Alexis Balayre

Small Scale Parallel Programming Assignment

School of Aerospace, Transport and Manufacturing
Computational Software of Techniques Engineering

MSc
Academic Year: 2023 - 2024

Supervisor: Dr Salvatore Filippone
26th February 2024

Abstract

This paper explores the effectiveness of different parallelization strategies for multiplying sparse matrices by fat vectors, focusing on the use of MPI in High Performance Computing (HPC). It compares the performance of sequential and parallel methods, including row-by-row, column-by-column, and non-zero element approaches. The results highlight the implications of each strategy on execution time and efficiency, while considering the environmental impact of HPC, suggesting avenues towards more sustainable computing systems.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

High-Performance Computing (HPC) is a branch of computing that uses supercomputers and server clusters to solve complex, computationally intensive problems. Unlike a personal computer with a single processor, an HPC system is made up of many processors working in parallel, considerably increasing processing capacity. This enables scientists and engineers to carry out detailed numerical simulations, such as forecasting the weather or solving structural engineering problems.

Cranfield University has two HPC systems: CRESCENT2 and DELTA. However, this report will focus exclusively on CRESCENT2, which is an HPC cluster designed to provide computing power for teaching and research. CRESCENT 2 nodes are equipped with Intel Xeon E5 2620 processors, and each node contains two 16-core processors and 16 gigabytes of RAM.

The aim of this report is to explore distributed memory parallel programming strategies for optimising the performance of sparse matrix multiplication by a fat vector, a common operation in numerical linear algebra.

Chapter 2

Methodology

2.1 Problem Statement

Consider a sparse matrix A of dimensions $m \times n$ and a fat vector X of dimensions $n \times k$. The objective is to perform the multiplication $A \times X$, yielding a result that is of dimensions $m \times k$.

The matrix A is defined as:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \tag{2.1}$$

where most elements of A are zeros.

The vector X is defined as:

$$X = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1k} \\ x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nk} \end{pmatrix} \tag{2.2}$$

2.2 Data Structures

2.2.1 Sparse Matrix

Two data structures were used to represent sparse matrices: Compressed Sparse Row (CSR) and ELLPACK formats. Both formats are designed to store and manipulate sparse matrices efficiently, reducing the storage space and computational load associated with zeros in the matrix.

2.2.1.1 Compressed Sparse Row (CSR) Format

The CSR format represents sparse matrices efficiently by storing only the non-zero elements and their positions. This format uses three arrays:

- **values**: An array holding all the non-zero elements of the matrix, stored sequentially as they appear in the matrix from top to bottom and left to right.
- **colIndices**: An array storing the column indices of each non-zero element in the *values* array, indicating the exact column position of each element.
- **rowPtr**: An array where each entry marks the starting point in the *values* and *colIndices* arrays for the non-zero elements of a specific row, facilitating direct access to each row's data.

2.2.1.2 ELLPACK

The ELLPACK format optimises the storage and computation for matrices with a relatively uniform distribution of non-zero elements per row. It employs two 2D arrays for this purpose:

- **values**: A 2D array where each row corresponds to a row in the original sparse matrix, containing the non-zero values. Rows are padded with zeros to equalize the length across all rows, determined by `maxNonZerosPerRow`.
- **colIndices**: A 2D array parallel to `values`, storing the column indices for each non-zero value in the matrix. It uses padding (typically with an invalid index such as -1) to match the structure of `values`.
- **maxNonZerosPerRow**: Specifies the fixed number of elements in each row of `values` and `colIndices`, determined by the row with the maximum number of non-zero elements.
- **numRows** and **numCols**: Indicate the dimensions of the matrix, specifically the total number of rows and columns.

ELLPACK format is particularly advantageous for parallel computations on GPUs due to its consistent row length, which enables efficient memory access patterns and simplifies parallelization strategies.

2.2.2 Fat Vector

The structure `FatVector` is designed to represent dense vectors or low-dimensional dense matrices:

- **values** : Contains the values of the dense vector or matrix.
- **numRows** and **numCols** : Indicate the dimensions of the vector or matrix, making it possible to represent both column vectors and matrices with several columns.

2.3 Sparse Matrix - Fat Vector Multiplication

2.3.1 Sequential Algorithm using CSR Format

Let A be a sparse matrix of size $m \times n$ with z non-zero elements, stored in CSR format, and X be a fat vector of size $n \times k$. The sequential algorithm for multiplying A by X is implemented in Appendix **??**.

2.3.1.1 Algorithm Flow

Algorithm 1 Sparse Matrix-Dense Vector Multiplication (CRS)

Require: A is an $m \times n$ sparse matrix in CRS format
Require: X is an $n \times k$ fat vector
Ensure: Y is an $m \times k$ fat vector, result of $A \times X$
  for $i = 0$ to $A.numRows - 1$ **do**
    for $j = A.rowPtr[i]$ to $A.rowPtr[i+1] - 1$ **do**
      $colIndex \leftarrow A.colIndices[j]$
      $value \leftarrow A.values[j]$
      for $k = 0$ to $X.numCols - 1$ **do**
        $yIndex \leftarrow i \times X.numCols + k$
        $xIndex \leftarrow colIndex \times X.numCols + k$
        $Y.values[yIndex] \leftarrow Y.values[yIndex] + value \times X.values[xIndex]$
      end for
    end for
  end for

The algorithmic flow can be more explicitly detailed by:

1. **Initialisation**: Prepare the result vector with the same number of rows as the sparse matrix and initialise all elements to zero.

2. **Iteration Over Rows**: For each row in the sparse matrix, use the rowPtr array to find the starting and ending indices of non-zero elements in that row.

3. **Iteration Over Non-Zero Elements**: For each non-zero element identified in the previous step, retrieve the column index and value from colIndices and values arrays, respectively.

4. **Multiplication and Accumulation**: Use the column index to identify the corresponding element(s) in the dense vector. Multiply each non-zero element by the corresponding element in the dense vector and accumulate the product in the appropriate position of the result vector. This step is repeated for each column in the dense vector if it has more than one column.

5. **Result Compilation**: After iterating through all rows and their non-zero elements, the result vector contains the product of the sparse matrix and the dense vector.

2.3.1.2 Temporal Complexity Analysis

Given a sparse matrix A of size $m \times n$ with NZ non-zero elements and a fat vector X of size $n \times k$, the serial algorithm for multiplying $A \times X$ iterates through each non-zero element of the matrix A to compute the product.

The algorithm performs two operations (a multiplication and an addition) for each non-zero element with respect to each column of X, resulting in a total of $2 \times NZ \times k$ operations.

Hence, the time complexity of the sparse matrix-fat vector multiplication algorithm can be expressed as:

$$T(n) = O(NZ \times k) \tag{2.3}$$

The CRS format is more efficient in terms of computation when the distribution of non-zero elements is irregular, as it only iterates over these elements. However, it may not be as efficient for parallel processing due to the irregular memory access patterns.

2.3.2 Sequential Algorithm using ELLPACK Format

2.3.2.1 Algorithm Flow

Algorithm 2 Sparse Matrix-Dense Vector Multiplication (ELLPACK)

Require: A is an $m \times n$ sparse matrix in ELLPACK format
Require: X is an $n \times k$ fat vector
Ensure: Y is an $m \times k$ fat vector, result of $A \times X$

```
for i = 0 to A.numRows − 1 do
    for j = 0 to A.maxNonZerosPerRow − 1 do
        colIndex ← A.colIndices[i × A.maxNonZerosPerRow + j]
        value ← A.values[i × A.maxNonZerosPerRow + j]
        if colIndex ≠ −1 then
            for k = 0 to X.numCols − 1 do
                yIndex ← i × X.numCols + k
                xIndex ← colIndex × X.numCols + k
                Y.values[yIndex] ← Y.values[yIndex] + value × X.values[xIndex]
            end for
        end if
    end for
end for
```

The algorithmic flow can be more explicitly detailed by:

1. **Initialisation**: Similarly, prepare the result vector with an appropriate size and initialise all values to zero.

2. **Iteration Over Rows**: Iterate over each row of the sparse matrix, given that the ELLPACK format stores a fixed number of elements (equal to the maximum number of non-zero elements in any row) for every row.

3. **Iteration Over Elements**: For each element in a row (up to the maximum number of non-zero elements per row), check if the column index is valid (not a padding indicator, such as -1).

4. **Multiplication and Accumulation**: For each valid non-zero element, perform multiplication with the corresponding element(s) in the dense vector, similar to the CRS algorithm. This involves using the column index to locate the correct element in the dense vector and accumulating the product in the result vector.

5. **Handling Padding**: Ignore any padding indicators (e.g., column index -1) during multiplication to ensure that they do not affect the result.

6. **Result Compilation**: After processing all rows, the result vector is fully populated with the product of the sparse matrix in ELLPACK format and the dense vector.

2.3.2.2 Temporal Complexity Analysis

Given a sparse matrix A of size $m \times n$ with $NZ_{\max}$ as the maximum number of non-zero elements in any row and a fat vector X of size $n \times k$. The sequential algorithm for multiplying $A \times X$ iterates iterates over each row and each column index/value pair within the row, resulting in a total of $2 \times m \times NZ_{\max} \times k$ operations.

Hence, the time complexity of the sparse matrix-fat vector multiplication algorithm can be expressed as:

$$T(n) = O(m \times NZ_{\max} \times k) \tag{2.4}$$

The ELLPACK format can lead to faster execution times in architectures that favour regular memory access patterns, despite potentially higher computational complexity due to padding, particularly when the sparse matrix is relatively dense or the maximum number of non-zero elements per row is close to the average number over all rows. However, its efficiency decreases with increasing filling (i.e. when the difference between the maximum and average number of non-zero elements per row is large).

2.4 Parallel Algorithms

2.4.1 OpenMP (Open Multi-Processing)

OpenMP provides a powerful framework for parallelising computational tasks in shared-memory architectures. When applied to sparse matrix-vector multiplication, OpenMP enables significant performance enhancements for both CSR and ELLPACK formats by distributing the computation across multiple CPU threads.

2.4.1.1 Workflow and Optimisation Strategy

The application of OpenMP in sparse matrix-vector multiplication encompasses a series of steps designed to efficiently parallelise the computation and optimise resource utilisation:

1. **Memory Allocation and Data Initialization:** Initially, the sparse matrix and dense vector are allocated in memory. OpenMP does not require explicit memory allocation on a separate device but operates directly on data in the process's address space.

2. **Kernel Execution:**

   - *For CSR Format:* An OpenMP parallel for loop iterates over the rows of the matrix, where each thread processes a subset of rows, computing dot products between the non-zero elements and the corresponding entries of the dense vector.
   - *For ELLPACK Format:* Similarly, an OpenMP parallel for loop is employed, with threads iterating over rows. The fixed-length rows of the ELLPACK format potentially offer more regular memory access patterns, which can be advantageous for performance.

   Both implementations utilise `#pragma omp parallel for` directives, optionally with dynamic scheduling to manage workload distribution among threads.

3. **Performance Measurement:** The execution time for the parallel region is captured using `omp_get_wtime()`, allowing for the calculation of performance metrics such as execution time and GFLOPS (Giga Floating-Point Operations Per Second).

4. **Data Retrieval and Cleanup:** Upon completion of the parallel computation, the result vector is available for further processing. Unlike CUDA, there is no need for explicit data transfer between host and device memory spaces.

5. **Resource Management:** OpenMP abstracts much of the resource management, simplifying the parallelization process. However, developers should still consider the optimal number of threads and scheduling strategies to maximise performance.

2.4.1.2 Expected Performance Improvements

Leveraging OpenMP for sparse matrix-vector multiplication offers potential for substantial performance gains, attributed to:

- **Parallel Processing:** Utilising multiple CPU cores to perform computations in parallel significantly reduces overall execution time.
- **Efficient Workload Distribution:** The ability to dynamically schedule work among threads can lead to more balanced computation, especially for matrices with uneven distributions of non-zero elements.
- **Reduced Overhead:** Compared to CUDA, OpenMP operates within the existing memory space of the application, eliminating the overhead associated with data transfer between host and device.

2.4.1.3 Challenges and Considerations

Despite the advantages, several considerations must be addressed to optimise performance:

- **Thread Overhead:** The benefits of adding more threads diminish beyond a certain point, where the overhead of thread management can outweigh performance gains.
- **Memory Access Patterns:** Especially relevant for the ELLPACK format, ensuring efficient memory access patterns can enhance performance.
- **Optimal Use of Resources:** Balancing the computational load across available CPU cores and selecting the appropriate chunk size for dynamic scheduling are key factors in optimizing performance.

2.4.2 CUDA

CUDA (Compute Unified Device Architecture) is a parallel programming model developed by NVIDIA. It enables graphics processing units (GPUs) to be used for general-purpose computing outside the traditional graphics context. With CUDA, developers can exploit the massively parallel computing power of NVIDIA GPUs for computationally intensive computing more efficiently than with traditional CPU approaches.

2.4.2.1 Workflow and Kernel Design

The CUDA-based implementation follows a structured workflow, incorporating specialised kernels to exploit parallel processing capabilities of GPUs:

1. **Memory Allocation and Data Transfer:** Initially, necessary memory for matrix data (values, column indices, and row pointers or ELLPACK structures), the dense vector, and the result vector is allocated on the GPU. Data is then transferred from the host to these allocated spaces.

2. **Kernel Execution:**

   - *CSR Kernel:* Processes the sparse matrix in CSR format. Each thread calculates a single element of the result vector by iterating over non-zero elements of a particular row, multiplying each by the corresponding vector element, and accumulating the result.
   - *ELLPACK Kernel:* Similarly, processes the matrix in ELLPACK format. Threads are assigned to matrix rows, with each thread iterating over the fixed-size row data, performing multiplication and accumulation.

   Both kernels use atomic operations to safely add results to the output vector in cases of potential write conflicts.

3. **Performance Measurement:** CUDA events are used to record the start and stop times of kernel execution, allowing for precise measurement of computation time and subsequent performance analysis.

4. **Data Retrieval:** After kernel execution, the resulting vector is transferred back to the host for further processing or analysis.

5. **Resource Cleanup:** Finally, allocated GPU memory is freed, and CUDA events are destroyed to clean up resources.

2.4.2.2 Expected Performance Improvements

CUDA parallelization offers significant speedups for sparse matrix-vector multiplication by leveraging the massive parallelism of GPU cores. Performance gains are realised through:

- **Fine-grained Parallelism:** Each non-zero element or row of the matrix can be processed in parallel, significantly reducing computation time.

- **Memory Bandwidth Utilisation:** Efficient use of memory bandwidth by coalescing memory accesses and minimising global memory transactions.
- **Load Balancing:** Dynamic assignment of work to threads helps mitigate performance degradation due to imbalanced non-zero element distribution across rows.

2.4.2.3 Challenges and Considerations

While CUDA accelerates sparse matrix operations, developers must consider factors such as the sparsity pattern of the matrix, the optimal configuration of CUDA threads and blocks, and the overhead of memory transfers between host and device. Proper tuning and optimization strategies, including choosing appropriate block sizes and utilizing shared memory, are crucial for maximizing performance on GPUs.

2.5 Implementation and Testing

In order to implement all the algorithms and test them, 2 main programs were created:

- `runCuda.cpp` to test the CUDA implementations.
- `runOpenMP.cpp` to test the OpenMP implementations.

2.5.1 Workflow

Both programs follow the same workflow:

1. Reading matrices from files in Matrix Market format.
2. Conversion of matrices from CRS format to ELLPACK format if necessary.
3. Generation of dense vectors with random values for multiplication.
4. Perform matrix-vector multiplications using both sequential and parallel algorithms.
5. Validate the results of the parallel algorithms against the sequential ones.
6. Measuring and displaying performance.

2.5.2 Comprehensive Testing Strategy

To validate and benchmark the efficiency of parallel implementations using OpenMP and CUDA for sparse matrix-vector multiplication, a systematic testing strategy is employed. This strategy encompasses a range of test parameters, correctness verification, and performance evaluation methods.

2.5.2.1 Test Parameters Configuration

2.5.2.1.1 OpenMP Parameters:

- **Matrix Sparsity:** Varied densities of hollow matrices are tested to simulate real-world scenarios and understand performance across different sparsity levels.
- **Vector Sizes:** Dense vector sizes are varied (1, 2, 3, 6) to assess the impact on performance, reflecting different use cases.
- **Thread Count:** Tests are conducted with 1 to 16 threads to identify the optimal concurrency level that maximizes performance.
- **Chunk Sizes:** To fine-tune dynamic workload distribution, chunk sizes are varied (2, 4, 8, 16, 32, 64, 128, 256), allowing for in-depth analysis of the parallel loop scheduling efficiency.

2.5.2.1.2 CUDA Parameters:

- **Matrix Sparsity:** Similar to OpenMP, different densities of hollow matrices are evaluated to gauge CUDA's effectiveness across varying sparsity patterns.
- **Vector Sizes (`ks`):** A range of dense vector sizes are tested to examine CUDA's adaptability to different data scales.
- **X Block Size:** CUDA block sizes along the X dimension are varied (8, 16, 32, 64) to optimize thread block configuration for the GPU architecture.
- **Y Block Size:** Similarly, Y block sizes (1, 2, 4, 8, 16) are adjusted to explore the impact on parallel execution efficiency.

The sparse matrices used for testing are summarised in Table 2.1.

2.5.2.2 Correctness Verification

To ensure the accuracy of both OpenMP and CUDA parallel implementations, a two-step verification process is adopted:

1. **Result Comparison:** The outcomes of the sequential (baseline) and parallel implementations are compared using the `areMatricesEqual` function. This function evaluates if the two matrices are identical within a predefined tolerance level, ensuring computational integrity.
2. **Tolerance Threshold:** A small tolerance is allowed for floating-point operations to account for numerical precision variances inherent in parallel computations.

2.5.2.3 Performance Evaluation

Performance testing is structured to provide a comprehensive understanding of the efficiency gains from parallelization:

1. **Execution Time Measurement:** The time taken by each implementation to complete the matrix-vector multiplication is precisely measured, using built-in timing functionalities like `omp_get_wtime()` for OpenMP and CUDA events for GPU measurements.
2. **GFLOPS Calculation:** Performance is quantitatively compared in terms of Giga Floating-Point Operations Per Second (GFLOPS), offering a normalized metric to evaluate computational speed.
3. **Repetitions for Accuracy:** Each test configuration is executed 20 times. This repetition ensures statistical significance, allowing for the calculation of average execution times and minimizing the impact of outliers.

Through this comprehensive testing approach, the parallel implementations' correctness and performance are meticulously evaluated, ensuring reliability and efficiency of the sparse matrix-vector multiplication algorithms.

Matrix Name	m	NZ	AVG NZR	MAX NZR	Symmetric
cavity10	2597	76367	29.4	62	False
PR02R	161070	8185136	50.8	92	False
nlpkkt80	1062400	28704672	27.0	28	True
Cube_Coup_dt0	2164760	127206144	58.8	68	True
roadNet-PA	1090920	3083796	2.8	9	True
ML_Laplace	377002	27689972	73.4	74	False
bcsstk17	10974	428650	39.1	150	True
mhda416	416	8562	20.6	33	False
af_1_k101	503625	17550675	34.8	35	True
thermal1	82654	574458	7.0	11	True
thermomech_TK	102158	711558	7.0	10	True
cage4	9	49	5.4	6	False
cant	62451	4007383	64.2	78	True
dc1	116835	766396	6.6	114190	False
raefsky2	3242	294276	90.8	108	False
rdist2	3198	56934	17.8	61	False
mcfe	765	24382	31.9	81	False
olm1000	1000	3996	4.0	6	False
lung2	109460	492564	4.5	8	False
webbase-1M	1000005	3105536	3.1	4700	False
mhd4800a	4800	102252	21.3	33	False
west2021	2021	7353	3.6	12	False
thermal2	1228045	8580313	7.0	11	True
adder_dcop_32	1813	11246	6.2	100	False
mac_econ_fwd500	206500	1273389	6.2	44	False
FEM_3D_thermal1	17880	430740	24.1	27	False
amazon0302	262111	1234877	4.7	5	False
cop20k_A	121192	2624331	21.7	81	True
olafu	16146	1015156	62.9	89	True
af23560	23560	484256	20.6	21	False

Table 2.1: Summary of sparse matrices

Chapter 3

Results and Discussion

3.1 Results

3.1.1 Sequential Algorithms

The following table shows the performance comparison of the sequential algorithms using the CRS and ELLPACK formats.

Table 3.1: Performance Comparison of CRS and ELLPACK Sequential Algorithms

Matrix	**CRS**	**ELLPACK**	**Best Structure**
Cube_Coup_dt0	6.313540	2.659870	CRS
FEM_3D_thermal1	5.939650	2.543850	CRS
ML_Laplace	6.629070	2.775640	CRS
PR02R	6.577900	2.755250	CRS
adder_dcop_32	4.481260	2.360730	CRS
af23560	5.804440	2.509250	CRS
af_1_k101	6.139610	2.608660	CRS
amazon0302	2.344020	0.968039	CRS
bcsstk17	6.256790	2.654560	CRS
cage4	0.871628	0.849791	CRS
cant	6.419710	2.687030	CRS
cavity10	6.032170	2.625250	CRS
cop20k_A	4.771770	2.155300	CRS
dc1	4.686860	2.349130	CRS
lung2	4.634780	2.319800	CRS
mac_econ_fwd500	3.684580	1.851230	CRS
mcfe	6.069120	2.634280	CRS
mhd4800a	5.474340	2.549000	CRS
mhda416	5.524470	2.516310	CRS
nlpkkt80	5.932140	2.567120	CRS
olafu	6.481010	2.724440	CRS
olm1000	4.775320	2.192310	CRS

Matrix	CRS	ELLPACK	Best Structure
raefsky2	6.556030	2.767780	CRS
rdist2	5.762990	2.465300	CRS
roadNet-PA	2.721200	1.240050	CRS
thermal1	4.599780	2.228430	CRS
thermal2	3.206760	1.437750	CRS
thermomech_TK	3.558870	1.592880	CRS
webbase-1M	3.750440	1.888800	CRS
west2021	4.184400	2.181630	CRS

A few key observations can be made:

- **CRS Dominance:** For the majority of matrices tested, the CRS format systematically outperforms the ELLPACK format. This is evident for matrices such as Cube_Coup_dt0, FEM_3D_thermal1, and ML_Laplace, where CRS not only handles large matrices efficiently but also matrices with high average and maximum numbers of non-zero elements per row.
- **Efficiency in Dense Matrices:** The CRS format appears to be particularly efficient at handling high-density matrices (higher AVG NZR and MAX NZR), which could be attributed to its storage efficiency and the way it streamlines the multiplication process for rows with varying lengths of non-zero elements.

3.1.2 OpenMP

3.1.2.1 Chunk Size Analysis

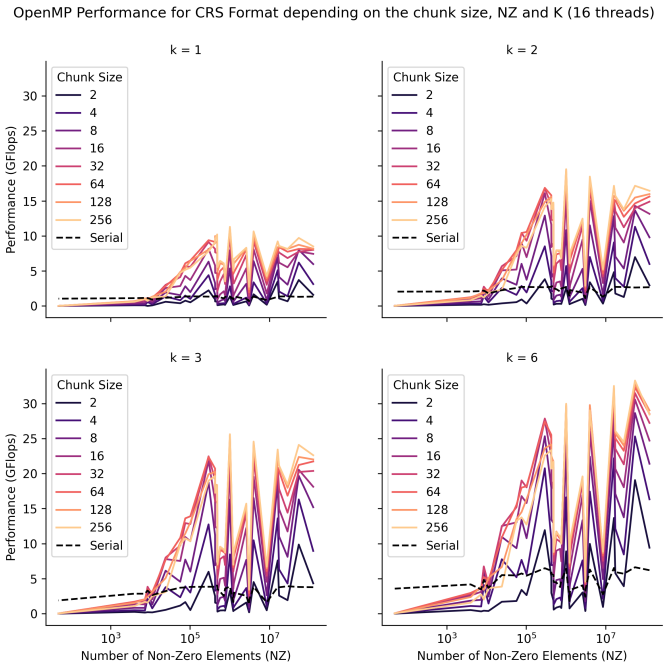

Figure 3.1: OpenMP Performance for CRS Format depending on Chunk Size

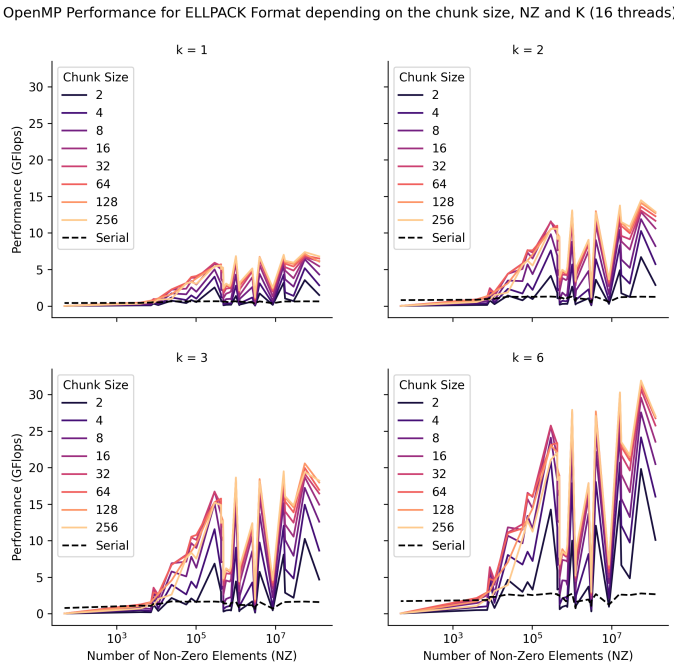

Figure 3.2: OpenMP Performance for ELLPACK Format depending on Chunk Size

As shown in Figures 3.1 and 3.2, the performance of the OpenMP parallel algorithms is influenced by the chunk size. For lower sparsity matrices, the optimal chunk size is 64 for the CRS format and 32 for the ELLPACK format. For higher sparsity matrices, the optimal chunk size is 128 or 256 for both formats.

3.1.2.2 Thread Count Analysis

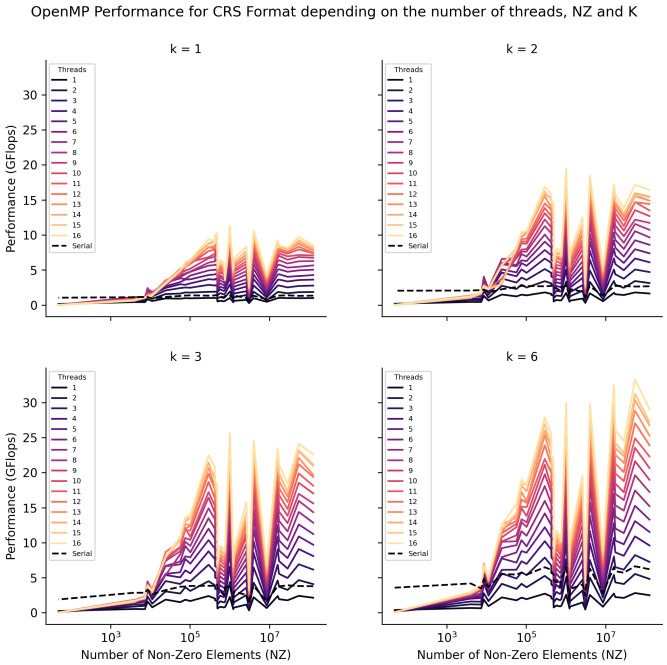

Figure 3.3: OpenMP Performance for CRS Format depending on Threads

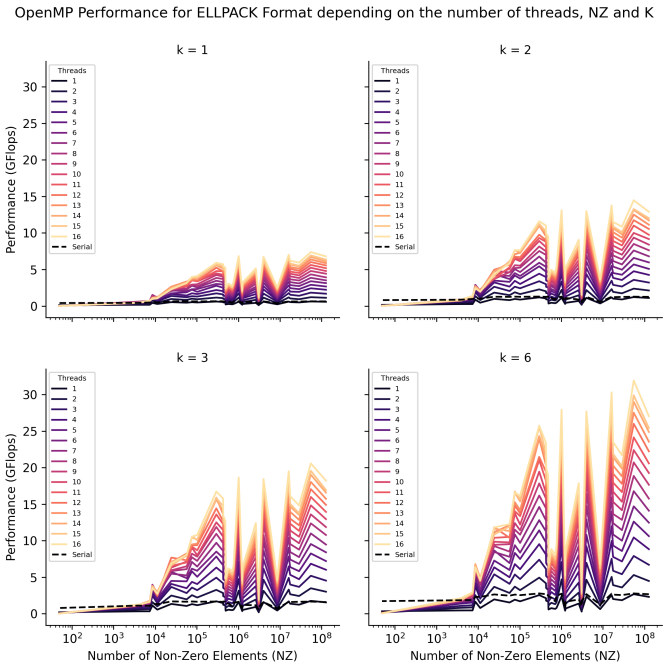

Figure 3.4: OpenMP Performance for ELLPACK Format depending on Threads

As shown in Figures 3.3 and 3.4, the performance of the OpenMP parallel algorithms is influenced by the number of threads. As more threads are used, the performance im-

proves up to a certain point, after which the performance starts to stagnate due to the overhead of managing the threads.

3.1.2.3 Performance Comparison

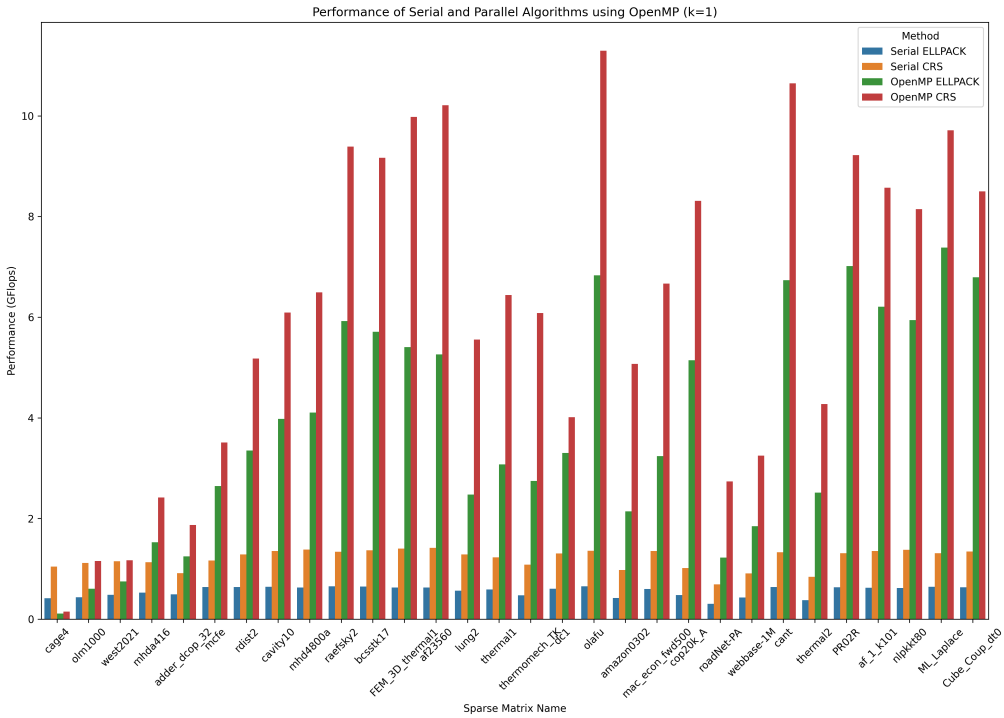

Figure 3.5: Performance of Serial and Parallel Algorithms using OpenMP for $k = 1$

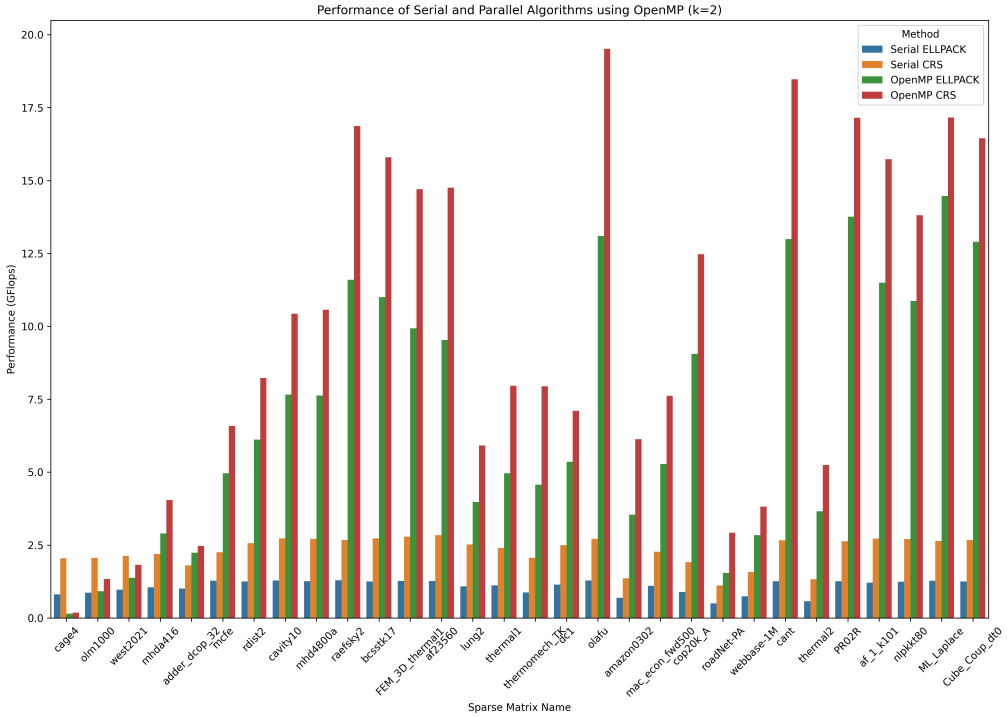

Figure 3.6: Performance of Serial and Parallel Algorithms using OpenMP for $k = 2$

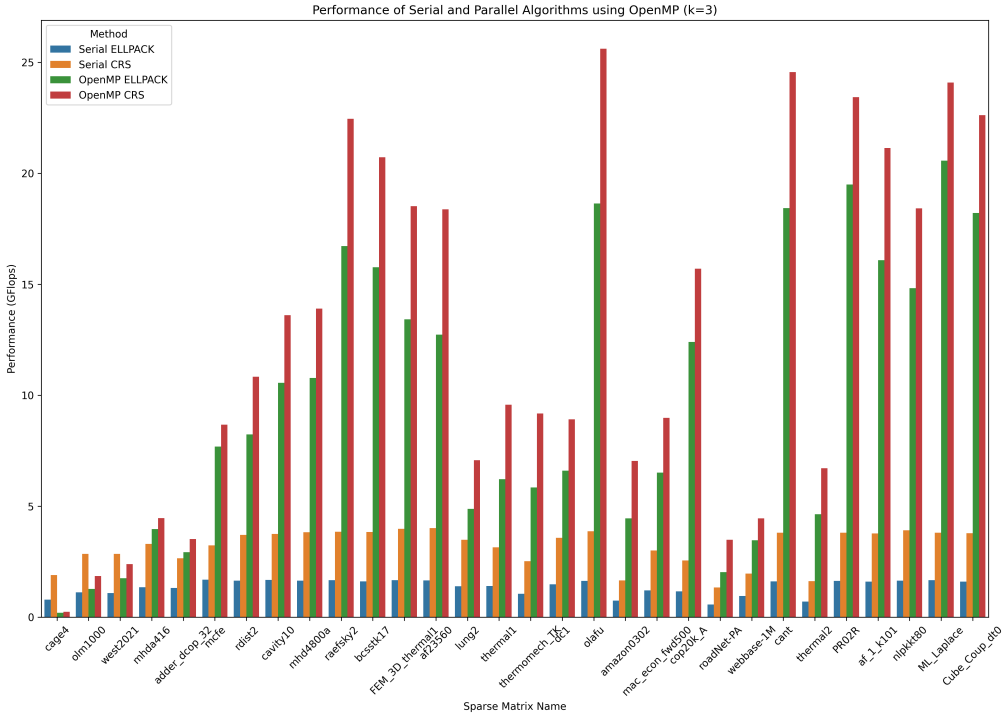

Figure 3.7: Performance of Serial and Parallel Algorithms using OpenMP for $k = 3$

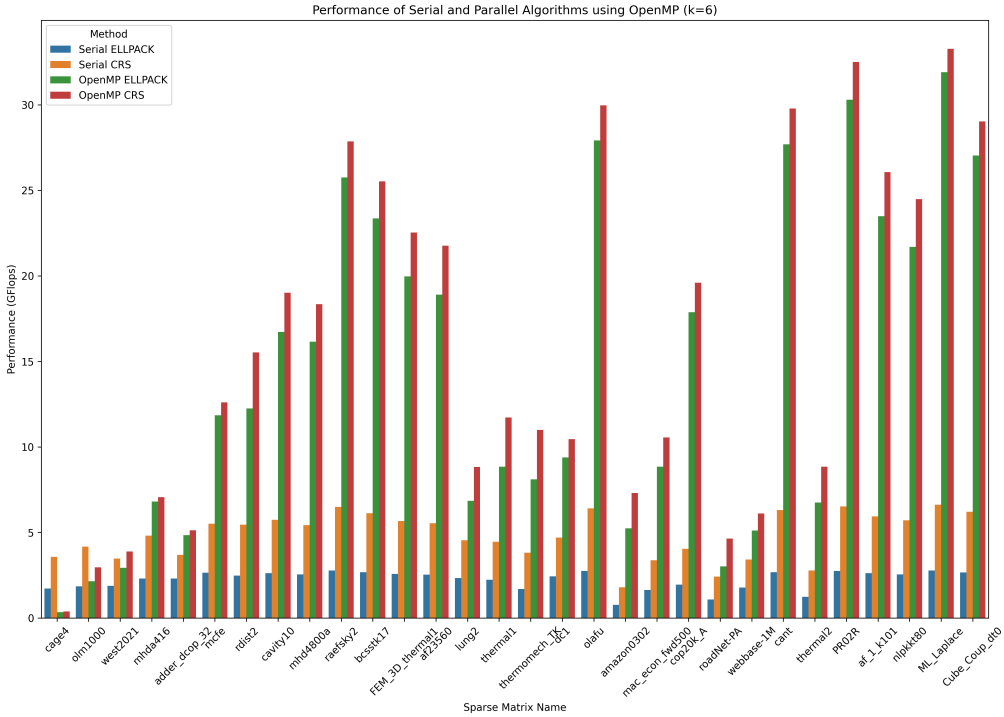

Figure 3.8: Performance of Serial and Parallel Algorithms using OpenMP for $k = 6$

Table 3.2: CRS vs ELLPACK using OpenMP

Matrix	**CRS**	**ELLPACK**	**Best Structure**	**Speedup**
Cube_Coup_dt0	29.0243	27.0415	CRS	4.597152
FEM_3D_thermal1	22.5277	19.9711	CRS	3.792766
ML_Laplace	33.2766	31.9121	CRS	5.019799
PR02R	32.5086	30.2968	CRS	4.942094
adder_dcop_32	5.12204	4.83417	CRS	1.142991
af23560	21.7606	18.9078	CRS	3.748958
af_1_k101	26.067	23.4946	CRS	4.245709
amazon0302	7.29883	5.23835	CRS	3.113809
bcsstk17	25.5293	23.3597	CRS	4.080255
cage4	0.385164	0.334653	CRS	0.441890
cant	29.7794	27.6956	CRS	4.638745
cavity10	19.0148	16.7258	CRS	3.152232
cop20k_A	19.5918	17.8729	CRS	4.105772
dc1	10.4524	9.39085	CRS	2.230150
lung2	8.832	6.85328	CRS	1.905592
mac_econ_fwd500	10.5515	8.84879	CRS	2.863691
mcfe	12.6097	11.8456	CRS	2.077682
mhd4800a	18.3501	16.1582	CRS	3.352021
mhda416	7.06099	6.80262	CRS	1.278130

Matrix	CRS	ELLPACK	Best Structure	Speedup
nlpkkt80	24.4853	21.694	CRS	4.127566
olafu	29.966	27.9111	CRS	4.623662
olm1000	2.95895	2.1558	CRS	0.619634
raefsky2	27.8548	25.7504	CRS	4.248730
rdist2	15.5283	12.2527	CRS	2.694487
roadNet-PA	4.63584	3.01227	CRS	1.703601
thermal1	11.7254	8.83925	CRS	2.549122
thermal2	8.84723	6.75174	CRS	2.758931
thermomech_TK	10.9908	8.10205	CRS	3.088284
webbase-1M	6.11469	5.10783	CRS	1.630393
west2021	3.89256	2.92762	CRS	0.930255

The performance evaluation of the CRS and ELLPACK formats, when parallelized with OpenMP, shows a preference for CRS in a majority of cases. This trend, indicated by a frequent superiority of the CRS format, highlights its suitability for OpenMP's parallelization capabilities, probably due to more optimal memory management and task distribution between threads.

The observed speed-up factor varies significantly between matrices, with particularly remarkable speed-ups for matrices such as *Cube_Coup_dt0* and *FEM_3D_thermal1*, highlighting the potential for improving performance via OpenMP's parallel optimisation.

Special cases such as *cage4* and *olm1000* show lower speed-ups, revealing the limits of parallelization for certain matrix structures. The influence of matrix density and size is also notable, with high densities and large dense vectors favouring the CRS format more, illustrating its effectiveness in processing large workloads under OpenMP.

3.1.3 CUDA

3.1.3.1 X Block Size Analysis

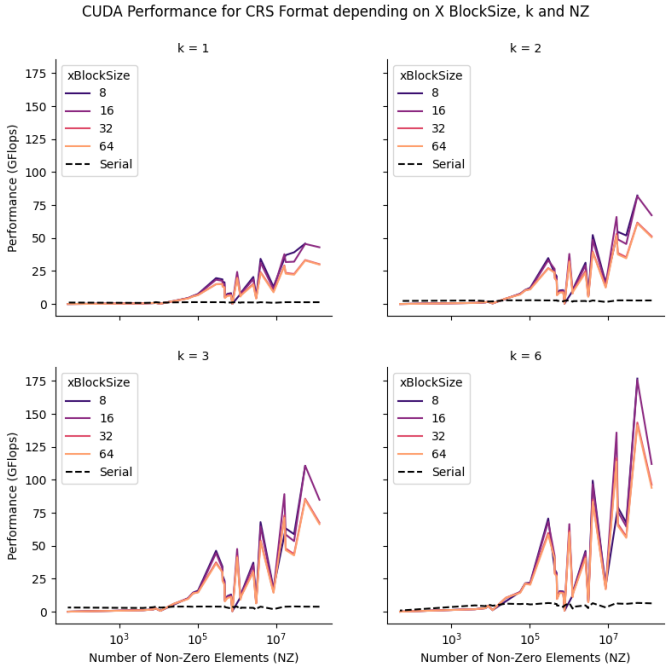

Figure 3.9: CUDA Performance for CRS Format depending on X BlockSize

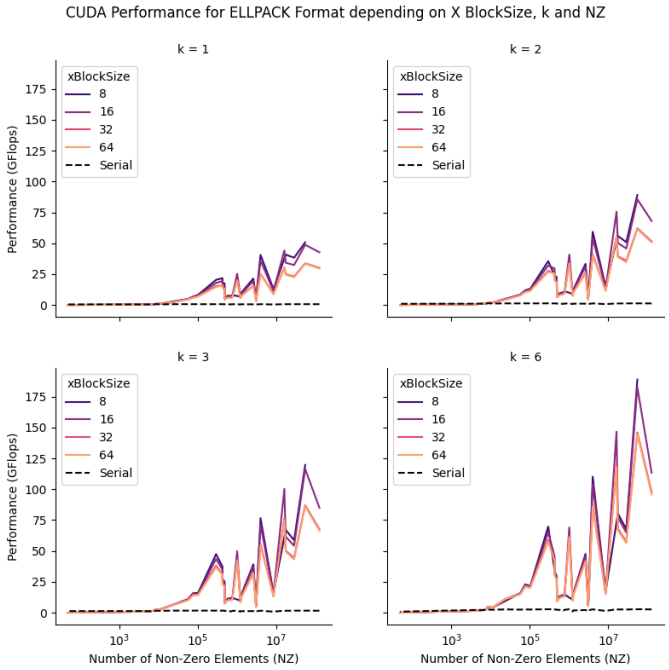

Figure 3.10: CUDA Performance for ELLPACK Format depending on X BlockSize

3.1.3.2 Y Block Size Analysis

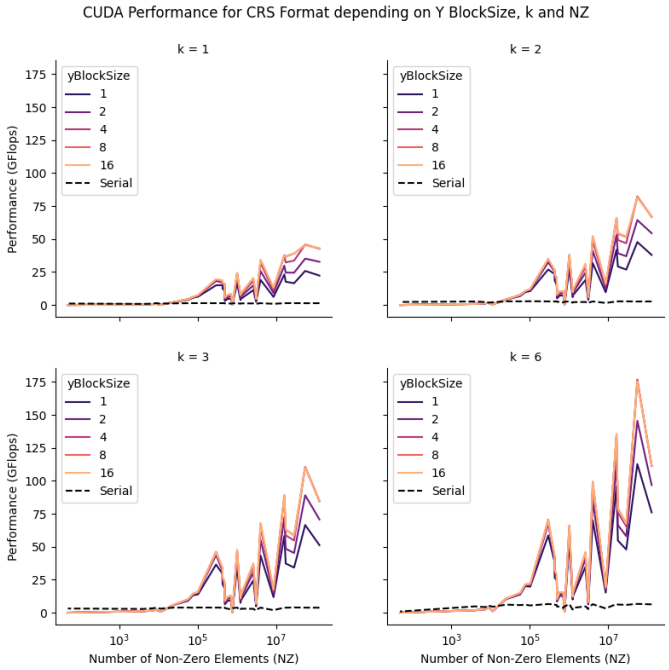

Figure 3.11: CUDA Performance for CRS Format depending on Y BlockSize

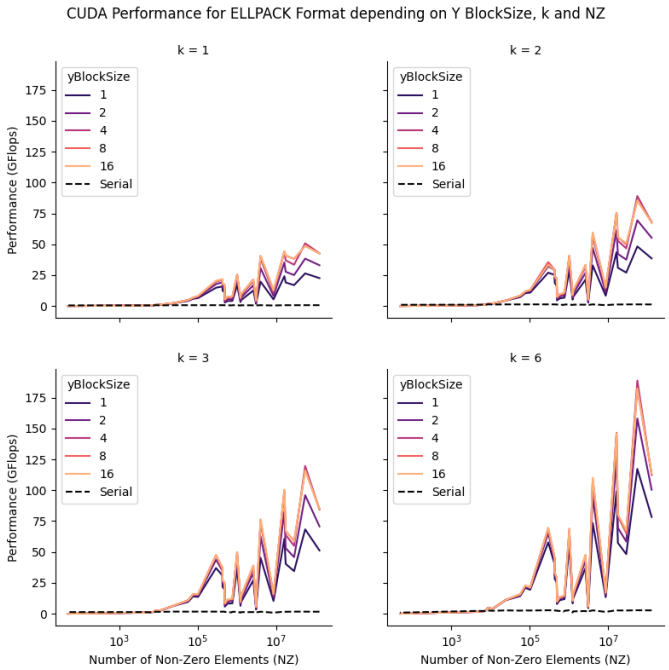

Figure 3.12: CUDA Performance for ELLPACK Format depending on Y BlockSize

3.1.3.3 Performance Comparison

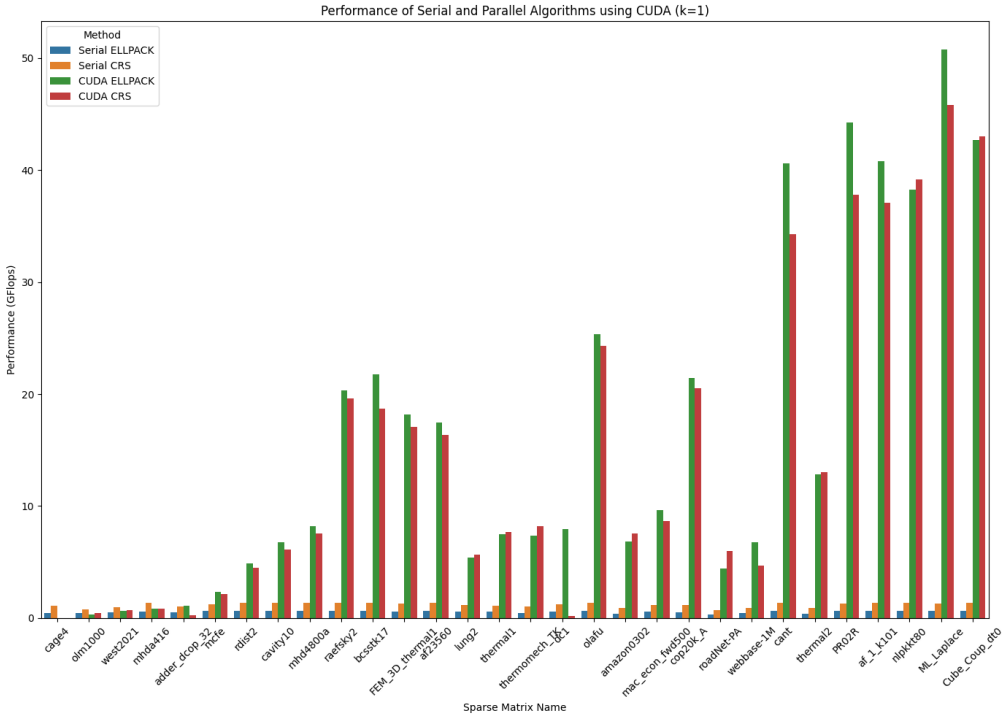

Figure 3.13: Performance of Serial and Parallel Algorithms using CUDA for $k = 1$

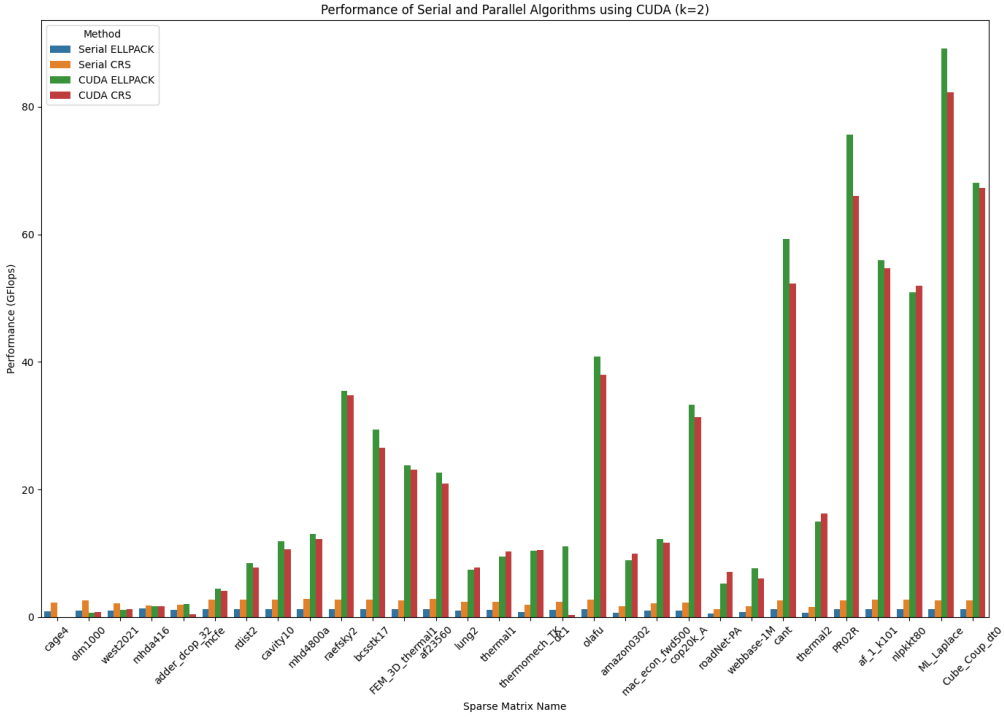

Figure 3.14: Performance of Serial and Parallel Algorithms using CUDA for $k = 2$

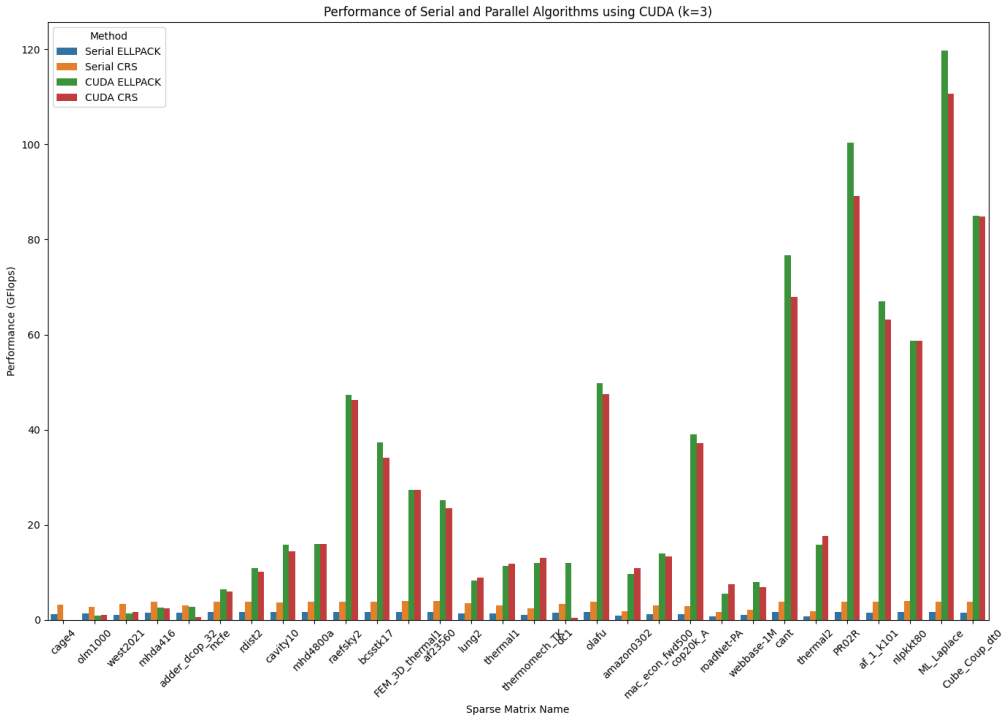

Figure 3.15: Performance of Serial and Parallel Algorithms using CUDA for $k = 3$

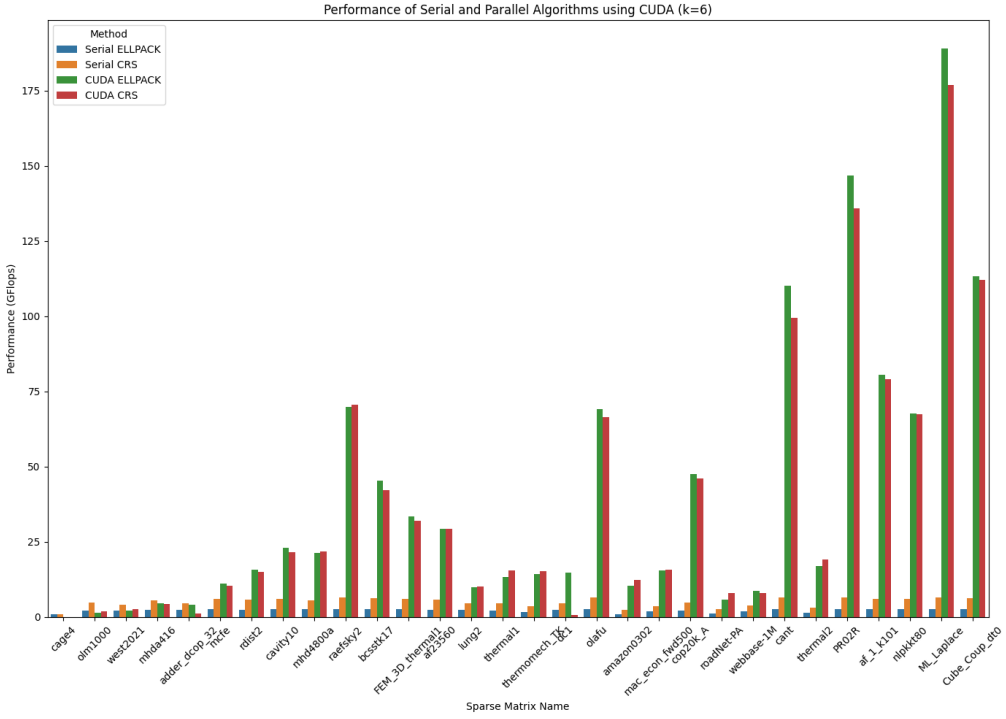

Figure 3.16: Performance of Serial and Parallel Algorithms using CUDA for $k = 6$

Table 3.3: CRS vs ELLPACK using CUDA

Matrix	CRS	ELLPACK	Best Structure	Speedup
amazon0302	12.3120	10.4822	CRS	5.252515
cage4	0.033586	0.031649	CRS	0.038533
lung2	10.2714	9.81548	CRS	2.216157
mac_econ_fwd500	15.6609	15.4014	CRS	4.250389
mhd4800a	21.9362	21.4274	CRS	4.007095
olm1000	1.89504	1.48348	CRS	0.396840
raefsky2	70.6624	69.7615	CRS	10.778230
roadNet-PA	8.07317	5.84257	CRS	2.966768
thermal1	15.5753	13.318	CRS	3.386097
thermal2	19.2541	17.0523	CRS	6.004222
thermomech_TK	15.3241	14.2953	CRS	4.305889
west2021	2.66929	2.26954	CRS	0.637915
Cube_Coup_dt0	111.963	113.348	ELLPACK	17.953161
FEM_3D_thermal1	32.0555	33.4671	ELLPACK	5.634524
ML_Laplace	176.774	188.886	ELLPACK	28.493590
PR02R	135.828	146.651	ELLPACK	22.294501
adder_dcop_32	1.20824	4.12316	ELLPACK	0.920089
af23560	29.2923	29.4083	ELLPACK	5.066518
af_1_k101	79.1365	80.4176	ELLPACK	13.098161
bcsstk17	42.2982	45.3715	ELLPACK	7.251562
cant	99.3766	110.127	ELLPACK	17.154513
cavity10	21.571	23.0165	ELLPACK	3.815625
cop20k_A	46.062	47.6264	ELLPACK	9.980867
dc1	0.733577	14.7532	ELLPACK	3.147779
mcfe	10.5186	11.2408	ELLPACK	1.852130
mhda416	4.46372	4.6875	ELLPACK	0.848498
nlpkkt80	67.4344	67.5926	ELLPACK	11.394303
olafu	66.3819	69.052	ELLPACK	10.654512
rdist2	15.0779	15.7503	ELLPACK	2.733008
webbase-1M	8.04237	8.68042	ELLPACK	2.314507

The results of parallelizing the CRS and ELLPACK algorithms with CUDA show a strong preference for the CRS format on certain matrices (such as *amazon0302*, *lung2*, and *thermal1*), attributable to better sparsity management and memory access patterns optimized for GPUs. In contrast, ELLPACK shows superior performance for matrices with regular structures, such as *Cube_Coup_dt0* and *ML_Laplace*, due to more uniform memory access.

The variability of the speed-up factor between matrices highlights the importance of the specific structure of the matrix in the relative efficiency of CRS and ELLPACK under CUDA. In particular, symmetric and dense arrays tend to favour ELLPACK, highlighting the significant role of density and symmetry in the performance of the formats.

3.1.4 Overall Performance Comparison

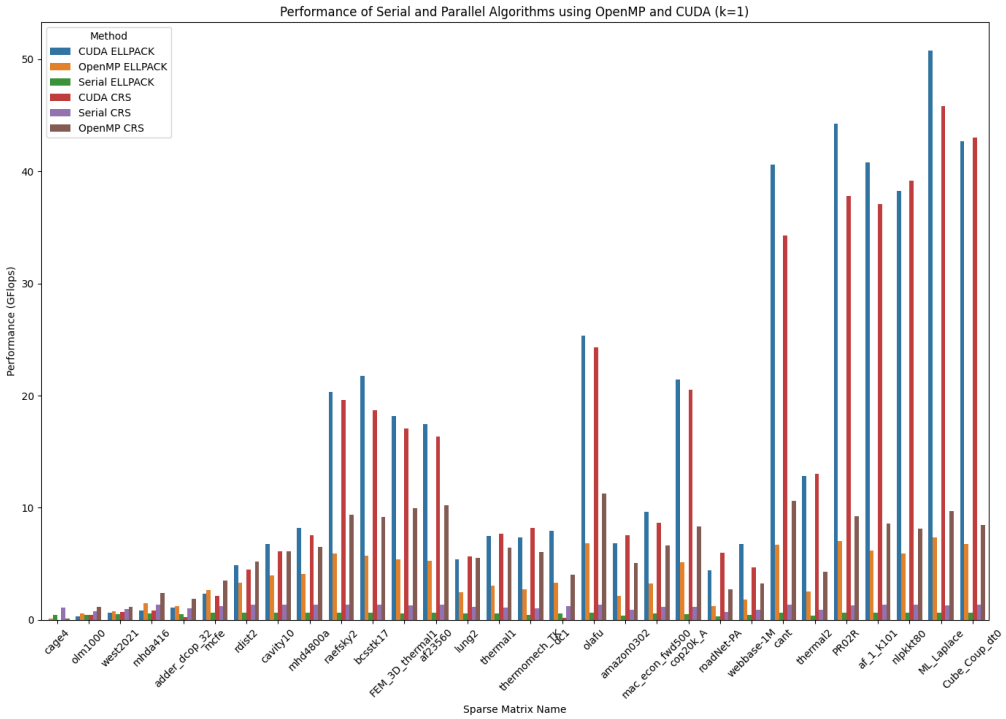

Figure 3.17: OpenMP vs CUDA Performance for $k = 1$

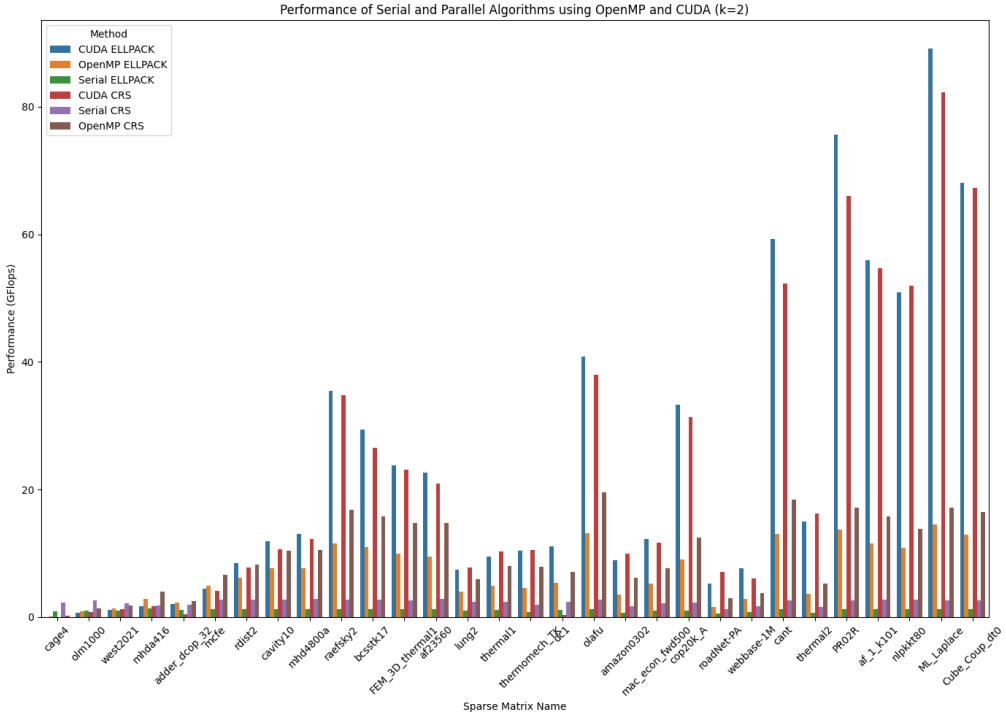

Figure 3.18: OpenMP vs CUDA Performance for $k = 2$

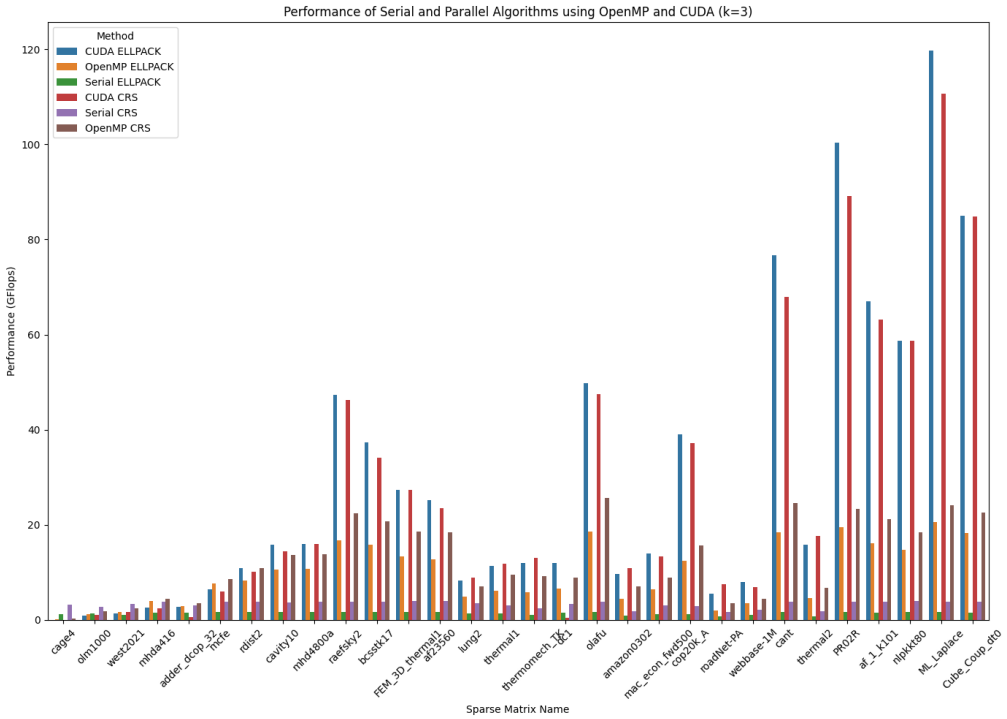

Figure 3.19: OpenMP vs CUDA Performance for $k = 3$

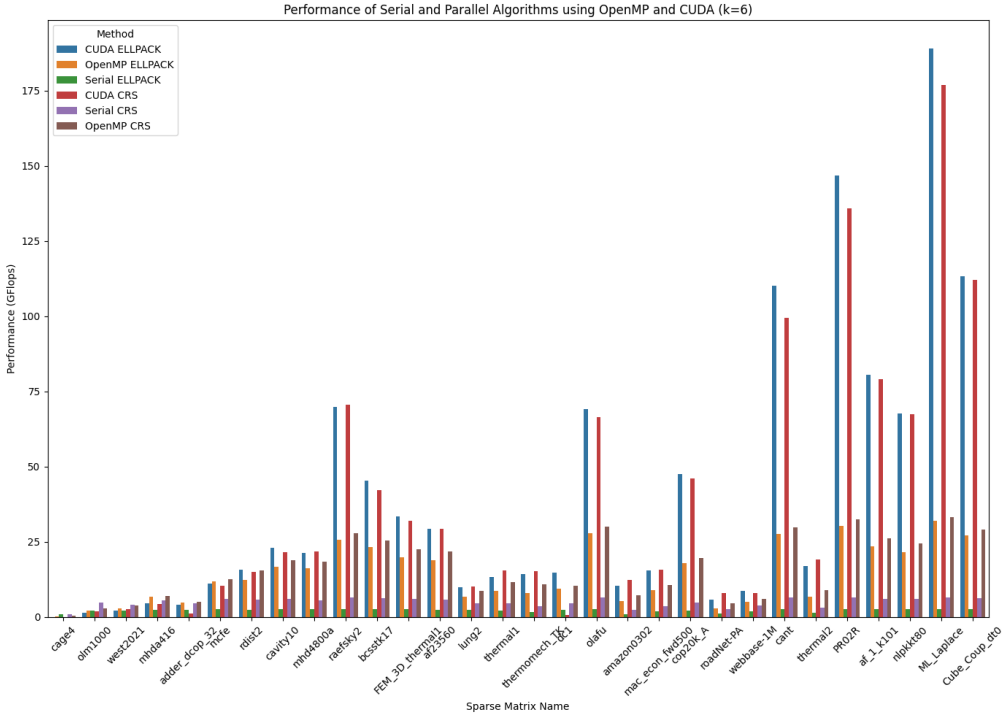

Figure 3.20: OpenMP vs CUDA Performance for $k = 6$

Table 3.4: Overall Performance Comparison

Matrix	**Best Performance**	**Best Structure**	**Best Method**	**Speedup**
amazon0302	12.3120	CRS	CUDA	5.252515
lung2	10.2714	CRS	CUDA	2.216157
mac_econ_fwd500	15.6609	CRS	CUDA	4.250389
mhd4800a	21.9362	CRS	CUDA	4.007095
raefsky2	70.6624	CRS	CUDA	10.778230
roadNet-PA	8.07317	CRS	CUDA	2.966768
thermal1	15.5753	CRS	CUDA	3.386097
thermal2	19.2541	CRS	CUDA	6.004222
thermomech_TK	15.3241	CRS	CUDA	4.305889
Cube_Coup_dt0	113.348	ELLPACK	CUDA	17.953161
FEM_3D_thermal1	33.4671	ELLPACK	CUDA	5.634524
ML_Laplace	188.886	ELLPACK	CUDA	28.493590
PR02R	146.651	ELLPACK	CUDA	22.294501
af23560	29.4083	ELLPACK	CUDA	5.066518
af_1_k101	80.4176	ELLPACK	CUDA	13.098161
bcsstk17	45.3715	ELLPACK	CUDA	7.251562
cant	110.127	ELLPACK	CUDA	17.154513
cavity10	23.0165	ELLPACK	CUDA	3.815625
cop20k_A	47.6264	ELLPACK	CUDA	9.980867
dc1	14.7532	ELLPACK	CUDA	3.147779
nlpkkt80	67.5926	ELLPACK	CUDA	11.394303
olafu	69.0520	ELLPACK	CUDA	10.654512
rdist2	15.7503	ELLPACK	CUDA	2.733008
webbase-1M	8.68042	ELLPACK	CUDA	2.314507
adder_dcop_32	5.12204	CRS	OpenMP	1.142991
mcfe	12.6097	CRS	OpenMP	2.077682
mhda416	7.06099	CRS	OpenMP	1.278130
cage4	0.871628	CRS	Serial	1.000000
olm1000	4.77532	CRS	Serial	1.000000
west2021	4.1844	CRS	Serial	1.000000

Chapter 4

Conclusion

In conclusion, this report has meticulously analysed various parallelization strategies for fat matrix vector multiplication, based on the MPI and PETSc frameworks, and through extensive experimentation and comparison, highlights the importance of choosing an appropriate parallelization approach based on matrix characteristics and computational resources. The results highlight the trade-offs between computational and communication overheads, emphasising the need for a balanced distribution of workloads to maximise efficiency. As expected the PETSc library outperforms the custom implementations in terms of execution time but not in terms of performance, as the matrices stay distributed even after the multiplication. The custom implementations are more efficient to reconstruct the final result from the gathered vector. In addition, the discussion of the environmental impact of HPC practices, with a nod to sustainable computing, reflects a broader perspective on the implications of computational research. Future directions could explore adaptive parallelization techniques, further optimizing specific models and matrix sizes, possibly integrating AI for dynamic algorithm selection based on real-time performance metrics. This study not only contributes to the field of HPC by providing insight into efficient algorithmic implementations, but also paves the way for more energy-efficient HPC systems, in line with global sustainable development goals.

References

Appendix A

Documentation

Appendix A.A Project tree

```
Source Code/
    scripts/
        batch_test.sh
        get_csv_all.sh
        get_csv_debug.sh
        get_csv_specific.sh
        mpi.sub
    MatrixDefinitions.h
    SparseMatrixFatVectorMultiply.h
    SparseMatrixFatVectorMultiply.cpp
    SparseMatrixFatVectorMultiplyRowWise.h
    SparseMatrixFatVectorMultiplyRowWise.cpp
    SparseMatrixFatVectorMultiplyColumnWise.h
    SparseMatrixFatVectorMultiplyColumnWise.cpp
    SparseMatrixFatVectorMultiplyNonZeroElement.h
    SparseMatrixFatVectorMultiplyNonZeroElement.cpp
    utils.h
    utils.cpp
    main.cpp
results/
    fat_vector_dim/
        <sparse_matrix>_<k>_<metric>.png
    matrix_dim/
        <sparse_matrix>_<k>_<metric>.png
```

Appendix A.B Getting Started

To run the program, follow these steps:

1. Install the required libraries: mpi & petsc
2. Compile the main program using the following command:

```
            mmpicxx -o <executable_name> -I${PETSC_DIR}/include -I${
PETSC_DIR}/${PETSC_ARCH}/include -L${PETSC_DIR}/${PETSC_ARCH}/lib -lpetsc
SparseMatrixFatVectorMultiply.cpp  main_verify.cpp utils.cpp
SparseMatrixFatVectorMultiplyColumnWise.cpp
SparseMatrixFatVectorMultiplyNonZeroElement.cpp
SparseMatrixFatVectorMultiplyRowWise.cpp
```

3. Run the program using the following command:

```
            mpirun -np <number_of_processes> <executable_name> <k> <
sparse_matrix_file_pathw>
```

Appendix A.C Methods Overview

A.C.1 Utils.h

A.C.1.1 ConvertPETScMatToFatVector

Description: Converts a PETSc matrix to a FatVector stucture.

Parameters:

- `Mat C` - PETSc matrix to be converted.

Returns: `FatVector` - Fat vector representation of the PETSc matrix.

A.C.1.2 areMatricesEqual

Description: Compares two matrices for equality within a specified tolerance.

Parameters:

- `FatVector &mat1` - First matrix.
- `FatVector &mat2` - Second matrix.
- `double tolerance` - Tolerance for comparison.

Returns: `bool` - True if matrices are equal within the tolerance, false otherwise.

A.C.1.3 readMatrixMarketFile

Description: Reads a matrix from a Matrix Market file into a sparse matrix format.

Parameters:

- `std::string &filename` - Name of the Matrix Market file.

Returns: `SparseMatrix` - Sparse matrix read from the file.

A.C.1.4 generateLargeFatVector

Description: Generates a random Fat Vector with specified dimensions.

Parameters:

- `int n` - Number of rows.
- `int k` - Number of columns.

Returns: `FatVector` - Generated fat vector.

A.C.1.5 serialize and deserialize

Description: Serializes and deserializes a FatVector to and from a flat array, respectively.

Parameters for serialize:

- `FatVector &fatVec` - fat vector to serialize.

Returns: `std::vector<double>` - Flat array containing the serialized data.

Parameters for deserialize:

- `std::vector<double> &flat` - Flat array to deserialize.
- `int rows` - Number of rows in the fat vector.
- `int cols` - Number of columns in the fat vector.

Returns: `FatVector` - Deserialized fat vector.

A.C.2 SparseMatrixFatVectorMultiply.h

A.C.2.1 sparseMatrixFatVectorMultiply

Description: Executes the multiplication using a sequential algorithm.

Parameters:

- `SparseMatrix &sparseMatrix` - The sparse matrix.
- `FatVector &fatVector` - The Fat Vector.
- `int vecCols` - Number of columns in the Fat Vector.

Returns: `FatVector` - Result of the multiplication.

A.C.3 SparseMatrixFatVectorMultiplyRowWise.h

A.C.3.1 sparseMatrixFatVectorMultiplyRowWise

Description: Multiplies a sparse matrix with a Fat Vector using row-wise distribution.

Parameters:

- `SparseMatrix &sparseMatrix` - The sparse matrix.
- `FatVector &fatVector` - The Fat Vector.
- `int vecCols` - Number of columns in the Fat Vector.

Returns: `FatVector` - Result of the multiplication.

A.C.4 SparseMatrixFatVectorMultiplyColumnWise.h

A.C.4.1 sparseMatrixFatVectorMultiplyColumnWise

Description: Executes the multiplication using column-wise parallel algorithm.

Parameters:

- `SparseMatrix &sparseMatrix` - The sparse matrix.
- `FatVector &fatVector` - The Fat Vector.
- `int vecCols` - Number of columns in the Fat Vector.

Returns: `FatVector` - Result of the multiplication.

A.C.5 SparseMatrixFatVectorMultiplyNonZeroElement.h

A.C.5.1 sparseMatrixFatVectorMultiplyNonZeroElement

Description: Executes the multiplication using non-zero element parallel algorithm.

Parameters:

- `SparseMatrix &sparseMatrix` - The sparse matrix.
- `FatVector &fatVector` - The Fat Vector.
- `int vecCols` - Number of columns in the Fat Vector.

Returns: `FatVector` - Result of the multiplication.

Appendix B

Source Codes

Appendix B.A Data Structures

Data stuctures of the sparse matrix and fat vector.