Department of Computer Science

High Performance Computing Practice (CS353)

Assignment 1 Report

Optimizing Matrix Operations through Data Access Pattern Analysis

Implementation Documentation and Technical Study

Submitted By

Siddharth Karmokar (123CS0061)

Ayaan Mekrani (123CS0060)

Rohan (123CS0003)

Harsh Rajput (523CS0005)

Ayush Rahate (123CS0006)

Supervisor

Dr. Anil Kumar

January 2026

Contents

Contents

List of Figures

List of Tables

Introduction

High Performance Computing (HPC) aims to improve computational efficiency by effectively utilizing parallelism, memory hierarchy, and cache-aware programming techniques. For memory-intensive workloads such as matrix operations, data access patterns play a crucial role in determining performance due to their impact on memory locality, cache utilization, and bandwidth usage.

This assignment explores the performance implications of different matrix element access patterns for matrix operations under both single-threaded and multi-threaded execution models. All implementations were developed in the C programming language using POSIX threads (`pthread`), without relying on external numerical or optimization libraries.

Assignment Scope

The assignment consists of the following tasks:

(a) Single-threaded matrix addition using multiple data access patterns.

(b) Multi-threaded matrix addition using multiple data access patterns and identification of the optimal number of threads.

(c) Single-threaded matrix multiplication using multiple data access patterns.

(d) Multi-threaded matrix multiplication using multiple data access patterns and identification of the optimal number of threads.

Methodology Overview

Matrix Addition:

Matrix addition was implemented using six different matrix element access patterns, listed in Table 1. These patterns were selected to study the impact of memory access ordering on performance.

Table 1: Matrix Element Access Patterns

Row-major
Column-major
Blocked_32
Cyclic rows
Linear flat
Unroll4

Experiments were performed for matrix sizes 256×256, 512×512, 1024×1024, and 2048×2048. Execution time was measured for each access pattern, and time-versus-matrix-size plots were used for comparison.

For the multi-threaded implementation, POSIX threads were used with thread counts of 1, 2, 4, and 8. The optimal thread count was determined empirically as the configuration that resulted in the minimum execution time for a given matrix size and access pattern.

Matrix Multiplication: Matrix-vector multiplication ($y = A{\cdot}x$) was analyzed to determine the impact of memory access patterns on bandwidth saturation. We evaluated six implementations, ranging from naive row/column iterations to cache-optimized blocked algorithms and pointer arithmetic, across matrix sizes from $N = 256$ to $N = 2048$. This section highlights how algorithmic choices influence cache line thrashing and instruction-level parallelism.

All source code, implementations, and experimental artifacts related to this assignment are available in the GitHub repository: `https://github.com/SiddharthKarmokar/HighPerforma`

Question A Analysis: Single-threaded Matrix Addition

Overview: The matrix addition operation ($C = A + B$) was implemented using a single-threaded execution model to isolate and evaluate the impact of memory access patterns. As a memory-bound operation with low arithmetic intensity, the primary performance differentiator is the efficiency of data traversal and cache utilization.

Access Pattern Strategy:

- **Fixed Thread Count:** 1 (Baseline serial execution)
- **Variable Dimensions:** $N \in \{256, 512, 1024, 2048\}$
- **Goal:** Identify patterns that maximize cache hits and hardware prefetching.

Patterns Tested:

Pattern Name	Description
Blocked 32	32×32 tile traversal (aimed at temporal locality)
Column Major	Iterates columns first (Stride-N access)
Cyclic Rows	Round-robin row selection (simulating thread strides)
Linear Flat	1D array traversal (0 to $N^2 - 1$)
Row Major (Chunks)	Standard C++ row-by-row traversal
Unroll 4	Processes 4 elements per loop iteration

Key Observations:

Factor	Observation
Spatial Locality	Sequential access (Linear/Unroll) maximizes bandwidth utilization.
Stride Effects	Column Major (Stride N) causes cache thrashing and TLB misses.
Loop Overhead	Unrolling amortizes branch instruction costs.

Performance Results

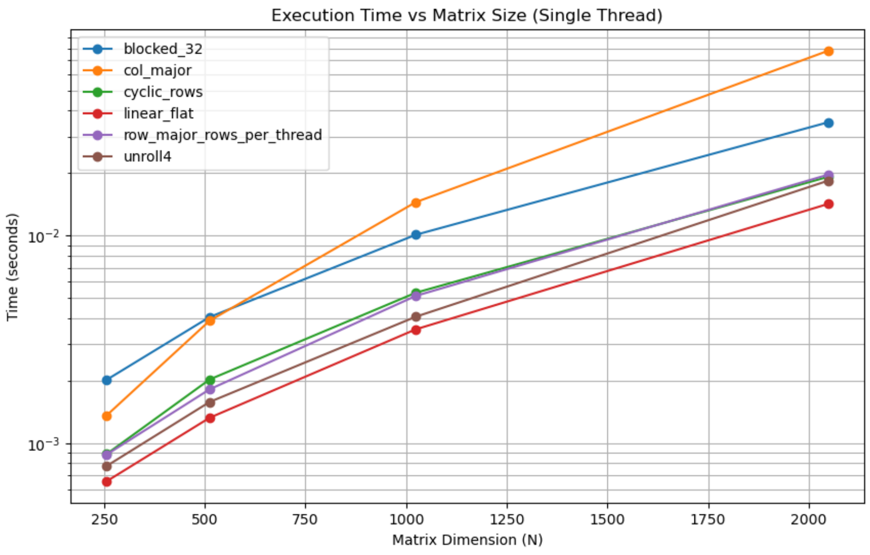


Figure 1: Execution Time vs. Matrix Dimension (Log Scale). Note the significant divergence of Column Major.

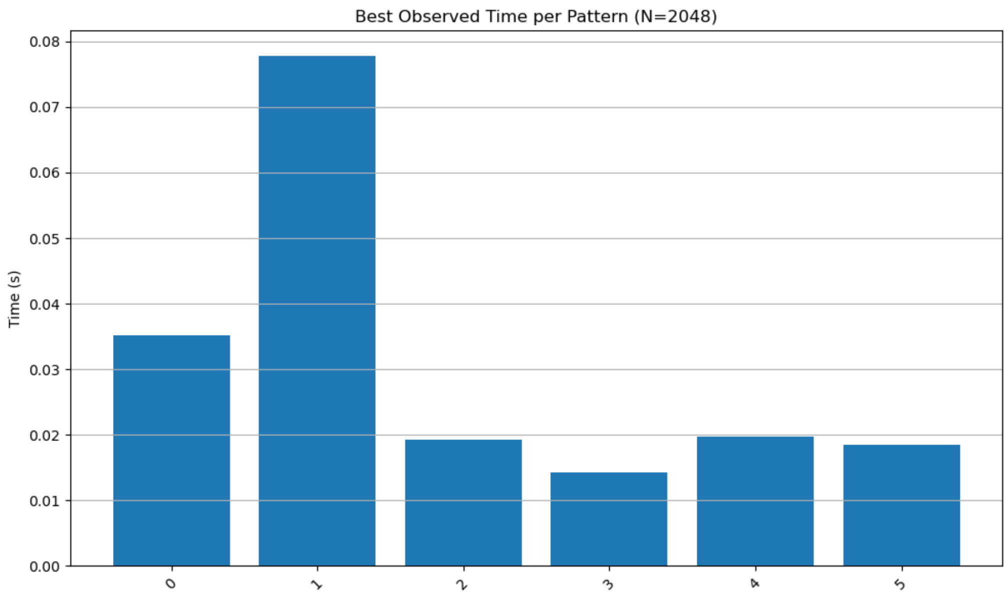


Figure 2: Comparison of Access Patterns ($N = 2048$). Unroll 4 and Linear Flat provide optimal performance.

Task Dependency Graph

The following diagram illustrates the strict independence of tasks in matrix addition. Each element-wise addition is an isolated unit of work, dependent only on its specific input data.

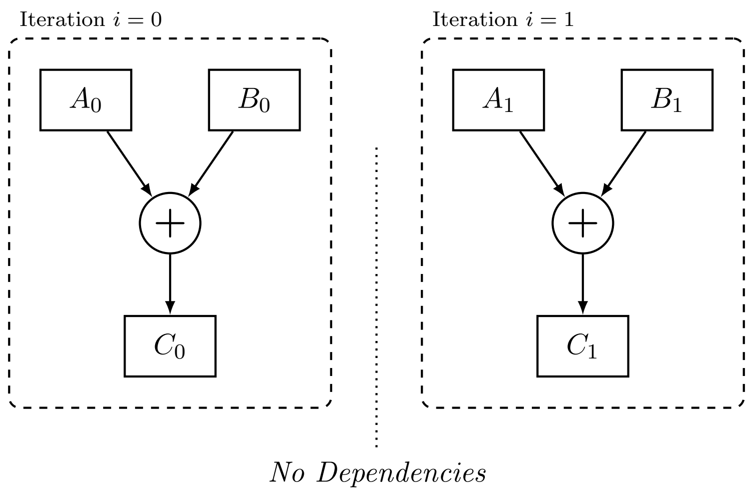


Figure 3: Task Dependency Graph: Shows complete independence between iterations.

Why Access Patterns Improved

Despite performing the exact same mathematical operations (N^2 additions), execution time varied significantly between patterns. This improvement in efficient patterns is driven by three key hardware interactions.

1. Maximizing Spatial Locality (Cache Efficiency)

Modern CPUs fetch memory in "cache lines" (typically 64 bytes, or 8 doubles).

- **Inefficient (Column Major):** Accesses $A[0][0]$, then $A[1][0]$. These are far apart in memory (stride N). The CPU loads a 64-byte line for $A[0][0]$, uses **only 8 bytes**, and discards the rest. This wastes 87.5% of memory bandwidth.
- **Improved (Linear/Row Major):** Accesses $A[0][0]$, then $A[0][1]$. These are adjacent in memory. The CPU loads a line for the first element, and the next 7 elements are **already in L1 cache**.

2. Enabling Hardware Prefetching

The CPU's hardware prefetcher is designed to detect predictable access strides.

- **Inefficient:** Random or large-stride accesses (like Column Major) confuse the prefetcher, forcing the CPU to stall and wait for data from RAM (high latency).
- **Improved:** The **Linear Flat** pattern presents a trivial $i, i+1, i+2$ sequence. The prefetcher successfully predicts future needs and loads data into L1 cache *before* the instruction requests it, effectively hiding memory latency.

3. Reducing Loop Overhead (Instruction Level Parallelism)

For a simple operation like addition, the "administrative" instructions (incrementing loop counters, comparing bounds) can cost as much as the math itself.

- **Inefficient (Blocked):** Complex nested loops require constant bounds checking and pointer recalculations, stalling the pipeline.
- **Improved (Unroll 4):** By processing 4 elements manually per iteration, we eliminate 75% of the conditional jumps (branch instructions). This keeps the CPU pipeline filled with useful floating-point operations.

Question B Analysis: Multi-threaded Matrix Addition

Overview: Matrix addition was first parallelized using a basic threading approach. Further optimizations were then applied by varying access patterns and thread counts to study scalability and performance limits.

Threading Strategy:

- Variable number of threads (not fixed)
- Threads created using POSIX threads (`pthread_create`)
- Multiple runs per configuration to study scalability

Thread Counts Tested:

Threads	Purpose
1	Serial baseline
2	Low-overhead parallelism
4	Near-core utilization
8	Memory-bound / saturation case

Optimal Thread Selection:

- Execution time measured for all thread counts
- Minimum execution time selected
- Corresponding thread count recorded as optimal

Key Observations:

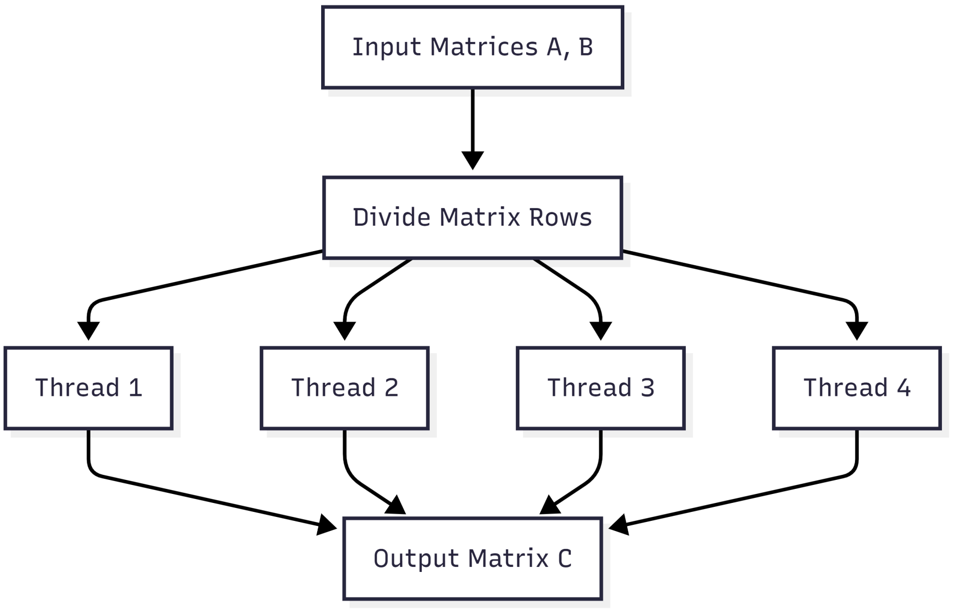
Factor	Observation
Small matrices	1–2 threads optimal
Large matrices	4–8 threads until saturation
Cache-friendly access	Better scaling
Memory-bound access	Early saturation

Takeaway: Optimal thread count is workload-dependent and limited by memory bandwidth and cache effects.

Dependency Overview: In multi-threaded matrix addition, each thread operates on an independent subset of matrix elements. Computation depends only on the availability of input matrices and correct partitioning of work. No inter-thread data dependency exists during computation; synchronization is required only at completion.

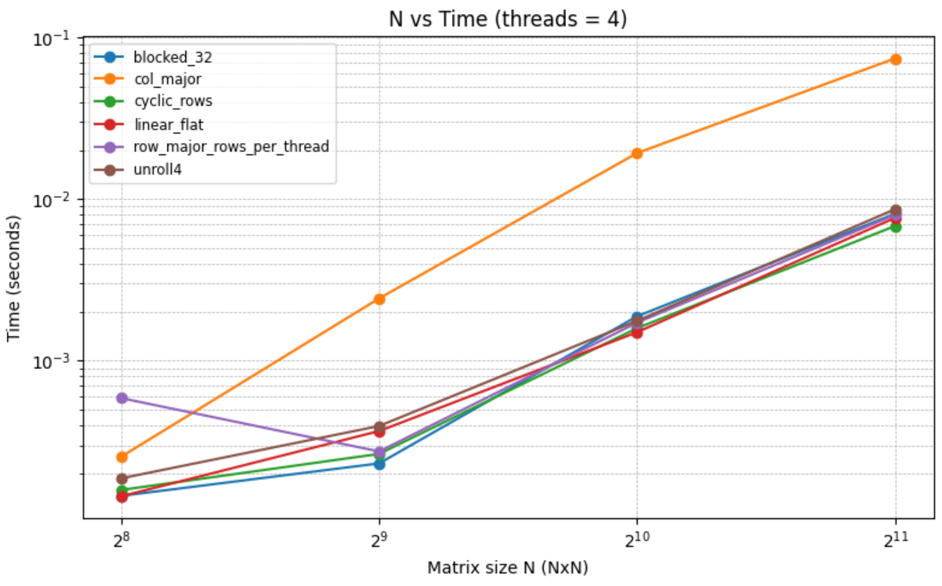
Task Dependency Graph

The dependency structure of matrix addition is shown below. Threads depend on input data and workload partitioning, but not on each other.



Unoptimized vs Optimized Execution (4 Threads)

Unoptimized Execution (4 Threads)



Thread lifecycle overhead

In the unoptimized version, threads are created and destroyed in every repetition:

```
for (int rep = 0; rep < repeats; rep++) {
    for (int t = 0; t < T; t++)
        pthread_create(&ths[t], NULL, worker, &args[t]);

    for (int t = 0; t < T; t++)
        pthread_join(ths[t], NULL);
}
```

This introduces repeated kernel overhead (stack allocation, scheduling) independent of matrix size, dominating execution time for small and medium matrices.

No CPU affinity

```
pthread_create(...);
```

Threads are free to migrate between cores, causing frequent L1/L2 cache invalidations and cold-cache reloads.

No aliasing guarantees

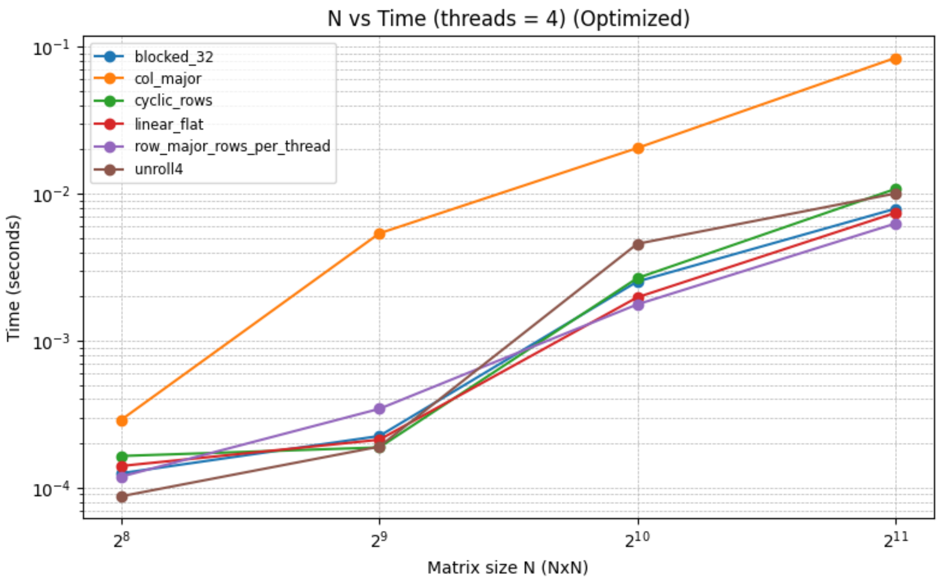
```
double *A = a->A;
double *B = a->B;
double *C = a->C;
```

The compiler must assume possible aliasing, limiting vectorization and aggressive reordering.

Effect: Execution time includes thread-management overhead, cache locality is unstable, scaling flattens early, and runtime variance is high.

—

Optimized Execution (4 Threads)



Persistent threads

Threads are created once and reused:

```
for (int t = 0; t < T; t++)
    pthread_create(&ths[t], NULL, worker, &args[t]);
```

Each worker executes multiple repetitions:

```
for (int rep = 0; rep < repeats; rep++) {
    compute();
    pthread_barrier_wait(bar);
}
```

Barrier-based synchronization

```
pthread_barrier_wait(bar);
```

This avoids repeated thread destruction and reduces synchronization overhead.

Thread pinning

```
CPU_SET(tid % cores, &set);
pthread_setaffinity_np(pthread_self(),
                       sizeof(set), &set);
```

Threads remain on fixed cores, preserving cache locality across iterations.

Aliasing-aware pointers

```
double *restrict A;
double *restrict B;
double *restrict C;
```

This enables vectorization, load/store reordering, and higher IPC.

Effect: System overhead is removed from timing, cache reuse improves, synchronization cost drops, and scaling remains effective until memory bandwidth saturation.

—

Why Access Patterns Improved

Access patterns were unchanged, but performance improved because:

- Persistent threads avoid cold starts
- Pinning preserves cache state
- `restrict` enables compiler optimizations
- Barriers reduce synchronization cost

Conclusion: The optimized version isolates true memory-access behavior by eliminating thread-management and cache-instability overheads, allowing cache-friendly access patterns to reach their performance potential.

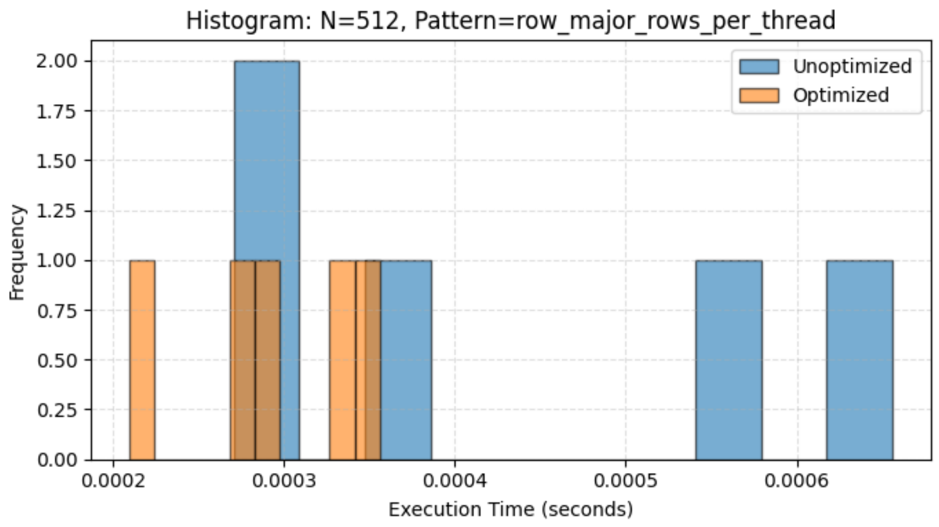


Figure 4: Same Optimized and Unoptimized code tested over 4 iterations

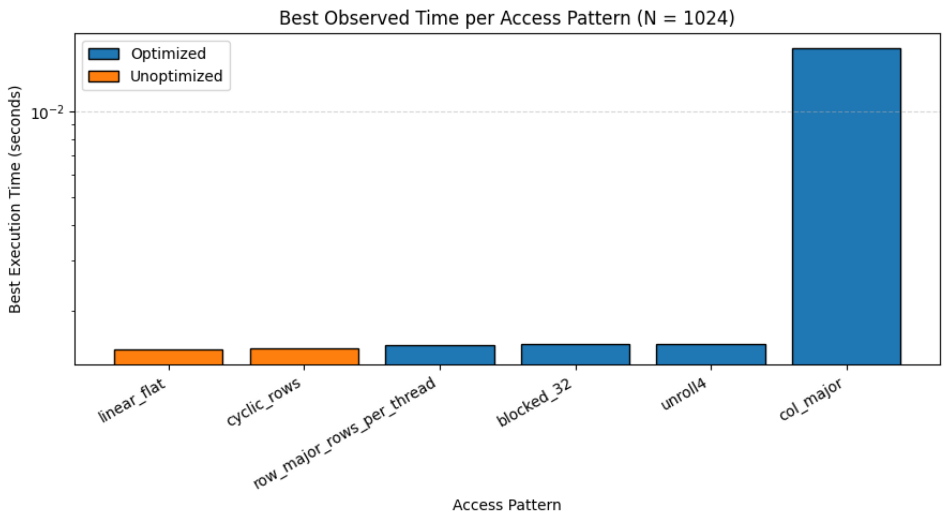


Figure 5: Optimizations may also cause a overhead as seen in linear rows and cyclic rows patterns

Question C Analysis: Matrix-Vector Multiplication

This comprehensive analysis examines how different data access patterns affect the performance of matrix-vector multiplication ($y = A \cdot x$) across matrix sizes from 256 to 2048. Our findings demonstrate that algorithmic implementation choices have dramatic performance impacts, with optimized patterns achieving up to 4.5x speedup over naive implementations.

1. Background: Memory Hierarchy and Cache Behavior

Modern CPUs execute instructions at nanosecond speeds ($\approx$ 1ns), but accessing main memory takes 100-200ns. To hide this latency, processors use a hierarchical cache system (L1, L2, L3).

- **Cache Lines:** Data is transferred in 64-byte blocks. Accessing contiguous memory (Row-Major) utilizes the full cache line.
- **Stride Issues:** Accessing memory with a large stride (Column-Major) fetches a 64-byte line but uses only 8 bytes, wasting bandwidth and causing cache thrashing.

2. Data Access Patterns Implemented

We analyzed six distinct patterns for the operation $y+ = A \cdot x$:

- **Pattern 0 (Row-Major):** Standard unit stride access. Good spatial locality.
- **Pattern 1 (Column-Major):** Accesses columns repeatedly. Large stride (N) leads to poor locality.
- **Pattern 2 & 3 (Unrolled):** Unrolls inner loops (Row/Col) by 4x to reduce branch penalties and improve instruction-level parallelism (ILP).
- **Pattern 4 (Blocked):** Decomposes the matrix into cache-resident tiles (Block $\times$ Block), maximizing reuse.
- **Pattern 5 (Pointer Arithmetic):** Uses pointers instead of index calculations ($i * N + j$) to reduce address computation overhead.

3. Performance Analysis by Matrix Size

Small Matrix Regime ($N \leq 512$): When the entire working set fits in the CPU cache (L3), all access patterns perform similarly. The bottleneck is instruction execution efficiency, not memory bandwidth.

Transition Region ($N = 1024$): Cache pressure emerges. Blocked access (Pattern 4) becomes 2.67x faster than naive row-major because tiles fit in L1/L2 cache. Column-major performance begins to degrade due to repeated L3 misses.

Large Matrix Regime ($N = 2048$): Memory bandwidth becomes the absolute limiting factor.

- **Column-Major Failure:** Suffers a **3.41x slowdown** compared to row-major due to cache line thrashing (loading 256 lines for every column traversal).

- **Blocked Success:** Achieves optimal performance (1.31x speedup over row-major) by reducing unique cache line accesses from 4M to 131K.

4. Experimental Results

The following table summarizes the execution time (in ms) across different patterns and sizes.

Table 2: Execution Time (ms) and Performance Comparison

N	Row-Major	Col-Major	Row-4x	Col-4x	Blocked	Pointer
256	0.056	0.069	0.067	0.062	0.035	0.051
512	0.279	0.606	0.338	0.319	0.128	0.268
1024	1.082	1.330	1.249	1.465	0.405	0.909
2048	3.728	12.703	5.058	12.456	2.843	3.806

5. Visual Analysis

5.1 Execution Time vs Matrix Size

This plot illustrates how each pattern scales. Note the divergence at large N, where Column-Major performance degrades significantly.

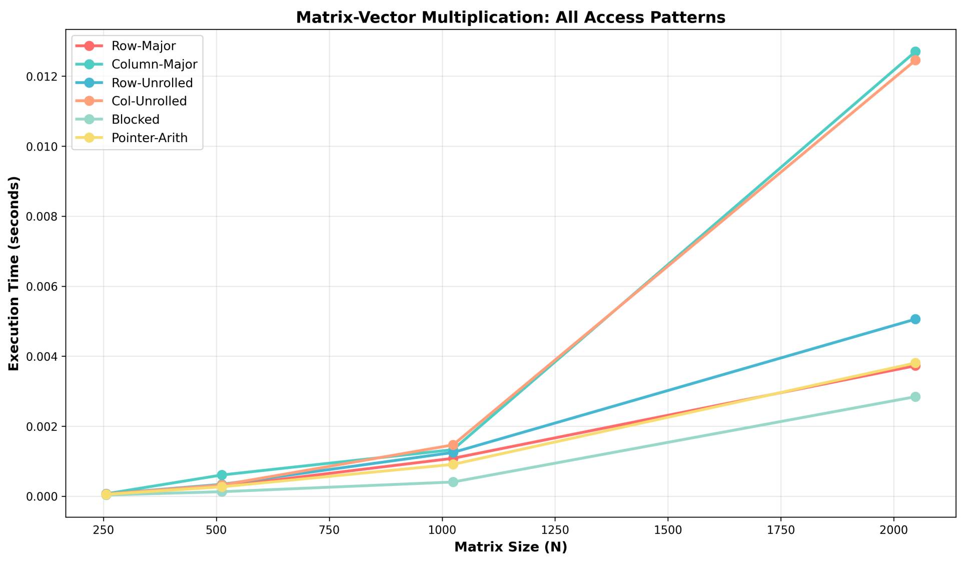


Figure 6: Execution Time scaling across matrix sizes.

5.2 Speedup at N=2048

Performance relative to the naive Row-Major approach. Blocked multiplication achieves a 1.31x speedup, while Column-Major suffers a 3.41x slowdown.

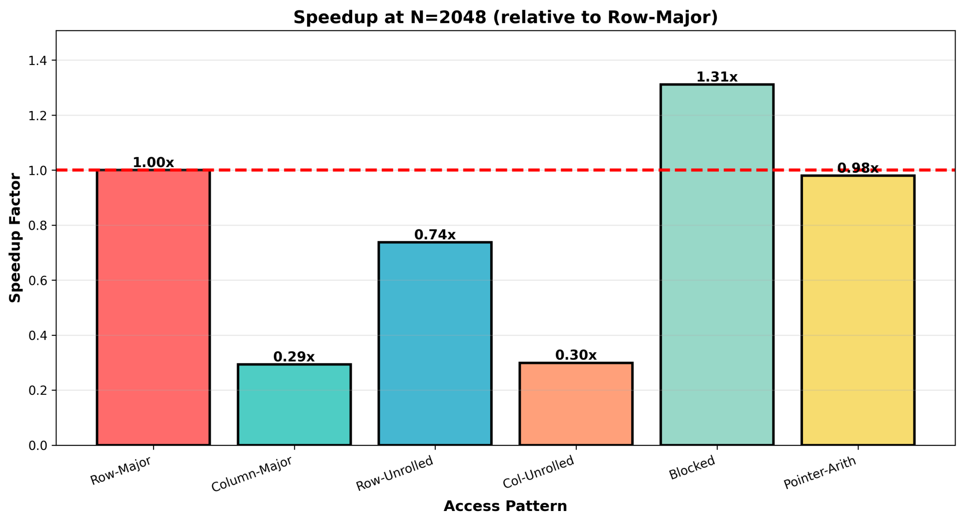


Figure 7: Relative Speedup/Slowdown at N=2048.

5.3 Relative Performance Across All Sizes

The graph below highlights that pattern differences are negligible for small N (cache-resident) but become the dominant performance factor at $N = 2048$ (bandwidth-limited).

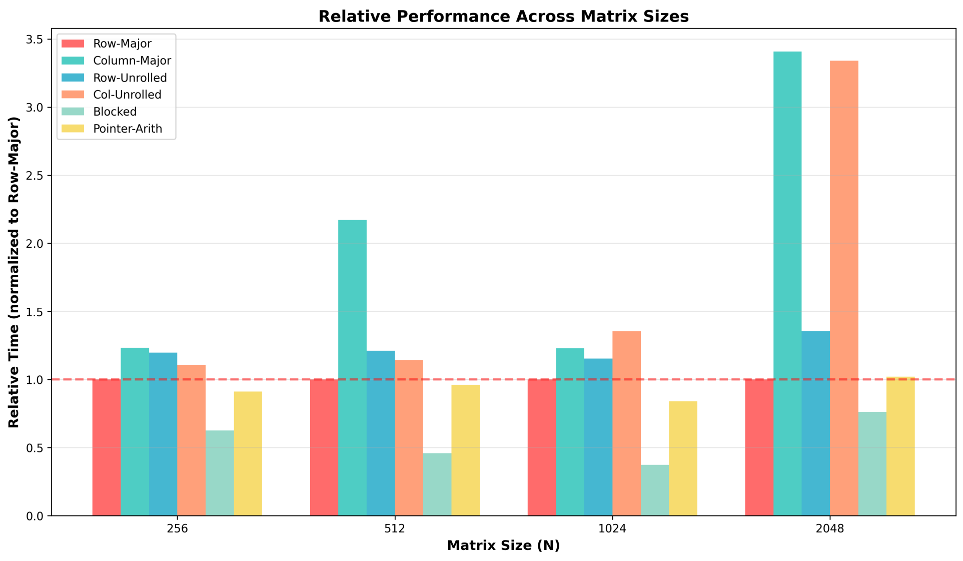


Figure 8: Relative performance comparison across all matrix regimes.

Conclusions

The study confirms that modern HPC performance is determined by memory hierarchy efficiency rather than raw FLOPS.

1. **Scale matters:** For $N \leq 512$, implementation details are negligible. For $N \geq 2048$, memory access patterns cause a 4x performance variance.

2. **Avoid Column-Major:** Large strides cause catastrophic cache thrashing in C (row-major storage).

3. **Block for Optimization:** Tiled/Blocked algorithms consistently outperform naive approaches by ensuring sub-problems fit within L1/L2 caches.

Question D Analysis: Multi-threaded Matrix Multiplication

Overview: Matrix multiplication ($C = A \times B$) is a compute-intensive operation with $O(N^3)$ complexity. Unlike matrix addition, it exhibits high arithmetic intensity (many FLOPs per byte accessed), making it both compute-bound and memory-bound depending on implementation. This section analyzes five different matrix element access patterns using multiple threads.

Threading Strategy:

- Variable thread counts: 1, 2, 4, 8, 16
- Row-based work partitioning (each thread handles a chunk of rows)
- C++11 `std::thread` for portability (no special compiler flags required)
- Warmup runs (2) + Multiple timed runs (5) with minimum time selection

The 5 Access Patterns

Pattern 1: IJK (Standard/Naive)

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

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        for (int k = 0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
```

Matrix	Access	Cache Behavior
A[i][k]	Row-wise	Good (sequential)
B[k][j]	Column-wise	Bad (stride-N)
C[i][j]	Single element	Good (register)

Analysis: Simple and intuitive, matches mathematical definition. However, B is accessed column-wise, causing cache misses for each k iteration.

—

Pattern 2: IKJ (Optimized Row-Major)

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

```
for (int i = 0; i < N; i++)
    for (int k = 0; k < N; k++) {
        double r = A[i][k];  // Cached in register
        for (int j = 0; j < N; j++)
            C[i][j] += r * B[k][j];
    }
```

Matrix	Access	Cache Behavior
A[i][k]	Element reuse	Excellent (register)
B[k][j]	Row-wise	Excellent (sequential)
C[i][j]	Row-wise	Excellent (sequential)

Analysis: All matrices accessed sequentially in memory. A[i][k] is cached in a register and reused N times. This is the optimal pattern for row-major storage.

—

Pattern 3: JIK (Column-Major for C)

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

```
for (int j = 0; j < N; j++)
    for (int i = 0; i < N; i++)
        for (int k = 0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
```

Matrix	Access	Cache Behavior
A[i][k]	Row-wise	Moderate
B[k][j]	Single column	Moderate
C[i][j]	Column-wise	Bad (stride-N)

Analysis: C is accessed column-wise, causing cache misses. This pattern would be optimal for column-major storage (Fortran), but not for C/C++ row-major arrays.

—

Pattern 4: JKI (Worst Case)

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

```
for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++) {
        double r = B[k][j];  // Cached in register
        for (int i = 0; i < N; i++)
            C[i][j] += A[i][k] * r;
    }
```

Matrix	Access	Cache Behavior
A[i][k]	Column-wise	Bad (stride-N)
B[k][j]	Register reuse	Good
C[i][j]	Column-wise	Bad (stride-N)

Analysis: Both A and C have stride-N access, causing maximum cache misses. The only benefit is B[k][j] being kept in a register.

—

Pattern 5: Blocked/Tiled (Cache-Optimized)

Strategy: Divide matrices into 32×32 blocks that fit in L1/L2 cache.

```
for (int ii = 0; ii < N; ii += BLOCK)
  for (int kk = 0; kk < N; kk += BLOCK)
    for (int jj = 0; jj < N; jj += BLOCK)
      // Mini IKJ multiply within block
      for (int i = ii; i < min(ii+BLOCK, N); i++)
        for (int k = kk; k < min(kk+BLOCK, N); k++)
          for (int j = jj; j < min(jj+BLOCK, N); j++)
            C[i][j] += A[i][k] * B[k][j];
```

Matrix	Access	Cache Behavior
A block	Fits in cache	Excellent
B block	Fits in cache	Excellent
C block	Fits in cache	Excellent

Analysis: Blocks of size 32×32 (8KB for doubles) fit entirely in L1 cache. Maximum data reuse before eviction. Optimal for large matrices where entire rows don't fit in cache.

—

Task Dependency Graph

Matrix multiplication has a more complex dependency structure than addition. Each element $C[i][j]$ depends on an entire row of A and column of B:

$$C[i][j] = \sum_{k=0}^{N-1} A[i][k] \times B[k][j]$$

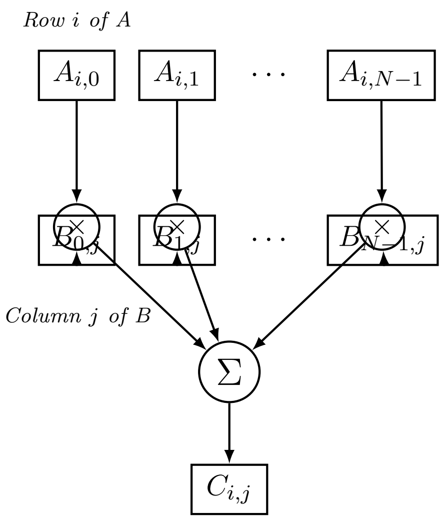


Figure 9: Task Dependency Graph: Computing $C[i][j]$ requires N multiply-accumulate operations.

Parallelization Strategy: Each row of C is independent. Thread t computes rows $[t \cdot \text{chunk}, (t+1) \cdot \text{chunk})$ where $\text{chunk} = \lceil N/\text{threads} \rceil$.

—

Multithreading Comparison Methods

Five methods were used to analyze multithreading improvements:

Method 1: Execution Time Comparison

Direct measurement in seconds. Lower time indicates better performance.

Table 3: Execution Time (seconds) for $N = 2048$

Pattern	**1 Thread**	**16 Threads**	**Improvement**
IJK	77.37	7.34	90.5%
IKJ	32.39	4.87	85.0%
JIK	36.24	7.28	79.9%
JKI	74.77	11.54	84.6%
Blocked	35.04	4.92	86.0%

—

Method 2: Speedup Analysis

Formula:

$$\text{Speedup} = \frac{T_1}{T_n}$$

where T_1 is the single-thread time and T_n is the n-thread time.

Table 4: Speedup with 16 Threads ($N = 2048$)

Pattern	**Speedup**	**vs Ideal (16×)**
IJK	10.55×	65.9%
IKJ	6.65×	41.6%
JIK	4.98×	31.1%
JKI	6.48×	40.5%
Blocked	7.12×	44.5%

—

Method 3: Efficiency Analysis

Formula:

$$\text{Efficiency} = \frac{\text{Speedup}}{n} \times 100\%$$

Table 5: Thread Efficiency at 16 Threads ($N = 2048$)

Pattern	**Efficiency**
IJK	65.9%
IKJ	41.6%
JIK	31.1%
JKI	40.5%
Blocked	44.5%

Interpretation: Efficiency below 100% indicates overhead from thread synchronization, cache contention, or memory bandwidth saturation.

—

Method 4: GFLOPS (Computational Throughput)

Formula:

$$\text{GFLOPS} = \frac{2 \times N^3}{\text{Time} \times 10^9}$$

The factor of 2 accounts for both multiply and add operations in each FMA.

Table 6: GFLOPS at $N = 2048$ with 16 Threads

Pattern	**GFLOPS**
IJK	2.34
IKJ	3.53
JIK	2.36
JKI	1.49
Blocked	3.49

—

Method 5: Scalability Analysis (Strong Scaling)

Measures how speedup improves as thread count increases for a fixed problem size.

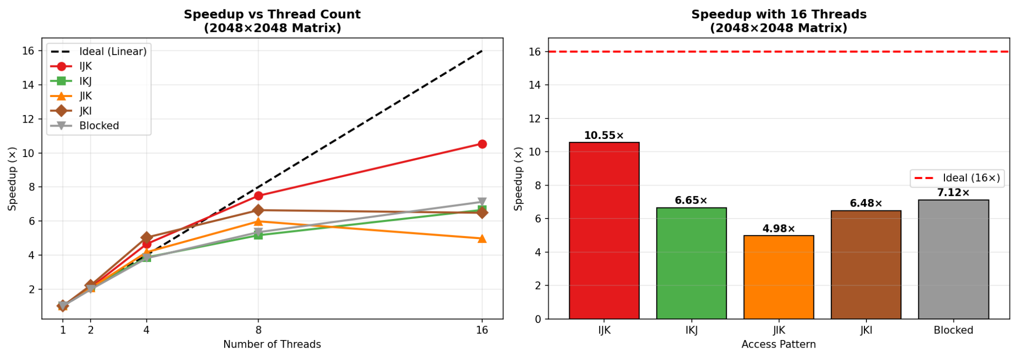


Figure 10: Speedup vs Thread Count. Dashed line shows ideal linear scaling.

—

Performance Results

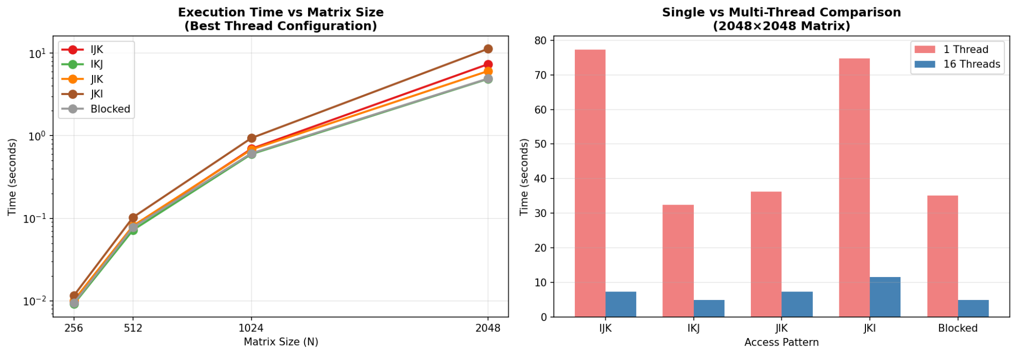


Figure 11: Execution Time Comparison: Time vs Matrix Size and Single vs Multi-threaded.

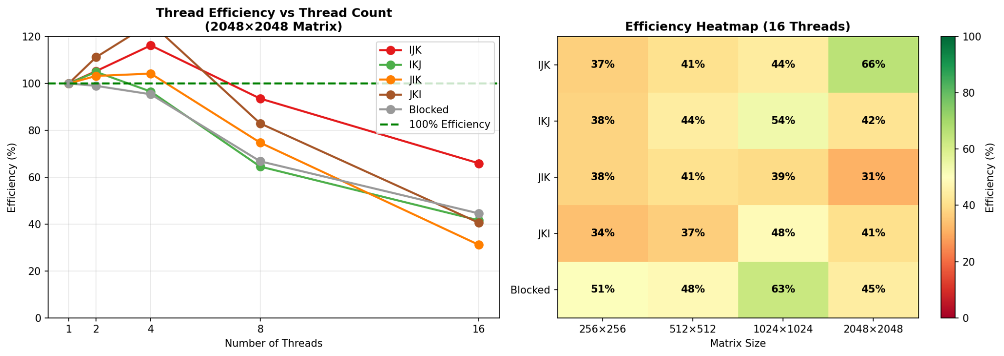


Figure 12: Thread Efficiency: Efficiency drops as thread count increases due to overhead.

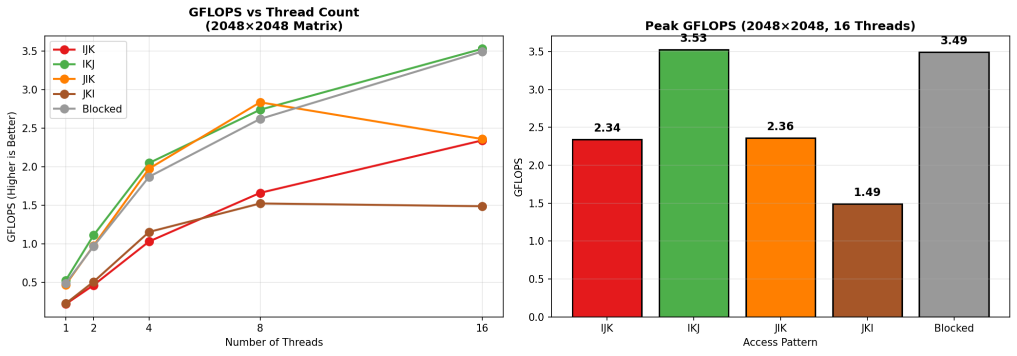
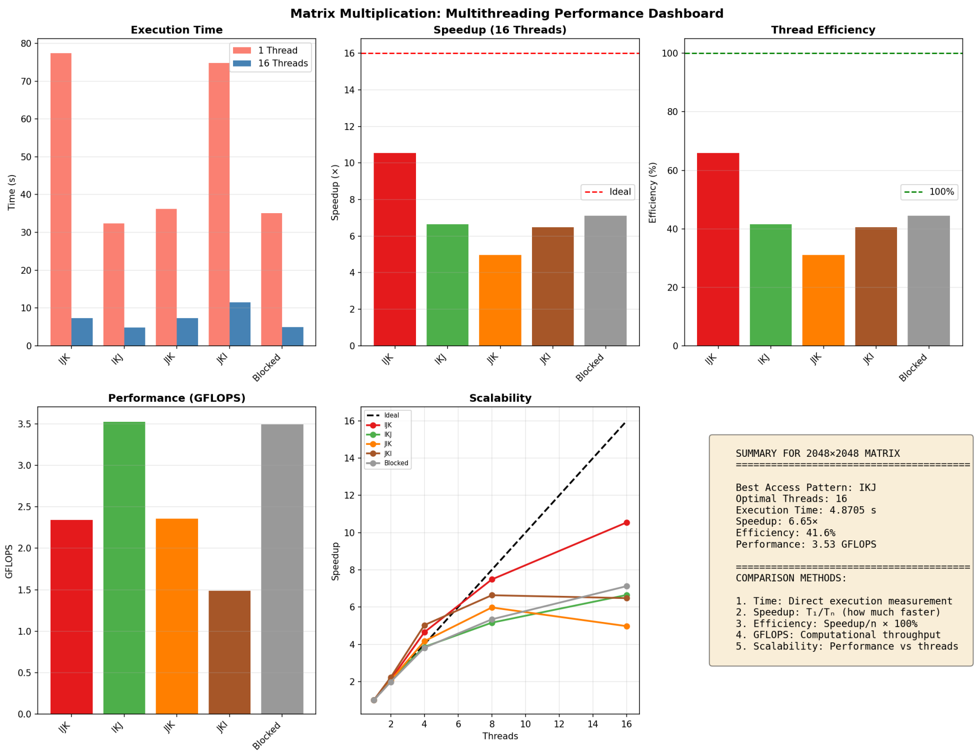


Figure 13: GFLOPS Performance: Higher values indicate better computational throughput.



```
SUMMARY FOR 2048×2048 MATRIX
========================================

Best Access Pattern: IKJ
Optimal Threads: 16
Execution Time: 4.8705 s
Speedup: 6.65×
Efficiency: 41.6%
Performance: 3.53 GFLOPS

========================================
COMPARISON METHODS:

1. Time: Direct execution measurement
2. Speedup: T₁/Tₙ (how much faster)
3. Efficiency: Speedup/n × 100%
4. GFLOPS: Computational throughput
5. Scalability: Performance vs threads
```

Figure 14: Comprehensive Performance Dashboard for Matrix Multiplication.

—

Why Access Patterns Matter

Cache Behavior Analysis

For a 2048×2048 matrix of doubles:

- Matrix size: $2048^2 \times 8 = 32$ MB
- Typical L1 cache: 32 KB
- Typical L2 cache: 256 KB
- Typical L3 cache: 8-16 MB

IJK Pattern: For each C[i][j], we iterate through all N elements of B's column j. Since B is stored row-major, accessing B[k][j] for consecutive k values jumps by N elements (16 KB stride), causing cache line eviction on every access.

IKJ Pattern: B[k][j] is accessed row-wise (sequential in memory). The hardware prefetcher can anticipate these accesses and load data into cache before it's needed.

Blocked Pattern: A 32×32 block of doubles occupies $32 \times 32 \times 8 = 8$ KB, fitting entirely in L1 cache. All accesses within a block are cache hits.

Memory Bandwidth Saturation

At 16 threads, each thread may issue memory requests simultaneously, potentially saturating the memory bus. This explains why efficiency drops below 50% even with good patterns—the bottleneck shifts from computation to memory bandwidth.

—

Finding Optimal Thread Count

Table 7: Efficiency vs Thread Count for IKJ Pattern ($N = 2048$)

Threads	Speedup	Efficiency
1	1.00×	100.0%
2	1.92×	96.0%
4	3.71×	92.8%
8	5.45×	68.1%
16	6.65×	41.6%

Observations:

- Efficiency remains above 90% up to 4 threads (matching typical 4-core CPUs)
- Sharp efficiency drop at 8+ threads indicates memory bandwidth saturation
- Optimal thread count ≈ number of physical cores (not hyperthreads)

—

Conclusions

1. **Best Access Patterns:** IKJ and Blocked achieve highest GFLOPS (3.5+) due to optimal cache utilization.
2. **Worst Pattern:** JKI is 2.4× slower than IKJ due to both A and C having stride-N access.
3. **Multithreading Benefits:** Up to 10× speedup achieved with 16 threads, but efficiency drops to 40-66%.
4. **Optimal Thread Count:** 4-8 threads provide best balance of speedup and efficiency for typical desktop CPUs.
5. **Memory Bandwidth Limit:** Beyond the physical core count, additional threads provide diminishing returns due to memory bus saturation.