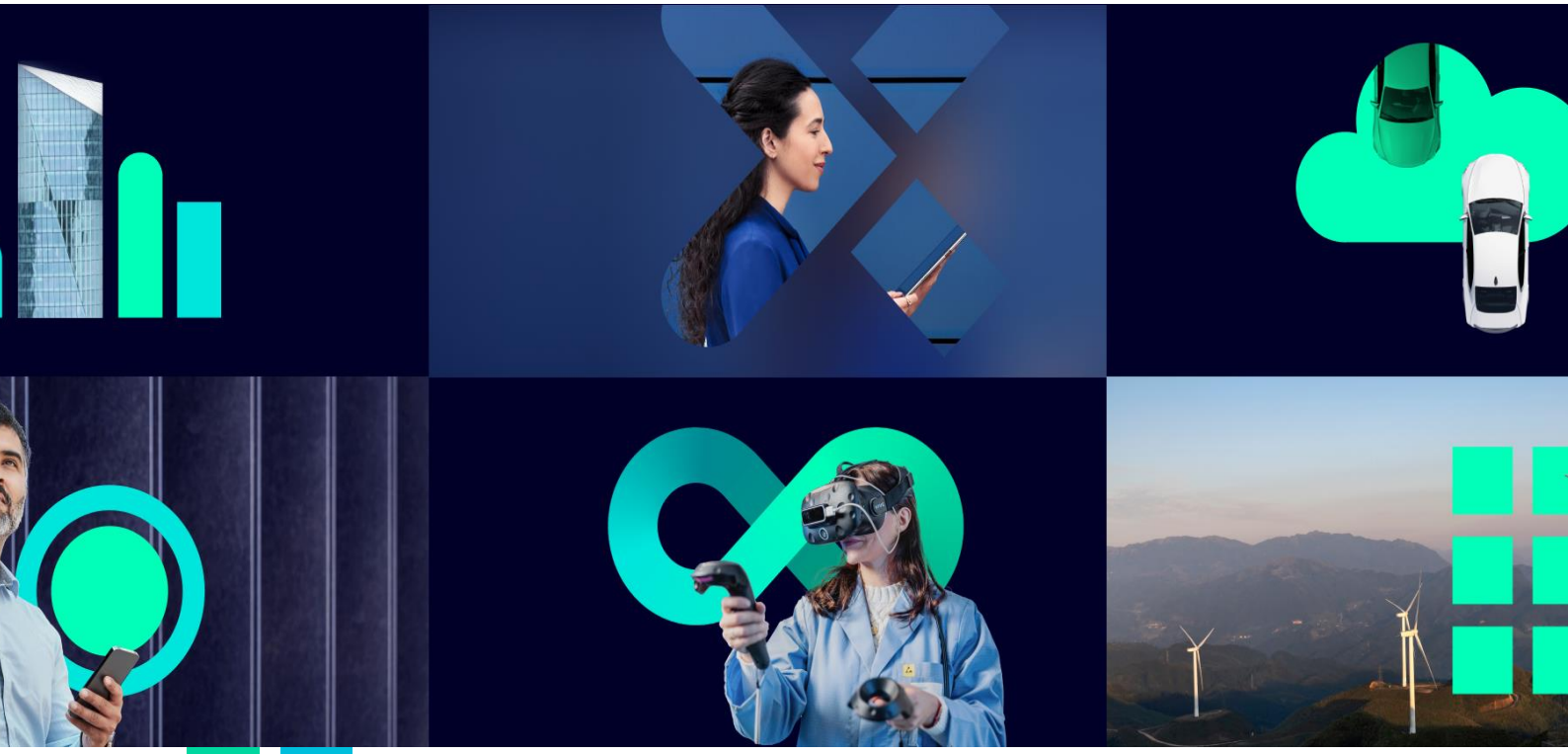


JULIEN GODFROY S427183

SPARSE MATRIX-MULTIVECTOR PRODUCT KERNEL

SMALL SCALE PARALLEL PROGRAMMING

CSTE CIDA Option

Cranfield University

Abstract:

“

This report delves into the realm of high-performance computing for the multiplication of sparse matrices, a computational problem with wide-ranging applications in science and engineering. In this context, we explore the efficiencies afforded by different approaches to sparse matrix multiplication. Using the Compressed Sparse Row (CSR) format, we systematically implement and analyse algorithms in serial, OpenMP, and CUDA paradigms, each subjected to a rigorous testing framework against matrices from the Suite Sparse Matrix Collection. The methodology is comprehensive, beginning with the reading of matrices from the MatrixMarket format and proceeding to the execution of multiplication tasks across various computational strategies. The results exhibit a significant uplift in performance for parallel implementations, with CUDA outshining its counterparts, particularly for larger matrices and heavier workloads. OpenMP also demonstrates notable improvements, although with diminishing returns as thread counts increase. These findings underscore the potential for tailored optimisation in sparse matrix operations, for more efficient high-performance computing applications.

”

Table of Contents

I - Introduction

In the realm of computational science, the efficient processing of large datasets, particularly sparse matrices, has become increasingly critical. This report delves into a project that stands at the confluence of High-Performance Computing (HPC) and sparse matrix manipulation, with a focus on developing a script for the multiplication of a CSR [1] (Compressed Sparse Row) sparse matrix by a long vector. The vectors in question vary in dimensionality, featuring 1, 2, 3, and 6 columns, all sourced initially from the MatrixMarket [2] file format. This endeavor spans across three distinct implementations: a serial version, a parallel version utilising OpenMP [3], and a version harnessing the computational power of CUDA [4].

The primary objective of this project is to demonstrate the effectiveness of HPC techniques in the manipulation of sparse matrices, a common challenge in scientific computing. By implementing the matrix-vector multiplication in serial, OpenMP, and CUDA, we aim to not only showcase the scalability of such operations across different hardware configurations but also to highlight the performance gains achievable through parallel computing. The comparison between these different implementations provides valuable insights into the adaptability and efficiency of HPC solutions in handling large-scale computational tasks.

The scope of this project is carefully defined to encompass the development of the matrix multiplication script, the rigorous testing for correctness and performance, and the analysis of the performance data. Through this focused approach, the project aims to contribute to the broader understanding of HPC's role in computational science, offering a detailed exploration of sparse matrix manipulation techniques. By pushing the boundaries of what's achievable with current computing resources, this project underscores the potential for significant advancements in data processing and scientific computation.

II – Background

HPC Fundamentals

In the domain of computing, High-Performance Computing (HPC) represents an invaluable framework designed to tackle problems of great complexity. HPC harnesses the power of parallel processing, employing multiple computing resources simultaneously to perform vast numbers of calculations in a fraction of the time required by traditional computing methods. This capability is crucial for scientific research, engineering, and data analysis tasks that demand intensive computation and large-scale data processing.

MatrixMaket format and CSR representation

Central to the exploration of HPC applications is the use of specific data formats and storage techniques, particularly for handling sparse matrices. The MatrixMarket file format emerges as a standard for the exchange of matrix data, especially useful in the context of sparse matrices. This format facilitates the efficient storage and access of matrices, which are often used in mathematical modelling and computational simulations. It provides a structured way to represent and share matrix data, ensuring that the sparse matrices are preserved and accurately communicated.

The Compressed Sparse Row (CSR) format is a pivotal element in the efficient storage and manipulation of sparse matrices. Sparse matrices, by nature, are filled predominantly with zero values and only a small fraction of non-zero elements. Storing every element, including zeroes, would lead to a substantial waste of memory and computational resources. The CSR format addresses this issue by storing only the non-zero elements and their respective positions within the matrix. This compact representation not only conserves memory but also optimises operations such as matrix-vector and matrix-matrix multiplication, which are common in various applications ranging from scientific computing to machine learning.

The concept of a "tall matrix" further complements the exploration. In the context of this project, a tall matrix refers to a dense matrix with significantly more rows than columns. This

structure is particularly relevant when dealing with vector operations alongside sparse matrices. The interaction between CSR sparse matrices and tall dense matrices encapsulates a wide array of computational challenges and opportunities, making it a fitting subject for investigation within the realm of HPC.

Through the lens of HPC fundamentals, the MatrixMarket format, and the efficient representation of sparse matrices via the CSR format, this background sets the stage for a detailed examination of sparse matrix multiplication. It underscores the importance of optimised data structures and algorithms in enhancing computational performance, a theme that is recurrent in the realms of scientific research and advanced data analysis.

III – Benchmark methodology

The methodology underpinning this project is designed to systematically evaluate the performance of sparse matrix multiplication across different computational approaches, namely serial, OpenMP, and CUDA implementations. Central to this investigation is the consistent use of 30 matrices in the MatrixMarket format and the creation of tall matrices for multiplication. Below is a detailed description of each segment of the methodology.

For each of these implementations, the input matrices and vectors remain consistent, ensuring that the observed performance differences are attributable solely to the computational strategy and not variations in data. By adopting this structured methodology, the project aims to provide a comprehensive and comparative analysis of sparse matrix multiplication techniques, highlighting the potential benefits and limitations of each approach.

Reading the 30 MatrixMarket Files using mmio and Creating Tall Matrices

The To facilitate a uniform comparison across all implementations, the project employs the Matrix Market I/O (mmio) library for reading sparse matrices stored in the MatrixMarket format. A bespoke function, read_mtx_and_convert_to_csr, is developed to parse these files, convert the matrices into the CSR format, and perform preliminary data validation. This function reads the matrix dimensions, non-zero counts, and then iterates over each non-zero element to populate intermediate arrays, which are subsequently used to construct the CSR format arrays (IRP, JA, and AS).

Additionally, a function, allocate_and_initialize_matrix, generates tall dense matrices with dimensions $M \times k$ (where $k \in \{1,2,3,6\}$), using random values to ensure consistency in input data across the different computational methods.

Serial Implementation

The serial implementation serves as the baseline for performance comparison. It directly translates the mathematical operations of sparse matrix multiplication into a sequential algorithm without leveraging any form of parallelism. The core of this implementation focuses on iterating through the rows of the sparse matrix (stored in CSR format), multiplying non-zero elements by corresponding elements in the tall matrix, and accumulating the results. The computation time and FLOPS (floating-point operations per second) are measured to assess performance.

OpenMP Implementation

The OpenMP implementation introduces shared-memory parallelism to the matrix multiplication process. By distributing the computation across multiple threads, this approach aims to significantly reduce execution time compared to the serial version. The implementation is tested with a range of threads from 1 to 16, allowing for an exploration of scalability and efficiency across different levels of parallelism. Similar to the serial approach, both computation time and FLOPS are recorded for performance evaluation. For the OpenMP tests, the script will be ran 7 times and the average will be mainly considered.

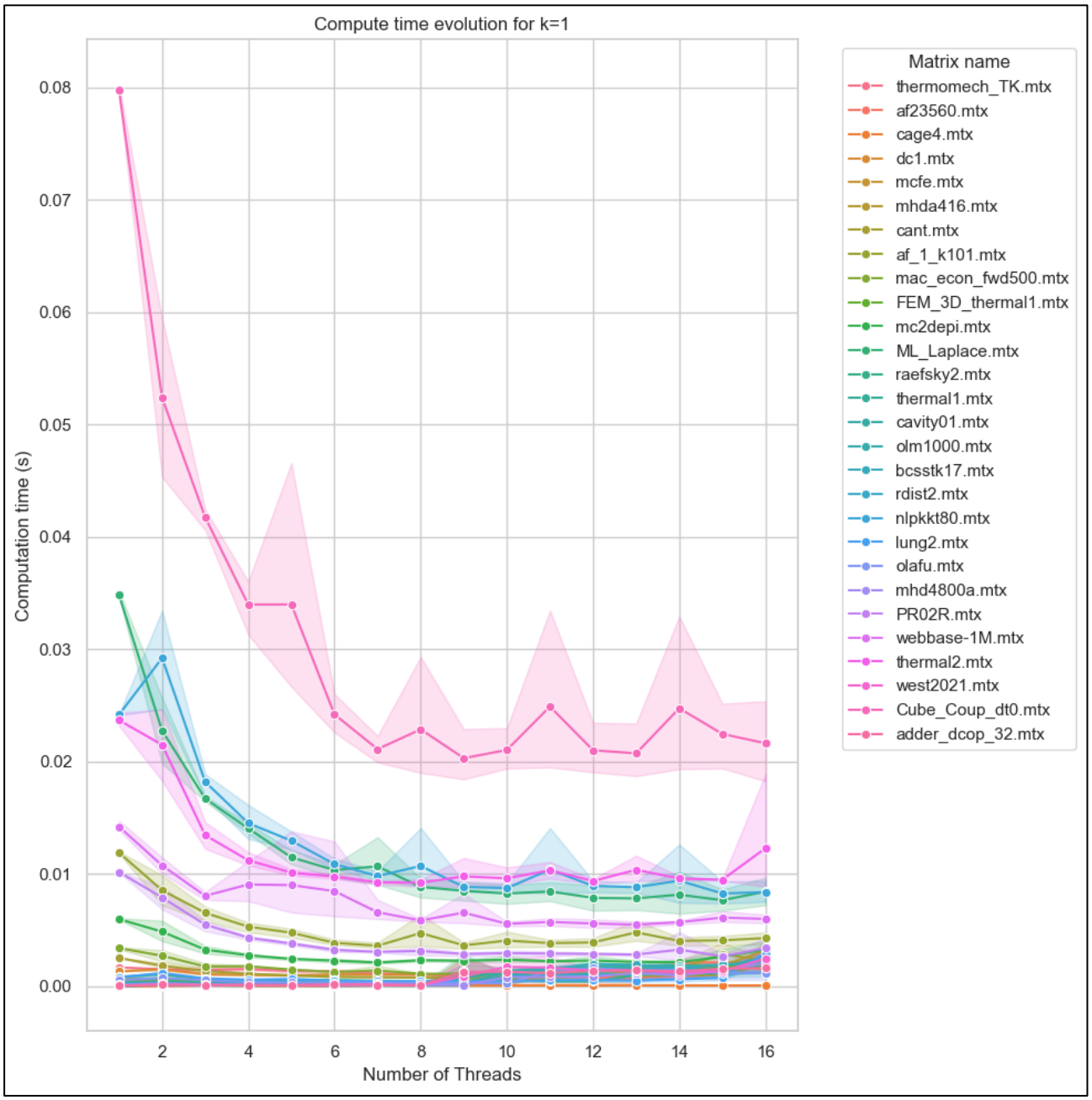
CUDA Implementation

Leveraging the parallel computing capabilities of NVIDIA GPUs, the CUDA implementation represents the most significant departure from serial computation. By assigning portions of the matrix multiplication workload to different blocks and threads within the GPU, this method seeks to exploit the massive parallelism inherent to modern GPUs. The implementation is tested with different block sizes, specifically {8, 16, 64, 128, 256}, to identify the optimal configuration for performance. As with the other implementations, both computation time and FLOPS are crucial metrics for assessing the efficacy of this approach.

IV –Results and Analysis

OpenMP Results

The analysis of the OpenMP implementation results is grounded on the observation of compute time evolution and GFLOPS performance across a varying number of threads (from to 16) and different tall matrix widths (denoted by k values of 1, 2, 3, and 6).



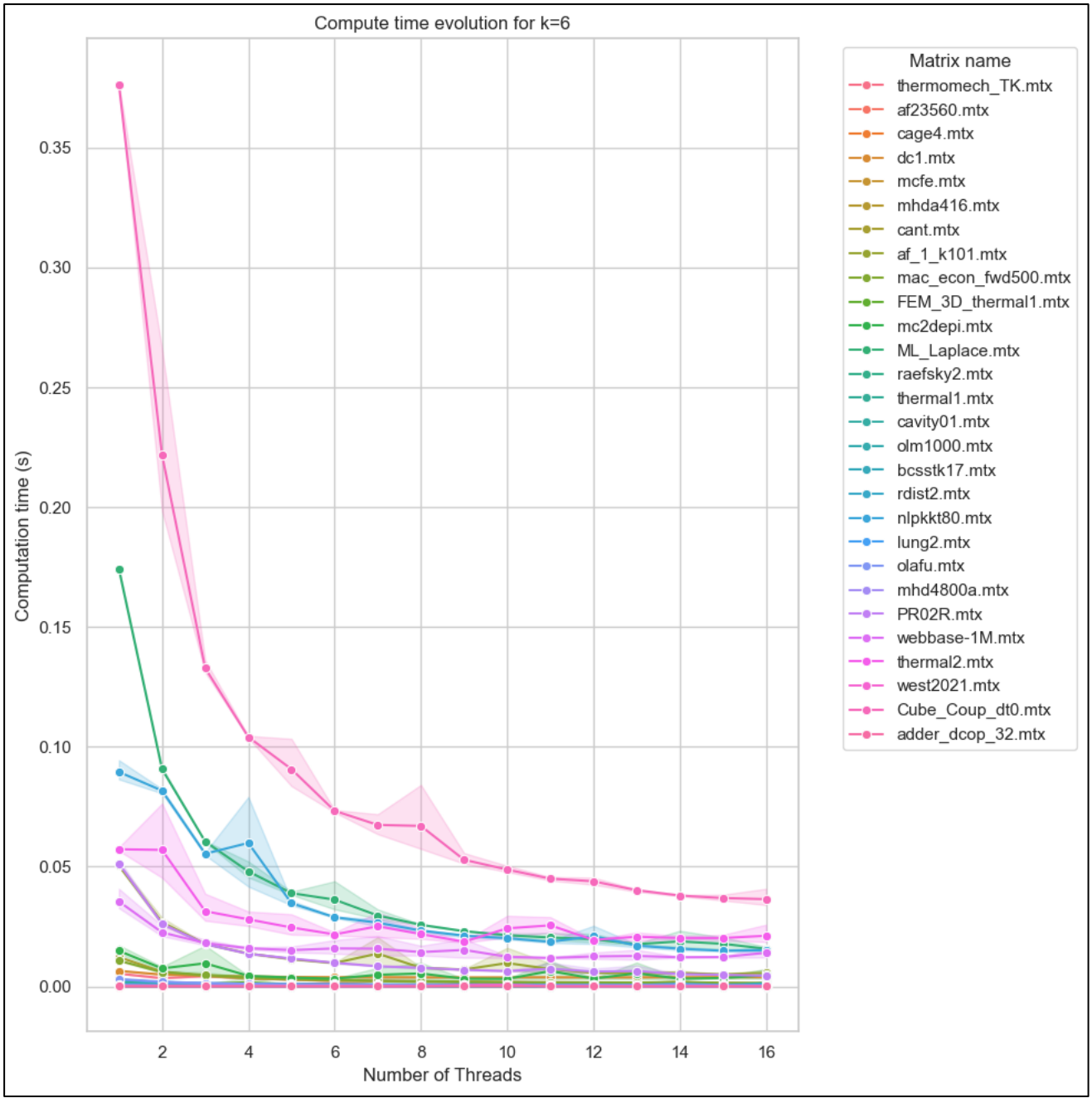
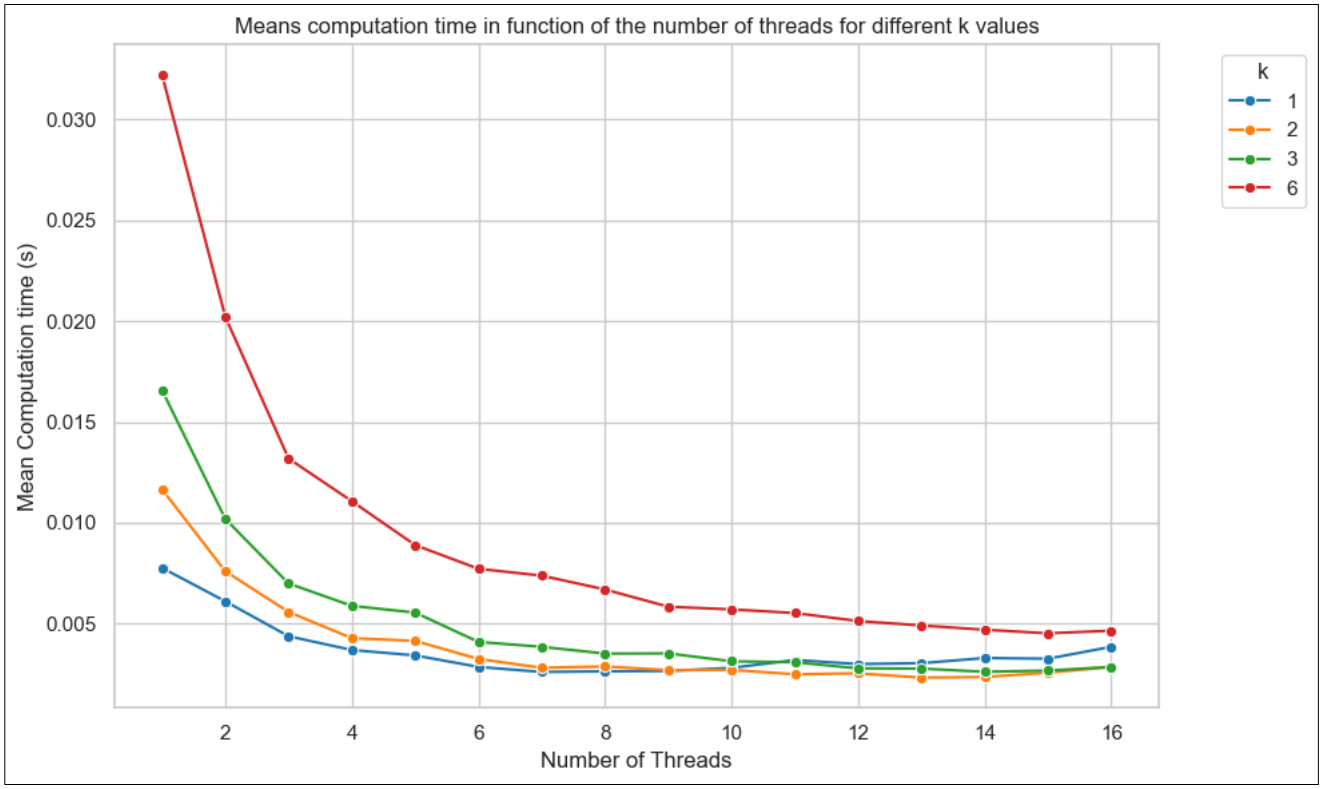


Figure 1 : Compute time evolution for k=1 and k=6



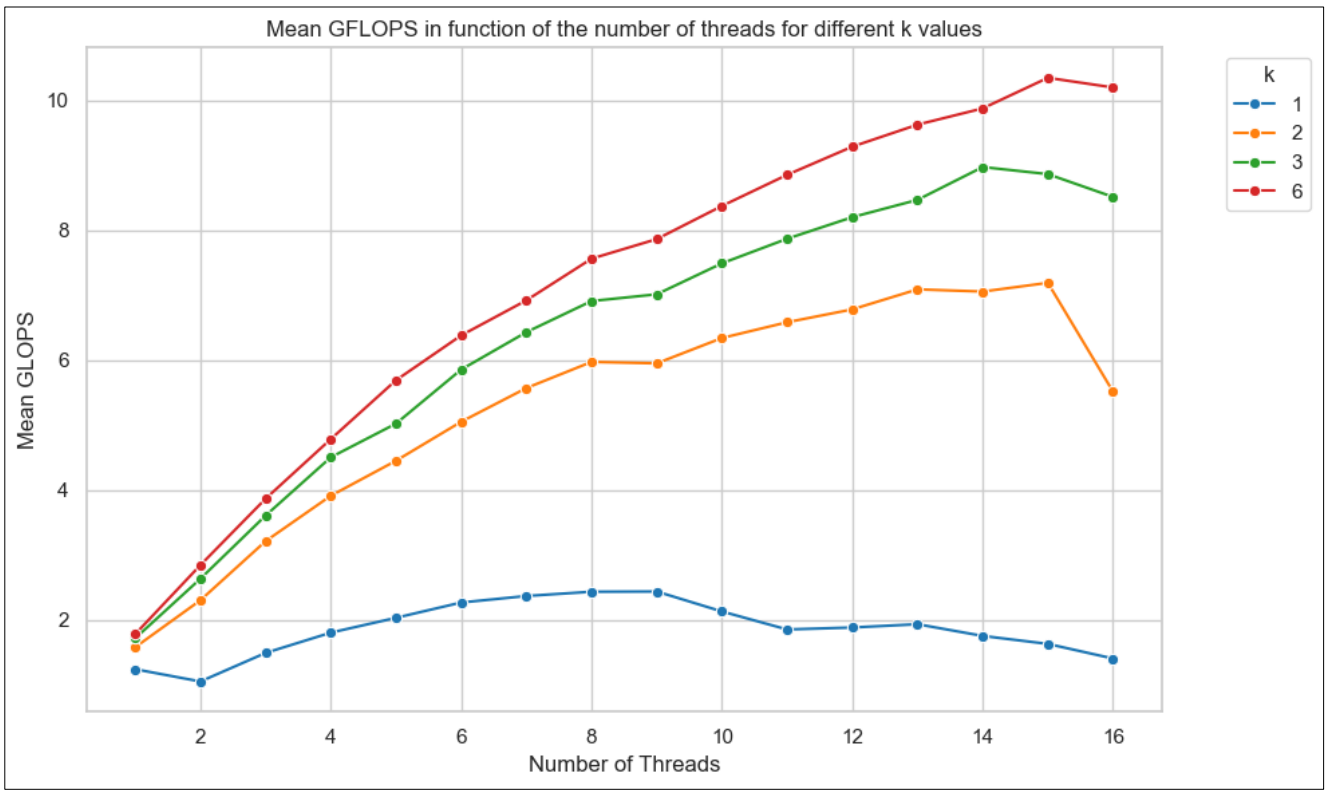


Figure 2 : Compute time and GFLOPS evolution in function of the number of threads

Compute time Performance (fig .1)

The compute time for k=1, k=2, k=3, and k=6 exhibits a clear decreasing trend as the number of threads increases. This inverse relationship is consistent across all test matrices, confirming the expected benefit of parallel execution. However, the rate of decrease in computation time is not linear. For the majority of matrices, the most significant reduction in time occurs as the thread count moves from 2 to around 8 threads. Beyond 8 threads, the reduction in computation time begins to plateau, suggesting that additional threads contribute less to performance gains. This behavior could be attributed to a number of factors, including but not limited to, memory bandwidth limitations, the overhead of managing additional threads, and the inherent parallelisability of the algorithm.

The variance in compute time is quite notable for different matrices. Certain matrices such as thermomech_TK.mtx and af23560.mtx show a higher degree of variability and a steeper drop in compute time as threads are added, which may point to these matrices being more amenable to parallel processing due to a bigger size.

GFLOPS Performance (fig. 2)

When looking at the GFLOPS performance, there is a clear upward trend as the number of threads increases, which is in line with the decreasing compute time. The rate of increase in GFLOPS performance is more pronounced for matrices with higher k values. This suggests that the parallel computation of matrix-vector products becomes more efficient as the size of the dense matrix increases, possibly due to better utilisation of the cache and reduction in overhead relative to the amount of computation performed.

The peak performance in GFLOPS is observed at different thread counts for different matrices, which indicates that the optimal number of threads for maximising GFLOPS is matrix-dependent. It is also evident that for certain matrices, the GFLOPS performance starts to decrease beyond a certain number of threads. This could be due to the same factors causing the plateau in computation time, such as synchronisation overhead and limited parallelism within the matrix itself.

CUDA Results

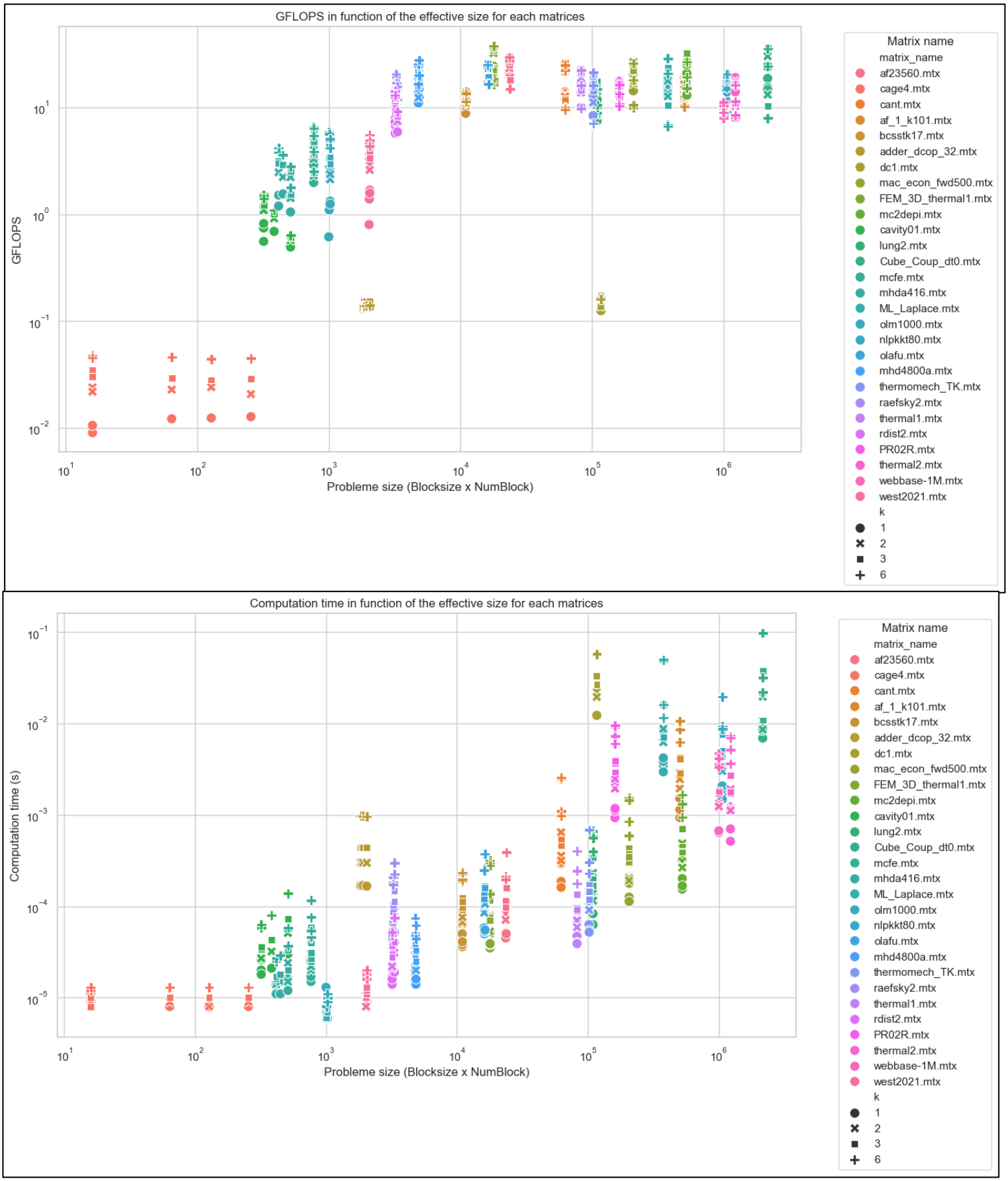


Figure 3 : GFLOPS and Computation time evolution with the problem size

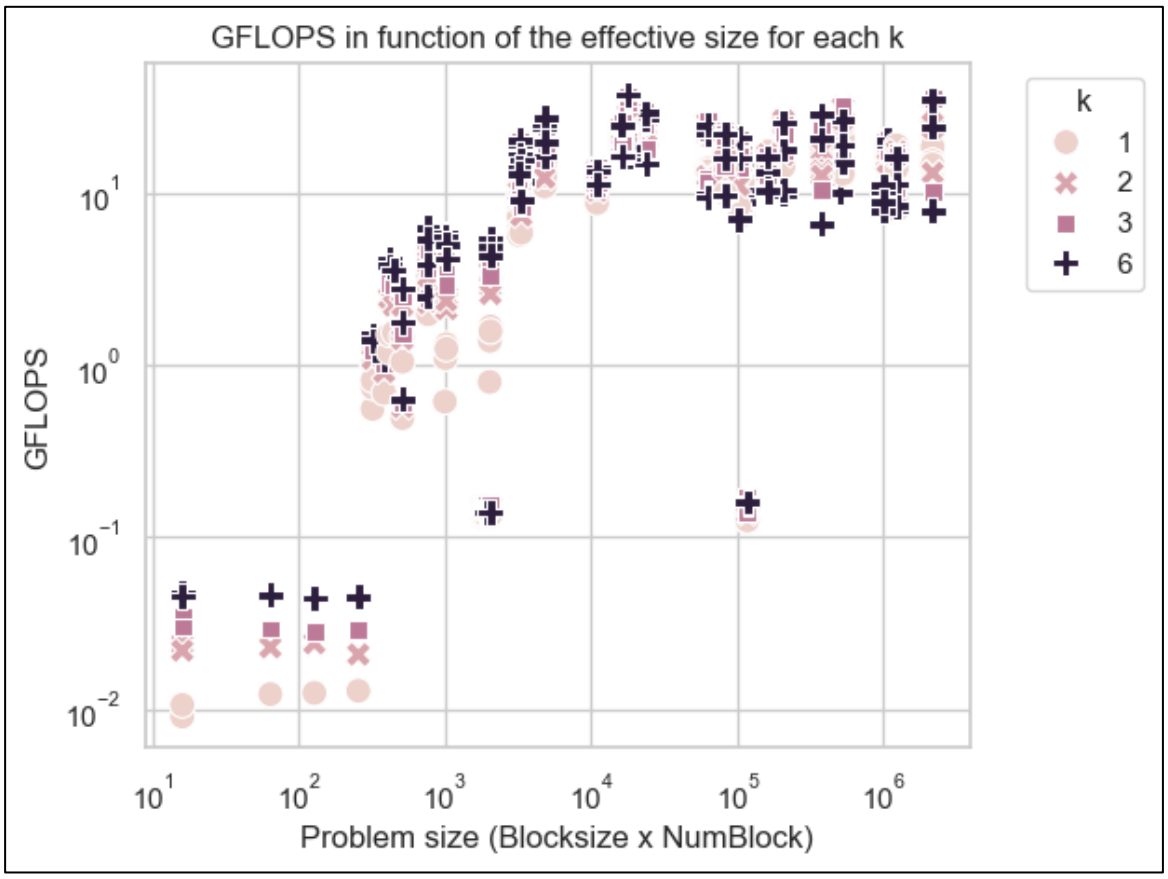


Figure 4 : GFLOPS evolution with the problem size for each k

he CUDA implementation results are analysed through the lens of GFLOPS performance and execution time, taking into account the effective size which is a product of block size and number of blocks.

GFLOPS Performance

The GFLOPS performance graph (fig. 3) indicates a varied impact of the effective size on the GFLOPS achieved across different matrices and k values. Generally, as the effective size increases, the GFLOPS performance improves, highlighting the parallel computing capabilities of the GPU. However, this trend is not uniform across all matrices, suggesting that the optimal block size and number of blocks depend on the specific characteristics of the matrices, such as sparsity pattern and dimensions.

For k=1, the GFLOPS are relatively lower (fig. 4), which may suggest that the overhead of CUDA kernel invocation and memory operations has a more significant impact when the computation per matrix element is minimal. As k increases, the GFLOPS performance also increases, which is consistent with the expectation that more extensive computation tasks

benefit more from the GPU's parallelism. The higher the k value, the more work each thread performs, leading to a better amortisation of the overhead and more efficient use of the GPU's resources.

Execution Time

The execution time graph (fig. 3) presents an inverse relationship to the GFLOPS graph, as expected. Lower execution times are observed with increasing effective sizes, especially noticeable in the mid-range of the effective size. This could be due to the better utilisation of the GPU's multiple cores, leading to faster computation. However, there is a limit to this improvement, and beyond a certain point, increasing the effective size doesn't lead to a proportional decrease in execution time. This plateau can be attributed to the saturation of the GPU's resources, where adding more blocks or increasing the block size no longer results in additional performance gains.

It is also evident that different matrices reach this plateau at different points, highlighting the importance of tailoring the CUDA configuration to the problem at hand. Some matrices with a specific sparsity pattern or size might benefit from a smaller effective size, where the parallel execution can be efficiently managed without overloading the GPU.

Comparison of OpenMP and CUDA Implementations with the Serial Version

The performance improvement of parallel computing over serial computing is quantified by the speed-up factor, which is the ratio of execution time for the serial implementation to that of the parallel implementation. The analysis of speed-up factors for both OpenMP and CUDA implementations provides insight into the efficiency and scalability of parallel processing for sparse matrix multiplication tasks.

OpenMP Speed-Up

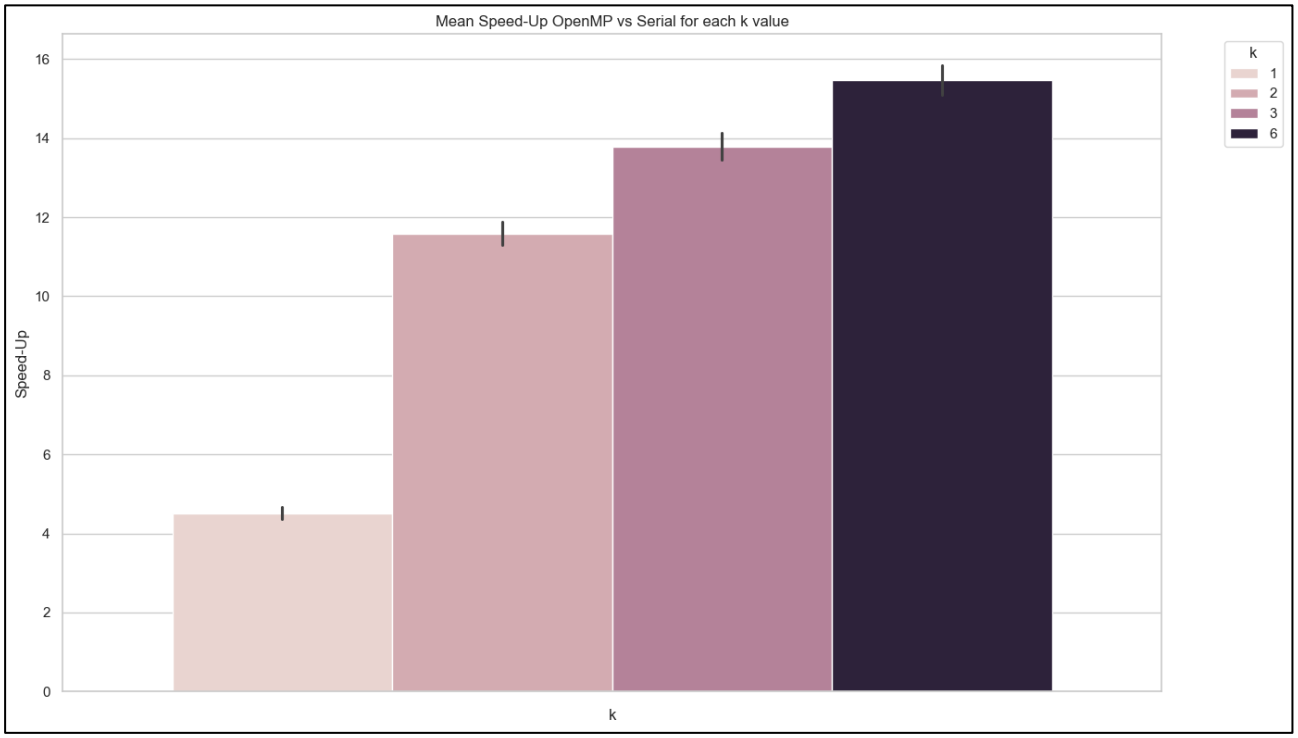


Figure 5 : Mean Speed-up OpenMP vs Serial for each k value

The first graph (fig. 5) illustrates the speed-up achieved by the OpenMP implementation over the serial version for various matrices and values of k, representing the number of columns in the matrix. The speed-up factor increases with the value of k, highlighting the effectiveness of parallel execution in scenarios where the workload per operation is higher. For k=1, the speed-up is modest, but as k increases to 2, 3, and 6, the benefits of parallel processing become more pronounced. This trend reflects the ability of OpenMP to better utilise multiple CPU cores as the amount of work increases, reducing the execution time significantly compared to the serial approach.

The error bars indicate variability in the speed-up factor, which suggests that different matrices may exhibit different levels of parallelisability due to their unique sparsity patterns and dimensions. Moreover, the optimal performance gain is not only a function of the workload but also the underlying architecture and how well it can handle concurrent operations.

CUDA Speed-Up

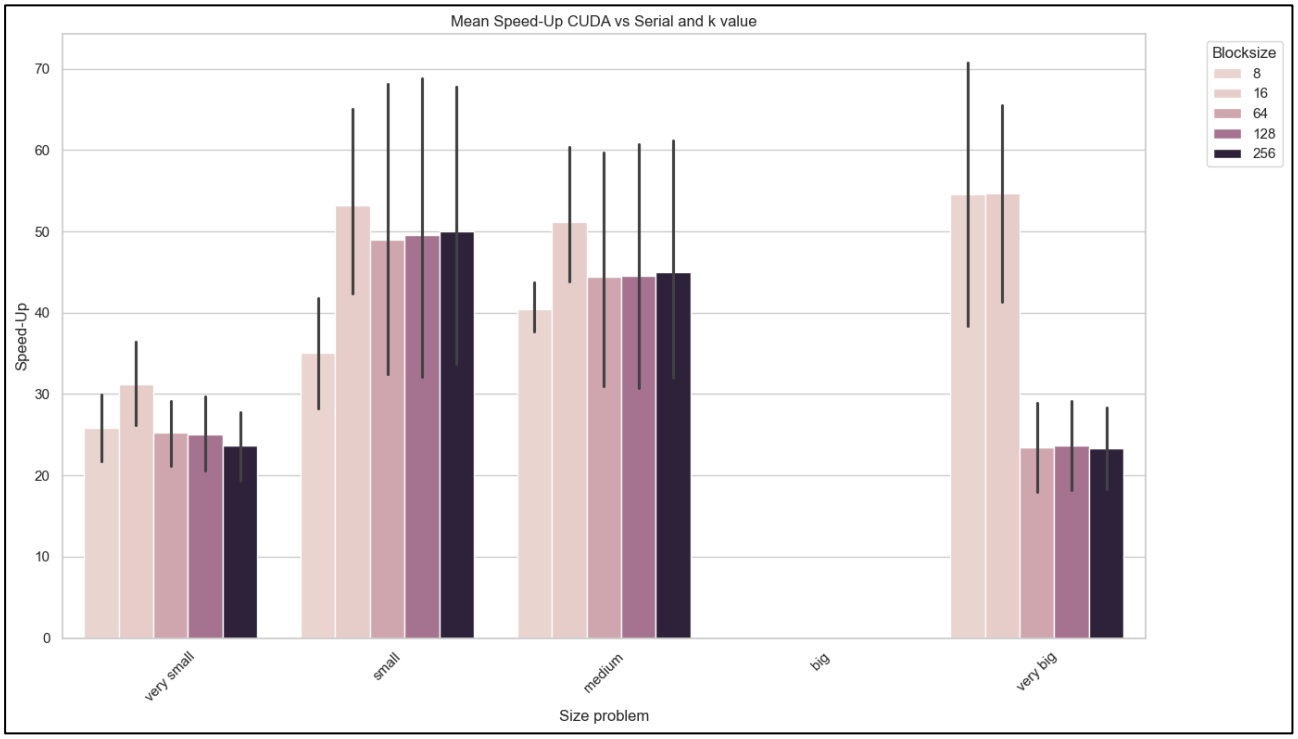


Figure 6 : Mean Speed-Up CUDA vs Serial evolution with the problem size for each k

The second graph (fig. 6) displays the speed-up factor for the CUDA implementation, contrasting various problem sizes (from "very small" to "very big") against different block sizes (8, 16, 64, 128, 256). The CUDA implementation shows a substantial speed-up over the serial version across all problem sizes, with larger problems generally seeing greater speed-up. This performance gain is a testament to the GPU's ability to handle large-scale parallel computations more efficiently than a CPU can in serial.

The choice of block size has a significant impact on the speed-up factor, with medium block sizes often achieving the highest speed-up, likely due to a balance between efficient resource utilisation and minimising overheads. Very small block sizes may underutilise the GPU, while very large ones may incur inefficiencies related to the management of threads and blocks.

IV – Conclusion

In conclusion, the project's journey through the realms of high-performance computing, from the meticulous crafting of sparse matrix multiplication algorithms to the rigorous evaluation of their performance, has yielded a wealth of insights. The endeavour's foundational goal was to propel the computational prowess of sparse matrix operations to new heights by harnessing the capabilities of serial, OpenMP, and CUDA implementations.

The findings from this exploration are illuminating. The OpenMP implementation, with its shared-memory parallelism, brought forth appreciable gains in efficiency, particularly when the workload per thread was substantial. The scalability of this method, however, was not without its limits, as evidenced by a plateau in performance improvements beyond a certain number of threads. This points to an intricate balance between the benefits of parallelism and the overheads intrinsic to managing an increasing number of threads.

Meanwhile, the CUDA implementation emerged as a formidable force, with its GPU parallelism delivering substantial speed-ups over the serial approach. The study revealed that the CUDA performance was most pronounced for larger problem sizes, where the GPU's parallelism could be fully exploited. Yet, here too, the performance was contingent upon an optimal configuration of block sizes, beyond which the returns diminished.

The comparison between the parallel approaches and the serial baseline underscored a universal truth of modern computing: the potential for acceleration is immense, yet it is deeply entwined with the nuances of both the hardware capabilities and the characteristics of the data. The serial implementation, while outpaced by its parallel counterparts, was instrumental as a benchmark, ensuring the correctness of the parallel algorithms.

Overall, the project not only advanced the understanding of sparse matrix multiplication within high-performance computing but also highlighted the intricate dance between algorithmic design, data structure, and hardware utilisation. The implications for future research and applications are profound, suggesting that the path to optimisation is not linear but requires a harmonious alignment of multiple factors.

References

[1] Sparse Matrix. Internet contributors. Wikipedia [Internet]; 2023 [cited 2024 Jan 30]. Available from https://en.wikipedia.org/wiki/Sparse_matrix

[2] The Matrix Market I/O Library. National Institute of Standards and Technology [Internet]; 2023 [cited 2024 Feb 22]. Available from: http://math.nist.gov/MatrixMarket/

[3] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 5.0. OpenMP [Internet]; 2023 [cited 2024 Feb 22]. Available from: https://www.openmp.org/specifications/

[4] NVIDIA CUDA Toolkit Documentation. NVIDIA Corporation [Internet]; 2023 [cited 2024 Feb 22]. Available from: https://docs.nvidia.com/cuda/