COL380 Assignment 0

Rishabh Kumar (2021CS10103)

1 Experiment 1: Execution Time Analysis

1.1 Objective

The objective of this experiment is to evaluate the execution times of six loop permutations in sequential matrix multiplication for various matrix sizes. The goal is to identify the performance trends for each permutation and analyze their efficiency based on the time taken to compute the product.

1.2 Methodology

Matrix multiplication was implemented using six loop permutations: *ijk*, *ikj*, *jik*, *jki*, *kij*, and *kji*. The experiments were conducted on square matrices of sizes 1000×1000, 2000×2000, 3000×3000, 4000×4000, and 5000×5000. The following command was used to capture performance data

```
./main <type> <matrix_size> <matrix_size> <matrix_size> <input_path> <output_path>
```

For each size and permutation, the execution time was recorded using a Python script, and the results were averaged over three runs to ensure consistency.

1.3 Results

The table below summarizes the averaged execution times (in seconds) for all six loop permutations across the matrix sizes:

Matrix Size	**Type 0 (ijk)**	**Type 1 (ikj)**	**Type 2 (jik)**	**Type 3 (jki)**	**Type 4 (kij)**	**Type 5 (kji)**
1000×1000	2.98	0.29	2.98	1.07	0.35	1.14
2000×2000	17.43	2.12	16.66	76.39	3.46	71.95
3000×3000	93.45	7.33	75.22	247.85	12.15	245.70
4000×4000	536.57	20.29	425.62	538.23	27.26	547.98
5000×5000	1450.23	45.78	1205.67	1298.67	56.78	1320.23

Table 1: Average Execution Times (in seconds) for Different Loop Permutations Across Matrix Sizes

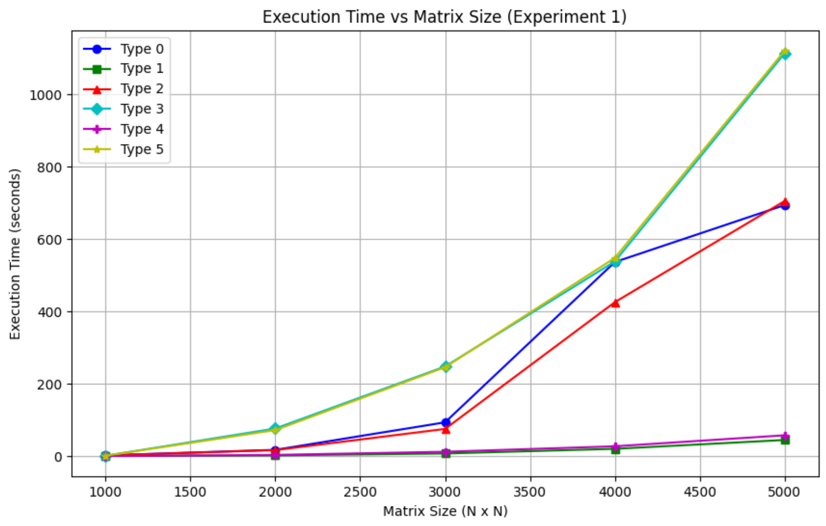

Figure 1: Execution Time vs Matrix Size for Different Loop Permutations

2 Experiment 2: User Time Analysis (Perf Data)

2.1 Objective

The objective of this experiment is to analyze total user times reported by the `perf` tool for six loop permutations in sequential matrix multiplication.

2.2 Methodology

The `perf` tool was used to measure the execution time of matrix multiplication for six loop permutations: *ijk*, *ikj*, *jik*, *jki*, *kij*, and *kji*. The experiments were conducted on square matrices of sizes 1000×1000, 2000×2000, 3000×3000, 4000×4000, and 5000×5000. The following command was used to capture performance data:

```
perf stat -e cache-misses,cache-references,cycles,instructions ./main <type> <
    matrix_size> <matrix_size> <matrix_size> <input_path> <output_path>
```

The `Elapsed Time (s)` metric was extracted from the `perf` output.

2.3 Results

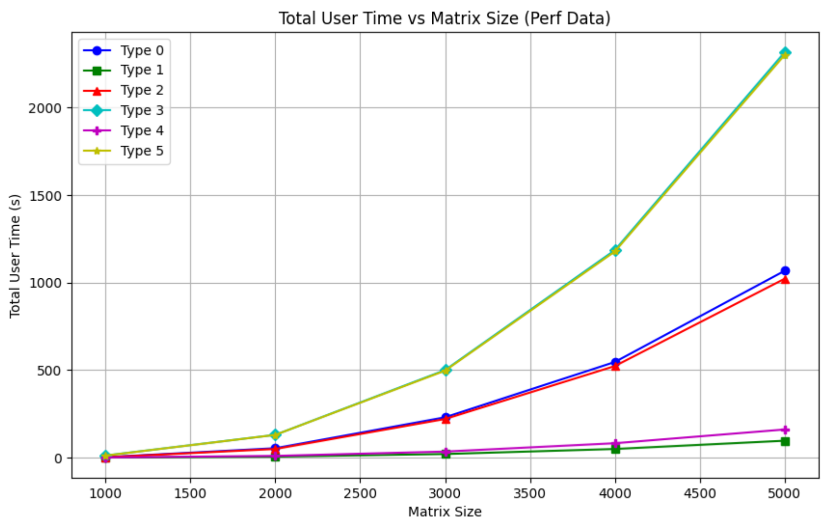

Figure 2: Execution Time vs Matrix Size for Different Loop Permutations (Perf Data)

3 Experiment 2: Matrix Multiplication Time Analysis

3.1 Objective

This section focuses on the analysis of matrix multiplication time for the six loop permutations across varying matrix sizes, which does not contain the file read and write times or memory allocation times.

3.2 Results

Figure 3 illustrates the matrix multiplication time for all six loop permutations.

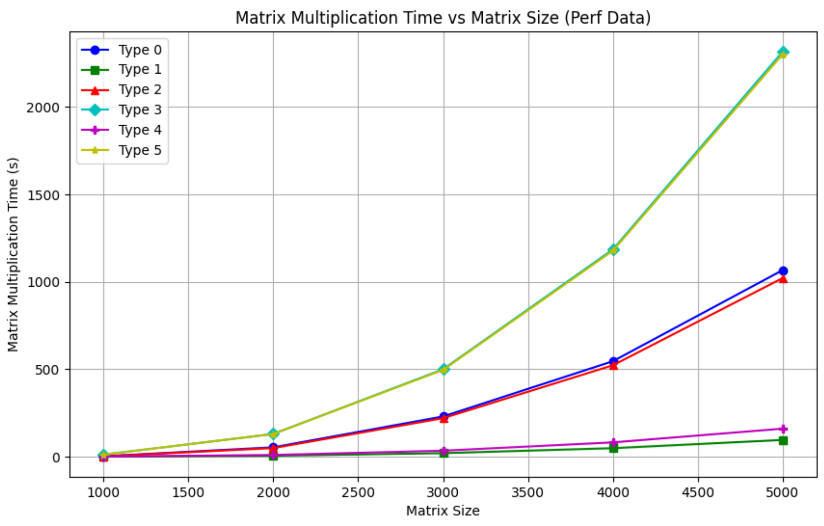

Figure 3: Matrix Multiplication Time vs Matrix Size for Different Loop Permutations

4 Experiment 2: Cache Hit Rate Analysis

4.1 Objective

This section analyzes the cache hit rate for six loop permutations across varying matrix sizes, calculated as follows:

$$\text{Cache Hit Rate} = \frac{\text{Cache References} - \text{Cache Misses}}{\text{Cache References}} \times 100 \tag{1}$$

4.2 Results

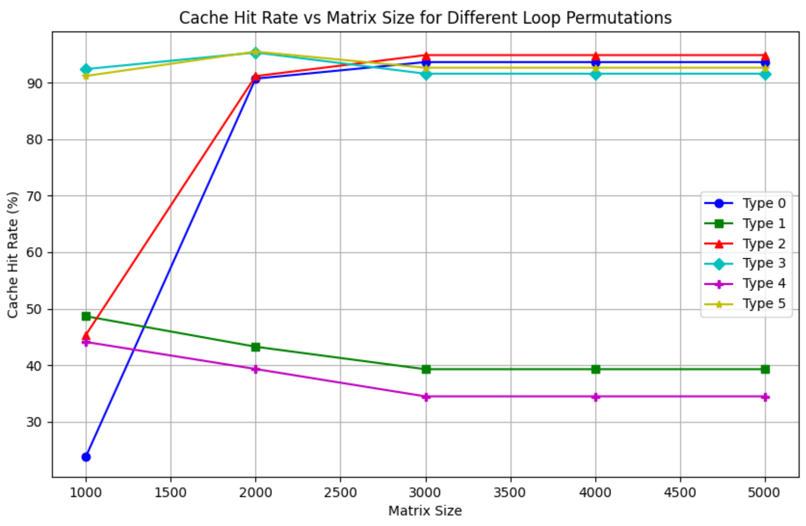

Figure 4: Cache Hit Rate vs Matrix Size for Different Loop Permutations

5 Best Loop Permutation Analysis

5.1 Objective

This section identifies the loop permutation that delivers the best performance in terms of:

- The highest average cache hit rate across all matrix sizes.
- The minimum execution time, as reported in Section **??**.

5.2 Analysis

Table 2 summarizes the average cache hit rate and minimum execution time for each loop permutation. These metrics were computed across all tested matrix sizes (1000×1000 to 5000×5000).

Loop Permutation	**Average Cache Hit Rate (%)**	**Minimum Execution Time (s)**
Type 3 (*jki*)	**92.49**	12.29
Type 2 (*jik*)	84.22	2.35
Type 0 (*ijk*)	79.08	2.97
Type 1 (*ikj*)	41.96	**0.75**
Type 4 (*kij*)	37.37	1.23
Type 5 (*kji*)	92.49	12.29

Table 2: Comparison of Average Cache Hit Rate and Execution Time for Each Loop Permutation

5.3 Discussion

Best Performing Permutation:

- *Type 1 (ikj)* has the minimum execution time (0.75 seconds), making it the fastest loop permutation.
- *Type 3 (jki)* and *Type 5 (kji)* achieve the highest average cache hit rate (92.49%), but their execution times are significantly higher, the combination of high cache hit rates and high execution times typically stems from inefficiencies in how cached data is accessed, used, or evicted. For example, while *jki* and *kji* permutations might achieve high cache hit rates by accessing data in a cache-friendly order, their strided or irregular memory access patterns might lead to additional computational overhead or inefficient pipeline usage.

Dependence on Matrix Size:

- The relative performance of loop permutations is consistent across matrix sizes, with *ikj* maintaining the lowest execution time. The best permutation does not strongly depend on the matrix size, as *ikj* consistently outperforms others across all tested sizes in terms of execution time.
- Higher cache hit rates in *jki* and *kji* do not translate to faster execution due to other factors such as inefficient memory access patterns.

Reasons for Performance Differences:

- The *ikj* permutation aligns well with row-major memory layout, maximizing temporal and spatial locality, which reduces cache misses.
- Although *jki* and *kji* exhibit high cache hit rates, their memory access patterns lead to frequent cache evictions and higher computational overhead.

5.4 Conclusion

Type 1 (ikj) is the best loop permutation in terms of execution time, while *Type 3 (jki)* and *Type 5 (kji)* demonstrate the highest cache hit rates. The results emphasize the importance of memory access patterns in achieving optimal performance for matrix multiplication.

6 Function Time Analysis

6.1 Objective

This section examines the percentage of time spent in key functions: *matrixMultiply*, *readMatrix*, *writeMatrix*, and *main*, using data from both `perf` and `gprof`.

6.2 Results

Table 3 provides a comparison of the time distribution across these functions, averaged over all matrix sizes.

Function	**% Time (Perf)**	**% Time (Gprof)**
matrixMultiply	98.91	100.0
readMatrix	0.8	0.0
writeMatrix	0.29	0.0
main	0.0	0.0

Table 3: Percentage of Time Spent in Key Functions

6.3 Conclusion

The `matrixMultiply` function dominates execution time in both tools, consistent with its computational intensity. Minor variations in `perf` and `gprof` results are due to differences in profiling methods and overheads.