MPS Parsing and JSON Output Report

Completed by:
Vasilis Tsavalias(aid25006)

A report submitted for MPS Parsing Project

University of Macedonia

October 2024

Abstract

This study investigates the transformation processes between files in MPS format and their matrix notation representations, evaluating the performance of various processing techniques. Utilizing Kaggle's cloud environment, we implemented single-threaded, multi-threaded (Concurrent Futures, Joblib, and Multiprocessing), and GPU-based (CUDA) methods to analyze the efficiency of transformations both from MPS to matrix notation and from matrix notation back to MPS.

The experimental results reveal that single-threaded processing consistently outperforms parallel and GPU-based approaches for smaller and moderately complex files, due to its low overhead. For larger files, certain parallel techniques demonstrated slight performance gains; however, these gains were marginal and often outweighed by the setup costs associated with inter-process communication. GPU-based processing, while theoretically advantageous, was hindered by substantial data transfer overheads, making it less suitable for these specific transformations.

This study also established foundational rules through JSON-based regular expressions for parsing and reconstructing MPS sections, enhancing the consistency of transformations. Due to time constraints, the complete reversal of intermediate MPS files back to their original format was not achieved, marking an area for future work. Additionally, custom parallel techniques and optimized GPU strategies, such as manual block allocations and dynamic memory management, are proposed as avenues for further performance optimization.

In conclusion, this research highlights the value of simplicity in processing choices for smaller tasks and suggests tailored parallel or GPU approaches for large-scale transformations. The insights gained offer a structured framework for efficient MPS transformations and guide future efforts in computational optimization.

Contents

Introduction

In this study, we explore the transformation processes between MPS files and their corresponding matrix notation, following a structured multi-step workflow. Figure 7.1 illustrates the overall process, divided into three primary stages:

- **Step 0**: Conversion from five MPS files to intermediate MPS files. This stage takes five original MPS inputs, represented by blue squares on the left, and processes each through an initial transformation, converging at an intermediate stage (shown as a green triangle).
- **Step 1**: Transformation of intermediate MPS files to matrix notation. In this central stage, the intermediate files undergo further processing (indicated by the green triangle) to generate matrix notation files, represented by gray circles.
- **Step 2**: Reverting matrix notation files back to intermediate MPS files. This final stage processes the matrix notation files back to the MPS format, completing the workflow and restoring the data to its original format.

This workflow supports both forward and reverse transformations between MPS and matrix notation formats, ensuring that data can be easily converted and reverted as needed.

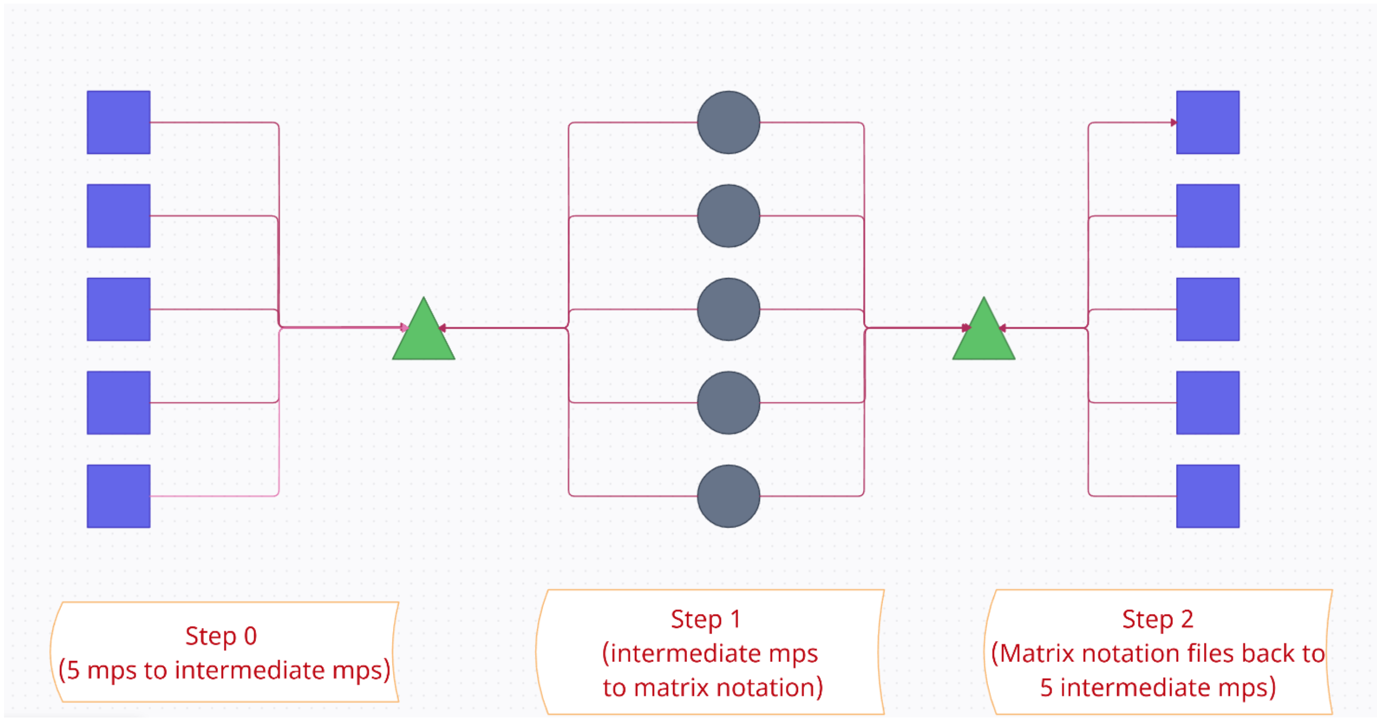

Figure 1.1: Workflow of the Transformation Process from MPS to Matrix Notation and Back

Literature Review

2.1 MPS Format

The MPS format is a structured plain text format widely adopted for representing linear programming (LP) problems. It divides the problem data into specific sections, each serving a unique role in defining the optimization problem. The `NAME` section indicates the name of the problem, while the `ROWS` section is dedicated to defining both the objective function and the constraints. In the `COLUMNS` section, the variables, along with their corresponding coefficients in the constraints and the objective function, are listed systematically.

The `RHS` section details the right-hand side values associated with the constraints, ensuring that the system is properly defined in terms of linear equalities or inequalities. An optional section, `BOUNDS`, can also appear to specify upper or lower limits on the variables, thereby adding more flexibility to the problem formulation. Finally, the `ENDATA` section marks the termination of the data, providing a clear signal for the end of the problem definition.

Each of these sections follows a strict syntactic structure, allowing for the consistent and precise representation of problems. This rigor in formatting simplifies problem sharing across different solvers but also presents challenges when it comes to parsing, given that minor deviations from the standard can lead to significant interpretation errors.

2.2 Matrix Notation

Matrix notation is a succinct and highly efficient way to express linear programming problems. In this form, an LP problem is written as minimizing the objective function, represented by the cost vector c, subject to a series of linear constraints $Ax = b$, with a non-negativity condition on the decision variables $x \geq 0$.

This notation condenses the problem into the following format:

$$\begin{aligned} \text{Minimize } & c^T x \\ \text{Subject to } & Ax = b \\ & x \geq 0 \end{aligned}$$

Here, c represents the coefficients of the objective function, while x denotes the vector of decision variables. The matrix A encapsulates the coefficients of the constraints, and the vector b holds the right-hand side values for these constraints. By expressing the problem in this format, both storage and computation become more efficient. This method is particularly well-suited for implementation in computational algorithms, where matrix operations can be performed efficiently using established mathematical libraries.

2.3 Tokenization

Tokenization refers to the process of breaking down the MPS file into smaller components for easier interpretation and processing. Each line of the MPS file is carefully read and parsed to identify keywords, variable names, coefficients, and other numerical values embedded within the file. This step is critical, as it structures the problem into a format that can be subsequently transformed into matrix notation.

Accurate tokenization ensures that all relevant data is extracted in a consistent manner, which in turn aids in the creation of matrices and vectors required for solving the optimization problem. Missteps in this phase can lead to errors in later stages of the process, making precise tokenization a vital component of MPS file parsing.

2.4 Sparse Matrices

Sparse matrices are an integral part of the efficient handling of large-scale optimization problems, where most of the elements in the matrix are zeros. This sparsity arises naturally in many real-world problems, where the number of variables and constraints is high, but the number of non-zero coefficients is relatively low. By focusing only on storing and processing the non-zero elements, sparse matrices offer significant advantages.

First, memory usage is dramatically reduced since only the non-zero values and their locations are stored, which allows for the handling of much larger problems than would be feasible with dense matrices. Furthermore, computational operations on sparse matrices are faster, as algorithms can bypass the zero elements entirely, focusing on the non-zero values. This leads to shorter execution times for many matrix operations, such as matrix-vector multiplications, which are common in optimization algorithms. Sparse matrix representations also enhance scalability, enabling the solution of larger and more complex problems by leveraging formats such as Coordinate List (COO) or Compressed Sparse Row (CSR). Each of these formats offers distinct advantages, such as simplicity in storage (COO) or efficient row-based access (CSR).

In summary, sparse matrices not only improve memory efficiency but also accelerate computation and enhance the ability to scale up optimization problems, making them a vital tool in the field of linear programming.

Methodology

3.1 Overview

The proposed approach is structured around three essential steps. First, we focus on creating an intermediate MPS format, which serves as a standardized representation that simplifies the subsequent conversion processes. This intermediate format ensures consistency and allows for easier manipulation of the problem's structure.

Next, the MPS file is converted into matrix notation. This transformation translates the problem into a more compact and computationally efficient form, enabling the use of matrix operations and algorithms. By representing the variables, constraints, and objective function as matrices and vectors, the problem becomes easier to handle programmatically.

Finally, the matrix notation is converted back into the MPS format. This reverse conversion ensures that the results of the computational processes, expressed in matrix form, are translated back into a format compatible with MPS solvers. This step closes the loop by returning the problem to its original format after it has undergone the necessary transformations and computations.

3.2 Step 0: Creating an Intermediate MPS Format

We began by analyzing the structure of several MPS files with the goal of standardizing them into a consistent intermediate format. This format would serve as a basis for further conversion, making the process easier and more efficient. Among the files reviewed, the *aircraft* MPS file stood out due to its balanced complexity and clear structure. This file was chosen as the model for our intermediate format.

The primary task involved transforming all other MPS files to align with this selected structure. This meant renaming variables, adjusting constraint prefixes, and ensuring consistent labels for objective functions and right-hand side (RHS) elements. The fixed-column format used across all MPS files required careful alignment to prevent parsing errors.

Analysis of MPS Files

Before finalizing the intermediate format, we conducted a detailed comparison of each MPS file. The main differences included variable naming conventions, constraint prefixes, and objective function labels. These elements are summarized in the following tables:

Variable Naming Conventions

In terms of variable names, the *AFIRO* file used the 'X' prefix, while files like *AIR* and *SCAGR7GX* used 'COL'. The *SC205* files, including its alternate version, used 'C'. To maintain consistency and ensure simpler parsing, we opted to standardize all variables to the 'COL' prefix, aligning with the format in the *AIR* and *SCAGR7GX* files.

File	**Variable Prefix**
AFIRO	X
AIR	COL
SC205	C
SC205 (Alt)	C
SCAGR7GX	COL

Table 3.1: Variable Naming Conventions across MPS Files

Constraint Naming Conventions

File	**Constraint Prefix**
AFIRO	R or X
AIR	ROW
SC205	R
SC205 (Alt)	R
SCAGR7GX	ROW

Table 3.2: Constraint Naming Conventions across MPS Files

The constraint naming conventions also varied. For instance, the *AFIRO* file used both ‘R‘ and ‘X‘ for constraints, while the *AIR* and *SCAGR7GX* files used the more standardized ‘ROW‘. We decided to adopt the ‘ROW‘ convention for all constraints, as it provided better clarity and consistency.

Objective Function Labels

File	**Objective Label**
AFIRO	COST
AIR	ROWOBJ
SC205	COST
SC205 (Alt)	COST
SCAGR7GX	ROWOBJ

Table 3.3: Objective Function Labels across MPS Files

When it came to objective function labels, files like *AFIRO* and *SC205* used ‘COST‘, while *AIR* and *SCAGR7GX* used ‘ROWOBJ‘. We standardized on ‘ROWOBJ‘ as it is more descriptive of the objective function’s role within the structure of the file.

Right-Hand Side (RHS) Labels

RHS labels were another area of divergence. While the majority of files used ‘RHS‘, the *AFIRO* file opted for ‘B‘. For the sake of consistency and to avoid confusion during parsing, we chose ‘RHS‘ as the standard label across all files.

File	RHS Label
AFIRO	B
AIR	RHS
SC205	RHS
SC205 (Alt)	RHS
SCAGR7GX	RHS

Table 3.4: RHS Labels across MPS Files

Row Type Indicators

All MPS files followed a standard row type indicator system, with 'N' representing the objective function and 'E', 'L', and 'G' used for equality, less-than-or-equal-to, and greater-than-or-equal-to constraints, respectively. This consistent use of row types simplified the standardization process.

Unique Features and Challenges

Certain files presented unique features. For example, the *SCAGR7GX* file introduced greater-than-or-equal-to constraints ('G' row type), while the *SC205* and its alternate version shared the same name but contained different data, reflecting different instances or versions. These distinctions were carefully managed to ensure accurate conversion.

Selection of the Optimal MPS Format

To select the most cost-effective MPS structure, we considered both the parsing efficiency and the effort required to modify each file to match the desired format. The following table summarizes our findings:

Structure	Parsing Efficiency Score	Modification Effort	Total Cost
File 2/5 (AIR/SCAGR7GX)	1	Medium	Low
File 1 (AFIRO)	2	High	High
File 3/4 (SC205)	2	Medium	Medium

Table 3.5: Cost Comparison for Different MPS File Structures

The structure used in *AIR* and *SCAGR7GX* proved to be the most efficient, both in terms of parsing time and modification effort. The clear, consistent naming conventions and labels reduced the computational burden associated with parsing these files. Moreover, aligning the other files to this structure required fewer modifications compared to the alternative approaches.

By standardizing on the *AIR/SCAGR7GX* format, we minimized the overall CPU time needed for parsing and reduced the complexity of the modification process. This approach also allowed for better scalability, as any future files added to the system would require fewer changes to fit this structure.

Conclusion

The intermediate MPS format adopted from the *AIR* and *SCAGR7GX* files provides the best balance of efficiency and ease of use. This structure's consistent variable and constraint naming conventions, along with standardized objective function and RHS labels, make it the ideal choice for further conversions. The decision to standardize on this format not only reduces CPU usage during parsing but also streamlines the modification process, ensuring a robust and scalable system moving forward.

3.3 Step 1: Converting MPS to Matrix Notation

The first step in the process involved translating MPS files into matrix notation, which required a combination of parsing, sparse matrix representation, and optimization techniques for efficient processing. This translation was structured to ensure both the accuracy of data extraction and optimal performance when handling large-scale MPS files.

Initially, the MPS files were processed line by line to extract essential components. Each MPS file contains a problem name, objective function, and a series of constraints, which need to be transformed into their corresponding matrix form. During this process, variable names, constraint names, and coefficient values were extracted from the different sections of the MPS file, such as `ROWS`, `COLUMNS`, and `RHS`. Special attention was given to separating the objective function row from the constraint rows, as they represent different aspects of the optimization problem.

Once the necessary data was extracted, the coefficient matrix A, which represents the linear constraints, was constructed. A sparse matrix format, specifically the Coordinate list (COO) format, was used for efficient storage of non-zero elements, ensuring that memory usage was optimized even for large problems with many zero entries. The sparse representation allowed for direct storage of only the non-zero elements, along with their corresponding row and column indices, thus reducing computational overhead when performing matrix operations.

In addition to the matrix A, vectors were created for the objective function coefficients c, which represent the weights of the variables in the objective function, and for the right-hand side values b, which define the constraints' bounds. To handle the nature of the constraints, an additional vector $Eqin$ was constructed, where equality and inequality constraints were mapped to numerical values, based on their type (`E` for equality, `L` for less-than or equal-to, and `G` for greater-than or equal-to). This mapping facilitated the subsequent optimization steps, as the system could process these constraints uniformly by converting them into a standardized numeric form.

To further enhance the performance of the MPS-to-matrix conversion, optimization techniques were employed. Both single-threaded and parallel processing methods were implemented, with the latter leveraging multi-core architectures to reduce processing time. Moreover, in some cases, GPU-based acceleration (CUDA) was used to offload intensive matrix operations to the GPU, enabling faster execution. Execution times for each method were measured to compare their efficiency, and the results provided valuable insights into the scalability of the process. For instance, when dealing with very large MPS files, parallel processing showed significant improvements over single-threaded execution, while GPU processing (when available) offered even greater acceleration.

By employing these techniques, the process of converting MPS files into matrix notation became not only more efficient but also scalable to accommodate the increasing complexity of real-world optimization problems. The next steps will involve utilizing these matrices and vectors for the optimization algorithms applied in subsequent phases of the project.

3.4 Step 2: Converting Matrix Notation Back to MPS

To reverse the transformation process and convert matrix notation back to MPS format, several steps were followed to ensure accuracy and consistency with the original structure.

The first step involved reading matrix notation files. In this stage, the sparse matrix data and associated vectors were parsed from the text files, and the original data structures used in the MPS format were reconstructed. This parsing process allowed the experiment to retrieve essential elements, such as coefficients, constraints, and bounds, for reassembly.

Following the initial parsing, each section of the MPS format was reconstructed meticulously:

In the **NAME** section, the original problem name was retained to ensure that the output MPS file could be identified accurately.
For the **ROWS** section, numerical constraint types were mapped back to the appropriate MPS row types. This mapping pres
erved the structure of equality and inequality constraints as originally defined.
The **COLUMNS** section was reassembled by assigning variable coefficients based on the sparse matrix data, carefully matching variables to their respective constraints.
In the **RHS** section, right-hand side values were restored to reflect the original problem setup accurately.
For the **BOUNDS** section, default bounds were applied, except where specific bounds had been defined. These bounds were critical for maintaining the integrity of the problem constraints.

Finally, to ensure consistency across transformations, saved mappings of variable and constraint names were used to match the structure of the intermediate MPS files. Additionally, strict adherence to the MPS standard formatting rules was observed, guaranteeing that the converted files met the requirements of MPS-compatible solvers.

By following this structured approach, the process successfully reversed matrix notation back into a fully consistent MPS format, retaining all essential details from the original problem setup.

Expiremental Setup

4.1 File Structure, Hardware, and Software Configuration

The experimental setup adheres to a structured directory layout to maintain an organized workflow. This structure ensures efficient access to data and supports streamlined analysis throughout the project. Key directories include the `intermediate_mps_transformation/` folder, which contains intermediate transformation files along with folders for graphs and metrics visualization. The `matrix_notation_query_a/` directory holds files and visual outputs related to the transformation of MPS files into sparse matrix notation. For the reverse transformations, the `reverse_operation_query_b/` directory organizes files and outputs associated with converting sparse matrix notations back to MPS format.

This directory structure provides a strong foundation, allowing each stage of the experiment to proceed smoothly and efficiently.

4.1.1 Hardware and Software Configuration

The experiments were conducted on Kaggle's cloud environment, which provides access to both a multi-core CPU and a GPU. The specific hardware configuration includes an Intel Xeon CPU for high-performance parallel tasks and an NVIDIA Tesla P100 GPU, which enables acceleration of matrix operations for computationally intensive processes.

The Intel Xeon CPU was employed for single-threaded and parallel processing tasks. Single-threaded processing used one core of the CPU to minimize overhead by avoiding inter-process communication, establishing a baseline for comparing execution times across various techniques. For parallel processing, three techniques—Concurrent Futures, Joblib, and Multiprocessing—distributed tasks across multiple CPU cores.

The Concurrent Futures approach, implemented through Python's `concurrent.futures` library with a `ProcessPoolExecutor`, managed parallel execution by creating separate processes for each task. Joblib, configured with `n_jobs=-1`, provided a high-level interface for parallel processing and utilized all available CPU cores, ensuring balanced workload distribution. Python's `multiprocessing.Pool` enabled flexible parallelization by creating separate processes for each task, though it introduced additional overhead related to process creation and data sharing.

The GPU-based processing, performed on an NVIDIA Tesla P100, used the CuPy library for GPU-accelerated matrix transformations. CuPy, compatible with the NumPy interface, allowed offloading of key transformations to the GPU. The matrix `A_matrix`, along with vectors `b_vector` and `c_vector`, was converted into CuPy arrays, leveraging the GPU's processing power. Using CuPy's sparse module, the matrix `A_matrix` was stored in a Coordinate (COO) sparse format, optimizing storage for large matrices. However, transferring data to and from the GPU introduced significant overhead, which impacted performance, especially for smaller files.

The software environment for this experiment was set up with Python and included various libraries essential for both data processing and visualization. The Python Standard Library provided foundational tools, such as `os` for directory management, `time` for performance measurement, and `json` for storing execution results. Matplotlib and Seaborn were used to generate visual representations of execution times across techniques, with Seaborn's `whitegrid` style enhancing readability.

Data handling was facilitated by Pandas, which managed execution times and prepared data for plotting. The SciPy and CuPy libraries enabled efficient storage and manipulation of sparse matrices; SciPy was used for CPU-based sparse operations, while CuPy allowed similar operations on the GPU. Finally, parallel processing capabilities were provided by the Joblib, Multiprocessing, and Concurrent Futures libraries, which enabled efficient distribution of tasks across CPU cores.

This combined hardware and software configuration provided a robust environment, allowing for efficient testing of various transformation techniques and enabling a comprehensive analysis of processing performance.

4.2 (Step 0) Intermediate MPS Transformation

The first figure, **Figure 4.1**, illustrates the execution time required to transform each MPS file into an intermediate representation using single-threaded processing. This graph, labeled as `transformation_execution_times_plot.png`, highlights the variability in processing times among different files, with execution times ranging from 20.52 ms for `afiro.mps` to 415.29 ms for `scagr7-2b-64.mps`.

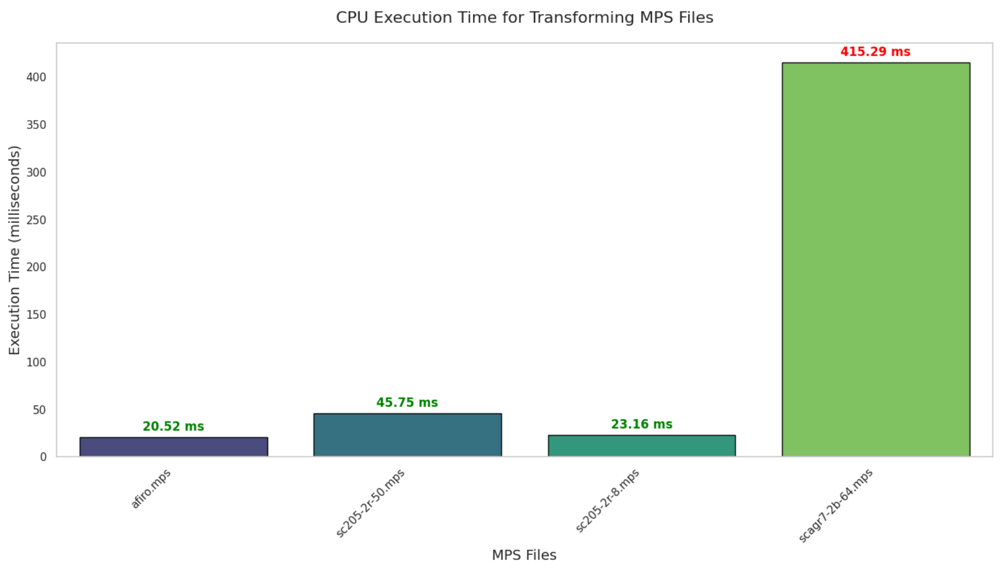

Figure 4.1: Execution time for single-threaded transformation of various MPS files into intermediate representations.

The noticeable variation in execution times is attributed to differences in file complexity, with larger or more complex MPS files, such as `scagr7-2b-64.mps`, requiring significantly more processing time. This baseline graph sets the stage for comparing parallel processing performance in subsequent transformations, demonstrating the challenges in handling larger datasets efficiently with single-threaded execution alone.

4.3 (Step 1) Matrix Notation Query A Transformation

Matrix Notation Query A involves transforming intermediate MPS files into sparse matrix notation. This section includes five individual comparison graphs, each examining the execution time for a specific file across single-threaded and four parallel processing techniques. The analysis explores whether parallel techniques enhance performance and, if so, to what extent.

4.3.1 Individual Time Comparison Graphs

The individual comparison graphs, labeled from **Figure 4.2** to **Figure 4.6**, provide insights into how each technique performs for different MPS files. The following figures correspond to each MPS file transformation:

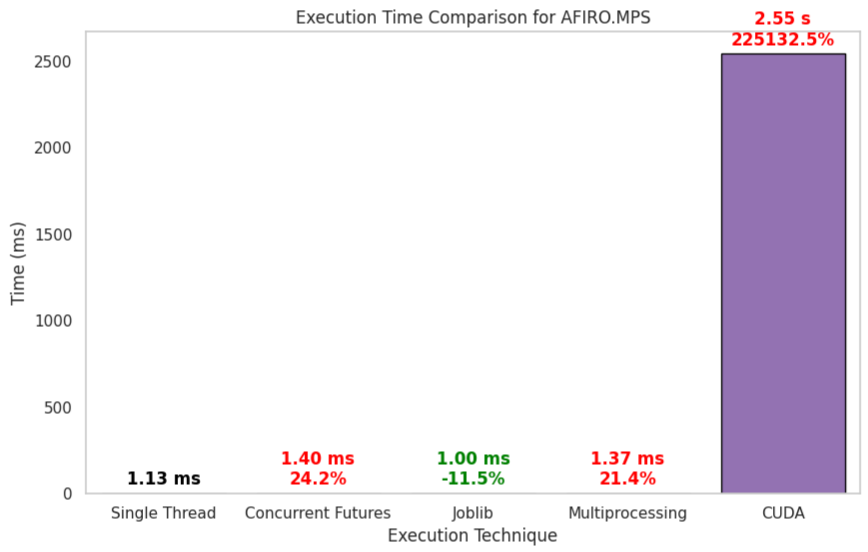

Figure 4.2: Execution time comparison for AFIRO.MPS: single-threaded vs. parallel techniques.

In **Figure 4.2**, we observe that `AFIRO.MPS` exhibits minimal performance improvement with parallel processing. The CUDA implementation, surprisingly, incurs a substantial overhead, with an execution time of 2.55 s compared to 1.13 ms for the single-threaded method, a 225,132.5% increase. This disparity suggests that the CUDA overhead outweighs its benefits for smaller, less complex files like `AFIRO.MPS`.

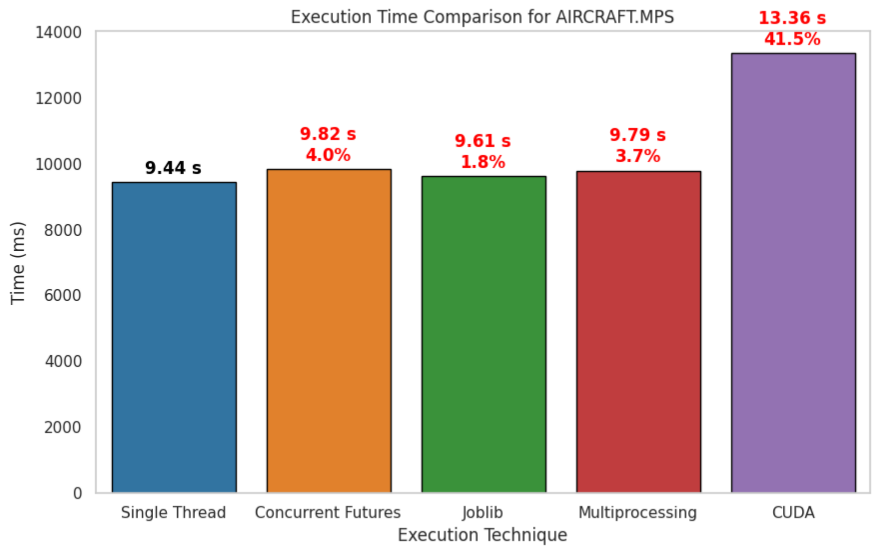

Figure 4.3: Execution time comparison for AIRCRAFT.MPS: single-threaded vs. parallel techniques.

Figure 4.3 showcases `AIRCRAFT.MPS`, where parallel processing techniques show marginal differences in execution time compared to single-threaded processing. The single-threaded approach achieves a time of 9.44 s, while CUDA takes the longest at 13.36 s, a 41.5% increase. This indicates that parallel processing may not be advantageous for moderately complex files, as overhead can offset gains.

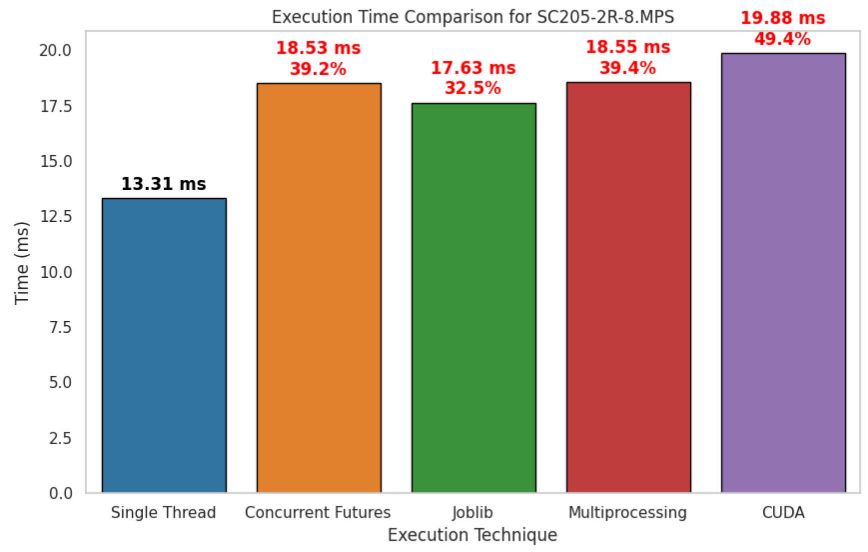

Figure 4.4: Execution time comparison for SC205-2R-8.MPS: single-threaded vs. parallel techniques.

Figure 4.4 presents the execution times for `SC205-2R-8.MPS`. Here, the single-threaded approach outperforms all parallel techniques, completing in 13.31 ms, while the CUDA method is slower at 19.88 ms, a 49.4% increase. This pattern reiterates that smaller file transformations are more efficient in single-thread mode due to parallel processing overhead.

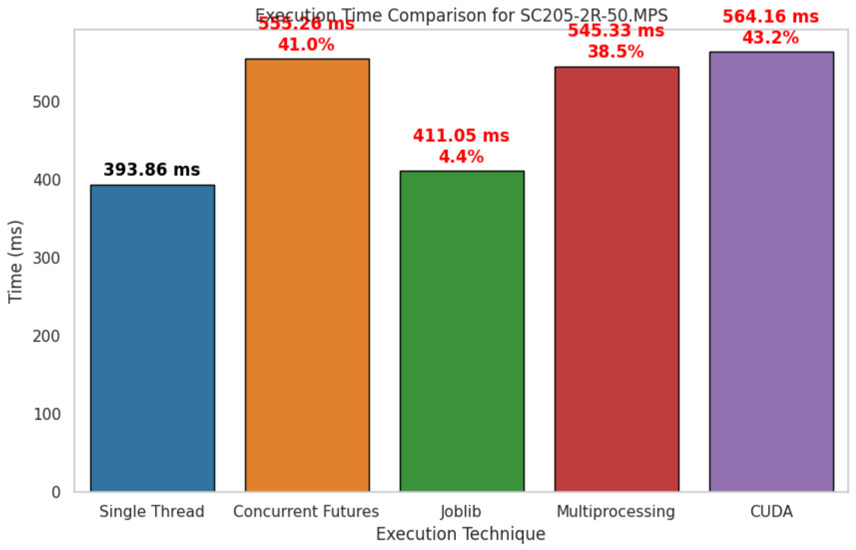

Figure 4.5: Execution time comparison for SC205-2R-50.MPS: single-threaded vs. parallel techniques.

In **Figure 4.5**, we analyze `SC205-2R-50.MPS`, where the single-threaded execution time is 393.86 ms. Here, CUDA performs notably slower, reaching 564.16 ms, a 43.2% increase. However, we observe that some parallel techniques, such as Joblib, achieve a closer execution time to single-threaded processing, suggesting that certain parallel methods may be more viable than others for medium-sized files.

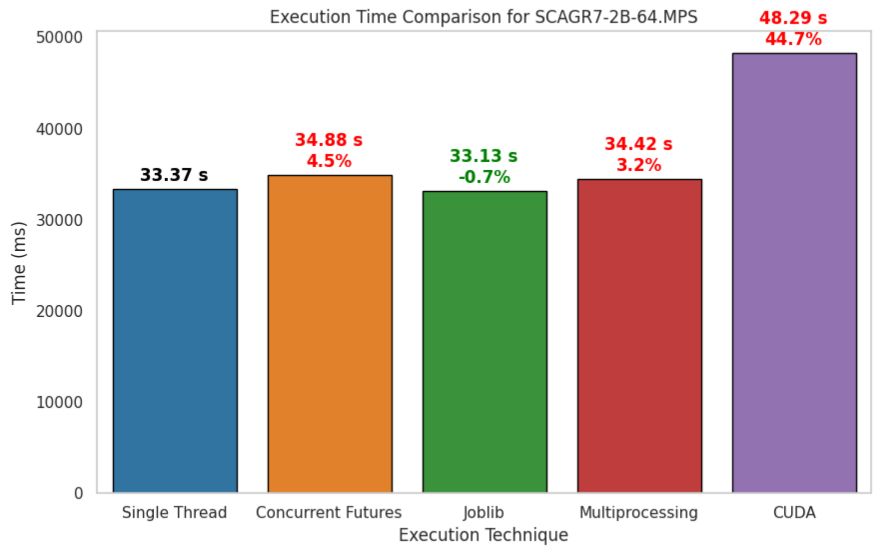

Figure 4.6: Execution time comparison for SCAGR7-2B-64.MPS: single-threaded vs. parallel techniques.

Finally, **Figure 4.6** shows `SCAGR7-2B-64.MPS`, where single-threaded execution takes 33.37 s. Parallel processing techniques such as Joblib offer a slight improvement with an execution time of 33.13 s, a 0.7% decrease. In contrast, CUDA significantly underperforms, with an execution time of 48.29 s, a 44.7% increase. This data suggests that, for larger files, only certain parallel techniques provide modest improvements, while others, like CUDA, introduce prohibitive overhead.

4.3.2 Summary of Individual Comparisons

These individual comparison graphs collectively highlight that single-threaded execution is often more efficient for smaller MPS files due to lower overhead. However, for larger files, specific parallel techniques (such as Joblib) can offer slight efficiency gains, whereas methods with higher setup costs, like CUDA, may perform worse. The findings suggest a nuanced approach to selecting the most appropriate technique based on file complexity and transformation demands.

4.3.3 Summary Graphs for Matrix Notation Query A

The following figures provide a summary of the execution time analysis for transforming intermediate MPS files into sparse matrix notation under Query A. These graphs include an all-in-one comparison, a heatmap visualization, and a counter indicating the most effective technique.

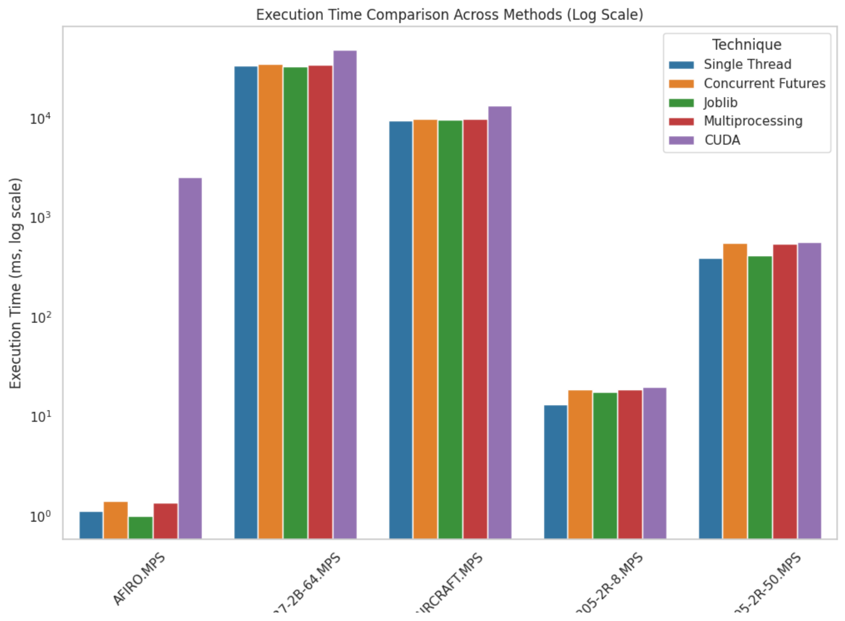

Figure 4.7: All-in-One comparison of execution times across processing techniques for Query A transformations, displayed in log scale.

Figure 4.7 shows the combined execution times for transforming MPS files under different processing techniques. The log scale helps highlight differences across techniques, with single-threaded processing consistently performing close to or better than most parallel techniques for smaller files, such as `AFIRO.MPS` and `SC205-2R-8.MPS`. However, for larger files like `SCAGR7-2B-64.MPS`, Joblib provides slightly better performance, underscoring its suitability for more complex transformations where the overhead of parallelization can be offset by gains in execution speed.

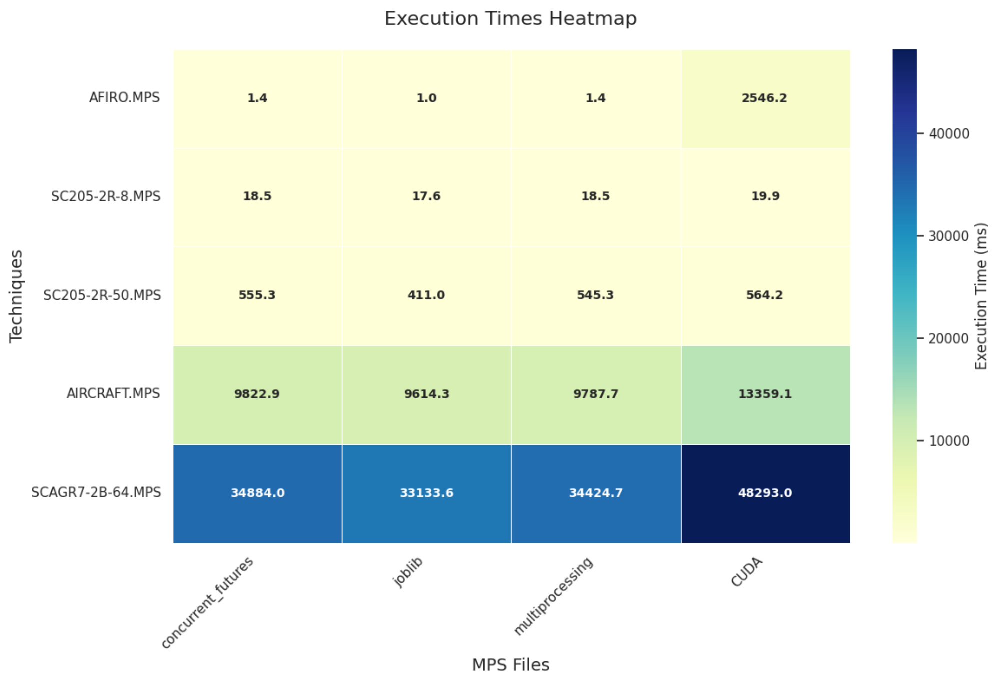

Figure 4.8: Heatmap visualization of execution times for Query A transformations, across different processing techniques and MPS files.

Figure 4.8 presents a heatmap of execution times for each MPS file and processing technique. This visualization further confirms the patterns observed in the combined comparison graph. For smaller files, single-threaded processing (yellow cells) generally yields the lowest execution times, while Joblib and Concurrent Futures show potential for larger files (green and blue shades, respectively). CUDA appears consistently slower, especially for simpler transformations, as indicated by the darkest shades in the heatmap.

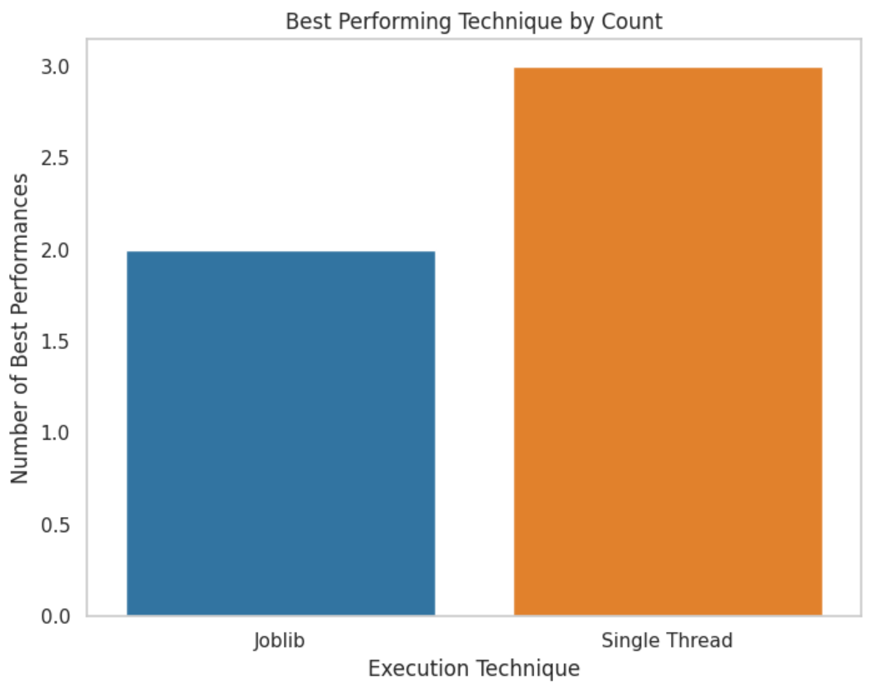

Figure 4.9: Summary of the most effective technique by count for Query A transformations.

Figure 4.9 displays a bar graph summarizing the number of best performances for each technique in Query A. Single-threaded processing leads with three best performances, followed by Joblib with two. This trend reinforces that single-threaded processing remains optimal for smaller files, whereas Joblib is competitive for larger and more complex files.

4.4 (Step 2) From matrix notation to the intermediate MPS - Query B

The following figures provide execution time comparisons for the reverse operation in Query B, which transforms sparse matrix notation files back into intermediate MPS format. Each figure shows the performance of single-threaded execution against four parallel processing techniques, allowing for a detailed analysis of efficiency and overhead.

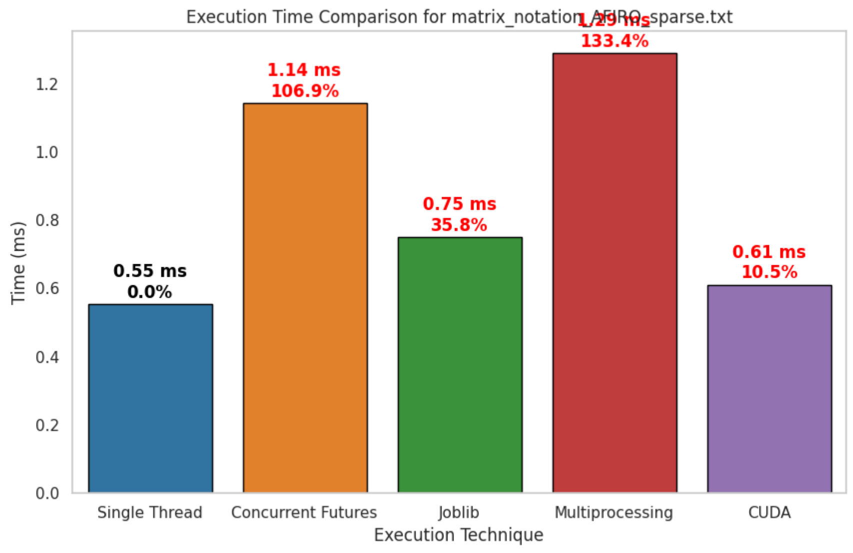

Figure 4.10: Execution time comparison for matrix_notation_AFIRO_sparse.txt: single-threaded vs. parallel techniques.

In **Figure 4.10**, which covers `matrix_notation_AFIRO_sparse.txt`, single-threaded processing achieves the fastest time at 0.55 ms. The CUDA method shows a 10.5% increase, reaching 0.61 ms, while Joblib and Concurrent Futures incur significant overhead, increasing the execution time by 35.8% and 106.9%, respectively. This figure suggests that for smaller files, single-threaded processing is optimal due to minimal overhead.

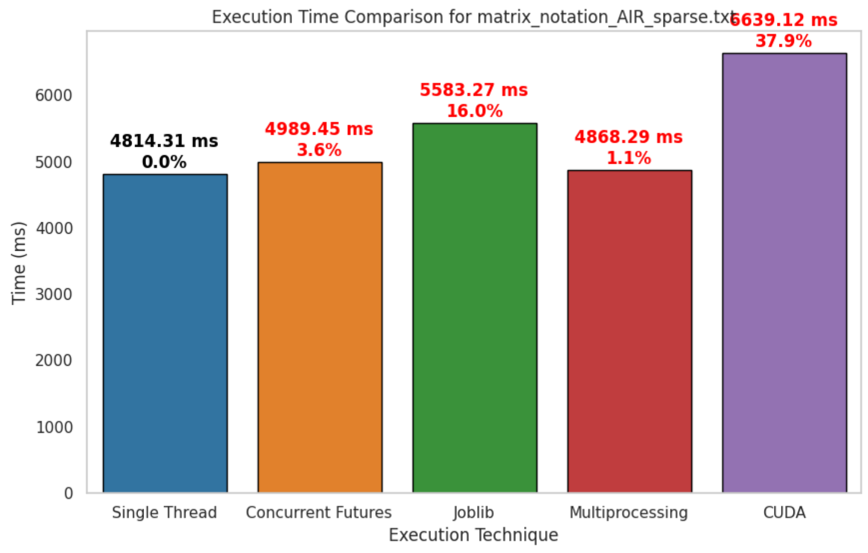

Figure 4.11: Execution time comparison for matrix_notation_AIR_sparse.txt: single-threaded vs. parallel techniques.

Figure 4.11 displays the results for `matrix_notation_AIR_sparse.txt`, where the single-threaded method completes in 4814.31 ms. The Multiprocessing approach slightly improves performance by 1.1%, while CUDA incurs a 37.9% increase in execution time. This trend suggests that for moderately complex files, single-threaded execution or Multiprocessing may be preferred over CUDA, which introduces substantial overhead.

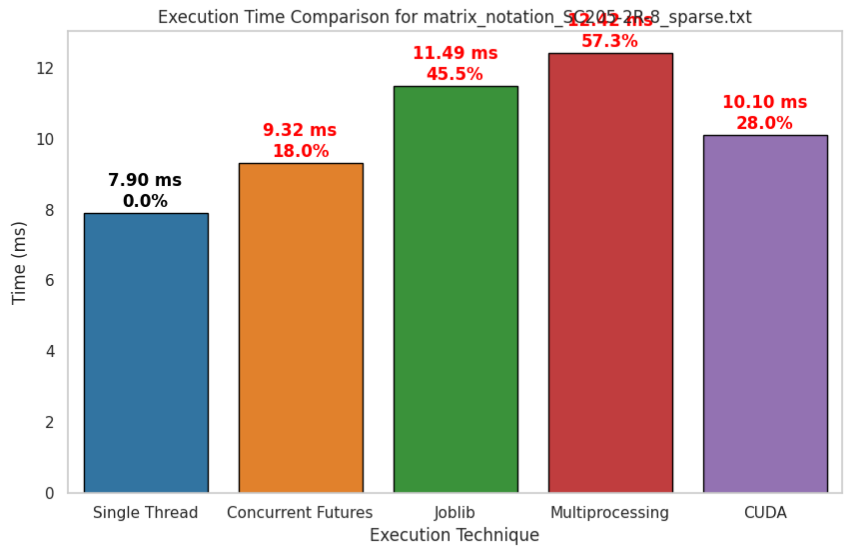

Figure 4.12: Execution time comparison for matrix_notation_SC205-2R-8_sparse.txt: single-threaded vs. parallel techniques.

In **Figure 4.12**, the execution time for `matrix_notation_SC205-2R-8_sparse.txt` is shortest with single-threaded processing at 7.90 ms. Here, Joblib and Multiprocessing demonstrate a considerable increase, with Multiprocessing reaching 57.3% more than the single-threaded time, indicating that parallel processing may not be suitable for smaller files in this transformation.

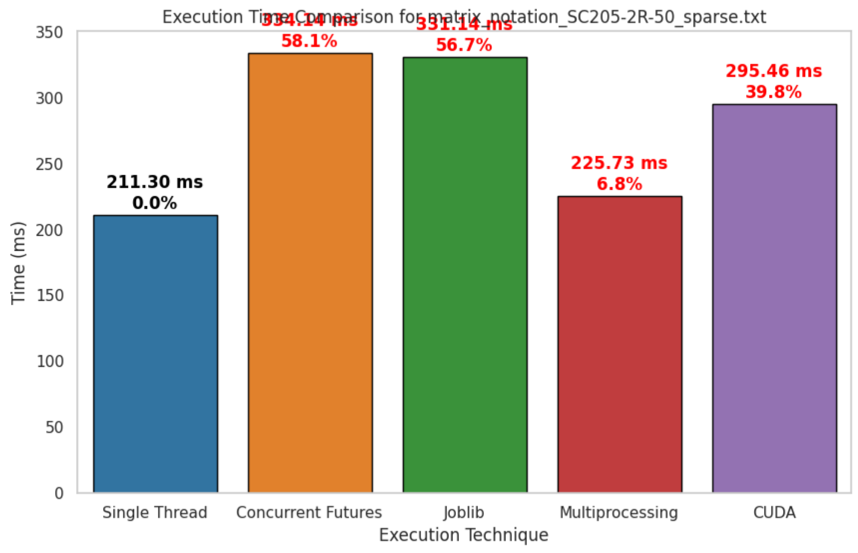

Figure 4.13: Execution time comparison for matrix_notation_SC205-2R-50_sparse.txt: single-threaded vs. parallel techniques.

Figure 4.13 examines the time for `matrix_notation_SC205-2R-50_sparse.txt`, where single-threaded execution completes in 211.30 ms. Multiprocessing shows a 6.8% increase, while Concurrent Futures incurs a 58.1% overhead. This result indicates that single-threaded execution remains efficient, even for moderately larger files, as parallel techniques add unnecessary complexity.

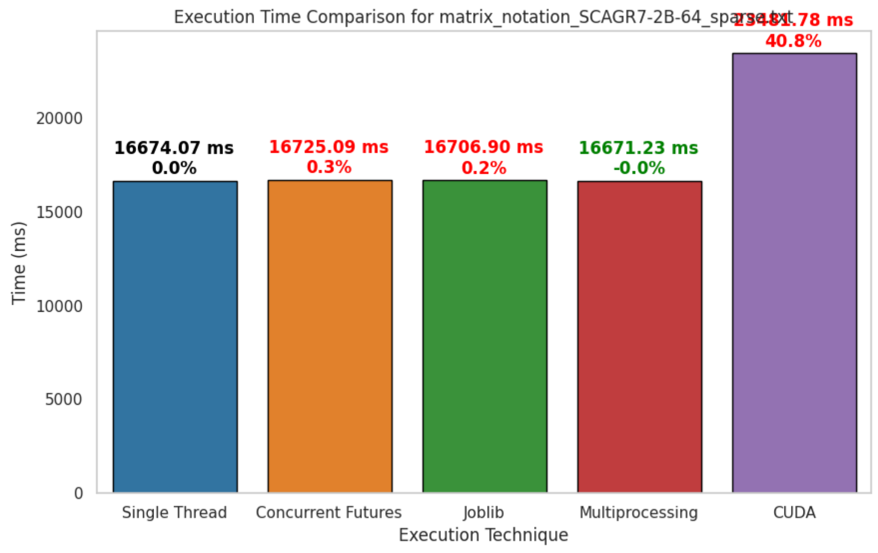

Figure 4.14: Execution time comparison for matrix_notation_SCAGR7-2B-64_sparse.txt: single-threaded vs. parallel techniques.

Finally, **Figure 4.14** shows the execution time for `matrix_notation_SCAGR7-2B-64_sparse.txt`. Here, single-threaded execution completes in 16674.07 ms, with Multiprocessing achieving a similar result, showing a negligible 0.0% change. CUDA, however, is significantly slower, with a 40.8% increase. This result suggests that single-threaded and Multiprocessing approaches are similarly effective for large, complex files, whereas CUDA is less suitable due to high overhead.

4.4.1 Summary of Individual Comparisons for Query B

The figures in this subsection demonstrate that, similar to Query A, single-threaded processing is generally optimal for smaller files due to minimal overhead. For larger files, Multiprocessing occasionally offers slight improvements. CUDA is consistently slower due to setup costs, reinforcing its unsuitability for this transformation type.

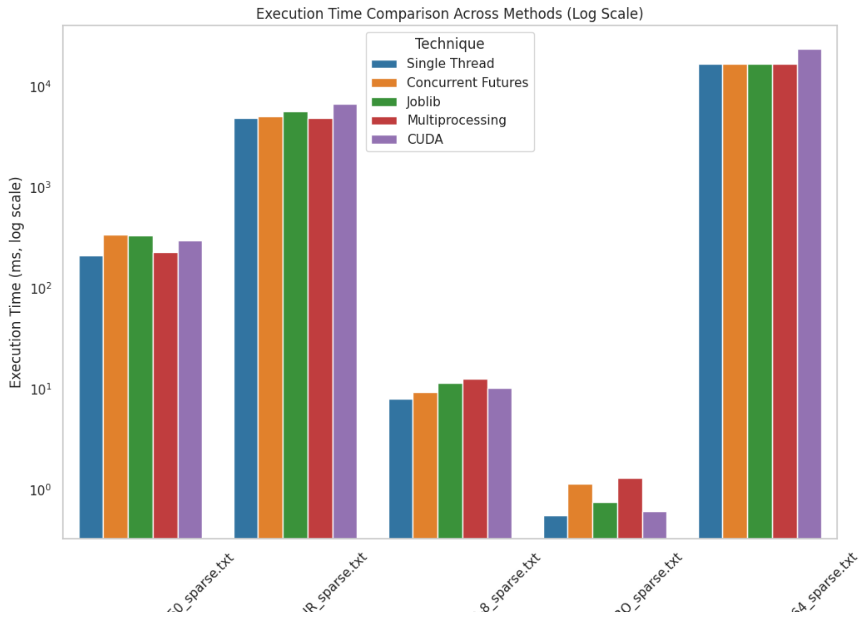

Figure 4.15: All-in-One comparison of execution times across processing techniques for Query B transformations, displayed in log scale.

In **Figure 4.15**, the combined comparison for Query B transformation times also uses a log scale, displaying consistent trends with Query A. Single-threaded execution remains efficient for smaller files, while Joblib shows minor improvements for more complex transformations. CUDA again performs poorly in this query, suggesting that it may not be suitable for these types of transformations due to high overhead.

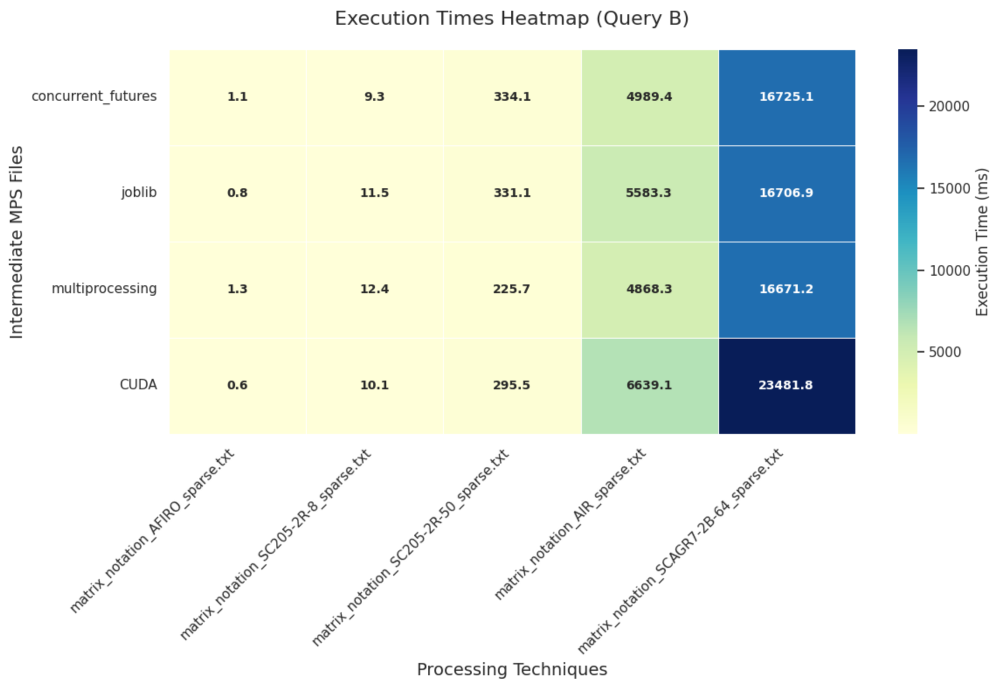

Figure 4.16: Heatmap of execution times for Query B transformations, across different processing techniques and MPS files.

Figure 4.16 shows the heatmap for Query B transformation execution times. The trends observed here align with those in Query A. Single-threaded processing demonstrates the lowest execution times for simpler files, while Joblib and Concurrent Futures perform better on complex files. CUDA, represented by the darkest cells, consistently records the highest execution times, affirming its inefficiency for these transformations.

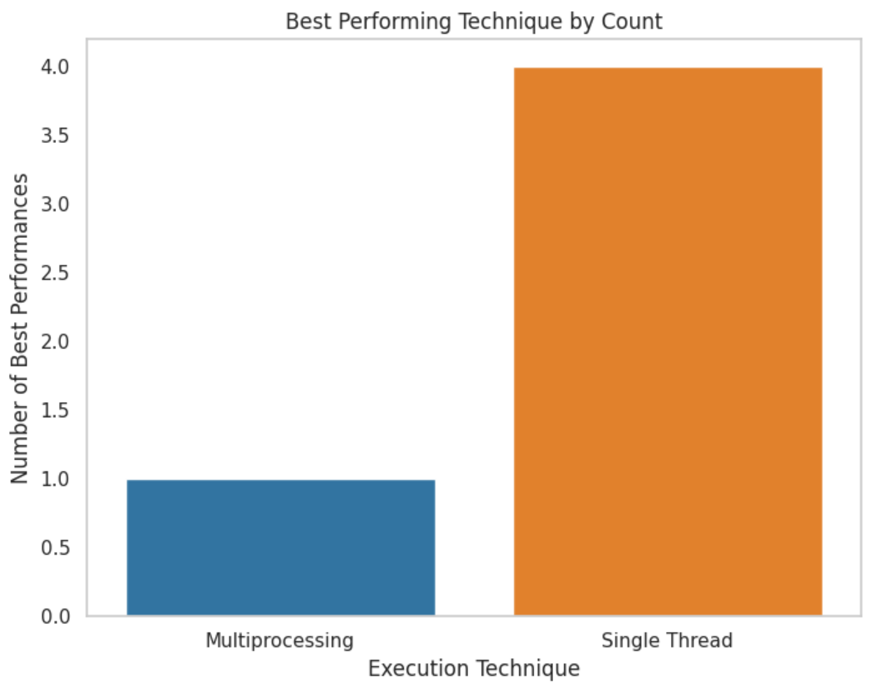

Figure 4.17: Summary of the most effective technique by count for Query B transformations.

Finally, **Figure 4.17** summarizes the best-performing techniques by count for Query B transformations. Similar to Query A, single-threaded processing leads, with four best performances, while Multiprocessing achieves one. This pattern suggests that single-threaded processing is broadly effective across both queries, with only limited scenarios where parallel processing, like Joblib, holds an advantage.

Evaluation and Results

5.1 Overview of Findings

The experiment aimed to assess the effectiveness of single-threaded processing against three parallelization techniques (Concurrent Futures, Joblib, and Multiprocessing) and one CUDA-based method for transforming MPS files to sparse matrix notation and back. Through comparative analysis across both Query A (MPS to matrix notation) and Query B (matrix notation to MPS), the results reveal distinct performance trends and underscore the importance of selecting techniques based on file complexity and processing overhead.

5.1.1 Patterns Observed in Query A and Query B

In both Query A and Query B, a clear pattern emerges: for smaller and moderately complex files, single-threaded processing consistently outperforms parallel and CUDA-based methods. This trend is primarily due to the low overhead associated with single-threaded execution, which avoids the setup and inter-process communication costs inherent in parallel processing. Conversely, for larger files, specific parallel techniques (particularly Joblib and Multiprocessing) occasionally provide slight improvements, although the gains are often marginal compared to the single-threaded approach.

The use of CUDA, while promising in theory, consistently underperforms in both queries. This underperformance is attributed to high setup costs and data transfer overheads that negate potential speed gains from GPU acceleration, especially for small and medium-sized transformations. The results suggest that for matrix transformations, the simplicity of single-threaded processing frequently outweighs the potential advantages of parallelization, particularly when processing smaller files.

5.1.2 Comparative Analysis of Techniques

Each technique evaluated in this experiment demonstrates distinct strengths and weaknesses, which can be summarized as follows:

Single-Threaded Processing: This method exhibits the most consistent performance across all file sizes, particularly excelling with smaller files. The minimal overhead involved allows it to achieve optimal execution times in the majority of transformations, both in Query A and Query B.

Concurrent Futures, Joblib, and Multiprocessing (Parallel Techniques): These techniques show slight improvements over single-threaded execution for the most complex transformations. However, the benefits are marginal and are often offset by the setup and communication costs. The results indicate that these techniques may be advantageous only for highly complex transformations or very large datasets.

CUDA: Although designed for high-performance parallel processing, CUDA consistently underperforms due to significant overhead. The data transfer and setup costs associated with GPU processing make CUDA inefficient for the file sizes and transformations involved in this study. These findings suggest that GPU acceleration may not be suitable for matrix transformations unless the data volume is exceptionally large.

5.1.3 Rationale Behind Observed Results

The observed results can be explained by the nature of matrix transformations and the inherent trade-offs between simplicity and parallelization overhead. Single-threaded processing benefits from minimal setup time and no inter-process communication, which are critical factors for smaller transformations where the processing time is brief. In contrast, parallel techniques introduce additional steps—such as data splitting, inter-process communication, and result aggregation—that increase overall execution time when the transformation size is small.

The slight improvements seen with Joblib and Multiprocessing in larger files can be attributed to their ability to divide the workload across multiple cores. However, even in these cases, the improvements are limited, indicating that the complexity of the transformation process often outweighs the benefits of parallel execution. CUDA's inefficiency, particularly pronounced in this study, emphasizes the importance of aligning processing techniques with specific workload characteristics. For transformations where overhead is high relative to processing time, simpler approaches are often more effective.

5.1.4 Philosophical Approach to Technique Selection

The findings of this experiment advocate for a pragmatic approach to technique selection. Rather than defaulting to parallel processing or GPU acceleration for all tasks, it is essential to evaluate the specific demands of each transformation. For computationally lightweight tasks, single-threaded processing offers a straightforward and efficient solution. For moderately complex tasks, limited parallelization (e.g., Multiprocessing or Joblib) may provide slight advantages, though these are context-dependent.

This study demonstrates that, in matrix transformations, adding complexity to the processing technique does not guarantee performance improvements and, in many cases, introduces unnecessary overhead. The guiding philosophy, therefore, is to match the complexity of the processing technique with the complexity of the task itself, prioritizing simplicity whenever possible to avoid the pitfalls of excessive overhead.

5.1.5 Summary of Key Results

The experiment reveals three key insights:
Single-threaded execution is optimal for small and moderately complex files, consistently outperforming parallel and CUDA-based processing due to its low overhead.
Limited parallelization can provide marginal improvements for large files, with Joblib and Multiprocessing occasionally offering better performance in highly complex transformations.
CUDA is inefficient for matrix transformations of the sizes tested, highlighting the importance of choosing processing techniques aligned with task requirements and avoiding

GPU acceleration where setup costs outweigh benefits.

These findings suggest that, for matrix transformations, simplicity often leads to better performance, especially when the transformation workload is relatively small. Future work could explore more advanced parallelization techniques for larger data volumes, but for transformations within the scope of this study, the benefits of single-threaded processing remain clear.

5.2 Future Directions

This study has established foundational rules for each JSON file, with a regular expression defined for every subsection of the MPS format. These rules facilitate the structured parsing and transformation processes by defining specific patterns for key sections like ROWS, COLUMNS, and RHS. For instance, in the file `patterns_afiro.json`, located within the `intermediate` folder under the `rules` directory, the JSON content specifies patterns for identifying subsections of the MPS file. The example configuration in `patterns_afiro.json` is as follows:

```
{
    "ROWS": "^\\s*(N|E|L|G)\\s+\\S+",
    "COLUMNS": "^\\s*\\S+\\s+\\S+\\s+[\\S\\s]+",
    "RHS": "^\\s*\\S+\\s+\\S+\\s+[\\S\\s]+"
}
```

These rules, tailored to match the structure of each MPS section, enhance the accuracy of data extraction and transformation, supporting the reproducibility and consistency of the experiment.

Due to time constraints, the reversal operation from the intermediate MPS format back to the original MPS files was not fully completed for the five selected MPS problems. Future work should aim to finalize this process to validate the efficacy of the transformation framework across diverse MPS files.

Additionally, further exploration of custom parallel techniques presents a promising direction for improving performance. One approach could involve manual block allocations, which may allow for more fine-grained control over task distribution. Techniques like dynamic allocation, combined with custom locks such as re-entrant locks, could potentially minimize contention and enhance the efficiency of parallel processing.

The same considerations apply to CUDA and GPU implementations, where custom block allocations and more sophisticated memory management strategies might yield better results. Exploring these advanced parallelization methods may lead to performance gains, particularly for large and complex transformations. This approach could enhance the adaptability and scalability of the transformation framework for broader applications in computational optimization.

In summary, the groundwork laid by this experiment offers multiple pathways for future improvement. Addressing the remaining technical challenges and optimizing parallel and GPU processing methods could significantly enhance the framework's robustness and scalability in MPS transformations.

5.3 Conclusion

This study explored the transformation processes between MPS and matrix notation formats, utilizing various parallel and GPU-based techniques to evaluate performance. Through a structured experimental setup and comprehensive analysis, we examined the efficiency of single-threaded, multi-threaded, and CUDA-based methods in both directions: converting MPS files to matrix notation and reverting matrix notation back to MPS.

The results consistently demonstrated that single-threaded processing outperformed parallel and GPU-based techniques for smaller and moderately complex files due to its minimal overhead. For larger files, specific parallel techniques, such as Joblib and Multiprocessing, occasionally provided slight improvements; however, the gains were marginal and often outweighed by setup and communication costs. GPU-based processing, though promising in theory, proved inefficient for these transformations due to significant overhead associated with data transfer and initialization. These findings highlight the importance of selecting the right processing technique based on the specific characteristics of the data and transformation task.

While the study successfully established foundational rules for handling MPS transformations and provided a structured framework for both analysis and performance evaluation, certain limitations must be acknowledged. Due to time constraints, the reversal operation from intermediate MPS files back to the original MPS format was not fully completed for all files. This partial completion points to an area for further refinement in future research. Additionally, custom parallel processing techniques, such as manual block allocations and dynamic allocation with custom locks, were identified as potential areas for performance improvement, particularly for larger and more complex transformations.

The conclusions drawn from this experiment suggest that simplicity often yields better performance in computational transformations, especially for smaller tasks where additional processing overhead can hinder performance. However, for large-scale transformations, further exploration of custom parallelization and GPU memory management techniques could lead to more substantial performance gains. Future studies should also investigate adaptive approaches that dynamically select the optimal processing method based on file complexity and size, potentially leveraging machine learning models for prediction and optimization.

In summary, this study provides a solid foundation for MPS transformations while offering insights into the limitations and opportunities for enhancement. The results underscore the need for a tailored approach to computational techniques, aligning the choice of processing method with the specific demands of the task to maximize efficiency and scalability.

References

1. **Understanding the MPS File Format**. YouTube. Available at: https://plato.asu.edu/cplex_mps.pdf
2. **Sparse Matrix Representations**. YouTube. Available at: https://www.youtube.com/watch?v=O64RH5Y9ZjU
3. **Parallel Processing in Python Playlist**. YouTube. Available at: https://www.youtube.com/watch?v=PJ4t2U15ACo&list=PLeo1K3hjS3uub3PRhdoCTY8BxMKSW7RjN

Appendices

7.1 Figures

Figure 7.1: Workflow of the Transformation Process from MPS to Matrix Notation and Back
This figure illustrates the structured workflow for converting MPS files to matrix notation and reversing them back to MPS format. Step 0 transforms five original MPS files into intermediate MPS files, Step 1 converts intermediate MPS files into matrix notation, and Step 2 reverses the matrix notation files back into the original MPS format. This bidirectional process ensures flexibility in data transformation.

Figure 7.2: Execution Time Comparison for Transforming MPS Files
This figure shows the CPU execution times for transforming various MPS files to matrix notation. Each bar represents the execution time in milliseconds for a different MPS file, with labels displaying the exact time. The figure compares single-threaded processing against four parallel processing techniques, highlighting the performance differences.

Figure 7.3: Execution Time Comparison for AFIRO.MPS
This figure provides a comparison of execution times for the AFIRO.MPS file across different processing techniques. Each bar represents a technique, with single-threaded execution as the baseline. The percentage differences are annotated, illustrating the impact of parallel and GPU processing on performance.

Figure 7.4: Execution Time Comparison for AIRCRAFT.MPS
This figure presents the execution times for transforming the AIRCRAFT.MPS file using different techniques. Single-threaded processing is compared against concurrent futures, Joblib, multiprocessing, and CUDA. The time differences are annotated, showing the percentage increase or decrease for each method.

Figure 7.5: Execution Time Comparison for SC205-2R-8.MPS
This figure illustrates the execution times for the SC205-2R-8.MPS file. The graph includes single-threaded execution as well as parallel and GPU-based techniques. Each bar shows the execution time in milliseconds, with annotations indicating performance deviations from the baseline.

Figure 7.6: Execution Time Comparison for SC205-2R-50.MPS
This figure compares the execution times for the SC205-2R-50.MPS file across single-threaded, parallel, and GPU techniques. The percentage differences in execution time are annotated above each bar to emphasize the impact of different processing methods.

Figure 7.7: Execution Time Comparison for SCAGR7-2B-64.MPS
This figure shows the execution times for the SCAGR7-2B-64.MPS file across multiple techniques. The single-threaded execution time serves as the baseline, with parallel and CUDA processing times annotated to highlight performance differences.

Figure 7.8: Best Performing Technique by Count (Query A)
This figure summarizes the best-performing technique across all MPS files for Query A. Each bar represents the count of files for which a technique provided the fastest execution time, showcasing the relative efficiency of single-threaded and parallel techniques.

Figure 7.9: Execution Time Comparison Across Methods (Log Scale, Query A)
This combined comparison graph presents the execution times for all MPS files in Query A, plotted on a log scale. The figure includes single-threaded, concurrent futures, Joblib, multiprocessing, and CUDA techniques to provide an overall view of performance variations.

Figure 7.10: Execution Times Heatmap (Query A)
This heatmap displays the execution times of each technique for all MPS files in Query A. The color gradient highlights the variations in execution time, with darker shades indicating longer processing times.

Figure 7.11: Execution Time Comparison for matrix_notation_AFIRO_sparse.txt
This figure shows the execution time for converting matrix notation back to MPS format for the file matrix_notation_AFIRO_sparse.txt. Each bar represents a different technique, with single-threaded execution as the baseline and annotations indicating the percentage difference.

Figure 7.12: Execution Time Comparison for matrix_notation_AIR_sparse.txt
This figure compares the execution time for the matrix_notation_AIR_sparse.txt file, showing the effect of single-threaded, parallel, and CUDA techniques on performance. Each bar shows the execution time, with percentage differences highlighted.

Figure 7.13: Execution Time Comparison for matrix_notation_SC205-2R-8_sparse.txt
This figure illustrates the time required to convert matrix notation to MPS for the matrix_notation_SC205-2R-8_sparse.txt file, comparing single-threaded and parallel techniques along with CUDA. Performance differences are annotated for each bar.

Figure 7.14: Execution Time Comparison for matrix_notation_SC205-2R-50_sparse.txt
This figure shows the execution time for transforming the matrix_notation_SC205-2R-50_sparse.txt file back to MPS format. It includes a comparison between single-threaded and parallel techniques, with annotations highlighting time differences.

Figure 7.15: Execution Time Comparison for matrix_notation_SCAGR7-2B-64_sparse.txt
This figure provides the execution time comparison for the matrix_notation_SCAGR7-2B-64_sparse.txt file. The figure includes single-threaded, parallel, and CUDA techniques, with annotated percentages showing the impact on execution time.

Figure 7.16: Best Performing Technique by Count (Query B)
This figure summarizes the best-performing technique across all matrix notation files for Query B. Each bar indicates the count of files where a technique was the fastest, comparing the relative performance of single-threaded and parallel approaches.

Figure 7.17: Execution Time Comparison Across Methods (Log Scale, Query B)
This combined comparison chart presents the execution times for all matrix notation files in Query B on a log scale. It provides an overview of single-threaded, concurrent futures, Joblib, multiprocessing, and CUDA performances.

Figure 7.18: Execution Times Heatmap (Query B)
This heatmap displays the execution times of each technique for all matrix notation files in Query B. The color gradient indicates variations in processing times, with darker colors representing longer durations.

7.2 Code and Reproducibility

For full reproducibility of the experiments and to allow other researchers to build upon this work, the complete code used in this project is available on Kaggle. The notebook contains all the steps from mps preprocessing to matrix notation transformation and the reverse operations.

- **Kaggle Notebook**: mps_matrix_form_and_reverse