



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

High-Performance Computing (HPC) systems are pivotal in solving complex computational problems. Leveraging such systems, this report investigates the optimization of sparse matrix-vector multiplication (SpMV) - a crucial operation in scientific computations. Two prevalent storage formats, Compressed Sparse Row (CSR) and ELLPACK, are parallelized using OpenMP and CUDA to enhance SpMV's efficiency on the CRES-CENT2 HPC cluster at Cranfield University.

The study reveals that CSR format, when parallelized with OpenMP, achieves superior performance across a majority of matrices due to efficient memory management and task distribution. CUDA parallelization exhibits significant speed-ups, especially for matrices with regular structures, indicating an intricate relationship between matrix properties and the efficiency of CSR and ELLPACK formats.

Experimental results suggest that the parallelization strategy should be selected based on matrix characteristics. For matrices with high density or regularity, CUDA outperforms OpenMP, whereas for less regular matrices, OpenMP provides sufficient speed-up. Sequential execution remains competitive for small matrices or those with unfavourable characteristics for parallelization.

This report underscores the necessity of aligning parallel programming strategies with matrix properties to fully exploit HPC capabilities, providing a foundation for future research towards more nuanced and effective computational techniques in scientific computing.

Table of Contents

Abstract	ii
Table of Contents	iii
List of Figures	vi
List of Tables	vii
1 Introduction	1
2 Methodology	2
2.1 Problem Statement	2
2.2 Data Structures	2
2.2.1 Sparse Matrix	2
2.2.1.1 Compressed Sparse Row (CSR) Format	2
2.2.1.2 ELLPACK	3
2.2.2 Fat Vector	3
2.3 Sparse Matrix - Fat Vector Multiplication	4
2.3.1 Sequential Algorithm using CSR Format	4
2.3.1.1 Algorithm Flow	4
2.3.1.2 Temporal Complexity Analysis	5
2.3.2 Sequential Algorithm using ELLPACK Format	6
2.3.2.1 Algorithm Flow	6
2.3.2.2 Temporal Complexity Analysis	7
2.4 Parallel Algorithms	8
2.4.1 OpenMP (Open Multi-Processing)	8
2.4.1.1 Workflow and Optimisation Strategy	8
2.4.1.2 Expected Performance Improvements	8
2.4.1.3 Challenges and Considerations	9
2.4.2 CUDA	10
2.4.2.1 Workflow and Kernel Design	10
2.4.2.2 Expected Performance Improvements	10
2.4.2.3 Challenges and Considerations	11
2.5 Implementation and Testing	12
2.5.1 Workflow	12
2.5.2 Comprehensive Testing Strategy	12
2.5.2.1 Test Parameters Configuration	12

2.5.2.1.1	OpenMP Parameters:	12
2.5.2.1.2	CUDA Parameters:	13
2.5.2.2	Correctness Verification	13
2.5.2.3	Performance Evaluation	13
3	Results and Discussion	15
3.1	Results	15
3.1.1	Sequential Algorithms	15
3.1.2	OpenMP	16
3.1.2.1	Chunk Size Analysis	16
3.1.2.2	Thread Count Analysis	18
3.1.2.3	Performance Comparison	19
3.1.3	CUDA	23
3.1.3.1	X Block Size Analysis	23
3.1.3.2	Y Block Size Analysis	24
3.1.3.3	Performance Comparison	25
3.1.4	Overall Performance Comparison	28
4	Conclusion	31
References		32
A	Documentation	33
A.A	Project tree	33
A.B	Getting Started	33
A.C	Methods Overview	34
A.C.1	Utils.h	34
A.C.1.1	ConvertPETScMatToFatVector	34
A.C.1.2	areMatricesEqual	34
A.C.1.3	readMatrixMarketFile	34
A.C.1.4	generateLargeFatVector	35
A.C.1.5	serialize and deserialize	35
A.C.2	SparseMatrixFatVectorMultiply.h	35
A.C.2.1	sparseMatrixFatVectorMultiply	35
A.C.3	SparseMatrixFatVectorMultiplyRowWise.h	36
A.C.3.1	sparseMatrixFatVectorMultiplyRowWise	36
A.C.4	SparseMatrixFatVectorMultiplyColumnWise.h	36
A.C.4.1	sparseMatrixFatVectorMultiplyColumnWise	36
A.C.5	SparseMatrixFatVectorMultiplyNonZeroElement.h	36
A.C.5.1	sparseMatrixFatVectorMultiplyNonZeroElement	36
B	Source Codes	37
B.A	Data Structures	37
B.B	Sequential Algorithm	39
B.B.1	CRS Format	39
B.B.2	ELLPACK Format	40
B.C	OpenMP Parallel Algorithm	42

B.C.1	CRS Format	42
B.C.2	ELLPACK Format	43
B.D	CUDA Parallel Algorithm	45
B.D.1	CRS Format	45
B.D.2	ELLPACK Format	47
B.E	Utilities	51
B.F	Main Program	54
B.F.1	OpenMP	54
B.F.2	CUDA	57

List of Figures

3.1	OpenMP Performance for CRS Format depending on Chunk Size	16
3.2	OpenMP Performance for ELLPACK Format depending on Chunk Size . .	17
3.3	OpenMP Performance for CRS Format depending on Threads	18
3.4	OpenMP Performance for ELLPACK Format depending on Threads . . .	18
3.5	Performance of Serial and Parallel Algorithms using OpenMP for $k = 1$.	19
3.6	Performance of Serial and Parallel Algorithms using OpenMP for $k = 2$.	20
3.7	Performance of Serial and Parallel Algorithms using OpenMP for $k = 3$.	20
3.8	Performance of Serial and Parallel Algorithms using OpenMP for $k = 6$.	21
3.9	CUDA Performance for CRS Format depending on X BlockSize	23
3.10	CUDA Performance for ELLPACK Format depending on X BlockSize . .	23
3.11	CUDA Performance for CRS Format depending on Y BlockSize	24
3.12	CUDA Performance for ELLPACK Format depending on Y BlockSize . .	24
3.13	Performance of Serial and Parallel Algorithms using CUDA for $k = 1$.	25
3.14	Performance of Serial and Parallel Algorithms using CUDA for $k = 2$.	25
3.15	Performance of Serial and Parallel Algorithms using CUDA for $k = 3$.	26
3.16	Performance of Serial and Parallel Algorithms using CUDA for $k = 6$.	26
3.17	OpenMP vs CUDA Performance for $k = 1$	28
3.18	OpenMP vs CUDA Performance for $k = 2$	28
3.19	OpenMP vs CUDA Performance for $k = 3$	29
3.20	OpenMP vs CUDA Performance for $k = 6$	29

List of Tables

2.1	Summary of sparse matrices	14
3.1	Performance Comparison of CRS and ELLPACK Sequential Algorithms .	15
3.2	CRS vs ELLPACK using OpenMP	21
3.3	CRS vs ELLPACK using CUDA	27
3.4	Overall Performance Comparison	29

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 shared 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:

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

where most elements of A are zeros.

The vector X is defined as:

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

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 B.B.

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:

$$T(n) = O(NZ \times k) \quad (2.3)$$

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
         $colIndex \leftarrow A.colIndices[i \times A.maxNonZerosPerRow + j]$ 
         $value \leftarrow A.values[i \times A.maxNonZerosPerRow + j]$ 
        if  $colIndex \neq -1$  then
            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 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 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:

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

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.
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

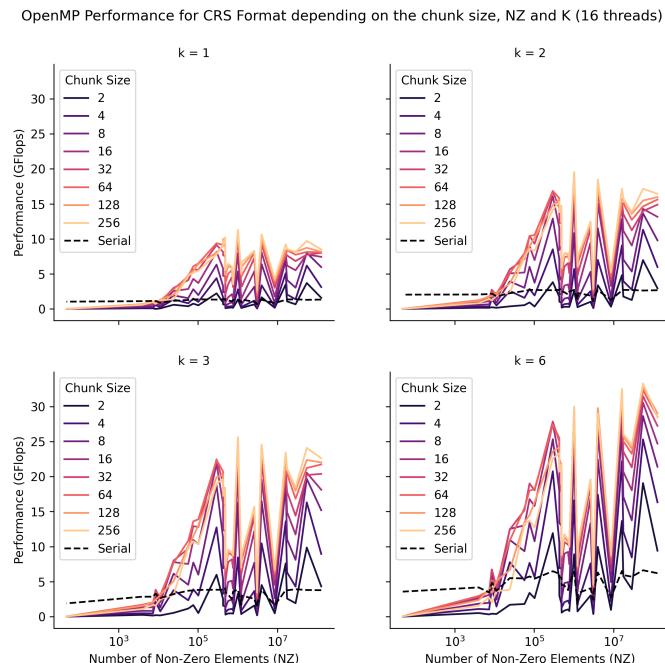


Figure 3.1: OpenMP Performance for CRS Format depending on Chunk Size

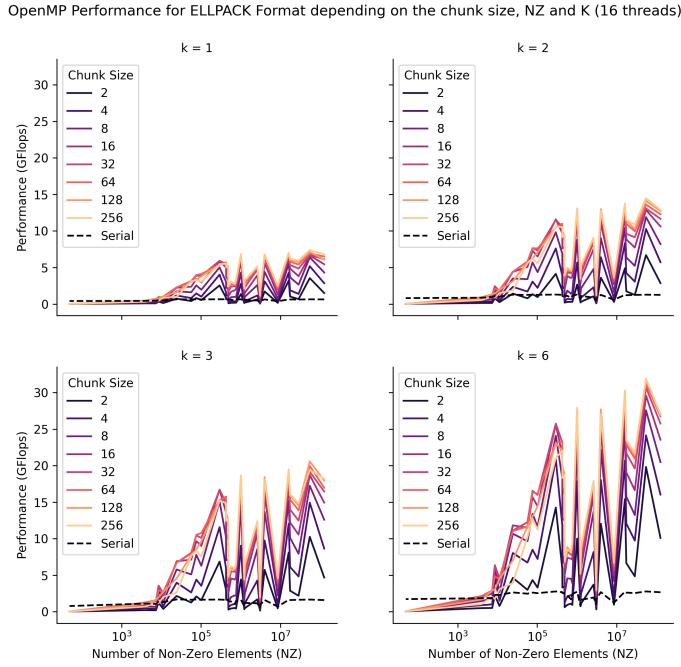


Figure 3.2: OpenMP Performance for ELLPACK Format depending on Chunk Size

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

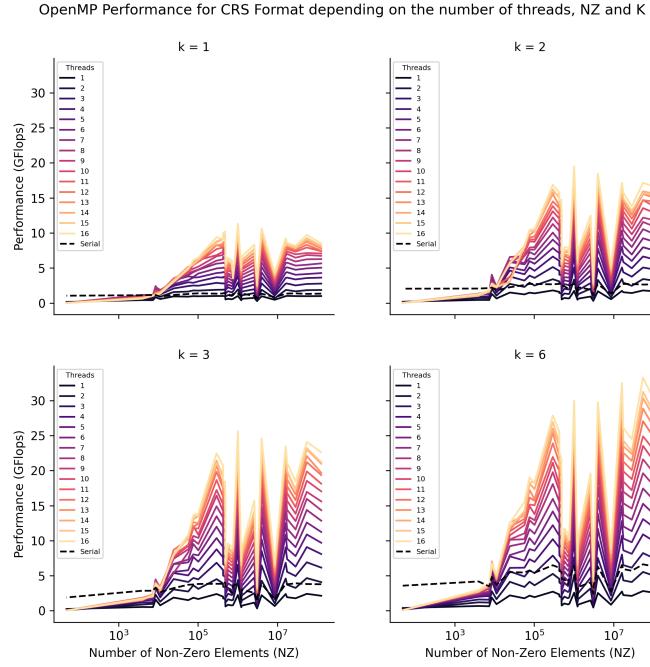


Figure 3.3: OpenMP Performance for CRS Format depending on Threads

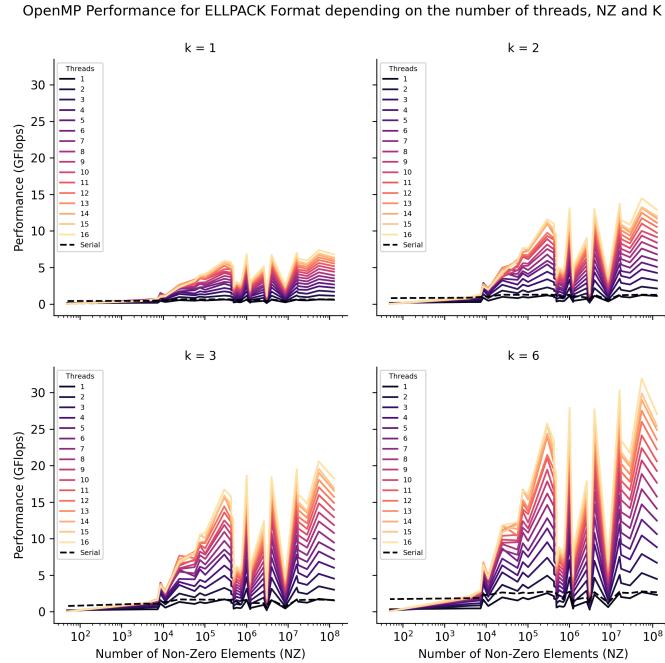


Figure 3.4: OpenMP Performance for ELLPACK Format depending on Threads

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

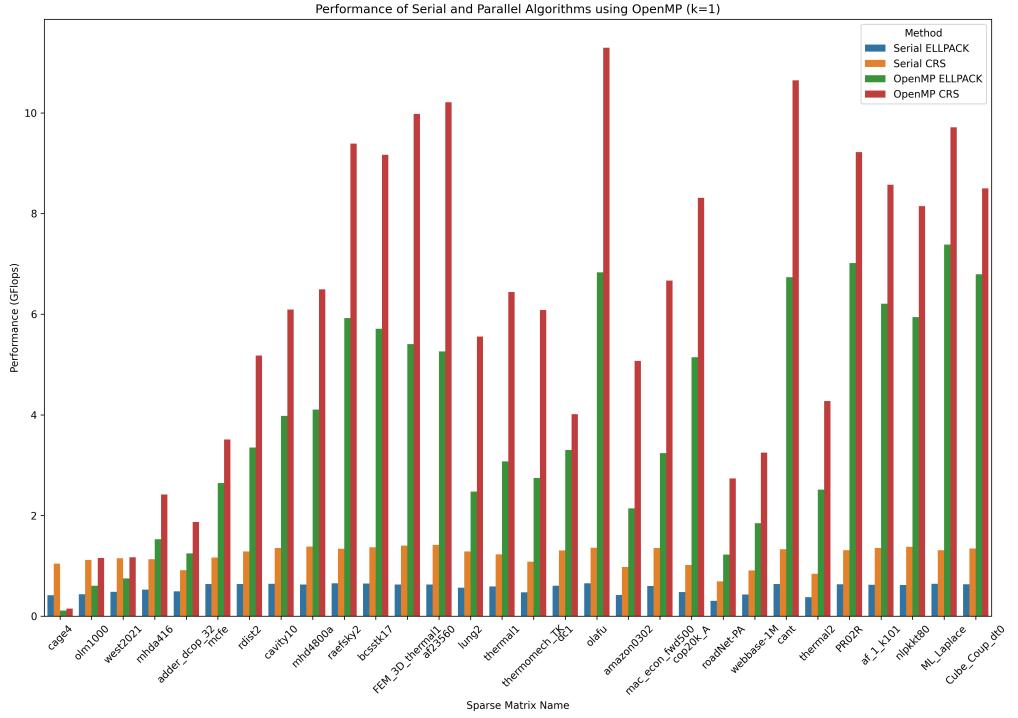
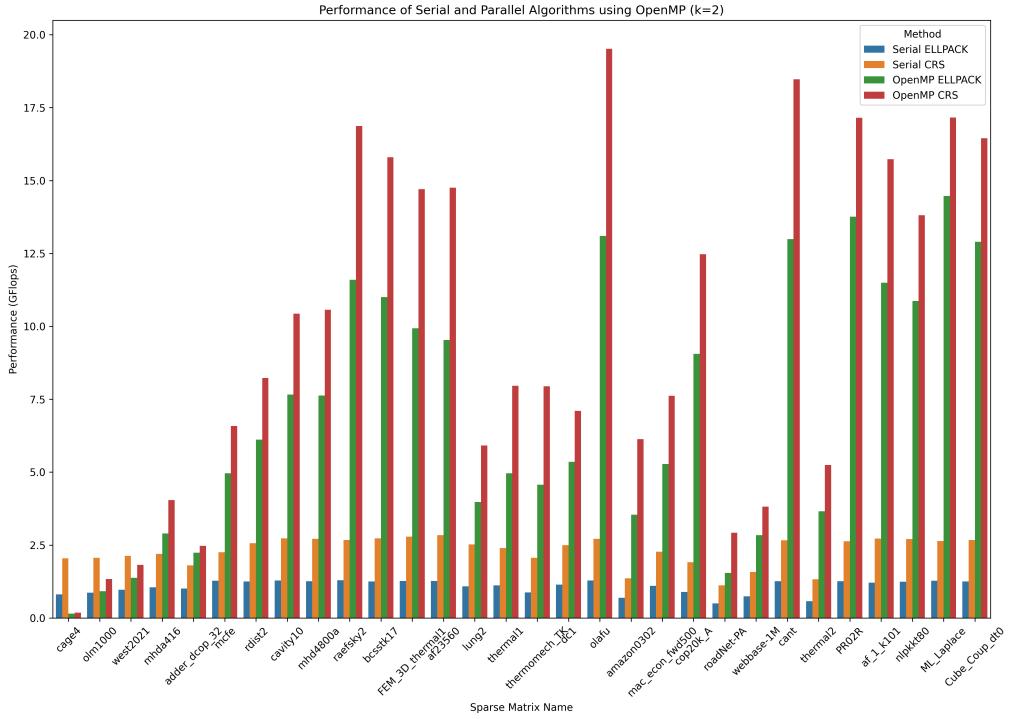
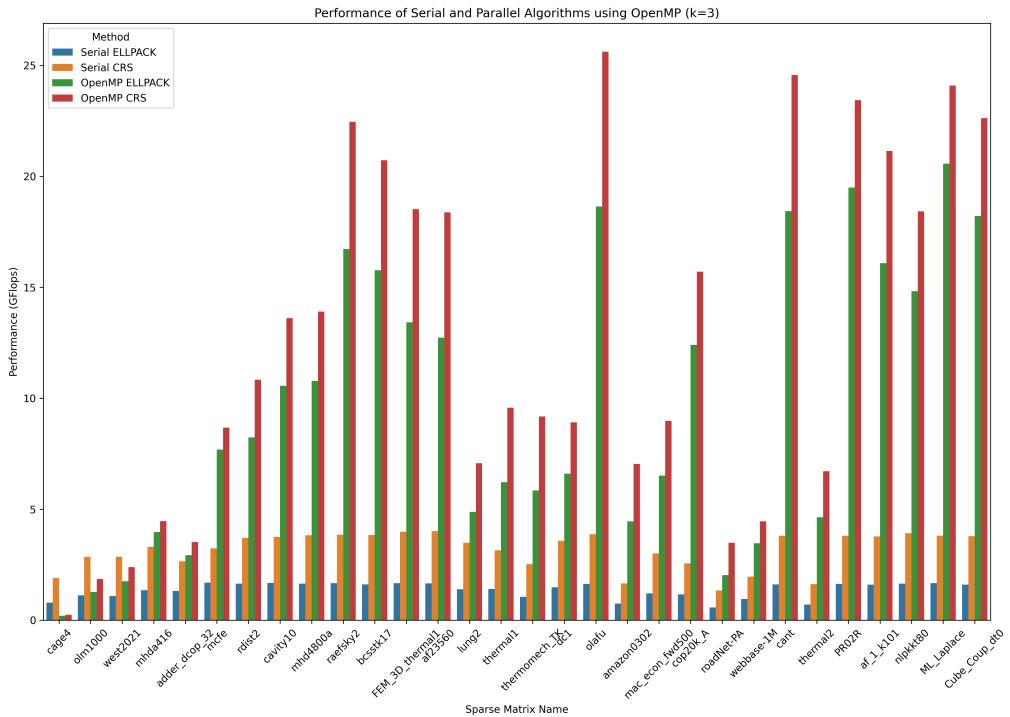
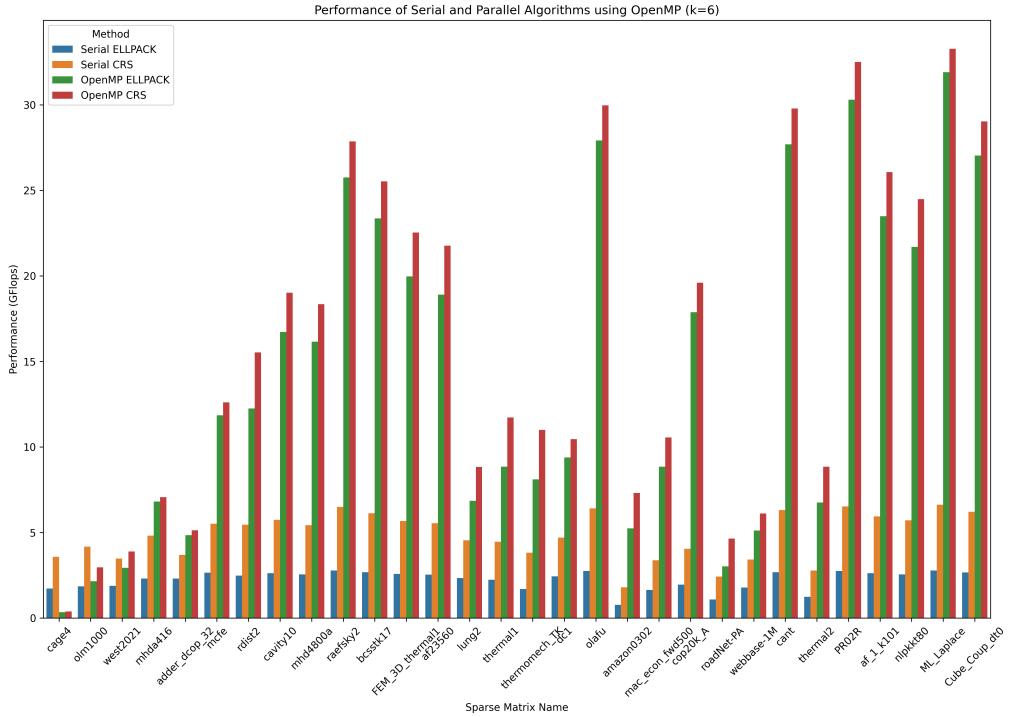


Figure 3.5: Performance of Serial and Parallel Algorithms using OpenMP for $k = 1$

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 $k = 6$

As shown in Figures 3.5,

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
mcf	12.6097	11.8456	CRS	2.077682
mhd4800a	18.3501	16.1582	CRS	3.352021

Matrix	CRS	ELLPACK	Best Structure	Speedup
mhda416	7.06099	6.80262	CRS	1.278130
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

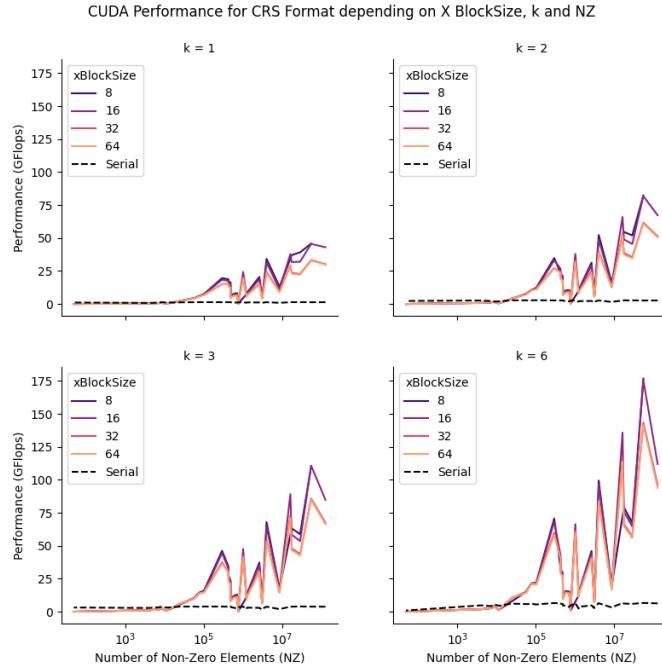


Figure 3.9: CUDA Performance for CRS Format depending on X BlockSize

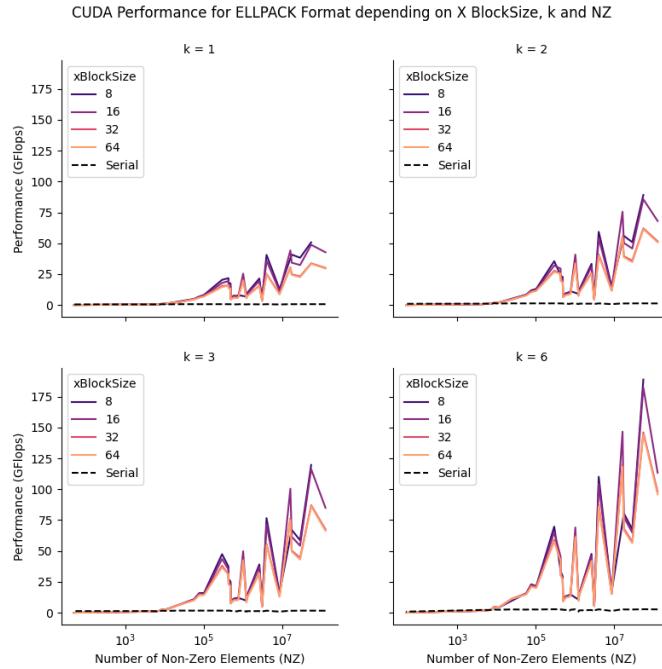


Figure 3.10: CUDA Performance for ELLPACK Format depending on X BlockSize

3.1.3.2 Y Block Size Analysis

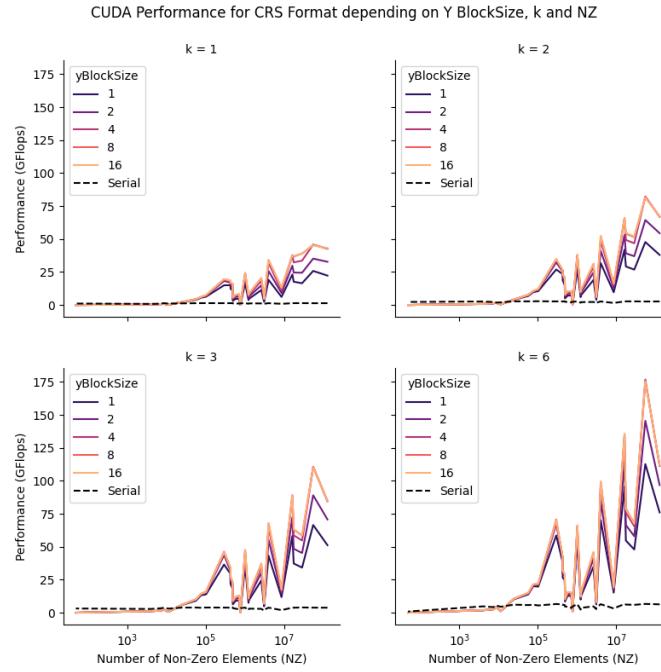


Figure 3.11: CUDA Performance for CRS Format depending on Y BlockSize

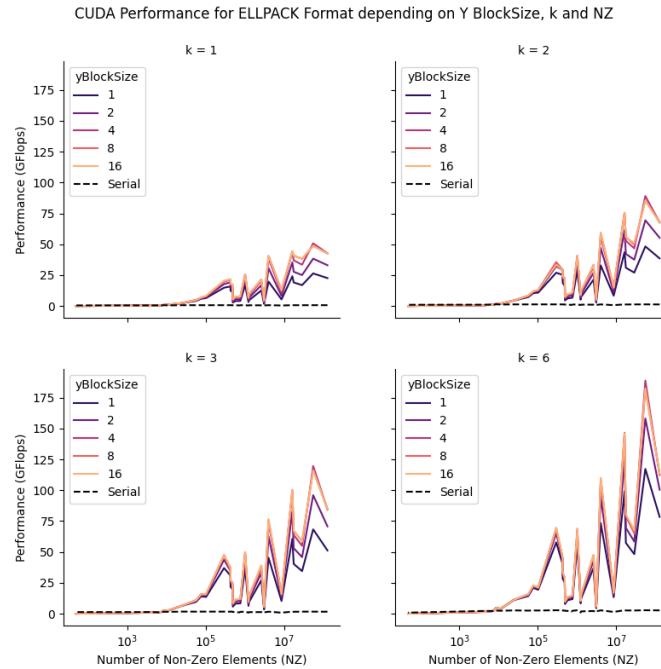


Figure 3.12: CUDA Performance for ELLPACK Format depending on Y BlockSize

3.1.3.3 Performance Comparison

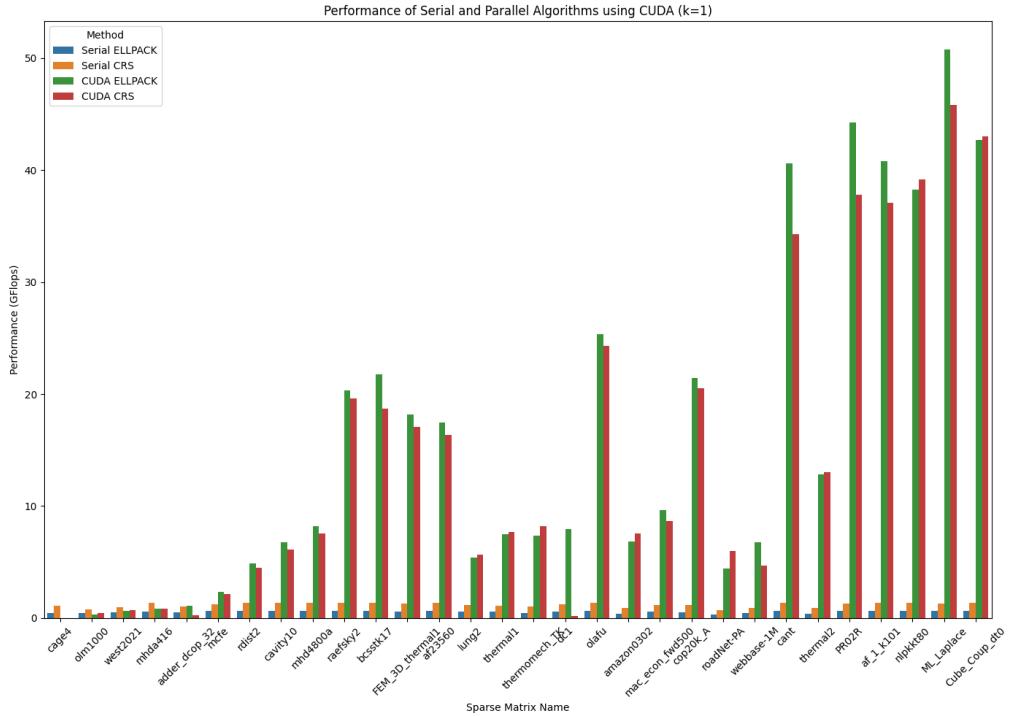


Figure 3.13: Performance of Serial and Parallel Algorithms using CUDA for $k = 1$

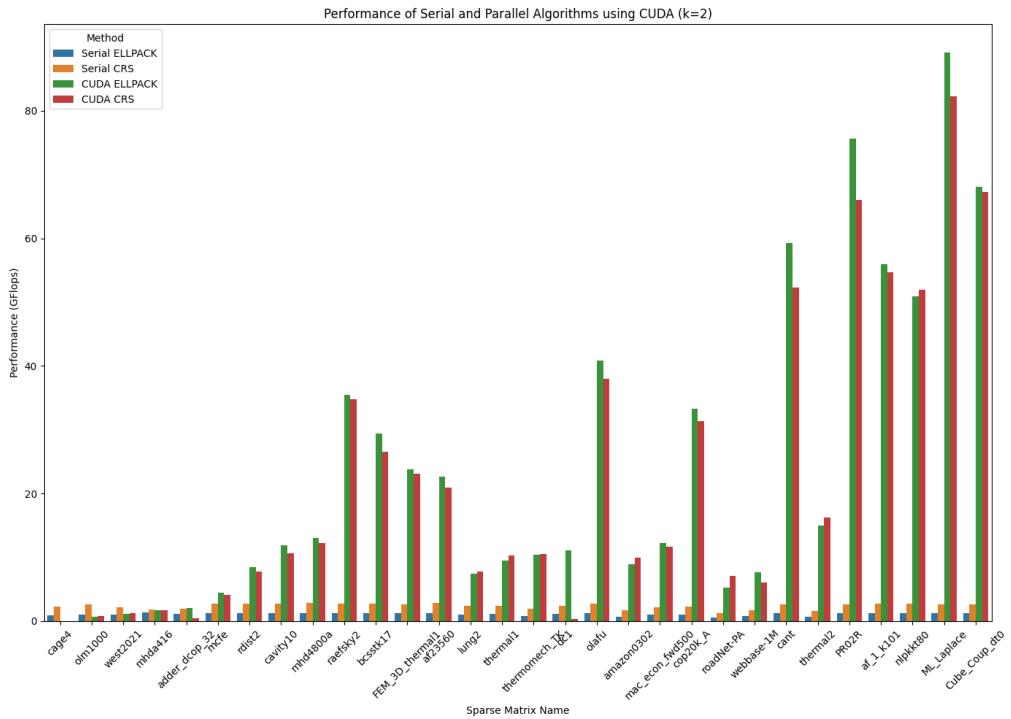


Figure 3.14: Performance of Serial and Parallel Algorithms using CUDA for $k = 2$

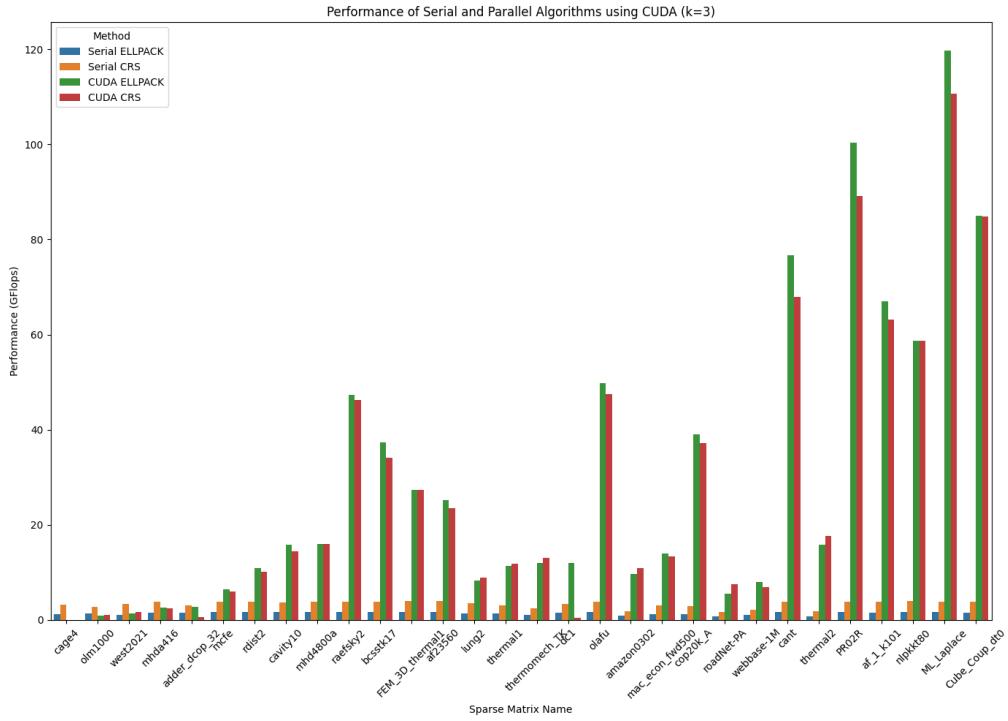
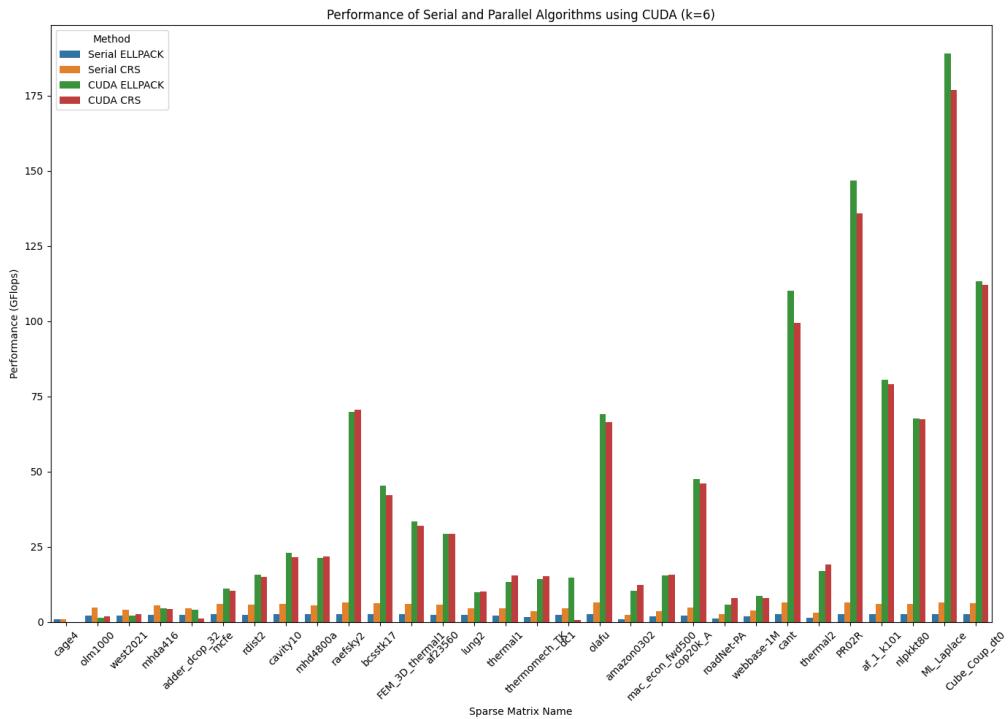
Figure 3.15: Performance of Serial and Parallel Algorithms using CUDA for $k = 3$ Figure 3.16: Performance of Serial and Parallel Algorithms using CUDA for $k = 6$

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
mhma416	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

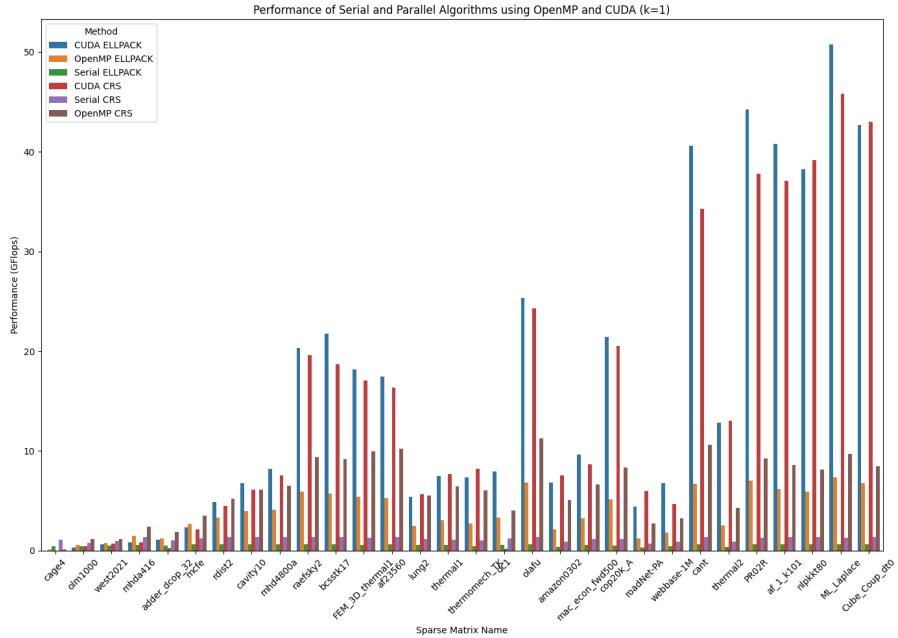


Figure 3.17: OpenMP vs CUDA Performance for $k = 1$

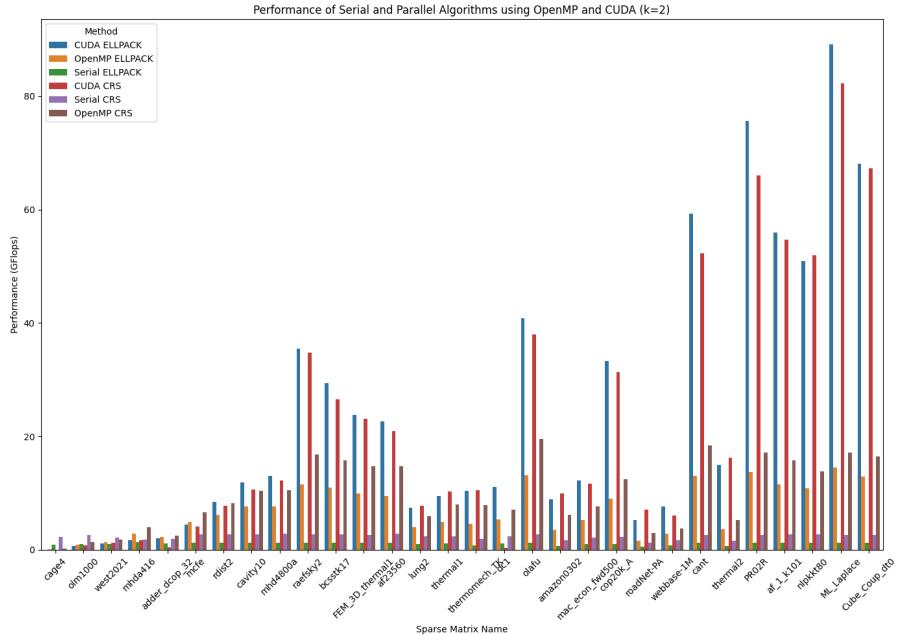


Figure 3.18: OpenMP vs CUDA Performance for $k = 2$

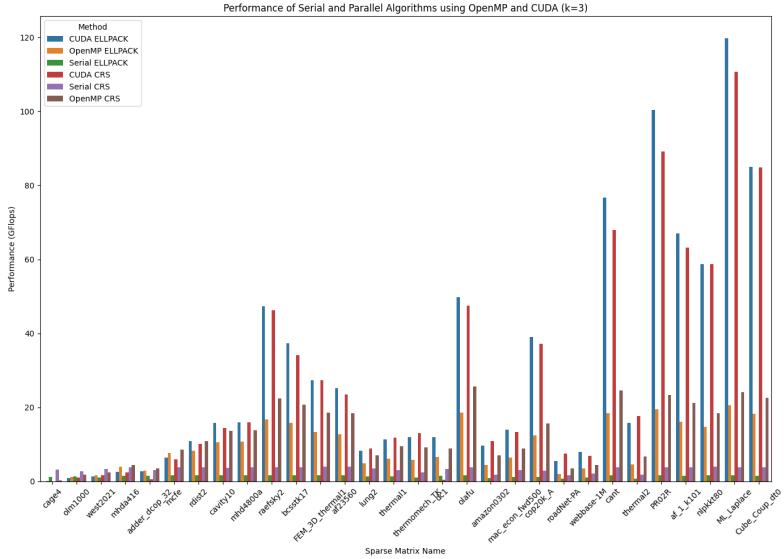
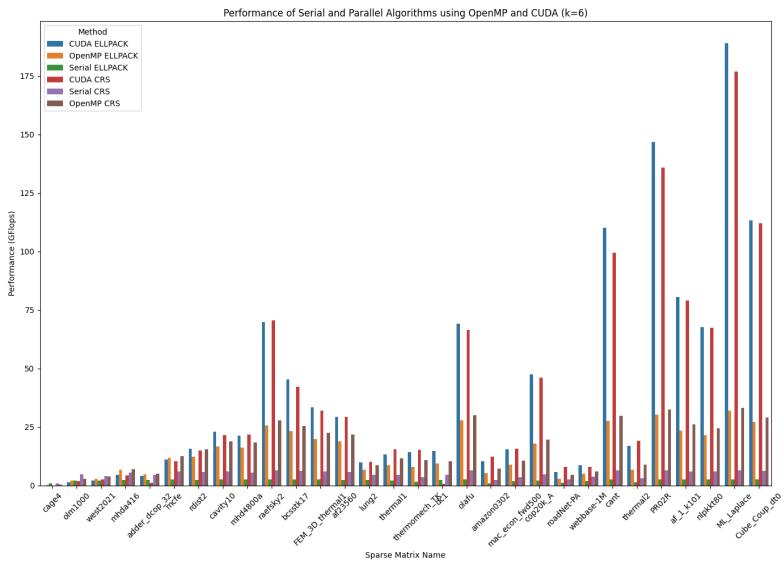
Figure 3.19: OpenMP vs CUDA Performance for $k = 3$ Figure 3.20: OpenMP vs CUDA Performance for $k = 6$

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

Matrix	Best Performance	Best Structure	Best Method	Speedup
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

Exceptionally high performance is observed for certain matrices under CUDA, such as *Cube_Coup_dt0* and *ML_Laplace*, revealing the match between certain data structures and GPU optimisation. On the other hand, OpenMP shows a moderate advantage for specific matrices, such as *adder_dcop_32* and *mcfe*, offering a significant performance improvement without the need for specialised hardware, thanks to parallelization on shared memory architectures.

However, matrices of small size or with characteristics less favourable to parallelization, such as *cage4*, *olm1000*, and *west2021*, show no significant improvement with OpenMP or CUDA, indicating that for some cases sequential execution remains the most suitable method.

These observations lead to the conclusion that the choice between CRS and ELLPACK formats, as well as the decision to use sequential execution, OpenMP or CUDA, should be informed by the specific properties of the matrices in question. Optimisation of the calculation parameters and careful evaluation of the data characteristics are essential to maximise the efficiency of operations on hollow matrices.

Chapter 4

Conclusion

In summary, this report explored the performance of the CSR and ELLPACK formats for dense vector multiplication of hollow matrices using the OpenMP and CUDA parallel programming paradigms. The results show a marked preference for the CSR format when parallelized with OpenMP, which is probably due to more efficient memory management and optimal task distribution between threads. On the other hand, CUDA performance is strongly influenced by the sparsity and structural regularity of matrices, with symmetric and dense matrices seemingly favouring the ELLPACK format.

The benefits of using CUDA on GPU architectures have been demonstrated, particularly for arrays that align well with memory access models optimised for these devices. However, it is clear that the benefits of parallelization with CUDA or OpenMP are closely related to the specific characteristics of the arrays in question, and that a sequential approach may be preferable for small arrays or those with patterns less conducive to parallelization.

These findings underline the importance of a thorough analysis of data structures and computational parameters to maximise the efficiency of operations on hollow matrices. Ultimately, the choice between CSR and ELLPACK formats, as well as the decision to use sequential execution, OpenMP or CUDA, must be informed by the specific properties of the matrices. Such a nuanced understanding will further optimise the performance of the scientific and engineering applications that depend on these intensive computations.

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_path>
```

Appendix A.C Methods Overview

A.C.1 Utils.h

A.C.1.1 ConvertPETScMatToFatVector

Description: Converts a PETSc matrix to a FatVector structure.

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 in the CRS and ELLPACK formats, as well as the FatVector structure.

```
1 #ifndef MATRIXDEFINITIONS_H
2 #define MATRIXDEFINITIONS_H
3
4 #include <vector>
5
6 /**
7  * @brief Struct to represent a sparse matrix in Compressed Row Storage (CRS)
8  * format
9  *
10 * @param values Non-zero values
11 * @param colIndices Column indices of non-zero values
12 * @param rowPtr Row pointers
13 * @param numRows Number of rows
14 * @param numCols Number of columns
15 */
16 struct SparseMatrixCRS
17 {
18     std::vector<double> values;
19     std::vector<int> colIndices;
20     std::vector<int> rowPtr;
21     int numRows;
22     int numCols;
23 };
24
25 /**
26  * @brief Struct to represent a sparse matrix in ELLPACK format
27  *
28  * @param values Non-zero values
29  * @param colIndices Column indices of non-zero values
30  * @param maxNonZerosPerRow Maximum number of non-zero values per row
31  * @param numRows Number of rows
32  * @param numCols Number of columns
33 */
34 struct SparseMatrixELLPACK
35 {
36     std::vector<double> values;
37     std::vector<int> colIndices;
38     int maxNonZerosPerRow;
39     int numRows;
40     int numCols;
41 };
42
43 /**
44  * @brief Struct to represent a fat vector
```

```
44  *
45  * @param values  Matrix values
46  * @param numRows  Number of rows
47  * @param numCols  Number of columns
48  */
49 struct FatVector
50 {
51     std::vector<double> values;
52     int numRows;
53     int numCols;
54 };
55
56 #endif
```

Appendix B.B Sequential Algorithm

B.B.1 CRS Format

```

1 #include "matrixMultivectorProductCRS.h"
2
3 /**
4 * @brief Perform the matrix-vector multiplication in the CRS format
5 *
6 * @param sparseMatrix Sparse matrix in CRS format
7 * @param fatVector Dense vector
8 * @param result Pointer to the result vector
9 * @param testNumber Number of iterations for the performance measurement
10 * @return double GFLOPS Performance of the kernel in GFLOPS
11 */
12 double matrixMultivectorProductCRS(const SparseMatrixCRS &sparseMatrix,
13                                     const FatVector &fatVector, FatVector &result,
14                                     const int testNumber)
15 {
16     std::vector<double> times(testNumber); // Vector for storing times of
17     // individual iterations
18
19     // Perform testNumber iterations
20     for (int i = 0; i < testNumber; ++i)
21     {
22         result.values.assign(result.values.size(), 0.0); // Free the result vector
23
24         auto start = std::chrono::high_resolution_clock::now(); // Start timing
25
25         int numRows = sparseMatrix.numRows; // Number of rows in the sparse matrix
26         int vecCols = fatVector.numCols; // Number of columns in the dense
27         // vector
28
28         // Iterate over the rows of the sparse matrix
29         for (int i = 0; i < numRows; ++i)
30         {
31             // Iterate over the non-zero elements in the current row
32             for (int j = sparseMatrix.rowPtr[i]; j < sparseMatrix.rowPtr[i + 1]; ++
33                 j)
34             {
35                 int colIndex = sparseMatrix.colIndices[j]; // Column index in the
36                 // sparse matrix
37                 double value = sparseMatrix.values[j]; // Value of the non-zero
38                 // element
39
39                 // Accumulate the product into the result vector
40                 for (int k = 0; k < vecCols; ++k)
41                 {
42                     int resultIndex = i * vecCols + k; // Calculate the flat
43                     // index for the result vector
44                     int vectorIndex = colIndex * vecCols + k; // Calculate the flat index for
45                     // the dense vector
46                     result.values[resultIndex] += value * fatVector.values[
47                         vectorIndex]; // Perform the multiplication and
48                         accumulation
49                 }
50             }
51         }
52     }
53
54     auto end = std::chrono::high_resolution_clock::now(); // End
55     // timing
56     std::chrono::duration<double, std::milli> duration = end - start; //
57     // Calculate the duration
58     times.push_back(duration.count()); // Store
59     // the duration
60 }

```

```

51     // Calculate performance
52     double avgTime = std::accumulate(times.begin(), times.end(), 0.0) / times.size()
53     () ; // Calculate average time
53     int NZ = sparseMatrix.values.size(); // Number of non-zero entries
54     int k = fatVector.numCols; // Number of columns
55     in matrix X
55     double T = avgTime / 1000.0; // Average time in
56     seconds
56     double FLOPS = (2.0 * NZ * k) / T; // Number of floating point
57     operations
57     double GFLOPS = FLOPS / 1e9; // Performance in
58     GFLOPS
59
59     // DISPLAY RESULTS (FOR DEBUGGING)
60     // std::cout << "Average Time (ms): " << avgTime << std::endl;
61     // std::cout << "GFLOPS (CRS): " << GFLOPS << std::endl;
62
63     return GFLOPS;
64 }
```

B.B.2 ELLPACK Format

```

1 #include "matrixMultivectorProductELLPACK.h" // Include guard for the ELLPACK
2     versio
3
4 /**
5  * @brief Perform the matrix-vector multiplication for an ELLPACK matrix
6  *
7  * @param sparseMatrix Sparse matrix in ELLPACK format
8  * @param fatVector Dense vector
9  * @param result Pointer to the result vector
10 * @param testNumber Number of iterations for the performance measurement
11 * @return double GFLOPS Performance of the kernel in GFLOPS
11 */
12 double matrixMultivectorProductELLPACK(const SparseMatrixELLPACK &sparseMatrix,
13                                         const FatVector &fatVector, FatVector &
13                                         result, const int testNumber)
14 {
15
16     std::vector<double> times(testNumber); // Store time for each invocation in
16     milliseconds
17
18     // Perform testNumber iterations
19     for (int n = 0; n < testNumber; ++n)
20     {
21         result.values.assign(result.values.size(), 0.0); // Free the result vector
22
23         auto start = std::chrono::high_resolution_clock::now(); // Start timing
24
25         int numRows = sparseMatrix.numRows; // Number of rows in the sparse matrix
26         int vecCols = fatVector.numCols; // Number of columns in the dense
26         vector
27
28         // Iterate over the rows of the sparse matrix
29         for (int i = 0; i < numRows; ++i)
30         {
31             // Iterate over each possible non-zero element in the row, up to the
31             maximum number per row
32             for (int j = 0; j < sparseMatrix.maxNonZerosPerRow; ++j)
33             {
34                 int colIndex = sparseMatrix.colIndices[i * sparseMatrix.
34                 maxNonZerosPerRow + j]; // Column index in the sparse matrix
35             }
35         }
36     }
37 }
```

```

35     double value = sparseMatrix.values[i * sparseMatrix.
36                                         maxNonZerosPerRow + j]; // Value of the non-zero element
37     // If the column index is -1, it indicates padding and should be
38     // ignored
39     if (colIndex != -1)
40     {
41         // Iterate over the columns of the dense vector
42         for (int k = 0; k < vecCols; ++k)
43         {
44             int resultIndex = i * vecCols + k;
45                                         // Calculate the flat
46                                         index for the result vector
47             int vectorIndex = colIndex * vecCols + k;
48                                         // Calculate the flat index
49                                         for the dense vector
50             result.values[resultIndex] += value * fatVector.values[
51                                         vectorIndex]; // Compute the result
52         }
53     }
54 }
55
56     auto end = std::chrono::high_resolution_clock::now(); // End
57     timing
58     std::chrono::duration<double, std::milli> duration = end - start; // Calculate the duration
59     times[n] = duration.count(); // Store the duration
60 }
61
62 // Calculate performance
63 double avgTimeMs = std::accumulate(times.begin(), times.end(), 0.0) / times.
64                                         size(); // Calculate average time in milliseconds
65 int NZ = sparseMatrix.numRows * sparseMatrix.maxNonZerosPerRow;
66                                         // Total potential non-zeros, adjustments for actual
67                                         non-zeros might be needed
68 int k = fatVector.numCols;
69                                         // Number of
70                                         columns in matrix X
71 double avgTimeSec = avgTimeMs / 1000.0;
72                                         // Convert milliseconds to
73                                         seconds for calculation
74 double FLOPS = (2.0 * NZ * k) / avgTimeSec;
75                                         // Number of floating point
76                                         operations
77 double GFLOPS = FLOPS / 1e9;
78                                         // Performance in
79                                         GFLOPS
80
81 // DISPLAY RESULTS (FOR DEBUGGING)
82 // std::cout << "Average Time: " << avgTimeMs << " ms" << std::endl;
83 // std::cout << "Performance (ELLPACK): " << GFLOPS << " GFLOPS" << std::endl;
84
85     return GFLOPS;
86 }

```

Appendix B.C OpenMP Parallel Algorithm

B.C.1 CRS Format

```

1 #include "matrixMultivectorProductCRSOpenMP.h"
2
3 /**
4 * @brief Perform the matrix-vector multiplication in the CRS format using OpenMP
5 *
6 * @param sparseMatrix Sparse matrix in CRS format
7 * @param fatVector Dense vector
8 * @param result Pointer to the result vector
9 * @param testNumber Number of iterations for the performance measurement
10 * @param numThreads Number of threads to use
11 * @param chunkSize Chunk size for the OpenMP parallel for
12 * @return double GFLOPS Performance of the kernel in GFLOPS
13 */
14 double matrixMultivectorProductCRSOpenMP(const SparseMatrixCRS &sparseMatrix,
15                                         const FatVector &fatVector, FatVector &
16                                         result, const int testNumber, const
17                                         int numThreads, const int chunkSize)
18 {
19     std::vector<double> times(testNumber); // Store time for each invocation
20
21     omp_set_num_threads(numThreads); // Set the number of threads for OpenMP
22
23     // Perform testNumber iterations
24     for (int n = 0; n < testNumber; ++n)
25     {
26         result.values.assign(result.values.size(), 0.0); // Free the result vector
27
28         double start_time = omp_get_wtime(); // Start timing
29
30         // Perform the matrix-vector multiplication
31         #pragma omp parallel for schedule(dynamic, chunkSize) // Parallelize the
32             // loop using OpenMP with dynamic scheduling and a chunk size
33         for (int i = 0; i < sparseMatrix.numRows; ++i)
34         {
35             double local_result[fatVector.numCols] = {0}; // Temporary local result
36             // array
37             // Iterate over the non-zero elements in the current row
38             for (int j = sparseMatrix.rowPtr[i]; j < sparseMatrix.rowPtr[i + 1]; ++
39                 j)
40             {
41                 int colIndex = sparseMatrix.colIndices[j]; // Column index in the
42                     // sparse matrix
43                 double value = sparseMatrix.values[j]; // Value of the non-zero
44                     // element
45                 // Accumulate the product into the result vector
46                 for (int k = 0; k < fatVector.numCols; ++k)
47                 {
48                     local_result[k] += value * fatVector.values[colIndex * //
49                         fatVector.numCols + k]; // Perform the multiplication and
50                         // accumulation
51                 }
52             }
53             // Combine local results
54             for (int k = 0; k < fatVector.numCols; ++k)
55             {
56                 // Combine local results
57                 #pragma omp atomic // Atomic operation
58                 result.values[i * fatVector.numCols + k] += local_result[k]; // //
59                     // Accumulate the local result into the global result
60             }
61         }
62
63         double end_time = omp_get_wtime(); // End timing
64         times[n] = end_time - start_time; // Store the duration
65     }
66 }
```

```

56     // Calculate performance
57     double avgTime = std::accumulate(times.begin(), times.end(), 0.0) / times.size()
58     ();
59     int NZ = sparseMatrix.values.size(); // Number of non-zero entries
60     int k = fatVector.numCols;          // Number of columns in matrix X
61     double T = avgTime;               // Average time in seconds
62     double FLOPS = (2.0 * NZ * k) / T; // Number of floating point operations
63     double GFLOPS = FLOPS / 1e9;      // Performance in GFLOPS
64
65     // DISPLAY RESULTS (FOR DEBUGGING)
66     // std::cout << "OpenMP Average Time (CRS): " << avgTime << " s" << std::endl;
67     // std::cout << "Performance (CRS): " << GFLOPS << " GFLOPS" << std::endl;
68
69     return GFLOPS;
}

```

B.C.2 ELLPACK Format

```

40
41         {
42             double value = sparseMatrix.values[i * sparseMatrix.
43                                         maxNonZerosPerRow + j]; // Value of the non-zero element
44             // Iterate over the columns of the dense vector
45             for (int k = 0; k < fatVector.numCols; ++k)
46             {
47                 local_result[k] += value * fatVector.values[colIndex *
48                                         fatVector.numCols + k]; // Compute the result
49             }
50         }
51     }
52     // Combine local results
53     for (int k = 0; k < fatVector.numCols; ++k)
54     {
55         #pragma omp atomic
56                                     // Atomic operation
57         result.values[i * fatVector.numCols + k] += local_result[k]; // Accumulate the local result into the global result
58     }
59 }
60
61     double end_time = omp_get_wtime(); // End timing
62     times[n] = end_time - start_time; // Store the duration
63 }
64
65 // Calculate performance
66 double avgTime = std::accumulate(times.begin(), times.end(), 0.0) / times.size()
67 () ; // Calculate average time
68 int NZ = sparseMatrix.values.size(); // Number of non-zero entries
69 int k = fatVector.numCols; // Number of columns
70
71     in matrix X
72 double T = avgTime; // Average time in seconds
73 double FLOPS = (2.0 * NZ * k) / T; // Number of floating point operations
74 double GFLOPS = FLOPS / 1e9; // Performance in GFLOPS
75
76 // DISPLAY RESULTS (FOR DEBUGGING)
77 // std::cout << "OpenMP Average Time (ELLPACK): " << avgTime << " s" << std::endl;
78 // std::cout << "Performance (ELLPACK): " << GFLOPS << " GFLOPS" << std::endl;
79
80     return GFLOPS;
81 }

```

Appendix B.D CUDA Parallel Algorithm

B.D.1 CRS Format

```

1 #include "matrixMultivectorProductCRSCUDA.h"
2 #include "../cudaUtils.cuh"
3
4 /**
5 * @brief Kernel for sparse matrix-vector product in CRS format
6 *
7 * @param values Non-zero values of the matrix
8 * @param colIndices Column indices of the non-zero values
9 * @param rowPtr Row pointer of the matrix
10 * @param vector Input vector
11 * @param result Output vector
12 * @param numRows Number of rows in the matrix
13 * @param vecCols Number of columns in the input vector
14 */
15 __global__ void spmvCrsKernel(const double *values, const int *colIndices, const
16     int *rowPtr, const double *vector, double *result, int numRows, int vecCols)
17 {
18     int row = blockIdx.x * blockDim.y + threadIdx.y; // Row index
19     int col = threadIdx.x; // Column index
20     // Check if the thread is within the matrix dimensions
21     if (row < numRows && col < vecCols)
22     {
23         double sum = 0.0; // Initialize the sum
24         int rowStart = rowPtr[row]; // Start index of the row
25         int rowEnd = rowPtr[row + 1]; // End index of the row
26         // Loop over the non-zero values of the row
27         for (int i = rowStart; i < rowEnd; ++i)
28         {
29             int colIndex = colIndices[i]; // Column index of the non-zero value
30             sum += values[i] * vector[colIndex * vecCols + col]; // Multiply and accumulate
31         }
32         atomicAddDouble(&result[row * vecCols + col], sum); // Store the result
33     }
34 }
35 /**
36 * @brief Perform matrix-vector product using CUDA
37 *
38 * @param sparseMatrix Sparse matrix in CRS format
39 * @param fatVector Input vector
40 * @param result Output vector
41 * @param testNumber Number of invocations
42 * @param xBlockSize X-dimension of the block
43 * @param yBlockSize Y-dimension of the block
44 * @return double Performance in GFLOPS
45 */
46 double matrixMultivectorProductCRSCUDA(const SparseMatrixCRS &sparseMatrix, const
47     FatVector &fatVector, FatVector &result, const int testNumber, const int
48     xBlockSize, const int yBlockSize)
49 {
50     std::vector<float> times(testNumber); // Store time for each invocation
51
52     const dim3 BLOCK_DIM(xBlockSize, yBlockSize); // Block dimensions
53
54     cudaEvent_t start, stop; // CUDA events for timing
55     cudaEventCreate(&start); // Start event
56     cudaEventCreate(&stop); // Stop event
57
58     // Device memory
59     double *d_values, *d_fatVector, *d_result; // Device
60     pointers for the values, input vector and result
61     int *d_colIndices, *d_rowPtr; // Device
62     pointers for the column indices and row pointer

```

```

59     size_t valuesSize = sparseMatrix.values.size();                      // Non-zero
60     values size
61     size_t colIndicesSize = sparseMatrix.colIndices.size();             // Column indices
62     size
63     size_t rowPtrSize = sparseMatrix.rowPtr.size();                      // Row pointer
64     size
65     size_t fatVectorSize = fatVector.values.size();                      // Input vector
66     size
67     size_t resultSize = sparseMatrix numRows * fatVector.numCols;        // Result size
68
69 // Allocate device memory
70 checkCudaErrors(cudaMalloc(&d_values, valuesSize * sizeof(double)));
71                                         // Allocate
72                                         memory for the non-zero values
73 checkCudaErrors(cudaMalloc(&d_colIndices, colIndicesSize * sizeof(int)));
74                                         // Allocate memory
75                                         for the column indices
76 checkCudaErrors(cudaMalloc(&d_rowPtr, rowPtrSize * sizeof(int)));
77                                         // Allocate
78                                         memory for the row pointer
79 checkCudaErrors(cudaMalloc(&d_fatVector, fatVectorSize * sizeof(double)));
80                                         // Allocate memory
81                                         for the input vector
82 checkCudaErrors(cudaMalloc(&d_result, resultSize * sizeof(double)));
83                                         // Allocate
84                                         memory for the result
85 checkCudaErrors(cudaMemcpy(d_values, sparseMatrix.values.data(), valuesSize *
86     sizeof(double), cudaMemcpyHostToDevice));                         // Copy non-zero values
87                                         to device
88 checkCudaErrors(cudaMemcpy(d_colIndices, sparseMatrix.colIndices.data(),
89     colIndicesSize * sizeof(int), cudaMemcpyHostToDevice));           // Copy column
90                                         indices to device
91 checkCudaErrors(cudaMemcpy(d_rowPtr, sparseMatrix.rowPtr.data(), rowPtrSize *
92     sizeof(int), cudaMemcpyHostToDevice));                         // Copy row pointer to
93                                         device
94 checkCudaErrors(cudaMemcpy(d_fatVector, fatVector.values.data(), fatVectorSize *
95     sizeof(double), cudaMemcpyHostToDevice));                         // Copy input vector to
96                                         device
97 checkCudaErrors(cudaMemset(d_result, 0, resultSize * sizeof(double)));
98
99 // Perform the test multiple times
100 for (int i = 0; i < testNumber; ++i)
101 {
102     checkCudaErrors(cudaMemset(d_result, 0, resultSize * sizeof(double))); // Reset result to zero
103
104     // Start timing
105     dim3 GRID_DIM((sparseMatrix numRows + yBlockSize - 1) / yBlockSize, 1, 1);
106                                         // Grid
107                                         dimensions
108     spmvCrsKernel<<<GRID_DIM, BLOCK_DIM>>>(d_values, d_colIndices, d_rowPtr,
109     d_fatVector, d_result, sparseMatrix numRows, fatVector.numCols); // Kernel launch
110     checkCudaErrors(cudaDeviceSynchronize());
111
112     // Wait for the kernel to finish
113     checkCudaErrors(cudaGetLastError());
114
115     // Check for errors
116     checkCudaErrors(cudaMemcpy(result.values.data(), d_result, resultSize *
117         sizeof(double), cudaMemcpyDeviceToHost));                         // Copy the result back to host
118     cudaEventRecord(stop);
119
120     // Stop timing
121     checkCudaErrors(cudaEventSynchronize(stop));

```

```

91         // Wait for the stop event to be recorded
92
93         // Calculate execution time
94         float milliseconds = 0;                                // Time
95         in milliseconds
96         checkCudaErrors(cudaEventElapsedTime(&milliseconds, start, stop)); // Calculate the elapsed time
97         times[i] = milliseconds;                                // Store
98         the time
99
100    }
101
102    // Free device memory
103    cudaFree(d_values);
104    cudaFree(d_colIndices);
105    cudaFree(d_rowPtr);
106    cudaFree(d_fatVector);
107    cudaFree(d_result);
108    cudaEventDestroy(start);
109    cudaEventDestroy(stop);
110
111    // Calculate performance
112    float avgTimeMs = std::accumulate(times.begin(), times.end(), 0.0f) / times.
113    size(); // Average time
114    int NZ = sparseMatrix.values.size();                         // Number of non-zero
115    entries
116    int k = fatVector.numCols;                                 // Number of
117    columns in matrix X
118    float GFLOPS = (2.e-6 * NZ * k) / avgTimeMs;           // Performance in GFLOPS
119
120    // DISPLAY PERFORMANCE (FOR DEBUGGING PURPOSES)
121    // std::cout << "Average CUDA operation time: " << avgTimeMs << " ms" << std::
122    endl;
123    // std::cout << "Performance (CRS): " << GFLOPS << " GFLOPS" << std::endl;
124
125    return GFLOPS;
126}

```

B.D.2 ELLPACK Format

```

1 #include "matrixMultivectorProductELLPACKCUDA.h"
2 #include "../cudaUtils.cuh"
3
4 /**
5  * @brief Kernel function to multiply sparse matrix with a fat vector using ELLPACK
6  *        format on CUDA
7  *
8  * @param values Non-zero values of the sparse matrix
9  * @param colIndices Column indices of the non-zero values
10 * @param vector Fat vector
11 * @param result Result vector
12 * @param numRows Number of rows in the sparse matrix
13 * @param maxNonZerosPerRow Maximum number of non-zero values per row
14 * @param vecCols Number of columns in the fat vector
15 */
16 __global__ void spmvEllpackKernel(const double *values, const int *colIndices,
17                                   const double *vector, double *result, int numRows, int maxNonZerosPerRow, int
18                                   vecCols)
19 {
20     int row = blockIdx.x * blockDim.y + threadIdx.y; // Row index
21     int col = threadIdx.x;                        // Column index
22     // Check for valid indices
23     if (row < numRows && col < vecCols)
24     {
25         double sum = 0.0; // Initialize sum

```

```

23     // Iterate over non-zero values of the row
24     for (int i = 0; i < maxNonZerosPerRow; ++i)
25     {
26         int idx = row * maxNonZerosPerRow + i; // Index of the non-zero value
27         int colIndex = colIndices[idx];           // Column index of the non-zero
28         value
29         // Check for valid column index
30         if (colIndex != -1)
31         {
32             sum += values[idx] * vector[colIndex * vecCols + col]; // Multiply
33             and add to sum
34         }
35     }
36 }
37
38 /**
39 * @brief Multiply sparse matrix with a fat vector using ELLPACK format on CUDA
40 *
41 * @param sparseMatrix Sparse matrix in ELLPACK format
42 * @param fatVector Fat vector
43 * @param result Result vector
44 * @param testNumber Number of times to run the operation
45 * @param xBlockSize X-dimension block size
46 * @param yBlockSize Y-dimension block size
47 * @return double Performance in GFLOPS
48 */
49 double matrixMultivectorProductELLPACKCUDA(const SparseMatrixELLPACK &sparseMatrix,
50                                              const FatVector &fatVector, FatVector &result, const int testNumber, const int
51                                              xBlockSize, const int yBlockSize)
52 {
53     std::vector<float> times(testNumber); // Store time for each invocation
54
55     const dim3 BLOCK_DIM(xBlockSize, yBlockSize); // Block dimensions
56
57     cudaEvent_t start, stop; // CUDA events to measure time
58     cudaEventCreate(&start); // Create start event
59     cudaEventCreate(&stop); // Create stop event
60
61     // Memory allocations and transfers
62     double *d_values, *d_fatVector, *d_result;                                // Device memory
63     pointers for values, fat vector and result
64     int *d_colIndices;                                                        // Device memory
65     pointer for column indices
66     size_t valuesSize = sparseMatrix.values.size();                           // Size of non-
67     zero values
68     size_t colIndicesSize = sparseMatrix.colIndices.size();                  // Size of column
69     indices
70     size_t fatVectorSize = fatVector.values.size();                           // Size of fat
71     vector
72     size_t resultSize = sparseMatrix.numRows * fatVector.numCols; // Size of result
73     vector
74
75     // Allocate device memory
76     checkCudaErrors(cudaMalloc(&d_values, valuesSize * sizeof(double)));      // Allocate
77     memory for non-zero values
78     checkCudaErrors(cudaMalloc(&d_colIndices, colIndicesSize * sizeof(int)));    // Allocate memory
79     for column indices
80     checkCudaErrors(cudaMalloc(&d_fatVector, fatVectorSize * sizeof(double)));    // Allocate memory
81     for fat vector
82     checkCudaErrors(cudaMalloc(&d_result, resultSize * sizeof(double)));        // Allocate
83     memory for result
84     checkCudaErrors(cudaMemcpy(d_values, sparseMatrix.values.data(), valuesSize *
85                               sizeof(double), cudaMemcpyHostToDevice)); // Transfer non-zero

```

```

    values to device
73  checkCudaErrors(cudaMemcpy(d_colIndices, sparseMatrix.colIndices.data(),
    colIndicesSize * sizeof(int), cudaMemcpyHostToDevice)); // Transfer column
    indices to device
74  checkCudaErrors(cudaMemcpy(d_fatVector, fatVector.values.data(), fatVectorSize
    * sizeof(double), cudaMemcpyHostToDevice)); // Transfer fat vector to
    device
75  checkCudaErrors(cudaMemset(d_result, 0, resultSize * sizeof(double)));
    // Initialize
    result memory
76
77 // Run the operation multiple times
78 for (int i = 0; i < testNumber; ++i)
79 {
80     checkCudaErrors(cudaMemset(d_result, 0, resultSize * sizeof(double))); // // Reset result memory
81
82     // Perform the matrix-vector multiplication
83     cudaEventRecord(start);
84
85     // Start measuring time
86     dim3 GRID_DIM((sparseMatrix.numRows + yBlockSize - 1) / yBlockSize, 1, 1);
87
88     // Grid dimensions
89     size_t sharedMemorySize = fatVector.numCols * sizeof(double);
90
91     // Allocate shared memory dynamically
92     spmvEllpackKernel<<<GRID_DIM, BLOCK_DIM, sharedMemorySize>>>(d_values,
93         d_colIndices, d_fatVector, d_result, sparseMatrix.numRows, sparseMatrix
94         .maxNonZerosPerRow, fatVector.numCols); // Kernel launch
95     checkCudaErrors(cudaDeviceSynchronize());
96
97     // Wait for kernel to finish
98     checkCudaErrors(cudaGetLastError());
99
100    // Check for errors
101    checkCudaErrors(cudaMemcpy(result.values.data(), d_result, resultSize *
102        sizeof(double), cudaMemcpyDeviceToHost)); // //
103
104    // Transfer result to host
105    cudaEventRecord(stop);
106
107    // Stop measuring time
108    checkCudaErrors(cudaEventSynchronize(stop));
109
110    // Wait for stop event
111
112    // Calculate time execution
113    float milliseconds = 0; // Time
114        in milliseconds
115    checkCudaErrors(cudaEventElapsedTime(&milliseconds, start, stop)); // // Calculate time
116    times[i] = milliseconds; // Store time
117 }
118
119 // Free device memory
120 cudaFree(d_values);
121 cudaFree(d_colIndices);
122 cudaFree(d_fatVector);
123 cudaFree(d_result);
124 cudaEventDestroy(start);
125 cudaEventDestroy(stop);
126
127 // Calculate performance
128 float avgTimeMs = std::accumulate(times.begin(), times.end(), 0.0f) / times.
129     size(); // Average time in milliseconds
130 int NZ = sparseMatrix.values.size(); // Number of non-zero
131     entries

```

```
110     int k = fatVector.numCols;                                // Number of
111     columns in matrix X
112     float T = avgTimeMs / 1000.0f;                           // Convert to seconds
113     double FLOPS = (2.0 * NZ * k) / T;                      // Calculate FLOPS
114     double GFLOPS = FLOPS / 1e9;                             // Convert to GFLOPS
115
116     // DISPLAY PERFORMANCE (FOR DEBUGGING)
117     // std::cout << "Average CUDA operation time: " << avgTimeMs << " ms" << std::
118     // endl;
119     // std::cout << "Performance (ELLPACK): " << GFLOPS << " GFLOPS" << std::endl;
120
121     return GFLOPS;
122 }
```

Appendix B.E Utilities

```

1 #include "utils.h"
2
3 /**
4 * @brief Read a matrix from a Matrix Market file
5 *
6 * @param filename Filepath to the Matrix Market file
7 * @param matrix Sparse matrix in CRS format (output)
8 */
9 void readMatrixMarketFile(const std::string &filename, SparseMatrixCRS &matrix)
10 {
11     std::ifstream file(filename);
12     if (!file.is_open())
13     {
14         throw std::runtime_error("Unable to open file: " + filename);
15     }
16
17     std::string line;
18     bool isSymmetric = false, isPattern = false;
19     while (std::getline(file, line))
20     {
21         if (line[0] == '%')
22         {
23             if (line.find("symmetric") != std::string::npos)
24             {
25                 isSymmetric = true;
26             }
27             if (line.find("pattern") != std::string::npos)
28             {
29                 isPattern = true;
30             }
31         }
32         else
33         {
34             break; // First non-comment line reached, break out of the loop
35         }
36     }
37
38     int numRows, numCols, nonZeros;
39     std::stringstream(line) >> numRows >> numCols >> nonZeros;
40     if (!file)
41     {
42         throw std::runtime_error("Failed to read matrix dimensions from file: " +
43                               filename);
44     }
45
46     // Clearing existing data
47     matrix.values.clear();
48     matrix.colIndices.clear();
49     matrix.rowPtr.clear();
50
51     matrix.rowPtr.resize(numRows + 1, 0);
52     std::vector<std::vector<std::pair<int, double>>> tempRows(numRows);
53
54     int rowIndex, colIndex;
55     double value;
56     for (int i = 0; i < nonZeros; ++i)
57     {
58         if (isPattern)
59         {
60             file >> rowIndex >> colIndex;
61             value = 1.0; // Default value for pattern entries
62         }
63         else
64         {
65             file >> rowIndex >> colIndex >> value;
66         }

```

```

67     if (!file)
68     {
69         throw std::runtime_error("Failed to read data from file: " + filename);
70     }
71
72     rowIndex--; // Adjusting from 1-based to 0-based indexing
73     colIndex--;
74
75     tempRows[rowIndex].emplace_back(colIndex, value);
76
77     if (isSymmetric && rowIndex != colIndex)
78     {
79         tempRows[colIndex].emplace_back(rowIndex, value);
80     }
81 }
82
83 // Sort each row by column index
84 for (auto &row : tempRows)
85 {
86     std::sort(row.begin(), row.end());
87 }
88
89 // Reconstruct SparseMatrixCRS structure
90 int cumSum = 0;
91 for (int i = 0; i < numRows; ++i)
92 {
93     matrix.rowPtr[i] = cumSum;
94     for (const auto &elem : tempRows[i])
95     {
96         matrix.values.push_back(elem.second);
97         matrix.colIndices.push_back(elem.first);
98     }
99     cumSum += tempRows[i].size();
100 }
101 matrix.rowPtr[numRows] = cumSum;
102
103 matrix.numRows = numRows;
104 matrix.numCols = numCols;
105 }
106
107 /**
108 * @brief Perform the matrix-vector multiplication for a CRS matrix
109 *
110 * @param crsMatrix Sparse matrix in CRS format
111 * @param ellpackMatrix Pointer to the ELLPACK matrix (output)
112 */
113 void convertCRSToELLPACK(const SparseMatrixCRS &crsMatrix, SparseMatrixELLPACK &
114 ellpackMatrix)
115 {
116     // Calculate the average non-zeros per row to estimate initial allocation
117     int totalNonZeros = crsMatrix.values.size();
118     int avgNonZerosPerRow = totalNonZeros / crsMatrix.numRows;
119
120     // Initialize with average non-zeros per row to optimize memory usage
121     ellpackMatrix.numRows = crsMatrix.numRows;
122     ellpackMatrix.numCols = crsMatrix.numCols;
123     ellpackMatrix.maxNonZerosPerRow = avgNonZerosPerRow;
124     ellpackMatrix.values.resize(ellpackMatrix.numRows * avgNonZerosPerRow);
125     ellpackMatrix.colIndices.resize(ellpackMatrix.numRows * avgNonZerosPerRow);
126
127     for (int row = 0; row < crsMatrix.numRows; ++row)
128     {
129         int rowStart = crsMatrix.rowPtr[row];
130         int rowEnd = crsMatrix.rowPtr[row + 1];
131         int nonZerosInRow = rowEnd - rowStart;
132
133         // For rows with non-zeros less than the average, this reduces wasted space
134         for (int j = rowStart, idx = 0; j < rowEnd; ++j, ++idx)
135         {
136             int flatIndex = row * avgNonZerosPerRow + idx;

```

```

136         if (idx < avgNonZerosPerRow)
137     {
138         ellpackMatrix.values[flatIndex] = crsMatrix.values[j];
139         ellpackMatrix.colIndices[flatIndex] = crsMatrix.colIndices[j];
140     }
141 }
142 }
143 }
144
145 /**
146 * Method to compare two matrices
147 * @param mat1 First matrix
148 * @param mat2 Second matrix
149 * @param tolerance Tolerance for comparison
150 * @return bool True if the matrices are equal, false otherwise
151 */
152 bool areMatricesEqual(const FatVector &mat1, const FatVector &mat2, double
153 tolerance)
154 {
155     if (mat1 numRows != mat2 numRows || mat1 numCols != mat2 numCols)
156     {
157         return false;
158     }
159
160     for (size_t i = 0; i < mat1 values.size(); ++i)
161     {
162         if (fabs(mat1 values[i] - mat2 values[i]) > tolerance)
163         {
164             return false;
165         }
166     }
167
168     return true;
169 }
170
171 /**
172 * @brief Generate a fat vector
173 *
174 * @param fatVector Pointer to the fat vector
175 * @param numRows Number of rows in the fat vector
176 * @param numCols Number of columns in the fat vector
177 */
178 void generateFatVector(FatVector &fatVector, int numRows, int numCols)
179 {
180     fatVector numRows = numRows;
181     fatVector numCols = numCols;
182     fatVector values.resize(numRows * numCols);
183
184     for (size_t i = 0; i < fatVector values.size(); ++i)
185     {
186         fatVector values[i] = static_cast<double>(rand()) / RAND_MAX; // Valeurs
187         aleatoires entre 0 et 1
188     }
189 }
```

Appendix B.F Main Program

B.F.1 OpenMP

```

1 #include "utils.h"
2 #include "MatrixDefinitions.h"
3 #include "CRS/matrixMultivectorProductCRS.h"
4 #include "CRS/matrixMultivectorProductCRSOpenMP.h"
5 #include "ELLPACK/matrixMultivectorProductELLPACK.h"
6 #include "ELLPACK/matrixMultivectorProductELLPACKOpenMP.h"
7 #include <iostream>
8 #include <vector>
9 #include <string>
10 #include <fstream>
11
12 int main()
13 {
14     // List of the filepaths to the matrices to test
15     std::vector<std::string> filepaths = {
16         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/cop20k_A.mtx",
17         ",",
18         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/adder_dcop_32.mtx",
19         ",",
20         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/bcsstk17.mtx",
21         ",",
22         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/af_1_k101.mtx",
23         ",",
24         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/af23560.mtx",
25         ",",
26         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/amazon0302.mtx",
27         ",",
28         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/cant.mtx",
29         ",",
30         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/cavity10.mtx",
31         ",",
32         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/cage4.mtx",
33         ",",
34         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/dc1.mtx",
35         ",",
36         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/Cube_Coup_dt0.mtx",
37         ",",
38         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/FEM_3D_thermal1.mtx",
39         ",",
40         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/mac_econ_fwd500.mtx",
41         ",",
42         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/mcfe.mtx",
43         ",",
44         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/mhd4800a.mtx",
45         ",",
46         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/ML_Laplace.mtx",
47         ",",
48         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/nlpkkt80.mtx",
49         ",",
50         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/olafu.mtx",
51         ",",
52         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/PRO2R.mtx",
53         ",",
54         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/raefsky2.mtx",
55         ",",
56         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/rdist2.mtx",
57         ",",
58         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/thermal1.mtx",
59         ",",
60         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/thermal2.mtx",
61         ",",
62         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/thermomech_TK.mtx",
63         ",",
64         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/west2021.mtx",
65         ",",
66         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/webbase-1M.mtx",
67         ",",
68         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/lung2.mtx",
69         ",",
70         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/olm1000.mtx",
71         ",",
72         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/roadNet-PA.mtx"};
```

```

45     std::vector<int> ks = {1, 2, 3, 6};                                // Number of
46     columns in the fat vector
47     int iterationsNum = 20;                                         // Number of
48     iterations for the performance measurement
49     int maxThreadsNum = 16;                                         // Number of
50     threads to use
51     std::vector<int> chunkSizes = {2, 4, 8, 16, 32, 64, 128, 256}; // Chunk sizes
52     for OpenMP
53
54     // Iterate over the matrices
55     for (const auto &filepath : filepaths)
56     {
57         SparseMatrixCRS sparseMatrixCRS;           // Matrix in CRS format
58         SparseMatrixELLPACK sparseMatrixELLPACK; // Matrix in ELLPACK format
59
60         // Read the matrix from the file and prepare the structures
61         readMatrixMarketFile(filepath, sparseMatrixCRS);           // Read the
62         matrix from the file
63         convertCRSToELLPACK(sparseMatrixCRS, sparseMatrixELLPACK); // Convert the
64         matrix to ELLPACK format
65
66         // Iterate over the number of columns in the fat vector
67         for (const int &k : ks)
68         {
69             // Generate the FatVector
70             FatVector fatVector;
71             generateFatVector(fatVector, sparseMatrixCRS numRows, k); // Generate
72             the fat vector
73
74             // Iterate over the number of threads
75             for (int threads = 1; threads <= maxThreadsNum; threads++)
76             {
77                 try
78                 {
79                     FatVector resultCRS, resultELLPACK; // Results
80
81                     // Perform the matrix-vector multiplication (CRS - Serial)
82                     resultCRS.values.resize(sparseMatrixCRS numRows * k, 0.0);
83                                         // Resize
84                     the result vector
85                     resultCRS.numCols = k;
86
87                     // Set the number of columns
88                     resultCRS numRows = sparseMatrixCRS numRows;
89
90                     // Set the number of rows
91                     double performance_crs = matrixMultivectorProductCRS(
92                         sparseMatrixCRS, fatVector, resultCRS, iterationsNum); // Perform the matrix-vector multiplication
93
94                     // Perform the matrix-vector multiplication (ELLPACK - Serial)
95                     resultELLPACK.values.resize(sparseMatrixELLPACK numRows * k,
96                         0.0);
97                                         // Resize
98                     the result vector
99                     resultELLPACK.numCols = k;
100
101                     // Set the number of columns
102                     resultELLPACK numRows = sparseMatrixELLPACK numRows;
103
104                     // Set the number of rows
105                     double performance_ellpack = matrixMultivectorProductELLPACK(
106                         sparseMatrixELLPACK, fatVector, resultELLPACK,
107                         iterationsNum); // Perform the matrix-vector multiplication
108
109                     // Iterate over the chunk sizes
110                     for (const int &chunkSize : chunkSizes)
111                     {
112                         FatVector resultCRSOpenMP, resultELLPACKOpenMP; // Results
113
114                         // Perform the matrix-vector multiplication (OpenMP)
115                         resultCRSOpenMP = matrixMultivectorProductCRS(
116                             sparseMatrixCRS, fatVector, resultCRS, iterationsNum);
117
118                         resultELLPACKOpenMP = matrixMultivectorProductELLPACK(
119                             sparseMatrixELLPACK, fatVector, resultELLPACK,
120                             iterationsNum);
121
122                         // Compare the results
123                         if (resultCRSOpenMP != resultELLPACKOpenMP)
124                             std::cout << "Error: Results do not match" << std::endl;
125
126                         // Release memory
127                         resultCRSOpenMP = resultELLPACKOpenMP = resultCRS =
128                         resultELLPACK = resultCRSOpenMP = resultELLPACKOpenMP = null;
129
130                     }
131
132                 }
133             }
134         }
135     }
136
137     // Release memory
138     filepaths = sparseMatrixCRS = sparseMatrixELLPACK = fatVector =
139     resultCRS = resultELLPACK = resultCRSOpenMP = resultELLPACKOpenMP = null;
140
141 
```

```

91         // Perform the matrix-vector multiplication (CRS - OpenMP)
92         resultCRSOpenMP.values.resize(sparseMatrixCRS.numRows * k,
93                                         0.0);
94
95         // Resize the result vector
96         resultCRSOpenMP.numCols = k;
97
98         // Set the number of columns
99         resultCRSOpenMP.numRows = sparseMatrixCRS.numRows;
100
101        // Set the number of rows
102        double performance_crs_openmp =
103            matrixMultivectorProductCRSOpenMP(sparseMatrixCRS,
104                                              fatVector, resultCRSOpenMP, iterationsNum, threads,
105                                              chunkSize); // Perform the matrix-vector multiplication
106                                              (CRS - OpenMP)
107
108        // Perform the matrix-vector multiplication (ELLPACK -
109        // OpenMP)
110        resultELLPACKOpenMP.values.resize(sparseMatrixELLPACK.
111                                         numRows * k, 0.0);
112
113        // Resize the result vector
114        resultELLPACKOpenMP.numCols = k;
115
116        // Set the number of columns
117        resultELLPACKOpenMP.numRows = sparseMatrixELLPACK.numRows;
118
119        // Set the number of rows
120        double performance_ellpack_openmp =
121            matrixMultivectorProductELLPACKOpenMP(
122                sparseMatrixELLPACK, fatVector, resultELLPACKOpenMP,
123                iterationsNum, threads, chunkSize); // Perform the
124                matrix-vector multiplication (ELLPACK - OpenMP)
125
126        // Compare the results
127        bool crsResultsEqual = areMatricesEqual(resultCRS,
128                                                resultCRSOpenMP, 1e-6);           // Compare the
129                                                results (CRS)
130        bool ellpackResultsEqual = areMatricesEqual(resultELLPACK,
131                                                resultELLPACKOpenMP, 1e-6); // Compare the results (
132                                                ELLPACK)
133
134        // Print the results
135        int rowsNb = sparseMatrixCRS.numRows;           // Number of
136                                                rows in the matrix
137        int nonZeroNb = sparseMatrixCRS.values.size(); // Number of
138                                                non-zero values
139        std::cout << "m: " << rowsNb << ", k: " << k << ", NZ: " <<
140        nonZeroNb << ", threads: " << threads << ", chunk size
141        : " << chunkSize << ", CRS: " << crsResultsEqual << ",
142        ELLPACK: " << ellpackResultsEqual << ", CRS performance
143        : " << performance_crs_openmp << ", ELLPACK performance
144        : " << performance_ellpack_openmp << ", Serial CRS
145        performance: " << performance_crs << ", Serial ELLPACK
146        performance: " << performance_ellpack << std::endl;
147
148        // Free the memory
149        resultCRSOpenMP.values.clear();
150        resultELLPACKOpenMP.values.clear();
151    }
152
153    // Free the memory
154    resultCRS.values.clear();
155    resultELLPACK.values.clear();
156}
157
158 catch (const std::exception &e)
159 {
160     std::cerr << "An error occurred: " << e.what() << std::endl;
161}

```

```

125         }
126
127         // Free the memory
128         fatVector.values.clear();
129     }
130
131     // Free the memory
132     sparseMatrixCRS.values.clear();
133     sparseMatrixCRS.colIndices.clear();
134     sparseMatrixCRS.rowPtr.clear();
135     sparseMatrixELLPACK.values.clear();
136     sparseMatrixELLPACK.colIndices.clear();
137 }
138
139     return 0;
140 }
```

B.F.2 CUDA

```

1 #include "utils.h" // Assuming this contains necessary utility functions like
2     matrix reader, vector generator, etc.
3 #include "MatrixDefinitions.h"
4 #include "CRS/matrixMultivectorProductCRS.h"
5 #include "CRS/matrixMultivectorProductCRSCUDA.h"
6 #include "ELLPACK/matrixMultivectorProductELLPACK.h"
7 #include "ELLPACK/matrixMultivectorProductELLPACKCUDA.h"
8 #include <iostream>
9 #include <vector>
10 #include <string>
11 #include <fstream>
12
13 int main()
14 {
15     // List of filepaths to the matrices
16     std::vector<std::string> filepaths = {
17         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/cop20k_A.mtx",
18         ",",
19         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/
20             adder_dcop_32.mtx",
21         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/bcsstk17.mtx",
22         ",",
23         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/af_1_k101.
24             mtx",
25         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/af23560.mtx",
26         ",",
27         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/amazon0302.
28             mtx",
29         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/cant.mtx",
30         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/cavity10.mtx",
31         ",",
32         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/cage4.mtx",
33         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/dc1.mtx",
34         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/
35             Cube_Coup_dt0.mtx",
36         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/
37             FEM_3D_thermal1.mtx",
38         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/
39             mac_econ_fwd500.mtx",
40         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/mcfe.mtx",
41         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/mhd4800a.mtx",
42         ",",
43         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/ML_Laplace.
44             mtx",
45         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/nlpkkt80.mtx",
46         ",",
47         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/olafu.mtx",
48         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/PR02R.mtx"
49     };
50 }
```

```

35         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/raefsky2.mtx
36         ",
37         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/rdist2.mtx",
38         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/thermal1.mtx
39         ",
40         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/thermal2.mtx
41         ",
42         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/thermomech_TK.mtx",
43         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/west2021.mtx
44         ",
45         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/webbase-1M.
46         mtx",
47         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/lung2.mtx",
48         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/olm1000.mtx"
49         ,
50         "/mnt/beegfs/home/s425500/small_scale/Assignment/sparse-matrix/roadNet-PA.
51         mtx"};
52     std::vector<int> ks = {1, 2, 3, 6}; // Column number of the fat
53     vector
54     std::vector<int> xbd0ptions = {8, 16, 32, 64}; // Block size options for the x
55     -axis
56     std::vector<int> ybd0ptions = {1, 2, 4, 8, 16}; // Block size options for the y
57     -axis
58     int iterationsNum = 20; // Number of iterations for the
59     performance measurement
60
61 // Iterate over the matrices
62 for (const auto &filepath : filepaths)
63 {
64     SparseMatrixCRS sparseMatrixCRS; // Matrix in CRS format
65     SparseMatrixELLPACK sparseMatrixELLPACK; // Matrix in ELLPACK format
66
67     // Read the matrix from the file and prepare the structures
68     readMatrixMarketFile(filepath, sparseMatrixCRS); // Read the
69     matrix from the file
70     convertCRSToELLPACK(sparseMatrixCRS, sparseMatrixELLPACK); // Convert the
71     matrix to ELLPACK format
72
73     // Iterate over the number of columns in the fat vector
74     for (const int &k : ks)
75     {
76         FatVector fatVector, resultCRS, resultELLPACK; // Fat vector
77         and the results
78         generateFatVector(fatVector, sparseMatrixCRS.numCols, k); // Generate
79         the fat vector
80
81         try
82         {
83             // Perform the matrix-vector multiplication (CRS - Serial)
84             resultCRS.values.resize(sparseMatrixCRS.numRows * k, 0.0);
85             // Resize the
86             result vector
87             resultCRS.numCols = k;
88
89             // Set the number of columns
90             resultCRS.numRows = sparseMatrixCRS.numRows;
91
92             // Set the number of rows
93             double performance_crs = matrixMultivectorProductCRS(
94                 sparseMatrixCRS, fatVector, resultCRS, iterationsNum); // //
95             Perform the matrix-vector multiplication
96
97             // Perform the matrix-vector multiplication (ELLPACK - Serial)
98             resultELLPACK.values.resize(sparseMatrixELLPACK.numRows * k, 0.0);
99             // //
100            Resize the result vector
101            resultELLPACK.numCols = k;
102
103            // Set the number of columns
104        }
105    }
106}

```

```

77         resultELLPACK numRows = sparseMatrixELLPACK numRows;
78
79         // Set the number of rows
80         double performance_ellpack = matrixMultivectorProductELLPACK(
81             sparseMatrixELLPACK, fatVector, resultELLPACK, iterationsNum);
82         // Perform the matrix-vector multiplication
83
84         // Iterate over the x-axis block size options
85         for (const int &xBlockSize : xbdOptions)
86         {
87             // Iterate over the y-axis block size options
88             for (const int &yBlockSize : ybdOptions)
89             {
90                 FatVector resultCRSCuda, resultELLPACKCuda; // Results
91
92                 // Perform the matrix-vector multiplication (CRS - CUDA)
93                 resultCRSCuda.values.resize(sparseMatrixCRS numRows * k,
94                     0.0);
95
96                 // Resize the result vector
97                 resultCRSCuda.numCols = k;
98
99                 // Set the number of columns
100                resultCRSCuda numRows = sparseMatrixCRS numRows;
101
102                // Set the number of rows
103                double performance_crs_cuda =
104                    matrixMultivectorProductCRSCUDA(sparseMatrixCRS,
105                        fatVector, resultCRSCuda, iterationsNum, xBlockSize,
106                        yBlockSize); // Perform the matrix-vector
107                        multiplication (CRS - CUDA)
108
109                // Perform the matrix-vector multiplication (ELLPACK - CUDA
110                )
111                resultELLPACKCuda.values.resize(sparseMatrixELLPACK numRows
112                    * k, 0.0);
113
114                // Resize the result vector
115                resultELLPACKCuda.numCols = k;
116
117                // Set the number of columns
118                resultELLPACKCuda numRows = sparseMatrixELLPACK numRows;
119
120                // Set the number of rows
121                double performance_ellpack_cuda =
122                    matrixMultivectorProductELLPACKCUDA(sparseMatrixELLPACK
123                        , fatVector, resultELLPACKCuda, iterationsNum,
124                        xBlockSize, yBlockSize); // Perform the matrix-vector
125                        multiplication (ELLPACK - CUDA)
126
127                // Compare the results
128                bool crsResultsEqual = areMatricesEqual(resultCRS,
129                    resultCRSCuda, 1e-6); // Compare the
130                    results (CRS)
131                bool ellpackResultsEqual = areMatricesEqual(resultELLPACK,
132                    resultELLPACKCuda, 1e-6); // Compare the results (
133                    ELLPACK)
134
135                // Print the results
136                int rowsNb = sparseMatrixCRS numRows; // Number of
137                    rows in the matrix
138                int nonZeroNb = sparseMatrixCRS values.size(); // Number of
139                    non-zero values
140                std::cout << "m: " << rowsNb << ", k: " << k << ", NZ: " <<
141                    nonZeroNb << ", xBlockSize: " << xBlockSize << ",
142                    yBlockSize: " << yBlockSize << ", CRS: " <<
143                    crsResultsEqual << ", ELLPACK: " << ellpackResultsEqual
144                    << ", CRS performance: " << performance_crs_cuda << ",
145                    ELLPACK performance: " << performance_ellpack_cuda <<
146                    ", Serial CRS performance: " << performance_crs << ",

```

```
108         Serial ELLPACK performance: " << performance_ellpack <<
109         std::endl;
110
111         // Free the memory
112         resultCRSCuda.values.clear();
113         resultELLPACKCuda.values.clear();
114     }
115
116         // Free the memory
117         resultCRS.values.clear();
118         resultELLPACK.values.clear();
119     }
120
121     catch (const std::exception &e)
122     {
123         std::cerr << "An error occurred: " << e.what() << std::endl;
124     }
125
126         // Free the memory
127         fatVector.values.clear();
128     }
129
130         // Free the memory
131         sparseMatrixCRS.values.clear();
132         sparseMatrixCRS.colIndices.clear();
133         sparseMatrixCRS.rowPtr.clear();
134         sparseMatrixELLPACK.values.clear();
135         sparseMatrixELLPACK.colIndices.clear();
136     }
137
138     return 0;
139 }
```