Benchmark Suite: Matrix-Matrix Multiplication (mmult)

Mohammed Mansour

March 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 **GitHub Runners** with the following setup:

- Runner Type: Ubuntu-latest GitHub Hosted Runner
- Operating System: Linux
- Processor (CPU): 4 cores
- Memory (RAM): 16 GB
- Storage (SSD): 14 GB
- Architecture: x64
- Workflow Label: ubuntu-latest, ubuntu-24.04
- Compiler: GCC 13.3.0 with optimization flags -O3

4.2 Results

Table 1 summarizes the execution times for different dataset sizes.

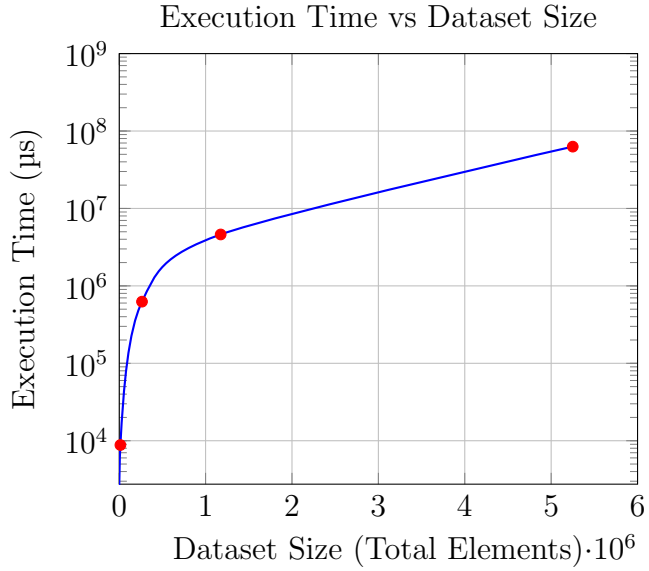
Dataset Size	Execution Time (ms)	Standard Deviation
Testing (16x8)	0.002	0.0002
Small (121x115)	8.836	1.05
Medium (550x480)	625.483	0.496743
Large (962x1221)	4612	4.87
Native (2500x2100)	62837.161	62837

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/actions`.

4.3 Graphical Representation

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



4.4 Testing Dataset Details

The following are the detailed results for the testing dataset:

```
Running "scalar_naive" implementation:
* Invoking the implementation 100 times .... Finished
* Verifying results .... Success
* Running statistics:
    + Starting statistics run number #1:
    - Standard deviation = 161
    - Average = 1977
    - Number of active elements = 100
    - Number of masked-off = 2
    + Starting statistics run number #2:
    - Standard deviation = 68
    - Average = 1956
    - Number of active elements = 98
    - Number of masked-off = 3
    + Starting statistics run number #3:
    - Standard deviation = 8
    - Average = 1944
    - Number of active elements = 95
    - Number of masked-off = 2
    + Starting statistics run number #4:
    - Standard deviation = 4
    - Average = 1943
    - Number of active elements = 93
    - Number of masked-off = 0
* Runtimes (MATCHING):  1943 ns
```

Listing 3: Testing Dataset Execution Details

4.5 Small Dataset Details

The following are the detailed results for the small dataset:

```
Running "scalar_naive" implementation:
* Invoking the implementation 100 times .... Finished
* Verifying results .... Success
* Running statistics:
    + Starting statistics run number #1:
    - Standard deviation = 1053204
    - Average = 8803383
    - Number of active elements = 100
    - Number of masked-off = 0
* Runtimes (MATCHING):  8803383 ns
```

Listing 4: Small Dataset Execution Details

4.6 Medium Dataset Details

The following are the detailed results for the medium dataset:

```
Running "scalar_naive" implementation:
* Invoking the implementation 100 times .... Finished
* Verifying results .... Success
* Running statistics:
    + Starting statistics run number #1:
    - Standard deviation = 496743
    - Average = 625518214
    - Number of active elements = 100
    - Number of masked-off = 2
    + Starting statistics run number #2:
    - Standard deviation = 232254
    - Average = 625516811
    - Number of active elements = 98
    - Number of masked-off = 2
    + Starting statistics run number #3:
    - Standard deviation = 177534
    - Average = 625494911
    - Number of active elements = 96
    - Number of masked-off = 1
    + Starting statistics run number #4:
    - Standard deviation = 169556
    - Average = 625489228
    - Number of active elements = 95
    - Number of masked-off = 1
    + Starting statistics run number #5:
    - Standard deviation = 161620
    - Average = 625483670
    - Number of active elements = 94
    - Number of masked-off = 0
* Runtimes (MATCHING):  625483670 ns
```

Listing 5: Medium Dataset Execution Details

4.7 Large Dataset Details

The following are the detailed results for the large dataset:

```
Running "scalar_naive" implementation:
* Invoking the implementation 100 times .... Finished
* Verifying results .... Success
* Running statistics:
    + Starting statistics run number #1:
    - Standard deviation = 4866044
    - Average = 4613444543
    - Number of active elements = 100
    - Number of masked-off = 2
    + Starting statistics run number #2:
    - Standard deviation = 2936481
    - Average = 4612904698
    - Number of active elements = 98
    - Number of masked-off = 1
    + Starting statistics run number #3:
    - Standard deviation = 2763573
    - Average = 4612799988
    - Number of active elements = 97
    - Number of masked-off = 0
* Runtimes (MATCHING):  4612799988 ns
```

Listing 6: Large Dataset Execution Details

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 8.836 ms. As matrix sizes increase, execution times rise considerably, with the medium dataset (550x480) requiring 625.483 ms, and the large dataset (962x1221) running for 4.612 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`