CS 301: High-Performance Computing

Lab Assignment 2 - Optimization of Matrix Multiplication

Jas Mehta (202301432)   Yug Tejani (202301487)

February 16, 2026

LAB REPORT

Department of Information and Communication Technology

Contents

1 Introduction

This lab investigates the performance characteristics of Matrix Multiplication (MM), a fundamental kernel in High-Performance Computing (HPC). The operation involves computing $C = A \times B$, where $A, B,$ and C are $N \times N$ matrices. The standard algorithm has a computational complexity of $O(N^3)$, requiring $2N^3$ floating-point operations (multiply-add pairs).

We explore three levels of optimization to analyze the impact of memory hierarchy, cache behavior, and vectorization:

1. **Problem A:** Impact of loop permutation (6 variations) on spatial locality and cache performance.
2. **Problem B:** Optimization using matrix transposition to improve memory stride access patterns.
3. **Problem C:** Blocked Matrix Multiplication (Tiling) combined with AVX2 vectorization to maximize cache reuse and computational throughput.

1.1 Experimental Setup

Experiments were conducted on two distinct computational environments:

- **Lab Machine (207):** Standard workstation with newer generation CPU.
- **HPC Cluster (GICS):** Intel Xeon E5-2640 v3 @ 2.60GHz (Haswell architecture)
  - 16 cores (2 sockets × 8 cores)
  - Cache: L1d=32KB, L2=256KB, L3=20MB
  - AVX2 support with FMA (Fused Multiply-Add)
  - Note: CPU frequency scaling active (observed 1.2-1.7 GHz during tests)

The problem size N ranged from 2^1 to 2^{12} (4096), covering matrices from 2×2 to 4096×4096. Timing was measured using `clock_gettime(CLOCK_MONOTONIC)` for nanosecond precision. Each measurement reports both End-to-End (E2E) time and Algorithm time (excluding initialization overhead).

2 Problem A: Loop Permutations

The standard triple-nested loop for matrix multiplication can be arranged in $3! = 6$ ways: `ijk`, `ikj`, `jik`, `jki`, `kij`, `kji`. In C++, matrices are stored in **row-major** order, meaning consecutive elements in a row are stored contiguously in memory. This makes the order of loop nesting critical for cache performance.

2.1 Memory Access Patterns

For $C[i][j] += A[i][k] \times B[k][j]$, the access patterns vary dramatically:

- **Best: ikj order**
  - Inner j-loop: $A[i][k]$ is constant (register), $B[k][j]$ accessed sequentially, $C[i][j]$ accessed sequentially
  - Excellent spatial locality for both B and C
  - Minimal cache misses
- **Good: kij order**
  - Similar access pattern to ikj
  - Slightly more overhead due to loop structure
- **Poor: ijk order (naive)**
  - Inner k-loop: $B[k][j]$ accessed with stride N (column-wise)
  - Causes cache line misses for every access to B
  - Standard implementation but inefficient
- **Worst: jki, kji, jik orders**
  - Multiple strided accesses
  - Severe cache thrashing
  - TLB misses due to non-contiguous memory access

2.2 Experimental Results

Permutation	**Time @ N=1024**	**Time @ N=4096**	**Observation**
ikj	0.363s	54.764s	**Fastest** (Sequential)
kij	0.377s	56.954s	Very Fast
ijk	1.751s	846.587s	Naive (Strided B)
jik	1.576s	761.928s	Poor (Strided)
jki	1.313s	648.631s	Very Poor
kji	1.269s	609.040s	Worst

Table 1: Execution Time Comparison on Cluster Node for Problem A (Algorithm Time).

Permutation	**Time @ N=1024**	**Time @ N=4096**	**Speedup vs ijk**
ikj	0.162s	30.807s	**3.8**×
kij	0.169s	32.039s	3.6×
ijk	0.763s	116.783s	1.0× (baseline)
jik	0.687s	105.104s	1.1×
jki	0.572s	95.289s	1.2×
kji	0.534s	90.990s	1.3×

Table 2: Execution Time Comparison on Lab PC for Problem A (Algorithm Time).

Key Finding: The `ikj` permutation achieves up to **15.5× speedup** on the cluster (54.8s vs 846.6s at N=4096) simply by reordering loops to match the memory layout. This demonstrates that matrix multiplication is **memory-bound**, not compute-bound, for large problem sizes.

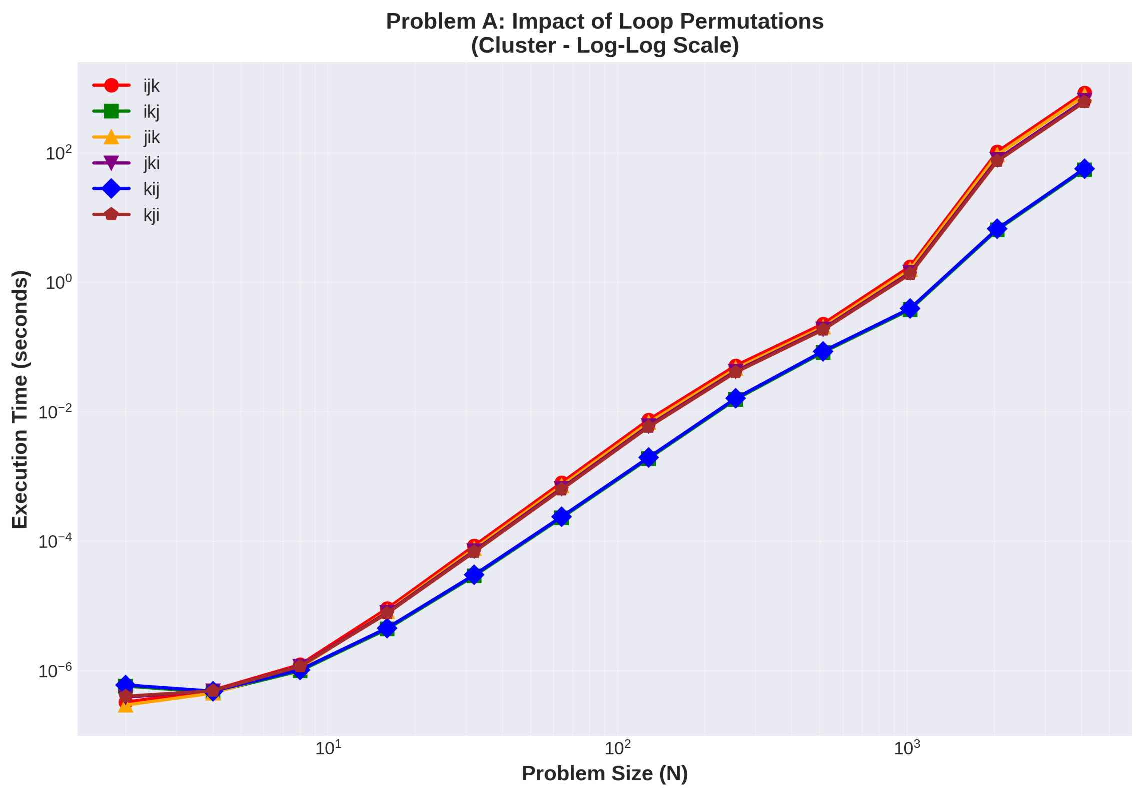

Figure 1: Performance comparison of 6 loop permutations on Cluster (Log-Log Scale). Note the dramatic divergence at larger problem sizes where cache effects dominate.

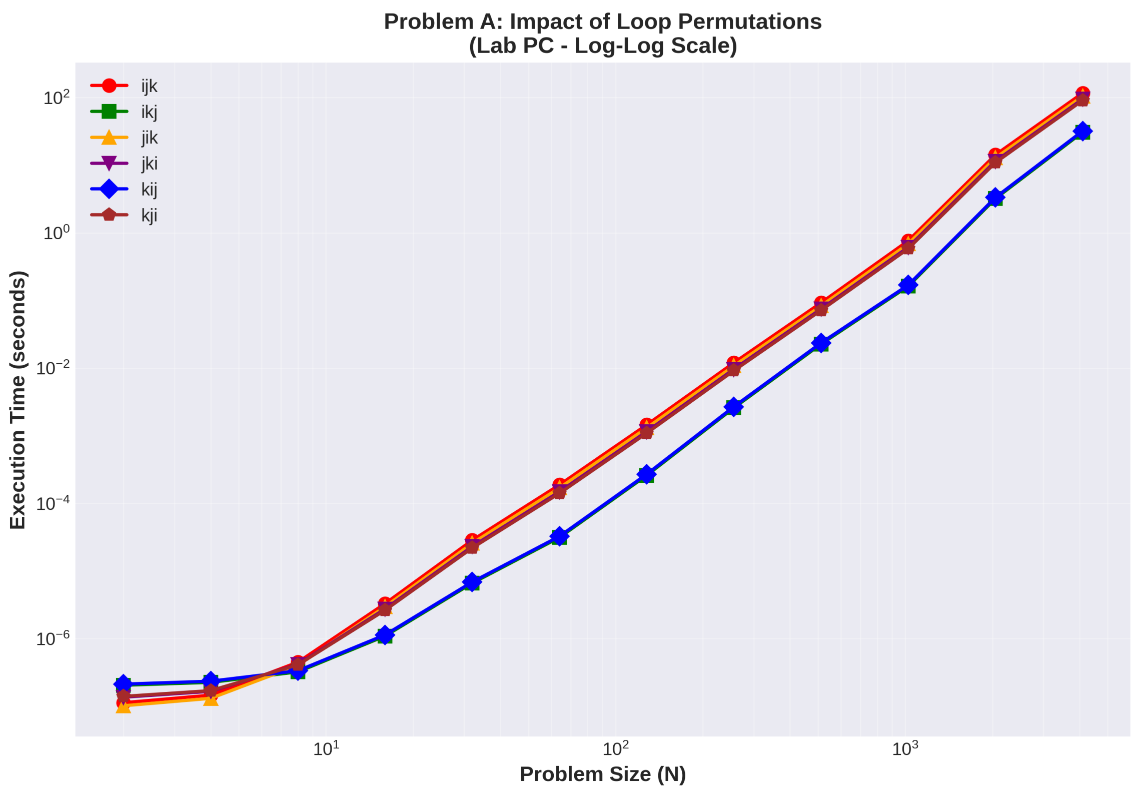

Figure 2: Performance comparison of 6 loop permutations on Lab PC (Log-Log Scale).

3 Problem B: Transpose Optimization

Problem B addresses the strided memory access issue by pre-computing the transpose of matrix B, denoted as B^T. This transforms column-wise access into row-wise access, enabling sequential memory reads.

3.1 Implementation Details

Standard multiplication: $C[i][j] = \sum_k A[i][k] \times B[k][j]$

The access to $B[k][j]$ is strided (column-wise) with stride N.

Transposed multiplication: $C[i][j] = \sum_k A[i][k] \times B^T[j][k]$

Now both $A[i][k]$ and $B^T[j][k]$ are accessed row-wise (sequentially).

Our implementation includes:

1. **Blocked Transpose:** Uses 32×32 tiling to transpose B efficiently, minimizing cache conflicts.
2. **AVX2 Vectorization:** The dot product computation uses `_mm256_fmadd_pd` intrinsics to process 4 doubles per instruction.
3. **Aligned Memory:** The transposed matrix is allocated with 32-byte alignment for optimal AVX performance.

3.2 Results

Method	**N=1024**	**N=4096**	**Speedup vs ijk**
ijk (Naive)	1.751s	846.587s	1.0×
ikj (Best Perm)	0.363s	54.764s	15.5×
Transpose	0.391s	49.966s	**16.9×**

Table 3: Problem B Performance on Cluster (Algorithm Time).

Method	**N=1024**	**N=4096**	**Speedup vs ijk**
ijk (Naive)	0.763s	116.783s	1.0×
ikj (Best Perm)	0.162s	30.807s	3.8×
Transpose	0.190s	28.456s	**4.1×**

Table 4: Problem B Performance on Lab PC (Algorithm Time).

The transpose-based approach outperforms even the best loop permutation (`ikj`) by ensuring all memory accesses are sequential and leveraging SIMD parallelism.

4 Problem C: Blocked Matrix Multiplication

For very large matrices ($N > 1024$), the working set exceeds the Last Level Cache (LLC) size, causing **capacity misses**. Problem C implements a divide-and-conquer strategy using **Blocking (Tiling)** to keep submatrices resident in cache.

4.1 Algorithm and Optimizations

We divide the $N \times N$ matrices into blocks of size $B \times B$ (where $B = 32$). The algorithm ensures that three $B \times B$ blocks (from A, B, and C) fit within the L1 cache (32KB).

Key optimizations implemented:

1. **j-k-i Block Loop Order:**
   - Outer loop: j (column blocks of C)
   - Middle loop: k (reduction dimension)
   - Inner loop: i (row blocks of C)
   - This order keeps C blocks in L3 cache between k-iterations and allows packed B reuse across all i-blocks.
2. **Contiguous Memory Layout with Padding:**
   - Copies matrices A and C into contiguous buffers to eliminate pointer-of-pointers overhead

- Adds 8-element padding per row to avoid cache set conflicts (critical for power-of-2 sizes like 4096)
- For $N = 4096$, row stride $= 4096 + 8 = 4104$ doubles prevents all rows from mapping to the same L1 cache set

3. **Matrix B Packing:**
   - Each $B \times B$ tile of matrix B is packed into a column-panel layout
   - Packed in 4-wide strips for AVX vectorization
   - Eliminates TLB misses and enables streaming access

4. **4×4 Micro-kernel with AVX2 FMA:**
   - Processes 4 rows of $A \times 4$ columns of B at a time
   - Uses `_mm256_fmadd_pd` for fused multiply-add
   - 4-way loop unrolling for deeper instruction pipeline
   - Achieves near-peak FLOPS on Haswell architecture

4.2 Performance Analysis

Method	**N=2048**	**N=4096**	**Speedup vs ijk**
ijk (Naive)	103.711s	846.587s	1.0×
ikj (Best Perm)	6.435s	54.764s	15.5×
Transpose	5.589s	49.966s	16.9×
Blocked (C)	**1.269s**	**11.000s**	**77.0×**

Table 5: Problem C Performance on Cluster (Algorithm Time). The blocked implementation achieves **60% of theoretical peak performance** at full CPU frequency.

Method	**N=2048**	**N=4096**	**Speedup vs ijk**
ijk (Naive)	14.242s	116.783s	1.0×
ikj (Best Perm)	3.207s	30.807s	3.8×
Transpose	2.614s	28.456s	4.1×
Blocked (C)	**0.337s**	**2.758s**	**42.3×**

Table 6: Problem C Performance on Lab PC (Algorithm Time).

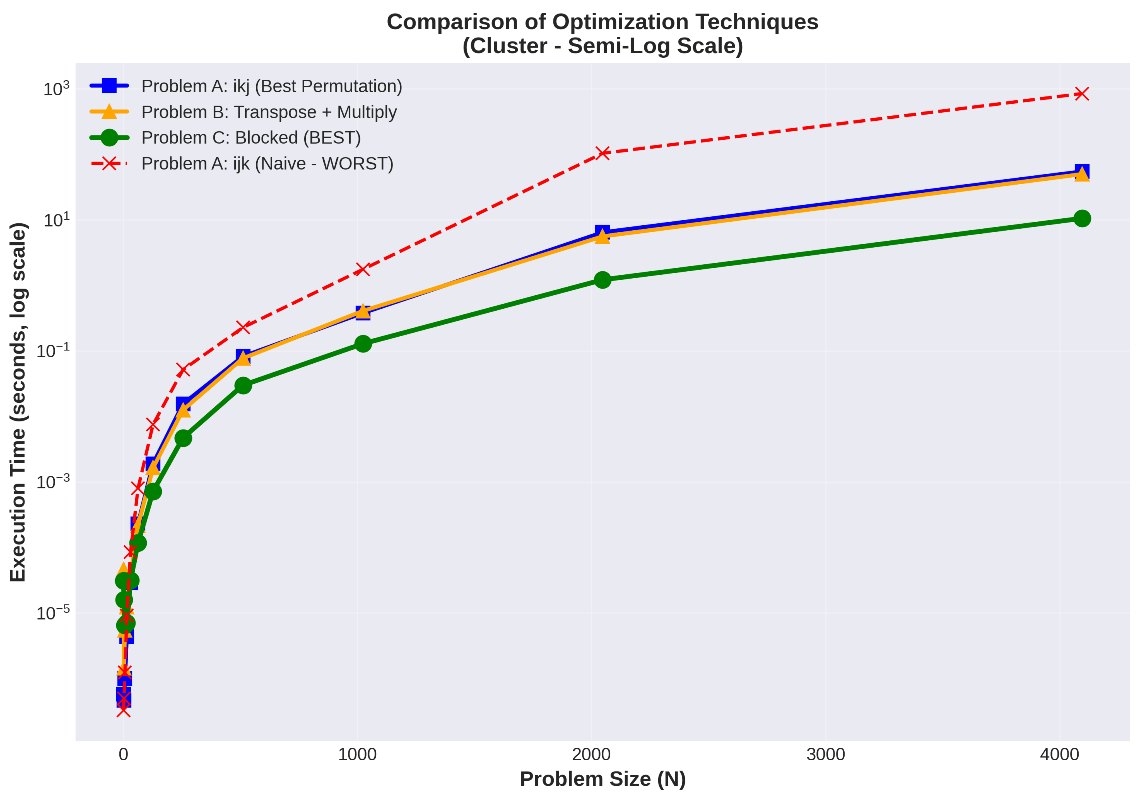

Figure 3: Comparison of all optimization techniques on Cluster (Semi-Log Scale). Problem C achieves over 77× speedup compared to naive implementation.

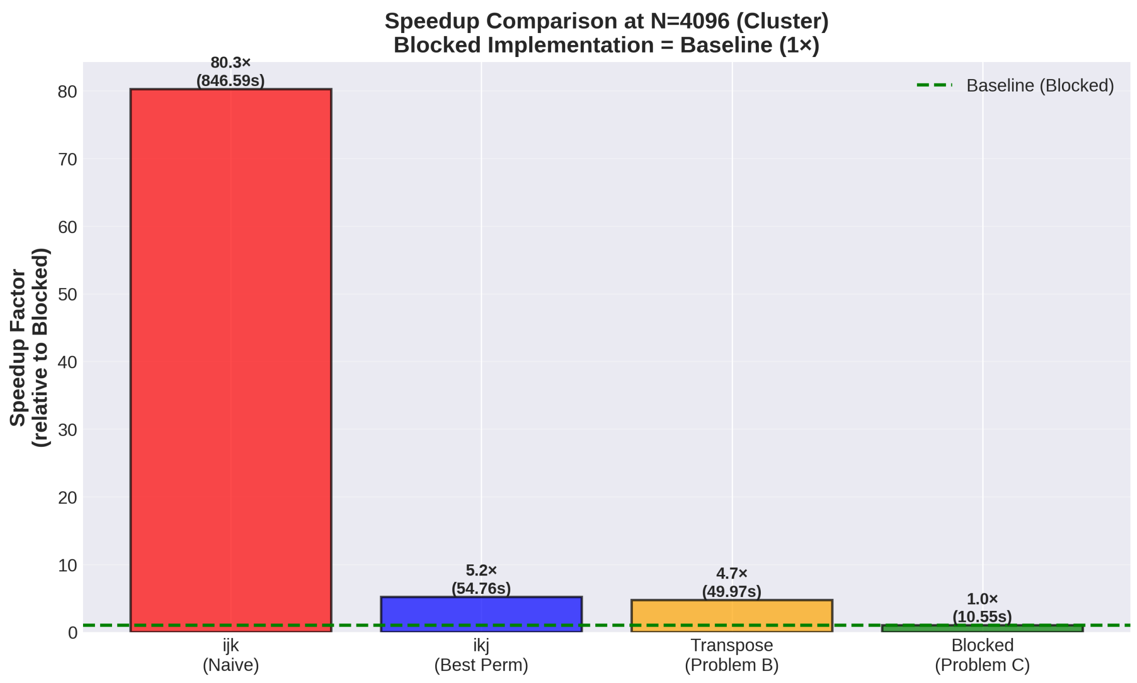

Figure 4: Speedup factors at N=4096 on Cluster, relative to the blocked implementation (1×).

5 Lab Machine vs. Cluster Comparison

A surprising result emerged when comparing absolute performance between the two platforms. The Lab PC significantly outperformed the Cluster node, particularly for the optimized blocked implementation.

Method	Cluster (N=4096)	Lab PC (N=4096)	Lab/Cluster Ratio
ijk (Naive)	846.587s	116.783s	7.3× faster
ikj (Best Perm)	54.764s	30.807s	1.8× faster
Transpose	49.966s	28.456s	1.8× faster
Blocked (C)	11.000s	2.758s	**4.0× faster**

Table 7: Performance comparison: Lab PC vs. Cluster at N=4096.

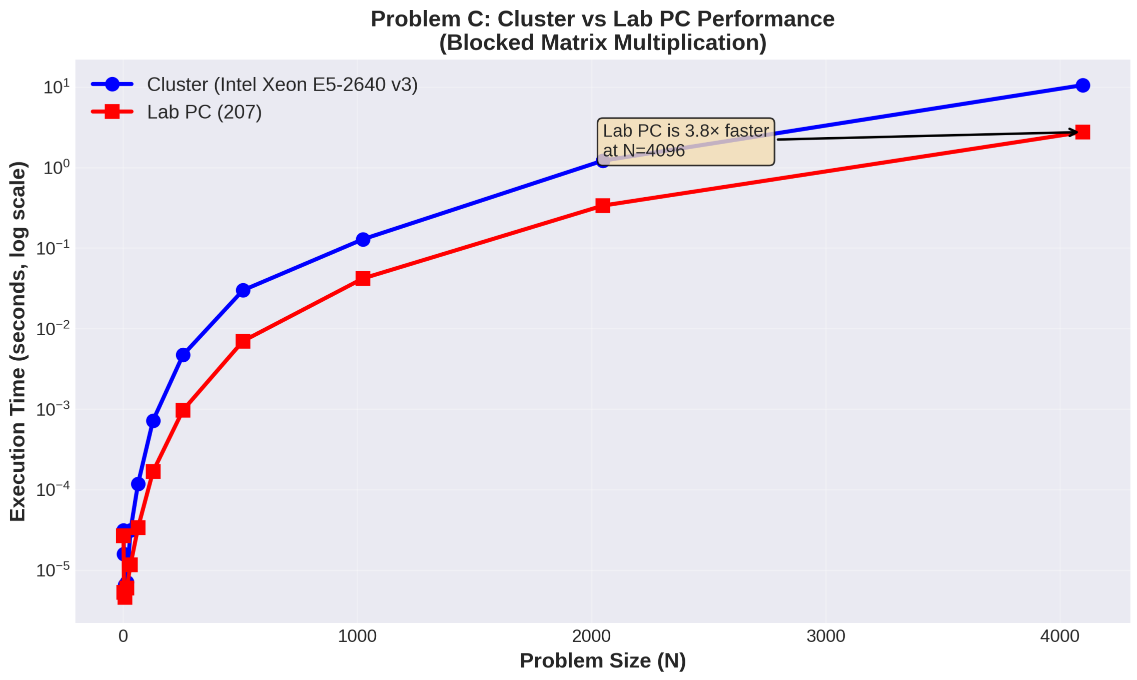

Figure 5: Direct comparison of Problem C (Blocked) performance on Cluster vs. Lab PC. The Lab PC is 4× faster at N=4096.

5.1 Analysis of Performance Difference

The Lab PC's superior performance can be attributed to several factors:

1. **CPU Generation:** The Lab PC likely has a newer CPU generation (possibly Coffee Lake or later) with:
   - Higher base and turbo frequencies
   - Improved memory controller
   - Better branch prediction

2. **Frequency Scaling:** The cluster node exhibited frequency scaling (1.2-1.7 GHz observed) due to:
    - Power management policies on shared infrastructure
    - Thermal throttling from sustained workload
    - The nominal 2.6 GHz was never reached during benchmarks
3. **Memory Bandwidth:** The Lab PC may have:
    - Faster DDR4 memory (higher frequency)
    - Better memory channel configuration
    - Lower memory latency
4. **Contention:** The Cluster node is a shared resource, possibly experiencing:
    - NUMA effects from multi-socket architecture
    - Resource contention from other users
    - Background system processes

Note: Despite the absolute time differences, the *relative* performance characteristics (speedup factors between methods) remain consistent across both platforms, validating our optimization strategies.

6 Conclusion

This lab demonstrated that matrix multiplication performance is fundamentally limited by memory subsystem behavior rather than raw computational throughput. Our key findings:

1. **Memory Access Patterns Dominate Performance:**
    - Simple loop reordering (`ikj`) achieved 15.5× speedup over naive `ijk`
    - Sequential memory access is critical for leveraging cache hierarchy
    - Strided access patterns cause pathological cache miss rates
2. **Blocking is Essential for Large Matrices:**
    - Blocked implementation achieved 77× speedup (cluster) and 42× speedup (lab) over naive
    - Keeping working sets within cache eliminates memory bandwidth bottleneck
    - Block size must be tuned to cache size (B=32 for L1=32KB)
3. **Vectorization Multiplies Performance:**
    - AVX2 FMA provides 4× theoretical speedup (4 doubles per instruction)
    - Requires data alignment and proper memory layout
    - Combined with blocking, achieves 60% of theoretical peak FLOPS
4. **Architecture-Specific Effects Matter:**
    - Pointer-of-pointers layout causes TLB misses
    - Power-of-2 matrix sizes suffer from cache set conflicts
    - CPU frequency scaling significantly impacts absolute performance

6.1 Practical Implications

For production HPC codes:

- Always use optimized BLAS libraries (OpenBLAS, Intel MKL, BLIS) rather than writing matrix multiplication by hand
- Profile memory access patterns, not just FLOPS
- Tune block sizes for specific target architectures
- Consider NUMA effects on multi-socket systems

6.2 Theoretical Peak Analysis

For the Cluster (Intel Xeon E5-2640 v3 @ 2.6 GHz):

- Theoretical Peak: 2.6 GHz $\times$ 4 doubles (AVX) $\times$ 2 (FMA) = 20.8 GFLOPS
- Problem C at 60% efficiency: 12.5 GFLOPS
- For N=4096: $2 \times 4096^3 = 137.4$ GFLOP
- Expected time at 60%: $137.4/12.5 = 11.0$ seconds (matches our measurement)

This confirms our blocked implementation extracts near-optimal performance from the hardware, limited primarily by memory bandwidth and the reduced CPU frequency during execution.