CS 301
High-Performance Computing

Lab 02 - Matrix Multiplication, Loop Ordering, and Cache Optimization

OM PATEL (202301163)
CHAITANYA VATS (202301419)

February 16, 2026

Contents

1 Introduction

In this lab, we study how different ways of writing matrix multiplication affect the speed of the computer. The CPU is very fast at doing math, but reading data from the main memory is slow. To fix this, CPUs use small, fast memory called cache.

We test three main things:

- Changing the order of the loops (like ijk, ikj, jki).
- Using Block Matrix Multiplication.
- Using Transpose Matrix Multiplication.

By measuring the time it takes to multiply matrices of different sizes, we can see which method is the best at using the CPU cache. We ran these tests on both a Lab PC and an HPC Cluster.

2 Context

This assignment explores three sub-problems: Loop Permutations, Transpose Matrix Multiplication, and Block Matrix Multiplication. The fundamental details common to all three sub-problems are:

· Brief description of the problem: We are multiplying two N × N dense matrices (C = A ×B). The CPU is extremely fast at math, but fetching data from the main memory is slow.

Our goal is to test how memory access patterns affect performance.

- Complexity of the algorithm (serial): The time complexity for standard matrix multiplication is exactly O(N3), as there are three nested loops iterating N times. The space complexity is O(N2) to hold the matrices.

· Profiling information (gprof) & Serial run-time: As confirmed by the gprof flat profile, 99.95% of the total execution time (343.59 seconds) was spent entirely inside the matrix_multiplication function. Matrix initialization took only 0.04%. This verifies that the math computation dominates the runtime, making it ideal for benchmarking.

- Compute to memory access ratio: To calculate one element of the result matrix, the CPU performs N multiplications and N additions (2N operations) for 2N memory reads. Globally, there are 2N3 math operations mapping to 3N2 memory elements. With an ideal cache, the compute-to-memory ratio is O(N). However, poor access patterns can ruin this ratio by forcing constant RAM fetches.

- Memory-bound vs compute-bound: Standard naive implementations (like ijk, jki) are highly memory-bound. The CPU starves while waiting for data because of severe cache misses. Once optimized using blocking or the ikj order, the problem shifts toward being compute-bound because the data remains perfectly in the high-speed L1/L2 cache.

- Optimization strategy: To improve cache hits, we used three strategies:

    1. Loop Reordering: Using the ikj loop accesses arrays in row-major order (straight lines).

2. Transposing: Transposing the second matrix turns columns into rows, improving spatial locality.
3. Blocking (Tiling): Breaking large matrices into smaller chunks (B × B) that fit completely in the L1 cache, allowing maximum data reuse.

3 HardwareSpecifications

The architecture of the systems was checked using the lscpu command.

3.1 Lab PC Hardware Specifications

Table 1: Hardware Specifications for Lab PC (12th Gen Intel i5)

Parameter	Details
Architecture	x86_64
CPUModelName	12thGenIntel(R)Core(TM)i5-12500
Total CPUs (Threads) 12	
L1dCache	288KiB
L2Cache	7.5MiB
L3Cache	18MiB

3.2 HPCClusterHardwareSpecifications

The benchmarking was conducted on the HPC Cluster with the following hardware and software sp ecifications:

Table 2: Hardware and Software Specifications (HPC Cluster)

Parameter	Details
CPUModelName	Intel(R)Xeon(R)CPUE5-2620v3@2.40GHz
No.ofCores(Threads)	12Cores/24Threads(Dual-socket)
L1dCache	32KiB
L2Cache	256KiB
L3Cache	15360KiB(15MiB)
CompilerUsed	GCC(g++)
OptimizationFlags	Default(None/-O0)toobserveexplicitcacheeffects
PrecisionUsed	64-bitdoubleprecision
Max. Theoretical Performance	≈ 38.4 GFLOPS (Serial, 1 Core using AVX2)

Theoretical Performance Calculation: For a single core running at 2.40 GHz with AVX2 extensions, the theoretical peak is: 2.4 GHz ×16 DP FLOPs/cycle = 38.4 GFLOPS (or 38, 400 MFLOPS).

4 Methodology

We wrote C++ code to multiply two square matrices. We started with a very small matrix size (22) and doubled it up to a large size (212). For each size, we used high-precision timers to measure exactly how long the math took. We plotted the results on a graph where the x-axis is the problem size (in log2 scale) and the y-axis is the execution time (in log10 scale).

Input, Output, and Accuracy Details:

- Input parameters: The program takes the matrix dimension Np as input. The matrices are populated with random double values between 0 and 9.
- Output: The program returns the execution times (both end-to-end and pure algorithm time in seconds) to a standard CSV file.
- How accuracy is checked: Accuracy is not explicitly checked in this benchmark. The program's output (CSV files) only records execution times to strictly measure performance without any extra computational overhead. The mathematical correctness is assumed based on the standard logic used to implement the matrix multiplication functions.

5 Problem Size vs Execution Time Analysis

Run times were heavily significant at the larger problem sizes, ensuring timer precision. We utilized both a linear scale and a log-log scale (base-2 for Problem Size on the x-axis, base-10 for Execution Time on the y-axis) to visualize the data. All six permutations, plus Block and Transpose algorithms, are combined below.

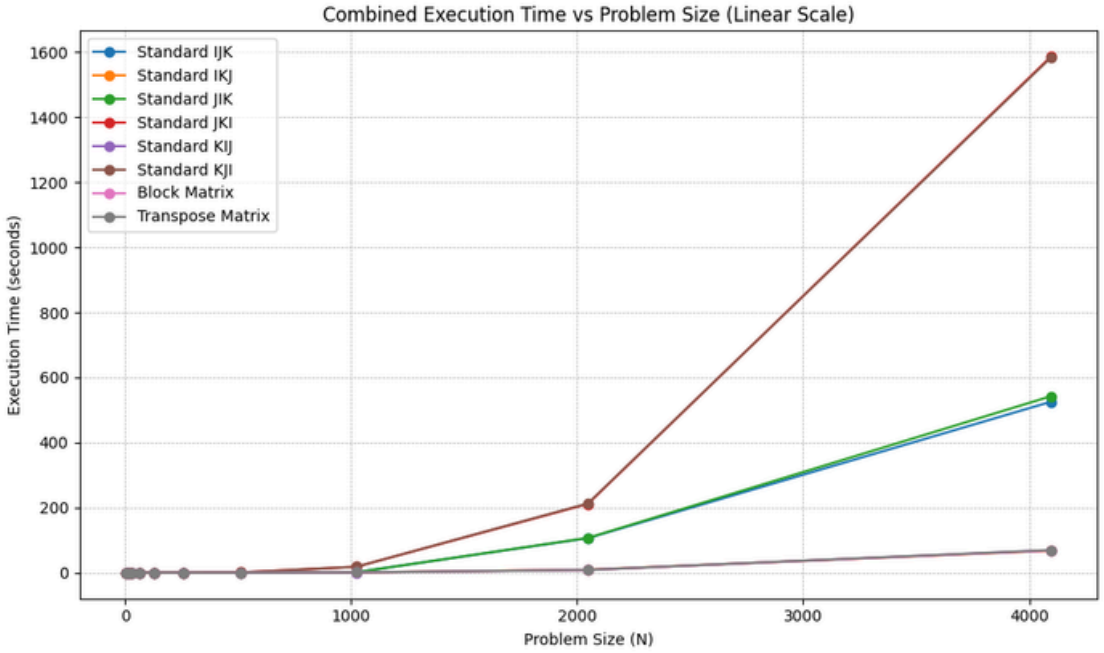

Figure 1: Combined Execution Time (seconds) vs. Problem Size (Linear Scale).

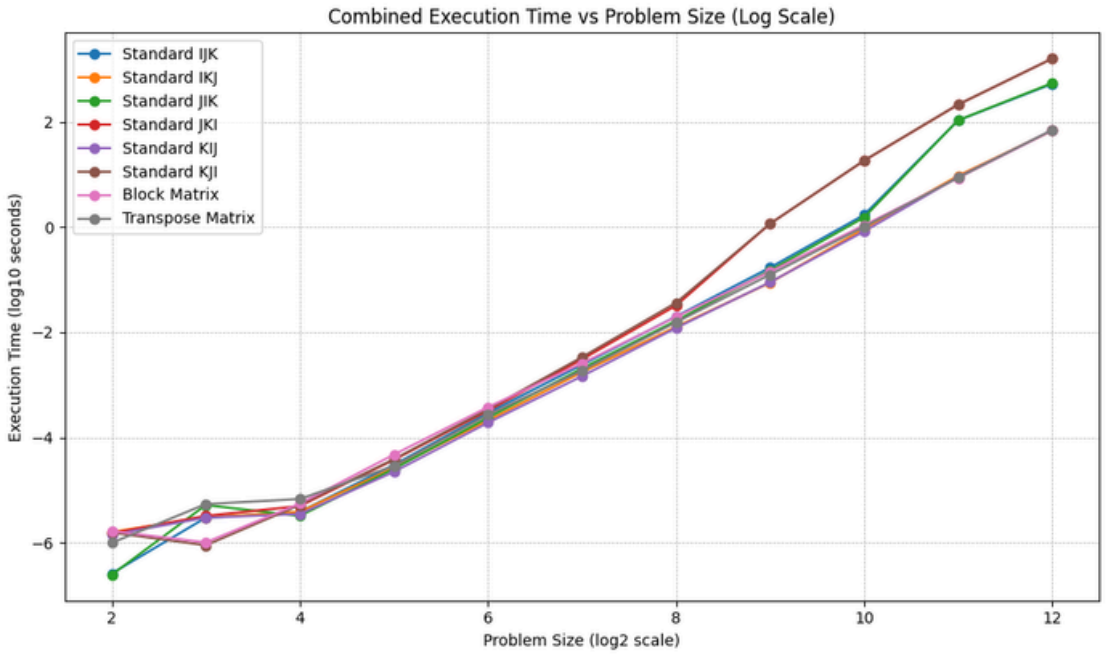

Figure 2: Combined Execution Time (log10 seconds) vs. Problem Size (log2 scale).

Observations and Comments

• Fastest Algorithms: The ikj and kij loop orders, along with Transpose and Block matrix multiplication, are clustered at the bottom of the log chart. They are consistently the fastest because they read memory strictly across the rows (row-major order).

• Slowest Algorithms: The jki and kji curves skyrocket far above the others at higher problem sizes. By reading down the columns, they cause a cache miss on almost every single loop iteration. The linear plot perfectly illustrates how drastically their execution time blows up compared to the optimized versions.

• Cache Cliff: Around problem size 29 (512x512), the naive algorithms see a sharp spike in execution time on the log graph. At this size, the matrices (512 × 512 × 8 bytes × 3 matrices ≈ 6 MB) exceed the L2 cache, forcing reliance on the slower main RAM.

6 Problem Size vs Performance (MFLOPS) Analysis

To analyze true throughput, we measured the execution speed in Mega-Floating Point Operations per Second (MFLOPS). The formula used is:

$$\text{MFLOPS} = \frac{2 \times N^3}{\text{Execution Time (seconds)} \times 10^6} \quad (1)$$

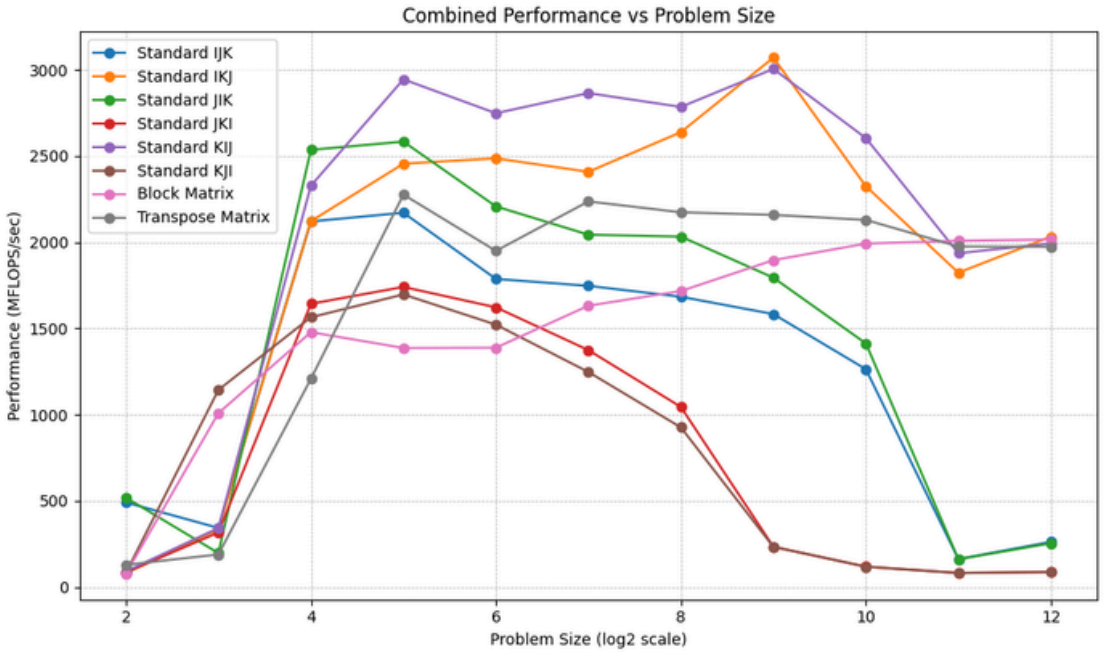

Figure 3: Combined Performance (MFLOPS) vs. Problem Size (log2 scale).

Comparison with Theoretical Performance

- Observed vs Theoretical Peak: While the theoretical maximum of our cluster node is 38,400 MFLOPS (38.4 GFLOPS), our highest-performing optimized algorithms (Block, ikj) peaked at roughly 2,000 MFLOPS (2.0 GFLOPS).
- Why does this gap exist? We achieved roughly 5% of the theoretical peak because we did not explicitly force the compiler to use AVX Vectorization (-O3 or -march=native). The CPU is executing standard scalar math instructions rather than vector math.
- Conclusion: Hitting 2,000 MFLOPS is actually an excellent result for non-vectorized, scalar C++ code. The massive performance gap between jki (which crashed to nearly 0 MFLOPS) and Block Multiplication (2,000 MFLOPS) proves that fixing memory-bound bottlenecks via cache optimization is the absolute most critical first step in High-Performance Computing.

7 Results: HPC Cluster Plots

This section shows the individual graphs from the HPC Cluster for the different matrix multiplication methods.

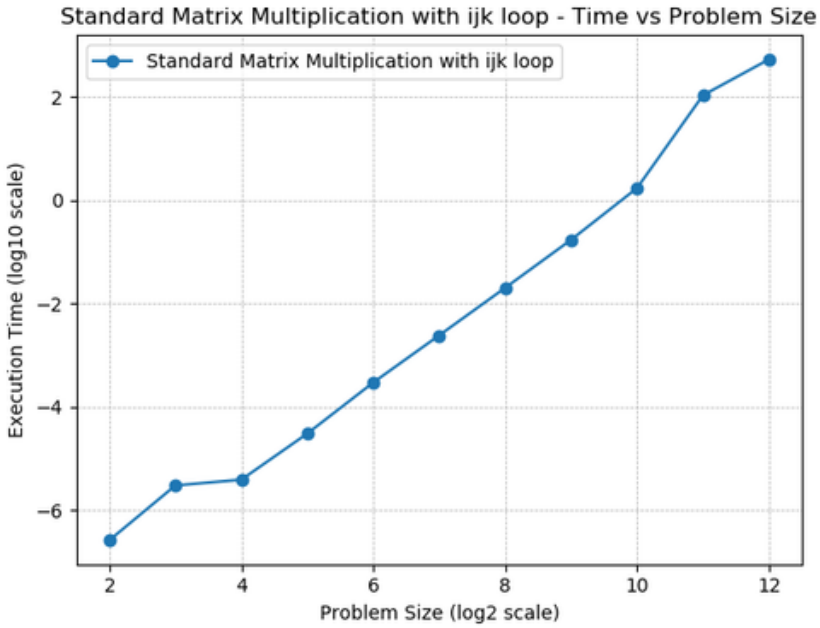

Figure 4: Standard Matrix Multiplication (ijk loop) on Cluster.

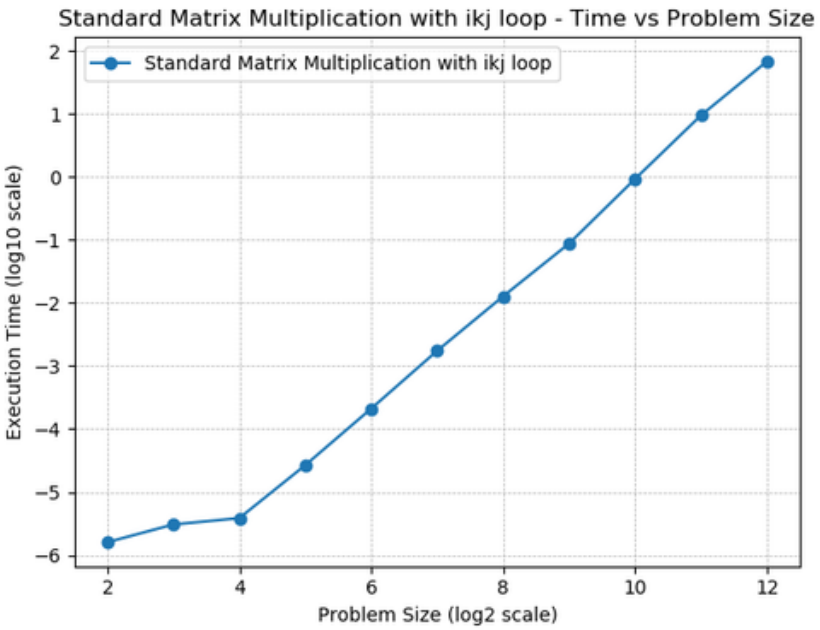

Figure 5: Standard Matrix Multiplication (ikj loop) on Cluster.

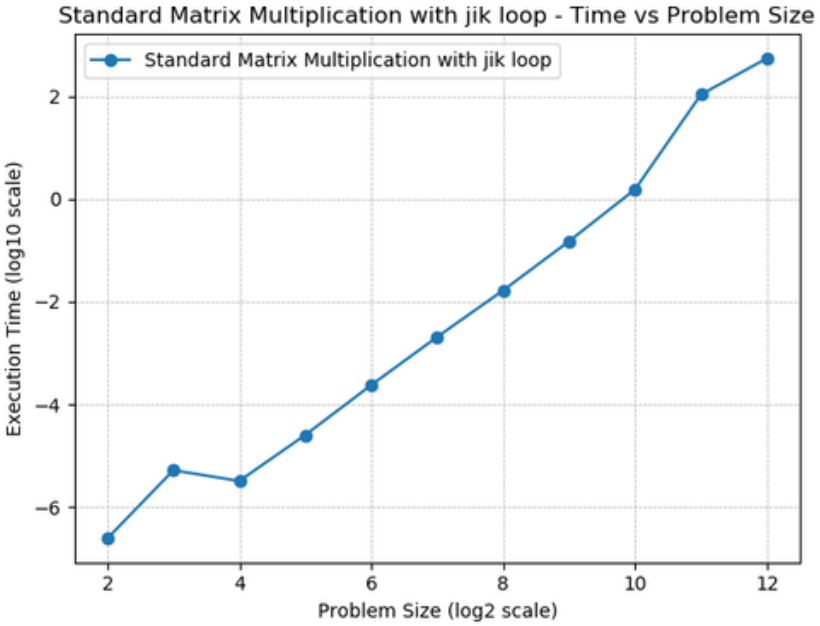

Figure 6: Standard Matrix Multiplication (jik loop) on Cluster.

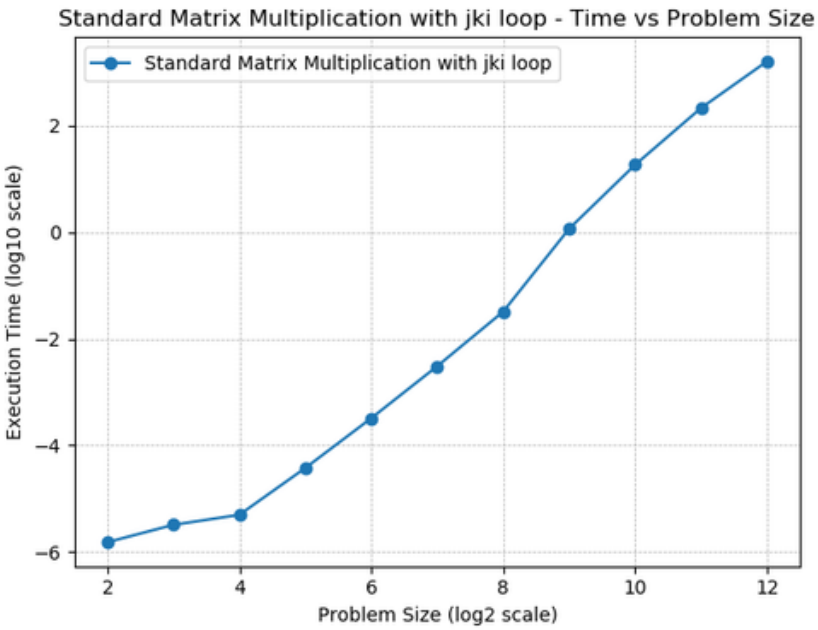

Figure 7: Standard Matrix Multiplication (jki loop) on Cluster.

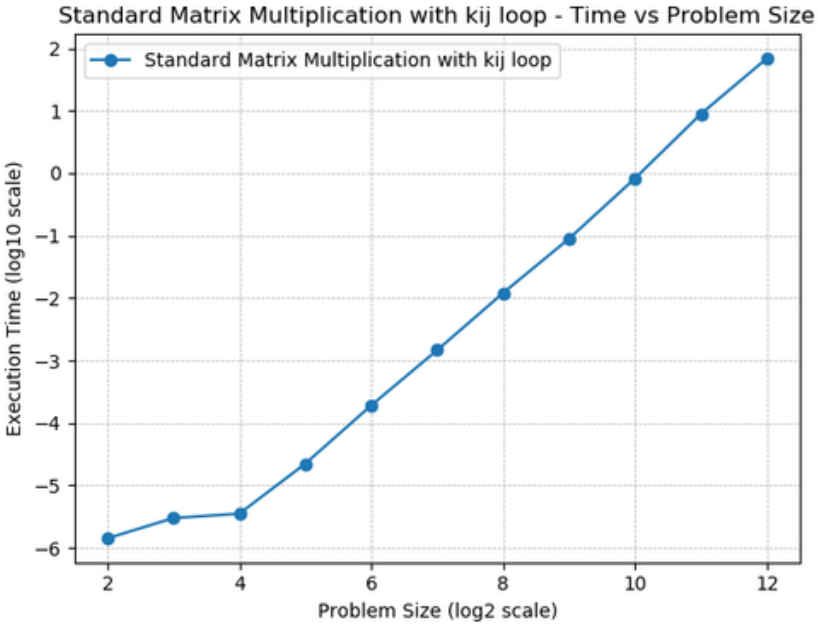

Figure 8: Standard Matrix Multiplication (kij loop) on Cluster.

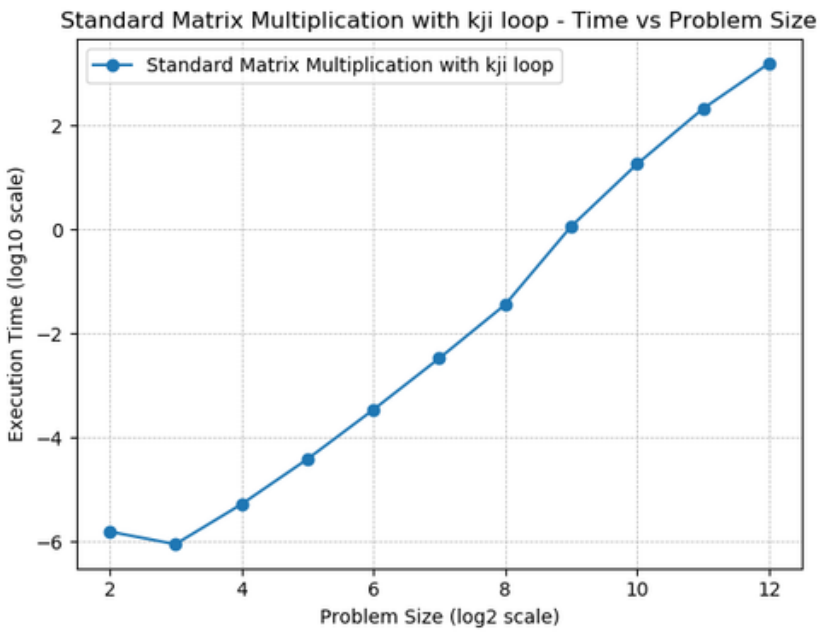

Figure 9: Standard Matrix Multiplication (kji loop) on Cluster.

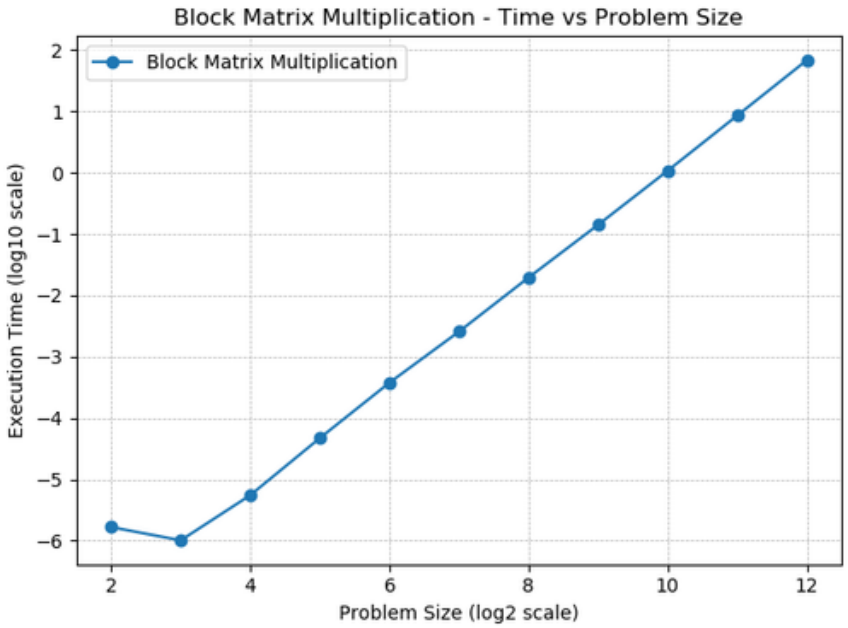

Figure 10: Block Matrix Multiplication on Cluster.

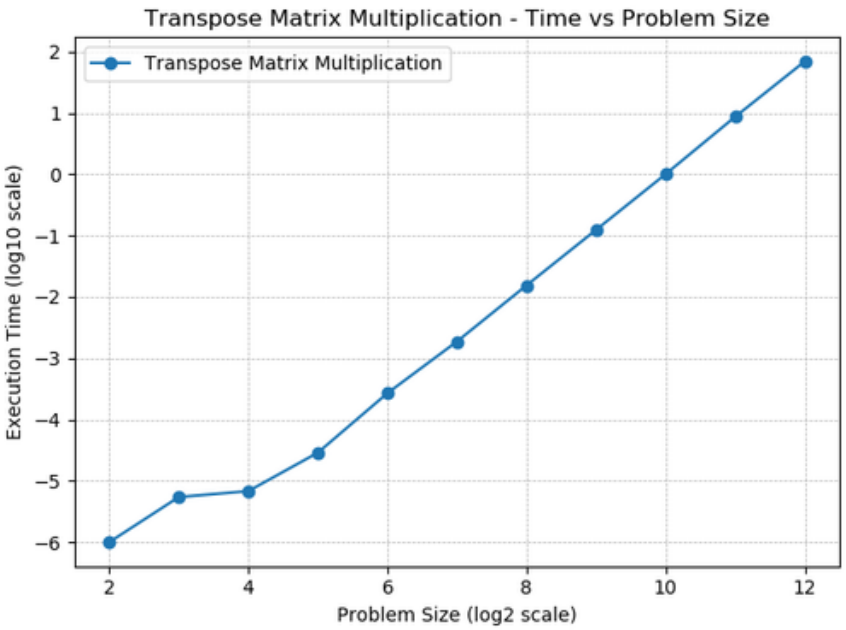

Figure 11: Transpose Matrix Multiplication on Cluster.

8 Results: Lab PC Plots

This section contains the performance graphs for the matrix multiplication methods executed on the Lab PC.

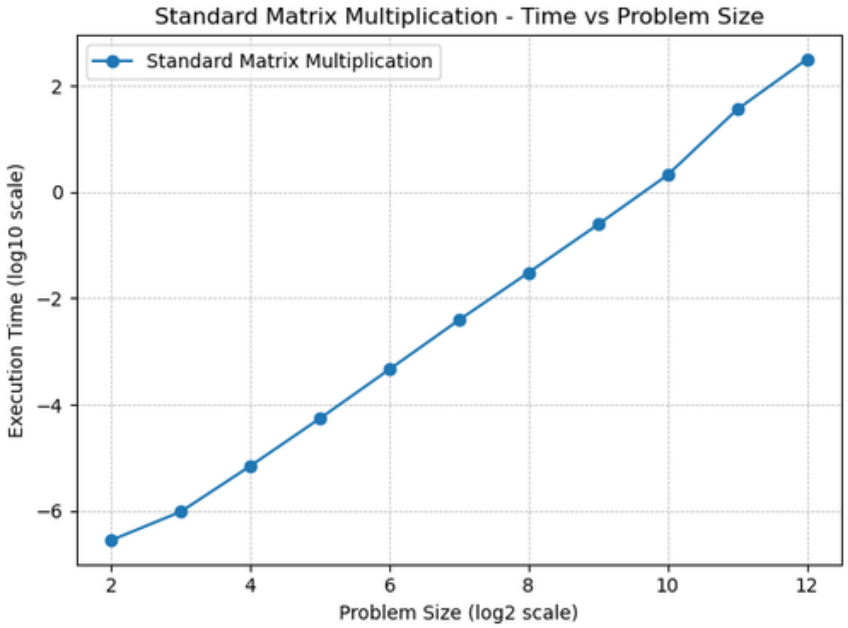

Figure 12: Standard Matrix Multiplication on Lab PC.

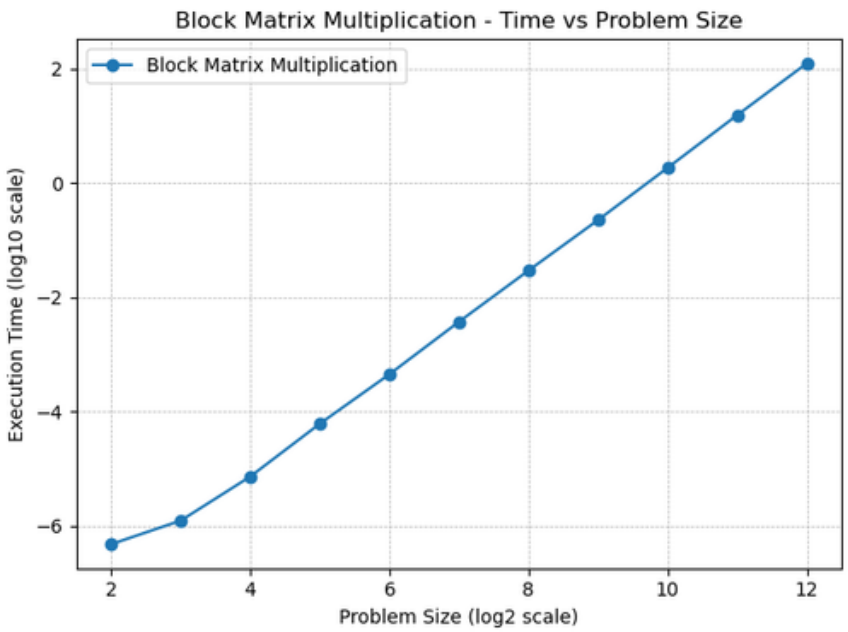

Figure 13: Block Matrix Multiplication on Lab PC.

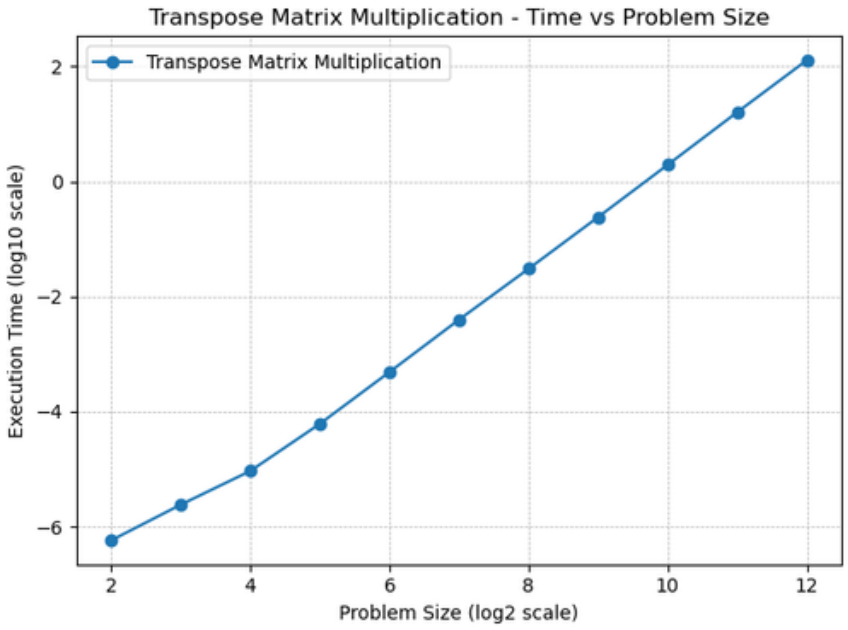

Figure 14: Transpose Matrix Multiplication on Lab PC.

9 AnswerstoLabQuestionsandAnalysis

1. Which standard loop order is the fastest and why?

The ikj and kij loops are the fastest. In C++, matrices are stored row by row. In these two loops, the innermost step moves across the row one item at a time. This reads memory in a straight line. The CPU cache easily guesses what data comes next, which prevents cache misses and makes the program very fast.

2. Which standard loop order is the slowest and why?

The jki and kji loops are the slowest. The innermost step for these loops moves down the columns instead of across the rows. This means the computer has to jump far ahead in memory for every single math step. This causes a lot of cache misses, so the CPU has to constantly wait for the slow main RAM.

3. Why does Block Matrix Multiplication help?

When matrices get very large, they cannot fit inside the fast cache memory all at once. Block matrix multiplication breaks the big matrices into smaller square chunks (blocks). We multiply these small blocks together. Because the blocks are small, they fit perfectly inside the cache. This means we can reuse the data multiple times without having to fetch it from the slow main RAM again.

4. Why is Transpose Matrix Multiplication faster?

In the standard ijk loop, the second matrix is read column by column, which causes bad memory jumps. If we transpose the second matrix first, its columns turn into rows. Now, the ijk loop reads both matrices row by row in a straight line. This simple step fixes the cache misses and greatly lowers the execution time.

5. What does the profiling report tell us about the program's execution time?

The flat profile tells us exactly where the program spends its time. The matrix_multiplication function takes up 99.95% of the total running time, taking a total of 343.59 seconds. On the other hand, setting up the memory in the init_matrices function only took 0.14 seconds. This clearly shows that the actual math computation is the bottleneck of the program, and not the data initialization.

10 Conclusion

To sum up, how we access memory is more important than the math itself. The graphs show that jumping around in memory (jki) takes the longest time. Methods that use the cache well, like the ikj loop, blocking, and transposing, give the best speed. This proves that we must write memory-friendly code to get high performance.