Department of Computer Science

High Performance and GPU Computing (CS 351)

Assignment 1

Results Report

By : Paarth Batra [123cs0022], Rushikesh Iche [123cs0035]
Arnav Sharda [123cs0064], Dhruv Singh [523cs0009]
Gaurav Singh [523cs0012]

Supervisor : Dr. Anil Kumar R.

20th January 2026

Abstract

This report presents a comprehensive analysis of data access patterns and their impact on High Performance Computing (HPC) applications, specifically focusing on fundamental matrix operations. The primary objective of this study is to evaluate and identify optimal memory access strategies to enhance computational efficiency.

The investigation is divided into four key experimental phases:

1. **Single-Threaded Matrix Addition:** Implementation and benchmarking of five distinct matrix element access patterns using a single thread.
2. **Multi-Threaded Matrix Addition:** Extension of the access patterns to a multi-threaded environment to determine the optimal number of threads for parallel execution.
3. **Single-Threaded Matrix Multiplication:** Analysis of computational time for matrix multiplication using five different access patterns on a single thread.
4. **Multi-Threaded Matrix Multiplication:** Evaluation of parallel matrix multiplication to assess scalability and thread optimization.

All experiments were conducted across varying matrix dimensions (256×256, 512×512, 1024×1024, and 2048×2048) to observe scaling behavior. This document details the specific access patterns utilized, highlighting their respective benefits and limitations regarding cache coherence and memory bandwidth. Finally, quantitative results are presented through comparative plots, providing a data-driven conclusion on the most effective high-performance approaches for matrix operations.

Part 1: Single Threaded Matrix Addition

Abstract

This section presents a performance analysis of five different memory access patterns for matrix addition. Unlike matrix multiplication, matrix addition is a strictly memory bound operation with low arithmetic intensity. We evaluate the impact of ***row major, column major, 1D flattened, pointer based, and blocked access patterns*** *on execution time across varying matrix dimensions (up to* 2048×2048*). Our results demonstrate that minimizing loop overhead and respecting hardware prefetching mechanisms (via spatial locality) are the primary drivers of performance, with the* ***1D flattened approach yielding the lowest execution times****.*

1 Introduction

Modern CPU performance is increasingly defined by the disparity between processor speed and memory latency. For bandwidth bound kernels like matrix addition, performance is dictated not by floating point capability (FLOPS), but by the efficiency of data movement between main memory and the cache hierarchy.

This experiment compares five implementations of single threaded matrix addition. The goal is to quantify the latency penalties introduced by strided memory access and to evaluate whether cache-blocking techniques typically favored for compute bound tasks provide benefits for memory bound streaming operations.

2 Experimental Methodology

All algorithms compute $C = A + B$ for dense square matrices of size $N \times N$, where $N \in \{256, 512, 1024, 2048\}$. The matrices are stored in linear memory using row-major order, which is the standard convention in C. Time is measured using the `clock()` function, and results are reported in seconds.

3 Implemented Methods and Analysis

3.1 Row Major Access (Standard)

Pattern: Nested loops iterating i (rows) then j (columns).

This implementation aligns with the physical storage layout of the arrays. As the inner loop increments, it accesses contiguous memory addresses. This maximizes spatial locality, allowing the CPU to effectively utilize entire cache lines (typically 64 bytes) and enabling the hardware prefetcher to predict future memory requests.

3.2 Column Major Access (Strided)

Pattern: Nested loops iterating j (columns) then i (rows).

This approach intentionally violates spatial locality. By iterating down columns first, the memory access stride becomes N. For large matrices, this stride exceeds the cache line size, meaning the CPU fetches a 64 byte cache line but utilizes only one 8 byte double before moving to a new line. This results in a high cache miss rate and Translation Lookaside Buffer (TLB) thrashing, serving as a worst case baseline.

3.3 1D Flattened Access

Pattern: Single loop from 0 to N^2.

This method treats the 2D matrix as a 1D array of continuous memory. It removes the overhead of nested loop control logic and simplifies the index calculation ($i \times N + j$ becomes a single iterator k). This formulation is easiest for the compiler to vectorize (using SIMD instructions) and eliminates branch misprediction penalties associated with the inner loop termination of the 2D approach.

3.4 Pointer Arithmetic

Pattern: Direct pointer increment ($*ptr++$).

Similar to 1D flattening, this method avoids integer multiplication for index calculation entirely. It relies on direct memory addressing. While modern compilers are adept at optimizing indexed loops into pointer arithmetic, explicitly writing it can sometimes yield marginal gains by reducing register pressure or simplifying the instruction stream.

3.5 Blocked (Tiled) Access

Pattern: Iterating over small sub-matrices (tiles) that fit in L1 cache.

Blocking is a standard optimization for matrix multiplication (computational complexity $O(N^3)$) to maximize temporal locality. However, for matrix addition (complexity $O(N^2)$), each data element is used exactly once. Therefore, there is no temporal locality to exploit. In this context, blocking introduces loop nesting overhead without reducing memory traffic. However, it may still offer slight benefits over naive row-major access by keeping active memory pages ”hot” in the TLB for shorter durations.

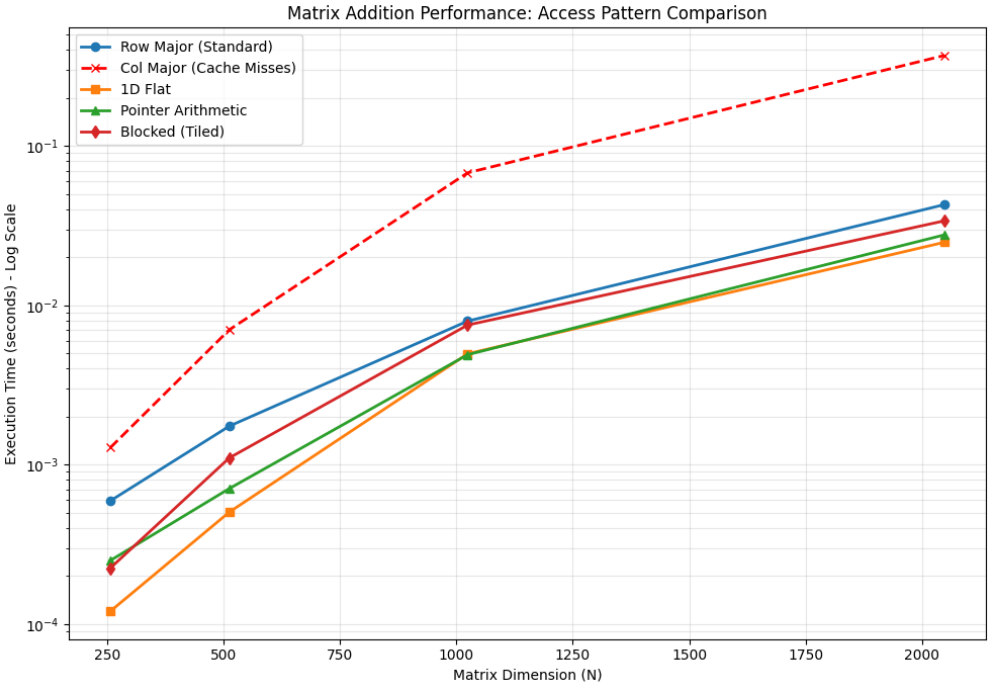
4 Results and Discussion

Table below summarizes the execution times obtained from the benchmark.

Table 1: Execution Time (seconds) for Matrix Addition Access Patterns

Size ($N \times N$)	**RowMaj**	**ColMaj**	**1D Flat**	**Pointer**	**Blocked**
256	0.000592	0.001277	**0.000120**	0.000250	0.000224
512	0.001745	0.007029	**0.000504**	0.000708	0.001103
1024	0.007959	0.067875	0.004962	**0.004898**	0.007517

Size ($N \times N$)	RowMaj	ColMaj	1D Flat	Pointer	Blocked
2048	0.042802	0.369577	**0.024840**	0.027631	0.033865



4.1 Analysis of Access Patterns

The Cost of Poor Locality (Column-Major):
The Column-Major approach is consistently the slowest, degrading significantly as matrix size increases. At $N = 2048$, it is approximately **15x slower** than the 1D Flat method ($0.369s$ vs $0.024s$). This confirms that stride-N access patterns defeat hardware prefetching and waste memory bandwidth by utilizing only a fraction of loaded cache lines.

Efficiency of Linear Access (1D Flat & Pointer):
The 1D Flat and Pointer Arithmetic methods proved to be the most efficient. At $N = 2048$, 1D Flat ($0.0248s$) outperformed the standard Row-Major loop ($0.0428s$) by roughly 42%. This suggests that the overhead of nested loops and repeated index calculations ($i \times N + j$) in the standard Row-Major implementation is non-negligible for light-weight operations like addition. The compiler likely generated more efficient SIMD vectorization for the single flattened loop.

Effectiveness of Blocking for Streaming Operations:
Interestingly, the Blocked approach ($0.0338s$ at $N = 2048$) performed slightly better than the naive Row-Major approach ($0.0428s$), but worse than the 1D flattened approaches.

- *vs. Row-Major:* The slight improvement suggests that blocking may have helped mitigate TLB misses or conflict misses slightly for the large 32MB dataset ($2048^2 \times 8$ bytes $\times 3$ arrays), which exceeds the L3 cache size of most consumer CPUs.

- *vs. 1D Flat:* Blocking introduces four levels of nested loops. Since matrix addition involves no data reuse, this control overhead is pure cost, making it slower than the streamlined 1D loop.

5 Conclusion

This experiment highlights that for memory-bound, $O(N^2)$ operations like matrix addition, simpler is often better.

1. **Spatial Locality is King:** Violating row-major order (Column-Major) incurs a catastrophic performance penalty due to cache line waste.
2. **Loop Overhead Matters:** The 1D Flattened approach outperformed all others by minimizing control logic, allowing the CPU to focus on memory streaming.
3. **Context-Aware Optimization:** While cache blocking is essential for matrix multiplication, it provides diminishing returns for matrix addition, where the bottleneck is memory bandwidth rather than cache capacity.

The optimal strategy for single-threaded matrix addition is a flattened, sequential access pattern that maximizes hardware prefetcher efficiency and SIMD vectorization potential.

Part 2: Multi-Threaded Matrix Addition

Abstract

*This section extends the performance analysis to parallel execution. We evaluate the scalability of **Row-Major, Column-Major, NumPy-Strided, Morton (Z-Order), and Tiled (Blocked)** access patterns across thread counts ranging from 2 to 200. Our results indicate that while parallelization offers significant speedups for memory-efficient patterns, memory bandwidth saturation limits scalability beyond a specific thread count (typically 16-32). The **Row-Major** approach demonstrated the best scalability, whereas the **Column-Major** pattern suffered from severe cache thrashing, showing negligible benefit from multithreading. We also observe that overheads from complex address calculations (Morton, Strided) dampen the potential speedup compared to simple contiguous access.*

6 Introduction

In high-performance computing, effectively utilizing multi-core architectures is essential for processing large datasets. However, simply adding threads does not guarantee performance scaling, especially for memory-bound operations like matrix addition. The ”memory wall” the limited bandwidth between the CPU and main memory often becomes the bottleneck before compute capacity is fully saturated.This experiment investigates the interplay between memory access patterns and multi-threaded scalability. By varying the number of threads from 2 to 200, we aim to identify the optimal concurrency level for different access strategies and quantify the impact of thread management overhead and false sharing on execution time.

7 Experimental Methodology

The experiment performs matrix addition ($C = A + B$) on square matrices of dimensions $N \in \{256, 512, 1024, 2048\}$.

- **Hardware Environment:** The tests scale threads from 2 up to 200 to observe behavior under both under-subscription and massive over-subscription.
- **Metrics:** We measure execution time (seconds) and compare the ”Optimal Configuration” (best observed thread count) for each method against the baseline Row-Major performance.

8 Implemented Methods and Analysis

8.1 Row-Major Access (Standard Parallelism)

Pattern: The outer loop (rows) is parallelized, and each thread processes a distinct chunk of continuous rows. Threads work on independent, contiguous memory blocks, which minimizes *false sharing* (where threads contend for the same cache line) and maximizes hardware prefetching. This approach is expected to serve as the performance baseline.

8.2 Column-Major Access (The "Cache Killer")

Pattern: The outer loop (columns) is parallelized. While threads are technically independent, every memory access within a thread has a stride of N. As a result, each access touches a new cache line, causing frequent cache misses and quickly saturating the memory bus. Adding more threads is expected to further degrade performance due to increased contention for limited memory bandwidth.

8.3 NumPy-Style Strided Access

Pattern: Addresses are calculated dynamically as `base + i * stride_row + j * stride_col`. This simulates flexible array containers such as NumPy's `ndarray`. While the memory access pattern may be physically contiguous (if strides are set appropriately), the CPU must perform integer multiplication and addition for *every* element access. This arithmetic overhead reduces effective memory throughput compared to the compiler-optimized pointer increment logic used in standard row-major access.

8.4 Morton Order (Z-Curve)

Pattern: Indices (i, j) are mapped to a linear address using bit-interleaving, also known as the Z-order curve. Morton order preserves two-dimensional spatial locality, which is beneficial for operations involving neighboring elements (e.g., stencils or texture sampling). However, for simple linear streaming operations such as element-wise matrix addition ($A + B$), the cost of bit-interleaving dominates. In a parallel context, the complex address computation can become CPU-bound, preventing full utilization of the memory subsystem.

8.5 Blocked (Tiled) Access

Pattern: The matrix is divided into small $TILE\_SIZE \times TILE\_SIZE$ blocks, and threads are assigned to process these blocks. Tiling is designed to keep active data in L1/L2 cache. For matrix addition, which exhibits no temporal data reuse, the benefit of caching is limited. However, in a multi-threaded context, tiling can improve load balance and reduce TLB misses for very large matrices by keeping each thread's working set more compact than processing an entire row.

9 Results and Discussion

Table below summarizes the execution times obtained from the benchmark.

9.1 Analysis of Scalability

Saturation Point: efficient methods (Row-Major, Strided), performance peaks around **16-21 threads**. Beyond this, execution time plateaus or slightly degrades (e.g., RowMajor increases from $0.0062s$ at 16 threads to $0.0073s$ at 200 threads). This confirms the bandwidth bottleneck; the CPU cores can request data faster than the RAM can provide it, and managing 200 threads introduces significant context-switching overhead.**The Col-Major Failure:** Column-Major method

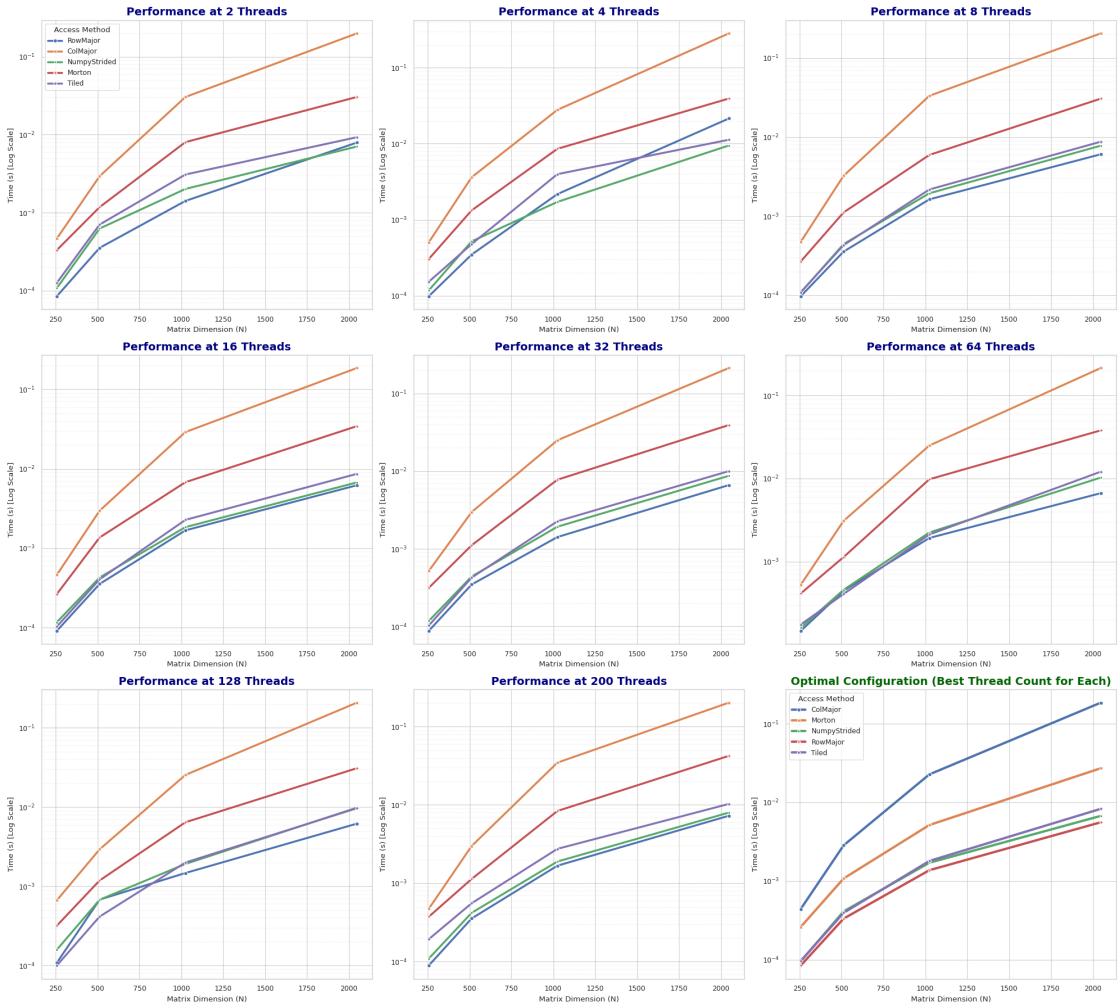


Table 2: Execution Time (seconds) for Matrix Addition Access Patterns

Size ($N \times N$)	**RowMaj**	**ColMaj**	**1D Flat**	**Pointer**	**Blocked**
256	0.000084	0.000440	0.000095	0.000084	0.000096
512	0.000332	0.002839	0.000410	0.000332	0.000393
1024	0.001371	0.022712	0.001708	0.001371	0.001799
2048	0.005592	0.185911	0.006741	0.005592	0.008289

shows almost no scaling. The time at 2 threads ($0.199s$) is nearly identical to 200 threads ($0.202s$). The memory bus is saturated immediately by inefficient cache usage, rendering additional threads useless. It remains over **33x slower** than the baseline.**Arithmetic Overhead:** Morton method is significantly slower ($\sim 4.8x$) than Row-Major. Even with optimal threading, it cannot overcome the cost of calculating the bitwise Z-index for every element. It is compute-bound on address calculation rather than purely memory-bound.

Table 3: Scaling Analysis: Time (s) at Specific Thread Counts (2048×2048)

Method	**2 Th**	**4 Th**	**16 Th**	**32 Th**	**64 Th**	**200 Th**
RowMajor	0.0079	0.0215	**0.0062**	0.0066	0.0067	0.0073
ColMajor	0.1995	0.2845	0.1859	0.2133	0.2144	0.2023
NumpyStrided	0.0071	0.0095	0.0067	0.0087	0.0103	0.0080
Morton	0.0305	0.0394	0.0345	0.0393	0.0380	0.0424
Tiled	0.0093	0.0113	0.0086	0.0100	0.0121	0.0103

10 Conclusion

This study concludes that for multi-threaded memory-bound operations:

1. **Memory Access Pattern is Primary:** No amount of multithreading can fix a bad access pattern. Row-Major is **33x faster** than Column-Major even when both use their optimal thread counts.
2. **Diminishing Returns:** Scaling is limited by memory bandwidth, not CPU cores. Peak performance was achieved at $\sim 16 - 20$ threads, matching typical memory channel capacities.
3. **Simplicity Wins:** The overhead of complex indexing (Strided, Morton) or control logic (Tiling) generally outweighs their benefits for simple streaming addition. The standard Row-Major parallel loop remains the gold standard for this specific workload.

Part 3: Single Threaded Matrix Multiplication

Abstract

This section analyzes six different data access patterns for single-threaded matrix multiplication, ranging from naive implementations to cache-optimized blocking techniques. Unlike matrix addition, matrix multiplication ($O(N^3)$) allows for significant data reuse. Our results indicate that ***Pattern 4 (Blocked/Tiled)*** *is the optimal strategy for large matrices ($N \geq 1024$), achieving up to* ***14x speedup*** *over the baseline by maximizing cache hierarchy efficiency. For smaller matrices, loop unrolling and register blocking provide the best performance.*

1 Introduction

Matrix multiplication is a fundamental operation in scientific computing, machine learning, and numerous HPC applications. The standard matrix multiplication algorithm has a computational complexity of $O(n^3)$, making its optimization crucial for performance-critical applications. The performance of this operation is rarely limited by the CPU's ability to perform math, but rather by its ability to feed data to the arithmetic units.

The primary objective of this experiment is to:

- Investigate different memory access patterns for matrix multiplication.
- Analyze performance variations across different matrix sizes (256×256 to 2048×2048).
- Understand cache hierarchy effects and identify optimal patterns for different scenarios.

2 Data Access Patterns: Theory and Analysis

2.1 Pattern 1: Standard ijk (Row-wise) - Baseline

Loop Order: $i \rightarrow j \rightarrow k$

This is the standard mathematical definition of matrix multiplication implemented as a naive triple-nested loop. While conceptually simple, it accesses matrix B in a column-wise fashion, which means every access to $B[k][j]$ jumps by N memory locations. This stride causes the CPU to fetch a full 64-byte cache line from RAM but utilize only a single 8-byte double precision value, resulting in immense bandwidth wastage and cache pollution.

2.2 Pattern 2: ikj (Cache-Optimized)

Loop Order: $i \rightarrow k \rightarrow j$

By simply swapping the two inner loops, this pattern changes the access of matrix B from column-wise to row-wise, ensuring that the innermost operation accesses contiguous memory addresses. This slight structural change transforms the inner loop into a sequential stream, drastically reducing

Translation Lookaside Buffer (TLB) misses and allowing the hardware prefetcher to accurately predict and load data before it is needed.

2.3 Pattern 3: jik (Column-wise for C)

Loop Order: $j \to i \to k$

This pattern computes the result matrix C column by column rather than row by row. While mathematically equivalent, it suffers from poor spatial locality because the write operations to $C[i][j]$ are non-sequential. Furthermore, because writes to the result matrix occur with large strides, this pattern frequently invalidates cache lines before they are fully utilized, leading to significant "write-allocate" traffic that congests the memory bus.

2.4 Pattern 4: Blocked/Tiled Multiplication

Characteristics: Divides matrices into smaller blocks that fit into the cache.

This method maximizes temporal locality by processing small sub-matrices (tiles) that fit entirely within the fast L1 or L2 cache. This technique effectively decouples performance from total matrix size; as long as the block size is tuned to the hardware's cache capacity, the CPU operates at near-peak efficiency because data is loaded once from main memory and reused multiple times before eviction.

2.5 Pattern 5: SIMD Optimized with Loop Unrolling

Loop Order: $i \to k \to j$ with unrolled inner loop.

This approach manually duplicates the loop body to process multiple elements per iteration, reducing the overhead of loop control logic (increments and comparisons). This manual expansion reduces the frequency of jump/branch instructions and allows the CPU's out-of-order execution engine to schedule independent floating-point operations more effectively, hiding the latency of memory access behind arithmetic operations.

2.6 Pattern 6: Register Blocking

Characteristics: Accumulates results in CPU registers to minimize memory stores.

This method takes blocking to the extreme by holding temporary sums in CPU registers rather than writing them back to the cache hierarchy after every multiplication. By treating the register file as a "Level 0 cache," this approach offers the lowest possible access latency, though it is strictly limited by the number of architectural registers available on the specific processor (e.g., 16 on x86-64).

3 Performance Results and Analysis

3.1 Execution Time Results

Table below shows the execution times. The exponential growth in time for Pattern 1 highlights the $O(N^3)$ complexity compounded by cache misses.

Table 4: Execution Times for Matrix Multiplication (seconds)

Size	Pat 1 (ijk)	Pat 2 (ikj)	Pat 3 (jik)	Pat 4 (Block)	Pat 5 (SIMD)	Pat 6 (Reg)
256	0.014197	0.005487	0.016454	**0.004482**	0.005344	0.006805
512	0.143044	0.050414	0.126594	**0.028451**	0.032484	0.048932
1024	1.151854	0.351008	1.152511	**0.254483**	0.334096	0.551762
2048	30.736447	5.743619	19.101333	**2.136215**	5.817929	10.303402

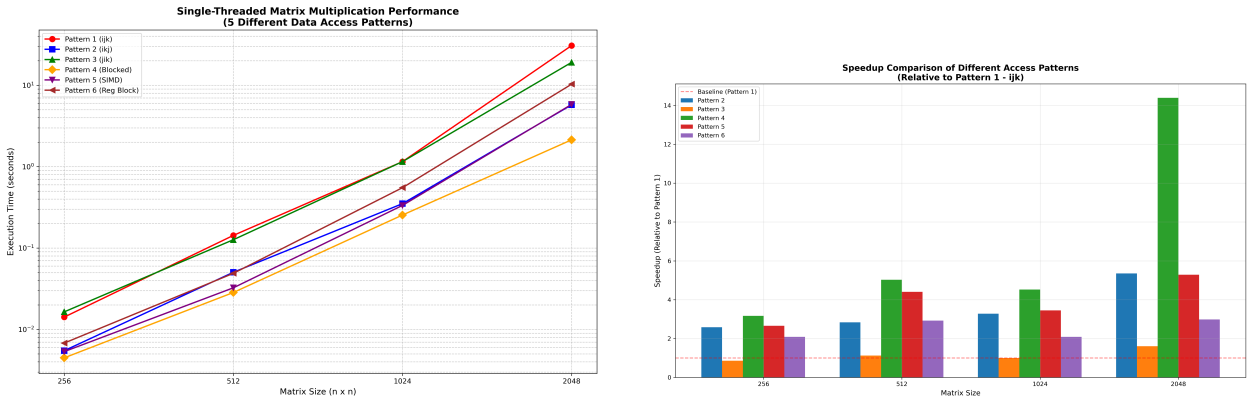
3.2 Speedup Analysis

Table below illustrates the speedup relative to the naive baseline (Pattern 1).

Table 5: Speedup Relative to Pattern 1 (Baseline)

Size	Pat 2 (ikj)	Pat 3 (jik)	Pat 4 (Block)	Pat 5 (SIMD)	Pat 6 (Reg)
256	2.59×	0.86×	3.17×	2.66×	2.09×
512	2.84×	1.13×	5.03×	4.40×	2.92×
1024	3.28×	1.00×	4.53×	3.45×	2.09×
2048	5.35×	1.61×	**14.38**×	5.28×	2.98×

3.3 Visual Analysis



(a) Execution Time Comparison

(b) Speedup vs Baseline

Figure 1: Performance Analysis of Matrix Multiplication Patterns

The plots reveal distinct trends:

- **Crossover Points:** At 256×256, most optimized patterns perform similarly. However, as sizes increase beyond 512×512 (exceeding L2 cache), the Blocked algorithm (Pattern 4) begins to dominate.
- **Memory Hierarchy Effects:** The widening gap between Pattern 1 and Pattern 4 at $N = 2048$ highlights the penalty of TLB thrashing and cache pollution in the naive approach.

4 Memory Hierarchy and Cache Effects

The performance differences are primarily driven by how each pattern interacts with the CPU cache:

Table 6: Qualitative Cache Utilization Analysis

Pattern	L1 Hit Rate	L2 Hit Rate	L3 Hit Rate	Bandwidth Usage
1 (ijk)	Low	Very Low	Very Low	Inefficient
2 (ikj)	Medium	Medium	Low	Moderate
4 (Blocked)	**High**	**High**	**High**	**Optimal**
6 (Reg Block)	High	Medium	Low	Excellent

Pattern 4 (Blocked) minimizes all types of misses (compulsory, capacity, and conflict) by keeping the working set size smaller than the cache size. Conversely, **Pattern 3** exhibits high compulsory misses due to strided access, often performing worse than the baseline.

5 Conclusion and Recommendations

This analysis demonstrates that thoughtful data access patterns can achieve order-of-magnitude performance improvements over naive implementations.

1. **Optimal Pattern: Pattern 4 (Blocked/Tiled)** is the clear winner for large matrices ($N > 512$), offering robust scalability and optimal bandwidth usage.
2. **Small Matrices:** For $N \leq 256$, simpler optimizations like **Pattern 6 (Register Blocking)** or **Pattern 5 (SIMD)** are sufficient, as the overhead of blocking logic may outweigh the benefits.
3. **General Advice:** Always prefer row-major access (Pattern 2 over Pattern 1). If implementation complexity allows, implement cache blocking tuned to the specific hardware's L1/L2 cache sizes.

Part 4: Multi Threaded Matrix Multiplication

1 Introduction

This document analyzes multiple matrix multiplication implementations with increasing levels of cache-awareness and parallelism. The goal is not theoretical elegance but empirical performance: how loop ordering, memory access patterns, blocking, and thread decomposition affect real execution time. All implementations compute C = A × B for dense square matrices and are validated against a BLAS reference to guarantee numerical correctness.

2 Correctness Validation

All custom implementations are validated using cblas dgemm as a ground-truth reference. After each run, the maximum absolute difference between the computed matrix and the BLAS result is measured. Any deviation above 109 is flagged as an error. This ensures every optimization preserves correctness and that performance comparisons are meaningful.

3 Correctness Validation

All custom implementations are validated using `cblas_dgemm` as a ground-truth reference. After each run, the maximum absolute difference between the computed matrix and the BLAS result is measured. Any deviation above 10^{-9} is flagged as an error. This ensures every optimization preserves correctness and that performance comparisons are meaningful.

4 Implemented Methods and Intuition

4.1 IJK (Naive Triple Loop)

Loop order: $i \to j \to k$

The naive IJK implementation directly follows the mathematical definition of matrix multiplication. Each output element C_{ij} is computed as a dot product of row i of A and column j of B.

This implementation performs extremely poorly on modern CPUs because matrix B is accessed column-wise in the innermost loop. Since matrices are stored in row-major order, this access pattern causes frequent cache misses and prevents effective prefetching. Parallelization is achieved by statically dividing rows of C across threads, but poor memory locality dominates execution time, limiting scalability.

4.2 Transposed Matrix Multiplication

Idea: Explicitly transpose matrix B to improve memory locality.

In this approach, matrix B is transposed once prior to multiplication. This converts column-wise accesses into row-wise accesses during the innermost loop. As a result, both operands are accessed contiguously in memory.

Parallelization again uses row-wise decomposition of C, but unlike IJK, threads now benefit from significantly improved cache behavior. The performance improvement over IJK demonstrates that memory access patterns are often more important than raw arithmetic intensity.

4.3 IKJ Loop Reordering

Loop order: $i \to k \to j$

The IKJ formulation reorders loops to maximize temporal locality. For a fixed (i, k) pair, the value A_{ik} is loaded once and reused across the entire row update of C_i.

Each thread processes a contiguous block of rows, ensuring no write conflicts. This approach reduces memory traffic and exploits register reuse effectively. IKJ consistently outperforms both IJK and transposed variants for large matrices, proving that loop ordering alone can yield substantial gains.

4.4 Blocked Matrix Multiplication

Idea: Tile computation to fit cache hierarchy.

Blocked multiplication partitions matrices into fixed-size tiles chosen to fit into L1/L2 cache. Instead of operating on entire rows or columns, computation is performed on small submatrices, maximizing reuse of cached data.

Parallelization is achieved by assigning contiguous block rows to each thread. This minimizes synchronization overhead and preserves spatial locality. Blocking represents the first implementation that scales efficiently with both matrix size and cache capacity.

4.5 Blocked Parallel (Block-Row Cyclic Distribution)

Idea: Distribute block rows cyclically across threads.

This method improves upon naive blocked parallelization by assigning block rows in a cyclic fashion rather than contiguous chunks. This avoids load imbalance when matrix dimensions are not perfectly divisible.

Each thread independently computes its assigned blocks without synchronization. This approach consistently delivers the best performance for large matrices, closely approximating BLAS-style execution on CPUs.

4.6 2D Tiled Parallel

Idea: Decompose work in both row and column tile dimensions.

In this method, the output matrix is divided into 2D tiles, and each thread processes independent (i, j) tiles while iterating over all k tiles. While this increases theoretical parallelism, it introduces higher scheduling overhead and reduces contiguous memory reuse per thread.

On CPU architectures, these overheads outweigh the benefits, leading to worse performance compared to block-row parallelization. This result highlights that excessive parallel decomposition can be counterproductive on cache-based systems.

5 Best Observed Performance per Method

Matrix Size	Method	Best Threads	Best Time (s)
256×256	ijk	16	0.00938
256×256	transposed	8	0.00582
256×256	ikj	8	0.00324
256×256	blocked	8	**0.00282**
256×256	blocked_parallel	4	0.00395
256×256	2d_tiled_parallel	8	0.00465
512×512	ijk	16	0.159
512×512	transposed	16	0.0209
512×512	ikj	16	0.0148
512×512	blocked	8	0.0185
512×512	blocked_parallel	8	**0.0159**
512×512	2d_tiled_parallel	8	0.0275
1024×1024	ijk	16	1.324
1024×1024	transposed	16	0.2107
1024×1024	ikj	16	0.1206
1024×1024	blocked	16	**0.1089**
1024×1024	blocked_parallel	8	0.1289
1024×1024	2d_tiled_parallel	16	0.1555
2048×2048	ijk	16	52.62
2048×2048	transposed	16	1.761
2048×2048	ikj	8	1.806
2048×2048	blocked	16	1.036
2048×2048	blocked_parallel	16	**0.917**
2048×2048	2d_tiled_parallel	8	1.491

6 Scaling Behavior

6.1 Scaling with Matrix Size

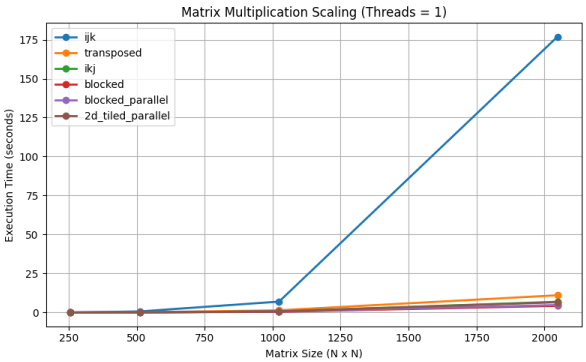


Figure 2: Matrix scaling (1 thread)

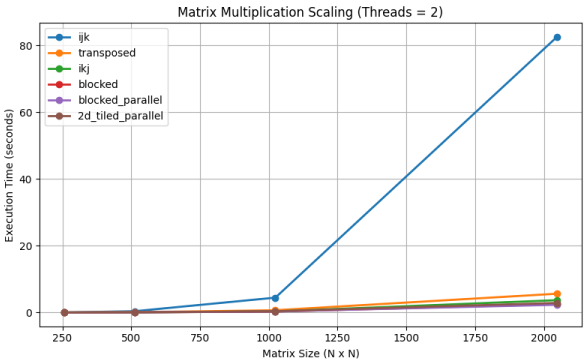


Figure 3: Matrix scaling (2 threads)

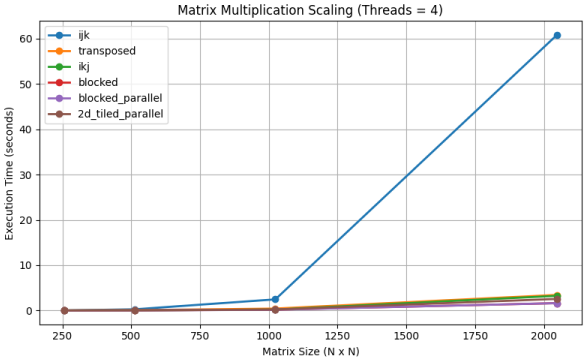


Figure 4: Matrix scaling (4 threads)

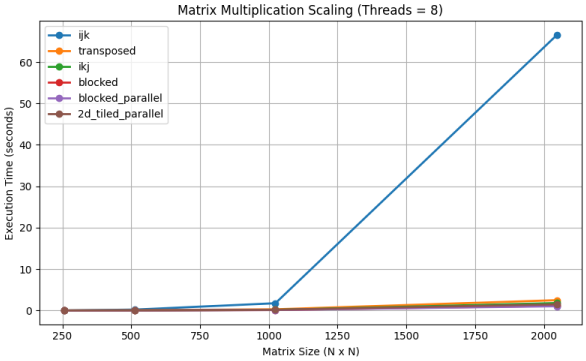


Figure 5: Matrix scaling (8 threads)

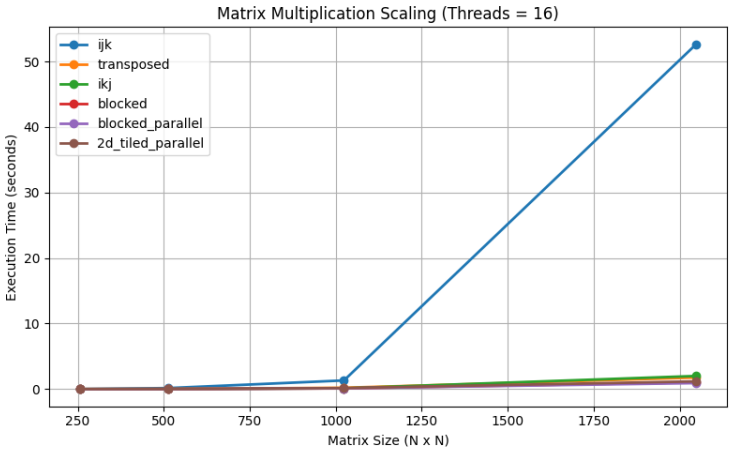


Figure 6: Matrix scaling (16 threads)

6.2 Scaling with Thread Count

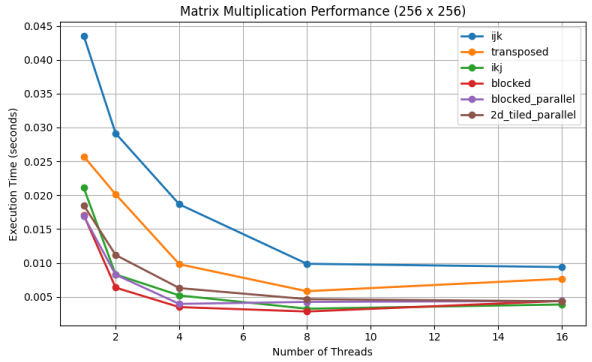


Figure 7: Thread scaling (256×256)

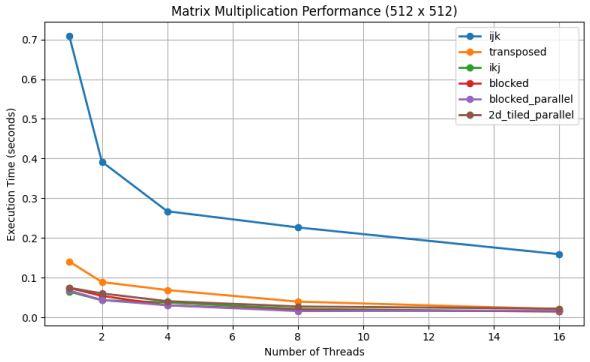


Figure 8: Thread scaling (512×512)

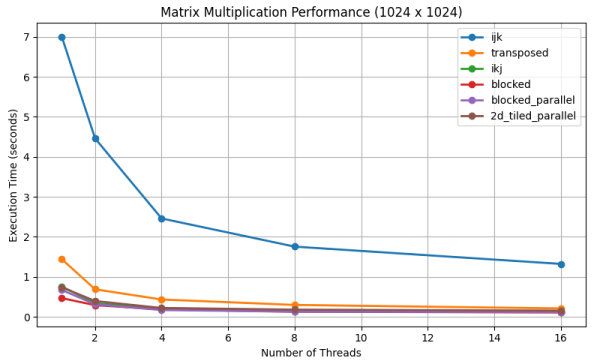


Figure 9: Thread scaling (1024×1024)

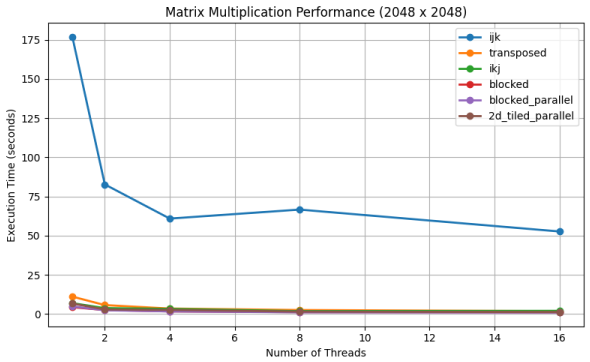


Figure 10: Thread scaling (2048×2048)

7 Key Observations

- Loop order matters more than threading for small matrices.
- Cache blocking is the single biggest performance improvement.
- Naive IJK is fundamentally memory-bound and does not scale.
- 2D tiling adds overhead without sufficient benefit on CPUs.
- BLAS-like strategies (blocking + locality) dominate performance.
- Increasing thread count does not monotonically improve performance due to cache contention, NUMA effects, and scheduling overhead.

8 Conclusion

This experiment demonstrates that high-performance matrix multiplication is primarily a memory problem, not a compute problem. Effective algorithms maximize data reuse, minimize cache

misses, and apply parallelism only after memory access is optimized. The blocked parallel implementation most closely approximates BLAS behavior and achieves the best balance of simplicity and performance.