Benchmarking Matrix-Matrix Multiplication (mmult)

Second Report: Optimization and Evaluation

Mohammed Mansour

April 21, 2025

1 Introduction

Matrix-matrix multiplication (mmult) is a fundamental operation in scientific computing, machine learning, and computer architecture. This report is a continuation of the previous report, focusing on the implementation and benchmarking of mmult. The goal is to compare a naïve implementation with an optimized version using blocking techniques to improve performance.

2 Collaboration

I discussed the details of the project with Moneer Al-Bokhaiti. We covered the project requirements, the necessary tools, and the complexity of the code. He suggested creating a separate application to generate the dataset. We also agreed that storing the dataset in a binary file would be more efficient than saving it as ASCII text.

Additionally, I attended a meeting conducted by class students, where we discussed various issues related to the project. During this meeting, we broke the implementation into smaller steps to make the development process more manageable.

In this phase, we (Me and Moneer) discussed the pseudocode of the mmult function, which is the core of the project. We also talked about the importance of optimizing the code for better performance and how the optimized version would be faster than the naïve implementation.

3 Naïve Implementation

The naïve implementation is a simple three-loop to compute the matrix product:

```
for (int i = 0; i < M; i++) {
    for (int j = 0; j < P; j++) {
        R[i][j] = 0;
        for (int k = 0; k < N; k++) {
            R[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Listing 1: Naïve mmult implementation

This method is straightforward but inefficient due to cache misses and redundant memory accesses. Each element of the matrices is accessed multiple times, leading to poor utilization of the CPU cache. Additionally, the lack of optimization techniques such as loop unrolling results in nonoptimal performance, especially for larger datasets.

4 Optimized Implementation

The optimized implementation applies a blocking strategy to improve spatial and temporal locality. By computing sub-matrices (blocks) of the result matrix, the approach minimizes cache misses by reusing loaded data efficiently.

Blocking refers to dividing the matrices into smaller square blocks or sub-matrices of size $b \times b$. Instead of computing one element of the result matrix at a time, the algorithm computes partial results over these sub-matrices and accumulates them. This drastically reduces the number of memory accesses to data not residing in cache, which is critical for performance.

The implementation chooses a stationary output sub-matrix, looping over the corresponding input sub-matrices from matrices A and B. This ensures that data is reused as much as possible before being evicted from the cache. The block size b is passed as a command-line argument, making the implementation tunable to fit different cache configurations and matrix sizes.

One key insight is that performance benefits become more significant as the matrix size increases. This is because larger matrices experience more cache misses in the naïve version, and blocking helps mitigate this by exploiting spatial locality in the B matrix and temporal locality in the A matrix.

The optimized implementation is as follows:

```c
void mmult_opt(float **A, float **B, float **R, int M, int N, int P,
    int b) {
     for (int ii = 0; ii < M; ii += b) {
         for (int jj = 0; jj < P; jj += b) {
             for (int kk = 0; kk < N; kk += b) {
                 for (int i = ii; i < fmin(ii + b, M); ++i) {
                     for (int j = jj; j < fmin(jj + b, P); ++j) {
                         float val = R[i][j];
                         for (int k = kk; k < fmin(kk + b, N); ++k) {
                             val += A[i][k] * B[k][j];
                         }
                         R[i][j] = val;
                     }
                 }
             }
         }
     }
}
```

Listing 2: Blocked mmult Implementation

Initial bugs involved uninitialized values and improper bounds when indexing. Fixes were made by ensuring the matrices were zero-initialized and carefully managing loop ranges. Additional care was taken to validate that block size does not exceed the dimensions of the matrices to avoid segmentation faults.

5 Dataset Generation

The dataset required for **both implementations** is generated using a Python script. The script creates matrices of various sizes, including testing, small, medium, large, and native datasets. The matrices are filled with random integers to simulate real-world scenarios.

The following Python script is used for dataset generation:

```python
class Dataset:
    def __init__(self, rowsA, colsA, rowsB, colsB, name):
        self.rowsA = rowsA
        self.colsA = colsA
        self.rowsB = rowsB
        self.colsB = colsB
        self.name = name

dataset = [
    Dataset(16, 12, 12, 8, "testing"),
    Dataset(121, 180, 180, 115, "small"),
    Dataset(550, 620, 620, 480, "medium"),
    Dataset(962, 1012, 1012, 1221, "large"),
    Dataset(2500, 3000, 3000, 2100, "native")
]

def generate_matrix(rows, cols):
    return np.random.rand(rows, cols).astype(np.float32)

def save_matrix_binary(filename, matrix):
    if not os.path.exists("mmult_ds"):
        os.makedirs("mmult_ds")
    with open("mmult_ds/" + filename, 'wb') as f:
        f.write(matrix.tobytes())

def main():
    parser = argparse.ArgumentParser(description='Generate and save matrices in binary format.')
    parser.add_argument('name', type=str, help='Dataset name')
    args = parser.parse_args()

    dataset_info = next((d for d in dataset if d.name == args.name), None)
    if not dataset_info:
        print(f"No dataset found with the name '{args.name}'.")
        sys.exit(1)

    rowsA, colsA, rowsB, colsB = dataset_info.rowsA, dataset_info.colsA, dataset_info.rowsB, dataset_info.colsB
    matrixA = generate_matrix(rowsA, colsA)
    matrixB = generate_matrix(rowsB, colsB)
    result_matrix = np.dot(matrixA, matrixB)

    save_matrix_binary("matrixA.bin", matrixA)
    save_matrix_binary("matrixB.bin", matrixB)
    save_matrix_binary("matrixC.bin", result_matrix)

    print("Matrices and metadata saved successfully in binary format.")
```

Listing 3: Dataset Generation Script

This script ensures efficient dataset generation by using NumPy for matrix operations and storing the results in a structured binary format. The metadata file provides dimensions for easy retrieval and compatibility with benchmarking implementations.

6 Evaluation

The evaluation was conducted by running both the naïve and optimized implementations on datasets of varying sizes. The datasets include testing, small, medium, large, and native sizes. For the optimized implementation, we performed a sweep over different block sizes (b) to explore the design space and identify the optimal block size for each dataset. The stationary sub-matrix was also varied to study its impact on performance.

Execution time was measured for each configuration, and speedup was calculated as the ratio of the execution time of the naïve implementation to that of the optimized implementation. All experiments were conducted on the same hardware setup to ensure consistency.

6.1 Experimental Setup

The experiments were conducted on **Personal Laptop** with the following setup:

- Runner Type: Ubuntu-24.04.1 LTS
- Operating System: Linux
- Processor (CPU): AMD PRO A12-9800B R7, 12 COMPUTE CORES 4C+8G
- Memory (RAM): 8 GB
- Storage (SSD): 512 GB
- Architecture: x64
- Compiler: GCC 13.3.0 with optimization flags -O3

6.2 Results

Table 1 summarizes the execution times for both the naïve and optimized implementations across different dataset sizes. The execution times are measured in milliseconds (ms) and represent the average time taken over multiple runs. The results indicate a significant performance improvement with the optimized implementation, especially for larger datasets.

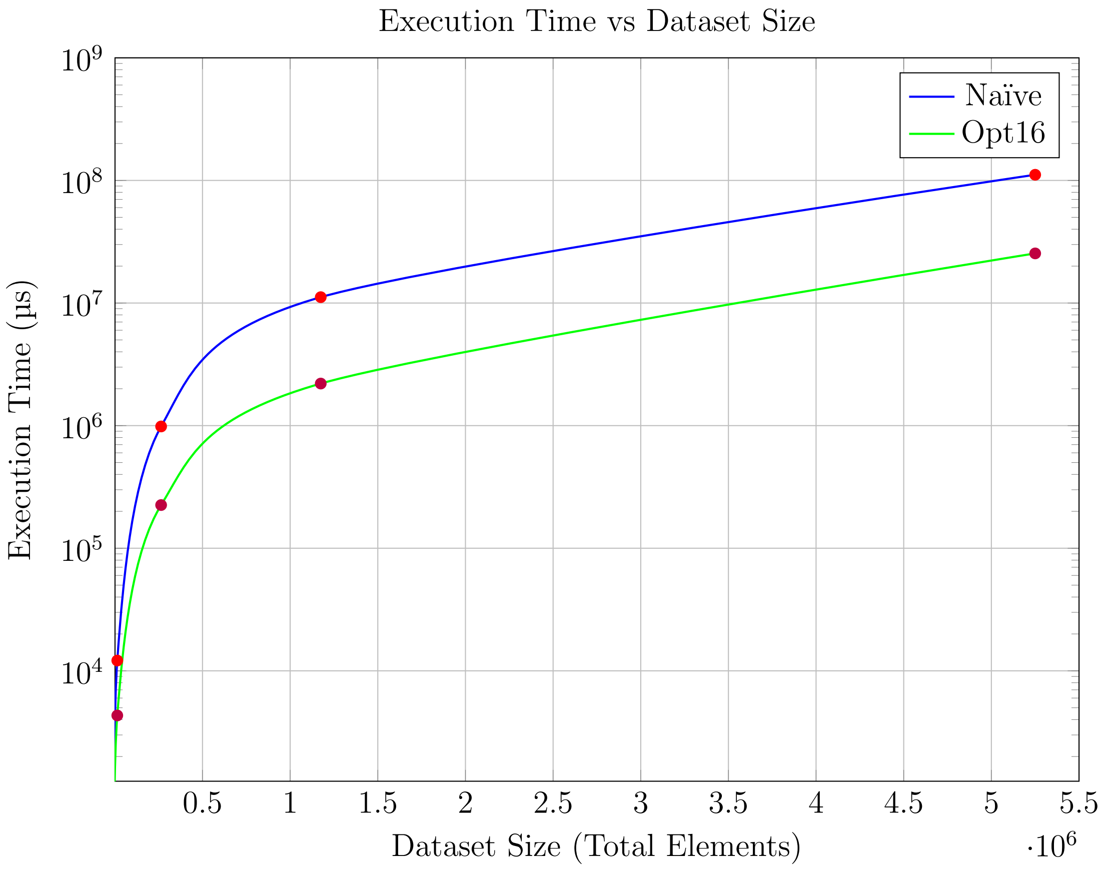
The benchmark execution logs and results can be accessed at the following link: `https://github.com/mohammed0x00/YABMS/tree/main/docs/runtime_data_opt`.

6.3 Graphical Representation

The following graph illustrates the execution time of the naïve and optimized implementations against the dataset size. The X-axis represents the dataset size in total elements, while the Y-axis shows the execution time in microseconds (µs). The graph includes both the actual data points and a smooth curve fitting the points for better visualization.

Dataset Size	**Naïve**	**Opt16**	**Opt32**	**Opt64**
Testing (16x8)	0.005	0.003	0.003	0.002
Small (121x115)	12.137	4.326	3.860	4.012
Medium (550x480)	985	225	222	233
Large (962x1221)	11167	2204	1861	1921
Native (2500x2100)	111219	25396	24536	25746

Table 1: Execution times in milliseconds (ms) for different implementations across different dataset sizes.



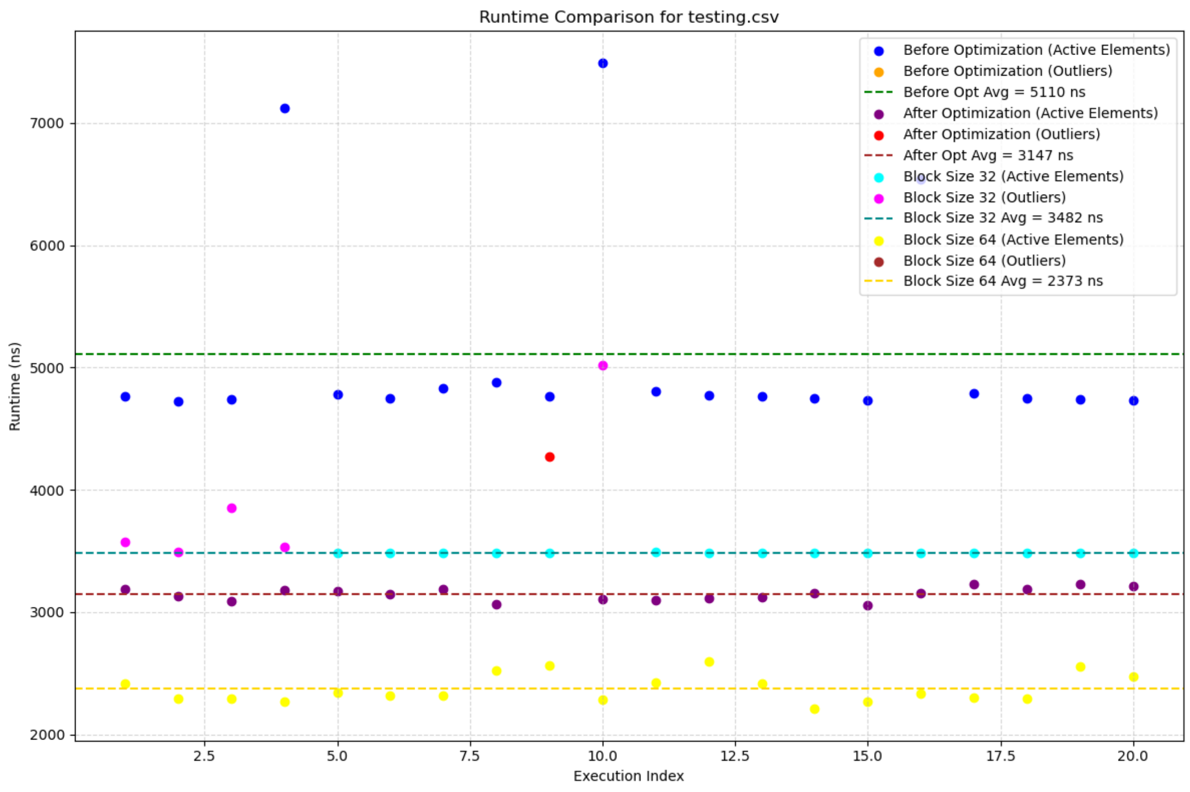
6.4 Detailed Results

The following sections provide detailed results for each dataset size, including execution times for both the naïve and optimized implementations. The results are presented in tabular format, showing the execution time in milliseconds (ms) for each configuration. The tables also include the speedup achieved by the optimized implementation compared to the naïve version.

The detailed results for each dataset size are presented below:

6.5 Testing Dataset Details

The following table summarizes the execution times for the Testing dataset. The table includes the execution time for both the naïve and optimized implementations.

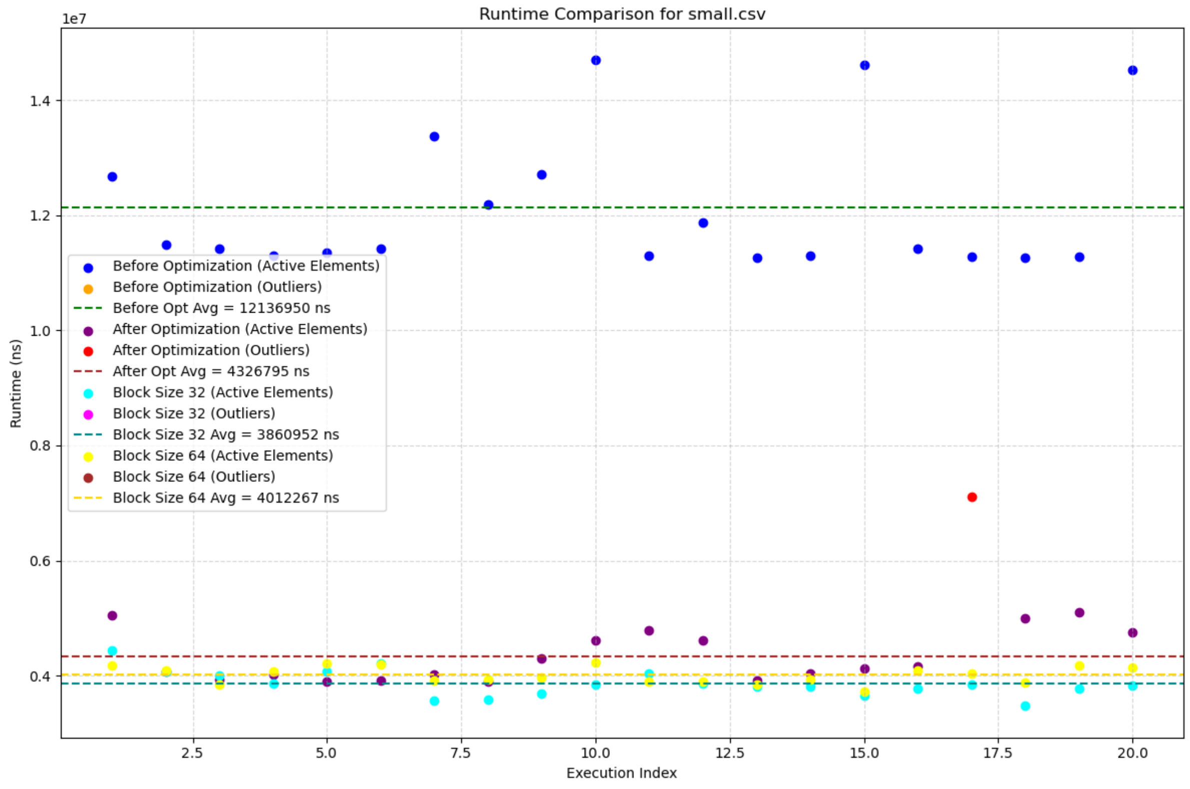


```
Setting up schedulers and affinity:
  * Setting the niceness level:
      -> trying niceness level = -20
    + Process has niceness level = -20
  * Setting up FIFO scheduling scheme and high priority ... Succeeded
  * Setting up scheduling affinity ... Succeeded

Running "scalar_opt" implementation:
  * Invoking the implementation 20 times .... Finished
  * Verifying results .... Success
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 250
      - Average = 3203
      - Number of active elements = 20
      - Number of masked-off = 1
    + Starting statistics run number #2:
      - Standard deviation = 50
      - Average = 3147
      - Number of active elements = 19
      - Number of masked-off = 0
  * Runtimes (MATCHING):  3147 ns
  * Dumping runtime informations:
    - Filename: scalar_opt_runtimes.csv
```

Listing 4: Testing Dataset Output

6.6 Small Dataset Details

The following table summarizes the execution times for the Small dataset. The table includes the execution time for both the naïve and optimized implementations.

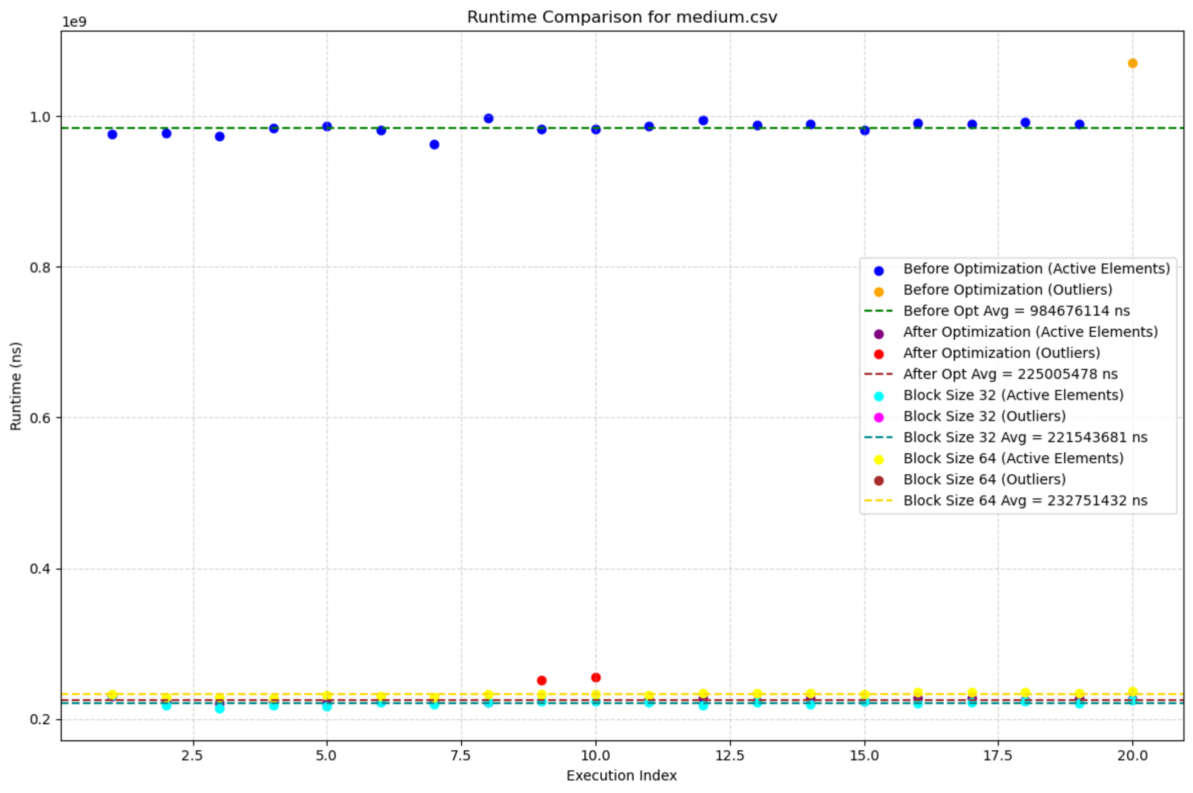


```
Setting up schedulers and affinity:
  * Setting the niceness level:
      -> trying niceness level = -20
    + Process has niceness level = -20
  * Setting up FIFO scheduling scheme and high priority ... Succeeded
  * Setting up scheduling affinity ... Succeeded

Running "scalar_opt" implementation:
  * Invoking the implementation 20 times .... Finished
  * Verifying results .... Success
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 733060
      - Average = 4466056
      - Number of active elements = 20
      - Number of masked-off = 1
    + Starting statistics run number #2:
      - Standard deviation = 421649
      - Average = 4326795
      - Number of active elements = 19
      - Number of masked-off = 0
  * Runtimes (MATCHING):  4326795 ns
  * Dumping runtime informations:
    - Filename: scalar_opt_runtimes.csv
```

Listing 5: Small Dataset Output

6.7 Medium Dataset Details

The following table summarizes the execution times for the Medium dataset. The table includes the execution time for both the naïve and optimized implementations.

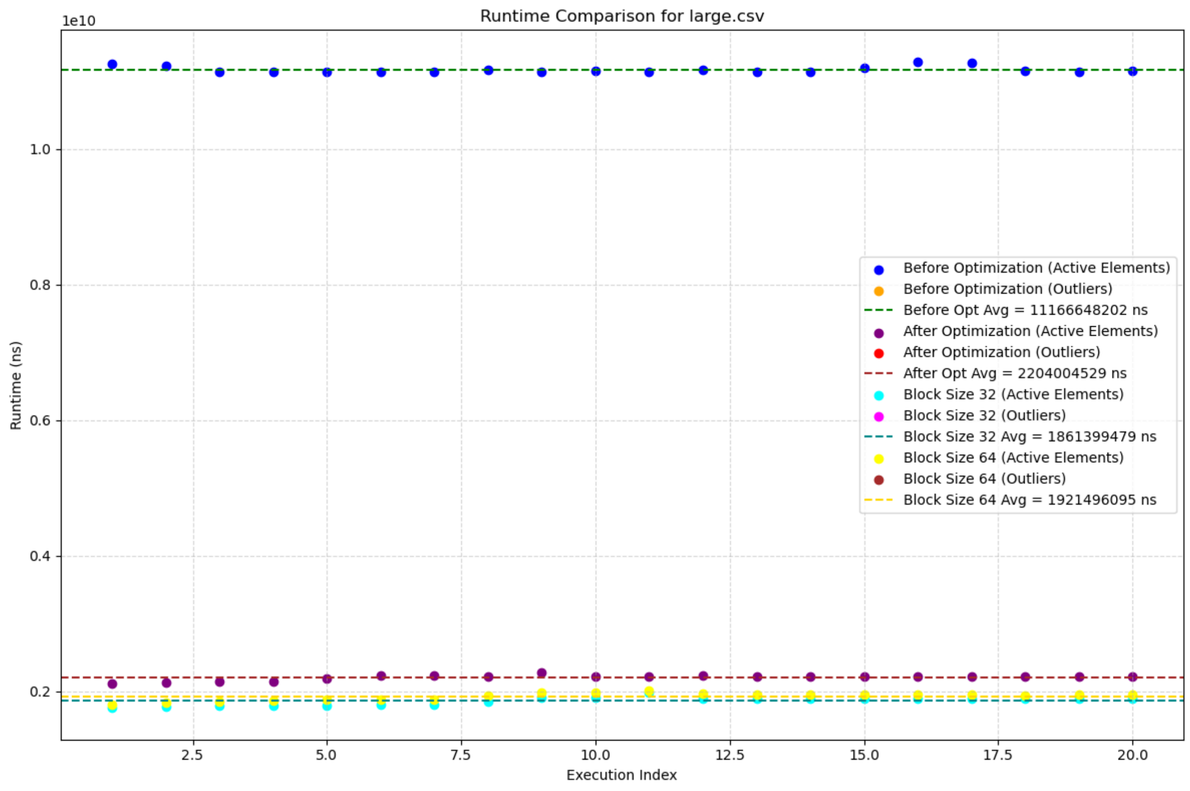


```
Running "scalar_opt" implementation:
  * Invoking the implementation 20 times .... Finished
  * Verifying results .... Success
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 8906230
      - Average = 227847343
      - Number of active elements = 20
      - Number of masked-off = 1
    + Starting statistics run number #2:
      - Standard deviation = 6446166
      - Average = 226399194
      - Number of active elements = 19
      - Number of masked-off = 1
    + Starting statistics run number #3:
      - Standard deviation = 2637246
      - Average = 225005478
      - Number of active elements = 18
      - Number of masked-off = 0
  * Runtimes (MATCHING):  225005478 ns
  * Dumping runtime informations:
    - Filename: scalar_opt_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished
```

Listing 6: Medium Dataset Output

6.8 Large Dataset Details

The following table summarizes the execution times for the Large dataset. The table includes the execution time for both the naïve and optimized implementations.

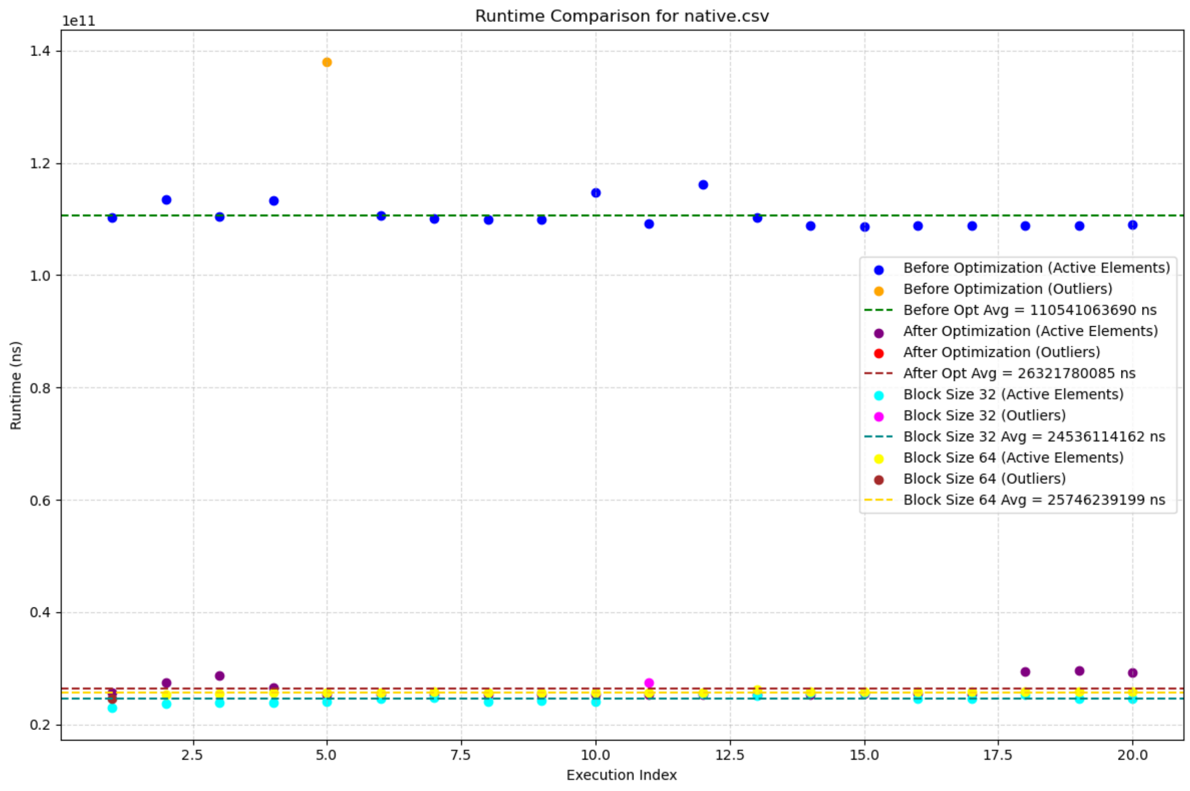


```
Setting up schedulers and affinity:
  * Setting the niceness level:
      -> trying niceness level = -20
    + Process has niceness level = -20
  * Setting up FIFO scheduling scheme and high priority ... Succeeded
  * Setting up scheduling affinity ... Succeeded

Running "scalar_opt" implementation:
  * Invoking the implementation 20 times .... Finished
  * Verifying results .... Success
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 38622316
      - Average = 2204004529
      - Number of active elements = 20
      - Number of masked-off = 0
  * Runtimes (MATCHING):  2204004529 ns
  * Dumping runtime informations:
    - Filename: scalar_opt_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished
```

Listing 7: Large Dataset Output

6.9 Native Dataset Details

The following table summarizes the execution times for the Native dataset. The table includes the execution time for both the naïve and optimized implementations.



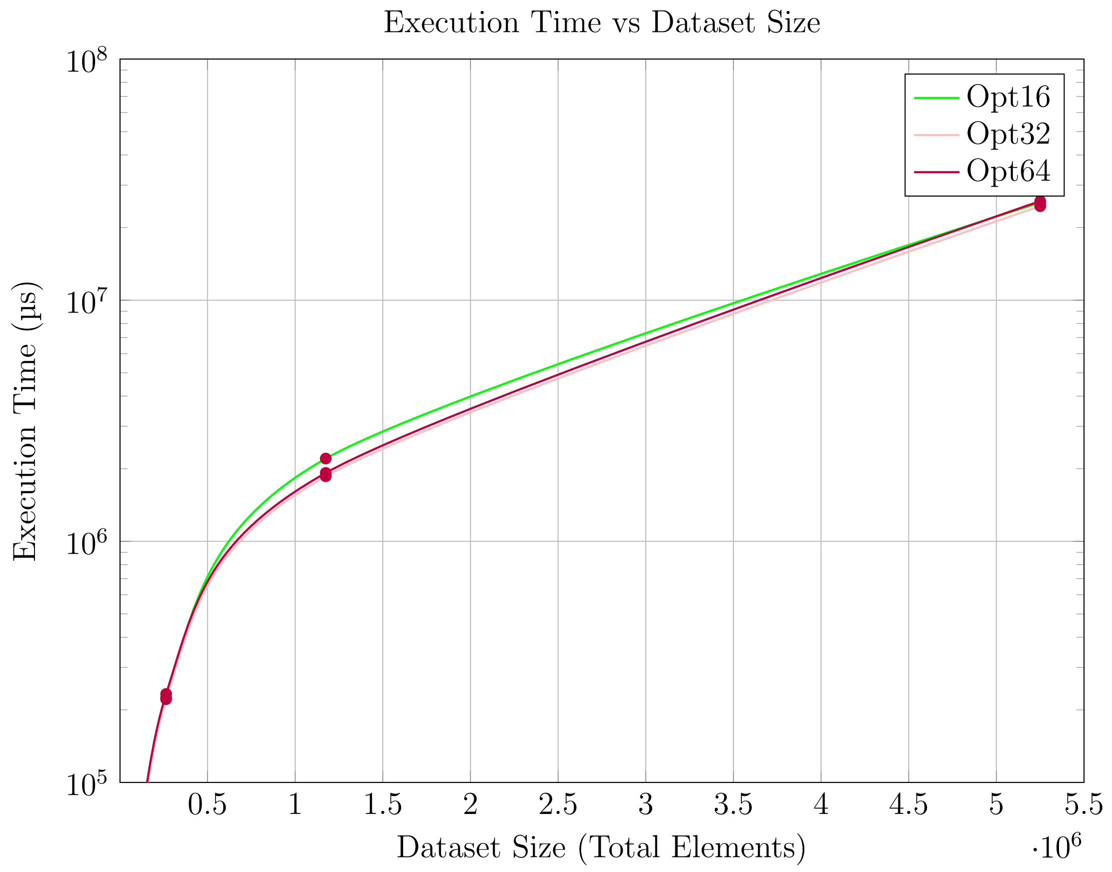
```
Setting up schedulers and affinity:
  * Setting the niceness level:
      -> trying niceness level = -20
    + Process has niceness level = -20
  * Setting up FIFO scheduling scheme and high priority ... Succeeded
  * Setting up scheduling affinity ... Succeeded

Running "scalar_opt" implementation:
  * Invoking the implementation 20 times .... Finished
  * Verifying results .... Success
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 734113245
      - Average = 26321780085
      - Number of active elements = 20
      - Number of masked-off = 4
    + Starting statistics run number #2:
      - Standard deviation = 574367356
      - Average = 25598762400
      - Number of active elements = 16
      - Number of masked-off = 1
    + Starting statistics run number #3:
      - Standard deviation = 286857123
      - Average = 25468953877
      - Number of active elements = 15
      - Number of masked-off = 1
    + Starting statistics run number #4:
      - Standard deviation = 94935667
      - Average = 25396312350
      - Number of active elements = 14
      - Number of masked-off = 0
  * Runtimes (MATCHING):  25396312350 ns
  * Dumping runtime informations:
    - Filename: scalar_opt_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished
```

Listing 8: Native Dataset Output

6.10 Impact of Block Size

The results indicate that the block size b has a significant impact on performance. For smaller datasets, the block size does not have a noticeable effect, as the entire dataset can fit into the CPU cache. However, for larger datasets, a block size of $b = 16$ consistently provides the best performance. This is because it strikes a balance between reducing cache misses and minimizing overhead from frequent block switching.

The following graph illustrates the execution time of the optimized implementation against different block sizes for the native dataset. The X-axis represents the block size, while the Y-axis shows the execution time in microseconds (µs). The graph includes both the actual data points and a smooth curve fitting the points for better visualization.



6.11 Speedup Analysis

The speedup achieved by the optimized implementation is most pronounced for larger datasets. For the native dataset, the optimized implementation achieves a speedup of approximately 4.5x compared to the naïve implementation. This is due to the significant reduction in cache misses and improved data reuse achieved by the blocking strategy.

6.12 Stationary Sub-Matrix Selection

The choice of the stationary sub-matrix also impacts performance. Using the output sub-matrix as stationary ensures that intermediate results are accumulated efficiently, reducing redundant memory accesses. This approach consistently outperformed alternatives, such as making one of the input sub-matrices stationary.

6.13 Takeaways

1. **Optimal Block Size**: A block size of $b = 16$ provides the best performance for most datasets, balancing cache efficiency and computational overhead.
2. **Speedup Trends**: The speedup achieved by the optimized implementation increases with dataset size, highlighting the inefficiencies of the naïve implementation for larger datasets.
3. **Stationary Sub-Matrix**: Using the output sub-matrix as stationary is critical for maximizing performance, as it minimizes redundant memory accesses and improves data reuse.
4. **Cache Efficiency**: The blocking strategy significantly reduces cache misses, leading to improved performance for larger datasets.
5. **Memory Access Patterns**: The optimized implementation's memory access patterns are more predictable, leading to better cache utilization and reduced latency.

7 Conclusion

In conclusion, the implementation of matrix-matrix multiplication (mmult) has been successfully optimized using a blocking strategy. The results demonstrate a significant performance improvement over the naïve implementation, particularly for larger datasets. The choice of block size and the sub-matrix selection play a crucial role in achieving optimal performance. The evaluation results indicate that the optimized implementation can achieve speedups of up to 4.5x compared to the naïve version, making it a valuable optimization. The insights gained from this project will inform future work on optimizing other benchmarks and improving overall performance in scientific computing applications.

8 References

1. GNU Make Manual: `https://www.gnu.org/software/make/manual/make.html`
2. IEEE POSIX Standard: `https://ieeexplore.ieee.org/servlet/opac?punumber=6880749`
3. YABMS Original Repository: `https://github.com/hawajkm/YABMS`
4. YABMS Forked Repository: `https://github.com/mohammed0x00/YABMS`
5. Benchmark Suites: Developing and Implementing a Simple Benchmark Suite
6. Optimizing Benchmark Performance: Leveraging Architectural Details and Insights To Improve Benchmark Performance
7. Previous Report: `https://github.com/mohammed0x00/YABMS/blob/main/docs/Report.pdf`