Engineering an Efficient Implementation of Pagh's Algorithm for Sparse Matrix Multiplication

Master's thesis in Computer science and engineering

SHUHAO GE

PAUL KLIEMANN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Engineering an Efficient Implementation of Pagh's Algorithm for Sparse Matrix Multiplication

SHUHAO GE

PAUL KLIEMANN

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Engineering an Efficient Implementation of Pagh's Algorithm for Sparse Matrix Multiplication
SHUHAO GE
PAUL KLIEMANN

© SHUHAO GE, 2025.
© PAUL KLIEMANN, 2025.

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

Master's Thesis 2025
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 2025

Engineering an Efficient Implementation of Pagh's Algorithm for Sparse Matrix Multiplication
SHUHAO GE
PAUL KLIEMANN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis explores approximate matrix multiplication through the algorithm proposed by R. Pagh in *Compressed Matrix Multiplication* (2013) and its implementation developed by J. Andersson and M. Karppa. We identify memory bandwidth as the primary performance bottleneck and implement support for handling sparse inputs and outputs.

We further introduce several optimizations to improve cache utilisation. While fitting the input data into the L3 caches of our CPU did not improve L3 hit rate, it did lead to the improvement in the L1 hit rate compared to the dense implementation. Our experimental evaluation demonstrates modest yet consistent runtime improvements, particularly on large-scale and sparse inputs, while retaining the mathematical guarantees of the original implementation.

Despite these optimisations, our implementations remain outperformed by the state-of-the-art libraries on all tested real-world datasets. However, our modifications reduce the asymptotic runtime of the compression step and significantly decrease memory usage when processing sparse matrices.

Based on our findings, we suggest potential directions for future research, including further optimization strategies. These should primarily focus on reducing memory traffic, as improvements in memory locality alone are unlikely to yield substantial gains.

Keywords: Matrix Multiplication, Sparse Matrix, High Performance Computing, Fast Fourier Transform, Hashing.

Acknowledgements

We would like to express our sincere gratitude to our supervisor, Matti Karppa, for his invaluable guidance and unwavering support throughout the duration of this project. As this research area was relatively new to both of us, his expertise, patience, and encouragement were instrumental in helping us understand the subject matter and successfully complete this work.

We are also grateful to Joel Andersson for his initial contributions to the `compmatmul` project, on which our implementation is built, as well as for his insightful feedback and technical suggestions during our development process.

We would further like to thank Samuel Kyletoft, Edvin Lunqvist Sternvik, Felix Hultin Morger, Geoff Carver, Martin Toremark and Simon Lanngren for their constructive comments and helpful discussions, which significantly contributed to the refinement of our ideas and approach.

In addition, Paul would like to thank Reglindis, Nietzsche, and Johanna for their companionship and support during the writing of this thesis.

Lastly, we extend our appreciation to our examiner, Peter Damaschke, for his thoughtful feedback and continued support throughout the thesis process.

Paul Kliemann & Shuhao Ge, Gothenburg, 2025-06-20

Contents

List of Figures **viii**

List of Tables **ix**

List of Acronyms **xi**

1 Introduction **1**
1.1 Motivation 2
1.2 Summary of Achievements 3
1.3 Limitations 4
1.4 Structure of the Thesis 4

2 Background **7**
2.1 Matrix Multiplication Algorithms 7
2.2 Sparse General Matrix-Matrix Multiplication Algorithms 9
2.3 Randomized Algorithms 11

3 Theory **13**
3.1 Matrices 13
3.2 Sparse Matrices 14
3.3 Transforms 20
3.4 Hash Functions 23
3.5 Sketching Techniques 26
3.6 Pagh's Algorithm 27

4 Methodology **31**
4.1 compmatmul 31
4.2 Minerva Cluster 32
4.3 Libraries and Tools 35
4.4 Datasets 36
4.5 Validation 37
4.6 Experimental Setup 38

5 Implementation **39**
5.1 Existing Implementation 39
5.2 Overview of Adaptations 41

5.3 Enhanced Compression 42
5.4 Enhanced Decompression 46

6 Results **51**
6.1 Computational Step Decomposition 51
6.2 Performance Characterization of the Compression Phase 53
6.3 Comparison Against Existing Libraries 60

7 Discussion **63**
7.1 General Analysis 63
7.2 Analysis by Variant 64
7.3 Discussion on Other Experiments 66
7.4 Theoretical Analysis 68
7.5 Potential Performance Optimizations 72
7.6 Future Research Directions 76
7.7 Numerical Accuracy 77

8 Conclusion **79**

Bibliography **81**

A Appendix 1 **I**
A.1 AI Usage I
A.2 Source Code I

List of Figures

3.1 Memory usage of different matrix storage formats as a function of matrix size and density. 19

6.1 Runtime breakdown of four computational steps across different matrix sizes. 52
6.2 Runtime proportions of substeps in the compression step across different implementations at matrix size $n = 2^{16}$. 54
6.3 Runtime performance of the compression step across different implementations across and matrix sizes (Normalized to V0 Baseline). 55
6.4 Roofline model of the compression step across different implementations at matrix size $n = 2^{16}$. 56
6.5 Data load distribution across the memory hierarchy of the compression step across different implementations and input sizes $n = 2^{14}$ to $n = 2^{17}$. 57
6.6 L3 cache miss rate of the compression step across different implementations and matrix sizes. 58
6.7 Runtime performance of the compression step across different inputs at matrix size $n = 2^{16}$. 59
6.8 Runtime performance of the compression step using V5 across different combinations of b_c and d_c (Normalized to V0 Baseline). 61
6.9 Logarithmic runtime performance of pure multiplication process across different libraries and datasets. 62

7.1 Intermediate bucket density d_{pa} as a function of input matrix density d_A. 75

List of Tables

3.1 Memory requirements for matrix storage formats storing a $n \times p$ matrix. 18
3.2 Total sizes of matrix storage formats storing a $n \times p$ matrix. 18
3.3 Time complexity of matrix format conversions. 19

4.1 LINPACK benchmark results for the Minerva cluster. 33
4.2 Bandwidth and arithmetic intensity characteristics across memory levels of the Minerva cluster. 34
4.3 Properties of medium-scale sparse matrices used in experiments: matrix dimension $n_A = p_A$, number of non-zeros N_A, density d_A, portion of empty rows r_0 and storage size in CSR/CSC format. 37
4.4 Properties of power-of-two dimension sparse matrices used in experiments: matrix dimension $n_A = p_A$, number of non-zeros N_A, density d_A, portion of empty rows r_0 and storage size in CSR/CSC format. . 37

7.1 Comparison of asymptotic runtime for original `comptmatmul` and our modifications in the compression phase. 68
7.2 Comparison of asymptotic runtime for original `comptmatmul` and our modifications in decompression phase. 69
7.3 Comparison of memory requirements across different `compmatmul` variants, in bytes. 69
7.4 Comparison of hashing and signings calls across different `compmatmul` variants. 71

List of Acronyms

AI	Arithmetic Intensity
API	Application Programming Interface
AVX-512	512-bit Advanced Vector Extensions
AVX2	Advanced Vector Extensions 2
BLAS	Basic Linear Algebra Subprograms
CCX	Core Complex
COO	Coordinate Format
CPU	Central Processing Unit
CSC	Compressed Sparse Column
CSR	Compressed Sparse Row
DFT	Discrete Fourier Transform
FFT	Fast Fourier Transform
FFTW	Fastest Fourier Transform in the West
FLOP	Floating-point Operation
FMA	Fused Multiply-Add
FP64	64-bit Floating Point (Double Precision)
FPGA	Field Programmable Gate Array
FWHT	Fast Walsh–Hadamard Transform
GEMM	General Matrix Multiplication
GFLOP	Giga Floating-point Operations per Second
GPU	Graphics Processing Unit

IDFT	Inverse Discrete Fourier Transform
IFFT	Inverse Fast Fourier Transform
IPC	Instructions Per Cycle
L1/L2/L3	Level 1/2/3 (Cache)
LINPACK	Linear System Package
MKL	Math Kernel Library
NumPy	Numerical Python
OpenMP	Open Multi-Processing
RAM	Random Access Memory
SciPy	Scientific Python
SIMD	Single Instruction Multiple Data
SpGEMM	Sparse General Matrix-Matrix Multiplication
STREAM	Simple Triad Memory Benchmark
TFLOP	Tera Floating-point Operations per Second
WHT	Walsh–Hadamard Transform

1 Introduction

Every day, vast quantities of high-dimensional data are processed across a wide range of scientific and engineering domains. Such data is often represented in the form of matrices, with **matrix multiplications** serving as a key primitive in numerous applications, from physical simulations and machine learning to statistical analysis and graph processing.

In many practical settings, these matrices exhibit structural properties that can be exploited to significantly improve computational efficiency. One such property is **sparsity**, which refers to the presence of a large number of zero-valued entries. Sparse matrices are ubiquitous in real-world problems and enable the use of specialized storage formats and optimized algorithms, dramatically reducing memory consumption and accelerating computation. They play a crucial role in several key domains:

- **Graph theory**: Adjacency matrices representing large-scale networks are typically sparse, since most nodes are connected only to a small subset of others [1].
- **Scientific computing**: In the numerical solution of partial differential equations (PDEs), discretized systems are often modelled using sparse matrices, with iterative solvers (particularly Krylov subspace methods) being widely used for their efficient performance [2].
- **Bioinformatics**: Gene expression datasets and genome assembly tasks frequently involve sparse representations due to the inherent sparsity in biological data [3].

Given the prevalence and importance of sparse matrices, developing efficient algorithms for operations involving them, especially matrix multiplication, has become a central challenge in modern computational science.

Formally, given $A \in \mathbb{R}^{n \times p}$ and $B \in \mathbb{R}^{p \times m}$, a matrix product $C = AB \in \mathbb{R}^{n \times m}$ is defined as:

$$C_{i,j} = \sum_{k=1}^{p} A_{i,k} B_{k,j}, \quad \text{for } i = 1, \ldots, n, \quad j = 1, \ldots, m$$

This foundational operation gives rise to a broad class of extended problems, influenced by constraints on matrix structure, optimization objectives, or computational resources. For example, specialized algorithms have been developed for square, diagonal, and sparse matrices, as well as for approximate, randomized, and blocked variants of matrix multiplication. Optimization goals may vary from minimizing theoretical time complexity to enhancing cache utilization and enabling parallel execution.

Among the various algorithmic approaches proposed for this problem, Pagh's algorithm stands out as a randomized method for approximate matrix multiplication. As a Monte Carlo algorithm, it offers probabilistic guarantees on error bounds while achieving reduced asymptotic complexity under favourable conditions. It compresses the input matrices into polynomials using 2-wise independent hash functions, performs efficient polynomial multiplication via the Fourier transform, and recovers an unbiased estimate of $C = AB$ using the inverse Fourier transform.

Building upon these foundations, this work presents an optimized C++ implementation of Pagh's algorithm, based on the `compmatmul` implementation and tailored specifically for sparse matrices. We introduce several performance improvements and conduct extensive benchmarking under realistic scenarios to evaluate the computational efficiency.

1.1 Motivation

The primary motivation for this work stems from the ubiquity of sparse datasets in real-world applications and the computational infeasibility due to prohibitive memory demands of dense matrix multiplication [4]. For example, a dense matrix with $n = 2^{16}$ rows storing double-precision floating-point values would require 34 GB of memory ($65,536 \times 65,536 \times 8$ bytes). This far exceeds the cache capacities of modern CPUs, necessitating frequent and costly data transfers from main memory. Although temporal locality in matrix multiplication allows for reuse of values across iterations, dense formats still waste significant resources on zero-valued elements.

Sparse storage formats, by contrast, compress data to store only non-zero entries, reducing memory footprint and mitigating cache misses. This makes them particularly suitable for large-scale computations involving sparse data.

Pagh's algorithm offers a potential path for efficient matrix multiplication, especially in the context of sparse matrices. The original implementation in `compmatmul` is designed for dense matrices. Adapting it to leverage sparse storage formats could yield possible performance improvements, by improving the L3 cache hit rate and reducing memory bandwidth, while enhancing compatibility with existing computational workflows that rely on sparse data representations.

1.2 Summary of Achievements

The following is a summary of the key results and insights obtained in this thesis:

- **Theoretical Foundations:** A comprehensive study of sparse matrix storage formats (CSR, CSC, COO) was conducted, laying the theoretical groundwork for adapting Pagh's algorithm to sparse settings. This analysis identified how these formats can be integrated with the compressed multiplication framework.

- **Performance Improvements:** Our improvements demonstrated meaningful performance gains over the dense version when applied to sparse matrices. These gains were primarily attributed to reduced memory bandwidth usage and better alignment with sparse data characteristics.

- **Cache Behaviour:** Despite the expected benefits of improved memory efficiency, no significant improvement in L3 cache utilization was observed on the Minerva Cluster. This suggests that current memory access patterns or other architectural factors may limit potential caching advantages.

- **Dependence on Matrix Characteristics:** Runtime was found to be highly sensitive to both the sparsity level and the distribution of non-zero elements across rows. Performance improvements became more evident as matrix density decreased, indicating that the algorithm is most effective for highly sparse inputs.

- **Comparison with Existing Libraries:** While the current implementation lags behind established libraries such as Intel MKL, Eigen and SuiteSparse, in terms of raw runtime performance, it extends the original formulation by supporting a broader range of sparse input and output formats. This compatibility opens up new avenues for integration into sparse computation pipelines and lays the groundwork for future optimizations targeting performance-critical components.

- **Potential Optimizations:** Several optimizations were introduced, particularly targeting the compression phase of the algorithm, resulting in asymptotic improvements. However, the decompression phase remains a bottleneck. Future work should focus on optimizing this stage, potentially through hybrid approaches or alternative data layouts.

Overall, our implementations successfully extends the applicability of compressed matrix multiplication to sparse settings while maintaining mathematical equivalence to the original formulation. Although computational efficiency improvements are limited in some contexts, the enhanced memory efficiency and format compatibility open new opportunities for practical deployment and further optimization.

1.3 Limitations

Our research focuses on optimizing sparse matrix multiplication for modern multi-core CPUs, particularly in the context of scientific computing applications. As such, the following limitations apply:

- **Matrix Size and Density**: Our study primarily focuses on large sparse matrices, with sizes ranging from $2^{13} \times 2^{13}$ up to approximately $2^{17} \times 2^{17}$. While smaller or denser matrices may also benefit from our approach, they are not the focus of this work.
- **Floating Point Values:** We focus on 64 bit floating point data. Other data formats, such as integers will not be considered.
- **Hardware Scope**: Our experiments and optimizations are conducted on multi-core CPU architectures supporting the x86-64 instruction set. We do not consider alternative computing platforms such as GPUs or processors based on different architectures (e.g., ARM).
- **Parallelism Model**: The implementation relies on a multi-threaded architecture using OpenMP. Therefore, distributed computing environments with multiple nodes are beyond the scope of this work.
- **Generalizability of Results**: The optimization and evaluations are based on the specific hardware configuration of the provided computing node. As a result, the observed performance may vary across different hardware setups due to differences in CPU architecture, memory hierarchy, or clock speed.

These limitations are necessary to maintain a focused and tractable research scope. However, they may limit the direct applicability of our findings to other hardware or software environments.

1.4 Structure of the Thesis

The thesis is organized as follows:

Chapter 2 provides an overview of the background and related work in the field. It introduces foundational concepts and reviews existing literature relevant to matrix multiplication and randomized algorithms.

Chapter 3 delves into the theoretical underpinnings of Pagh's algorithm and explores the properties of sparse matrices.

Chapter 4 outlines the methodologies employed in the implementation and evaluation. This includes the experimental setup, hardware environment, software tools and measurement protocols used throughout the study.

Chapter 5 details the implementation of key components in Pagh's algorithm. It also presents the modifications we introduced to adapt the algorithm for sparse matrices,

along with a theoretical analysis of efficiency.

Chapter 6 presents the experimental results during evaluation and benchmarking with analysis.

Chapter 7 and Chapter 8 provide further discussions and present the concluding remarks of this thesis, respectively.

2

Background

This chapter provides a comprehensive review of the foundational literature and key concepts relevant to the field. It examines key works on matrix multiplication. Additionally, this chapter discusses studies closely related to the work undertaken in this thesis, establishing the context and motivation for the research presented in the following chapters.

2.1 Matrix Multiplication Algorithms

Matrix multiplication is a fundamental primitive in numerous scientific and engineering applications, and its computational complexity has profound implications across fields such as numerical linear algebra, optimization, and theoretical computer science.

2.1.1 Theoretical Work

The classical algorithm requires $\mathcal{O}(n^3)$ operations to multiply two $n \times n$ matrices, and for decades this was considered optimal. However, in 1969, Strassen [5] challenged this assumption by introducing a recursive algorithm that partitions matrices into submatrices.

This reduction lowered the time complexity to $\mathcal{O}(n^{2.8074})$, marking a classical algorithm for matrix multiplication. His breakthrough initiated the study of the matrix multiplication exponent ω, defined as the infimum over all exponents such that two $n \times n$ matrices can be multiplied in $\mathcal{O}(n^{\omega+\epsilon})$ time for any $\epsilon > 0$. The presence of ϵ reflects that current algorithms only achieve this bound asymptotically; the existence of an algorithm with true $\mathcal{O}(n^{\omega})$ complexity remains unproven.

Over the following decades, successive advances progressively reduced this bound. In 1978, Pan improved the bound to $\omega < 2.795$ [6], initiating an iterative optimization process. This effort was later accelerated by Strassen's laser method [7], which improved the bound to $\omega < 2.48$, by a technique that strategically "focuses" on key components in the tensor decomposition to enhance efficiency. Further progress came with the algebraic framework introduced by Coppersmith-Winograd [8], reducing the bound to $\omega < 2.376$. Contemporary approaches, building on these foundations through tensor decomposition optimizations and combinatorial enhancements,

culminated in Williams et al.'s 2023 bound of $\omega < 2.371552$ [9].

While these theoretical improvements are of significant mathematical interest, they have limited applicability in practice. Strassen's algorithm, despite being the simplest among subcubic methods, often underperforms for moderate matrix sizes due to irregular memory access patterns and large constant factors. More sophisticated algorithms that yield tighter bounds on ω are even less practical, involving intricate constructions that incur enormous overhead [10]. This disconnect between asymptotic improvements and real-world efficiency has become a defining challenge in the study of fast matrix multiplication.

As a result, contemporary research proceeds along two complementary paths. One continues to pursue further reductions in ω by refining algebraic frameworks and exploring deeper structural properties of tensors. The other prioritizes the optimization of practical implementations, leveraging architecture-aware designs that exploit hardware parallelism (e.g., GPU tensor cores and distributed memory systems) and data structure optimizations to achieve real-world efficiency. These efforts, though differing in emphasis, collectively contribute to the broader objective of improving both the theoretical and practical aspects of fast matrix multiplication.

2.1.2 General Matrix Multiplication Optimizations

Optimizations to the **General Matrix Multiplication** (GEMM) primitive have served as a cornerstone in the evolution of matrix multiplication algorithms.

Formally, GEMM is represented as:

$$C \leftarrow \alpha AB + \beta C, \text{ where } A \in \mathbb{R}^{n \times p},\ B \in \mathbb{R}^{p \times m},\ C \in \mathbb{R}^{n \times m}$$

Here, α and β are scalars coefficients used to scale the input matrices before performing the update.

Early work by Bo Kågström et al. [11] established GEMM's central role in the BLAS standard, with the conventional exact $\mathcal{O}(n^3)$ implementation serving as baseline for both theoretical analysis and practical benchmarking. Research in this domain continues to evolve along two major dimensions: theoretical characterization and hardware-specific optimization. Among the latter, modern implementations employ layered optimization strategies to improve efficiency across different computational hierarchies.

At the microarchitecture level, Van Zee et al. [12] effectively leveraged small register tiles to reduce memory traffic. At the thread level, Intel MKL [13] implements nested parallelism using OpenMP tasks, efficiently distributing workloads across multiple CPU cores. At the accelerator level, NVIDIA cuBLAS [14] exploits tensor cores via warp-level matrix instructions, significantly accelerating dense matrix multiplications on GPUs.

This body of work collectively demonstrates that GEMM optimizations present a co-design challenge, requiring a careful balance between computational efficiency,

memory bandwidth, and parallelism. The interplay between these factors becomes particularly critical when extending GEMM to sparse operands.

2.2 Sparse General Matrix-Matrix Multiplication Algorithms

Sparse General Matrix-Matrix Multiplication (SpGEMM) operates on sparse matrices and optimizes by avoiding unnecessary computation and deploying efficient indexing and storage, while regular GEMM assumes dense matrices and performs all operations regardless of element value. As a result, SpGEMM requires more complex indexing and storage schemes but can significantly reduce computation time and memory usage for sparse data.

In this section, we take a broader view on the current research on SpGEMM from two perspectives, the theoretical one and the practical one, with detailed algorithms' examples and will discuss some examples of each.

2.2.1 Theoretical Work

The number of purely theoretical papers on SpGEMM is relatively small. Among them, we have selected a few representative works for reference.

Gustavson's algorithm [15], combines a combinatorial approach with fast rectangular matrix multiplication techniques. It follows a row-by-row SpGEMM formulation. This method effectively exploits the reuse of the output matrix's sparse structure when multiple multiplications share the same sparsity pattern (fixed positions of non-zero elements). Consequently, when A and B share identical sparsity patterns, C retains its sparsity pattern across successive multiplications, eliminating the need for recomputation and reallocation, thereby saving time and memory.

Yuster and Zwick [16] proposed an outer-product-based SpGEMM formulation. They evaluated the performance of OuterSPACE against Intel MKL on a multi-core Xeon CPU, revealing suboptimal efficiency due to limited data sharing on conventional hardware. Their approach does not assume any specific structure in the input matrices. Instead, they partition matrices into dense and sparse components, applying a fast algebraic algorithm to the dense portion and a naive sparse multiplication for the sparse part. The final result is obtained by summing these two components. Their method achieves an asymptotic time complexity of $\mathcal{O}(n^{2+\epsilon})$ when the number of non-zero elements $N \leq n^{1.14}$, where ϵ is a small positive term approaching zero.

2.2.2 Practical Work

Research has explored various SpGEMM implementations, adapting algorithms to different computing architectures, including CPUs, GPUs, and FPGAs. In this work, we focus on CPU implementations.

OuterSPACE [17] employs a outer-product SpGEMM formulation. When evaluated by performance against Intel MKL on a multi-core Xeon CPU, it revealed suboptimal efficiency due to limited data sharing on conventional hardware. To address this issue, OuterSPACE develops custom reconfigurable hardware, achieving performance improvements specifically tailored to its architectural design.

MatRaptor [18] explores the row-wise product approach after analysing the data flows of SpGEMM. It introduces C^2SR, a new sparse storage format designed to enhance memory bandwidth utilization. Leveraging both the row-wise product approach and the C^2SR format, MatRaptor achieves a speedup of 1.8 times and reduces power consumption by a factor of 7.2 compared to OuterSPACE.

2.2.3 State-of-the-Art Libraries

A number of state-of-the-art libraries provide highly optimized SpGEMM implementations. We will introduce some libraries widely used in both academic research and industry due to their performance and maturity, which will be considered baselines in the experimental evaluation conducted in this thesis.

Intel MKL [13] incorporates adaptive execution strategies that consider the sparsity pattern and dimensional characteristics of the input matrices. For example, it dynamically selects between row-wise and column-wise traversal strategies depending on the matrix shape and density distribution. Internally, it employs architecture-aware optimizations and multi-threaded execution paths based on OpenMP and Intel's Threading Building Blocks. Despite being a proprietary and closed-source library, Intel MKL remains a valuable benchmark for evaluating the performance of SpGEMM algorithms, particularly due to its efficient cache utilization and runtime tuning capabilities.

Eigen [19] is a lightweight, C++ library designed for flexibility and ease of use. It allows the construction of sparse matrices using the *triplet format*, where each non-zero element is stored as a tuple (i, j, v) representing the row index, column index, and value. This format is equivalent to the coordinate (COO) format, which will be introduced later. This format facilitates dynamic matrix assembly when the sparsity pattern is not known in advance. Internally, Eigen converts the triplet array into a compressed storage format (typically column-major) by performing a stable sort followed by index pointer computation. This compressed structure enables more efficient arithmetic operations, including sparse matrix-matrix multiplication (SpGEMM). Compared to high-performance libraries like Intel MKL, Eigen prioritizes usability and general-purpose applicability over low-level performance optimizations.

SuiteSparse:GraphBLAS [20] is a high-performance implementation of the GraphBLAS API, which provides an expressive linear algebra interface for implementing graph algorithms. It supports internal formats such as CSR and CSC and uses OpenMP to enable parallelism on modern multi-core architectures. Internally, the library performs runtime format selection and task decomposition: for instance, large matrices are partitioned into tiles, and threads are assigned based on estimated workload to ensure balanced task distribution and reduced contention. Additionally,

GraphBLAS supports user-defined semirings and operations, allowing a wide range of algebraic formulations. Its backend handles memory layout transformations and execution parallelism transparently, enabling both flexibility and high performance across diverse sparse workloads.

2.3 Randomized Algorithms

Randomized algorithms use probabilistic decision-making in the computational processes, offering novel optimization strategies for matrix computation challenges. These algorithms exhibit two key advantages in the context of both GEMM and SpGEMM. First, they enable complexity reduction by replacing deterministic decision-making with probabilistic selections, significantly lowering computational overhead. Secondly, they mitigate worst-case scenarios by transforming input-pattern sensitivity into controlled probabilistic events, thereby circumventing the worst-case complexity pitfalls often encountered in traditional algorithms. These properties are particularly valuable in large-scale matrix operations, where conventional approaches struggle with scalability.

Monte Carlo algorithms are randomized algorithms that guarantee bounded running time while producing correct results with a probability greater than a specified threshold. As such, they are particularly well-suited for approximation-tolerant applications. A notable example is the Monte Carlo matrix approximation algorithm proposed by Drineas et al. [21], which utilizes random projections to accelerate matrix multiplication. By selectively sampling matrix entries, this method achieves sublinear time complexity in high-dimensional GEMM problems, offering substantial speed-ups over classical deterministic approaches at the cost of a controllable approximation error.

3

Theory

This chapter explores the theoretical foundations of sparse matrices and Pagh's algorithm, which is the central focus of this thesis. It presents the core principles of the algorithm, including its theoretical correctness and time complexity. Additionally, it introduces relevant sparse matrix storage formats used in the implementation along with a comparative analysis.

3.1 Matrices

Matrices are typically stored as contiguous arrays in memory, where every element is placed in a sequential manner. To illustrate different matrix storage formats for matrices, consider the following example matrix $E \in \mathbb{Z}^{n \times p}$, where $n = 4$, $p = 3$:

$$E = \begin{bmatrix} 11 & 0 & 33 & 0 \\ 22 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

In **row-major** order, elements of each row are stored consecutively before proceeding to the next row:

$$E = [11, 0, 33, 0, 22, 0, 0, 0, 0, 0, 0, 0]$$

Conversely, in **column-major** order, elements of each column are stored sequentially before advancing to the next column:

$$E = [11, 22, 0, 0, 0, 0, 33, 0, 0, 0, 0, 0]$$

As every matrix element is explicitly stored, dense storage is inefficient for sparse matrices. It consumes the same amount of memory as a fully dense matrix of the same size, even when most entries are zero.

3.1.1 Stride

When extracting columns from a row-major $n \times p$ matrix A, the elements must be accessed at **strides** of n, leading to non-contiguous memory access. Specifically, to load column j, one must iterate over all rows i and access the following elements:

$$A[0, j], A[1, j], A[2, j], \ldots, A[p-1, j]$$

In memory, these correspond to the addresses:

$$\text{base} + j, \quad \text{base} + j + n, \quad \text{base} + j + 2n, \quad \ldots, \quad \text{base} + j + (p-1)n$$

where *base* is the starting memory address of matrix A. Since column access in row-major order involves strided memory access, it exhibits poor spatial locality, making it less cache-friendly.

Additionally, applying vectorized instructions to column-wise operations requires complex interleaving, reducing computational efficiency. While x86-64 architectures support gather and scatter instructions (e.g., `VGATHERDPS`, `VSCATTERDPS` in AVX2 and AVX-512), these are significantly slower than contiguous memory operations like loads and stores, due to their reliance on non-unit stride patterns and limited prefetch efficiency.

This issue is symmetric: attempting to access a row in a column-major matrix results in the same strided memory access pattern, leading to similar challenges.

3.1.1.1 Transposition

One approach to mitigating these inefficiencies is to transpose the matrix into column-major format before performing column-wise operations, or the opposite for row-wise operations. This transformation ensures that column elements are stored contiguously, improving cache efficiency and enabling more effective vectorization. This can be achieved by using cache-oblivious algorithms that run in $\mathcal{O}(n \cdot p)$ time, via a divide-and-conquer strategy [22].

3.2 Sparse Matrices

Efficient matrix storage plays a crucial role in large-scale computational problems. By reducing memory consumption and computational overhead, it not only enhances performance but also enables the handling of larger matrices. Sparse matrices provide a natural solution to this challenge by exploiting the predominance of zero-valued elements in many real-world datasets.

A matrix is typically regarded as **sparse** when the number of non-zero elements is considerably smaller than the total number of elements. In contrast, *dense* matrices contain relatively few zero entries.

To quantify density, consider a matrix $A \in \mathbb{R}^{n \times p}$. We define:

- **Number of non-zero elements**:

  $$N_A = \sum_{i=1}^{n} \sum_{j=1}^{p} \mathbb{I}(A_{ij} \neq 0),$$

  where $\mathbb{I}(\cdot)$ is the indicator function that evaluates to 1 when its condition is satisfied and 0 otherwise.

- **Density**:

$$d_A = \frac{N_A}{n \cdot p},$$

  representing the fraction of non-zero entries in A.

- **Number of non-zero elements in row i or column j:** For matrix A, $N(A_{i,:})$ and $N(A_{:,j})$ denote the number of non-zero entries in row i and column j, respectively.

- **Fraction of empty rows:** Let $r_0(A)$ denote the fraction of rows in matrix $A \in \mathbb{R}^{n \times p}$ that are entirely zero (i.e., contain no non-zero elements). It is defined as:

$$r_0(A) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}(A_{i,:} = \mathbf{0}),$$

  where $A_{i,:} \in \mathbb{R}^{1 \times p}$ denotes the i-th row of A, $\mathbf{0} \in \mathbb{R}^{1 \times p}$ is the zero vector of compatible dimension, and $\mathbb{I}(\cdot)$ is the indicator function, returning 1 if the condition inside is true and 0 otherwise.

For example, the matrix E in Section 3.1 exhibits the characteristics of a sparse matrix, as it contains only three non-zero elements: $N_E = 3$, $d_E = \frac{3}{12}$ and $r_0(E) = \frac{1}{3}$.

3.2.1 Storage Formats for Sparse Matrices

To efficiently represent and operate on sparse matrices, a number of specialized storage formats have been proposed, aiming to minimize memory usage and accelerate computation.

3.2.1.1 SciPy Sparse Arrays

Python's `scipy.sparse` [23] package is a widely used library in scientific computing for handling sparse matrices.

It implements various sparse matrix formats that only store the values of non-zero elements as well as additional **metadata** indicating their positions within the matrix. The key difference among these formats lies in how this metadata is structured and encoded, which affects storage size and the performance of various arithmetic and conversion operations.

Throughout this discussion, we will use the variable names defined in `scipy.sparse` to describe the components of each format.

Each format also has a stricter internal representation, referred to as the **canonical format**, which imposes structural constraints. Unless otherwise specified, we assume that input data conforms to this expected structure, as it simplifies implementation and avoids edge cases. All examples given in this section follow their respective canonical format.

3.2.1.2 Coordinate (COO) Format

The **Coordinate (COO) format** represents a sparse matrix using three arrays:

1. `data`: Stores non-zero values of the matrix. In canonical format, the entries are ordered by their row-major order and free of duplicates. It may include explicit zero values.
$$\texttt{data}(E) = [11, 22, 33]$$

2. `row`: Contains the row indices corresponding to each entry in `data`.
$$\texttt{row}(E) = [0, 1, 0]$$

3. `col`: Contains the column indices corresponding to each entry in `data`.
$$\texttt{col}(E) = [0, 0, 2]$$

Iterating over rows or columns with a non-canonical COO matrix requires searching the entire matrix, since corresponding elements might be distributed throughout the entire matrix. When the matrix is in canonical format, iterating over rows is easier, since elements are stored sequentially. However, finding the first element of a row requires a binary search, and iterating over columns still requires traversing the entire matrix.

3.2.1.3 Compressed Sparse Row (CSR) Format

The **Compressed Sparse Row (CSR) format** efficiently stores sparse matrices by grouping non-zero elements row-wise. It consists of three arrays:

1. `data`: Stores the non-zero values of the matrix in row-major order. In the canonical CSR format, entries are sorted by rows, and duplicates are not allowed.
$$\texttt{data}(E) = [11, 33, 22]$$

2. `indices`: Contains the column indices corresponding to each entry in `data`.
$$\texttt{indices}(E) = [0, 2, 0]$$

3. `indptr`: Stores the cumulative count of non-zero elements before each row. That is, `indptr`[i] equals the total number of non-zero entries in all rows before row i.
$$\texttt{indptr}(E) = [0, 2, 3, 3]$$

Since CSR stores row-wise data contiguously, iterating over rows is efficient, benefiting from spatial locality and reduced memory overhead. While, column-wise traversal is inefficient, as it requires searching all N indices, leading to increased computational complexity and memory access overhead.

3.2.1.4 Compressed Sparse Column (CSC) Format

The **Compressed Sparse Column (CSC) format** is structurally identical to CSR but stores elements column-wise instead of row-wise. It consists of three arrays:

1. `data`: Stores the non-zero values of the matrix in column-major order. In the canonical CSC format, entries are sorted by columns, and duplicates are not allowed.
$$\texttt{data}(E) = [11, 22, 33]$$

2. `indices`: Contains the row indices corresponding to each entry in `data`.
$$\texttt{indices}(E) = [0, 1, 0]$$

3. `indptr`: Stores the cumulative count of non-zero elements before each column. In other words, `indptr`[j] equals the total number of non-zero entries in all columns before column j.
$$\texttt{indptr}(E) = [0, 2, 2, 3, 3]$$

Similar to CSR, CSC enables efficient column-wise iteration, while row-wise access is computationally expensive due to scattered memory accesses.

Note that, a CSC matrix is equivalent to the CSR representation of its transpose. This property allows for efficient conversion between the two formats without modifying the underlying data layout, only the interpretation of rows and columns changes.

3.2.2 Format Comparison

While many other sparse matrix formats are employed in specialized scenarios, such as diagonal storage, which is optimized for matrices with most non-zero elements concentrated along a few diagonals [23], this work focuses on these three commonly used formats due to their widespread applicability and general-purpose efficiency.

The examples given in the previous sections show that E can be stored with a range of metadata, between 8 and 11 entries. We will now further formalise this observation. The memory requirements of each field within the compared formats, along with their respective maximum values, are summarized in Table 3.1.

Let v denote the size of each value field which we assume to be a 64-bit floating point number. Similarly, let i represent the size of metadata entries, which are usually 32-bit or 64-bit integers. We will assume these are 32-bit integers for now and revisit this assumption in Section 3.2.2.2. We have calculated the total memory requirements for each format in Table 3.2, and further visualised how they are related to matrix size and density in Figure 3.1.

From these observations we conclude the following:

Format	Array	Array length	Element size	Maximum value
Dense	data	$n \cdot p$	Value type	–
COO	data	N	Value type	–
	row	N	Signed Integer	$n-1$
	col	N	Signed Integer	$p-1$
CSR	data	N	Value type	–
	indices	N	Signed Integer	$p-1$
	indptr	$n+1$	Signed Integer	N
CSC	data	N	Value type	–
	indices	N	Signed Integer	$n-1$
	indptr	$p+1$	Signed Integer	N

Table 3.1: Memory requirements for matrix storage formats storing a $n \times p$ matrix.

Format	Total size	Total size with $v = 8$, $i = 4$
Dense	$n \cdot p \cdot v$	$n \cdot p \cdot 8$ bytes
COO	$N \cdot (v + 2i)$	$N \cdot 16$ bytes
CSR	$N \cdot (v+i) + n \cdot i$	$N \cdot 12 + n \cdot 4$ bytes
CSC	$N \cdot (v+i) + p \cdot i$	$N \cdot 12 + p \cdot 4$ bytes

Table 3.2: Total sizes of matrix storage formats storing a $n \times p$ matrix.

- **CSC and CSR vs. Dense Format:** When storing matrix A, CSC is more space-efficient when $d_A < \frac{2}{3} - \frac{1}{3n}$, while CSR becomes advantageous when $d_A < \frac{2}{3} - \frac{1}{3p}$.
- **CSC vs. CSR:** When $p = n$, the memory footprints of CSC and CSR are identical. However, CSC outperforms CSR when $p > n$, while the opposite holds true if $n > p$. This behaviour stems from the differing indexing strategies used in row-major and column-major formats.
- **COO vs. Dense Storage:** The COO format outperforms dense storage when $d_A < \frac{1}{2}$, due to the COO's structure requiring only two metadata entry fields and one value field.
- **CSR/CSC vs. COO:** CSR becomes more efficient when $d_A > \frac{1}{p}$, and similarly CSC outperforms COO when $d_A > \frac{1}{n}$. Thus, as matrix dimensions increase, the relative storage advantage of CSR and CSC over COO grows, making them increasingly favourable for large-scale sparse data.
- **Impact of the Size of Value Element:** The memory savings offered by sparse storage formats over the dense format become more pronounced as the size of individual value elements increases. Since the metadata overhead remains constant regardless of the size of value element, compression efficiency is higher for 8-byte floating-point numbers than for 4-byte ones.

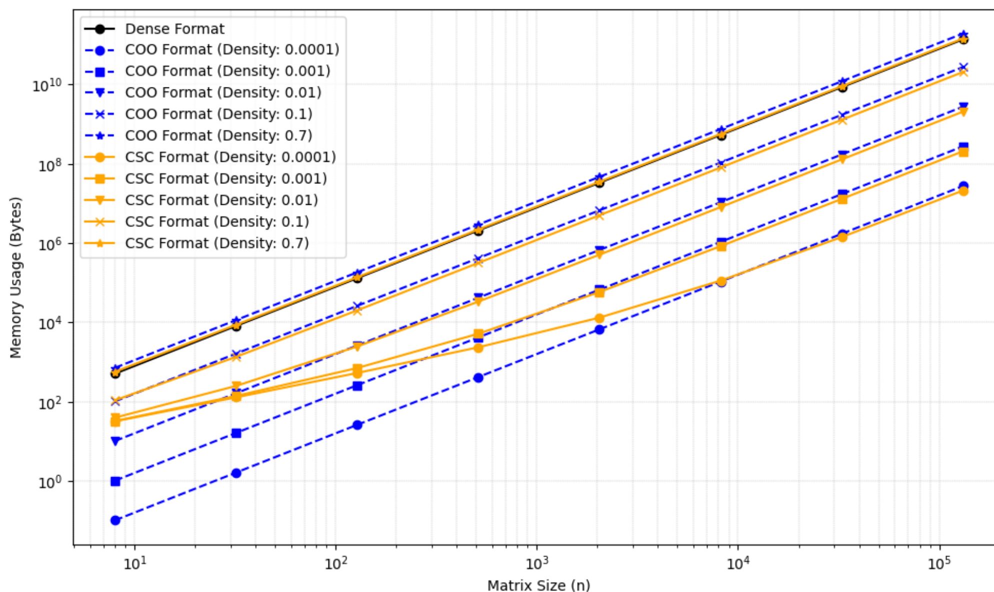

Figure 3.1: Memory usage of different matrix storage formats as a function of matrix size and density.

None of the sparse matrix formats discussed are lossy, which allows for the exact reconstruction of the original matrix. All numerical values are preserved without approximation. However by applying a preprocessing step that removes very small entries, for instance based on a chosen threshold. This results in a sparser matrix, which may then be stored more efficiently. Several methods have explored such strategies [24], [25], but a detailed discussion of these techniques is beyond the scope of this thesis.

3.2.2.1 Conversion between Formats

When performing frequent row accesses on a CSC matrix or column accesses on a CSR matrix, it may be beneficial to convert the matrix to a format better suited for the access pattern. Table 3.3 summarizes the time complexity of common format conversions for an $n \times p$ sparse matrix with N non-zero elements:

From\To	CSC	CSR	COO	Dense
CSC	-	$\mathcal{O}(N+n)$ [26]	$\mathcal{O}(N)$ [27]	$\mathcal{O}(np)$
CSR	$\mathcal{O}(N+p)$ [26]	-	$\mathcal{O}(N)$ [27]	$\mathcal{O}(np)$
COO	$\mathcal{O}(N+n)$ [27]	$\mathcal{O}(N+p)$ [27]	-	$\mathcal{O}(np)$
Dense	$\mathcal{O}(np)$	$\mathcal{O}(np)$	$\mathcal{O}(np)$	-

Table 3.3: Time complexity of matrix format conversions.

3.2.2.2 Metadata Data Type

The `scipy.sparse` module allows the `row`, `col`, `indices`, and `indptr` metadata arrays to be explicitly set to either 32-bit or 64-bit signed integers. Unsigned integers, or other sizes, are not supported. Furthermore, all metadata arrays must use the same integer type: if one uses 64-bit integers, then all others must do so as well.

Since these metadata arrays constitute a significant portion of the total sparse matrix size (approximately $\frac{2}{3}$ for COO and at least $\frac{1}{2}$ for CSR and CSC when using 64 bit integers), the choice of data type has a notable impact on overall memory efficiency. For this reason, 32-bit integers are generally preferred in practice.

The use of 64-bit integers is typically required only in cases where the matrix dimensions exceed the capacity of 32-bit signed integers, i.e., when any of the relevant dimensions reach or surpass 2^{31}. Specifically:

- **COO**: $n \geq 2^{31}$ or $p \geq 2^{31}$.
- **CSR**: $N \geq 2^{31}$ or $p \geq 2^{31}$.
- **CSC**: $N \geq 2^{31}$ or $n \geq 2^{31}$.

When generating sparse matrices in *scipy*, it is important to explicitly specify the integer size used for index arrays. Different systems and implementations may default to different sizes, which can lead to unnecessary type conversions during data exchange.

For most practical applications, matrices with $n > 2^{31}$ and $p > 2^{31}$ are already computationally infeasible due to memory and processing constraints, even on modern hardware. Therefore, using 32-bit integers remains not only safe but also efficient for the vast majority of real-world scenarios involving sparse data.

3.2.2.3 Locality

The compared sparse storage formats also differ from dense matrices in memory access patterns. The former enables linear access, making iteration straightforward. The latter, in contrast, require accessing three separate arrays, resulting in reduced spatial locality.

The `indptr` array in the CSR/CSC formats is of particular interest, as row/column lookups require identifying the corresponding interval in this array. Since `indptr` is already sorted, binary search can be used to locate entries in $\mathcal{O}(\log n)$ time on average. However, this results in a highly irregular access pattern, which can have implications for cache efficiency and parallelization.

3.3 Transforms

Transforms are fundamental tools in signal processing and numerical computing. They convert data from one domain to another, typically from the time or spatial do-

main to a frequency or spectral domain, enabling more efficient analysis, compression, and manipulation by exploiting mathematical structure.

3.3.1 Fourier Transform and Inverse Fourier Transform

The **Fourier Transform** [28] decomposes a function into its constituent frequencies, serving as a cornerstone of signal analysis and structured data processing. For a continuous-time function $f(t)$, it is defined as:

$$F(\xi) = \mathcal{F}\{f(t)\}(\xi) = \int_{-\infty}^{\infty} f(t)e^{-2\pi it\xi}dt$$

This transformation can be inverted under mild integrability conditions using the **Inverse Fourier Transform**, which reconstructs the original function from its frequency representation:

$$f(t) = \int_{-\infty}^{\infty} F(\xi)e^{2\pi it\xi}d\xi$$

Together, these two operations form a bijective transform pair that preserves information and energy in both domains.

3.3.2 Discrete Fourier Transform (DFT) and its Inverse (IDFT)

In discrete computational settings, the **Discrete Fourier Transform** (DFT) maps a finite sequence $x_0, x_1, \ldots, x_{n-1}$ to another complex sequence $X_0, X_1, \ldots, X_{n-1}$ via:

$$X_k = \sum_{n=0}^{n-1} x_n e^{-2\pi ikn/n}, \quad k = 0, 1, \ldots, n-1$$

The **Inverse Discrete Fourier Transform** (IDFT) recovers the original sequence from the DFT coefficients:

$$x_n = \frac{1}{n}\sum_{k=0}^{n-1} X_k e^{2\pi ikn/n}, \quad n = 0, 1, \ldots, n-1$$

The normalization factor $\frac{1}{n}$ and the positive exponent ensure orthogonality of the basis functions and conservation of energy across the transform pair.

Note that the DFT and IDFT of a zero vector also result in zero vectors, which is consistent with the linearity of the transforms.

3.3.3 Fast Fourier Transform (FFT) and its Inverse (IFFT)

Direct computation of the DFT has time complexity $\mathcal{O}(n^2)$, which becomes computationally prohibitive for large values of n. The **Fast Fourier Transform** (FFT) [28],

particularly the Cooley–Tukey algorithm, reduces this complexity to $\mathcal{O}(n \log n)$ by recursively dividing the problem into smaller subproblems.

Specifically, when n is even, the DFT can be decomposed into two smaller DFTs of size $\frac{n}{2}$, corresponding to the even- and odd-indexed elements of the input sequence:

$$X_k = E_k + \omega_n^k O_k$$
$$X_{k+n/2} = E_k - \omega_n^k O_k$$

where E_k and O_k denote the DFTs of the even and odd subsequences, respectively, and $\omega_n = e^{-2\pi i/n}$ is the primitive n-th root of unity.

The **Inverse Fast Fourier Transform** (IFFT) provides an efficient implementation of the IDFT with the same $\mathcal{O}(n \log n)$ complexity as the forward FFT. It follows a nearly identical recursive decomposition:

$$x_k = \frac{1}{n}\left(E_k + \omega_n^{-k} O_k\right)$$
$$x_{k+n/2} = \frac{1}{n}\left(E_k - \omega_n^{-k} O_k\right)$$

where E_k and O_k denote the IFFTs of the even and odd subsequences, respectively, and $\omega_n = e^{-2\pi i/n}$ is the primitive n-th root of unity.

Compared to the forward FFT, this formulation introduces a conjugate symmetric kernel (ω_n^{-k} instead of ω_n^k) and includes a final normalization by $\frac{1}{n}$, ensuring energy conservation and exact reconstruction of the original sequence.

3.3.4 Walsh–Hadamard Transform

The **Walsh–Hadamard Transform** (WHT) [29] is a linear, orthogonal transformation that maps an input vector from its original (often time or spatial) domain to a new domain called the *Hadamard domain*, where the basis functions are square waves (as opposed to sinusoids in the Fourier domain).

The transformation uses Hadamard matrices as its basis. The simplest Hadamard matrix is defined for $n = 1$ as:

$$H_1 = [1]$$

For input sizes where the dimension is 2^n (with n being a non-negative integer), the corresponding Hadamard matrix is constructed recursively:

$$H_{2n} = \begin{bmatrix} H_n & H_n \\ H_n & -H_n \end{bmatrix}$$

This recursive definition yields a matrix that is composed solely of $+1$ and -1 elements, and which are orthogonal, i.e., $H_n H_n^\top = nI$.

Given an input vector $x \in \mathbb{R}^n$, its Walsh–Hadamard Transform is defined by:

$$y = H_n x$$

Here, y is said to be the representation of x in the Hadamard domain.

3.3.5 Fast Walsh–Hadamard Transform

The **Fast Walsh–Hadamard Transform** (FWHT) [29] is an efficient algorithm for computing the Walsh–Hadamard Transform, achieving a computational complexity of $\mathcal{O}(n \log n)$. Unlike FFT, which exploits the rotational symmetries of complex exponentials, FWHT relies exclusively on simple additions and subtractions. This is possible because Hadamard matrices contain only $+1$ and -1 entries, thus eliminating the need for multiplication operations involving complex numbers.

The FWHT algorithm leverages the recursive structure of H_n by breaking the computation into smaller subproblems using divide-and-conquer techniques. At each recursive step, the algorithm partitions the input vector into two halves and applies pairwise addition and subtraction operations, commonly referred to as "butterfly" operations. This yields the recurrence:

$$T(n) = 2\,T\left(\frac{n}{2}\right) + \mathcal{O}(n),$$

which resolves to the overall complexity of $\mathcal{O}(n \log n)$.

The computational efficiency of the FWHT makes it especially attractive in applications that demand real-time processing and low computational overhead. Additionally, the algorithm's inherent suitability for parallel processing further boosts its appeal in high-performance computing environments.

3.4 Hash Functions

Hash functions are used to implement randomized algorithms by sampling over a predefined hash function family.

The hash function $h : \mathcal{U} \to \mathcal{T}$, maps keys from a large universe $\mathcal{U}$ to a target set $\mathcal{T}$. A well-designed hash function should satisfy the following characteristics:

- **Well-defined behaviour:** As a mathematical function, $h(x)$ produces a consistent output for any given input.
- **Efficiency:** The time required to compute $h(x)$ should be constant on average, i.e., $\mathcal{O}(1)$ in typical usage scenarios.
- **Collision resistance:** The function should mimic randomness, minimizing the probability that two distinct inputs $x \neq y$ produce the same output, i.e., $h(x) = h(y)$.

In practice, many randomized hash function families incorporate an additional parameter known as a *seed*, which serves to control the internal behaviour of the function. By varying the seed, different mappings can be generated from the same underlying hash function.

3.4.1 Universal Hashing

A family $\mathcal{H}$ of hash functions is **universal** if:

$$\forall x \neq y \in \mathcal{U}, \quad \Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{|\mathcal{T}|}$$

A classic construction is given by:

$$h_{a,b}(x) = ((ax + b) \mod p) \mod m$$

where the parameters satisfy: $a \in \{1, 2, ..., p - 1\}$, $b \in \{0, 1, ..., p - 1\}$. p is a prime number such that $p \geq |\mathcal{U}|$ and m is the size of the hash table. a and b are chosen independently and uniformly at random from their respective ranges.

3.4.1.1 Multiply-shift Hashing

For settings where $\mathcal{U} = \{0, ..., 2^w - 1\}$ and $\mathcal{T} = \{0, ..., 2^l - 1\}$, an efficient alternative is **multiply-shift hashing**, defined as:

$$h_a(x) = (ax \mod 2^w) \gg (w - l)$$

where a is an odd integer in $[1, 2^w)$ and "$\gg$" denotes bitwise right-shift. This method achieves 2-universality using simple bitwise operations, making it particularly attractive for some computations.

3.4.1.2 Multiply-add-shift Hashing

For input $\mathcal{U} = \{0, \ldots, 2^w - 1\}$ and target $\mathcal{T} = \{0, \ldots, 2^l - 1\}$, the **multiply-add-shift** scheme is defined as:

$$h_{a,b}(x) = ((a \cdot x + b) \mod 2^w) \gg (w - l)$$

where $a, b \in \{0, \ldots, 2^w - 1\}$ are independently chosen random integers. The number of output bits is determined by l, and w typically corresponds to the bit width used for fixed-size integer arithmetic on the target architecture (e.g., 64 bits in modern 64-bit systems).

This scheme achieves strong universality under mild constraints, such as selecting a uniformly from odd integers. In particular, for any two distinct inputs $x \neq y$, the pair $(h_{a,b}(x), h_{a,b}(y))$ is uniformly distributed over $\mathcal{T} \times \mathcal{T}$, ensuring minimal collision probability.

In practical implementations, the modulo operation mod 2^w is implicit due to fixed-width arithmetic on modern hardware. The use of high-order bits (via shifting) ensures a uniform distribution over the output space, and the entire computation avoids division, making it highly efficient for large-scale randomized algorithms, including hashing-based compression and sketching.

3.4.2 k-wise Independent Hashing

A hash family $\mathcal{H}$ is *k-wise independent* [30] if for any distinct $x_1, ..., x_k$ and any $y_1, ..., y_k$, we have:

$$\Pr_{h \in \mathcal{H}}[h(x_1) = y_1 \wedge \cdots \wedge h(x_k) = y_k] = 1/m^k$$

In other words, each key is hashed independently.

A standard construction employs degree $(k-1)$ polynomials over finite fields:

$$h_{a_0,...,a_{k-1}}(x) = \sum_{i=0}^{k-1} a_i x^i \bmod p$$

where p is prime and a_i are random coefficients. This method ensures strong randomness properties while maintaining an evaluation cost of $\mathcal{O}(k)$, making it useful in randomized algorithms, load balancing, and sketching techniques.

In practice, exact k-wise independence may be computationally expensive. Instead, ϵ-almost k-wise independent hashing relaxes the strict probability requirement to:

$$\left|\Pr[h(x_1) = y_1 \wedge \cdots \wedge h(x_k) = y_k] - 1/m^k\right| \leq \epsilon$$

This trade-off enables more efficient implementations while maintaining sufficient randomness for practical applications.

3.4.3 Simple Tabulation Hashing

Simple tabulation hashing [31] is a highly efficient alternative to classical hash functions. In this method, an input x is first decomposed into c characters $x_1, x_2, \ldots, x_c$, typically treated as bytes or fixed-width characters. For each position $i \in \{1, \ldots, c\}$, a random lookup table T_i is initialized such that $T_i : \Sigma \to \mathbb{U}$, where Σ denotes the character alphabet and $\mathbb{U}$ the output space (often 32 or 64 bits). The hash function is then defined as the bitwise XOR ($\oplus$) of table lookups:

$$h(x) = T_1[x_1] \oplus T_2[x_2] \oplus \cdots \oplus T_c[x_c]$$

Despite its conceptual simplicity, simple tabulation hashing achieves a collision probability of $\mathcal{O}(n^{-1/c})$ for c-character inputs, making it competitive with more sophisticated hashing schemes. Additionally, it satisfies 3-wise independence.

Furthermore, the method is highly cache-friendly. Since the lookup tables T_i are small and frequently accessed, they can often be retained in the CPU's cache, leading to fast and predictable access times. This property makes simple tabulation hashing particularly appealing for applications such as hash tables, load balancing, and other latency-sensitive structures requiring fast key lookups.

3.5 Sketching Techniques

Sketching is a linear dimensionality reduction technique that compresses high-dimensional data while preserving essential statistical properties, trading off accuracy for reduced space complexity, with theoretical error bounds. Two widely used sketching techniques are the AMS sketch and the CountSketch. Both rely on hash-based randomization but serve different purposes and offer distinct computational trade-offs.

3.5.1 AMS Sketch

AMS sketch introduced by Alon, Matias, and Szegedy [32], is primarily designed to estimate frequency moments in data streams, particularly the second frequency moment:

$$F_2 = \sum_{j=1}^{n} f_j^2,$$

where f_j denotes the frequency of item j.

The sketch works by assigning a random sign $s(j) \in \{-1, +1\}$ to each index j. Given a stream vector $x \in \mathbb{R}^n$, an estimator is constructed as:

$$Z = \sum_{j=1}^{n} s(j) \cdot x_j$$

Then, Z^2 is an unbiased estimator for F_2, i.e., $\mathbb{E}[Z^2] = F_2$. However, due to high variance in individual estimates, multiple independent sketches are maintained and averaged to improve accuracy. This approach is especially useful in streaming settings with limited memory, where logarithmic space suffices for a good approximation.

3.5.2 CountSketch

CountSketch proposed by Charikar, Chen, and Farach-Colton [33], is tailored for estimating point frequencies and identifying the elements that occur most frequently in data streams, particularly under skewed distributions.

CountSketch uses two sets of 2-wise independent hash functions:

- $h_i : [n] \to [b]$ maps each item to one of b buckets.
- $s_i : [n] \to \{-1, +1\}$ assigns a random sign to each item.

Let $V = \{v_1, v_2, \ldots, v_n\} \subseteq [n]$ denote a stream of items, each associated with a weight w_k. Let t be the number of hash function pairs used, and let $C \in \mathbb{Z}^{t \times b}$ be a matrix of counters, initialized to zero. For each element v_k and for each row $i \in \{1, \ldots, t\}$, the algorithm updates C with:

$$C[i, h_i(v_k)] \mathrel{+}= s_i(v_k) \cdot w_k$$

Finally, to estimate the weight of a query element q, CountSketch returns:

$$\hat{f}(q) = \text{median}_{i\in[t]} \left(C[i, h_i(q)] \cdot s_i(q) \right)$$

This estimator is unbiased, and its variance depends on the distribution of frequencies and the degree of hash collisions. The median-of-means technique effectively reduces the impact of outliers due to collisions.

3.6 Pagh's Algorithm

Pagh's algorithm [34] offers an efficient method for matrix multiplication, providing an unbiased estimator of the product matrix. The variance of this estimator is bounded by the Frobenius norm of the resulting matrix. This method leverages sketching-based compression to reduce computational complexity while ensuring accuracy within defined bounds. It operates on the following inputs:

- $A \in \mathbb{R}^{n\times m}$: the left input matrix.
- $B \in \mathbb{R}^{m\times p}$: the right input matrix.
- seed $\in \mathbb{Z}$: an seed for randomised hashing.
- $d \in \mathbb{Z}$: the number of sketch repetitions, controlling the trade-off between accuracy and runtime.
- $b \in \{2^k \mid k \in \mathbb{Z}\}$: the polynomial and sketch size, which also influences the trade-off between compression and approximation quality.
- $h_1, h_2 : [n] \to [b]$: two 2-wise independent hash functions.
- $s_1, s_2 : [n] \to \{-1, +1\}$: two 2-wise independent sign hash functions.

The core idea of the algorithm is to apply CountSketch-inspired hashing to compress both A and B into low-dimensional sketches. These compressed sketches are then combined via FFT to perform convolution, which approximates the matrix product $C \approx AB$. By performing computations in the frequency domain and aggregating contributions across hash buckets, the algorithm aims to achieve speedup over the naive matrix multiplication (i.e., the cubic-time exact method), while maintaining theoretical error bounds.

3.6.1 Compression

In Pagh's algorithm, each sketching iteration constructs two **polynomials**, derived from the rows of matrix A and the columns of matrix B, using hash functions h_1, h_2 and sign functions s_1, s_2, similar to the CountSketch technique. The sketch of the matrix product is then computed via FFT-based polynomial convolution, which effectively aggregates contributions of individual entries into hash buckets.

These hashing and signing functions are used to construct the following polynomial

representation:

$$p(x) = \sum_{k=1}^{n} \left(\sum_{i=1}^{n} A_{ik} s_1(i) x^{h_1(i)} \right) \left(\sum_{j=1}^{n} B_{kj} s_2(j) x^{h_2(j)} \right)$$

The product polynomial $p(x)$ is then evaluated using FFT to compute the coefficients $c_0, \ldots, c_{b-1}$, which satisfy:

$$\sum_i c_i x^i = (p(x) \bmod x^b) + (p(x) \operatorname{div} x^b)$$

The total time complexity for this procedure is $\widetilde{\mathcal{O}}(n^2 + nb)$.

Algorithm 1 provides pseudocode for this compression step, where $\mathbf{0}_a$ denotes a zero vector of length a.

Algorithm 1 Pagh's Algorithm: Compression Step [34]

```
1: function CompressedProduct(A, B, b, d)
2:     for t ← 1 to d do
3:         s1[t], s2[t] ← random maps {1, ..., n} → {−1, +1}
4:         h1[t], h2[t] ← random maps {1, ..., n} → {0, ..., b − 1}
5:         p[t] ← 0_{b/2+1}
6:         for k ← 1 to n do
7:             pa, pb ← 0_b
8:             for i ← 1 to n do
9:                 pa[h1(i)] ← pa[h1(i)] + s1[t](i) · A_ik
10:            end for
11:            for j ← 1 to n do
12:                pb[h2(j)] ← pb[h2(j)] + s2[t](j) · B_kj
13:            end for
14:            pa ← FFT(pa)
15:            pb ← FFT(pb)
16:            for z ← 1 to b do
17:                p[t][z] ← p[t][z] + pa[z] · pb[z]
18:            end for
19:        end for
20:    end for
21:    for t ← 1 to d do
22:        p[t] ← IFFT(p[t])
23:    end for
24:    return (p, s1, s2, h1, h2)
25: end function
```

3.6.2 Decompression

After d independent repetitions, the approximate value of any entry C_{ij} in the product matrix is recovered by querying the appropriate hash bucket and applying the inverse

of the sign and hash functions, followed by a median aggregation—analogous to the recovery step in CountSketch. Specifically, for a given entry (i, j), the estimator C_{ij} is computed as:

$$C_{ij} = s_1(i)s_2(j)c_{(h_1(i)+h_2(j)) \bmod b}$$

This yields an unbiased estimate of the true entry AB, with the variance of the estimate depending on the parameter b.

The corresponding pseudocode is shown in Algorithm 2, where the function `Decompress` aggregates estimates from all d independent sketches and returns the median value as the final estimate of C_{ij}. By leveraging CountSketch-inspired compression and

Algorithm 2 Pagh's Algorithm: Decompression Step [34]

```
1: function DECOMPRESS(i, j)
2:     for t ← 1 to d do
3:         X_t ← s_1[t](i) · s_2[t](j) · p[t][(h_1(i) + h_2(j)) mod b]
4:     end for
5:     return Median(X_1, ..., X_d)
6: end function
```

decompression, Pagh's algorithm computes a provably accurate approximation of matrix products in sub-quadratic time, particularly effective for sparse matrices. As such, it serves as a powerful tool in large-scale numerical linear algebra applications, where exact matrix multiplication is often computationally prohibitive.

3.6.3 Correctness

The result from Pagh's algorithm is an approximation of the matrix product AB. The estimator for entry $(AB)_{ij}$ is given by:

$$C_{ij} = s_1(i)s_2(j)c_{h_1(i)+h_2(j) \bmod b}$$

The variance of this estimator is bounded by $\frac{\|AB\|_F^2}{b}$, ensuring a controlled approximation error. With high probability, the algorithm computes the matrix product AB exactly in time $\widetilde{\mathcal{O}}(max(N_A, N_B) + n \cdot N_C)$ [34].

The accuracy of the approximation is inversely proportional to the number of small numerical entries, with accuracy being maximized when most entries are small, and only a few are large.

3.6.4 Resource Usage

The notation $\widetilde{\mathcal{O}}$ is used to suppress polylogarithmic factors; that is, it hides terms involving logarithmic components such as $\log(N)$ or $\log(n+b)$. The overall asymptotic runtime of Pagh's algorithm is $\widetilde{\mathcal{O}}(N + nb)$. Under the assumption that the input matrices are sparse and sufficiently large, this complexity represents a significant improvement over the conventional exact matrix multiplication methods, which has

a cubic time complexity $\mathcal{O}(n^3)$. Thus we expect Pagh's algorithm to demonstrate increasingly better relative performance compared to other methods for very large n.

In addition to the space required for storing the input and output matrices, the algorithm also has moderate memory requirements. Specifically, Pagh's algorithm requires $\mathcal{O}(d)$ space for the polynomial hash functions and an additional $\mathcal{O}(db)$ space for the rest of the computation [34].

3.6.5 Parameter Choice

The parameters b and d govern the trade-off between computational efficiency and the accuracy of the results. For the complete Pagh's algorithm, which includes both compression and decompression, increasing either parameter generally improves accuracy but increases computational cost. Prior work provides the following guidelines for parameter selection [4]:

$$d = \log_2(n) \cdot d_c, \text{ with } 0.1 < d_c < 10,$$

$$b = n \cdot b_c, \text{ with } 0.1 < b_c < 10$$

Based on these recommendations, we can assume that the parameters are chosen such that $d = \mathcal{O}(\log_2 n)$ and $b = \mathcal{O}(n)$.

4

Methodology

This chapter outlines the methodologies, tools, and techniques employed in the development and evaluation of the algorithm implementation. It covers the prototyping process, the design and integration of optimizations, as well as the testing strategies and benchmarking frameworks used to assess performance. Additionally, we describe the datasets utilized in the experiments, along with their relevance to the study.

4.1 compmatmul

`compmatmul` is an implementation of Pagh's algorithm for approximate matrix multiplication, currently under active development by J. Andersson and M. Karppa [4]. Written in C++ and parallelized using **OpenMP**, it provides Python API bindings, offering a high-performance solution for large-scale matrix operations.

`compmatmul` can employ one of three hashing schemes in the compression phase: *multiply-shift hashing*, *multiply-add-shift hashing*, and *simple tabulation hashing*.

For transformation, it supports both FFT, as used in the original formulation of Pagh's algorithm, and the FWHT as an alternative. In our experiments, we focus primarily on the FFT-based approach due to its theoretical guarantees and compatibility with existing implementations.

4.1.1 Libraries Used

`compmatmul` integrates the **FFTW** (Fastest Fourier Transform in the West) library [35]. FFTW generates execution **plans**, which are optimized instruction sequences tailored to specific transform sizes, types, and memory layouts. Although plan generation is computationally expensive, the overhead is amortized over repeated usage. Upon initialization, `compmatmul` precomputes and caches plans for both FFT and IFFT operations.

To further optimize low-level vector operations, `compmatmul` integrates either **Intel's Math Kernel Library** (MKL) or **OpenBLAS**, depending on the system architecture. Both libraries require data alignment to cache-line boundaries to enable efficient SIMD execution and minimize runtime overhead caused by data interleaving [35].

An important property of `compmatmul` is that its execution time is independent of

the input data. This stems from two factors:

1. The algorithm does not differentiate in execution paths based on the values in the input matrices.

2. The underlying FFTW and MKL functions are executed in a manner that is independent of the input data [35], [36].

4.1.2 Data Format

Currently, it supports double-precision (8-byte) floating-point arithmetic, which will be the focus of our subsequent analysis. While support for single-precision (4-byte) floats has been implemented, it exhibits numerical instability in practice, and thus is not recommended for production use [4].

While existing implementation assumes dense matrices as both input and output, Pagh's algorithm is inherently well-suited for sparse matrices. An alternative implementation has been proposed that accepts sparse matrix formats as input. Such an extension would improve memory efficiency and enable direct integration with sparse data generated in earlier stages of complex computational pipelines, eliminating the need for costly conversions to dense format.

4.2 Minerva Cluster

All experiments were performed on the **Minerva** computing cluster, managed by the Department of Computer Science and Engineering at Chalmers University of Technology.

4.2.1 CPUs

Each node of Minerva, features two *Intel Xeon Gold 6548N*, *Emerald Rapids* CPUs, with a total of 64 physical cores. They support AVX-512 instructions, which allow for the processing of 8 double-precision floating point variables (8 bytes each) per instruction [37]. As an Intel system the MKL library is used for vector operations.

4.2.1.1 Theoretical Performance

The CPUs operate at a base clock speed of 2.8 GHz, and 4.1 GHz with turbo boost [37]. Each core has two floating-point execution units capable of processing 1 FMA (Fused Multiply-Add) instruction per cycle [38].

The theoretical peak floating-point throughput (P_{peak}) is calculated as:

$$P_{\text{peak}} = f \times N_{\text{cores}} \times L_{\text{vec}} \times N_{\text{ex}} \times N_{\text{FMA}} \times \text{IPC} \tag{4.1}$$

where:

- $f = 2.8\,\text{GHz}$ (CPU base frequency),

- $N_{\text{cores}} = 2 \times 32 = 64$ (number of physical cores),
- $L_{\text{vec}} = 8$ (64-bit floating point elements per vector),
- $N_{\text{ex}} = 2$ (floating-point execution units per core),
- $N_{\text{FMA}} = 2$ (FMA allows for two floating-point operations per instruction),
- $\text{IPC} = 1$ (instructions per cycle).

Substituting these values, the peak throughput for 64 bit floating point operations is:

$$P_{\text{peak, FP64}} = 2.8 \times 10^9 \times 64 \times 8 \times 2 \times 2 \times 1 = 5.735\,\text{TFLOP/s}$$

4.2.1.2 Measured Performance

While theoretical performance provides an upper bound, actual performance is influenced by various system-level factors, including memory bandwidth, cache hierarchy, and workload characteristics. To evaluate the floating-point performance under ideal conditions, we conducted the LINPACK benchmark [39], with results summarized in Table 4.1.

Scope	**Expected GFLOPs**	**Measured GFLOPs**	$\Delta\%$
Single Core w/o boost	89.6	112.7	125.8%
Single Core w/ boost	131.2		85.99%
Single CPU	2867.5	1744.1	60.82%
Dual CPU	5735.0	3529.9	61.55%

Table 4.1: LINPACK benchmark results for the Minerva cluster.

The observed single-core performance exceeds the theoretical baseline, which can be attributed to the CPU's turbo boost technology that increases clock frequency under light-threaded workloads.

4.2.2 Memory

The Minerva cluster node features a multi-level memory hierarchy, with each CPU core equipped with private L1 and L2 **caches**. The L2 cache is inclusive of the L1 cache, ensuring that all data present in the L1 cache is also available in the L2. Each CPU further includes a 60 MB shared L3 cache (approximately 1.875 MB per core), serving as the last-level cache. The L3 cache implements a victim caching policy: rather than strictly including or excluding lines from lower-level caches, it selectively retains evicted lines from the L2 cache to improve overall cache utilization.

When multiple threads share data, the system can load will load a single copy of the data into the shared L3 cache, allowing all threads to access it efficiently without redundant copies. This improves cache utilization and reduces pressure on main

memory bandwidth. In contrast, if each thread maintains its own copy, cache capacity is wasted, which can degrade performance, especially on cache-constrained tasks.

The system is equipped with 512 GiB of main memory, operating at a speed of 5200 MT/s[1].

To characterize the bandwidth characteristics of each memory level, we conducted measurements using the STREAM benchmark [40]. The benchmark was compiled with optimization flags enabled and executed using OpenMP parallelism. To isolate performance at each memory level, the working set size was selected to exceed the capacity of smaller caches, thereby targeting the desired level. Multiple runs were performed, and the maximum observed bandwidth was recorded to minimize the impact of transient system noise. Results are summarized in Table 4.2.

Memory level (Size)	**Shared by**	**Bandwidth**	AI_{ridge}
L1 cache (80KB)	Each core	9663.36 GB/s	0.75
L2 cache (2MB)	Each core	2931.84 GB/s	2.46
L3 cache (60MB)	Each CPU	880.60 GB/s	3.96
Main memory (512GB)	Entire node	252.08 GB/s	14.00

Table 4.2: Bandwidth and arithmetic intensity characteristics across memory levels of the Minerva cluster.

4.2.2.1 Machine Balance

The **arithmetic intensity (AI)** of a computational task was introduced by Williams et al. [41]. AI is defined as the ratio of total floating-point operations (FLOPs) to the total amount of data movement (in bytes).

A high arithmetic intensity indicates that a computation performs many operations per byte accessed, making it more likely to be compute-bound — limited primarily by the available computational throughput. In contrast, a low arithmetic intensity suggests that performance is constrained by memory bandwidth, rendering the workload memory-bound.

The **machine balance** or ridge point, denoted as AI_{ridge}, represents the minimum arithmetic intensity required for a computation to fully utilize the peak floating-point performance of a processor. It is defined as the ratio of peak computational throughput to peak memory bandwidth. This value reflects the equilibrium point between computation and memory access capabilities between computational throughput and memory bandwidth for each memory level.

Using our empirically measured memory bandwidths and core throughput values, we computed the machine balance for each relevant memory level on the Minerva cluster.

[1]Mega Transfers per second

4.3 Libraries and Tools

We employed a range of third-party libraries and tools to support development, measurement, and testing across different stages of the project.

4.3.1 Prototyping Phase

Our research methodology adopted a two-phase development approach to balance algorithmic understanding with computational efficiency. The initial prototyping phase focused on exploring Pagh's algorithm and validating its core components before performance optimization.

We chose Python as the primary implementation language due to several advantages:

- **Rapid Development**: Python's high-level syntax and dynamic typing enabled efficient translation of theoretical concepts into executable code, accelerating our exploration of Pagh's algorithm.
- **Algorithmic Clarity**: The language's pseudocode-like expressiveness helped isolate core algorithmic components from implementation complexities, particularly beneficial for understanding the interaction between FFT and FWHT operations.
- **Ecosystem Support**: Python's scientific computing stack, particularly NumPy and SciPy, provided numerical routines that served as both implementation tools and verification references.

Primarily, we relied on two core libraries:

NumPy: Provided essential array operations and broadcasting mechanisms for matrix manipulations. While sufficient for prototyping, we observed limitations in its native sparse matrix support for large-scale problems.

SciPy: Extended NumPy capabilities through its `scipy.sparse` module, offering implementations of CSR, CSC and COO sparse formats. This was particularly valuable for implementing the sparse recovery components of Pagh's algorithm. Additionally, SciPy's FFT implementations provided essential functional verification for subsequent optimizations [42].

4.3.2 Implementation Phase

To transition from prototype validation to performance-optimized execution, we utilized **pybind11** for seamless Python-C++ interoperability. This library enabled us to:

- Migrate necessary components to C++
- Retain Python-based test harnesses for continuous integration

- Share data between NumPy arrays and C++ buffers via the buffer protocol

4.3.3 Measurement Phase

To evaluate and optimize the performance of our C++ implementation, we used a combination of profiling tools and compiler optimizations:

- **Linux Perf**: A standard open-source performance monitoring tool for Linux systems. Perf allowed us to measure key CPU events such as cache hits/misses, instructions per cycle (IPC), and memory accesses, which were crucial for diagnosing performance issues. While Perf supports profiling specific functions, we found it challenging to isolate individual code components due to the complexity of the software stack of Python and OpenMP.
- **Intel VTune** [43]: A commercial performance profiler developed by Intel, designed for deep analysis of x86 microarchitectures. It provided insights into microarchitecture exploitation, bottlenecks and memory access analysis.
- **Compiler:** We compiled our binaries using *g++*, applying modern C++ standards (`-std=c++23`) and aggressive optimization flags (`-O3`, `-flto` and `-march=native`) to maximize performance on our target architecture.

4.4 Datasets

For our experimental evaluation, we selected a set of matrices from the SuiteSparse Matrix Collection [44]. All datasets were loaded in the Matrix Market (`.mtx`) format, a standard file format specifically designed for sparse matrices, and subsequently converted to the relevant sparse format in *scipy*.

To realistically evaluate Pagh's algorithm, which benefits from improved asymptotic runtime and scales more efficiently with larger matrices, we intended to benchmark with the largest possible matrices. However, due to to memory limitations, we focused on square matrices with sizes n close to or below $2^{17} = 131072$. This size range enables comparison across between sparse and dense implementations, as larger sizes would exhaust Minerva's memory if stored in a dense format.

The selected datasets are representative of real-world sparse problems, and are divided into two groups.

4.4.1 Datasets with Comparable Dimensions

The first group, summarized in Table 4.3, consists of relatively large matrices with dimensions n close to 2^{17}, yet still being manageable by the dense implementation. These matrices vary in structure and density, providing a diverse testbed for evaluating algorithm behaviour under real world conditions.

Datasets	n_A	N_A	d_A	r_0	CSR/CSC Size
xenon2	157,454	3,866,668	1.56E-04	0.00000	44.85 MB
G2_circuit	150,102	438,388	1.95E-05	0.00000	5.59 MB
TEM152078	152,078	3,305,702	1.43E-04	0.00000	38.41 MB
Ga19As19H42	133,123	4,508,981	2.54E-04	0.00000	52.11 MB
soc-sign-epinions	131,828	841,372	4.84E-05	0.27695	10.13 MB

Table 4.3: Properties of medium-scale sparse matrices used in experiments: matrix dimension $n_A = p_A$, number of non-zeros N_A, density d_A, portion of empty rows r_0 and storage size in CSR/CSC format.

4.4.2 Datasets Varying in Dimensions

The second group, shown in Table 4.4, consists of matrices with sizes ranging from $n = 2^{13}$ to $n = 2^{17}$. These matrices allow us to evaluate how algorithm performance scales with input size.

Datasets	n_A	N_A	d_A	r_0	CSR/CSC Size
delaunay_n17	131,072	786,352	4.58E-05	0.130547	9.50 MB
delaunay_n16	65,536	393,150	9.15E-05	0.13367	4.75 MB
delaunay_n15	32,768	196,548	1.83E-04	0.131256	2.37 MB
delaunay_n14	16,384	98,244	3.66E-04	0.1329956	1.19 MB
delaunay_n13	8,192	49,094	7.32E-04	0.131958	0.59 MB

Table 4.4: Properties of power-of-two dimension sparse matrices used in experiments: matrix dimension $n_A = p_A$, number of non-zeros N_A, density d_A, portion of empty rows r_0 and storage size in CSR/CSC format.

While these matrices exceed the L3 cache capacity of our test CPU when stored in dense format, they fit comfortably within the cache when represented using the CSR or CSC sparse formats.

4.5 Validation

`compmatmul` includes a set of built-in testing scripts that validate the algorithm's correctness using predefined input matrices. To verify our modifications, we extended these scripts to include comparisons between the original and our adapted implementation, ensuring consistent output for identical inputs.

In addition, we developed a custom script to generate large-scale random sparse matrices. This allows us to evaluate both versions under more diverse and realistic conditions, beyond the scope of the default test cases. The expanded testing framework enables rapid validation across a wide range of input configurations, thereby strengthening confidence in the correctness of our modified implementation.

Using this comprehensive validation approach, we verified the correctness of each individual modification step, ensuring that no change introduced unintended be-

haviour.

4.6 Experimental Setup

All experiments were conducted on the Minerva cluster, using $t = 64$ threads with one thread pinned to each physical core. Thread affinity was enforced using `taskset`, a Linux utility for binding processes to specific CPU cores.

In previous measurements, the implementation using the *multiply-add-shift* hashing scheme demonstrated a favourable balance between performance and randomisation properties [4], and was therefore selected for use in all subsequent measurements.

To evaluate the performance of the matrix multiplication primitive, we computed the self-multiplication A^2 for each input matrix A. We used a constant seed for all experiments, and unless otherwise specified, set $d_c = b_c = 2$.

To ensure stable performance measurements, we performed three iterations of every measurement, and reported the average result. Averaging across multiple runs helps reduce the impact of transient system noise, scheduling artifacts, and other sources of non-deterministic variability, resulting in more reliable measurements.

5

Implementation

This chapter further elaborates on `compmatmul`, and provides a detailed description of several optimizations introduced, along with their theoretical analysis. The full source code of our modifications is available in Appendix A.2.

5.1 Existing Implementation

Pagh's algorithm can be divided into two primary phases: **Compression** and **Decompression**. The following discussion outlines their implementation in `compmatmul`.

5.1.1 Baseline Compression

The `compmatmul` implementation allows for the multiplication of $A^T B = C$, where A is transposed. It ensures that each column of A can be iterated over without expensive gather operations, and the associated cache line inefficiencies, as discussed in Section 3.1.1.

A^T and B are processed row by row. For each row, every element is hashed and signed before being accumulated into corresponding buckets, referred to as `pa` and `pb` respectively, as shown in Algorithm 3. Each element in the input matrix is hashed d times independently.

Algorithm 3 Binning of k-th row in matrix A^T into pa (`compmatmul`)

1: Initialize $\mathbf{pa} \leftarrow \mathbf{0}_b$
2: **for** $i \leftarrow 0$ to $n_a - 1$ **do**
3: $\quad \mathbf{pa}[h_1^t(i)] \leftarrow \mathbf{pa}[h_1^t(i)] + s_1^t(i) \cdot A[k \cdot n_a + i]$
4: **end for**

As a result, the asymptotic runtime of this step becomes $\mathcal{O}(n \cdot p \cdot d + p \cdot m \cdot d) = \mathcal{O}(p \cdot d \cdot (n + m))$.

Before each hashing iteration, `pa` and `pb` are initialized to zero using the standard library function `std::memset`. After hashing, each bucket is compressed using FFT. Subsequently, each row from A^T is multiplied with its corresponding column from B. This process is repeated d times, and the results from all iterations are aggregated into `p_fft`.

The binning is performed in parallel using multiple threads. To improve load balancing and adapt to varying workloads across iterations, a dynamic scheduling strategy with `guided` chunk size is employed. To ensure mutual exclusion during concurrent access by multiple threads, a separate **lock** is assigned to each row.

The multiplication and addition operations leverage highly optimized MKL scalar-vector multiplication and addition routines. The implementation is detailed in Algorithm 4.

Algorithm 4 Compressing *pa* and *pb* via FFT (`compmatmul`)

1: $\mathbf{pa} \leftarrow \texttt{FFT}(\mathbf{pa})$
2: $\mathbf{pb} \leftarrow \texttt{FFT}(\mathbf{pb})$
3: $\mathbf{pa} \leftarrow \mathbf{pa} \cdot \mathbf{pb}$ ▷ MKL elementwise multiplication
4: Acquire lock on $\mathbf{p_fft}[\mathbf{t}]$
5: $\mathbf{p_fft}[\mathbf{t}] \leftarrow \mathbf{p_fft}[\mathbf{t}] + \mathbf{pa}$ ▷ MKL elementwise addition
6: Release lock on $\mathbf{p_fft}[\mathbf{t}]$

5.1.1.1 Loop Ordering

Pagh's algorithm requires d iterations over each input matrix to hash and sign each element into pa and pb. However, repeatedly loading large matrices from memory can be expensive and may lead to increasing cache miss rates. Ideally, A^T, B, `pa`, and `pb` should remain cached to minimize redundant memory reads across loop iterations. Based on this assumption, we can estimate the ideal cache usage for Pagh's algorithm as:

$$\begin{aligned}
\text{Cache Usage (Pagh's algorithm)} &= A^T + B + pa \cdot t + pb \cdot t \\
&= m \cdot n + n \cdot m + b \cdot t + b \cdot t \\
&= 2n^2 + 2b \cdot t \\
&\approx 2n^2 + 2\log(n) \cdot t \quad (\text{since } \mathcal{O}(b) = \log(n))
\end{aligned}$$

Caching this becomes infeasible for larger n, as the size of the matrices are too large to fit into any cache.

To mitigate this issue, the developers of `compmatmul` adopted a row-wise processing strategy. Specifically, a single row from A^T is first hashed into d separate copies of pa, named `pas`, followed by binning a single row from B into `pbs`. The resulting bins are then transformed via FFT, multiplied, and aggregated before moving to the next row, as illustrated in Algorithm 5 [4].

This approach ensures that only one row from either input matrix needs to reside in the cache at any given time, thereby significantly increasing cache locality. Then, cache usage can be expressed by:

$$\begin{aligned}
\text{Cache Usage } (\texttt{compmatmul}) &= A^T[i] \cdot t(\text{or } B[i] \cdot t) + pas \cdot t + pbs \cdot t \\
&= p \cdot t + 2 \cdot b \cdot d \cdot t \\
&= t \cdot (p + 2b \cdot d)
\end{aligned}$$

Algorithm 5 Loop Ordering (`compmatmul`)

1: `pas` $\leftarrow$ `allocate`$(b \cdot d)$
2: `pbs` $\leftarrow$ `allocate`$(b \cdot d)$
3: **for** $t \leftarrow 0$ to $d - 1$ **do**
4: Bin A^T into `pas`$[t]$
5: **end for**
6: **for** $t \leftarrow 0$ to $d - 1$ **do**
7: Bin B into `pbs`$[t]$
8: **end for**
9: **for** $t \leftarrow 0$ to $d - 1$ **do**
10: FFT, multiply and collect `pas`$[t]$ and `pbs`$[t]$
11: **end for**

Under this configuration, `pas` and `pbs` dominate the cache usage. However, this benefit comes at a cost: each thread must maintain larger numbers of copies of *pa* and *pb*, one for each row. As a result, the total memory allocation increases to $\mathcal{O}(b \cdot d \cdot t)$. Since each thread allocates its own private copy of these arrays, they cannot be shared through the L3 cache - unlike matrices A^T and B, which are shared across threads. Consequently, when the cache usage of `pas` and `pbs` exceeds the capacity of the L1 and L2 caches, it causes substantial L3 cache pollution by filling it with data that other threads will never access.

5.1.2 Baseline Decompression

During the decompression phase, C is reconstructed from the polynomial, by iterating over every element in a row-major order, as shown in Algorithm 6. Rows are decompressed in parallel by different threads.

Algorithm 6 Parallel Decompression [4]

`#pragma omp for schedule(guided)`
1: **for** $i \leftarrow 0$ to $n - 1$ **do**
2: **for** $j \leftarrow 0$ to $m - 1$ **do**
3: MATMUL_DECOMPRESS_FFT$(p, C, i, j,$ `sketch`, `scratch`$)$
4: **end for**
5: **end for**

For each individual element in C, it is then computed as detailed in Algorithm 7.

The `median` function internally uses `std::nth_element` to determine the median value, which has a time complexity of $\mathcal{O}(n \log n)$ [45].

5.2 Overview of Adaptations

To support further development, we implemented custom C++ data structures for common sparse matrix formats. These structures are carefully designed to mirror the

Algorithm 7 Decompression (`compmatmul`) [4]

1: **for** $t \leftarrow 0$ to $d-1$ **do**
2: $\quad h \leftarrow (h_1[t](i) + h_2[t](j)) \bmod b$
3: $\quad s \leftarrow s_1[t](i) \cdot s_2[t](j)$
4: $\quad$ `scratch`$[t] \leftarrow s \cdot p[t,h]$
5: **end for**
6: $C[i,j] \leftarrow$ `median`(`scratch`)

memory layouts of their Python counterparts, ensuring seamless integration within the Python ecosystem.

We also developed a set of helper functions to enable efficient interoperability between C++ native pointers and SciPy sparse data structures. These conversion utilities preserve cache-line alignment and minimize redundant data copying, thereby improving performance in cross-language interfaces.

For matrix multiplication scenarios requiring different input/output format combinations, we provide three specialized implementations:

- **Dense-to-dense:** This is the original `compmatmul` implementation, which takes inputs in dense format and produces an output in dense format.
- **Sparse-to-dense:** This variant modifies only the compression phase to accept inputs in sparse format, while retaining the standard decompression logic to produce an output in dense format.
- **Sparse-to-sparse:** This variant introduces an alternative decompression, unlike the standard one used in *sparse-to-dense*, and produces an output in sparse format.

This modular design allows flexible pairing of compression and decompression components to meet specific format requirements.

5.3 Enhanced Compression

To enhance the compression phase, we introduce two key improvements: support for matrix inputs in sparse formats, and computational optimizations tailored for processing such inputs.

5.3.1 Processing Sparse Input Data

To efficiently handle input matrices stored in sparse formats, we preserve a row-wise iteration structure, which also facilitates further optimizations discussed later. As shown in Section 3.2, the CSR format and canonical COO format enable efficient row traversal. In contrast, processing rows or columns from a CSC-formatted matrix typically requires N_A memory accesses per row or column, resulting in a total of

$n \cdot N_A$ operations. Therefore, when performing row-wise iterations, we choose the CSR format due to its superior memory access efficiency.

Instead of requiring the input matrix A to be explicitly transposed, we can transpose it in $\mathcal{O}(1)$ by recasting between CSC and CSR as discussed in Section 3.2.

Algorithm 8 illustrates how k-th column of matrix A (namely, k-th row of matrix A^T) is binned into pa, when A is stored in CSC format, but treated as CSR.

Algorithm 8 Binning of k-th row in matrix A^T into pa, adapted for CSC input

1: **for** $i \leftarrow A.\texttt{indptr}[k]$ to $A.\texttt{indptr}[k{+}1] - 1$ **do**
2: $\quad \mathbf{pa}[h_1^t(A.\texttt{indices}[i])] \leftarrow \mathbf{pa}[h_1^t(A.\texttt{indices}[i])] + s_1^t(A.\texttt{indices}[i]) \cdot A.\texttt{data}[i]$
3: **end for**

Similarly, Algorithm 9 shows how the k-th row of matrix B is processed when B is in CSR format.

Algorithm 9 Binning of k-th row in matrix B into pb, adapted for CSR input

1: **for** $i \leftarrow B.\texttt{indptr}[k]$ to $B.\texttt{indptr}[k{+}1] - 1$ **do**
2: $\quad \mathbf{pb}[h_2^t(B.\texttt{indices}[i])] \leftarrow \mathbf{pb}[h_2^t(B.\texttt{indices}[i])] + s_2^t(B.\texttt{indices}[i]) \cdot B.\texttt{data}[i]$
3: **end for**

In both algorithms above, the non-zero elements of the k-th row are accessed using the index range from `indptr[k]` to `indptr[k+1]`. Since the algorithm involves multiplying the signed hash values by the corresponding matrix entries, zero-valued elements can be safely ignored. Furthermore, since the input matrix is only accessed during the compression phase, the remainder of the processing pipeline remains unaffected and can be left unchanged.

To mitigate workload imbalance, which is a common issue in sparse data due to uneven distributions of non-zero elements, we continue employing the `guided` scheduling strategy in OpenMP.

5.3.2 Efficiency Optimisations

In addition to the modified binning procedure, the adaption to sparse input data allows us to further optimise the compression:

5.3.2.1 Early Skipping of Zero-Valued Rows

If a row in A^T or B contains no non-zero elements, it can be excluded from processing. This optimization reduces both computational overhead and memory accesses.

Unlike in the original dense implementation, where detecting an empty row requires scanning all its entries, sparse formats allow for constant-time detection in $\mathcal{O}(1)$ using the `indptr` array, as shown in Algorithm 10. Specifically, if $A.\texttt{indptr}[k] = A.\texttt{indptr}[k+1]$ (or similarly for B), the corresponding row is empty and can be omitted from further computation.

This optimization avoids not only the binning step but also subsequent operations including FFT, multiplication, and accumulation into the global result structure. It is particularly effective for highly sparse matrices that contain a significant number of empty rows.

Algorithm 10 Skipping Empty Rows

```
1: if A.indptr[k] = A.indptr[k+1] or B.indptr[k] = B.indptr[k+1] then
2:     skip row
3: else
4:     process row
5: end if
```

5.3.2.2 Compiler-Assisted Vectorization via Aliasing-Free Access

When the input matrices are required to be in canonical formats, the index values within each row or column are guaranteed to be unique. However, the compiler may not inherently recognize this property and assume potential aliasing, which limits vectorization and other performance optimizations. By utilizing the `__restrict` or `omp simd` pragmas, we explicitly inform the compiler that aliasing does not occur. This enables more efficient vectorized instructions generation.

5.3.2.3 Memory-Efficient Loop Reordering

When processing sparse matrices, the input matrices A^T and B are significantly smaller than in their dense form. Consequently, for minimizing cache misses, the required cache size becomes:

$$\begin{aligned} \text{Cache Usage (Sparse Case)} &= A^T[i] \cdot t(\text{or } B[i] \cdot t) + pas \cdot t + pbs \cdot t \\ &= \max(N(A_i), N(B_i)) \cdot t + 2 \cdot b \cdot d \cdot t \\ &\approx \max(N(A_i), N(B_i)) \cdot t + 2\log(n) \cdot n \cdot t \end{aligned}$$

As such, the cache-based loop ordering optimization described in Section 5.1.1.1 becomes less relevant. Instead, in Algorithm 11, we implemented a reordering loops scheme, which reduces memory overhead and improves cache locality by fully compressing each row before proceeding to the next one.

Algorithm 11 Reordering Loops

```
1: pa ← allocate(b)
2: pb ← allocate(b)
3: for t ← 0 to d − 1 do
4:     Bin A^T into pa
5:     Bin B into pb
6:     FFT, multiply and collect pa and pb
7: end for
```

This restructuring reduces the cache requirement, for minimal additional loads, from $n + 2\log(n) \cdot n$ to:

$$\begin{aligned} \text{Cache Usage (Reordering Loops)} &= A + B + pa + pb \\ &= N_A + n + N_B + n + 2b \\ &= N_A + N_B + 2n + 2b \end{aligned}$$

However, this is an optimistic case. In practice, cache usage depends heavily on the content of private caches (L1/L2). If they are primarily filled with data from A or B, then up to t copies may reside across cores. Conversely, if A and B are predominantly stored in the shared L3 cache, then only a single copy would exist, resulting in less cache usage, while the `pa` and `pb` data consists of t copies, regardless of which caches it is contained within.

This change also reduces the overall memory allocation from $\mathcal{O}(b \cdot t)$ to $\mathcal{O}(b)$ by having each thread use a single pair of `pa` and `pb` arrays across all d iterations.

5.3.2.4 Merged Multiply-Add

By modifying the current sequence of multiplication and collection steps shown in Algorithm 4, it is possible to reduce load-bound bottlenecks caused by intermediate data movement.

Specifically, instead of first computing the full product vector $pa[i] \cdot pb[i]$ and temporarily storing it in `pa`, we directly accumulate the result into the shared `p_fft` structure. In the existing design, the first elements may be evicted from cache before the subsequent step begins, leading to expensive reloads of previously loaded data; this avoids unnecessary cache evictions and reloads.

The revised approach, outlined in Algorithm 12, replaces the optimized MKL addition and multiplication routines with a more cache-aware alternative. This implementation is expected to benefit from compiler auto-vectorization and we do not expect any significant loss of throughput compared to the optimised MKL implementation.

Algorithm 12 Compressing *pa* and *pb* with combined addition and multiplication operations

1: $\mathbf{pa} \leftarrow \texttt{FFT}(\mathbf{pa})$
2: $\mathbf{pb} \leftarrow \texttt{FFT}(\mathbf{pb})$
3: Acquire lock on $p_fft[t]$
4: **for** $i = 0$ to $\texttt{length}(\mathbf{pa}) - 1$ **do**
5: $\quad \mathbf{pa}[i] \leftarrow \mathbf{pa}[i] \cdot \mathbf{pb}[i]$
6: $\quad p_fft[t][i] \leftarrow p_fft[t][i] + \mathbf{pa}[i]$
7: **end for**
8: Release lock on $p_fft[t]$

However, this change increases the duration for which the lock on `p_fft[t]` must be held. On architectures such as Emerald Rapids, where the throughput of AVX-512 double-precision fused multiply-add instructions is comparable to performing

multiplication and addition sequentially [38], the additional computational overhead within the lock is negligible.

Nevertheless, this approach introduces a trade-off: while it reduces bandwidth usage, the extended lock holding time, caused by additional variable caching, may increase contention on the shared `p_fft` structure. Therefore, further evaluation is necessary to assess its overall impact.

5.4 Enhanced Decompression

To better accommodate sparse outputs, we also modified the decompression phase to produce results in CSR and COO formats, rather than as a dense matrix. Further details on these output format choices are discussed in Section 5.4.2.

5.4.1 Memory Allocation

The `compmatmul` implementation requires the caller to preallocate memory for the output matrix and pass it as a pointer. This design avoids costly dynamic memory allocation during execution. While this approach is straightforward for dense matrices whose size is fixed and known in advance, it poses a challenge for sparse matrices.

Specifically, the number of non-zero elements in the output matrix, which is required by all the sparse formats discussed, cannot be determined until after the decompression phase completes. As a result, our implementation must be responsible for both determining the number of non-zero entries N and allocating the corresponding memory for the output matrix C. This issue becomes even more challenging when attempting to parallelize the decompression phase.

Several approaches have been proposed to address this common challenge in sparse matrix multiplication algorithms [46].

One potential solution would be to first compute the result as a dense matrix, count the non-zero elements and then convert the result into a sparse format. However, this approach introduces unnecessary overhead without performance benefits. Moreover, it risks exhausting available memory: while the final sparse output might fit within system limits, the intermediate dense representation can be prohibitively large, especially for large-scale applications.

5.4.1.1 Size Prediction

Prediction methods involve estimating the number of non-zero elements in the output matrix before the computation begins. During this preliminary step, an estimate is generated to guide memory allocation decisions.

By precomputing an **upper bound** on the number of non-zero entries in C, we can allocate sufficient memory up front. By recording the non-zero elements for each row independently, we can not only multithread the counting process without requiring synchronization for a shared variable, but also easily derive the `indptr[i]` values

required for CSR or CSC output. This independence also enables straightforward parallelization when filling the output matrix C, as each thread knows exactly where to write its results in the final data structure.

A commonly used upper bound for the number of non-zero elements in the i-th row of the product matrix $C = AB$ is given by [46]:

$$u_i = \sum_{j \in N(A_{i,:})} N(B_{:,j}),$$

where $j \in N(A_{i,:})$ denotes the set of columns indices in row i of matrix A that contain non-zero entries.

However, since this is only an upper bound, it may lead to over-allocation of memory. Although it is possible to reallocate and copy data into correctly sized structures afterward, doing so introduces additional overhead and complexity. For this reason, we decided not to use this method.

Alternatively, we reuse the algorithm in the decompression phase to count the **exact number** of non-zero elements. However, it introduces a performance cost of iterating over the matrix C twice. Nevertheless, since our goal is to determine whether each entry is zero (rather than its exact value), we can optimize the counting process in two key aspects, as shown in Algorithm 13:

1. Instead of computing the median of the `scratch` entries, we can count the number of positive and negative entries to determine whether the resulting median is zero. Specifically, in all `scratch` entries, if `positive` $\geq \left\lfloor \frac{d}{2} \right\rfloor$, the median is positive; if `negative` $\geq \left\lfloor \frac{d}{2} \right\rfloor$, the median is negative. This allows us to replace the $\mathcal{O}(d \cdot \log d)$ function with an $\mathcal{O}(d)$ implementation.
2. Since the sign of an element does not affect whether it is zero, we can defer the signing operation until after verifying that the element is non-zero. This avoids unnecessary computations involving signed values for the vast majority of elements in a sparse output.

The rest of the decompression process remains unaffected but may benefit from the reduced overhead due to the early filtering of zero elements.

5.4.2 Outputting Sparse Data

After determining the number of non-zero elements in matrix C, we allocate memory to store it in a sparse format. As we iterate through the entries of C, only non-zero values are computed and inserted into the structure. A temporary variable, `counter`, keeps track of the current index within the active row or column.

Since the iteration order over C can be easily modified by swapping the i and j loops, as shown in Algorithm 14, we maintain the flexibility to output the matrix in any of the supported formats. By combining a template parameter with C++'s `if constexpr`

Algorithm 13 Optimised non-zero counting in decompression phase

```
 1: positive ← 0
 2: negative ← 0
 3: for t ← 0 to d − 1 do
 4:     result ← calculate result
 5:     if result ≠ 0.0 then
 6:         s ← s_1[t](i) · s_2[t](j)                ▷ only calculate sign if needed
 7:         result ← result · s
 8:         if result < 0.0 then
 9:             negative ← negative + 1
10:         else
11:             positive ← positive + 1
12:         end if
13:     end if
14: end for
15: if negative ≥ ⌊d/2⌋ or positive ≥ ⌊d/2⌋ then      ▷ non-zero found
16:     indptr[i+1] ← indptr[i+1] + 1
17: end if
```

Algorithm 14 Sparse Filling

```
    #pragma omp for schedule(guided)
 1: for i ← 0 to n_a − 1 do
 2:     counter ← C.indptr[i]
 3:     for j ← 0 to n_b − 1 do
 4:         result ← compute result
 5:         if result ≠ 0.0 then
 6:             add result to C at row_counter + counter
 7:             row_counter ← row_counter + 1
 8:         end if
 9:     end for
10: end for
```

branching, we can generate format-specific implementations at compile-time. This eliminates runtime branching overhead, albeit at the cost of slightly increased binary size. Moreover, this design allows for straightforward extension to support additional sparse formats in the future.

5.4.2.1 Filling CSR

To fill out the CSR structure, we use the `indptr` array along with the `indices` and `data` arrays as follows in Algorithm 15:

Algorithm 15 CSR Entry Insertion

```
1: if result ≠ 0.0 then
2:     C.indices[row_counter] ← j
3:     C.data[row_counter] ← result
4: end if
```

Since we iterate over each row left to right, C will be sorted in row-major order. Additionally since we only iterate over each element once, no duplicates are introduced, thus leading to a *canonical* output matrix.

5.4.2.2 Filling COO

To achieve COO output, we adapt the CSR implementation by also filling the `x` indices, as shown in Algorithm 16. Since `indptr` is no longer needed after the computation, it is deallocated before C is returned. This served more as a demonstration of the

Algorithm 16 COO Entry Insertion

```
1: if result ≠ 0.0 then
2:     C.x[row_counter] ← i
3:     C.data[row_counter] ← result
4:     C.y[row_counter] ← j
5: end if
```

flexibility of our implementation. Due to COO being less memory efficient than CSR, as shown in section 3.2.2, we will focus on the CSR output from here on.

5.4.2.3 Considerations for CSC

Unlike CSR and COO, supporting the CSC format would require a more substantial modification to the accumulation strategy due to its column-major storage layout. Specifically, this would involve counting the number of non-zero entries in each column of C, which can be achieved by reordering the loops in Algorithm 6. We leave the implementation of this extension for future work.

6

Results

In this chapter, we present the methodology used for our measurements and report the corresponding results. The observed trends and performance improvements are discussed in Chapter 7.

6.1 Computational Step Decomposition

To better understand the performance characteristics and to support further measurements, we decompose `compmatmul` into the following major steps and substeps:

- **Compression**:
  - **Binning**: The processing of each non-zero element by hashing and signing it before accumulating it into the corresponding bin.
  - **memset**: A standard library function used to reset working vectors to zero at the beginning of each iteration.
  - **Vector Operations**: Intel MKL routines for vector addition and multiplication, used to accumulate intermediate results.
  - **Lock**: OpenMP synchronization primitives that ensure thread-safe access when multiple threads write to shared memory locations. Cycles are primarily used when waiting for the lock to be released.
  - **Other**: Any residual CPU time not attributed to the other tracked substeps.
- **IFFT**: Part of the compression step in Pagh's algorithm. We separate this step because the IFFT computation is independent of the input and output matrices. It is executed on separately spawned threads and was not modified in our implementation.
- **Decompression**:
  - **Non-zero Counting**: A new step introduced in our sparse-aware implementations; not present in the original dense version.

- **Median Computation**: Computes the median value using the standard library's median function. This step accounts for approximately 34% of CPU time in the original decompression phase, making it a key performance bottleneck.
- **Signing**: Performs signing operation for size prediction, as described in Section 5.4.1.1, and contributes about 2% to the total decompression time.

- **Other**: Includes data conversion between Python and C++, memory allocation, FFTW plan creation, and initialization of shared variables.

Using the second group of datasets mentioned in Section 4.4, we benchmarked the original `compmatmul` implementation and measured the runtime of each individual step across a range of matrix sizes. The results are illustrated in Figure 6.1.

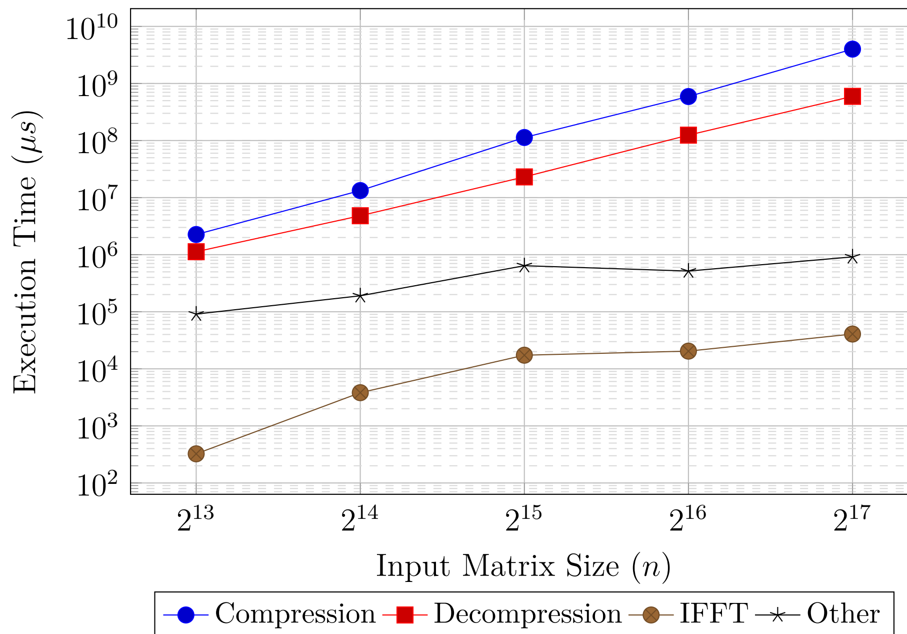

Figure 6.1: Runtime breakdown of four computational steps across different matrix sizes.

As shown, the **Compression** and **Decompression** phases dominate the overall execution time, with their runtimes growing significantly as the matrix dimension n increases. In contrast, the **IFFT** and **Other** components contribute relatively little to the total runtime and exhibit slower growth with increasing problem size.

This highlights the importance of optimizing the compression phase, which becomes increasingly performance-critical as matrix size increases.

6.2 Performance Characterization of the Compression Phase

To evaluate the effectiveness of our optimizations to the compression phase, we prepare six implementations for comparison ranging from the original baseline to increasingly optimized variants:

- **V0 Baseline**: The original dense-oriented implementation serving as the reference.
- **V1 Baseline Sparse** (Section 5.3.1): Implements sparse inputs.
- **V2 Empty-Row Skipping** (Section 5.3.2.1): Introduces row-skipping logic to avoid processing empty rows.
- **V3 Aliasing-Free Access** (Section 5.3.2.2): Improves compiler generated vectorisation code.
- **V4 Loop Reordering** (Section 5.3.2.3): Restructures loop nesting order to enhance cache locality.
- **V5 Merged Multiply-Add** (Section 5.3.2.4): Combines multiplication and addition operations into a single step for improved instruction-level parallelism.

Each implementation builds upon its predecessor, except V3, which will be discussed in more detail in the next chapter. This incremental approach allows us to isolate the performance impact of each change by incorporating all previously introduced optimizations.

6.2.1 Substep Analysis

To better understand where time is spent within the compression phase, we profiled its substeps using Intel VTune. The results, displayed in Figure 6.2, show the breakdown of runtime across different implementation versions. Notably, the *binning* substep becomes too small to reliably measure starting from V1, and similarly, *vector operations* become indistinguishable in V5. This does not imply that these steps consume zero runtime, but rather indicates that the number of measurable CPU events within those code regions fell below the threshold required for accurate profiling.

Profiling tools such as `perf` have limitations in restricting measurements to a single execution phase. To address this limitation, we modified the algorithm to skip the IFFT and decompression steps during cache-related evaluations. While certain auxiliary operations like memory initialization are not fully isolated from the measurement, analysis using Intel VTune confirms that their impact on the results is negligible.

Using the second group of the datasets, we measured the normalized runtime of

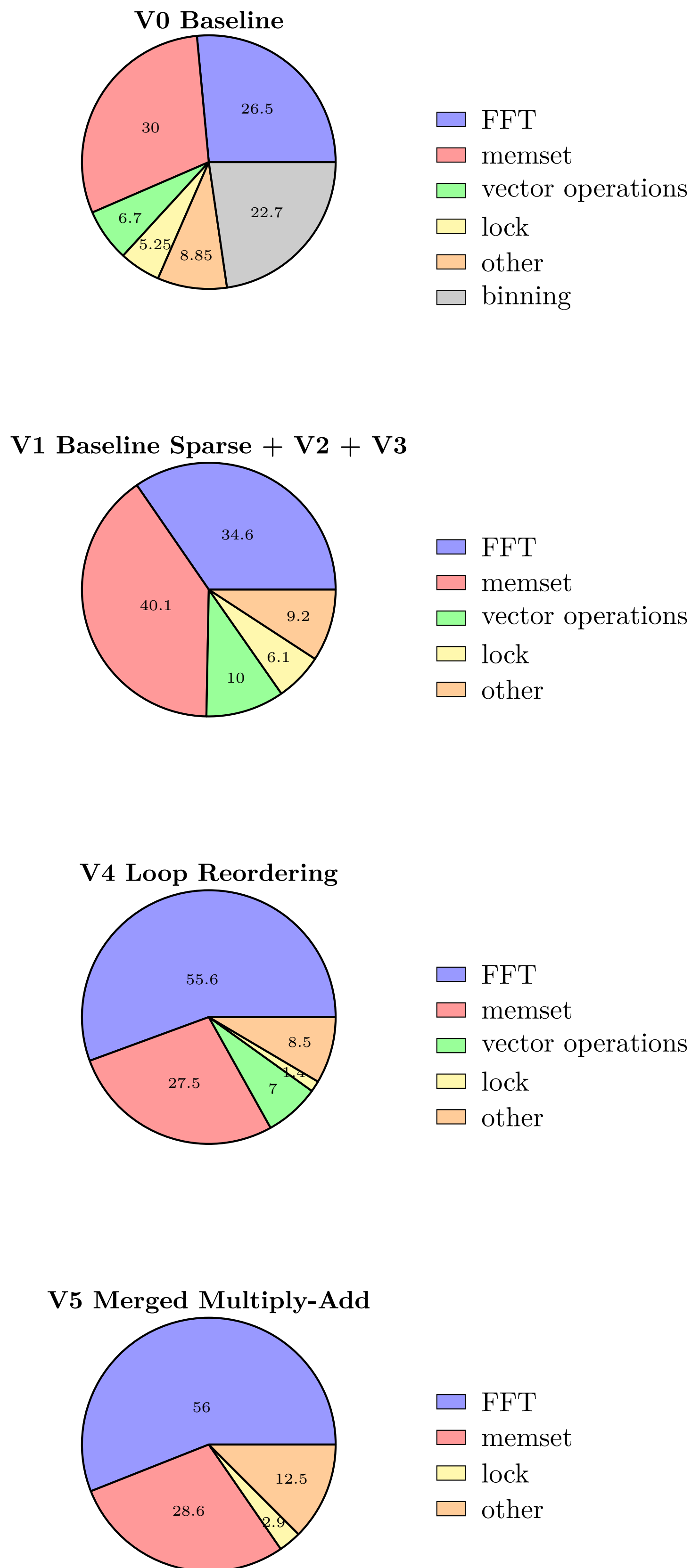

Figure 6.2: Runtime proportions of substeps in the compression step across different implementations at matrix size $n = 2^{16}$.

the compression phase across matrix sizes $n = 2^{13}$ to $n = 2^{17}$ for several sparse implementations, compared with V0, presented in Figure 6.3.

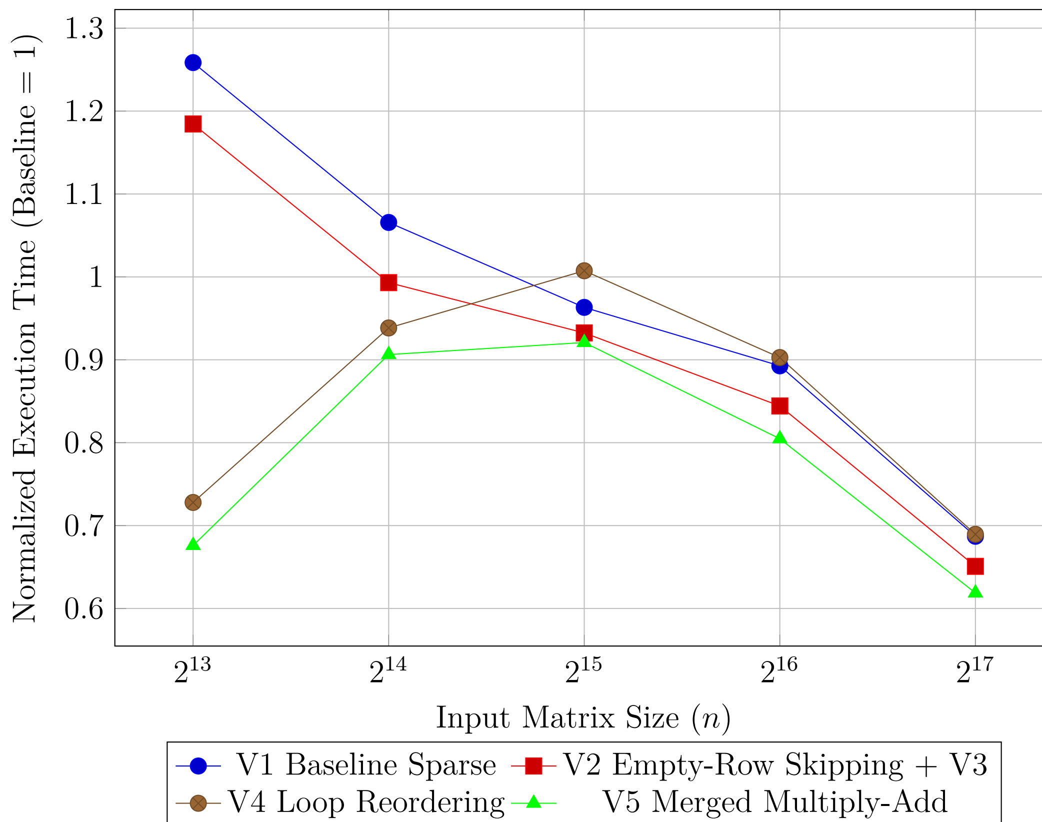

Figure 6.3: Runtime performance of the compression step across different implementations across and matrix sizes (Normalized to V0 Baseline).

6.2.2 Roofline-based Performance Modelling

Initial profiling using `perf` and Intel VTune indicated that a significant portion of CPU cycles were stalled due to memory-bound operations. To quantify this behavior, we used the **Roofline model** [41], a widely adopted performance analysis model that characterizes an application's computational efficiency by plotting its floating-point performance (FLOPs/s) against its memory bandwidth usage (GB/s). It reveals how closely an implementation approaches the theoretical performance limits imposed by the underlying hardware.

We applied the Roofline model to the compression phase using the `delaunay_n16` dataset, in order to determine whether the implementation is compute-bound or memory-bound. For this we measured the total number of floating-point operations and the volume of data transferred during execution. These metrics allowed us to compute the arithmetic intensity for each implementation and plot them on the Roofline diagram, using empirically measured memory bandwidth limits of Minerva's memory hierarchy.

We visualised the results for each implementation in terms of its arithmetic intensity

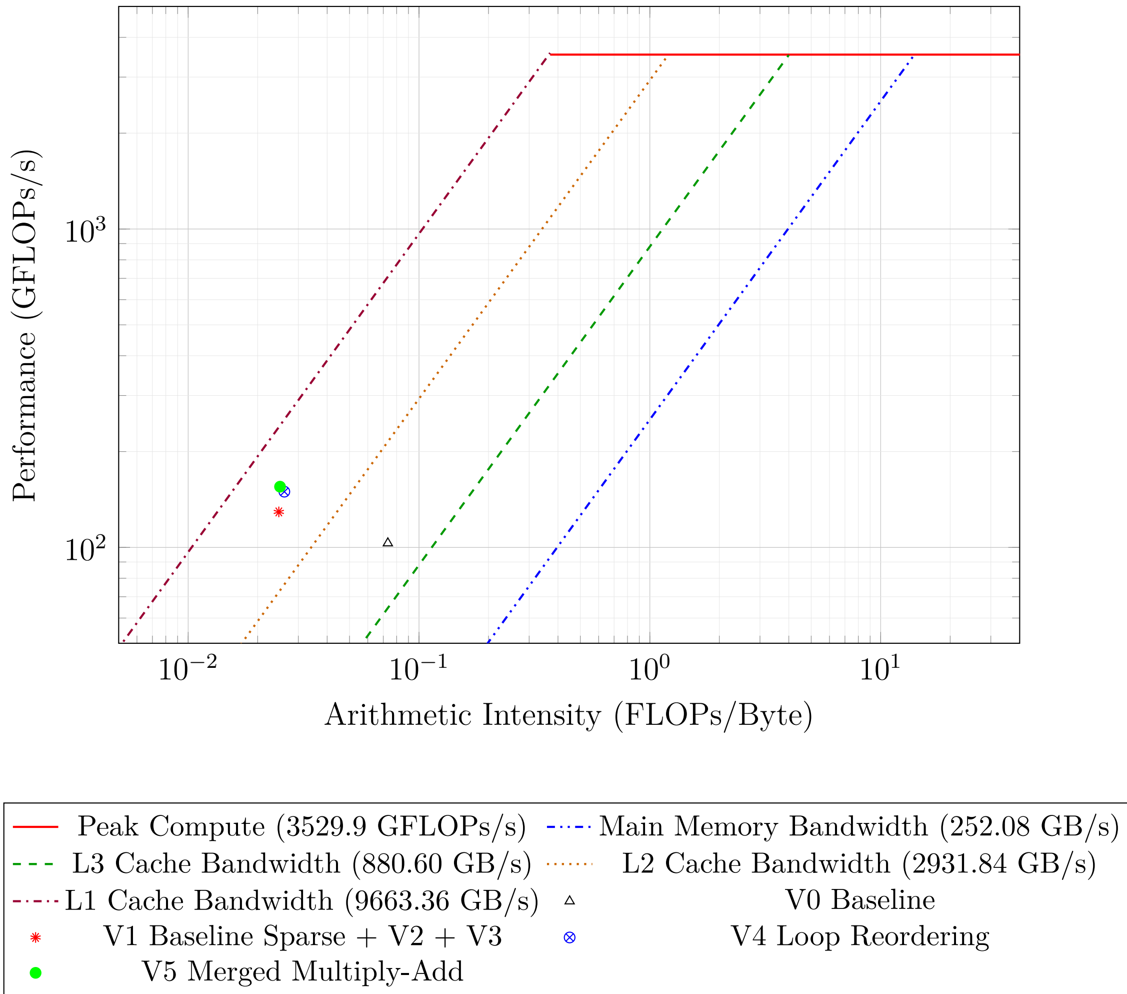

Figure 6.4: Roofline model of the compression step across different implementations at matrix size $n = 2^{16}$.

and achieved performance in Figure 6.4.

6.2.3 Cache Utilization

To better understand how our implementations interact with the memory hierarchy, we analyzed the distribution of data loads across different cache levels, using the second group of the datasets. Figure 6.5 shows the breakdown of data load sources (L1, L2, L3 caches, and the Main Memory) across various implementation versions and input sizes.

Prior work theorised that when input matrices fit within the L3 cache, it could have a significant impact on performance [4]. To improve readability and highlight this effect, we also present the L3 hit rate separately in Figure 6.6.

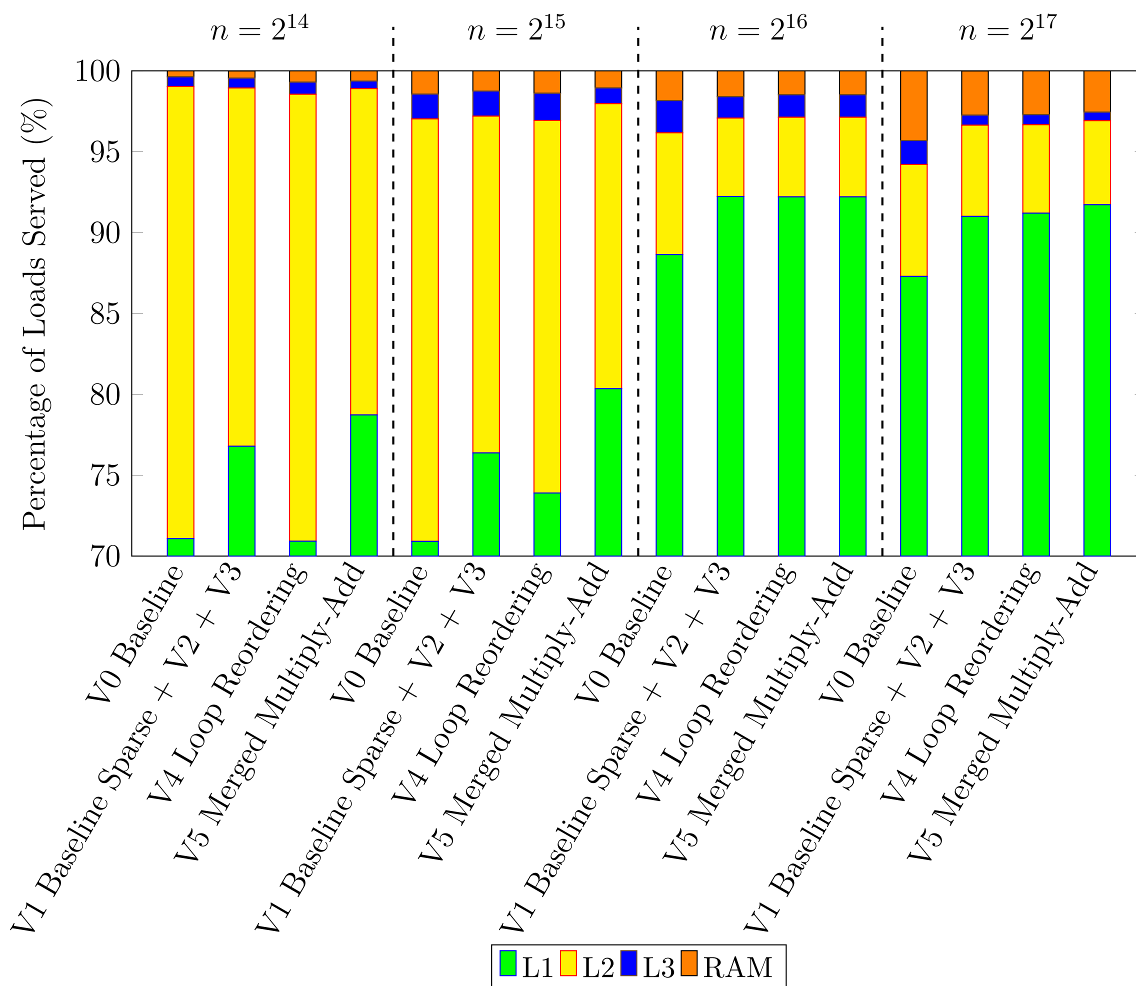

Figure 6.5: Data load distribution across the memory hierarchy of the compression step across different implementations and input sizes $n = 2^{14}$ to $n = 2^{17}$.

Note: The y-axis starts at 70% because most loads are served by the L1 cache.

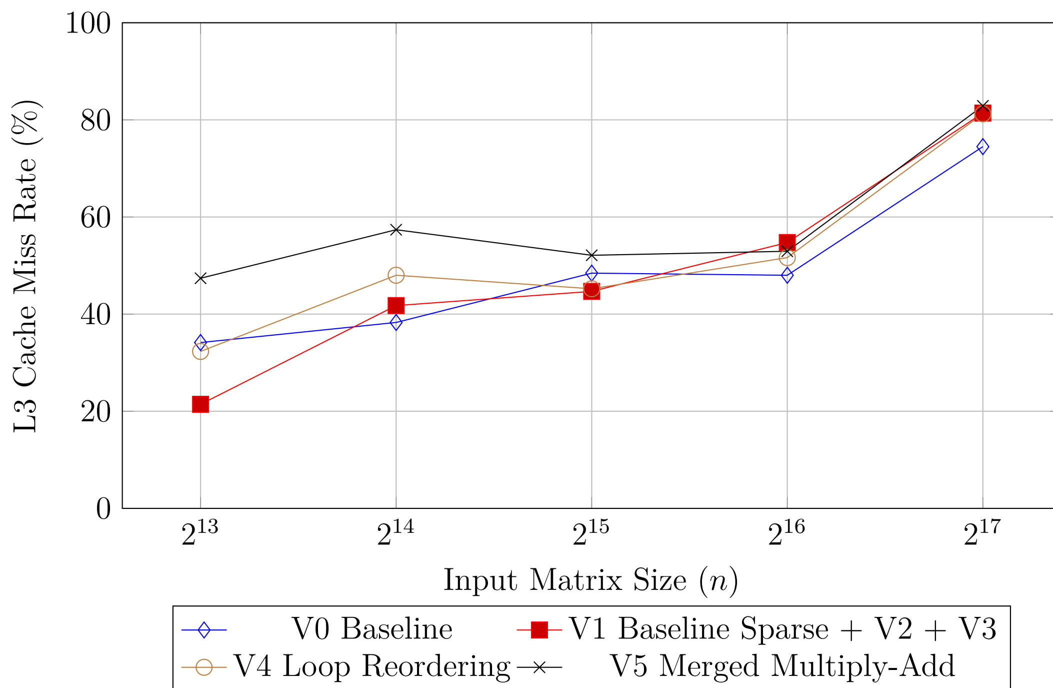

Figure 6.6: L3 cache miss rate of the compression step across different implementations and matrix sizes.

6.2.4 Runtime Sensitivity to Inputs

The runtime performance of our algorithm is not solely determined by the overall sparsity or density of the input matrices. Rather, it is heavily influenced by how non-zero elements are distributed across rows and columns. This sensitivity arises from two key factors:

1. **Workload Imbalance Across Threads:** An uneven distribution of non-zero elements among rows can lead to load imbalance during binning. In extreme cases, a single thread may process the majority of non-zero elements while others remain idle, severely limiting parallel efficiency.

2. **Performance Gains from Empty Rows:** Due to the V2 optimisation that skips empty rows entirely, inputs containing many such rows can result in runtime improvements, as unnecessary processing is avoided.

To better understand how sparsity and its structure affect performance, we conducted experiments using synthetic matrices. These allow us to control both the total number of non-zero elements and their distribution patterns across rows. Specifically, we focus on three distinct configurations:

- **Maximum r_0:** concentrates non-zero elements into as few rows as possible, maximizing opportunities for row skipping. This represents a best-case scenario for skipping empty rows, and the worst case for unequal thread workload distribution.

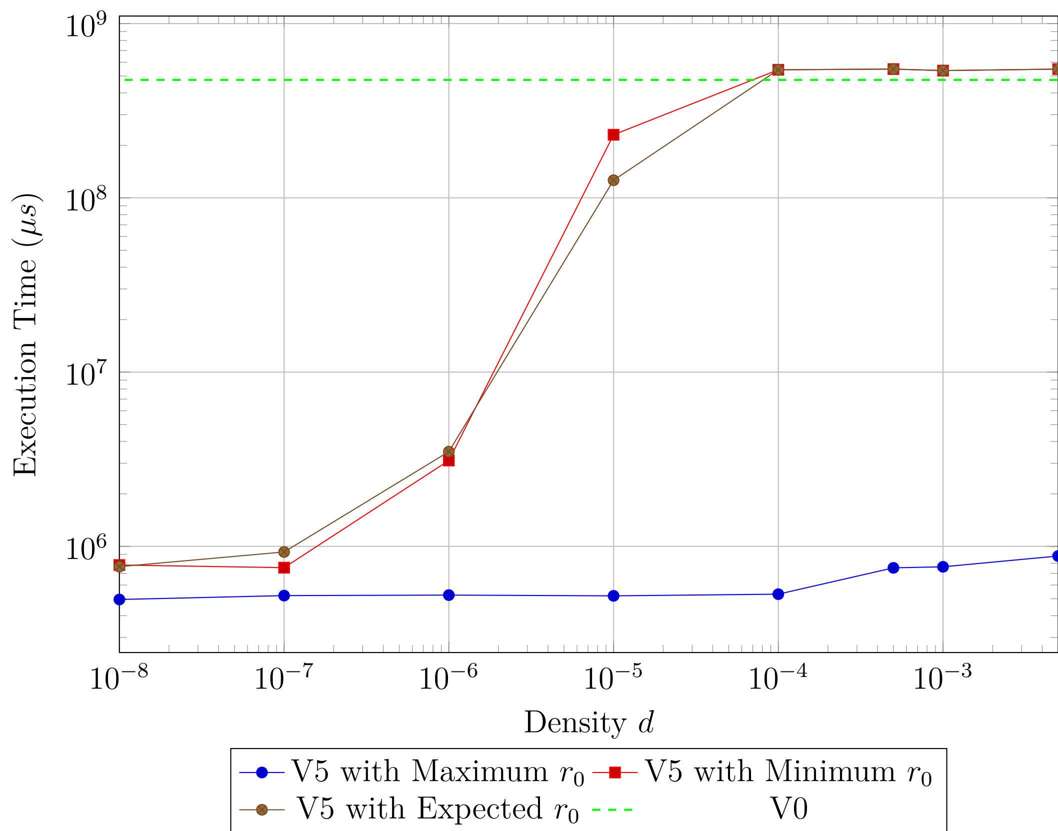

Figure 6.7: Runtime performance of the compression step across different inputs at matrix size $n = 2^{16}$.

- **Minimum r_0:** attempts to assign at least one non-zero element to each row minimizing potential gains from empty rows. This serves as a worst-case scenario for our optimisation.
- **Expected r_0:** randomly assigns non-zero elements to rows, preserving the overall sparsity while ensuring that the expected number of empty rows follows a binomial distribution. As a result, it generates matrices with variable row lengths that mimic realistic sparsity patterns, thereby providing a balanced and representative workload for performance evaluation.

For each configuration and target density, we generated two random matrices of size $n = 2^{16}$ and measured the compression time using the V5 implementation. The results are shown in Figure 6.7, with the baseline V0 implementation included for comparison.

6.2.5 Runtime Sensitivity to Parameters

The runtime of Pagh's algorithm is highly dependent on two key parameters: the number of sketch repetitions (d) and the length of the sketch vector (b). These parameters control the redundancy and accuracy of the sketch, directly influencing both the accuracy of the result and the computational overhead.

The goal of this section is to empirically evaluate how variations in d and b affect runtime performance when compared to the dense matrix multiplication baseline implemented in `compmatmul`. The results could provide possible guidance for practical parameter selection and highlight associated trade-offs in execution time.

To achieve this, we perform a grid search over a range of d_c and b_c values (which determine d and b, mentioned in Section 3.6.5) using our optimized implementation V5. All experiments are conducted on the `delaunay_n17` dataset. To enable fair comparison across different configurations, we normalize execution time with respect to the V0 baseline. The results are visualized in Figure 6.8.

6.3 Comparison Against Existing Libraries

To evaluate the practical performance of our approach, we benchmark our implementations against three widely used sparse matrix computation libraries: Intel MKL, Eigen, and SuiteSparse:GraphBLAS. These libraries are highly optimized for sparse operations and are commonly used as performance baselines in both academic research and industry applications.

Our objective is to assess how our compressed matrix multiplication method performs in terms of runtime of pure multiplication process across various datasets, and to determine whether the potential benefits justify the overhead introduced.

To compare with the three established libraries, we evaluate two versions of compmatmul: the baseline V0 and the optimized V5 variant.

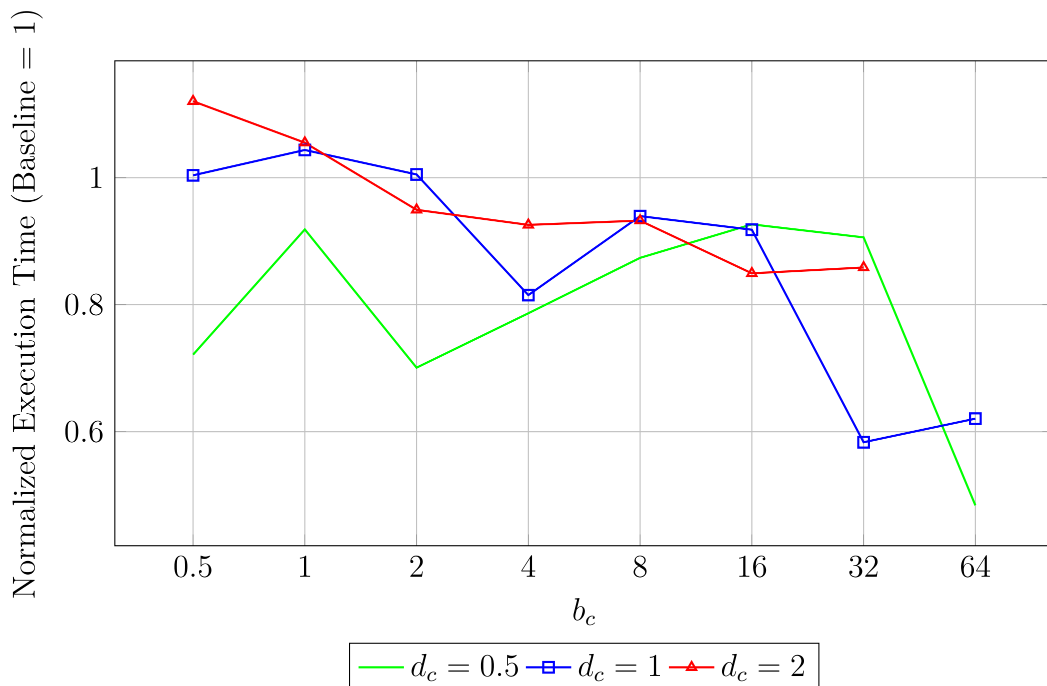

Figure 6.8: Runtime performance of the compression step using V5 across different combinations of b_c and d_c (Normalized to V0 Baseline).

For input data, we selected the first group of datasets, shown in Section 4.4, which consist of large-scale matrices. The motivation for using real-world datasets is to reflect realistic computational scenarios and enable a meaningful comparison with state-of-the-art libraries under actual application workloads.

We fix the sketching parameters to $d = 2$ and $b = 16384 = 2^{14}$, which satisfies the lowest requirement (namely, $d_c = 0.1$ and $b_c = 0.1$) for the datasets selected, as mentioned in Section 3.6.5, and present our results in Figure 6.9.

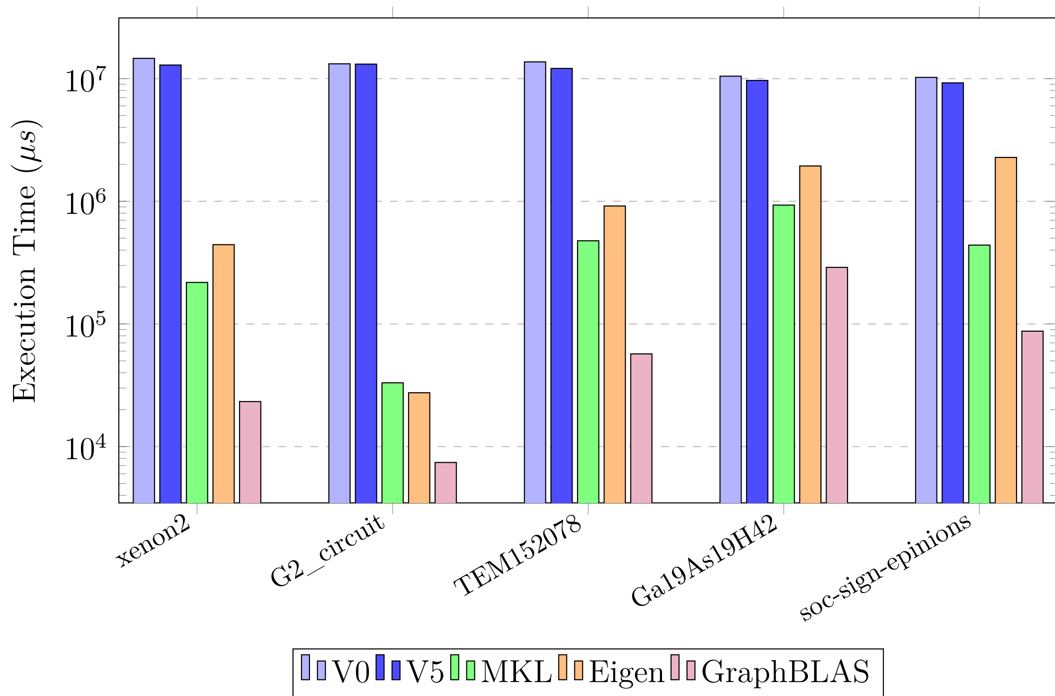

Figure 6.9: Logarithmic runtime performance of pure multiplication process across different libraries and datasets.

7

Discussion

In this chapter, we begin with a general analysis that sets the stage for understanding the overall outcomes. We then proceed to dissect each variant in detail, offering insights into their specific behaviours and performance characteristics. Other comparative experiments and theoretical underpinnings are also discussed. Additionally, we examine potential path for performance optimization and reflect on the limitations of our current work and propose promising directions for future research.

7.1 General Analysis

For all variants of our implementation, the arithmetic intensity (AI) remains very low (significantly below 1 FLOP/Byte), placing them firmly within the bandwidth-bound regime of the Roofline model. Analysis using Intel VTune revealed that store instructions account for less than 20% of total memory accesses and are not a major limiting factor for throughput. This further confirms that performance is primarily constrained by data loading operations.

The vast majority of memory accesses are served by the L1 and L2 caches, especially for smaller input sizes ($n = 2^{14}$ to $n = 2^{16}$), indicating that `compmatmul` effectively utilizes fast on-chip memory for most operations. However, the L3 cache serves only a small proportion of loads and exhibits a high miss rate. As the input size increases, the L3 miss rate rises significantly across all implementations, suggesting that larger working sets place greater pressure on the memory hierarchy and lead to more frequent access to slower memory levels. Surprisingly, our proposed optimizations have minimal impact on the L3 hit rate, and adopting the sparse implementation leads to a noticeable decrease in cache hit rate.

As input size increases, the L1 cache hit rate also improves significantly across all versions. This implies that a growing portion of load operations exhibit strong data locality, thereby benefiting from faster cache access.

Despite using the minimum recommended values for parameters d_c and b_c, our implementation is significantly outperformed by all compared libraries, by 1-2 orders of magnitude. This highlights the need for further optimization to close the performance gap.

7.2 Analysis by Variant

Understanding how each optimization affects memory behavior is crucial for reducing bandwidth stalls and improving overall runtime efficiency.

7.2.1 V0 Baseline

The dense implementation of `compmatmul` fails to fully utilize the floating-point capabilities of Minerva, achieving less than 10% of the peak performance measured by the LINPACK benchmark. This inefficiency stems from the extremely low arithmetic intensity, which places the implementation deep in the memory-bound regime.

Additionally, the reliance on slower memory levels such as L2 and RAM, combined with a significant gap between observed performance and the theoretical GFLOPs at this arithmetic intensity level, suggests that substantial gains could be achieved by increasing data locality or reducing the working set size.

7.2.2 V1 Baseline Sparse

V1 variant introduces a sparse-specific data layout that reduces the working set size and avoids unnecessary operations such as hashing, signing, and reading non-zero values. For large matrices, this results in improved performance by effectively eliminating the binning step. However, it also reduces the arithmetic intensity, making the algorithm increasingly bandwidth-bound. These observations suggest that the binning phase itself exhibits relatively higher computational intensity compared to the rest of the computation.

In contrast, for smaller matrices, runtime performance deteriorates slightly. We attribute this degradation to thread imbalance caused by short execution times (less than 0.1 seconds). Since our primary focus is on large-scale inputs, we leave a more detailed investigation of this behavior to future work.

Furthermore, switching from the dense to the sparse implementation significantly improves the L1 cache hit rate across all input sizes. However, the L3 miss rate increases for larger datasets while decreasing for smaller ones.

7.2.3 V2 Empty-Row Skipping

This optimization speeds up the compression phase by approximately 6–7% across all input sizes. However, this gain is lower than the approximately 12% proportion of empty rows in the datasets, suggesting that the improvement does not scale linearly with the number of skipped rows.

Since this change does not modify memory access patterns or computation structure, further characteristics, such as the memory usage, remain unchanged.

7.2.4 V3 Aliasing-Free Access

While our aliasing improvements enabled the compiler to replace scalar instructions with vectorized ones, no measurable performance improvement was observed. Without this change, over 90% of the executed floating-point instructions already utilize the longest supported vector length (512 bits), leaving limited potential for further vectorisation. Moreover, vectorisation only improves core throughput, which is not the performance bottleneck in our case.

As a result, we reverted these changes and excluded them from subsequent experiments.

7.2.5 V4 Loop Reordering

Compared to the V2 variant, loop reordering yields a modest improvement in throughput. The arithmetic intensity remains unchanged, which is expected, as only the order of loop iterations was modified without altering the number of floating-point operations or memory accesses.

The performance gain primarily stems from improved cache reuse during `memset` and vector operations, particularly benefiting smaller datasets. Additionally, due to the short execution time of these operations, lock contention is reduced, further contributing to the observed speedup.

Compared to V1 and V2, this optimization showed significant improvements in runtime for smaller datasets but led to performance degradation for larger ones. In experiments with other datasets, we observe slight improvements; however, we omit them here for brevity. Further investigation is required to determine the conditions under which this optimisation proves effective.

7.2.6 V5 Merged Multiply-Add

Variant V5 also delivered notable performance improvements. It reduced the time spent on vector operations to the point where they became negligible. As expected, this led to a slight increase in lock spinning, as threads now spend more time waiting for the completion of combined multiplication and addition steps. However, this overhead does not negate the overall performance benefits gained from avoiding repeated memory loads.

This optimization also increased the L1 cache hit rate across all input sizes. This is expected, as the multiplication operation no longer requires reloading data after the addition step. The improvement is more pronounced for smaller matrices, likely because the operands used in addition remain in the L1 cache. This also explains the higher L3 miss rate observed for smaller sizes: as more loads are satisfied at the L1 level, fewer accesses reach the L3 cache, and those that do are less likely to find recently accessed data still resident there.

The effectiveness of this optimization may vary depending on system architecture.

Our benchmarks were conducted on a dual-CPU system; however, single-CPU systems with lower latency for shared variables are likely to experience even greater performance gains.

7.2.7 Potential for Further Improvements

The V5 variant operates very close to the L1 cache bandwidth limit, suggesting that it already utilizes a substantial portion of the theoretically attainable memory bandwidth. Even under an idealized scenario in which all memory accesses are served exclusively from the L1 cache, the maximum potential performance gain would be limited to approximately 45%. Achieving this upper bound, however, is impractical due to the size of the working set, which exceeds the capacity of the L1 cache.

Therefore, future optimization efforts should prioritize reducing the amount of memory accessed during execution. At this stage, the FFT and `memset` operations account for the majority of the remaining runtime, making them the most promising candidates for further performance improvements.

7.3 Discussion on Other Experiments

This section provides a detailed examination of additional experimental results that shed light on the runtime behavior of our implementation under varying conditions.

7.3.1 Runtime Sensitivity to Inputs

The runtime of the compression step is highly sensitive to both the density and distribution of non-zero elements in the input. Our experiments demonstrate that execution time can vary by up to three orders of magnitude depending on these factors.

The optimized V5 variant exhibits a notable performance degradation at much lower densities than predicted by the threshold where dense storage becomes more efficient, established in Section 3.2.2. This early performance drop indicates that the overhead of accessing data from three distinct memory locations dominates computation earlier than expected, thereby diminishing the potential benefits of the optimization. This behaviour is consistently observed for the minimum and expected r_0 configuration.

The maximum r_0 configuration, which concentrates non-zero elements into few rows as possible, achieves the shortest execution times. This highlights the performance improvements achieved by skipping large portions of the input and confirms that thread-level load balancing is not a significant concern in this scenario.

7.3.2 Runtime Sensitivity to Parameters

The results of our parameter benchmarks are highly erratic overall. However, we observe one consistent trend: the execution time of the compression step generally decreases as b_c increases, regardless of d_c. This suggests that increasing the sketch

size leads to better relative performance, likely due to improved cache utilization of the smaller `pa` and `pb`. Conversely, configurations with larger d_c and smaller b_c tend to result in degraded performance. Notably, the setting $d_c = b_c = 2$ represents a performance sweet spot, yielding a 30% performance improvement on this dataset.

7.3.3 Comparison Against Existing Libraries

The runtime of pure matrix multiplication varies significantly across implementations on all datasets. Our optimized variant V5 consistently outperforms the baseline V0, demonstrating that the introduced optimizations effectively reduce computational overhead.

However, despite using the smallest recommended values for parameters d and b, our implementation remains significantly slower than all compared libraries (Intel MKL, Eigen, and SuiteSparse:GraphBLAS).

Below is a detailed breakdown of key observations from this comparison:

- **Performance Trends in V0 and V5**:

  Both V0 and V5 exhibit decreasing runtimes with smaller matrix sizes. However, their performance is not strongly influenced by the number of non-zero elements, indicating that sparsity alone does not dominate execution time.

- **Impact of Optimization (V0 vs. V5)**:

  The transition from V0 to V5 results in substantial performance improvements, particularly on larger datasets. We attribute these gains to reduced redundant computation and improved data locality. However due to the small d_c and b_c selected, the improvements are not as large as previous benchmarks.

  The consistent speedup across all datasets suggests that the optimizations address fundamental algorithmic bottlenecks rather than being specific to any particular input structure.

- **Comparison with Existing Libraries**:

  Among the three compared existing libraries, GraphBLAS performs competitively.

  When comparing matrices such as `G2_circuit` and `xenon2`, which have nearly identical dimensions, we observe significant differences in runtime due to variations in densities, differing by almost an order of magnitude.

  In comparisons involving `xenon`, `G2_circuit`, and `Ga19As19H42`, it becomes evident that the number of non-zero elements plays a critical role in determining performance. This suggests that these libraries scale more efficiently with sparsity.

- **Dataset-Specific Behavior**:

Certain datasets, such as `soc-sign-epinions`, exhibit unexpected performance characteristics. Despite having the smallest matrix dimensions and the second-lowest number of non-zeros, it performs worst across all three libraries compared to other datasets.

This behaviour underscores the sensitivity of existing libraries to structural properties such as irregular sparsity patterns or access behaviours, suggesting that performance cannot be predicted solely based on matrix size or density, and that the other libraries use other optimisation techniques that we have not considered.

Overall this indicates that the compared libraries are able to exploit further optimisations based on the structure of the inputs. Further research is required on weather these optimisations can be applied to *compmatmul*.

7.4 Theoretical Analysis

This section presents a comprehensive theoretical and algorithmic analysis of our modifications to `compmatmul`.

7.4.1 Runtime Complexity

We complement our empirical runtime measurements with an asymptotic runtime analysis.

7.4.1.1 Compression Phase

In the compression phase, as shown in Table 7.1, our modified binning strategy significantly improves the asymptotic runtime complexity by reducing the dominant term to $\mathcal{O}(d \cdot N)$. However, in contrast, the computational complexity of the FFT and IFFT steps remains unchanged at $\mathcal{O}(d \cdot b \cdot \log b)$.

Step	Original (From Dense)	From Sparse
Binning	$\mathcal{O}(p \cdot d \cdot (n+m))$	$\mathcal{O}(d \cdot (N_A + N_B))$
FFT	$\mathcal{O}(d \cdot b \cdot \log b)$	$\mathcal{O}(d \cdot b \cdot \log b)$
IFFT	$\mathcal{O}(d \cdot b \cdot \log b)$	$\mathcal{O}(d \cdot b \cdot \log b)$
Total	$\mathcal{O}(p \cdot d \cdot (n+m))$	$\mathcal{O}(d \cdot max(N_A, N_B, b \cdot \log b))$

Table 7.1: Comparison of asymptotic runtime for original `comptmatmul` and our modifications in the compression phase.

This complexity analysis reveals several important insights:

- **Improved Scalability with Sparsity:** The sparser the input matrices (i.e., $N \ll mn$), the greater the the benefit from the sparse modification. This enables scaling to much larger problem sizes under similar computational constraints.

- **Performance Dependency on Matrix Dimensions:** The runtime of the sparse binning step becomes independent of the matrix size. As long as the amount of non-zero elements is maintained, larger matrices can be processed without a proportional increase in computation time.
- **Bottleneck Shift:** When $N > b \log b$, the binning substep remains the performance bottleneck. However, if $N < b \log b$, then the FFT and IFFT steps dominate the runtime.
- **Impact on Dense Inputs:** For dense inputs where $N \sim mn$, the benefit of the sparse binning strategy diminishes, and the asymptotic runtime remains identical.

7.4.1.2 Decompression Phase

In the decompression phase, we introduce an additional non-zero counting step that eliminates the need for the $\mathcal{O}(d \cdot \log d)$ `std::median` function. However, as the filling stage remains unchanged, the overall runtime complexity remains the same in both the original and optimized implementations.

Step	Original (to Dense)	To Sparse
Non-zero Counting	-	$\mathcal{O}(d \cdot m \cdot n)$
Filling	$\mathcal{O}(m \cdot n \cdot d \cdot \log d)$	$\mathcal{O}(m \cdot n \cdot d \cdot \log d)$
Total	$\mathcal{O}(m \cdot n \cdot d \cdot \log d)$	$\mathcal{O}(m \cdot n \cdot d \cdot \log d)$

Table 7.2: Comparison of asymptotic runtime for original `comptmatmul` and our modifications in decompression phase.

Unlike the compression phase, where sparsity leads to significant improvements, the decompression phase exhibits consistent asymptotic behaviour regardless of output and input density.

7.4.2 Memory Requirements

Our modified implementation can operate on larger matrix sizes where the original version of `compmatmul` would have failed due to memory constraints. Table 7.3 compares the memory requirements across different variants of the algorithm based on the insights gained in Section 3.2.2.

<table>
<tr><th>Step</th><th>Dense-to-dense</th><th>Sparse-to-dense</th><th>Sparse-to-sparse</th></tr>
<tr><td>Input</td><td>$8 \cdot (n \cdot p + p \cdot n)$</td><td colspan="2">$12 \cdot (N_B + N_A) + 4 \cdot (m + p)$</td></tr>
<tr><td>Output</td><td colspan="2">$8 \cdot (n \cdot m)$</td><td>$12 \cdot (N_C + m)$</td></tr>
<tr><td>Scratchpad</td><td>$8 \cdot (b \cdot d \cdot t)$</td><td colspan="2">$8 \cdot (b \cdot t)$</td></tr>
</table>

Table 7.3: Comparison of memory requirements across different `compmatmul` variants, in bytes.

The transition from dense-to-dense to sparse-to-dense significantly reduces the memory requirement for the input matrices, providing their density falls below the $\frac{2}{3} - \frac{1}{3p}$ threshold discussed in Section 3.2.2. The transition to sparse-to-sparse further reduces the memory requirement of C for sparse output matrices.

The memory used for scratchpad during computation is significantly reduced due to the loop reordering strategy.

These combined optimisations allows our sparse-to-sparse implementation to scale to much larger problem sizes without exhausting available memory. This is particularly valuable in resource-constrained environments or for handling with very large matrices.

Among all variants, only the **sparse-to-sparse** decouples its memory requirement from the dense size of the processed matrices. This makes it particularly well-suited for applications requiring sustained sparsity in downstream computations, such as iterative solvers or deep learning layers with sparse activations.

7.4.3 Input Format Requirements

The requirement for matrices A and B to be provided in CSC and CSR formats, respectively, imposes certain constraints on input preparation. However, this limitation is relatively mild compared to the preprocessing overhead associated with earlier dense variants of `compmatmul`.

We identify two main usage scenarios based on the format of the input data:

- **Sparse Input Formats:** If the user already has data in another sparse format, converting it to CSR or CSC incurs an asymptotic cost of $\mathcal{O}(N + n)$ for matrix A, and $\mathcal{O}(N + m)$ for matrix B. This is significantly more efficient than the $\mathcal{O}(n \cdot m)$ cost required by the previous implementation to transpose dense inputs.
- **Dense Input Formats:** When starting from dense matrices, conversion to CSR/CSC costs $\mathcal{O}(n \cdot m)$ runtime, which matches the cost of transposing the input as required by the original dense version. Therefore, asymptotically, there is no performance gain or loss in this case.

7.4.4 Metadata Size

We chose not to implement 64-bit metadata types in our current version, as they are only necessary under specific conditions: namely, when $N_A > 2^{32}$, $N_B > 2^{32}$, or $p > 2^{32}$. These cases are computationally infeasible even with our implementation and were therefore not prioritized.

However, this limitation can be addressed with effort by introducing a separate `struct` and using template parameters to support 64-bit indexing when required.

Alternatively, if only $N_A > 2^{32}$, one could represent A^T in a 32 bit COO format

and process it with a slightly modified compression step. This would still be more memory-efficient than a 64 bit CSR data structure.

An alternative design could also support the multiplication of dense and sparse matrices, further extending the applicability of the algorithm.

7.4.5 Canonical Format

Our implementation does not require A^T and B to be in canonical row-major or column-major formats.

However, duplicate non-zero entries will be binned multiple times, leading to over-counting and inaccuracies in the final result. Therefore, while our algorithm is robust to input ordering, it assumes that the input matrices are free of such duplicates.

Zero-valued elements do not affect the output correctness but may degrade performance by increasing memory bandwidth usage and preventing efficient detection of empty rows. In particular, zero elements within otherwise empty rows may prevent those rows from being classified as empty, leading to unnecessary computation during subsequent stages.

7.4.6 Hash Function Usage

The frequency of hashing and signing function calls varies significantly across different implementation modes, as summarized in Table 7.4.

Step	**Dense-to-dense**	**Sparse-to-dense**	**Sparse-to-sparse**
Compression	$2d \cdot m \cdot n$	$2d \cdot N$	$2d \cdot N$
Decompression	$2d \cdot m \cdot n$	$2d \cdot m \cdot n$	$4d \cdot m \cdot n$
Total	$4d \cdot m \cdot n$	$2d \cdot (m \cdot n + N)$	$2d \cdot (2m \cdot n + N)$

Table 7.4: Comparison of hashing and signings calls across different `compmatmul` variants.

The Sparse-to-dense variant significantly reduces the number of hashing operations during compression compared to the original Dense-to-dense implementation, and thus places lower demands on hash function performance. In contrast, the Sparse-to-sparse variant doubles the number of operations during decompression, leading to a much higher total cost.

Therefore, when designing or selecting hash functions for use with `compmatmul`, it is important to consider the target execution mode. All else being equal, for Sparse-to-dense, slower but collision-resistant hash functions may be more acceptable, whereas for Sparse-to-sparse, lightweight and high-throughput hash functions might be more preferable to avoid introducing significant overhead.

7.4.7 Impact of Data Precision

A 32-bit floating point version of `compmatmul` was tested in previous work. If this approach is further explored in future research, it may lead to improvements in memory bandwidth and cache utilization, as twice as many floating-point values could be stored and transferred while using the dense implementation.

However, in our sparse implementation, the potential gains from reduced floating-point precision would be more limited. This is due to the metadata components of the sparse matrix formats which do not scale with the size of the floating point values, as discussed in Section 3.2. As a result, the overall memory footprint reduction would be smaller than expected based on solely the compression of floating-point data.

7.4.8 Considerations for Non-Square Matrices

While our measurements were limited to square matrices, the implementation naturally extends to general sparse matrix multiplication involving non-square inputs. We observe that the runtime is most sensitive to changes in the number of rows n in matrix A due to the following reasons:

- Larger n results in increased memory pressure during binning, as each row requires temporary storage and processing. This can significantly degrade performance due to reduced cache efficiency.
- Larger n also increase the likelihood of rows not being empty (assuming a random distribution of non-zero elements).
- In contrast, increases in m or p mainly affect the size of the input matrices and the total number of non-zero elements, but do not increase per-row memory usage or cache requirements.

Thus overall, our algorithm is much more sensitive to n than the other matrix dimensions.

7.5 Potential Performance Optimizations

This section outlines a set of further performance optimizations that can be applied to our implementation.

7.5.1 FFTW Usage

One of the most significant performance bottlenecks identified in our implementation is the use of FFTW.

In the current `compmatmul` implementation, each thread creates $2 \cdot t$ independent FFTW plans using the `FFTW_MEASURE` planner: one FFT and one IFFT plan per thread. Given that all plans share identical transform size, type, and memory layout, this approach introduces redundant planning overhead in both time and memory.

Additionally, each plan operates on a fixed preallocated buffer, which requires copying input data before execution and result data after execution. This leads to unnecessary memory traffic and cache pressure.

The following optimizations are proposed to address these issues:

- **New-array Execute Functions:** FFTW supports execution functions that allow specifying an arbitrary memory address at runtime, instead of binding the plan to a fixed buffer during creation. Using these functions can eliminate the need for data copying by allowing direct pointer passing to the input and output locations [47].
- **Sharing Plans Between Threads:** Currently, each thread generates its own FFTW plan. The size of each plan depends primarily on the matrix dimension n, for $n = 2^{16}$ these require approximately 1.3 MB, while one for $n = 2^{17}$ requires around 2.2 MB. This results in substantial memory usage across 32 threads. Due to their large size, these plans do not fit within the shared L3 cache, forcing threads to load them from main memory. Instead of generating separate plans per thread, a single FFT and a single IFFT plan could be shared among all threads. FFTW plans are internally thread-safe for execution as long as they are not concurrently modified, making this optimization both feasible and safe [47]. This would reduce initialization overhead by enabling a single, turbo-boosted thread to generate the plan for all others, while also potentially improving L3 cache utilization and reducing cache miss rates.
- **FFTW Plan Many Interface:** Replacing multiple consecutive FFT calls with a single batched call using the `fftw_plan_many_dft` interface allows executing multiple FFTs of the same size and type simultaneously, leveraging internal optimizations for improved throughput compared to individual calls. The same approach can be applied to IFFT operations for similar benefits.

7.5.2 Thread Pooling

The current OpenMP-based implementation incurs overhead from thread creation and destruction at multiple stages of the algorithm. We believe this overhead can be reduced by employing a thread pooling mechanism, which reuses threads across different phases of execution.

In particular, threads created during the compression phase could be repurposed to perform the IFFT step with minimal modifications to the existing codebase. This would eliminate redundant thread initialization and teardown.

7.5.3 Explicit Memory Alignment Specification

Although the input data is already aligned to cache-line boundaries (64 bytes on modern x86 architectures) as required by FFTW and MKL, the C++ compiler cannot fully exploit this alignment within the `compmatmul` code without explicit directives.

To address this, we propose using the C++ `std::assume_aligned` intrinsic to explicitly convey the alignment properties to the compiler. This would allow the compiler to generate aligned memory access instructions, enabling more efficient vectorization.

Initial benchmarks of this modification showed no measurable runtime improvement. This is expected, as the change does not address the current main bottleneck, memory bandwidth constraints, and has no impact on FFTW and MKL calls, which already assume aligned data access. However, future modifications may alter this behaviour.

7.5.4 Split Collection of Polynomials

Currently, all threads write intermediate results into a single shared `p_fft` array, leading to frequent cache line invalidations and coherence traffic, particularly costly due to synchronization latency between CPUs.

To mitigate this issue, we propose allocating a dedicated `p_fft` instance for each CPU. This would prevent concurrent modifications from multiple threads, minimizing cache pollution and coherence overhead.

While this change would require an additional final step to merge the per-CPU results, the potential performance gains from reduced cache contention may outweigh the overhead of merging, which should be evaluated through benchmarking.

On modern AMD architectures, a similar strategy could be extended by allocating one `p_fft` copy per CCX (Core Complex), further reducing inter-core communication costs.

7.5.5 Sparse Transformation

During the compression phase of `compmatmul`, N_A non-zero elements of matrix A^T are hashed into b buckets using an independent and uniform hash function.

For a single bucket, the probability that a given non-zero element does not hash into it is $1-\frac{1}{b}$. Since the hash operations are independent across elements, the probability that none of the N elements land in a specific bucket is $\left(1-\frac{1}{b}\right)^N$. Therefore, the expected fraction of non-empty buckets becomes:

$$d_{pa} = 1 - \left(1 - \frac{1}{b}\right)^N$$

This value will be strictly less than $1 - e^{-1} \approx 0.632$ when $N = b$, and it decreases further when $N < b$. Assuming that $N = n \cdot p \cdot d_A$, and setting $n = p = b$, we can rewrite this expression as:

$$d_{pa} = 1 - \left(1 - \frac{1}{n}\right)^{n^2 \cdot d_A}$$

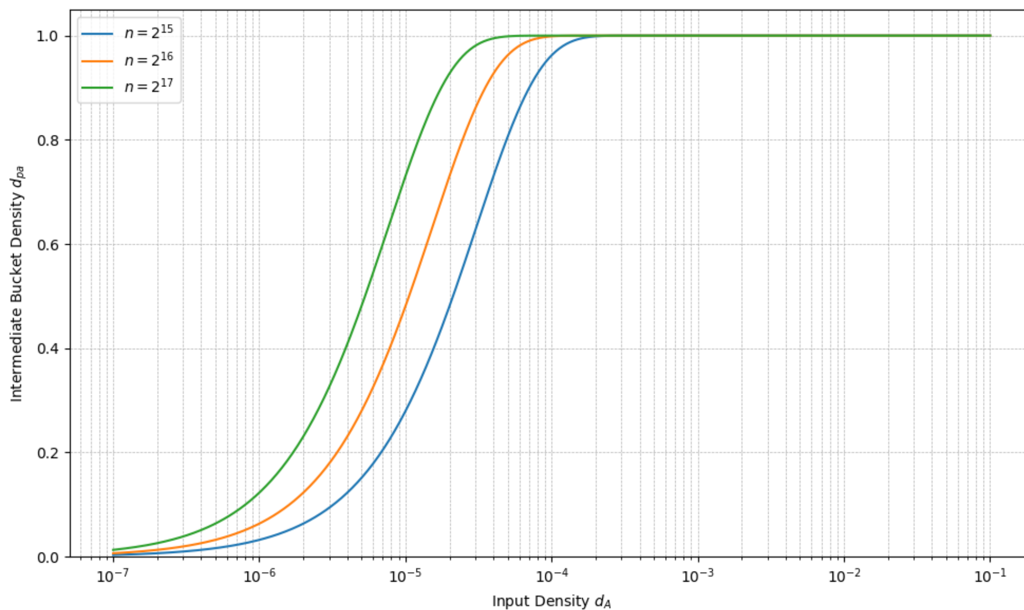

Figure 7.1: Intermediate bucket density d_{pa} as a function of input matrix density d_A.

Importantly, we observe that $d_{pa} > d_A$ for all $d_A \in (0, 1)$, meaning that binning a sparse, non-empty matrix always results in a higher intermediate bucket density than the original matrix. The degree of increase depends on both the number of buckets b and the constant b_c, as shown in Figure 7.1.

When processing very sparse data, the generated buckets are also highly sparse. By modifying the binning step to produce sparse rows instead of dense representations, and using a sparse FFT or FWHT transform, the overall computation can be significantly accelerated and memory consumption reduced.

The asymptotic runtime of these transforms, as well as the memory usage of the bins, would also become less dominated by the length of the polynomial, which in turn would allow for better scaling with larger b.

7.5.6 Skip Empty Rows with IFFT

Similar to how we skip empty rows during the compression phase, the same technique could be applied during the IFFT step. Specifically, rows containing no non-zero elements could be bypassed entirely, avoiding unnecessary computation.

This optimization was not pursued in the current implementation due to time constraints. Furthermore, given that the IFFT step contributes only a small fraction to the total runtime, any performance gains are expected to be modest.

7.6 Future Research Directions

This section presents several promising research directions for future exploration. These suggestions include advanced algorithmic techniques and theoretical enhancements that could significantly improve performance or broaden applicability, although many remain to be fully explored or validated in practice.

7.6.1 Efficient Decompression with Error-Correcting Codes

One promising direction for future work is the integration of error-correcting codes into the decompression phase. This approach builds upon the concept of sublinear result extraction introduced by Pagh [34], which enables efficient reconstruction of dominant entries without fully decoding the entire compressed structure.

The key idea lies in identifying and extracting only the most significant elements (e.g., those exceeding a certain threshold), rather than reconstructing the full matrix or vector. Structured hashing and coding schemes can be used to detect potential errors or deviations during decompression, allowing computation to focus exclusively on relevant components.

By incorporating such mechanisms, it may be possible to significantly reduce computational complexity. Specifically, instead of performing an exhaustive $\mathcal{O}(n^2)$ scan over all combinations, the search space could be narrowed down to a subset of candidates with high probability of significance. If a small overhead in time and space during compression is acceptable, this can yield a runtime improvement down to approximately $\mathcal{O}(l \log^2 n)$, where l is the number of target non-zero entries.

This method not only accelerates decompression but also maintains strong theoretical guarantees on approximation accuracy, especially when the data exhibits sparsity or near-sparsity in an appropriate domain.

However, implementation introduces several technical challenges, particularly in parallelizing the counting of element occurrences and filtering out infrequent ones. These aspects warrant further investigation.

7.6.2 Saving Indices to Avoid Redundant Computation

An alternative optimization involves reusing non-zero positions identified during the counting of the non-zero elements. Instead of computing the result of each cell twice, the indices of the counting run coudl be used to narrow down the calculations in the filling operations. This would require some form of dynamic data structure to cache these indices, which is why we chose not to implement it here. This would allow for the `median` function to be only called for confirmed non-zeros, thus improving the asymtopic runtime of the decompression step.

7.6.3 Cancellation-Aware Estimation of Non-zero Count

Alternatively, Pagh's randomized doubling strategy [34] provides a robust approach for estimating the number of non-zero entries in C, even in the presence of cancellations. The method begins by preconditioning the input matrices using random diagonal matrices D_1 and D_2, which ensures that distinct multiplication paths do not cancel each other out with high probability (based on the Schwartz-Zippel Lemma [48]).

Then, it iteratively computes CountSketches for increasing sketch sizes $b = 2, 4, 8, \dots$ until a large fraction (e.g., $\frac{4}{5}b$) of the buckets are observed to be empty. At this point, the true number of non-zero entries in AB is likely to be less than $\frac{b}{5}$, yielding an $\mathcal{O}(1)$-approximation.

This technique provides scalable and output-sensitive estimation of the non-zero count, addressing limitations of prior approaches that fail to account for cancellation effects. It relies on 2-wise independent hash and sign functions, which are sufficient for achieving high-probability correctness guarantees.

Due to the aforementioned complexity of allocating C with only a size estimate, we leave the implementation for future work.

7.6.4 Benchmarking Loop Optimizations with the Dense Implementation

The vectorisation merging, discussed in Section 5.3.2.4 does not rely on sparse-specific modifications and could, in principle, also benefit the dense version of `compmatmul`.

7.6.5 Relevance to FWHT Implementation

Most of our optimisations can be directly applied to the FWHT implementation of `compmatmul`. However, due to its different structure, which already incorporates a more efficient loop order compared to the FFT-based implementation, the effectiveness of our loop reordering, should be evaluated independently.

FWHT outperforming FFT is primarily dependent on if it can reduce the amount of memory reads required, as we are already close to the L1 roofline, leaving little room for further gains from memory locality improvements. Given the strong similarities between the butterfly operations in FWHT and FFT [49], we doubt that switching to FWHT would achieve this, as their memory accesses patterns are quite similar.

7.7 Numerical Accuracy

All modifications to the algorithm preserve its underlying mathematical formulation and are intended to produce results that are numerically consistent with those generated by the `compmatmul` implementation.

However, due to factors such as the varying order of generated random numbers, race conditions during parallel execution and differences in the sequence of floating-point

operations, the outputs may not be identical across runs or between different versions.

These discrepancies are expected and stem from the inherent non-determinism and precision limitations of floating-point arithmetic, rather than from any change to the core computational logic.

8

Conclusion

This thesis presents both theoretical and practical contributions to the field of approximate matrix multiplication, with a focus on R. Pagh's algorithm introduced in *Compressed Matrix Multiplication* (2013). We conducted a comprehensive study of three widely used sparse matrix storage formats (CSR, CSC, and COO) with an emphasis on their compatibility with Pagh's framework. This analysis laid the groundwork for extending the algorithm to better support sparse formats.

On the practical side, we benchmarked the performance of `compmatmul` on a Minerva cluster node and applied a roofline model to identify key performance bottlenecks. These findings guided our efforts to improve the algorithm's efficiency. We implemented support for sparse matrix multiplication where the left-hand side matrix is stored in CSR format and the right-hand side matrix is stored in CSC format. Additionally, we extended output compatibility to CSR and COO formats, further enhancing its integration potential with downstream applications that rely on sparse formats. We also introduced several optimizations targeting the compression phase of the algorithm, each supported by theoretical reasoning and empirical validation.

Our experimental results reveal that `compmatmul` is inherently bandwidth-bound. The use of sparse formats are more memory efficient compared to the dense format, especially when dealing with highly sparse data. However, despite these memory benefits and the expectation that sparse data would better fit in the L3 cache, we observed no corresponding improvement in L3 cache utilization, indicating that memory access patterns or other factors may still limit performance.

We observed that runtime is highly sensitive to the density and the distribution of non-zero elements in the inputs. Performance improvements become more pronounced as density decreases, indicating that the algorithm is best suited for applications involving highly sparse matrices.

Despite these gains, we identified a notable bottleneck in the decompression phase, which currently performs slower than the original implementation. However, we achieved asymptotic improvements in the compression phase. For scenarios where the resulting dense matrix exceeds available memory, the *sparse-to-sparse* implementation is preferred; otherwise the *sparse-to-dense* implementation is generally more efficient for sparse inputs.

Importantly, we emphasize that our modified algorithm retains mathematical equivalence to the original formulation. While efficiency improvements remain limited in certain contexts, the enhanced memory efficiency and broader compatibility with sparse data formats open new possibilities for practical deployment and further optimization.

In summary, this work successfully extends the scope of compressed matrix multiplication to sparse inputs and outputs, improving its adaptability and resource efficiency. Future research directions include refining the decompression process and exploring hybrid approaches outlined in the last chapter.

Bibliography

[1] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "A unified framework for numerical and combinatorial computing," *Computing in Science & Engineering*, vol. 10, no. 2, pp. 20–25, 2008. DOI: 10.1109/MCSE.2008.45.

[2] S. Graillat, F. Jézéquel, T. Mary, and R. Molina, "Adaptive precision sparse matrix–vector product and its application to krylov solvers," *SIAM Journal on Scientific Computing*, vol. 46, no. 1, pp. C30–C56, 2024. DOI: 10.1137/22M1522619. eprint: https://doi.org/10.1137/22M1522619. [Online]. Available: https://doi.org/10.1137/22M1522619.

[3] O. Selvitopi, S. Ekanayake, G. Guidi, G. A. Pavlopoulos, A. Azad, and A. Buluç, "Distributed many-to-many protein sequence alignment using sparse matrices," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–14. DOI: 10.1109/SC41405.2020.00079.

[4] M. Karppa, Personal communication, 2024.

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

[6] V. Y. Pan, "New fast algorithms for matrix operations," *SIAM Journal on Computing*, vol. 9, no. 2, pp. 321–342, 1980. DOI: 10.1137/0209027. eprint: https://doi.org/10.1137/0209027. [Online]. Available: https://doi.org/10.1137/0209027.

[7] V. Strassen, "The asymptotic spectrum of tensors and the exponent of matrix multiplication," *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pp. 49–54, 1986. [Online]. Available: https://api.semanticscholar.org/CorpusID:15077423.

[8] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," *J. Symb. Comput.*, vol. 9, no. 3, pp. 251–280, Mar. 1990, ISSN: 0747-7171. DOI: 10.1016/S0747-7171(08)80013-2. [Online]. Available: https://doi.org/10.1016/S0747-7171(08)80013-2.

[9] V. V. Williams, Y. Xu, Z. Xu, and R. Zhou, *New bounds for matrix multiplication: From alpha to omega*, 2023. arXiv: 2307.07970 [cs.DS]. [Online]. Available: https://arxiv.org/abs/2307.07970.

[10] J. Alman and V. V. Williams, "A refined laser method and faster matrix multiplication," *TheoretiCS*, vol. Volume 3, Sep. 2024, ISSN: 2751-4838. DOI: 10.46298/theoretics.24.21. [Online]. Available: http://dx.doi.org/10.46298/theoretics.24.21.

[11] B. Kågström, P. Ling, and C. van Loan, "Gemm-based level 3 blas: High-performance model implementations and performance evaluation benchmark," *ACM Trans. Math. Softw.*, vol. 24, no. 3, pp. 268–302, Sep. 1998, ISSN: 0098-3500. DOI: 10.1145/292395.292412. [Online]. Available: https://doi.org/10.1145/292395.292412.

[12] F. V. Zee and D. Matthews, *Blis: A framework for rapidly instantiating blas functionality*, https://github.com/flame/blis, 2013. [Online]. Available: https://github.com/flame/blis.

[13] Intel Corporation, *Intel® Math Kernel Library*, Accessed: 2025-06-04, 2021. [Online]. Available: https://software.intel.com/en-us/mkl.

[14] NVIDIA Corporation, *cuBLAS — NVIDIA developer*, Accessed: 2025-06-04, 2023. [Online]. Available: https://developer.nvidia.com/cublas.

[15] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, pp. 250–269, Sep. 1978, ISSN: 0098-3500. DOI: 10.1145/355791.355796. [Online]. Available: https://doi.org/10.1145/355791.355796.

[16] R. Yuster and U. Zwick, "Fast sparse matrix multiplication," *ACM Trans. Algorithms*, vol. 1, no. 1, pp. 2–13, Jul. 2005, ISSN: 1549-6325. DOI: 10.1145/1077464.1077466. [Online]. Available: https://doi.org/10.1145/1077464.1077466.

[17] S. Pal, J. Beaumont, D.-H. Park, *et al.*, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 724–736. DOI: 10.1109/HPCA.2018.00067.

[18] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 766–780. DOI: 10.1109/MICRO50266.2020.00068.

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

[20] The GraphBLAS community, *Graphblas*, https://graphblas.org/, 2025.

[21] 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. DOI: 10.1137/S0097539704442684. eprint: https://doi.org/10.1137/S0097539704442684. [Online]. Available: https://doi.org/10.1137/S0097539704442684.

[22] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, IEEE, 1999, pp. 285–297.

[23] The Scipy community, *Sparse matricies (scipy.sparse)*, `https://docs.scipy.org/doc/scipy/reference/sparse.html`, Accessed: 2024-11-05.

[24] N. Alon and R. Yuster, "Matrix sparsification and nested dissection over arbitrary fields," *Journal of the ACM (JACM)*, vol. 60, no. 4, pp. 1–18, 2013.

[25] R. Bhattacharjee, G. Dexter, C. Musco, A. Ray, S. Sachdeva, and D. P. Woodruff, "Universal matrix sparsifiers and fast deterministic algorithms for linear algebra," *arXiv preprint arXiv:2305.05826*, 2023.

[26] 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.

[27] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[28] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, 3rd. Upper Saddle River, NJ: Pearson Education, 2009, ISBN: 9780131988422.

[29] N. Ahmed and K. R. Rao, "Walsh-hadamard transform," in *Orthogonal Transforms for Digital Signal Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1975, pp. 99–152, ISBN: 978-3-642-45450-9. DOI: `10.1007/978-3-642-45450-9_6`. [Online]. Available: `https://doi.org/10.1007/978-3-642-45450-9_6`.

[30] S. Kopparty, "Lecture 5: K-wise independent hashing and applications," *Lecture notes for Topics in Complexity Theory and Pseudorandomness. Rutgers University*, 2013.

[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] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 20–29.

[33] 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, Automata, Languages and Programming, ISSN: 0304-3975. DOI: `https://doi.org/10.1016/S0304-3975(03)00400-6`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0304397503004006`.

[34] R. Pagh, "Compressed matrix multiplication," *ACM Trans. Comput. Theory*, vol. 5, no. 3, Aug. 2013, ISSN: 1942-3454. DOI: `10.1145/2493252.2493254`. [Online]. Available: `https://doi.org/10.1145/2493252.2493254`.

[35] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, Special issue on "Program Generation, Optimization, and Platform Adaptation".

[36] M. Frigo and S. G. Johnson, *FFTW benchmarking methodology*, `https://www.fftw.org/speed/method.html`, Accessed: 2025-05-13, 2003.

[37] Intel Corporation, *Intel® Xeon® Gold 6548N Processor*, https://www.intel.com/content/www/us/en/products/sku/237567/intel-xeon-gold-6548n-processor-60m-cache-2-80-ghz/specifications.html, Accessed: 2024-11-25, 2024.

[38] Real-Time and Embedded Systems Lab, Universität des Saarlandes, *Uops.info: Micro-operation tables*, https://uops.info/table.html, Accessed on January 28, 2025, 2025.

[39] J. J. Dongarra, J. R. Bunch, C. Moler, and G. W. Stewart, *LINPACK Users' Guide*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1979, ISBN: 978-0-898711-72-1. [Online]. Available: https://www.netlib.org/linpack/.

[40] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," *Technical Report*, vol. CS-94-297, 1995. [Online]. Available: http://www.cs.virginia.edu/stream/.

[41] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[42] The SciPy community, *Scipy documentation*, Available at https://docs.scipy.org/doc/scipy/.

[43] I. Corporation, *Intel® VTune™ Profiler*, Accessed: 2025-06-04, 2025. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[44] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011, ISSN: 0098-3500. DOI: 10.1145/2049662.2049663. [Online]. Available: https://doi.org/10.1145/2049662.2049663.

[45] cppreference.com contributors, *Std::nth_element — C++ reference*, Accessed: 2025-03-25, 2025. [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/nth_element.

[46] 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.

[47] M. Frigo and S. G. Johnson, *Fftw 3.3.10 manual*, Section 5.4: *Thread safety*, Massachusetts Institute of Technology, 2023. [Online]. Available: https://www.fftw.org/fftw3_doc/Thread-safety.html#Thread-safety.

[48] R. Zippel, "Probabilistic algorithms for sparse polynomials," in *Symbolic and Algebraic Computation*, E. W. Ng, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1979, pp. 216–226, ISBN: 978-3-540-35128-3.

[49] J. Andersson, “Efficient and numerically stable gpu implementations of the fast walsh-hadamard transform: Exploring the practical applicability of a new fast walsh-hadamard transform algorithms and methods to improve rounding errors when calculating walsh-hadamard transforms,” Master’s Thesis, Chalmers University of Technology, Department of Computer Science, Engineering, Division of Data Science, and AI, 2025.

A

Appendix 1

A.1 AI Usage

In this work, we utilized generative AI tools to assist in the development of various components, as detailed below:

- Python scripts
  - `randomTest.py` generates large random matrices and compares results between the original and our modified algorithm. This script was primarily generated by AI and later adapted manually.
  - `densityTest.py` generates sparse matrices with a specified density, including those with empty rows. It was largely developed with the help of AI.
  - `calculate_d0.py` loads a `.mtx` file and computes its r_0. This script was created using AI assistance.
  - The scripts used to generate Figures 7.1 and 3.1 were initially drafted by AI and later refined by us.
- C++ code
  - The `load_matrix_market` function, which is used to load `.mtx` files for benchmarking purposes, was partially generated with the aid of AI.

A.2 Source Code

The source code for the implementation described in this thesis is publicly available at `https://git.chalmers.se/kliemann/compmatmul`.

A.2.1 Compression

```
1 /**
2    * Computes the compressed product sketch of A^T B
3    * A^T is assumed to be a CSR sparse matrix
```

```
4    * B is assumed to be a CSC sparse matrix
5    * The result is d polybnomials of length b, that is, p is
       an array of length d*b
6    */
7   template<typename Hash>
8   matmul_fft_sketch<Hash>
       matmul_compressed_product_fft_sparse(const int32_t
       n_inner, const csr_data* A, const csr_data* B, const
       int32_t d, const int32_t b,
9     std::optional<uint64_t> seed = std::nullopt) {
10        // draw hash functions
11    if (!seed) {
12      seed = std::optional<uint64_t>(rng()());
13    }
14    matmul_fft_sketch<Hash> sketch(d, b, *seed);
15    // size of the real-transformed polynomial
16    const int32_t fft_poly_size = b/2 + 1;
17    // intermediate array for transformed polynomials
18    std::complex<double>* p_fft = array::zeros<std::complex<
        double>>(fft_poly_size*d);
19
20    int32_t num_threads;
21    #pragma omp parallel shared(num_threads)
22    {
23      #pragma omp single
24      {
25        num_threads = omp_get_num_threads();
26      }
27    }
28    // one fftw transformer per thread
29    std::vector<std::unique_ptr<fft_transformer>> ffts(
        num_threads);
30    for (int32_t i = 0; i < num_threads; ++i) {
31      ffts[i] = std::make_unique<fft_transformer>(b);
32    }
33
34    std::vector<omp_lock_t> p_locks(d);
35    for (int32_t t = 0; t < d; ++t)
36      omp_init_lock(&p_locks[t]);
37
38    #pragma omp parallel
39    {
40      std::complex<double>* fft_pa = array::allocate<std::
          complex<double>>(fft_poly_size);
41      std::complex<double>* fft_pb = array::allocate<std::
          complex<double>>(fft_poly_size);
42
43      int32_t thread_num = omp_get_thread_num();
44      fft_transformer& fft = *ffts[thread_num];
```

```
45
46        // untransformed polynomials
47        double* pa = array::allocate<double>(b);
48        double* pb = array::allocate<double>(b);
49
50
51        #pragma omp for schedule(guided)
52        for(int32_t k = 0; k < n_inner; k++) {
53          if (A->indptr[k]==A->indptr[k+1] || B->indptr[k]==B->
                indptr[k+1])
54          {
55            continue;
56          }
57          for (int32_t t = 0; t < d; t++)
58          {
59            const Hash& h1t = sketch.h1[t];
60            const Hash& s1t = sketch.s1[t];
61            array::zero(b, pa);
62            for (uint32_t i = A->indptr[k]; i < static_cast<
                 uint32_t>(A->indptr[k+1]); ++i) {
63              pa[h1t(A->indices[i])] += s1t.sign(A->indices[i])
                    * A->data[i];
64            }
65
66              const Hash& h2t = sketch.h2[t];
67              const Hash& s2t = sketch.s2[t];
68              array::zero(b, pb);
69              for (uint32_t i = B->indptr[k]; i < static_cast<
                   uint32_t>(B->indptr[k+1]); ++i) {
70                pb[h2t(B->indices[i])] += s2t.sign(B->indices[i
                      ]) * B->data[i];
71              }
72
73            fft.fwd(fft_pa, pa);
74            fft.fwd(fft_pb, pb);
75
76            // acquire lock for p_fft[t]
77            omp_set_lock(&p_locks[t]);
78            for (uint32_t a = 0; a < static_cast<uint32_t>(
                 fft_poly_size); ++a)
79            {
80              p_fft[t * fft_poly_size + a] += fft_pa[a]*fft_pb[
                   a];
81            }
82
83            omp_unset_lock(&p_locks[t]);
84          }
85        }
86        array::deallocate(pa);
```

```
87        array::deallocate(pb);
88        array::deallocate(fft_pa);
89        array::deallocate(fft_pb);
90      }
91
92      double* p = sketch.p.get();
93      #pragma omp parallel
94      {
95        fft_transformer& fft = *ffts[omp_get_thread_num()];
96        #pragma omp for
97        for (int32_t t = 0; t < d; ++t) {
98          fft.inv(p + t*b, p_fft + t*fft_poly_size);
99        }
100     }
101
102     for (int32_t t = 0; t < d; ++t)
103       omp_destroy_lock(&p_locks[t]);
104     array::deallocate(p_fft);
105     return sketch;
106   }
```

Listing A.1: Source code for compression phase

A.2.2 Decompression

```
1  /**
2   * Computes unbiased estimate of C=A^T B.
3   * That is, AT is expected to have shape n_inner * n_a,
4   * B the shape n_inner * n_b, and C the shape n_a * n_b
5   * All assumed to be row-major
6   */
7   template <typename Hash, SparseFormat Format>
8   constexpr auto matmul_fft_sparse_to_sparse(const int32_t
        n_a, const int32_t n_inner, const int32_t n_b, const
        csr_data* AT,
9     const csr_data* B, const int32_t d, const int32_t b,
10    std::optional<uint64_t> seed = std::nullopt) {
11
12    assert(reinterpret_cast<uintptr_t>(AT) % 64 == 0);
13    assert(reinterpret_cast<uintptr_t>(B) % 64 == 0);
14    auto sketch = matmul_compressed_product_fft_sparse<Hash>(
         n_inner,AT, B, d, b, seed);
15
16
17    auto indptr = array::allocate<int32_t>(n_a+1);
18    memset(indptr, 0, sizeof(int32_t) * (n_a+1));
19 #pragma omp parallel
20    {
21        double* scratch = array::allocate<double>(d);
22 #pragma omp for schedule(guided)
```

```
23        for (int32_t i = 0; i < n_a; ++i)
24        {
25          const double* p = sketch.p.get();
26          int32_t d = sketch.d;
27          int32_t b = sketch.b;
28          for (int32_t j = 0; j < n_b; ++j)
29          {
30            const double* p = sketch.p.get();
31            int32_t d = sketch.d;
32            int32_t b = sketch.b;
33            for (int32_t t = 0; t < d; ++t) {
34              uint32_t h = (sketch.h1[t](i) + sketch.h2[t](j))
                   & sketch.mod_mask;
35              int32_t s = sketch.s1[t].sign(i) * sketch.s2[t].
                   sign(j);
36              scratch[t] = s * p[t*b + h];
37            }
38            auto result = array::median(d, scratch);
39            if (result != 0.0)
40              indptr[i+1]++;
41          }
42        }
43
44      }
45      // Sum up indptr from the amount of non zeros of each row
46      for (int32_t i = 0; i < n_a; i++)
47      {
48        indptr[i+1] += indptr[i];
49      }
50
51
52      if constexpr (Format == SparseFormat::COO) {  //
           ---------- COO -------------
53        compmatmul::coo_data* C = new compmatmul::coo_data;
54        C->nnz = indptr[n_a];
55        C->data = array::allocate<double>(C->nnz);
56        C->x = array::allocate<int32_t>(C->nnz);
57        C->y = array::allocate<int32_t>(C->nnz);
58        C->rows = n_a;
59        C->cols = n_b;
60  #pragma omp parallel
61        {
62          double* scratch = array::allocate<double>(d);
63
64  #pragma omp for schedule(guided)
65          for (int32_t i = 0; i < n_a; ++i) {
66            uint32_t col_counter = indptr[i];
67            for (int32_t j = 0; j < n_b; ++j) {
68              const double* p = sketch.p.get();
```

```
69                int32_t d = sketch.d;
70                int32_t b = sketch.b;
71                for (int32_t t = 0; t < d; ++t) {
72                  uint32_t h = (sketch.h1[t](i) + sketch.h2[t](j)
                       ) & sketch.mod_mask;
73                  int32_t s = sketch.s1[t].sign(i) * sketch.s2[t
                       ].sign(j);
74                  scratch[t] = s * p[t*b + h];
75                }
76                auto result = array::median(d, scratch);
77                if (result != 0.0)
78                {
79                  C->x[col_counter] = i;
80                  C->data[col_counter] = result;
81                  C->y[col_counter] = j;
82                  col_counter++;
83                }
84              }
85            }
86            array::deallocate(scratch);
87
88          }
89          return C;
90        } else if constexpr (Format == SparseFormat::CSR) {     //
            ------------ CSR ------------
91          compmatmul::csr_data* C = new compmatmul::csr_data;
92          C->nnz = indptr[n_a];
93          C->data = array::allocate<double>(C->nnz);
94          C->indices = array::allocate<int32_t>(C->nnz);
95          C->indptr = indptr;
96          C->rows = n_a;
97          C->cols = n_b;
98  #pragma omp parallel
99          {
100           double* scratch = array::allocate<double>(d);
101
102 #pragma omp for schedule(guided)
103           for (int32_t i = 0; i < n_a; ++i) {
104             uint32_t col_counter = C->indptr[i];
105             for (int32_t j = 0; j < n_b; ++j) {
106               const double* p = sketch.p.get();
107               int32_t d = sketch.d;
108               int32_t b = sketch.b;
109               for (int32_t t = 0; t < d; ++t) {
110                 uint32_t h = (sketch.h1[t](i) + sketch.h2[t](j)
                       ) & sketch.mod_mask;
111                 int32_t s = sketch.s1[t].sign(i) * sketch.s2[t
                       ].sign(j);
112                 scratch[t] = s * p[t*b + h];
```

```
113             }
114             auto result = array::median(d, scratch);
115             if (result != 0.0)
116             {
117               C->indices[col_counter] = j;
118               C->data[col_counter] = result;
119               col_counter++;
120             }
121           }
122         }
123         array::deallocate(scratch);
124 
125       }
126       return C;
127     } else if constexpr (Format == SparseFormat::CSC) {
128         // Left for future work
129     }
130 
131   }
```

Listing A.2: Source code for the decompression phase