CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

# Efficient GPU acceleration of Compressed Matrix Multiplication

## Implementation and analysis of Pagh's algorithm for compressed matrix multiplication on massively parallel hardware

Master's thesis in High-performance computer systems

Egil Guting

Samuel Runmark Thunell

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

# Efficient GPU acceleration of Compressed Matrix Multiplication

Implementation and analysis of Pagh's algorithm for compressed matrix multiplication on massively parallel hardware

Egil Guting

Samuel Runmark Thunell

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Efficient GPU acceleration of Compressed Matrix Multiplication
Implementation and analysis of Pagh's algorithm for compressed matrix multiplication on massively parallel hardware
Egil Guting
Samuel Runmark Thunell



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

Master's Thesis 2026
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 2026

Efficient GPU acceleration of Compressed Matrix Multiplication
Implementation and analysis of Pagh's algorithm for compressed matrix multiplication on massively parallel hardware
Egil Guting
Samuel Runmark Thunell
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

# Abstract

Compressed Matrix Multiplication (CMM), introduced by Rasmus Pagh in 2013, is an approximation algorithm for computing matrix products where the result os dominated by a small number of large entries. The algorithm decomposed the matrix product into a sum of outer products, where each outer product is independently compressed into polynomial coefficients. Since these are independent of one another, the algorithm is inherently highly parallelizable.

The research path being approached is therefore focused on how Compressed Matrix Multiplication can utilize the massively parallel hardware of GPUs to surpass the performance of standard General Matrix Multiplication, specifically NVIDIA's cuBLAS.

To evaluate this, we developed a implementation of Pagh's Compressed Matrix Multiplication on the GPU using the CUDA platform. This implementation was benchmarked against the cuBLASXt library using dense matrices of sizes $2^{10}$ to $2^{17}$ with result matrices with few elements impacting the Frobenius norm. The benchmarking results demonstrate that it was able to surpass cuBLAS with a maximum recorded speedup of 60.38 on generated matrices tailored for CMM. This suggests that Pagh's algorithm can be useful on massively parallel hardware for certain matrices and that further work should be pursued to realize more concrete implementations.

Keywords: CUDA, algorithms, matrix, multiplication, GPU, high-performance, thesis, computer science

# Acknowledgements

We would like to express our gratitude to our supervisor and examiner Matti Karppa for the opportunity to pursue this thesis project, and for much needed support and feedback. We would also like to thank the CSE department at Chalmers and GU, for providing access to the Minerva compute cluster for experimentation.

Egil Guting, Samuel Runmark Thunell, Gothenburg, 2026-06-15

# Contents

**Glossary** **xi**

**List of Figures** **xiii**

**1 Introduction** **1**
1.1 Aim . . . 2
1.2 Ethical considerations . . . 2
1.3 Summary of contributions . . . 3

**2 Background** **5**
2.1 Fast Matrix Multiplication . . . 5
2.2 Compressed Matrix Multiplication . . . 5
2.3 GPU architecture and the CUDA platform . . . 6
2.4 Matrix multiplication on GPUs . . . 8
2.5 Block Matrix Multiplication . . . 9
2.6 Algorithm engineering . . . 9

**3 Methodology** **11**
3.1 Prototype . . . 11
3.2 GPU implementations . . . 11
3.2.1 Naive . . . 12
3.2.2 Naive with cuFFTDx . . . 12
3.2.3 NVIDIA streams . . . 12
3.2.4 NVIDIA streams with cuFFTDx . . . 13
3.2.5 Tiled . . . 13
3.3 Experimental setup . . . 13
3.4 Testing and Benchmarking . . . 14
3.4.1 Synthetic matrix datasets and caching . . . 14
3.4.2 Correctness and stability testing . . . 15
3.4.3 Implementation profiling and kernel-level analysis . . . 16
3.4.4 Benchmarking suite . . . 16

**4 Results** **17**
4.1 Implementation Variants . . . 17
4.2 Accuracy Validation and Performance Constraints . . . 18
4.3 Kernel-level performance evaluation . . . 18

4.4 Roofline Model . . . . . 20
4.5 Scaling analysis . . . . . 20
4.6 Speedup comparisons . . . . . 26

**5 Discussion** **27**
5.1 Known performance limitations . . . . . 27
5.1.1 Branch divergence issue . . . . . 27
5.1.2 Subpar performance of *cucmm-cufftdx-stream* . . . . . 27
5.2 Unknown characteristics of cuBLASXt . . . . . 28
5.3 Future Work . . . . . 28
5.3.1 Multi-GPU implementation . . . . . 28
5.3.2 Tiled implementation utilizing cuFFTDx library . . . . . 28
5.3.3 Performance comparison for other matrix types . . . . . 29

**6 Conclusion** **31**

**References** **33**

**A Execution times of implementations** **I**

# Glossary

| | |
|---|---|
| **CMM** | Compressed Matrix Multiplication |
| **CUDA** | Compute Unified Device Architecture |
| **Sparse matrix** | A matrix with few non-zero elements, often sparsely distributed |
| **Sparse matrix representations** | Storage formats for sparse matrices, an example of this is Compressed Sparse Row (CSR) |
| **HPC** | High-Performance Computing |
| **GEMM** | General Matrix Multiplication |
| **BLAS** | Basic Linear Algebra Subprograms |
| **SIMD** | Single Instruction, Multiple Data |
| **SIMT** | Single Instruction, Multiple Threads |

# List of Figures


4.1 Kernel breakdown comparison across implementations . . . . . 19
4.2 Roofline model of *cucmm-tiled* . . . . . 21
4.3 Roofline model of *cucmm-cufftdx* . . . . . 22
4.4 Roofline model of *cucmm-naive* . . . . . 23
4.5 Roofline model of *cucmm-cufftdx-stream* . . . . . 24
4.6 Scaling analysis . . . . . 25
4.7 Scaling analysis for all implementations . . . . . 26
4.8 Speedup comparison of select implementations . . . . . 26

# 1

# Introduction

Matrix multiplication (MM) is a fundamental computational operation in scientific computing and the backbone of a wide range of applications, including numerical simulation, machine learning, and large-scale data analysis [6]. Given a matrix $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$, their product $C = AB$ is defined as

$$C_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

The thesis of this work will demonstrate the use of Pagh's algorithm [25] for compressed matrix multiplication (CMM) on the GPU using CUDA, a programming model developed by NVIDIA that allows development of applications utilizing NVIDIA GPUs [22]. Pagh's algorithm compresses the outer product of matrices into polynomial coefficients to favor resulting matrices which are comprised of mostly small entries.

The product $AB$ can be expressed as a sum of outer products $\sum_{k=1}^{n} A[:, k] \cdot B[k, :]$, and the algorithm compresses each outer product into polynomial coefficients using randomized hash functions. Fast polynomial multiplication via FFT is then used to compute the compressed product efficiently. This also encompasses common MM optimization practices, special considerations for sparse matrices, and appropriate design patterns for GPU architecture.

The project aims to improve the performance using established HPC techniques for GPU architectures. The theoretical findings in the paper will provide insights into the algorithm's core components, which will be essential when redesigning the algorithm for GPU-based systems.

There exists algorithms for fast general MM (GEMM) such as Strassen's algorithm [29]. Although its time complexity of $\mathcal{O}(n^{\log 7})$ is lower than the naive algorithm it hides a constant large enough that the algorithm is only really effective at large matrix sizes. Instead, most state-of-the-art implementations focuses on effectively performing parallelized operations using different strategies, such as row-splitting or merge-based implementations [32].

The Basic Linear Algebra Subprograms (BLAS) is a set of routines that are useful for implementing vector and matrix operations [30], and cuBLAS and rocBLAS are libraries of BLAS implemented for Nvidia and AMD GPUs respectively [21, 4]. These allows running MM on specific hardware such as Tensor cores and since

GPUs allow much more parallelism also often perform much better compared to CPU implementations of GEMM.

Redesigning an algorithm for use in GPU kernels is non-trivial, and aligns with observations from the HPC community. An empirical study of such applications shows that improper recognition of common performance problems like inefficient parallelization, poor memory usage, and a mismatch in algorithmic design towards hardware are the biggest challenges for these kinds of projects [17]. No existing work is known to implement or evaluate CMM on GPU architectures. The project therefore involves determining whether this method can be efficiently realized for GPUs.

## 1.1 Aim

The overall goal of the thesis is to investigate the viability of utilizing compressed multiplication on GPUs by implementing and evaluating Pagh's randomized algorithm [25] using the CUDA platform [22] in order to surpass the current state-of-the-art implementations for GEMM on the GPU. This also includes the necessity to evaluate the scalability and runtime when varying complexity-impacting parameters such as the matrix size, sketch size, and the number of independent sketches. While traditional dense and sparse matrix multiplication have been optimized rigorously on GPUs, compressed matrix multiplication has yet to be analyzed on GPU architecture. This project therefore aims to bridge this research gap by exploring how Pagh's algorithm holds up on massively parallel hardware.

There are also multiple challenges associated with our proposed goals. Pagh's algorithm was not originally defined with GPU constraints in mind. Adapting it towards SIMD-style (Single Instruction, Multiple Data) parallelism and a differently structured memory hierarchy introduces both algorithmic and implementation challenges. The algorithm involves a step requiring the use of hash functions which introduces irregular data access patterns. This challenges typical GPU optimization techniques like coalescing, which will require domain-specific techniques. Another challenge of the project is the notion of creating a proper benchmark and how to evaluate correctness and accuracy due to the inherent probabilistic nature of the algorithm.

## 1.2 Ethical considerations

This thesis does not involve any human participants, personal data, or sensitive information. Therefore, the proposed ethical considerations focus primarily on academic integrity and appropriate research practices.

All code and produced material as part of this thesis will be developed entirely by the authors. It is important that is based on publicly available information and be assumed to not be derivative of other people's work. This will ensure the integrity and novelty of the thesis' results.

It can be difficult to pinpoint where one might draw the line on what is considered

intellectual theft. Since one of the core aspects of this master's thesis is writing programming code in the purpose of furthering the understanding in high-performance computing, the members of this project have decided to avoid using generative AI since it will produce information or code that is derivative of other people's work.

## 1.3 Summary of contributions

This thesis demonstrates that Pagh's algorithm for Compressed Matrix Multiplication can be efficiently implemented for GPU hardware using CUDA. The final proposed implementation *cucmm-tiled* achieves a speedup of 60.38× over cuBLASXt's GEMM baseline for large *logunit* matrices. The results confirms viability of CMM on massively parallel architectures, and there is also the observation that memory transfer overhead is the primary bottleneck for further optimization.

# 2

# Background

Since this thesis consists of adapting the CMM algorithm to GPU, this chapter will give a brief summary on topics regarding matrix multiplication including its history, applications and various implementations and algorithms. Some time is also spent creating a baseline of information regarding GPU hardware that will be helpful in later chapters.

## 2.1 Fast Matrix Multiplication

Despite the operation's apparent conceptual simplicity, a general case matrix multiplication (GEMM) is expensive, requiring $\Theta(mnp)$ operations to compute [8, 14] for a matrix multiplication between $m \times n$ and $n \times p$ matrices. The combination of being an important matrix operation and featuring expensive computation has made it a popular target for optimization in the community for decades [6, 14, 15, 2].

In 1969, Strassen demonstrated that the cubic bound of the complexity for square matrices is not optimal [29]. Strassen's algorithm is an important cornerstone in the progression of reducing the complexity of matrix multiplication [18, 13]. Since then, a large number of proposed improvements has progressively lowered the theoretical bound on complexity $\mathcal{O}(n^{\omega})$, where $\omega$ is the bound for the smallest real number for which two $n \times n$ matrices can be multiplied [14, 15, 7]. Most notably is the efforts of Coppersmith and Winograd who utilized the laser method discovered by Strassen [28, 10], which is also used by most research within fast MM algorithms [2].

Despite having their theoretical importance proven, these algorithms are rarely used in practice, and are outperformed by traditional methods in most widely-used state-of-the-art libraries such as multiple implementations of BLAS [8, 30, 13, 21, 4].

## 2.2 Compressed Matrix Multiplication

Pagh's algorithm replaces the explicit construction of the product matrix $AB$ with a probabilistic data structure derived from Count sketching [9], sketching allows a significantly smaller and probabilistic representation but introduces error since multiple inputs can be seen as the same output because of hash collisions.

The product of two matrices A and B can be seen as the sum of outer products. The algorithm operates by sketching one outer product at a time, each sketch is an

independent operation. For each $k$ the column $A[:, k]$ and row $B[k, :]$ are compressed into 'buckets', which are coefficients of two separate polynomials $p_A$ and $p_B$ of size $b$. Each element is assigned a sign and index in the polynomial, this is controlled by the random, 2-wise independent sign and bucket hashes. The property of 2-wise independence for the hash families guarantees that two independently chosen sets of hashes are considered independent [20].

The polynomials are then multiplied which means every coefficient is multiplied with each coefficient in the other polynomial. Since every element from $A[:, k]$ and $B[k, :]$ has been accumulated this is equivalent to multiplying them together. Normally multiplying polynomials in coefficient form is $\Theta(n^2)$, where n is the size of the polynomials, but a FFT corresponds to a point-value representation of a polynomial which allows multiplying them element-wise in $\mathcal{O}(n)$ time, although the FFT is still $\mathcal{O}(n \log n)$ time [11]. An inverse FFT then returns a polynomial $p_r$ with size $b$ that contains the polynomial multiplication. Repeating this over $k$ generates a complete sketch of the entire product matrix.

For an entry $(i, j)$, the algorithm finds whichever buckets the indices in the entry relates to and applies its associated signs. Then it finds which coefficient in $p_r$ has $A_i B_j$. Since a coefficient in $p_r$ not only contains $A_i B_j$ for a given entry but also a sum from several $A_{i^*} B_{j^*}$ from other entries $(i^*, j^*)$ it is important to recover the correct value. Using $d$ independent repetitions, each using varying permutations of its sign and hash functions, gives multiple $d$ unbiased estimates for every element. Continually, a median of the reconstructed values over all repetitions $d$ reduces the variance of each entry $(i, j)$ in $AB$ when $d > 1$. Algorithm 1 shows all the same steps in pseudocode.

The algorithm achieves a time complexity of $\tilde{O}(d(n^2 + nb))$ using space $\mathcal{O}(db + d)$ when replacing the explicit MM with FFT-based sketch reconstruction where $d$ is the amount of sketches, $b$ is the amount of buckets and $n$ is the amount of columns in $A$ and rows in $B$.

Time complexity reduces noticeably in relation to the sparsity of the calculated output matrix. This is due to the inherent property of the algorithm being output-sensitive. The amount of significant entries in $AB$ also strongly affects variance of the sketched output. For one sketch each element has a variance bounded by $\frac{||AB||_F^2}{b}$ where $||AB||_F^2$ is the squared Frobenius norm. The randomized hashing of terms also provides probabilistic accuracy guarantees, heavy entries of $AB$ can be recovered with high probability. The estimator's variance will also decrease as the number of independent sketches $d$ increases, this motivates the algorithm's suitability for input matrices whose output contains sparse or unevenly distributed large entries.

## 2.3 GPU architecture and the CUDA platform

The structure of a GPU has similarities to a CPU, both have computing cores, DRAM, L1 and L2 cache (although most GPUs lack the L3 cache that CPUs have)

**Algorithm 1** Pagh's compression and decompression algorithms

**function** CompressedProduct($A, B, b, d$)
  **for** $t \in [d]$ **do**
    Draw $s_1[t], s_2[t] : [n] \to \{-1, +1\}$ from a 2-wise independent hash family
    Draw $h_1[t], h_2[t] : [n] \to [b]$ from a 2-wise independent hash family
    $p[t] \leftarrow \mathbf{0}_b$ ▷ Real-valued
    **for** $k \in [n]$ **do**
      $(pa, pb) \leftarrow (\mathbf{0_n}, \mathbf{0_n})$
      **for** $i \in [m]$ **do**
        $pa[h_1(i)] \leftarrow pa[h_1(i)] + s_1[t](i)A_{ik}$
      **end for**
      **for** $j \in [p]$ **do**
        $pb[h_2(j)] \leftarrow pb[h_2(j)] + s_2[t](j)B_{kj}$
      **end for**
      $(pa, pb) \leftarrow (FFT(pa), FFT(pb))$
      **for** $z \in [b]$ **do**
        $p[t][z] \leftarrow p[t][z] + pa[z]pb[z]$
      **end for**
    **end for**
  **end for**
  **for** $t \in [d]$ **do**
    $p[t] \leftarrow FFT^{-1}(p[t])$
  **end for**
  **return**$(p, s_1, s_2, h_1, h_2)$
**end**

**function** Decompress($i, j$)
  **for** $t \in [d]$ **do**
    $X_t \leftarrow s_1[t]s_2[t](j)p[t][(h_1(i) + h_2(j)) \bmod b]$
  **end for**
  **return** MEDIAN$(X_1, ..., X_d)$
**end**

[22, 3]. The cores on NVIDIA GPUs are called streaming multiprocessors (SMs). The real difference lies in the distribution of resources, compared to the memory units the GPU has vastly more units for computing. This allows GPUs to perform paralellized tasks with great efficiency since computing units can run concurrently.

CUDA parallelizes tasks into thread blocks with a maximum of 1024 threads per block [22]. This is done by dispatching warps with 32 threads that performs each instruction in the task concurrently, also described as being SIMD processing [22]. Each dispatch is added to the SIMT stack where it serializes the separate tasks performed by the threads. [1]. Therefore, it is often preferable to choose block sizes with a multiple of 32 which allows all warp dispatching to be evenly distributed. Usually threads allocate temporary data in local memory but by using the shared memory in the L1 cache it is possible to store data that all threads in that block can access. Since local memory is actually just a private container in DRAM the L1 cache also provide better bandwidth and lower latency that can be utilized for parts in a task that require high performance. CUDA uses versioning system called *compute capability* to describe the available functionalities a NVIDIA GPU might have [22]. As such, some features are exclusive to specific compute capabilities, like the before mentioned shared memory. Furthermore, it also includes things such as the amount shared memory a GPU is able to allocate for each thread block and how much resources each SM has.

## 2.4 Matrix multiplication on GPUs

Modern GPUs are massively parallel throughput processors whose thousands of lightweight threads are optimal for regular, data-parallel workloads such as dense matrix multiplication. GPU vendor libraries such as NVIDIA's cuBLAS [21] and AMD's rocBLAS [4] implement dense GEMM using state-of-the-art methods for exploiting the hardware's throughput and memory bandwidth [6].

Recent generations of NVIDIA devices also provide specialized matrix-multiply-accumulate units for accelerating mixed-precision computing in the form of *tensor cores*. Exploiting these tensor cores for GEMM kernels substantially improves performance and energy efficiency compared to conventional CUDA cores [16].

Significant performance increases can be observed when designing GEMM to be implemented on GPU hardware. This is mostly due to the high compatibility between the operations for dense MM and that of the hardware's underlying architecture [33]. For sparse matrices however, performance is less dominated by arithmetic throughput and more by irregular memory access patterns, poor cache utilization, and the unpredictability of the output size [27]. These issues are well-documented in state-of-the-art implementations on the GPU such as cuSPARSE [23]. Most commonly, these implementation restructure the algorithm for compatibility with efficient storage formats and multi-phase computations.

## 2.5 Block Matrix Multiplication

To allow for MM for matrix sizes with memory requirements beyond the VRAM on the GPU means that algorithms have to find ways to segment the matrices into blocks that could fit on device memory and still ensure that the algorithms are functionally correct. To compute GEMM blocks it is required to separate each block result into multiple operations. If the matrix is divided into blocks of $p \times q$ an element $C_{ij}$ in result matrix $C$ will need to compute

$$C_{ij} = \sum_{k=1}^{q} A_{ik} B_{kj}.$$

Given A and B

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Each element in $C$ can be seen as a sum of block products

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

In this way it is possible to see that for bigger matrices than what fits on the VRAM the amount of block operations required to compute the result matrix will increase linearly. Comparing to CMM the process is much more streamlined. The accumulation of outer products into polynomials is done independently for each element in each matrix as can be seen in algorithm 1. This means that scaling the size of matrices to above the size of what fits in the VRAM has no impact on how polynomials need to be accumulated or even multiplied.

## 2.6 Algorithm engineering

Redesigning an algorithm for use in GPU kernels is non-trivial, and requires careful consideration due to the architectural differences. An empirical study of such applications have shown that improper recognition of common performance problems like inefficient or missing parallelization, poor memory usage, but especially poor algorithmic design towards hardware are some of the biggest challenges for these kinds of projects [17]. For example, using the global memory on the GPU instead of local or shared memory brings higher latency. Continually, GPU's contain streaming multiprocessors which can make it difficult to effectively map workload onto the GPU while also ensuring efficient parallelization. Finally, since the implementation of the algorithm matters greatly to the resulting performance it is important that the algorithm is understood well to simplify the process of building it on the GPU. A reoccurring problem in GPU applications is branch divergence, since threads in warps execute independently it is possible for threads to diverge to different parts when executing [1]. Most GPU architectures can solve this by serializing each part that the threads targeted and adding them to the SIMT stack. This results in resolving

each part one at a time while only allowing the threads involved in each divergence to execute. Since some threads are idle during each execution this leads to inefficient computation. Either removing divergence or ensuring the whole warp can perform on a target ensures that no threads are idle.

# 3

# Methodology

This chapter describes the structure of the work, the implementation workflow, and any experimental methodology used for evaluating the GPU-based implementation.

## 3.1 Prototype

The first step of the work was to implement the algorithm in a high-level language. A prototype was developed in python using NumPy for matrix representation and calculations, and SciPy to generate random sparse test matrices. This provided a reference implementation to validate correctness and stability. The naive prototype served two purposes: it clarifies the algorithmic flow and any existing dependencies, and it provides a baseline for validating the GPU implementation at each step. The prototype implementation is intentionally minimal, it contains only simple versions of the hash functions and the compression and decompression of the polynomials. Mainly so that it closely resembles the originally proposed version shown in algorithm 1 from Rasmus Pagh's paper [25].

Due to the necessary hashing and randomized steps of the algorithm, one can not assume that the same results are achieved across different implementations. Not achieving identical hashing across different targets of comparison will not guarantee the implementations correctness. This was apparent when developing the uniform randomization used for the hashing, as different implementations of the Marsenne Twister gave different results. The NumPy implementation of MT19937 was at first used for the prototype, this was however changed to instead reproduce the same integer state initialization as the C++ standard library uses for its implementation.

## 3.2 GPU implementations

The GPU implementations were developed iteratively, starting from a correctness-oriented translation of the original algorithm into CUDA kernels and gradually incorporating GPU-specific optimizations. All implementations share a common high-level pipeline structure of adapting the algorithm to kernels, consisting of the following steps:

1. accumulation of the input matrices into polynomial coefficients
2. application of a forward Fourier transform

3. point-wise multiplication and accumulation in the transformed domain
4. application of an inverse Fourier transform
5. decompression of the sketch polynomial to get the output matrix

### 3.2.1 Naive

The initial, naive GPU implementation prioritized functional correctness. Non-conflicting part of the algorithm was specifically targeted and divided into separate kernels and major interleaving optimizations was not considered. Hash functions were generated on the host and copied to the GPU to simplify pseudo-random generation. The compression step was divided into two CUDA kernels to simplify the Fourier transforms as they had to be initiated on the host CPU. Therefore, one kernel accumulates additions through scatter-add for the matrix entries into the complex-valued polynomials. The Fourier transform is then executed with a defined cuFFT 'plan' which is made up of batched 1D FFTs without copying anything between the host and device. After the second compression kernel, which performs the point-wise multiply-and-accumulate operation, an inverse cuFFT operation produces the time-domain sketch polynomial. The decompression then constructs each entry by using shared memory to stage per-block working data. This is followed by calculating the median of all sketched values to get the final entry result.

### 3.2.2 Naive with cuFFTDx

To reduce apparent CUDA library overhead and improve locality, cuFFT was replaced with cuFFTDx, which provides compile-time FFT primitives that can be executed within CUDA kernels utilizing the same kernel parameters as the existing kernels. This enabled tightly coupled kernels to perform compression, forward FFT, point-wise multiply-and-accumulate, and inverse FFT while keeping intermediate data in the device's registers and shared memory.

Because the cuFFTDx library configurations have strict requirements regarding device resources, this implementation dispatches specialized kernels based on the number of buckets (the FFT size in ours case) and uses extended, opt-in dynamic shared memory when required.

### 3.2.3 NVIDIA streams

The approach was to use CUDA's streams to interleave the copying from host memory to device memory with the cuFFT and the kernels performing the compression step. This required the use of asynchronous versions of methods used in the Naive implementation (e.g. cudaMemcpyAsync and cudaMallocAsync) as well as different streams for copying memory and the compression step.

### 3.2.4 NVIDIA streams with cuFFTDx

After having integrated the FFT operation tightly into the existing CUDA kernels, streams became relevant as a possibility for additional concurrency. The intended design was to schedule groups of sketches as separate non-blocking streams. Since sketching is independent until the final decompression step, this enables concurrent kernel execution across sketches and overlaps any transfer with computation.

Since every sketch has unique hashes each stream could generate its own set of hash families to avoid race conditions. By using the cuRAND library the generation of the hashes could be performed on the GPU instead. Similarly to cuFFTdx the hashes could be generated directly on shared memory, allowing faster memory access and hash generation. Since the FFT is generated after the hashes are used, the shared memory can be reused leading to more efficient handling of memory.

In addition, the layout of the input matrices was changed to improve memory coalescing and reduce global memory traffic for the compression kernels.

### 3.2.5 Tiled

A major iteration of the implementation was to introduce tiling/batching to provide possible scaling beyond the memory capacity of the GPU required for the full intermediate buffers used in the CUDA kernels. The matrices are processed as sub-matrices (tiles) along the inner dimensions so that only a single tile of each input matrix and the corresponding polynomial chunk need to reside on the device at any time. For each tile in the inner dimension, the compressed polynomials are generated and accumulated in the same way as the naive implementation.

A major issue with this approach was its non-deterministic results introduced by the use of atomic addition for accumulation in parallel. Accumulation of floating-point values in a non-deterministic order can lead to different rounding errors across runs. This required several benchmarking runs to be made in order to have valid results.

## 3.3 Experimental setup

Early development and correctness validation of the naively implemented GPU kernels were performed on local workstations equipped with NVIDIA 3070 Ti GPUs with 8GB VRAM and compute capability of 8.6. This setup provided a convenient development and experimentation environment for the first cycle of the project.

All experiments reported in this thesis were executed on nodes of the Minerva compute cluster, hosted by the Chalmers Computer Science and Engineering department. Minerva provides a more stable, research-grade experimentation environment compared to the local workstations. Experimentation was executed through Minerva's centralized login node with job scheduling and dispatching through Slurm.

The reported runs used nodes (named *io*, *europa*, *ganymede*, *callisto*) on the Minerva cluster equipped with 8×NVIDIA L4 GPUs, packaged with 24GB VRAM and a compute capability of 8.9. These nodes were also equipped with two Intel Xeon

GOLD 6548N CPUs (2×64 cores), 512GiB RAM. All of Minerva's nodes run Ubuntu Linux 24.04 LTS.

Reporting the exact hardware configuration is essential for interpreting the results as GPU resources such as available VRAM and per-block shared memory heavily influences the kernel configurations of the implementations. Most notable are the cuFFTDx-based implementations which uses hardware limits to determine optimal FFT kernel configurations.

## 3.4 Testing and Benchmarking

To ensure that the GPU implementations displays performative and accurate results, the implementations were benchmarked and the evaluation had to address the following two concerns: the accuracy of the randomized algorithm with respect to theoretical guarantees, along with performance and scaling behavior across matrix sizes and chosen parameters.

The project repository [26] was therefore provided with separate testing binaries, micro-benchmarks, scaling benchmarks, and profiling scripts utilizing NVIDIA Nsight Compute [24].

### 3.4.1 Synthetic matrix datasets and caching

Any meaningful testing and benchmarking of the implementations requires input with a known structure and controllable complexity according to the algorithm's theoretical guarantees. The project used synthetic matrix dataset construction methods implemented in a separate, structured component. At first, it was only considered to use uniformly random matrices, but this was later extended to include two more generators used extensively. Therefore, the project includes a *logunit*-style, and *diagonal/permutation*-based instances. These generators were based on the ones used in previous work by Andersson and Karppa [5].

The logunit matrices were generated such that 2 dense matrices results in a matrix with only 1 element in each row and column that is non-zero, for a $n \times n$ matrix $\log n$ of these are 1 and $n - \log n$ are $10^{-4}$. The diagonal matrices are similar but all non-zero elements in the resulting matrices were within $0.5 \leq |x| \leq 1$.

To ensure that all implementations are evaluated on identical input and respective output, and to avoid repeatedly regenerating massively large matrices, the project includes a persistent matrix dataset cache. This cache was implemented utilizing the HDF5 file format, which provided efficient storage of large numerical datasets [31]. Input matrices $A$ and $B$, and the exact product $C$ are stored in a single file whose naming encodes the generation parameters, which includes matrix dimensions, randomization seed, and generator type. The cache is used by all interactions with the implementations, including testing and benchmarking, which reduced the overhead variance between runs and made it possible to evaluate implementations identically over time.

### 3.4.2 Correctness and stability testing

Correctness checks were performed by comparing the output of the GPU computations against an exact reference compute on the CPU. This reference was later replaced by an implementation through cuBLAS on the GPU as the tested matrix sizes grew larger. For a given pseudo-randomization seed and matrix generation routine, input matrices $A$ and $B$ were generated deterministically and the exact product $C = AB$ was computed. This makes it possible to validate changes in the proposed CUDA kernels without conflating kernel correctness with changes in the experimental input.

Unit tests were implemented for the various components of the algorithm using the Catch2 library, these tests were compiled as dedicated test executables as to not interfere with the main implementation executables. Testing was mainly used for the implementation of the algorithm, to ensure its correctness and stability across changes. It was however later used to accommodate implementation utilities, most notably the matrix generation routines, which require strong correctness guarantees to be useful for accurate results.

Multiple different test frameworks were considered for unit testing, this being Google Test [12] and CTest, which is part of the CMake build system [19]. The main reason for not choosing these frameworks was their unnecessary complexity for the intended use case. The Catch2 framework was ultimately chosen for its simple, yet powerful design and ease of integration with the existing Cmake-based build system. Another big reason was the possibility of utilizing Catch2 for both unit testing and benchmarking, which simplified the build system and kept dependency management more straightforward.

Two primary types of unit tests are defined for validating the compressed matrix multiplication routine. The first, *SparseExactness*, validates element-wise correctness by comparing the output of the CMM implementation's output against the exact product for 256×256 *logunit* matrices. The parameters for this comparison were chosen to be large enough to ensure the algorithm's correctness according to Theorem 3.4 from Pagh's paper [25], which states that the matrix product $AB$ can be computed exactly if $AB$ has at most $b/8$ non-zero entries, and $d \geq 6\log_2(n)$. The second, *FrobeniusNormBoundPassRate*, verifies the probabilistic error guarantee of the algorithm in conjunction with Theorem 3.1 from Pagh's paper [25]. It was executed with 1000 randomly seeds, using a single sketch, 256 hash families, and 16×16 generated matrices, the test checks whether the Frobenius norm of the error stays within the theoretical bound of $2\|C_F\sqrt{mk}\,/\sqrt{b}$ at least 95% of the time. This provided experimental confirmation that the implemented algorithm respects its own theoretical guarantees.

An issue with testing which became apparent during development was that exact bit-level reproducibility can not be assumed across different implementations, due to the variation in operation order and deterministic hashing. This was especially apparent when implementing the tiled version of the algorithm to accommodate larger matrix sizes on GPUs with limited VRAM. The introduction of atomic additions during parallel accumulation and differing thread scheduling produces small, non-deterministic variations even when the implementation is correct and identical

seeds are used. Therefore, the testing strategy was adapted to use tolerance-based comparisons rather than exact error bounds. Several runs of an implementation were also required to validate the stability of the results.

### 3.4.3 Implementation profiling and kernel-level analysis

NVIDIA Nsight Compute was primarily used to profile each implemented CUDA kernels and identify distinct performance bottlenecks for the various implementations. Two profiling approaches were used. The first was runtime-focused kernel breakdown profiling, which provided execution time and resource utilization metrics for each kernel. This was used to identify any mistakes during development which caused excessive runtimes, most notably this was to identify thread divergence and improper kernel launch configurations.

The second approach was to utilize Nsight Compute's metrics for a roofline analysis. This collects statistics on floating-point operations, DRAM bytes transferred, and kernel duration to determine whether each kernel is either compute-bound or memory-bound relative to the currently utilized GPU's theoretical peak performance. These profiling approaches were compiled into a set of useful scripts to automate the collection of profiling data across implementations and configurations, which was used to visualize key improvements using Bash and Python. This was further used to generate the various charts and plots used in chapter 4 Results.

### 3.4.4 Benchmarking suite

The benchmarking suite includes three different *levels* of performance measurements. Micro-benchmarks, implemented using the Catch2 benchmarking macros, measures the end-to-end execution time of several parts of the implementations. One important use for micro-benchmark were for generating the matrices used as input with their accompanied product matrix. This was used to improve the scaling of the generation in order to be able to generate massively large matrices. The primary scaling benchmark sweeps across matrix sizes from $2^{10}$ to $2^{17}$. The benchmark records the minimum, maximum and median execution times to estimate the scaling behavior of the implementations. Timing was more crucial in this benchmark and was therefore performed by CUDA events (cudaEventRecord) on the default CUDA stream. This provided GPU-accurate timestamps without introducing unnecessary synchronization with the host CPU, which could heavily skew the results. These benchmarks were then compiled into CSV files that could later be used to generate scaling and error plots.

# 4

# Results

This chapter presents the final benchmark results obtained from the GPU implementations of Pagh's algorithm for CMM. The primary focus is execution time, scaling behavior, and the relative effect of the different strategies used when developing the implementations. The results showcase the feasibility of a GPU-based approach for the algorithm and that a functional GPU implementation scales better in comparison with a state-of-the-art implementation of GPU-based GEMM.

All of the subsequent results are gathered with benchmarks utilizing a synthetic dataset of the *logunit* set of matrices with characteristics which the algorithm is designed to accelerate [5]. The benchmarks are exclusively performed on square matrices. When referring to the size of a matrix, the use of a single size (e.g. $2^{12}$), should be interpretted as a square matrix with that size as it's side length (e.g. $2^{12} \times 2^{12}$). The *cucmm-stream* will also be omitted from the results, as it was scraped in favor of the NVIDIA streams with cuFFTDx implementation.

## 4.1 Implementation Variants

To evaluate the different strategies for implementing Pagh's algorithm on a GPU, we have developed four distinct 'implementations', or versions of the algorithm, each with its own approach to parallelization, but most notably the involved Fourier Transformations. These implementations are all presented to showcase positive and negative effects on their performance as they were developed iteratively over the project's course. The four implemented versions are as follows:

- **Naive (*cucmm-naive*):** This version is a straightforward translation of the prototype implemented in the planning phase of the project, which in itself is a naive CPU-based implementation.
- **Naive with cuFFTDx library (*cucmm-cufftdx*):** This version utilizes the cuFFTDx library, which is a high-performance sub-library from NVIDIA's MathDx Package, a device extension package which lets kernels seamlessly utilize the GPU's shared memory for FFT computations.
- **Stream-based with cuFFTDx library (*cucmm-cufftdx-stream*):** This version combines the use of CUDA streams to overlap computation and data transfer with the cuFFTDx library for FFT computations. This approach aims to combine the performance benefits of both techniques.

- **Tiled/chunked approach (*cucmm-tiled*):** A tiled implementation made to reduce the memory requirements for massively large matrices. Based upon the naive implementation, and developed by employing widely used tiling approach to parallelism. Breaks up the matrices into smaller chunks to same allocated VRAM.

These implementations are later compared with a baseline tiled GEMM implementation (*cublas-tiled*). This version utilizes the cuBLASXt API library, which is a GPU-accelerated version of the Basic Linear Algebra Subprograms (BLAS) library with extensions for multi-gpu and tiling behavior. This specific implementation of BLAS was used due to the requirements of massively large matrices which would not fit in the experimental setup's VRAM.

## 4.2 Accuracy Validation and Performance Constraints

Because compress matrix multiplication involves intermediate structures such as sketches and polynomial buckets, our initial versions maps these entire structures to individual GPU thread blocks. As the matrix dimension $N$ scales, the required allocation storage needed rapidly exceeds the hardware limits of the GPU's VRAM (24 GB for the NVIDIA L4's used for experimentation). This results in a constrained dataset for the evaluation of the early developed implementations.

The matrix dimension limitations were overcome with the modifications of the *cucmm-tiled* implementation, this allowed a larger dataset for evaluations as massively larger matrices could be processed.

It's worth to note that the accuracy of the output relies heavily on the parameters of the input which are the number of sketches $d$, number of accuracy-building repetitions, and the associated hash functions of bucket size $b$. These parameters will evaluate to different performance gains and accuracy guarantees. Therefore, it was deemed necessary to establish a procedure for selecting appropriate parameters. It was decided to follow the proposed procedure by Andersson and Karppa for *logunit-type* matrices in [5]. The used formulas for parameters are:

$$d = 2 \cdot \left\lfloor \frac{\log_2 n}{8} \right\rfloor + 1$$

$$b = \frac{n}{4}$$

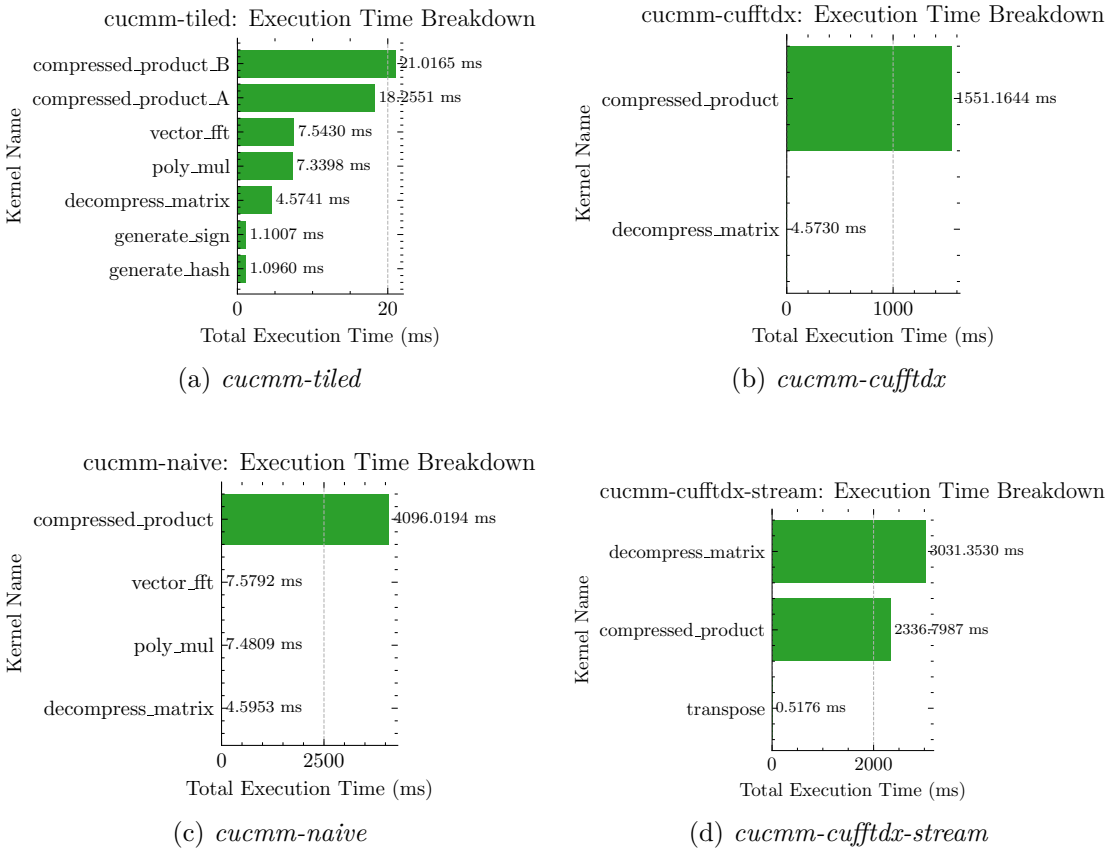
## 4.3 Kernel-level performance evaluation

Figure 4.1 shows a dissected view of the execution times of the different kernels involved in the implementations, which inherently correspond to the algorithm's different stages (e.g. FFT, element-wise multiplication, and hashing).

The benchmark runs for the kernel breakdowns in Figure 4.1 were run with a matrix size of $2^{12}$. This is a matrix size that all implementations support, and they also show the implementation's defining characteristics.

Profiling data illustrates the change to *cucmm-tiled* that moves the generation of the hashes directly on GPU memory. It also gives clear indication that the compression kernel is more expensive than all other kernels combined for all implementations. Since *cucmm-cufftdx* incorporates the FFT in its compression kernel, there is also a lack of data since the implementation essentially only has two kernels.

It is also worth noting the large spike in execution time for the decompression kernel for the *cucmm-cufftdx-stream* implementation. This is due to the profiling characteristics in how Nsight Compute calculates the runtime for the separate kernels. This implementation utilizes multiple CUDA streams for concurrent execution, this then shows that the kernels are launched at the same time, and that the decompression kernel is the last one to finish. That means that the kernel is most likely idle while waiting for the prior step of the compression kernel to be finished executing. Therefore, these implementation's execution times shouldn't be compared to each other, but to visualize the share of execution time being consumed by each kernel.




(a) *cucmm-tiled* (b) *cucmm-cufftdx*

(c) *cucmm-naive* (d) *cucmm-cufftdx-stream*

Figure 4.1: Kernel breakdown comparison across implementations

## 4.4 Roofline Model

The roofline models in Figures 4.2, 4.3, 4.4, and 4.5, demonstrates the operational intensity (FLOPs/byte) of our kernels. They primarily indicate that our implementations are currently predominantly memory-bound rather than compute-bound, which is expected given the nature of the implemented algorithm. This also validated our analysis that global memory traffic is the primary bottleneck of performance. This elevates the reasoning that maximizing shared memory usage (and re-usage) is one of the most promising optimization steps.

Figures 4.2, 4.3, 4.4, and 4.5 were run with a matrix size of $2^{12}$. This is a matrix size that all implementations support, and they also produced roofline models which highlighted the kernels most accurately. The theoretical bounds are calculated from the specifications of the L4 GPUs, as part of the experimental setup described in Section 3.3.

## 4.5 Scaling analysis

Figure 4.6 shows the total execution time as a function of matrix size for each implementation, also including the baseline implementation of cuBLAS in 4.6b. The expected scaling behavior of the algorithm is present, and also shows improved scaling compared to that of the baseline GEMM implementation.

Figures 4.6c, 4.6d, 4.6e illustrates the implementations supported input matrix sizes $2^{10}$ to $2^{12}$. Sizes over $2^{12}$ are unsupported for these versions. Figures 4.6a, 4.6b instead shows the two tiled implementations which supports up to the maximum matrix size for the benchmarks $2^{17}$.

Figure 4.7 combines the execution time scaling curves into a comparison graph for ease of analysis.

All of the benchmark results used for these graphs are also present in table A.1.

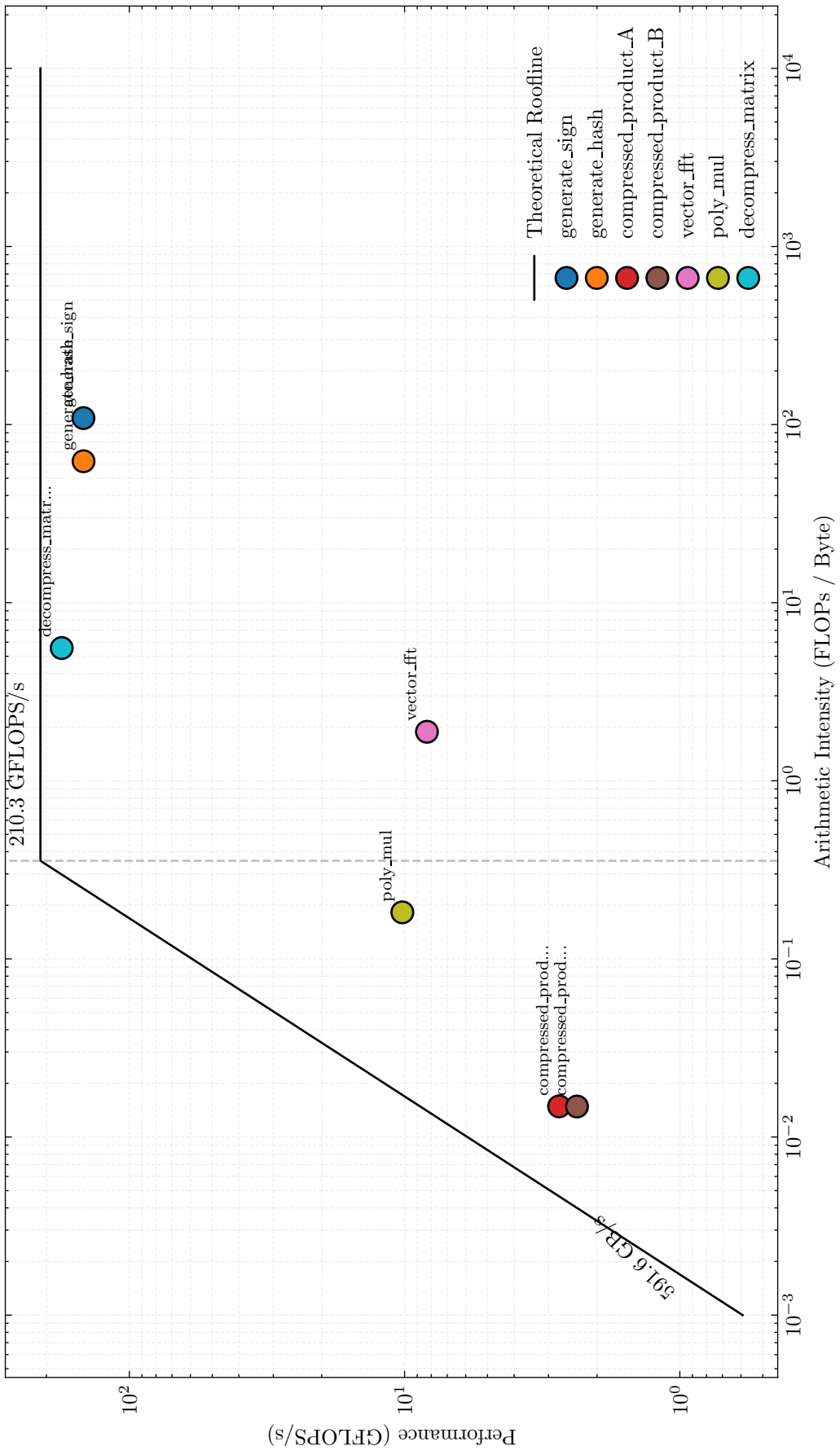
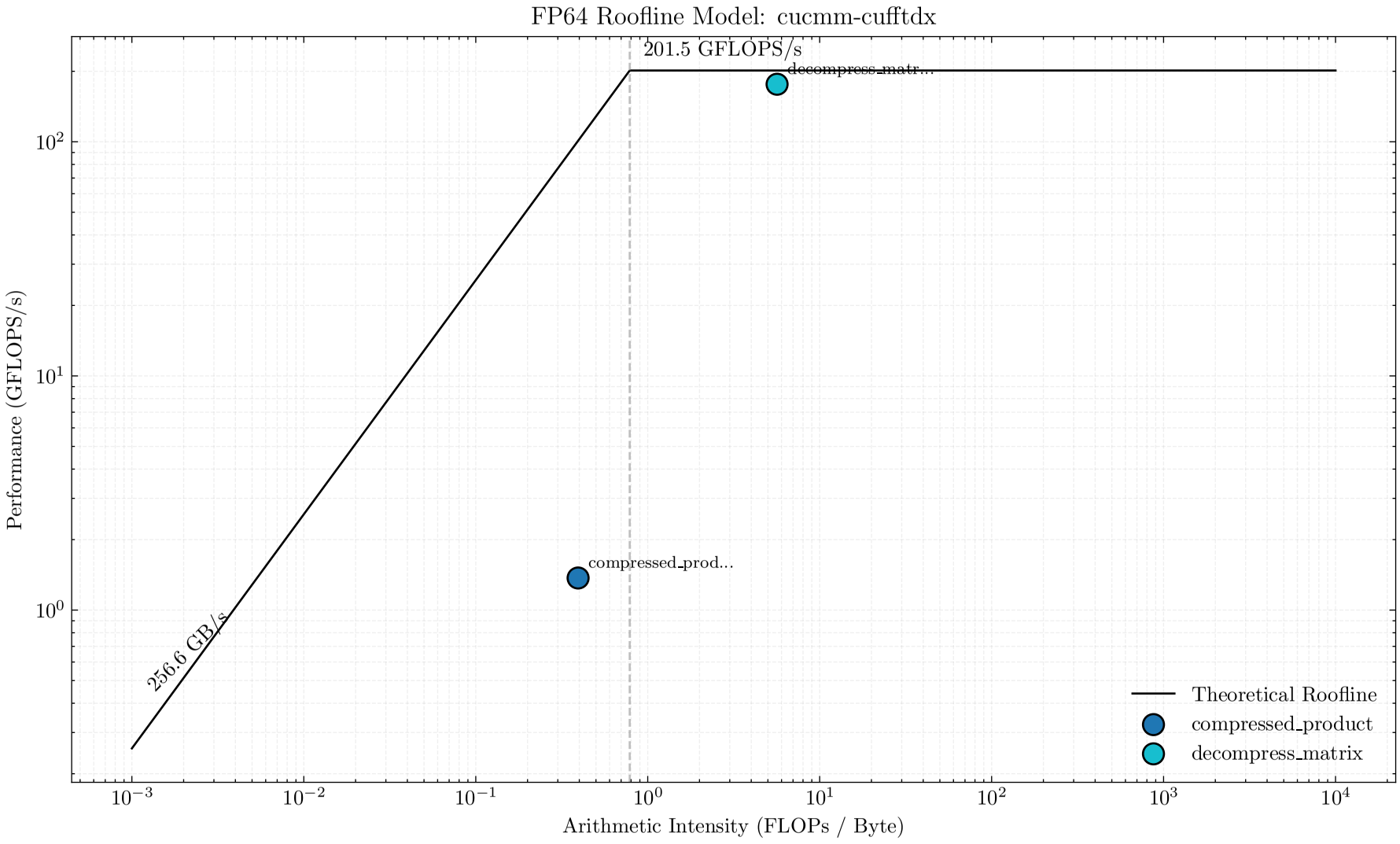



Figure 4.2: Roofline model of *cucmm-tiled*

Figure 4.3: Roofline model of *cucmm-cufftdx*



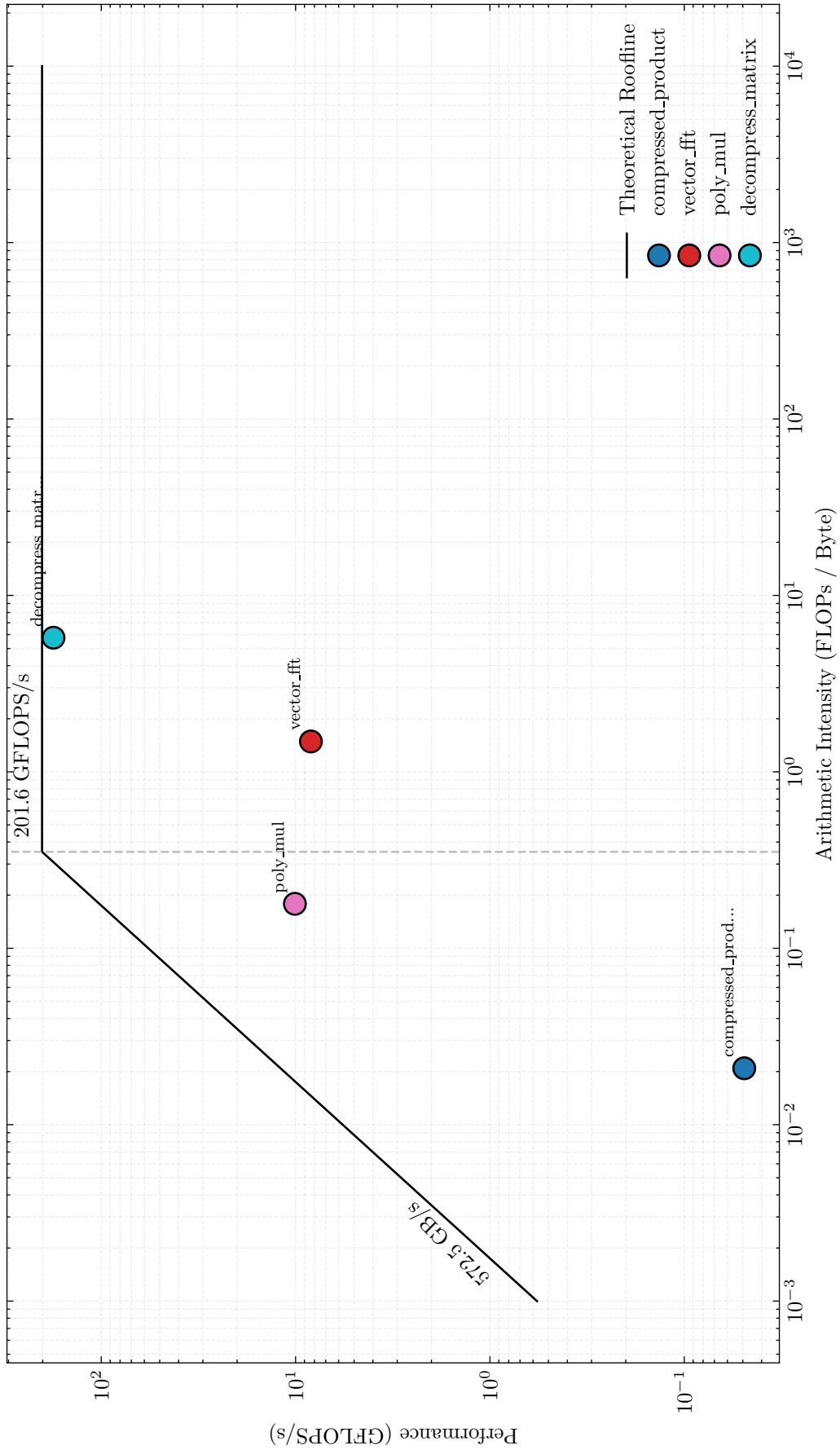
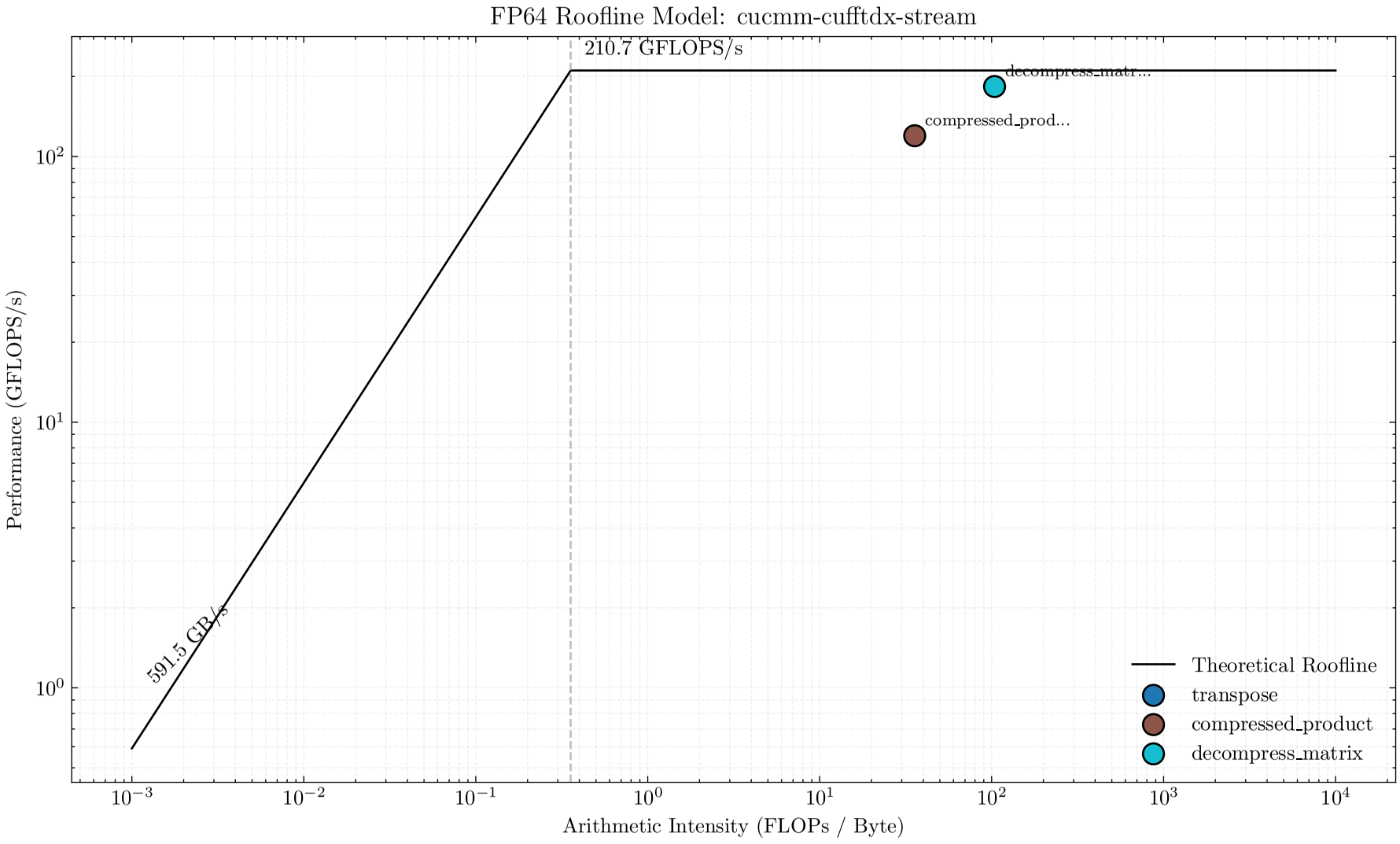
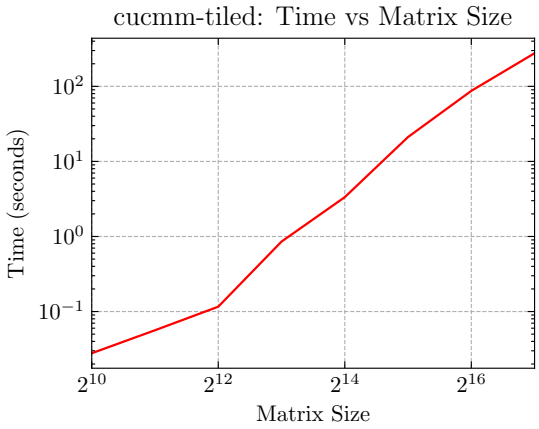


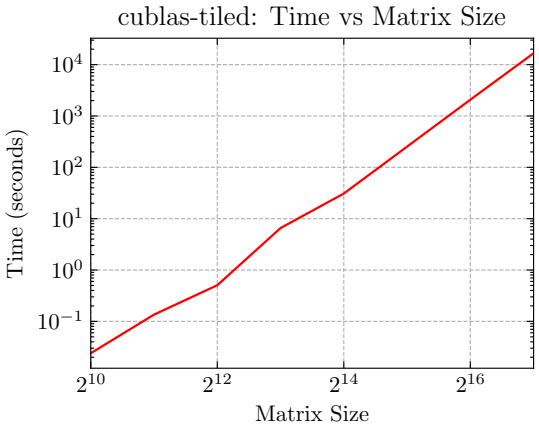

Figure 4.4: Roofline model of *cucmm-naive*

Figure 4.5: Roofline model of *cucmm-cufftdx-stream*

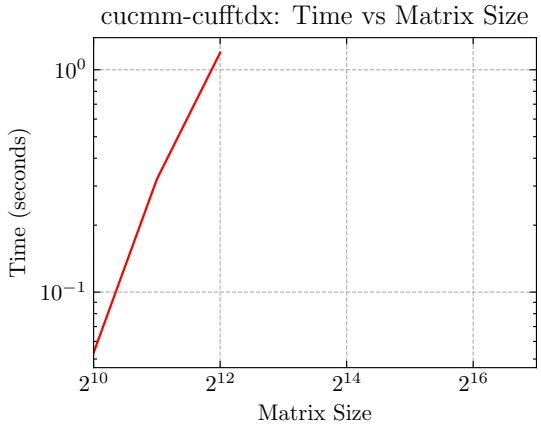




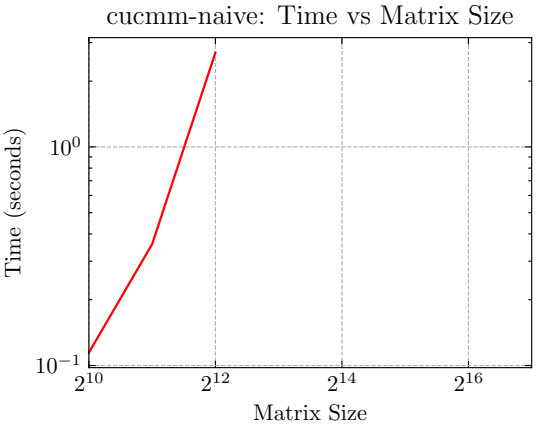

(a) *cucmm-tiled*



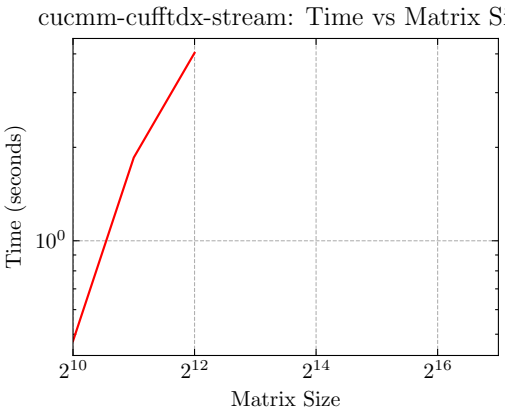

(b) *cublas-tiled*




(c) *cucmm-cufftdx*




(d) *cucmm-naive*




(e) *cucmm-cufftdx-stream*

Figure 4.6: Scaling analysis

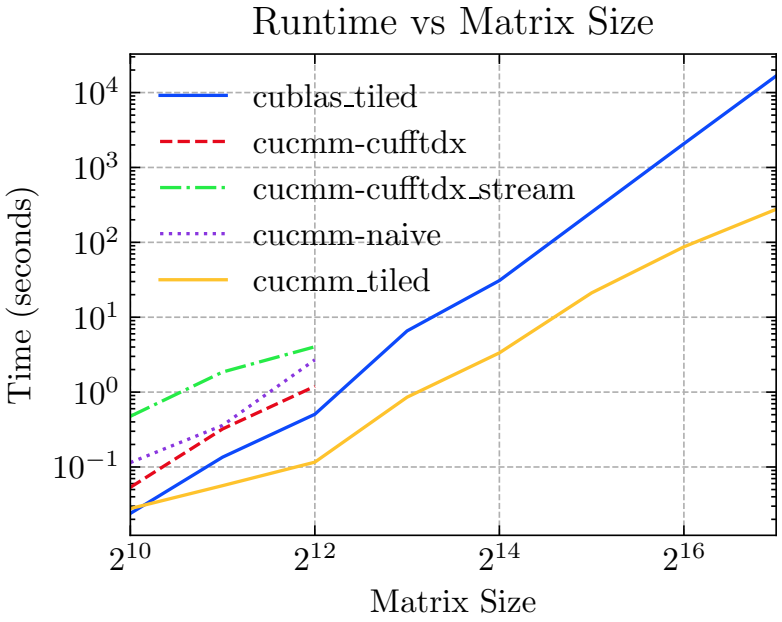



Figure 4.7: Scaling analysis for all implementations

## 4.6 Speedup comparisons

Figure 4.8 compares speedup gains of the tiled and naive implementations in comparison to the baseline, tiled cuBLAS implementation. Another comparison between the naive and tiled implementations are also included which shows the progress of performance over the course of development.

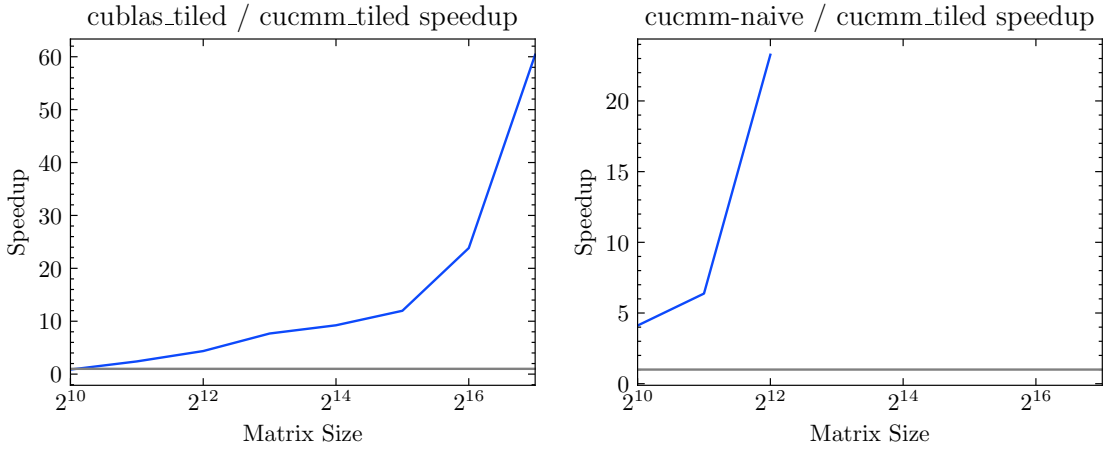
The gray line present in the speedup graphs indicate a speedup of 1 relative to *cublas-tiled*.

All of the benchmark results used for these graphs are also present in table A.2.

Figure 4.8: Speedup comparison of select implementations

(a) *cublas-tiled* / *cucmm-tiled*

(b) *cucmm-naive* / *cucmm-tiled*



# 5

# Discussion

The results demonstrate that the proposed implementation *cucmm-tiled* is capable of outperforming the cuBLAS baseline for sufficiently large matrix sizes, while the earlier implementations remain limited by limiting bottlenecks. This chapter analyzes the main causes behind these performance differences, and gives suggestions for potential improvements to performance and scalability.

## 5.1 Known performance limitations

As seen in figure 4.7 the tiled implementation starts to outperform cuBLAS at matrix sizes of $2^{11}$ and larger. This can not be said for the other implementations which never succeed in surpassing cuBLAS. Considering the asymptotic difference in the algorithms it was inevitable that all implementations would become faster at a certain matrix size threshold, unfortunately the other implementations can not handle sizes bigger than what is able to fit on the VRAM on the GPU so it's unknown when it would have happened. This asymptotic difference comes from the fact that CMM is not limited entirely by matrix size, instead it is also limited by the amount of sketches and buckets used. The specific matrix type that was used ensures that the amount of sketches scale logarithmically while the buckets scale linearly.

### 5.1.1 Branch divergence issue

The subpar performance from some of the earlier implementations is mostly due to severe branch divergence which led to a sizable portion of threads being idle in the SIMT stack. There was an attempt to ensure that the polynomials were written to in a more scheduled manner to avoid race conditions due to inherent random access patterns when using the random 2-wise independent hash families for indexing. Some threads were then deliberately ignored for better access patterns but this instead led to worse performance.

### 5.1.2 Subpar performance of *cucmm-cufftdx-stream*

The *cucmm-cufftdx-stream* implementation does not suffer from the branch divergence issue. Instead it seems to suffer from less effective thread utilization due to the streams not being properly handled. Continually, the generation of hashes on shared memory meant that both the compression step and decompression step needed to

create the hashes from scratch since shared memory is freed automatically when the thread block has finished executing a CUDA kernel. Therefore, there was no apparent way to efficiently transfer the content stored in the L1-cache to the decompression step.

## 5.2 Unknown characteristics of cuBLASXt

An important consideration to make is the effectiveness of the baseline implementation of the cuBLAS library and the cuBLASXt API. It was developed according to the associated programming guide and all parameters for the tiling approach of the API are unchanged [22]. This was decided to keep the baseline to it's most default configuration, with no tuning of parameters. There are options in cuBLASXt to, as an example, set a manual block dimension used for the tiling of the matrices for the subsequent Math function calls [21].

## 5.3 Future Work

There are some paths that could be interesting to investigate given more time on the project. There are also a few that wouldn't provide much improvement. For example, changes or additions to the compression step are less important since it represents a small fraction of the entire execution time.

### 5.3.1 Multi-GPU implementation

Using multiple GPUs might be effective if each GPU makes an unique sketch. Sketches are entirely independent until the decompression step. This would mean that each GPU could finish the compression and decompression step for one sketch and then accumulate the results to one of the GPUs and perform median estimates for each element. For the logunit-type matrices this would not scale that well since not many sketches are needed for accurate results but this could work well with other types of matrices that might need more sketches for accurate results. cuBLASXt already allows for scheduling over several GPU devices which makes it straightforward for benchmarking.

### 5.3.2 Tiled implementation utilizing cuFFTDx library

As shown in figure 4.7 cuFFTDx had the best execution time out of the initial implementations. This was due to the immensely faster version of cuFFT that can run directly in a CUDA kernel. Due to lack of time cuFFTDx was not implemented in final tiled version. However, during internal testing it was estimated that the speedup to the FFT was approximately 2 which means that for the tiled implementation this would probably equate to a 1.1 speedup, since the FFT step represents a 1/4th of the execution time. Ultimately, no metrics in figure 4.1 can show this since cuFFTDx was used in the same kernel as the compression step so this estimation is speculative.

### 5.3.3 Performance comparison for other matrix types

All results gathered during this thesis are focused on the use of the *logunit* matrix instance. Although other matrix types were considered and correctly implemented, they were ultimately excluded due to its lack of relevance to the thesis. One can argue that the reported speedups compared to the baseline could be widely different depending on the used matrix, which is likely due to the algorithm's focus on specific matrix multiplication circumstances. A analysis of other matrix types could therefore be deemed necessary and should be carried out to demonstrate the robustness of the proposed implementation.

# 6

# Conclusion

The implementation of Pagh's CMM on the GPU using the CUDA programming model surpassed cuBLASXt's implementation of GEMM with a speedup of 60.38 when performed on sufficiently big matrix sizes of the *logunit* instance. This suggests that CUDA is a practical platform for CMM since the algorithm contains several characteristics that allows it to utilize parallelized hardware, although it's main performance limitation is host-device memory transfers. There are still possible optimizations that might increase performance. Like a implementation targeting multiple GPU's implementation which could assist in reducing memory transfer overhead, which is possible with the independent sketching of the CMM algorithm.

# Bibliography


[1] Tor M. Aamodt, Wilson Wai Lun Fung, and Timothy G. Rogers. *General-Purpose Graphics Processor Architectures*. Synthesis Lectures on Computer Architecture. Springer International Publishing, 2018. ISBN: 978-3-031-00631-9. DOI: 10.1007/978-3-031-01759-9. URL: https://link.springer.com/10.1007/978-3-031-01759-9.

[2] Josh Alman and Virginia Vassilevska Williams. "A Refined Laser Method and Faster Matrix Multiplication". In: *TheoretiCS* Volume 3 (Sept. 2024), pp. 522–539. ISSN: 2751-4838. DOI: 10.46298/THEORETICS.24.21. URL: https://theoretics.episciences.org/14213.

[3] AMD. *Accelerator and GPU hardware specifications*. URL: https://rocm.docs.amd.com/en/docs-6.2.2/reference/gpu-arch-specs.html.

[4] AMD. *rocBLAS 5.1.1 Documentation*. URL: https://rocm.docs.amd.com/projects/rocBLAS/en/latest/.

[5] Joel Andersson and Matti Karppa. "Engineering Compressed Matrix Multiplication with the Fast Walsh-Hadamard Transform". Jan. 2026. URL: http://arxiv.org/abs/2601.09477.

[6] Mufakir Qamar Ansari and Mudabir Qamar Ansari. "Accelerating Matrix Multiplication: A Performance Comparison Between Multi-Core CPU and GPU". July 2025. URL: https://arxiv.org/pdf/2507.19723.

[7] Grey Ballard et al. "Communication-Optimal Parallel Algorithm for Strassen's Matrix Multiplication". In: *Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Feb. 2012), pp. 193–204. DOI: 10.1145/2312005.2312044. URL: https://arxiv.org/pdf/1202.3173.

[8] L. Susan Blackford et al. "An Updated Set of Basic Linear Algebra Subprograms (BLAS)". In: *ACM Transactions on Mathematical Software* 28.2 (June 2002), pp. 135–151. ISSN: 00983500. DOI: 10.1145/567806.567807. URL: https://dl.acm.org/doi/pdf/10.1145/567806.567807.

[9] Moses Charikar, Kevin Chen, and Martin Farach-Colton. "Finding frequent items in data streams". In: *Theoretical Computer Science* 312.1 (Jan. 2004), pp. 3–15. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(03)00400-6.

[10] Don Coppersmith and Shmuel Winograd. "Matrix multiplication via arithmetic progressions". In: *Journal of Symbolic Computation* 9.3 (Mar. 1990), pp. 251–280. ISSN: 0747-7171. DOI: 10.1016/S0747-7171(08)80013-2.

[11] Thomas H Cormen et al. *Introduction to Algorithms*. Third Edition. The MIT Press, 2009, pp. 898–911. ISBN: 978-0-262-04630-5.

[12] Google. *GoogleTest - Google Testing and Mocking Framework*. URL: https://github.com/google/googletest.

[13] Nicholas J. Higham. "Exploiting Fast Matrix Multiplication Within the Level 3 BLAS". In: *ACM Transactions on Mathematical Software (TOMS)* 16.4 (Jan. 1990), pp. 352–368. ISSN: 15577295. DOI: 10.1145/98267.98290. URL: https://dl.acm.org/doi/10.1145/98267.98290.

[14] Jianyu Huang. "Practical fast matrix multiplication algorithms". Aug. 2018. DOI: 10.15781/T2V11W511. URL: http://hdl.handle.net/2152/69013.

[15] Xiaohan Huang and Victor Y. Pan. "Fast Rectangular Matrix Multiplication and Applications". In: *Journal of Complexity* 14.2 (June 1998), pp. 257–299. ISSN: 0885-064X. DOI: 10.1006/JCOM.1998.0476. URL: https://www.sciencedirect.com/science/article/pii/S0885064X98904769.

[16] Aaron Jarmusch and Sunita Chandrasekaran. "Microbenchmarking NVIDIA's Blackwell Architecture: An in-depth Architectural Analysis". 2025. URL: https://arxiv.org/pdf/2512.02189v1.

[17] Md Abul Kalam Azad et al. "An Empirical Study of High Performance Computing (HPC) Performance Bugs". In: *Proceedings - 2023 IEEE/ACM 20th International Conference on Mining Software Repositories, MSR 2023* 20 (2023), pp. 194–206. DOI: 10.1109/MSR59073.2023.00037. URL: https://ieeexplore.ieee.org/document/10174003.

[18] Elaye Karstadt and Oded Schwartz. "Matrix multiplication, a little faster". In: *Annual ACM Symposium on Parallelism in Algorithms and Architectures* Part F129316.17 (July 2017), pp. 101–110. DOI: 10.1145/3087556.3087579. URL: /doi/pdf/10.1145/3087556.3087579?download=true.

[19] Kitware. *CMake*. URL: https://cmake.org/.

[20] Rajeev. Motwani and Prabhakar. Raghavan. *Randomized algorithms*. Cambridge University Press, Aug. 1995, p. 476. ISBN: 0521474655.

[21] NVIDIA Corporation. *cuBLAS 13.0 documentation*. URL: https://docs.nvidia.com/cuda/cublas/index.html.

[22] NVIDIA Corporation. *CUDA Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-programming-guide/index.html.

[23] NVIDIA Corporation. "cuSPARSE Release 13.1". In: (2026).

[24] NVIDIA Corporation. *Nsight Compute | NVIDIA Developer*. URL: https://developer.nvidia.com/nsight-compute.

[25] Rasmus Pagh. "Compressed matrix multiplication". In: *ACM Transactions on Computation Theory* 5.3 (Aug. 2013). ISSN: 19423462. DOI: 10.1145/2493252.2493254. URL: https://dl.acm.org/doi/10.1145/2493252.2493254.

[26] Samuel Runmark Thunell and Egil Guting. *cuCMM*. 2026. DOI: 10.5281/zenodo.20447310. URL: https://doi.org/10.5281/zenodo.20447310.

[27] Shaohuai Shi, Qiang Wang, and Xiaowen Chu. "Efficient sparse-dense matrix-matrix multiplication on GPUs using the customized sparse storage format". In: *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS* 2020-December.26 (Dec. 2020), pp. 19–26. ISSN: 15219097. DOI: 10.1109/ICPADS51040.2020.00013. URL: https://ieeexplore.ieee.org/document/9359142.

[28] V. Strassen. "Proc. 27th Ann. Symp. on Foundation of Computer Science (FOCS)". In: *Annual Symposium on Foundations of Computer Science (Proceedings)* 27 (1986), pp. 49–54. ISSN: 02725428. URL: https://api.semanticscholar.org/CorpusID:15077423#id-name=S2CID.

[29] Volker Strassen. "Gaussian elimination is not optimal". In: *Numerische Mathematik* 13.4 (Aug. 1969), pp. 354–356. ISSN: 0029599X. DOI: 10.1007/BF02165411/METRICS. URL: https://link.springer.com/article/10.1007/BF02165411.

[30] The BLAS Technical Forum. *BLAS (Basic Linear Algebra Subprograms)*. DOI: 10.1145/355841.355847. URL: https://www.netlib.org/blas/.

[31] The HDF Group. *The HDF5ő Library & File Format*. URL: https://www.hdfgroup.org/solutions/hdf5/.

[32] Carl Yang, Aydn Buluç, and John D. Owens. "Design Principles for Sparse Matrix Multiplication on the GPU". In: *Euro-Par 2018: Parallel Processing*. Ed. by Marco Aldinucci, Luca Padovani, and Massimo Torquati. Springer International Publishing, 2018, pp. 672–687. ISBN: 978-3-319-96983-1. URL: https://link.springer.com/10.1007/978-3-319-96983-1_48.

[33] Orestis Zachariadis et al. "Accelerating sparse matrixmatrix multiplication with GPU Tensor Cores". In: *Computers & Electrical Engineering* 88 (Dec. 2020), p. 106848. ISSN: 0045-7906. DOI: 10.1016/J.COMPELECENG.2020.106848. URL: https://www.sciencedirect.com/science/article/pii/S0045790620307011?via%3Dihub.

# A

# Execution times of implementations

Table A.1 reports the median execution times over a number of runs for all implementations, matrix sizes $2^{10}$ to $2^{14}$ has a median over 100 runs, $2^{15}$ to $2^{16}$ has a median over 10 runs, and $2^{17}$ was run 1 time due to its scale. Each run employed it's own generated matrices with a defined randomization seed. Table A.2 reports the speedups of implementation *cucmm-tiled* in comparison with baselines *cublas-tiled* and *cucmm-naive*. The reported speedups are calculated according to the execution times from table A.1.

Table A.1: Execution times of the implementations in seconds

| Matrix Size | cucmm_tiled | | | cublas_tiled | | | cucmm-cufftdx | | | cucmm-naive | | | cucmm-cufftdx_stream | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ | min | median | max | min | median | max | min | median | max | min | median | max | min | median | max |
| $2^{10}$ | 0.016 | 0.028 | 0.627 | 0.021 | 0.024 | 0.255 | 0.042 | 0.053 | 13.361 | 0.101 | 0.115 | 0.336 | 0.237 | 0.475 | 2.066 |
| $2^{11}$ | 0.050 | 0.056 | 0.192 | 0.131 | 0.135 | 0.450 | 0.159 | 0.322 | 13.653 | 0.350 | 0.359 | 0.631 | 0.932 | 1.854 | 1.888 |
| $2^{12}$ | 0.109 | 0.116 | 0.425 | 0.501 | 0.506 | 0.676 | 1.173 | 1.194 | 1.455 | 2.675 | 2.699 | 2.835 | 3.566 | 4.030 | 4.341 |
| $2^{13}$ | 0.839 | 0.855 | 2.490 | 6.554 | 6.566 | 9.167 | – | – | – | – | – | – | – | – | – |
| $2^{14}$ | 1.867 | 3.341 | 5.014 | 29.909 | 30.849 | 32.147 | – | – | – | – | – | – | – | – | – |
| $2^{15}$ | 14.278 | 21.140 | 21.777 | 250.117 | 253.137 | 484.370 | – | – | – | – | – | – | – | – | – |
| $2^{16}$ | 56.032 | 87.086 | 97.081 | 2021.900 | 2075.270 | 2162.520 | – | – | – | – | – | – | – | – | – |
| $2^{17}$ | 276.358 | 276.358 | 276.358 | 16685.700 | 16685.700 | 16685.700 | – | – | – | – | – | – | – | – | – |

Table A.2: Speedups relative to tiled_single_gpu with baselines cublas_tiled and naive_single_gpu.

| Matrix Size | cublas_tiled / tiled_single_gpu | naive_single_gpu / tiled_single_gpu |
|---|---|---|
| $2^{10}$ | 0.86x | 4.11x |
| $2^{11}$ | 2.40x | 6.38x |
| $2^{12}$ | 4.36x | 23.27x |
| $2^{13}$ | 7.68x | – |
| $2^{14}$ | 9.23x | – |
| $2^{15}$ | 11.97x | – |
| $2^{16}$ | 23.83x | – |
| $2^{17}$ | 60.38x | – |