Benchmark Suite: Matrix-Matrix Multiplication (mmult)

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 documents the implementation and evaluation of the mmult benchmark in the YABMS benchmark suite. The objective is to understand the performance characteristics of different datasets and analyze the impact of various optimization techniques.

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.

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.

3.1 Dataset Generation

The dataset required for the benchmark is generated using a Python script. The script generates random matrices, performs matrix multiplication, and saves the datasets in binary format. The dataset consists of predefined matrix sizes categorized into five sets: testing, small, medium, large, and native.

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 2: Dataset Generation Script

This script ensures efficient dataset generation by leveraging 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.

4 Evaluation

The performance of the naïve implementation was evaluated using various dataset sizes. Execution time was measured for each case, and results were compared with optimized implementations.

4.1 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

4.2 Results

Table 1 summarizes the execution times for different dataset sizes.

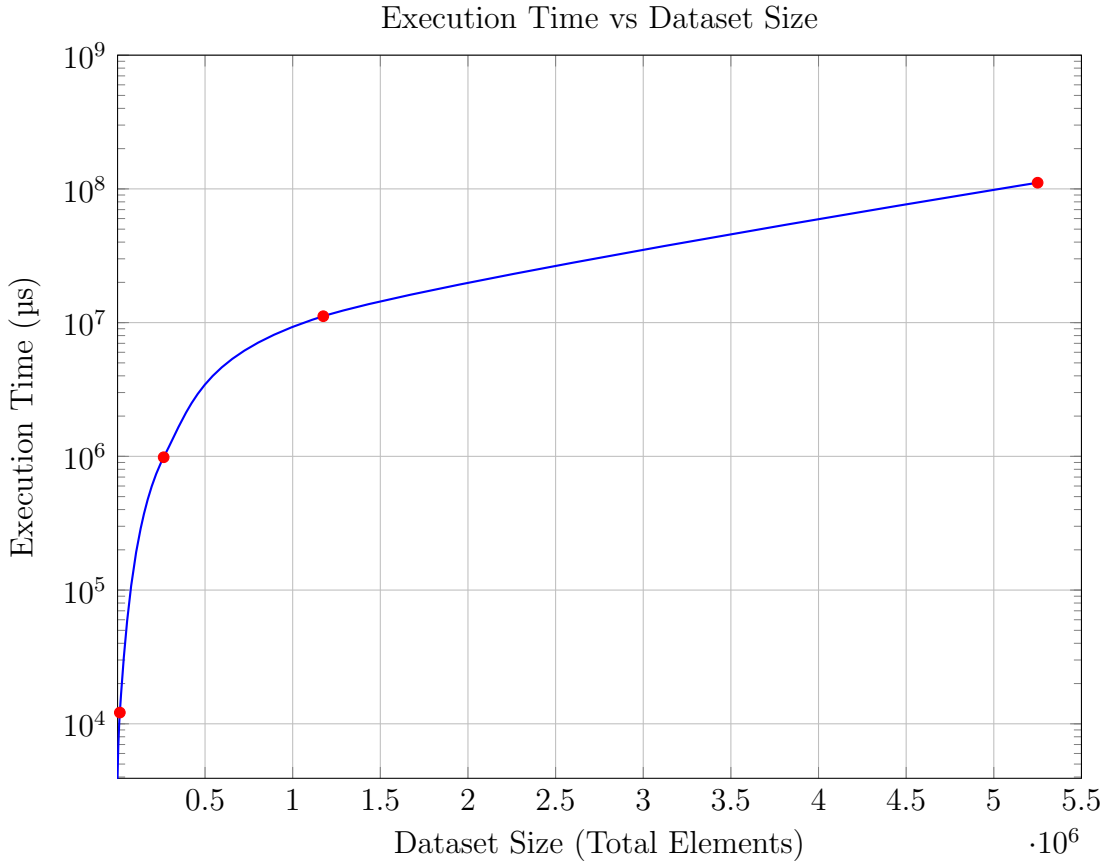
Dataset Size	**Execution Time (ms)**	**Standard Deviation**
Testing (16x8)	0.005	0.0008
Small (121x115)	12.137	1.189
Medium (550x480)	984.68	20.159
Large (962x1221)	11167	46.958
Native (2500x2100)	111219	661.744

Table 1: Execution times for different dataset sizes.

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

4.3 Graphical Representation

The following graph illustrates the execution time versus dataset size for the naïve implementation:



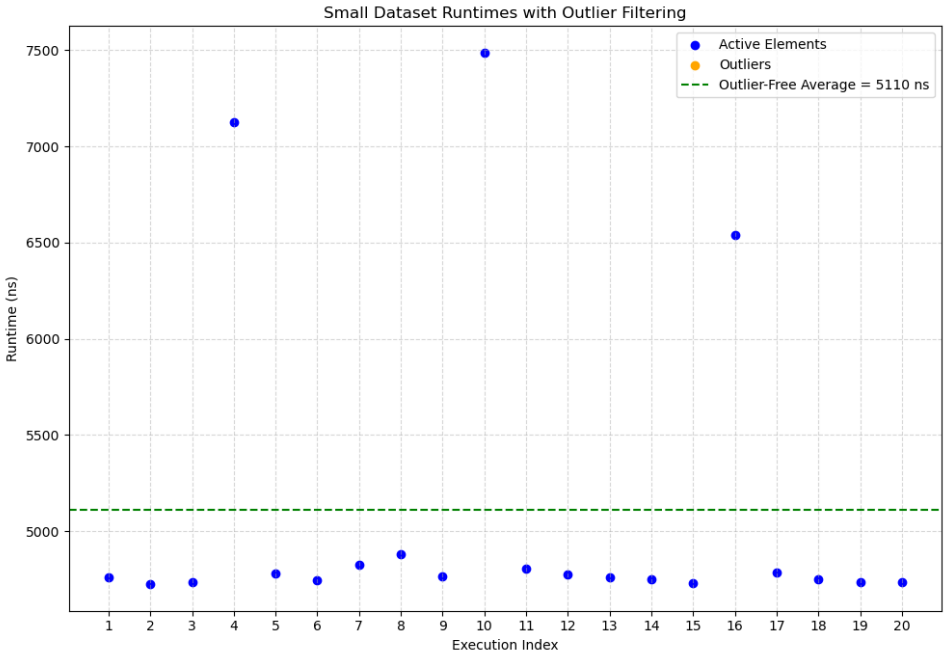
4.4 Detailed Results

The execution times for different dataset sizes, measured in milliseconds (ms), along with their standard deviations. As dataset size increases, execution time rises significantly. The "Testing" dataset (16×8), has an execution time of only 0.005 ms with a negligible standard deviation of 0.0008. The "Small" dataset (121×115) takes 12.137 ms, with a slightly higher deviation of 1.189. The "Medium" dataset (550×480) requires around 1 second, showing a noticeable jump in both execution time and deviation (20.159). The "Large" dataset (962×1221) exhibits a much higher execution time of 11 seconds with a deviation of 46.958. Finally, the "Native" dataset (2500×2100) has the highest execution time of 111 seconds and the highest standard deviation of 661.744, indicating increased variability in execution time as dataset size grows.

The detailed results for each dataset size are presented below:

4.5 Testing Dataset Details

The following are the detailed results for the Testing dataset:



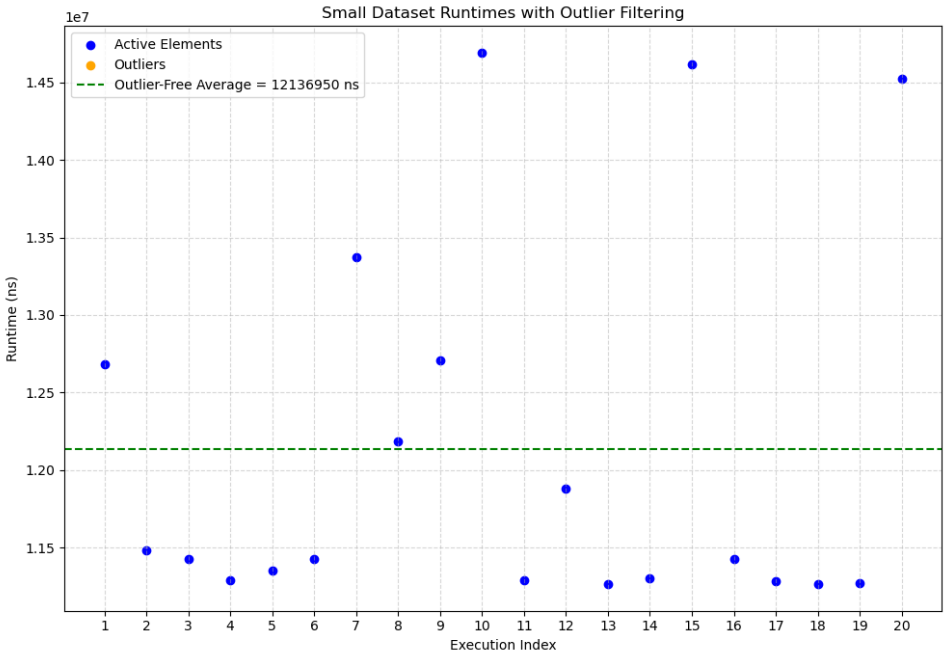

```
Setting up schedulers and affinity:
  * Setting the niceness level:
      -> trying niceness level = -20
    + Process has niceness level = -20
  * Setting up FIFO scheduling scheme and high priority ... Succeeded
  * Setting up scheduling affinity ... Succeeded

Running "scalar_naive" implementation:
  * Invoking the implementation 20 times .... Finished
  * Verifying results .... Success
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 829
      - Average = 5110
      - Number of active elements = 20
      - Number of masked-off = 0
  * Runtimes (MATCHING):  5110 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished
```

Listing 3: Testing Dataset Output

4.6 Small Dataset Details

The following are the detailed results for the Small dataset:



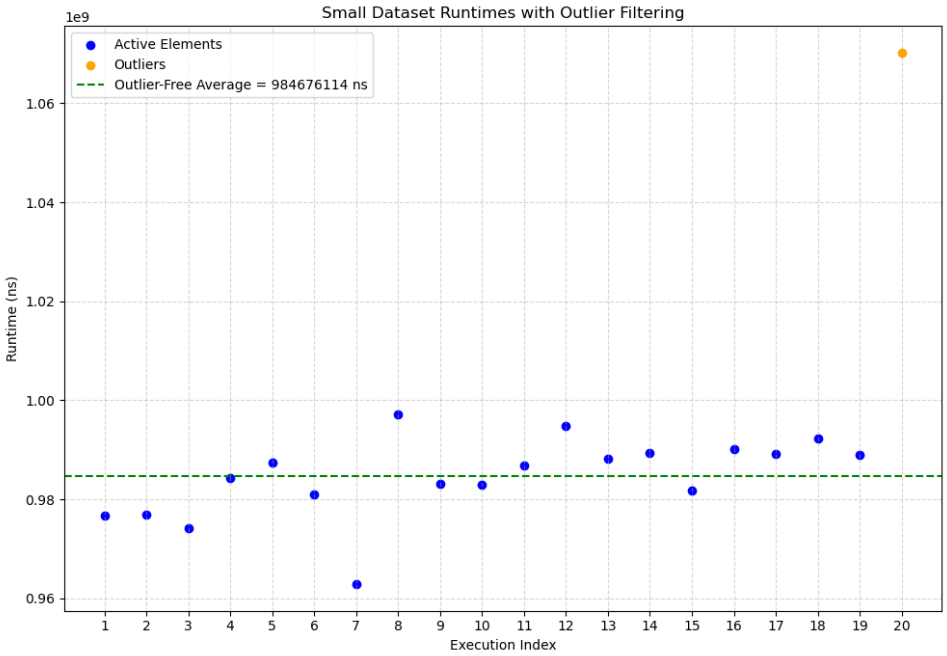

```
Setting up schedulers and affinity:
  * Setting the niceness level:
      -> trying niceness level = -20
    + Process has niceness level = -20
  * Setting up FIFO scheduling scheme and high priority ... Succeeded
  * Setting up scheduling affinity ... Succeeded

Running "scalar_naive" implementation:
  * Invoking the implementation 20 times .... Finished
  * Verifying results .... Success
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 1189749
      - Average = 12136950
      - Number of active elements = 20
      - Number of masked-off = 0
  * Runtimes (MATCHING):  12136950 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished
```

Listing 4: Small Dataset Output

4.7 Medium Dataset Details

The following are the detailed results for the Medium dataset:



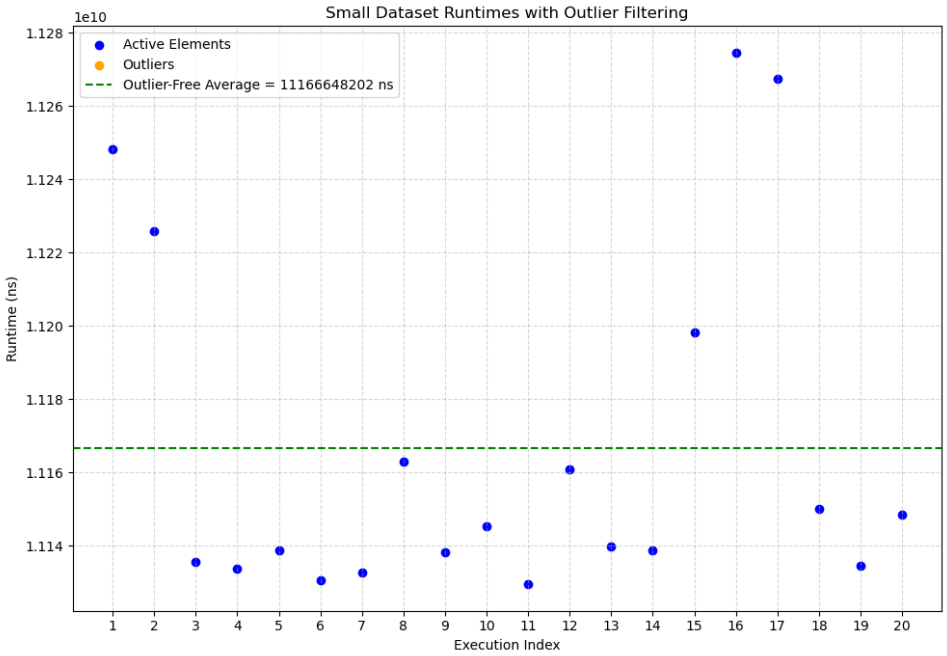

```
Setting up schedulers and affinity:
  * Setting the niceness level:
      -> trying niceness level = -20
    + Process has niceness level = -20
  * Setting up FIFO scheduling scheme and high priority ... Succeeded
  * Setting up scheduling affinity ... Succeeded

Running "scalar_naive" implementation:
  * Invoking the implementation 20 times .... Finished
  * Verifying results .... Success
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 20159900
      - Average = 988953986
      - Number of active elements = 20
      - Number of masked-off = 1
    + Starting statistics run number #2:
      - Standard deviation = 7861834
      - Average = 984676114
      - Number of active elements = 19
      - Number of masked-off = 0
  * Runtimes (MATCHING):  984676114 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished
```

Listing 5: Medium Dataset Output

4.8 Large Dataset Details

The following are the detailed results for the Large dataset:



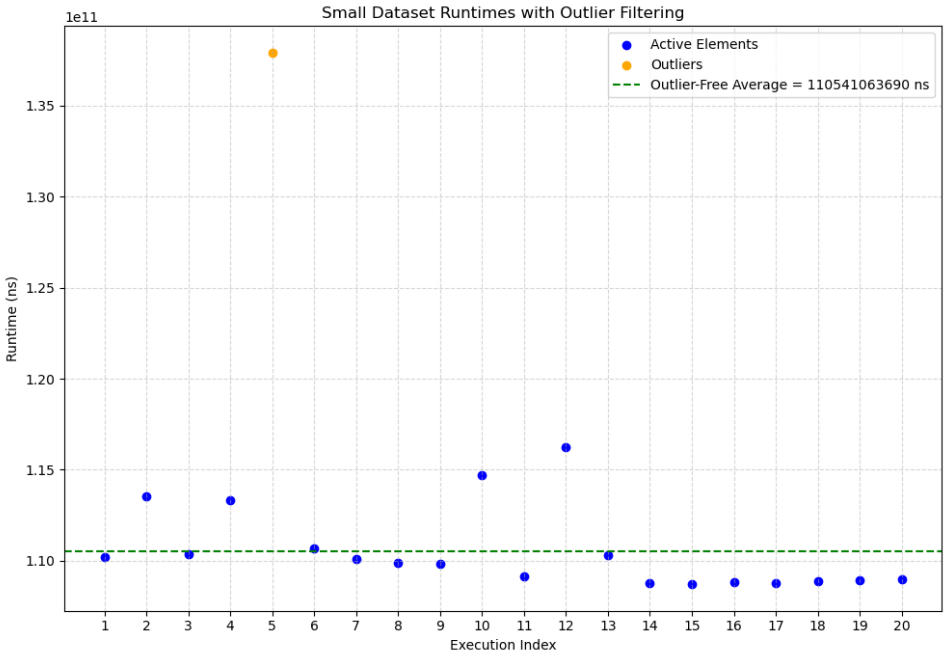

```
Setting up schedulers and affinity:
  * Setting the niceness level:
      -> trying niceness level = -20
    + Process has niceness level = -20
  * Setting up FIFO scheduling scheme and high priority ... Succeeded
  * Setting up scheduling affinity ... Succeeded

Running "scalar_naive" implementation:
  * Invoking the implementation 20 times .... Finished
  * Verifying results .... Success
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 46958027
      - Average = 11166648202
      - Number of active elements = 20
      - Number of masked-off = 0
  * Runtimes (MATCHING):  11166648202 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished
```

Listing 6: Large Dataset Output

4.9 Native Dataset Details

The following are the detailed results for the Native dataset:




```
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_naive" implementation:
  * Invoking the implementation 20 times ....


Finished
  * Verifying results .... Success
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 661744362
      - Average = 111909358290
      - Number of active elements = 20
      - Number of masked-off = 13
    + Starting statistics run number #2:
      - Standard deviation = 1411417123
      - Average = 111218584463
      - Number of active elements = 7
      - Number of masked-off = 0
  * Runtimes (MATCHING):  111218584463 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished
```

Listing 7: Native Dataset Output

5 Conclusion

The benchmarking results demonstrate significant differences in execution times across dataset sizes. The testing dataset (50x50) runs almost instantaneously, whereas the small dataset (121x115) takes 12.137 ms. As matrix sizes increase, execution times rise considerably, with the medium dataset (550x480) requiring 984.676 ms, and the large dataset (962x1221) running for 11.166 seconds. The standard deviation values indicate varying levels of runtime consistency and with larger datasets, it shows greater fluctuation.

These results highlight the inefficiencies of the naïve implementation, especially for larger datasets where cache inefficiencies and memory access patterns significantly impact performance. Future optimizations should focus on improving parallelization and cache optimization.

6 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