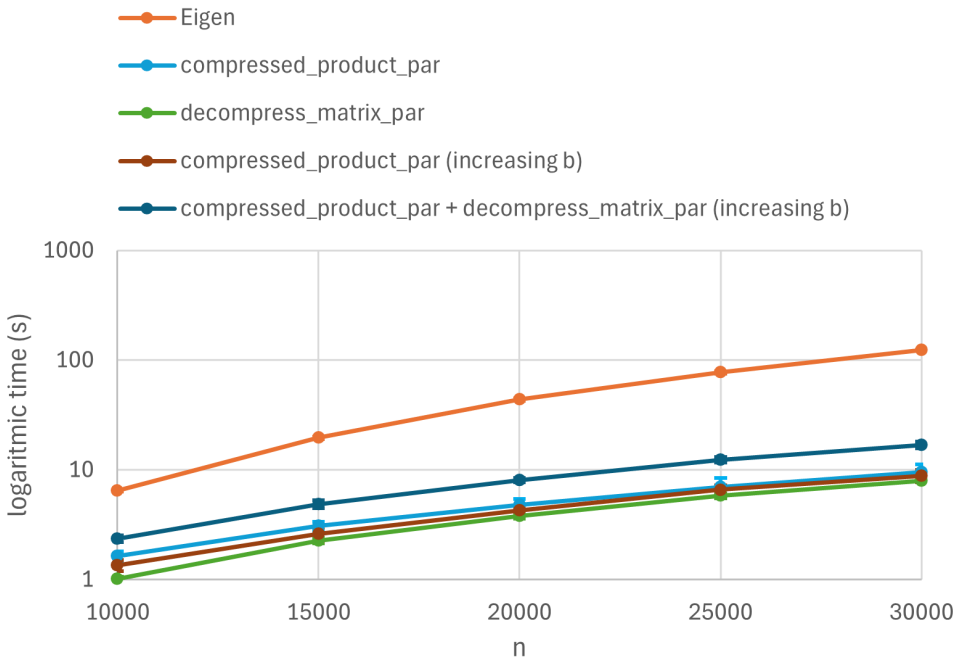

Figure 6.3: Benchmark of different functions when n is incrementing, the parameters that are used are $b = 20\,000$ (except when otherwise specified), $d = 11$, number of cores = 40.

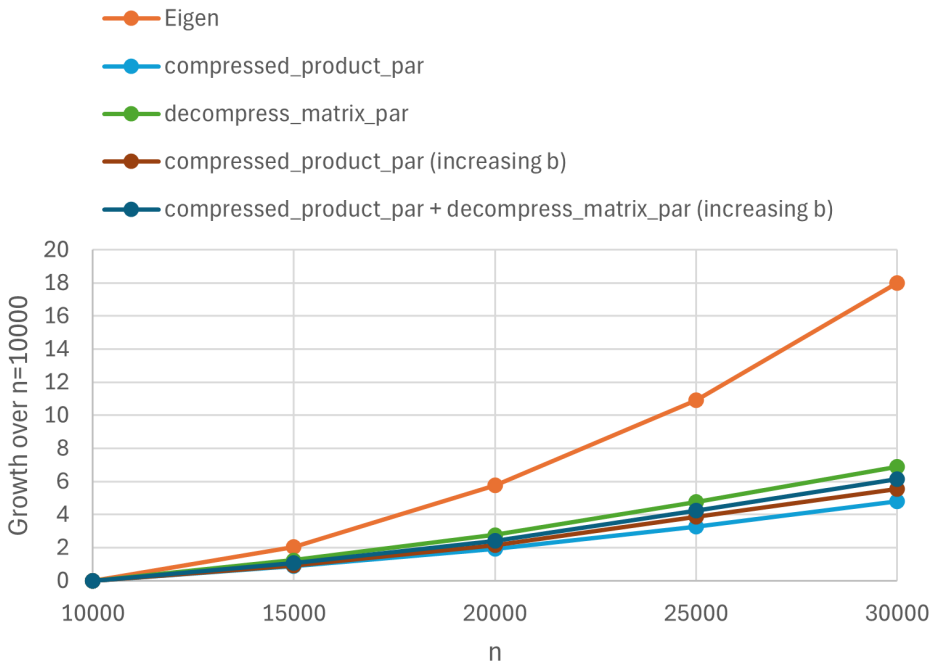

Figure 6.4: Another interpretation of the benchmark where n is incrementing, this shows how the algorithms scales with n compared to when $n = 10\,000$. The input parameters were $b = 20\,000$, $d = 11$ and number of cores used were 40.

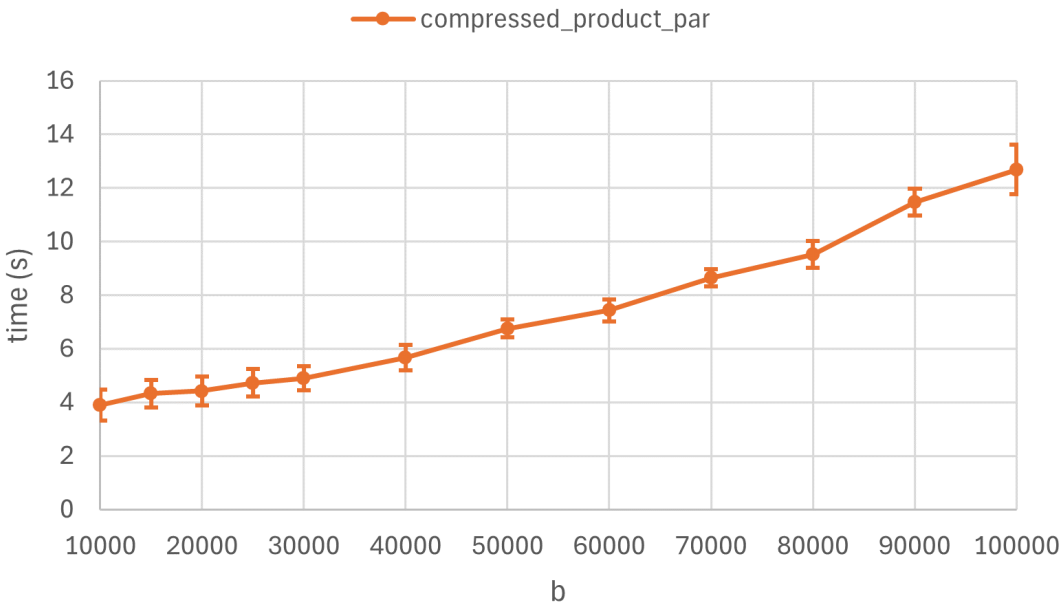

Figure 6.5: Benchmark of different functions when b is incrementing, the parameters that are used are $n = 20\,000$ (except when otherwise specified), $d = 11$, number of cores = 40.

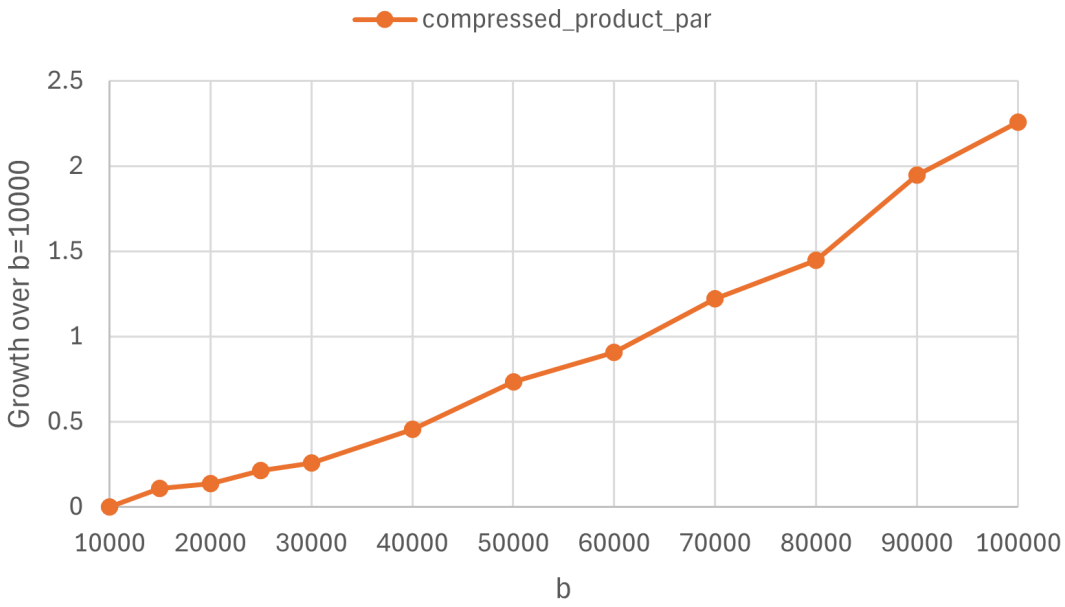

Figure 6.6: Another interpretation of the benchmark where b is incrementing, this shows how the algorithms scales with b compared to when $b = 10\,000$. The input parameters were $n = 20\,000$, $d = 11$ and number of cores used were 40.

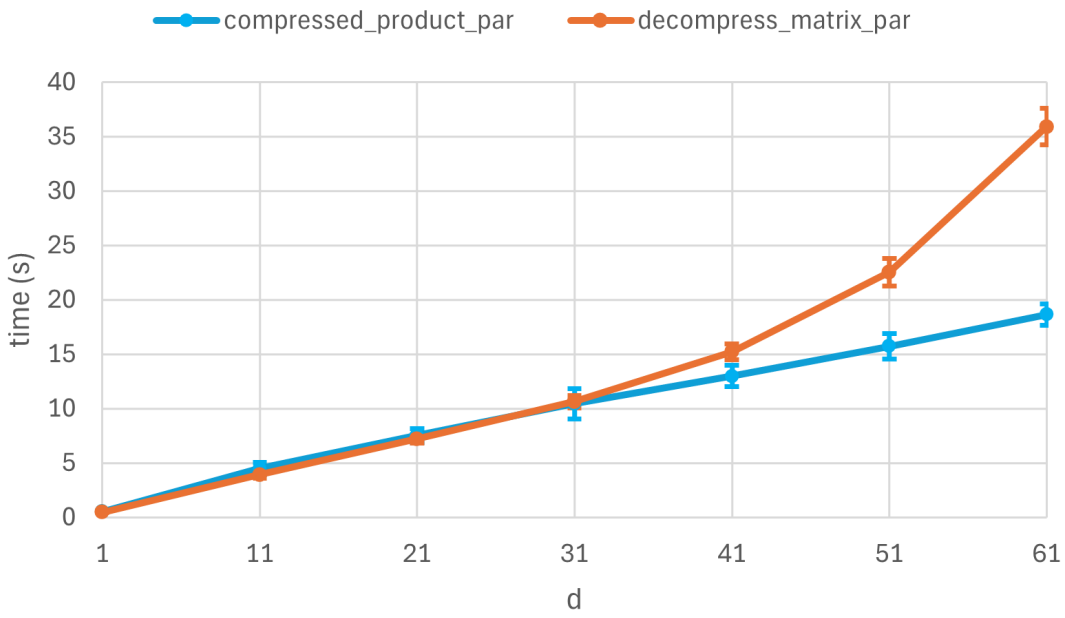

Figure 6.7: Benchmark of the execution time for the different functions with varying values of d.

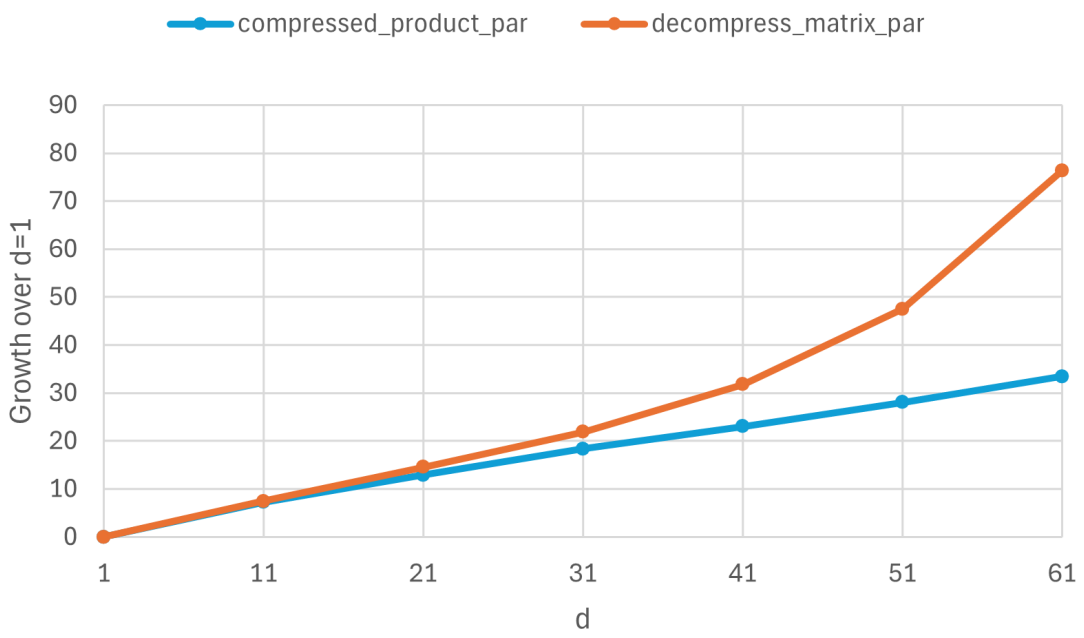

Figure 6.8: Benchmark of the scalability for the different functions with varying values of d.

Another benchmark that was performed was comparing the hashing functions that have been implemented. These functions were tested on both `compressed_product_par` and `decompress_matrix_par` with varying n. The other parameters for these benchmarks were set to $b = 20\,000$, $d = 11$ and the number of cores used were set to 40. The execution time for these hashing functions for `compressed_product_par` can be seen in Figure 6.9, here it can be seen that the fully random hash and the multiply-shift hash seems to be equally fast, whilst the tabulation hash is the slowest of the three. How well the hashing functions scale in `compressed_product_par` can be seen in Figure 6.10. The results for `decompress_matrix_par` can be found in Figure 6.11 and Figure 6.12. Whilst fully random hashing is faster for these values of n, it scales equally well with the tabulation hash for different values of n, where multiply-shift scales the best.

Another experiment that were performed were by testing the accuracy of the implementation when calculating the sample covariance matrix. The distance between two matrices will be used for measuring as described in Section 6.3. The experiments were tested by increasing b and d individually to see how much they each affect the distance. The results for increasing values of b can be seen in Figure 6.13. It can be seen that b needs to be increased a lot in order to get decent result, the accuracy gets diminishing results when increasing b by a set constant each time.

When instead testing the accuracy for only increasing values of d it can be seen in Figure 6.14 that much smaller values of d is needed compared to b. The graph does seem to behave in a similar way to Figure 6.13 in that increasing the values of d seems to have diminishing results in the accuracy.

Figure 6.9: Benchmark of the different hash functions with varying values of n on `compressed_product_par`.

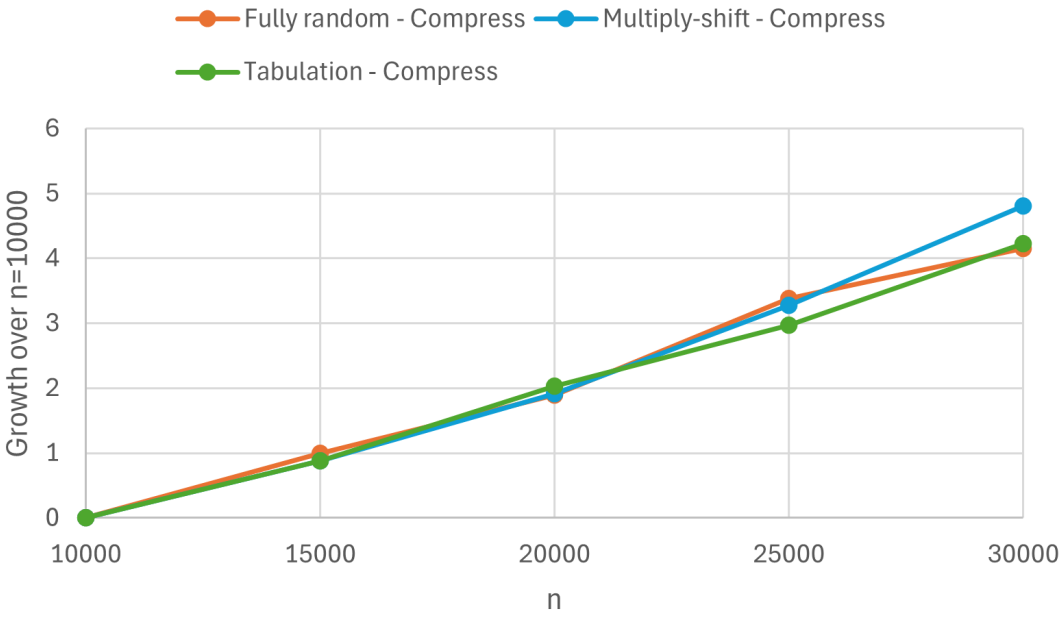

Figure 6.10: Benchmark of the scalability of the different hash functions with varying values of n on `compressed_product_par`.

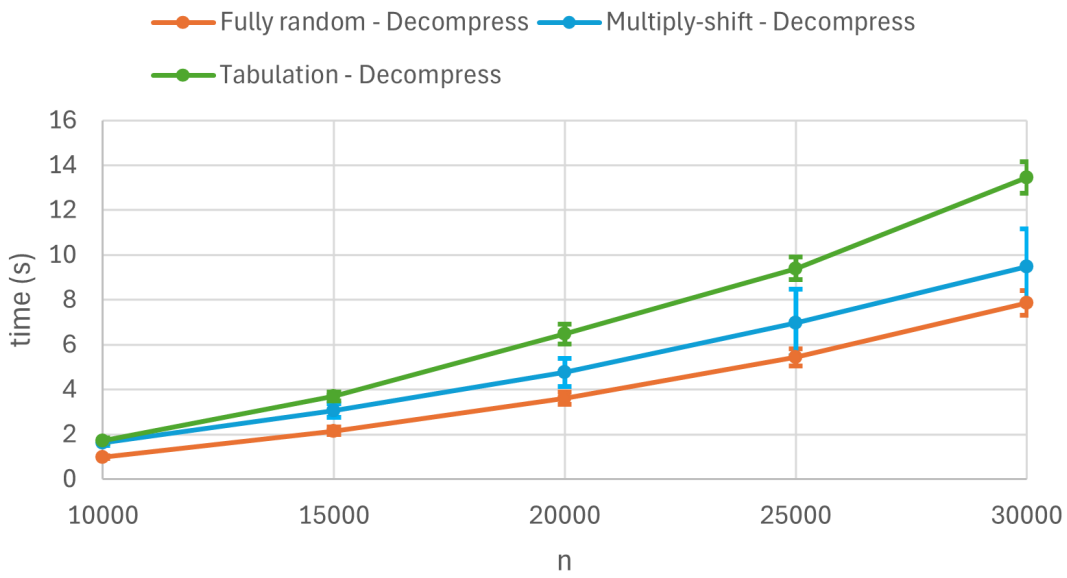

Figure 6.11: Benchmark of the different hash functions with varying values of n on `decompress_matrix_par`.

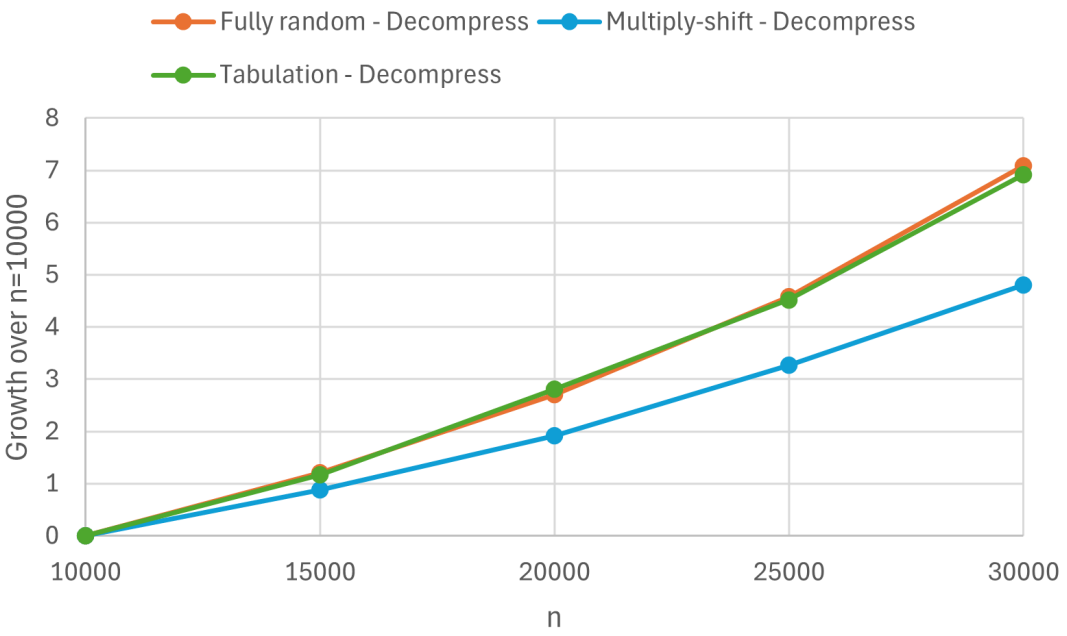

Figure 6.12: Benchmark of the scalability of different hash functions with varying values of n on `decompress_matrix_par`.

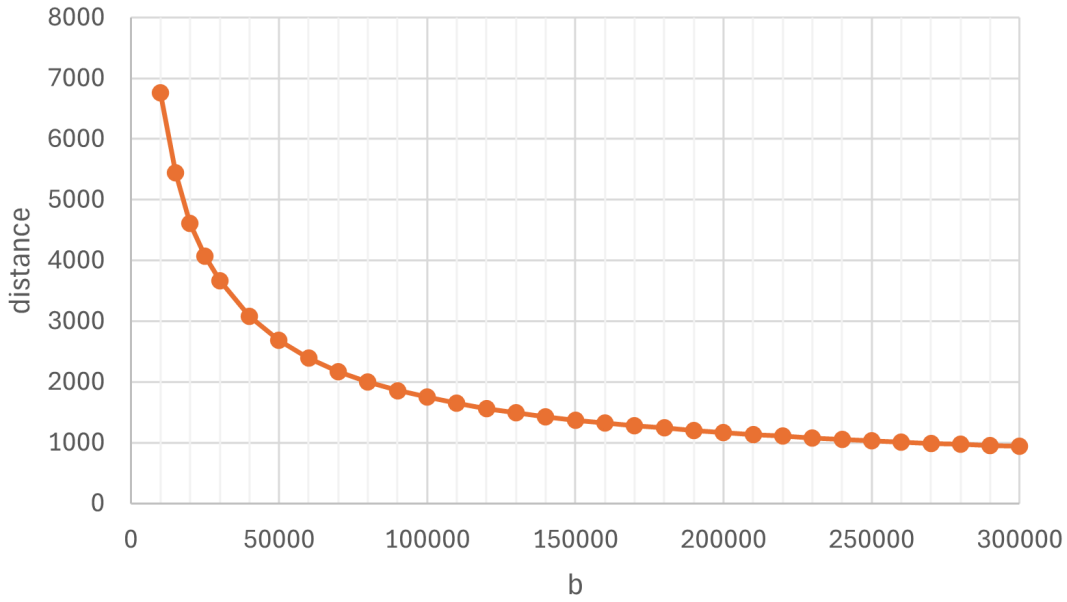

Figure 6.13: The accuracy of the approximated covariance when increasing b. The other values that were used were $n = 20\,000$ and $d = 11$

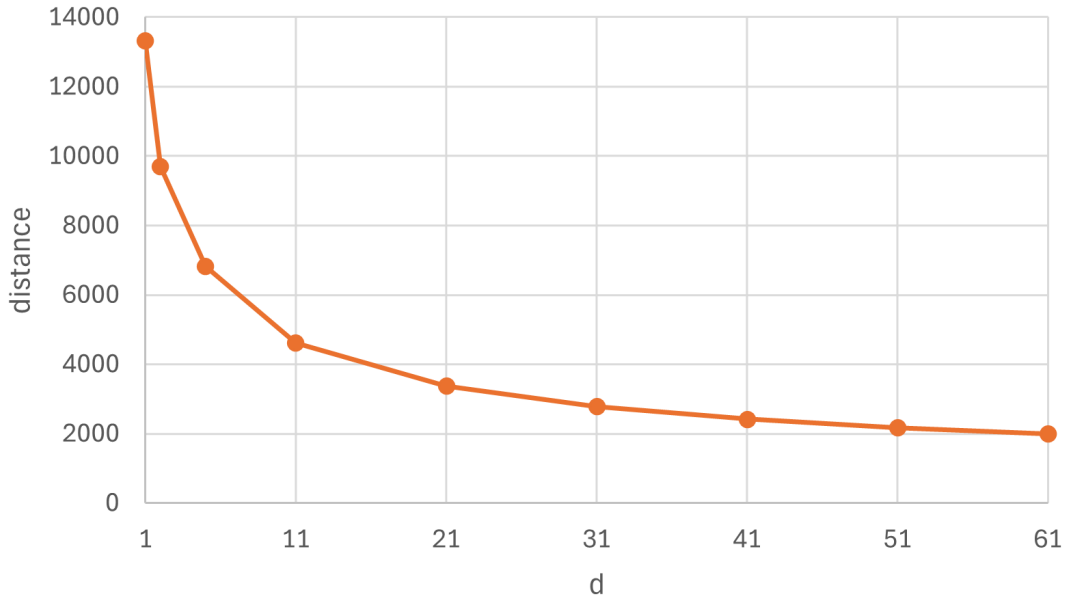

Figure 6.14: The accuracy of the approximated covariance when increasing d. The other values that were used were $n = 20\,000$ and $b = 20\,000$

7

Discussion

This chapter includes the discussions of the results as well as possible problems and limitations that were encountered during the project. Additionally, future possible work is also discussed.

7.1 Discussing the results

The results from the benchmarks have generally been seen as good results, though there are some places that could be further improved. There are also some unexpected results, with a clear example being the benchmarked that tested varying amounts of cores, from Figure 6.1. Due to how not only `compressed_product_par`'s and `decompress_matrix_par`'s results worsened when comparing 30 to 40 cores, but also Eigen's own matrix multiplication implementation, it seems likely that this strange behavior can be traced to the machine that it is run on. This behavior will be further discussed in section 7.4. By looking at Figure 6.1 again it can be seen that `decompress_matrix_par` is near-linear for 40 cores and below, which makes it almost as good with multiple threads as it could be. On the other hand, `compressed_product_par` does not parallelize nearly as well. This is likely to be largely influenced by Amdahl's law [49], which states that the possible speedup a program can achieve with multiple parallel processes is proportional to how much of the code is parallelizable.

Then the benchmarks tested how Eigen, `compressed_product_par` and `decompress_matrix_par` behaved for varying value of n, b and d. For different value of n it can be seen that `compressed_product_par` and `decompress_matrix_par` together outperforms Eigen in both execution time and scalability. However, these two benchmarks are only tested when $d = 11$, as it would be impossible to test every single combination of n, b and d. This means that the values that were tested does not have a guarantee on accuracy. Depending on the sought after accuracy of the result, higher values of d as well as b may be chosen, meaning that Eigen may prove faster in those circumstances.

For the benchmarks of varying values of b there are some observations that will be mentioned here. FFT routines are in general sensitive to the length of the input vector, which is b in Pagh's algorithm, and the running time may vary by quite a lot between tiny changes to b. This was one of the main reasons for testing `compressed_product_par` for larger values of b than the normally used 10 000 to 30 000, and the

other reason being that b had a much smaller performance impact compared to n and b. In Figure 6.5 and Figure 6.6 it can be seen that b is not linear, even though it increases slowly.

The benchmarks on varying values of d shows, as would be expected, that d affects the execution time of the program the most out of n, b and d. Compress `compressed_product_par` behaves linear, as should be expected due to its definition. As can be read in section 5.2.1, `compressed_product_par` the only usage of d is to determine the number of sketches. Thus it makes sense that `compressed_product_par` behaves linearly. However, `decompress_matrix_par` seem to behave exponential on the recorded samples. As can be read in section 5.2.2, `decompress_matrix_par` is defined by `nth_element` that has an expected execution time that is linear. By definition, `decompress_matrix_par` is expected to behave linear assuming that `nth_element` is a the dominant region of the function. This however is clearly not the case as `decompress_matrix_par` seems to grow exponentially.

The last benchmarks were the benchmarks that tested each hash function for both `compressed_product_par` and `decompress_matrix_par`. For both of the functions it is clear that tabulation is the slowest of the three benchmarks, though it is unclear how well it scales for `compressed_product_par` with the data that was recorded. For `compressed_product_par` all the hashing functions seem to scale equally the given parameters for those tests. For `decompress_matrix_par` it is more obvious that whilst the fully random hashing function is the fastest initially, multiply-shift will eventually perform better with larger values of n.

The results from the experiments that were performed on the accuracy of the implementation with varying values of b and d were as expected. From the two graphs it seems likely that both will reach a distance that is close to zero given large enough parameters. In order to get the best results, increasing both b and d together will yield even better results, as the graphs shows that both b and d individually have diminishing results on the accuracy. For the tests of increasing values of b, a wider range of values were used compared to the benchmarks. This was chosen to get a better understanding of how the graph behaves.

7.2 Dynamic allocation

In high performance computing it is common practise to preallocate all locations in memory that a function will operate on. Dynamic allocation is slow and performing it repeatedly will often slow down a program. For our algorithm it seems to be faster to dynamically allocate some of the vectors within `compressed_product_par`. While this is not optimal, the benchmarks on our local computers and the cluster are consistent. The reason for this is not clear, as both L1 and TLB cache-misses have both been analyzed with *perf*, with neither of the results showing any results that indicates that cache-misses were the cause of the problem.

One possible explanation for the poor parallelism which was partially solved by each thread allocating it's own memory inside of the function is the occurrence of false sharing. Before we changed it so that each thread allocated it's own memory,

each thread instead operated on specific rows of preallocated arrays. If multiple threads operate on rows that are small enough to share the same cache line, then the actions of one thread might cause the other threads to need to reload the cache line, thus incurring a performance penalty. The increase in performance achieved by each thread allocating it's memory in the function might indicate that this is something which occurred earlier, since with the changes the threads operate on different locations in memory.

7.3 Libraries

We had previously stated that our implementation should be benchmarked and compared to MKL and AOCL, or possibly even OpenBLAS, but neither of these ended up being used.

The initial plan was to benchmark with MKL, as it is supposed to be the fastest library available. However, as the hardware we were given had an AMD CPU instead of an Intel CPU, this was no longer an option and we had to settle for using AOCL instead. This turned out to not work out either. For reasons that are not yet clear, AOCL did not perform well on the cluster, even outside of our implementation. We have managed to install AOCL on a separate computer and have gotten significant speedups when using the AOCL library compared to using Eigen's matrix multiplication and FFTW's built in routines. From this result we know that AOCL can work well on our implementation, despite it not performing well on the cluster. Additionally we tried using OpenBLAS as an alternative to Eigen's matrix multiplication. This ended up performing worse than Eigen's built-in matrix multiplication, with OpenBLAS being about 30% slower than Eigen's matrix multiplication. It was signifiantly faster than how AOCL peformed on the cluster, but still worse than Eigen.

Since MKL, AOCL and OpenBLAS were either not an option anymore or performed worse than eigen, we ended up using Eigen's built in matrix multiplication. FFTW's FFT routines also ended up being used since MKL and AOCL which both provided their own implementations either were not available or performed worse than FFTW.

7.4 Computational cluster

When running on the computational cluster there were 48 physical cores available. 40 of these were used for almost all the benchmarks, except for the experiments that involved using a set number of cores. It can be seen in Figure 6.1 that 40 cores seems to perform worse than expected. This observation was something that was noticed very late into the project and is not something that is trivially solved. On the cluster during the tests, no other people were running anything during the these tests. While 40 cores were the most amount of cores that were benchmarked, there were always between 42 and 46 cores that were explicitly specified by *taskset*. Which 40 cores that were used cannot be known due to `omp_set_num_threads` only specifying to OpenMP that any 40 cores should be used.

7.5 Median calculation

A large part of `decompress_matrix_seq` and `decompress_matrix_par` goes to calculating the median of a vector. The median is rather simple in theory, as discussed in section 5.2.2 it is enough to partially sort an array. If the size of the vector, d, is an odd number, then no more calculations will need to be performed. To calculate the median of an even sized vector, more calculations will need to be performed. One possibility is to perform another `nth_element`, to partially sort the first half of the vector. This would be rather costly, and as such decompress with an odd number of elements are faster than even elements by quite a margin, even when the even numbers are slightly larger than the odd numbers. This is the reason for only choosing odd values on d when benchmarking, as otherwise the performance difference between the values of odd and even d would have likely affected the benchmarks to a large degree.

One possible improvement for even values of d could have been to simply call `max_element` on the first half of the vector. This could in theory work, given that the first half of the vector is smaller than the already calculated `nth_element`. This possible improvement have not been extensively tested nor has it been benchmarked on the cluster.

Further more, it may also be that calling `nth_element` may not be optimal. The reasoning is that the median will only be calculated by vectors of length d, and that it is sufficient for the algorithm that $d = \mathcal{O}(ln\ n)$ to achieve strong guarantees [1]. Therefore, since it is sufficient for the algorithm that d is quite small, another function that suits median calculations for smaller sized vectors may be better suitable for Pagh's algorithm compared to `nth_element`.

7.6 Future work

There are multiple ways that this project can be expanded upon. One possible extension would be to investigate using Fast Walsh–Hadamard transform (FWHT) instead of FFT. FWHT have some interesting properties that might be beneficial in this algorithm. Another extension can be found in Pagh's paper [1] where he introduced a method for extracting significant coefficients of the output matrix, *FindSignificantEntries*.

Our implementation is only implemented to perform square matrix multiplication, but it is likely possible to generalize this algorithm to work for non-square matrices as well.

Out implementation were only implemented with the intention to run on the CPU, but it may be possible to make use of a GPU to further improve upon the implementation. In the field, many algorithms are constructed to specifically run on the GPU, and it may be possible that this algorithm can make use of the GPU as well.

As already discussed in section 7.5, further improvements can be made on the median calculation part of `decompress_matrix_par`.

8

Conclusion

In this thesis, Pagh's algorithm for compressed matrix multiplication has been implemented in the programming language C++. The performance of the implementation has been extensively studied through multiple facets. The degree of parallelism available in Pagh's algorithm has been studied and compared to the matrix multiplication implementation of the linear algebra library Eigen. We found that our implementation scales better with the number of cores than that of Eigen.

In addition, the running time of the implementation has been evaluated for input matrices of increasing sizes and for different values of the parameters determining the accuracy of the approximation. The results of these tests have been presented in graphs to show the behavior and scaling of the two components of Pagh's algorithm, the compression and decompression function. The running time of Eigen's matrix multiplication has been tested for matrices of increasing size and the results have been compared to the running time of our implementations. The comparison between the running times of Eigen's matrix multiplication and our implementation of Pagh's algorithm confirms that Pagh's algorithm scales better than standard matrix multiplication methods.

The running time of the implementation has also been measured for the following classes of hash functions: fully random hashes, multiply-shift hashes and tabulation hashes. It was found that fully random and multiply-shift hashing were relatively equivalent for the compression function whilst fully random hashing was faster for the decompression function. Tabulation hashing was slower than the other classes of hash function in all tests.

We have also applied our implementation to the problem of calculating sample covariance matrices and have measured the accuracy of the approximations from Pagh's algorithm for this problem. The accuracy has been measured for varying different values of the parameters b and d. The observed behaviour for both of the parameters b and d was a sharp increase in accuracy for increases at lower values of b and d and the growth decreasing more and more as the value of b and d increased.

Bibliography

[1] R. Pagh, "Compressed matrix multiplication," *ACM Transactions on Computation Theory (TOCT)*, vol. 5, no. 3, pp. 1–17, 2013.

[2] V. Strassen *et al.*, "Gaussian elimination is not optimal," *Numerische mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

[3] V. Y. Pan, "Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations," in *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (SFCS)*, IEEE, 1978, pp. 166–176.

[4] V. Strassen, "Relative bilinear complexity and matrix multiplication," *Journal für die reine und angewandte Mathematik*, pp. 406–443, 1987.

[5] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," in *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, 1987, pp. 1–6.

[6] V. V. Williams, Y. Xu, Z. Xu, and R. Zhou, "New bounds for matrix multiplication: From alpha to omega," in *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, SIAM, 2024, pp. 3792–3835.

[7] V. Y. Pan, "Fast feasible and unfeasible matrix multiplication," *arXiv preprint arXiv:1804.04102*, 2018.

[8] L. S. Blackford, A. Petitet, R. Pozo, *et al.*, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[9] R. Yuster and U. Zwick, "Fast sparse matrix multiplication," *ACM Transactions On Algorithms (TALG)*, vol. 1, no. 1, pp. 2–13, 2005.

[10] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978.

[11] R. R. Amossen and R. Pagh, "Faster join-projects and sparse matrix multiplications," in *Proceedings of the 12th International Conference on Database Theory*, 2009, pp. 121–126.

[12] J. Gao, W. Ji, F. Chang, *et al.*, "A systematic survey of general sparse matrix-matrix multiplication," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–36, 2023.

[13] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2020, pp. 766–780.

[14] P. Drineas, R. Kannan, and M. W. Mahoney, "Fast monte carlo algorithms for matrices i: Approximating matrix multiplication," *SIAM Journal on Computing*, vol. 36, no. 1, pp. 132–157, 2006.

[15] L. J. Guibas, D. E. Knuth, and M. Sharir, "Randomized incremental construction of delaunay and voronoi diagrams," *Algorithmica*, vol. 7, no. 1-6, pp. 381–413, 1992.

[16] N. Pham and R. Pagh, "Fast and scalable polynomial kernels via explicit feature maps," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2013, pp. 239–247.

[17] O. A. Malik and S. Becker, "Low-rank tucker decomposition of large tensors using tensorsketch," in *Proceedings of the 31st Advances in Neural Information Processing Systems (NeurIPS)*, vol. 31, 2018.

[18] S. P. Kasiviswanathan, N. Narodytska, and H. Jin, "Network approximation using tensor sketching," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, 2018, pp. 2319–2325.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[20] J. Wacker, *Random features for polynomial kernels and dot product kernels*, `https://github.com/joneswack/dp-rfs`, 2023.

[21] M. Heideman, D. Johnson, and C. Burrus, "Gauss and the history of the fast fourier transform," *IEEE Applications of Signal Processing (ASSP) Magazine*, vol. 1, no. 4, pp. 14–21, 1984.

[22] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[23] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," in *Proceedings of the 9th Annual ACM Symposium on Theory of computing*, 1977, pp. 106–112.

[24] M. N. Wegman and J. L. Carter, "New hash functions and their use in authentication and set equality," *Journal of Computer and System Sciences*, vol. 22, no. 3, pp. 265–279, 1981.

[25] A. Pagh, R. Pagh, and M. Ruzic, "Linear probing with constant independence," in *Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC)*, 2007, pp. 318–327.

[26] A. Siegel, "On universal classes of extremely random constant-time hash functions," *SIAM Journal on Computing*, vol. 33, no. 3, pp. 505–543, 2004.

[27] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen, "A reliable randomized algorithm for the closest-pair problem," *Journal of Algorithms*, vol. 25, no. 1, pp. 19–51, 1997.

[28] M. Dietzfelbinger, "Universal hashing and k-wise independent random variables via integer arithmetic without primes," in *Proceedings of the 13th Symposium on Theoretical Aspects of Computer Science Grenoble (STACS)*, Springer, 1996, pp. 567–580.

[29] M. Thorup, "High speed hashing for integers and strings," *arXiv preprint arXiv:1504.06804*, 2015.

[30] A. L. Zobrist, "A new hashing method with application for game playing," University of Wisconsin, Tech. Rep., 1970.

[31] M. Pătraşcu and M. Thorup, "The power of simple tabulation hashing," *Journal of the ACM (JACM)*, vol. 59, no. 3, pp. 1–50, 2012.

[32] D. P. Woodruff *et al.*, "Sketching as a tool for numerical linear algebra," *Foundations and Trends in Theoretical Computer Science*, vol. 10, no. 1–2, pp. 1–157, 2014.

[33] P. Drineas and M. W. Mahoney, "RandNLA: Randomized numerical linear algebra," *Communications of the ACM*, vol. 59, no. 6, pp. 80–90, 2016.

[34] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Theoretical Computer Science*, vol. 312, no. 1, pp. 3–15, 2004.

[35] Catchorg, *Catch2*, `https://github.com/catchorg/Catch2`, version v3, 2024.

[36] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, vol. 42, ACM New York, NY, USA, 2007, pp. 89–100.

[37] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision," in *Proceedings of the USENIX Annual Technical Conference*, 2005, pp. 17–30.

[38] J. Weidendorfer, "Sequential performance analysis with callgrind and kcachegrind," in *Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing*, Springer, 2008, pp. 93–113.

[39] A. C. De Melo, "The new linux 'perf' tools," in *Slides from Linux Kongress*, vol. 18, 2010, pp. 1–42.

[40] A. Crolotte, "Issues in benchmark metric selection," in *Performance Evaluation and Benchmarking: First TPC Technology Conference (TPCTC)*, Springer, 2009, pp. 146–152.

[41] Google, *Benchmark*, `https://github.com/google/benchmark`, 2024.

[42] E. Wang, Q. Zhang, B. Shen, *et al.*, "Intel math kernel library," *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*, pp. 167–188, 2014.

[43] Advanced Micro Devices, Inc., *AMD Optimizing CPU Libraries (AOCL)*, Available online, Accessed on 27-Mar-2024, 2024. [Online]. Available: `https://www.amd.com/en/developer/aocl.html`.

[44] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013, pp. 1–12.

[45] G. Guennebaud, B. Jacob, *et al.*, *Eigen v3*, http://eigen.tuxfamily.org, 2010.

[46] M. Frigo and S. G. Johnson, "Fftw: An adaptive software architecture for the fft," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, vol. 3, 1998, pp. 1381–1384.

[47] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[48] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, Springer, 2003, pp. 44–60.

[49] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the Spring Joint Computer Conference*, 1967, pp. 483–485.