CS 301
High-Performance Computing

Lab 02 – Matrix Multiplication Performance

Diya Patel (202301216)
Yesha Joshi (202301462)

February 2026

Contents

1 Introduction **3**

2 Algorithmic Analysis **3**
2.1 Computational Complexity 3
2.2 Compute-to-Memory Access Ratio 4

3 Hardware Details **4**
3.1 Lab 207 Machine 4
3.2 HPC Cluster 4

4 Input Parameters and Accuracy **5**

5 Problem A – Conventional Matrix Multiplication **5**
5.1 Loop Permutations 5
5.2 Lab Machine 6
5.3 Cluster 7
5.4 Conclusion for Problem A 8

6 Problem B – Transpose Matrix Multiplication **9**
6.1 Lab Machine 9
6.2 Cluster 10

7 Problem C – Block Matrix Multiplication **11**
7.1 Lab Machine 11
7.2 Cluster 12

8 Comparison of All Methods **13**
8.1 Cluster Comparison 13

9 Conclusion **14**

1 Introduction

In this lab, we analyze the performance of different matrix multiplication algorithms. The objective is to measure runtime and computational performance as a function of problem size ranging from 2^2 to 2^{12}.

The following implementations were studied:

- Conventional matrix multiplication (6 loop permutations)
- Matrix multiplication using transpose
- Block matrix multiplication (divide and conquer)

All implementations were executed on:

- Lab 207 Machine
- HPC Cluster

Each experiment was repeated 5 times to ensure statistical accuracy.

2 Algorithmic Analysis

2.1 Computational Complexity

Matrix multiplication requires three nested loops.

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} B_{kj}$$

Total operations:

$$Multiplications = N^3$$

$$Additions = N^3$$

$$Total\ FLOPs = 2N^3$$

Therefore, time complexity:

$$O(N^3)$$

2.2 Compute-to-Memory Access Ratio

For each iteration:

- 2 floating point operations
- 2 memory reads
- 1 memory write

Total memory accesses:

$$3N^2$$

Arithmetic intensity:

$$AI = \frac{2N^3}{3N^2}$$

$$AI = \frac{2N}{3}$$

As N increases, arithmetic intensity increases, making computation more dominant.

3 Hardware Details

3.1 Lab 207 Machine

- CPU: Intel Core i5-12500
- Cores: 6 physical cores
- Max Frequency: 4.6 GHz
- Cache:
  - L1 Cache: 288 KB
  - L2 Cache: 7.5 MB
  - L3 Cache: 18 MB

3.2 HPC Cluster

- CPU: Intel Xeon E5-2620 v3
- Cores: 12 physical cores
- Frequency: 2.4 GHz
- L3 Cache: 15 MB

4 Input Parameters and Accuracy

- Matrix sizes: 2^2 to 2^{12}
- Data types used:
  - Integer (4 bytes)
  - Double precision (8 bytes)

Correctness was verified by comparing results with a reference implementation.

5 Problem A – Conventional Matrix Multiplication

5.1 Loop Permutations

Six possible loop permutations were implemented by interchanging the order of indices i, j, and k. Each permutation affects the memory access pattern and performance.

- **Loop Type 1: i – j – k**

  Outer loop over i, middle loop over j, and inner loop over k. Each element of matrix C is computed row-wise by accumulating products across index k.

- **Loop Type 2: i – k – j**

  Outer loop over i, middle loop over k, and inner loop over j. A row of matrix A is reused to update the entire row of matrix C.

- **Loop Type 3: j – i – k**

  Outer loop over j, middle loop over i, and inner loop over k. Each column of matrix C is computed by traversing rows of matrix A.

- **Loop Type 4: j – k – i**

  Outer loop over j, middle loop over k, and inner loop over i. A column of matrix C is updated using corresponding elements from matrices A and B.

- **Loop Type 5: k – i – j**

  Outer loop over k, middle loop over i, and inner loop over j. Elements of matrix A and rows of matrix B are reused to update rows of matrix C.

- **Loop Type 6: k – j – i**

  Outer loop over k, middle loop over j, and inner loop over i. Columns of matrix C are updated using elements from matrices A and B.

5.2 Lab Machine

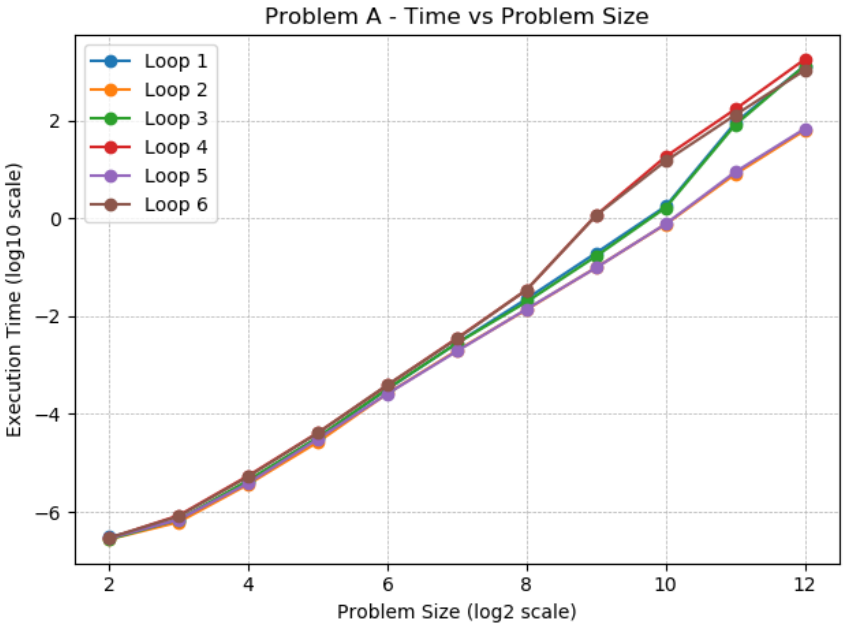

Figure 1: Lab Machine: Runtime vs Problem Size (Problem A)

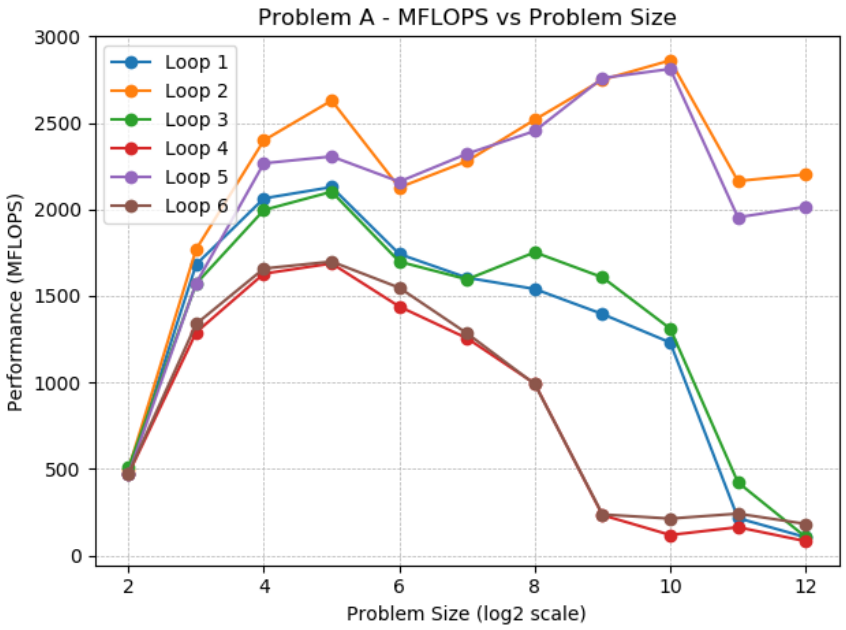

Figure 2: Lab Machine: MFLOPS vs Problem Size (Problem A)

5.3 Cluster

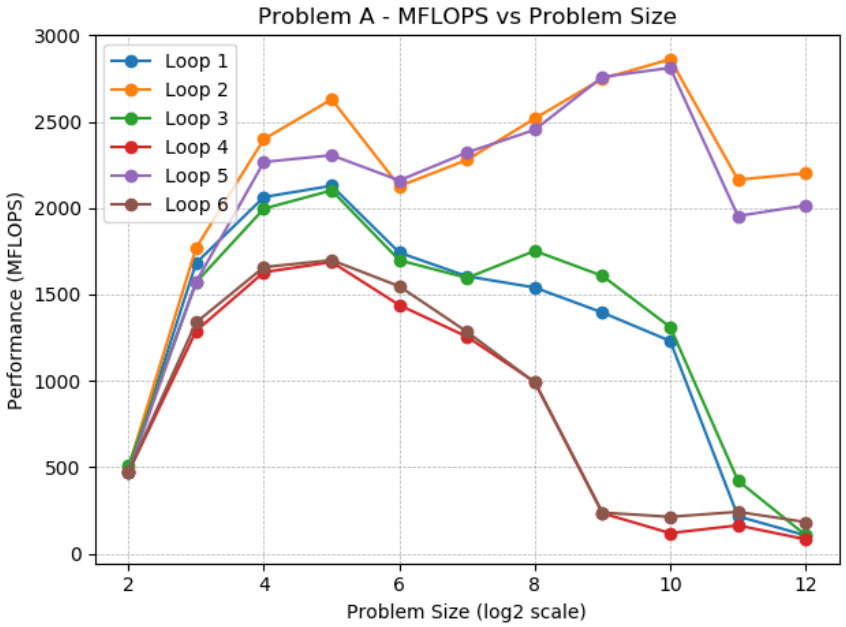

Figure 3: Cluster: Runtime vs Problem Size (Problem A)

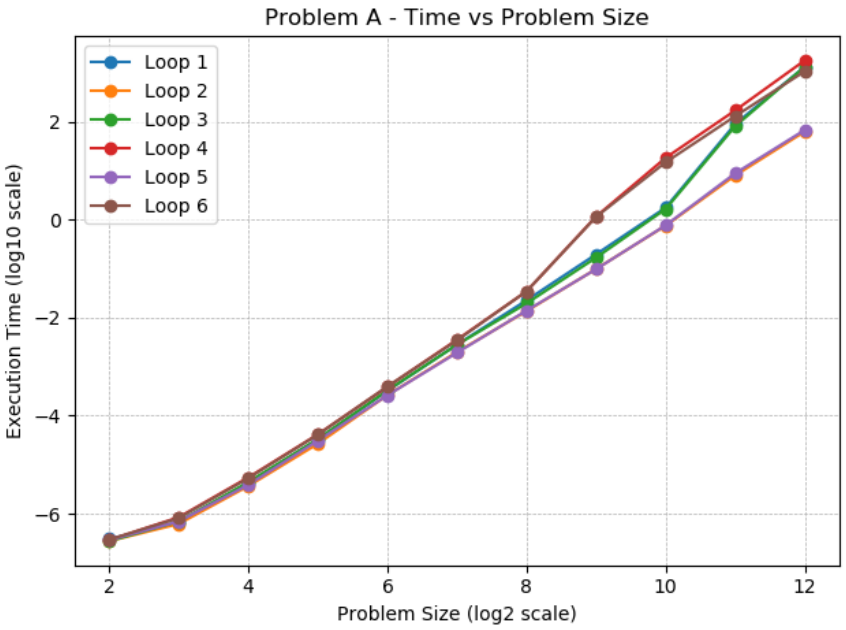

Figure 4: Cluster: MFLOPS vs Problem Size (Problem A)

5.4 Conclusion for Problem A

The performance of conventional matrix multiplication varies significantly with loop ordering due to differences in memory access patterns.

Among the six permutations, Loop Type 2 (i–k–j) and Loop Type 5 (k–i–j) achieved the best performance because they provide better cache locality and reuse of matrix elements.

Loop types such as Loop type 4(j–k–i) and Loop type 6(k–j–i) showed poorer performance due to inefficient memory access and increased cache misses. As problem size increases, performance initially improves but eventually decreases when the matrices exceed cache capacity, making the computation memory-bound. These results demonstrate the importance of loop ordering in optimizing matrix multiplication perform

6 Problem B – Transpose Matrix Multiplication

6.1 Lab Machine

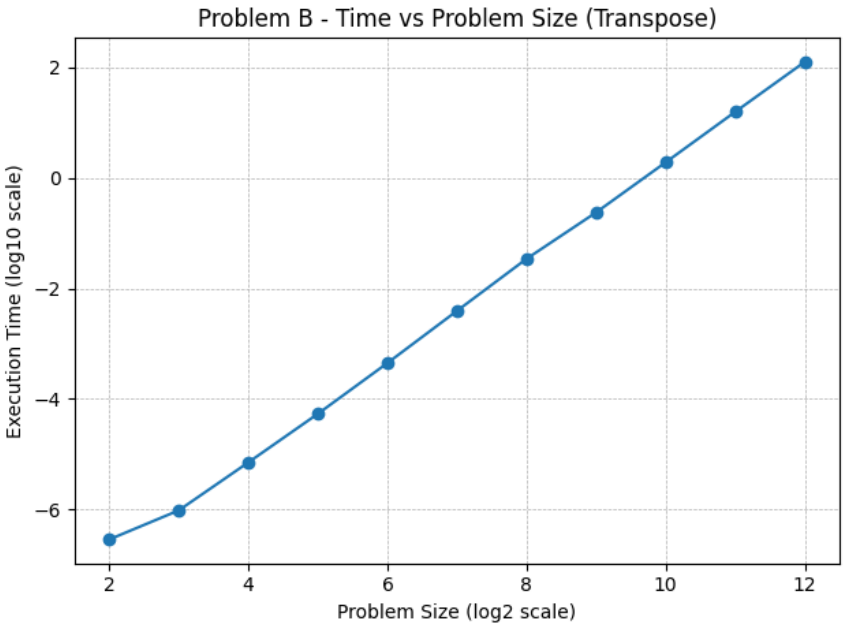

Figure 5: Lab Machine: Runtime vs Problem Size (Problem B)

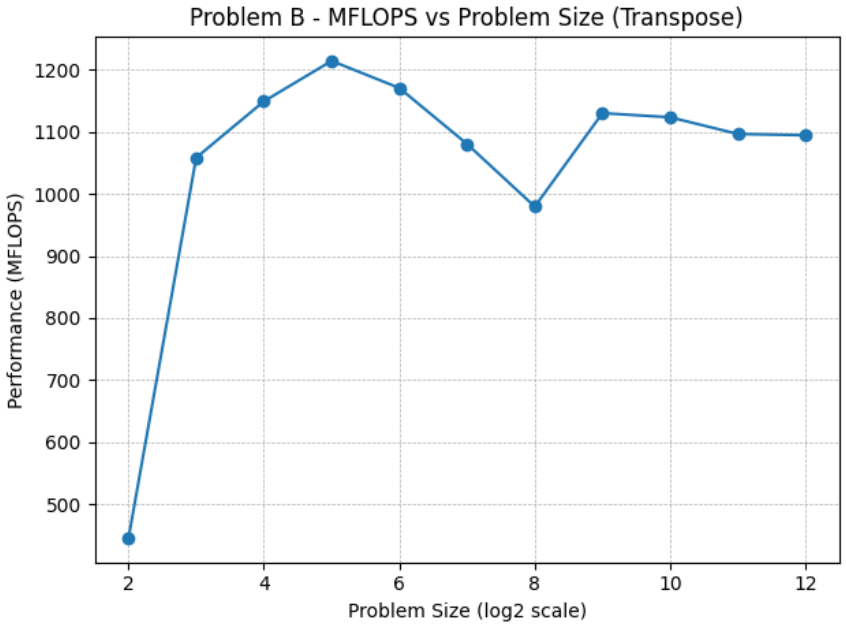

Figure 6: Lab Machine: MFLOPS vs Problem Size (Problem B)

6.2 Cluster

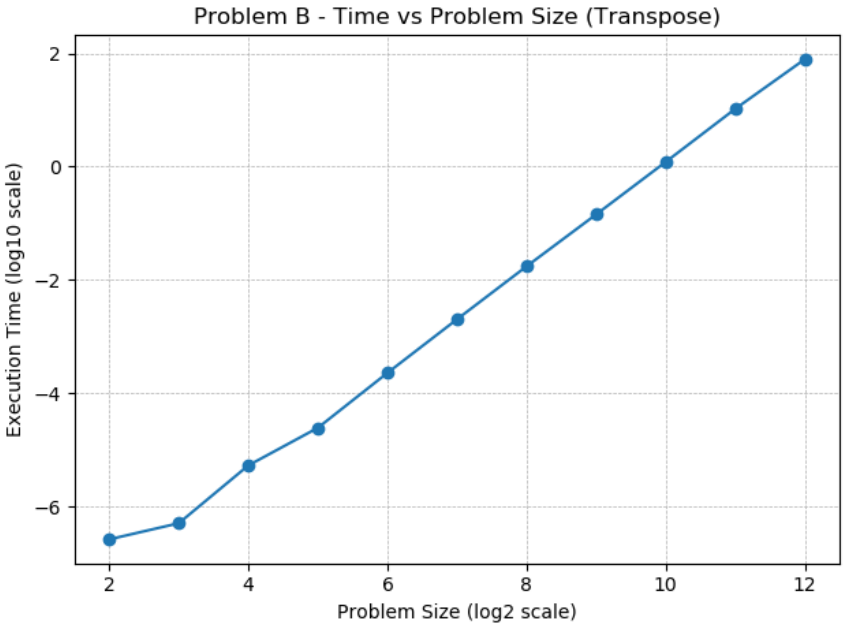

Figure 7: Cluster: Runtime vs Problem Size (Problem B)

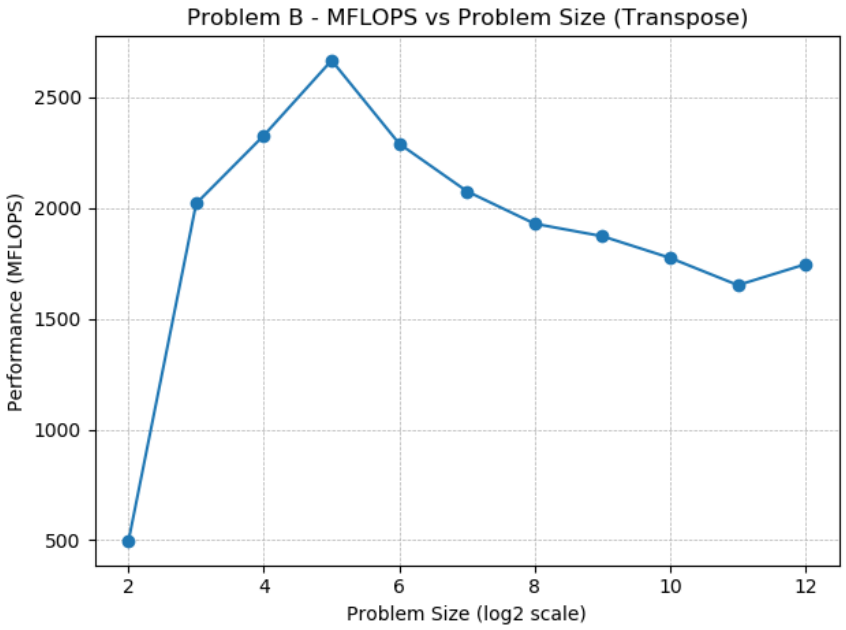

Figure 8: Cluster: MFLOPS vs Problem Size (Problem B)

7 Problem C – Block Matrix Multiplication

7.1 Lab Machine

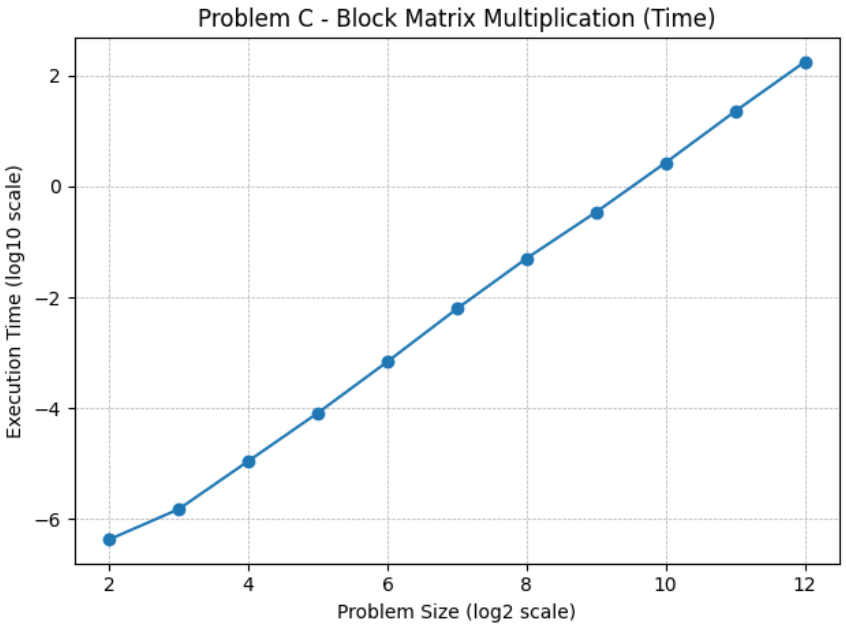

Figure 9: Lab Machine: Runtime vs Problem Size (Problem C)

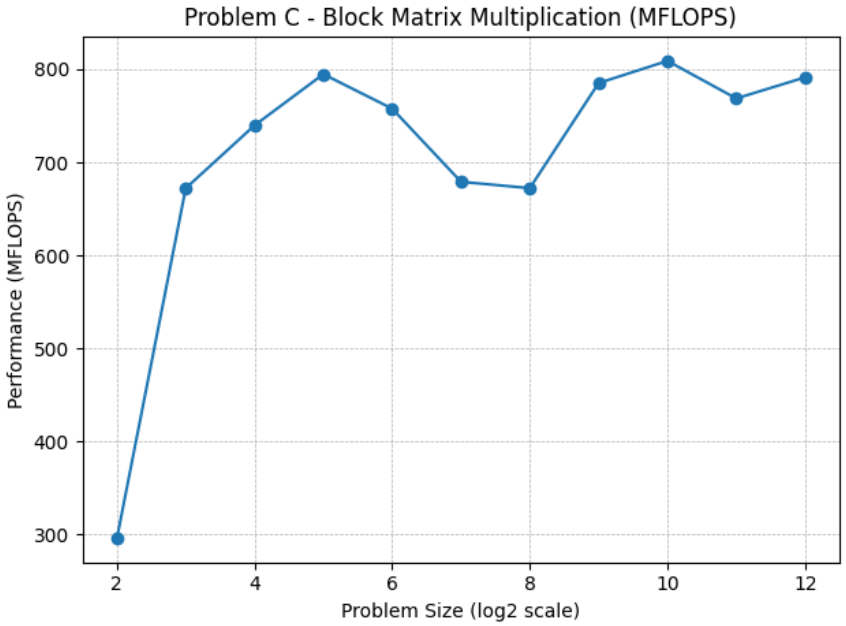

Figure 10: Lab Machine: MFLOPS vs Problem Size (Problem C)

7.2 Cluster

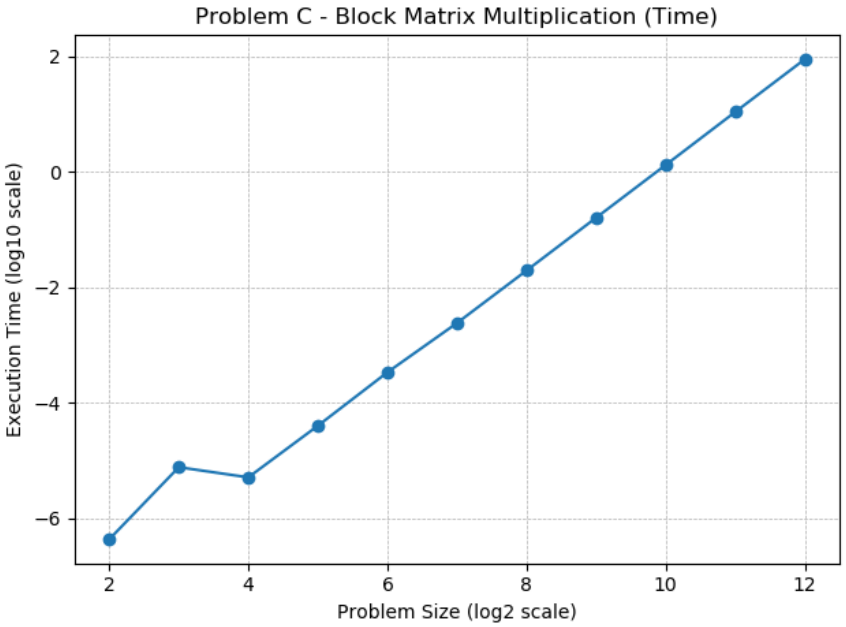

Figure 11: Cluster: Runtime vs Problem Size (Problem C)

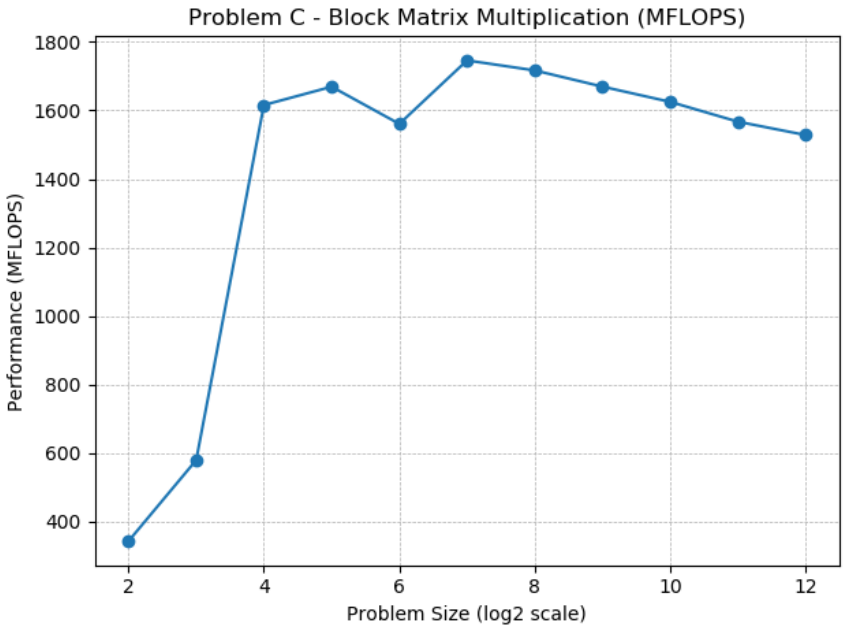

Figure 12: Cluster: MFLOPS vs Problem Size (Problem C)

8 Comparison of All Methods

8.1 Cluster Comparison

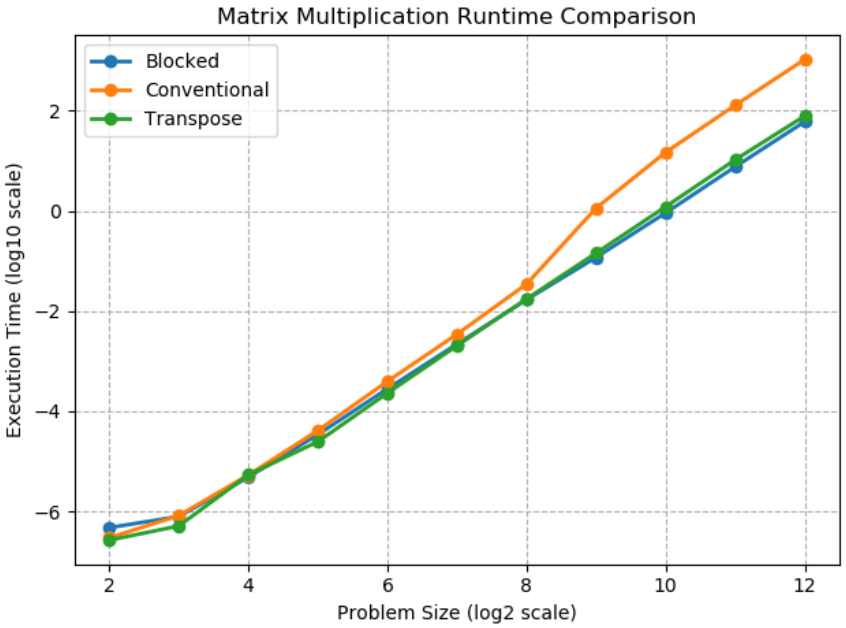

Figure 13: Cluster: Runtime Comparison

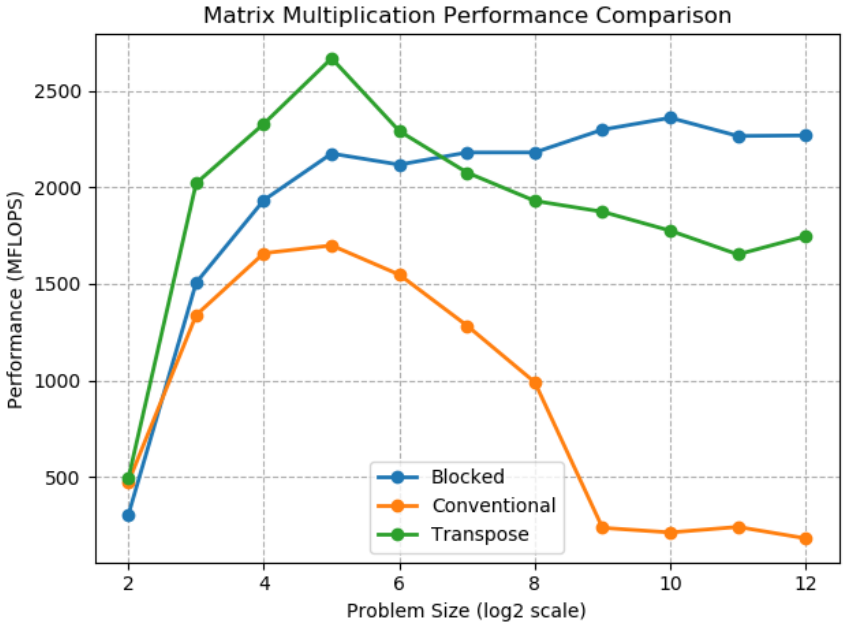

Figure 14: Cluster: MFLOPS Comparison

9 Conclusion

- All matrix multiplication methods have the same theoretical complexity of $O(N^3)$, but their performance differs due to memory access patterns and cache utilization.
- In conventional matrix multiplication, loop orderings such as i–k–j and k–i–j achieved the best performance due to improved cache locality, while other loop orders showed lower performance due to inefficient memory access.
- The transpose-based multiplication improved performance by converting column-wise accesses into row-wise accesses, resulting in better spatial locality and reduced cache misses.
- The block matrix multiplication provided the highest performance for larger problem sizes, as it improves cache reuse by operating on smaller submatrices that fit into cache.
- As problem size increases, performance initially improves but eventually becomes memory-bound due to cache and memory bandwidth limitations.
- Overall, the results demonstrate that optimizing memory access patterns through loop reordering, transpose, and blocking significantly improves matrix multiplication performance, with blocking being the most effective optimization technique.