U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Performance Evaluation of Matrix Multiplication Algorithms: Single-Core and Multi-Core Implementations

Bruno Huang[1], Nelson Neto[2], and Ricardo Yang[3]

[1]up202207517
[2]up202108117
[3]up202208465

March 2025

Abstract

This report, conducted as part of the CPD (Parallel and Distributed Computing) course, evaluates the performance of matrix multiplication algorithms in single-core and multi-core environments, focusing on the impact of memory hierarchy and parallelization. Three algorithms were implemented: a basic row-column approach, an element-wise approach, and a block-oriented approach. The implementations were developed in C++ and C#, with performance measured using processing time, speedup, efficiency, and cache misses (L1 and L2) via the Performance API (PAPI). OpenMP was utilized to parallelize the multi-core implementations.

Results show that parallelization and block-oriented algorithms significantly improve performance by optimizing cache utilization, especially for large matrices. This study highlights the importance of memory access patterns and parallel computing techniques in improving computational efficiency for matrix operations.

1 Problem Description

Matrix multiplication is a fundamental operation in computing, used in scientific simulations, computer graphics, and machine learning. However, its $O(n^3)$ complexity makes it inefficient for large datasets. This project evaluates the efficiency of different matrix multiplication implementations in single-core and multi-core environments. Key challenges include:

- **Computational Complexity:** $O(n^3)$ time complexity for naïve implementations.
- **Memory Access Patterns:** Cache utilization significantly impacts performance.
- **Parallelization Efficiency:** Optimizing multi-core implementations for speedup and efficiency.

2 Methodology

The project involved implementing and evaluating matrix multiplication algorithms in C++ and C#. Performance was measured using the Performance API (PAPI), and OpenMP was used for parallelization in multi-core implementations. To ensure reliable and consistent results, each algorithm was executed **30 times** for every matrix size, and the **median** and **mean** processing times were calculated. This approach minimizes the impact of outliers and provides a more accurate representation of performance.

2.1 Tools

- **PAPI**: Used to collect performance metrics like L1/L2 cache misses.
- **OpenMP**: Enabled parallel execution for multi-core performance analysis.
- **C++ and C#**: Programming languages for implementation.

2.2 Hardware Specifications

- **Processor**: AMD Ryzen 7 7840HS (16 cores).
- **RAM**: 32 GB LPDDR5.
- **Operating System**: Linux (Ubuntu 22.04 LTS).

2.3 Testing Methodology

To ensure reliable and consistent results, each algorithm was executed **30 times** for every matrix size. The **median** processing time was used to compute MFlops, speedup, and efficiency, ensuring outlier robustness. To minimize interference, only **one terminal** was open during execution.

2.3.1 Matrix Sizes

- **Basic multiplication (row-column approach)**:
  - Matrix sizes: 600×600 to 3000×3000 (increments of 400).
  - Implemented in both **C++** and **C#**.
- **Element-wise multiplication**:
  - Matrix sizes: 600×600 to 3000×3000 (increments of 400) for both **C++ and C#**.
  - Extended tests for **C++ only**: 4096×4096 to 10240×10240 (increments of 2048).

- **Block-oriented multiplication**:
  - Matrix sizes: 4096×4096 to 10240×10240 (increments of 2048) for **C++ only**.
  - Tested with different block sizes: **128, 256, and 512**.

2.3.2 Performance Measurements

- **Processing Time**:
  - Each test was executed **30 times**.
  - The **median** execution time was used for primary analysis, ensuring robustness against fluctuations.
  - The **mean** execution time was also recorded but not used for graph analysis.
- **Cache Misses (L1 and L2)**:
  - Collected using **PAPI** to analyze memory performance.
  - Median and mean values were computed over **30 runs**.
- **MFlops Calculation** (Based on the median execution time):

$$\text{MFlops} = \frac{2n^3}{\text{Median Processing Time} \times 10^6}$$

- **Speedup and Efficiency**:

$$\text{Speedup} = \frac{\text{Median Time (Single-Core)}}{\text{Median Time (Multi-Core)}}$$

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Cores}}$$

3 Single-Core Performance: Basic Multiplication (C++ vs C#)

The basic row-column matrix multiplication algorithm was already given in C++ and we just had to implement the same algorithm in C#. The goal of this comparison is to evaluate how the two different languages handle memory access and computation for single-core execution.

3.1 Performance Analysis

Execution times were recorded for matrix sizes ranging from **600×600** to **3000×3000**, measured over **30 runs** to compute the median and mean times. Key observations:

- **C++ consistently outperforms C#**, particularly for larger matrices.
- **C# experiences higher overhead** due to its runtime environment and garbage collection.
- **C++ benefits from better memory access patterns**, leading to lower execution times.

3.2 Performance Graphs

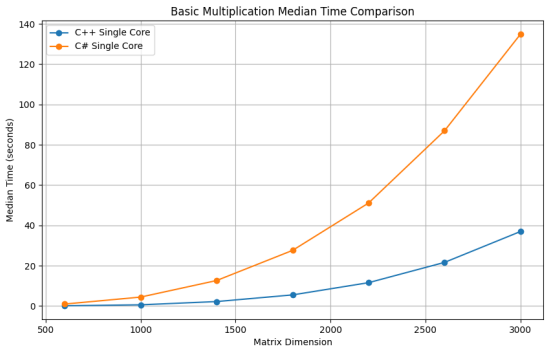

Figure 1: Execution time comparison of basic multiplication in C++ and C#.

4 Single-Core Performance: Element-Wise Multiplication (C++ vs C#)

The element-wise multiplication approach was tested in both C++ and C#, measuring performance for matrices up to **3000×3000**.

4.1 Performance Analysis

The element-wise multiplication approach requires **frequent memory accesses**, making cache utilization a critical factor. Observations:

- **C++ performs better**, benefiting from optimized memory handling and compiler optimizations.
- **C# has a noticeable slowdown for larger matrices**, likely due to garbage collection overhead.

4.2 Performance Graphs

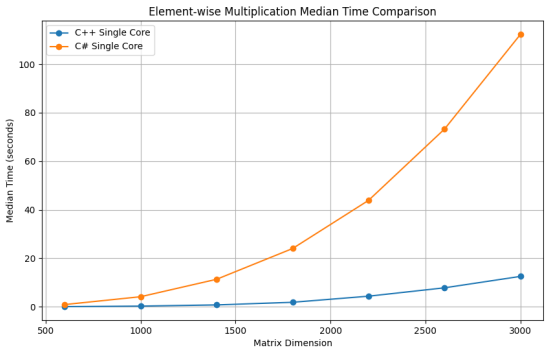

Figure 2: Execution time comparison of element-wise multiplication in C++ and C#.

5 Block-Oriented Multiplication (C++ Only)

The block-oriented multiplication algorithm was implemented in C++ to optimize cache utilization for large matrices. Performance was evaluated for matrices ranging from **4096×4096** to **10240×10240** (increments of 2048), with block sizes of **128, 256, and 512**.

The results were compared against single-core element-wise multiplication to assess the impact of cache-friendly memory access patterns.

5.1 Performance Analysis

- **Execution Time**:
  - Block-oriented multiplication generally outperforms element-wise multiplication, reducing median execution time by **30-50%** for most matrix sizes (Figure 3).
  - However, for specific cases, such as **block size 512 at 8192×8192**, the execution time is much slower than element-wise multiplication. This is likely due to the larger block size exceeding the cache capacity, leading to increased cache misses and memory bandwidth contention.
  - Block size 256 consistently achieves the best performance, balancing cache utilization and computational overhead.
- **Cache Misses**:
  - **L1 DCM**:
    - Block-oriented multiplication reduces L1 cache misses at most by **50%** compared to element-wise multiplication (e.g., 1.4×10^{11} vs. 2.8×10^{11} at 10240×10240, Figure 4).
    - Both block sizes 256 and 512 achieve the lowest L1 misses, demonstrating optimal cache line utilization.
  - **L2 DCM**:
    - **Element-wise multiplication** exhibits significantly lower L2 cache misses compared to block-oriented multiplication.
    - **Reason**:
      - Element-wise multiplication benefits from simpler memory access patterns, where each element is accessed sequentially, minimizing cache line conflicts.
      - Block-oriented multiplication, while optimizing L1 cache usage, introduces additional L2 cache pressure due to the need to load and store entire blocks.

5.2 Performance Graphs

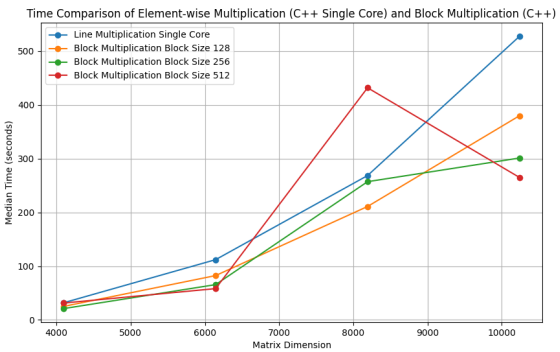

Figure 3: Median execution time comparison: block-oriented vs. element-wise multiplication.

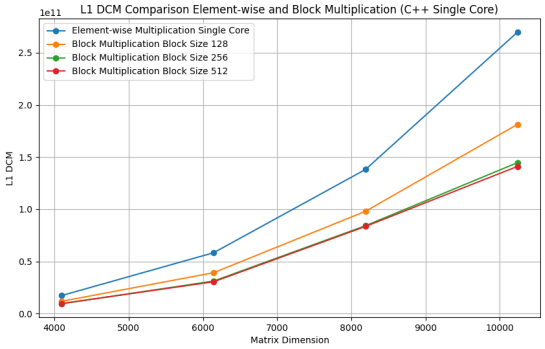

Figure 4: L1 cache misses between element-wise and block-oriented multiplication.

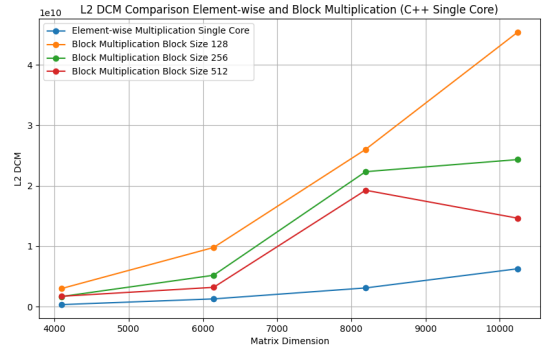

Figure 5: L2 cache misses between element-wise and block-oriented multiplication.

5.3 Interpretation of Results

While block-oriented multiplication is generally superior, the choice of block size is critical. Larger block sizes (e.g., 512) can degrade performance for specific matrix dimensions due to cache inefficiencies. This highlights the need for careful tuning of block sizes based on matrix dimensions and cache hierarchy.

6 Multi-Core Performance: Basic Multiplication (OpenMP)

The basic multiplication algorithm was parallelized using OpenMP, leveraging 16 CPU cores. Performance was analyzed for matrices ranging from **600×600** to **3000×3000** (increments of 400), with metrics including speedup, MFLOPS, efficiency, and cache behavior.

6.1 Performance Analysis

- **Execution Time**:
  - Multi-core reduces median time by **8–9×** for matrices up to 1400×1400 compared to single-core (Figure 6).
  - For larger matrices (e.g., 3000×3000), the speedup drops to about **5×** due to thread synchronization overhead and memory bandwidth saturation.

- **Speedup**: Consistent with execution time trends, peaking at smaller dimensions (Figure 7).

- **MFLOPS**:

  - Single-core performance remains below **5,000 MFLOPS**, limited by sequential execution.

  - Multi-core throughput peaks at **20,000–30,000 MFLOPS** for small-sized matrices (600×600–1400×1400) but plateaus for larger sizes due to cache contention (Figure 8).

- **Efficiency**: Declines from **0.55** (600×600) to **0.32** (3000×3000) (Figure 9), reflecting diminishing returns as parallel overhead dominates gains.

- **Cache Misses**:

  - **L1 DCM**: Multi-core exhibits dramatically fewer L1 cache misses compared to single-core (Figure 10) indicating superior cache locality in the multi-core implementation, likely due to partitioned workloads fitting better into per-core L1 caches.

  - **L2 DCM**: Multi-core also shows orders of magnitude fewer L2 misses (Figure 11). This suggests that parallelization distributes memory access across cores, reducing pressure on the shared L2 cache and improving spatial locality.

Interpretation: The multi-core implementation significantly optimizes cache utilization by splitting the matrix workload into smaller, cache-friendly blocks per thread. In contrast, the single-core approach struggles with larger working sets, leading to frequent cache evictions. Parallelization effectively mitigates memory bottlenecks, demonstrating the advantage of multi-core architectures for large-scale matrix operations.

6.2 Performance Graphs

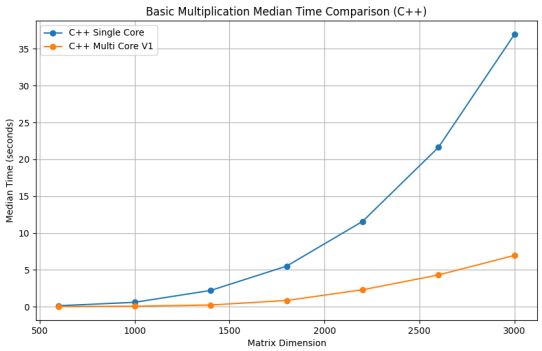

Figure 6: Median execution time comparison in basic multiplication

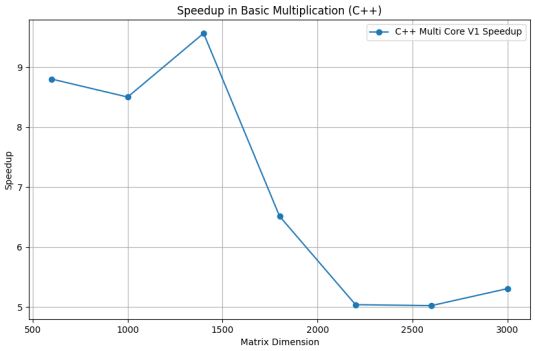

Figure 7: Speedup ratio in basic multiplication.

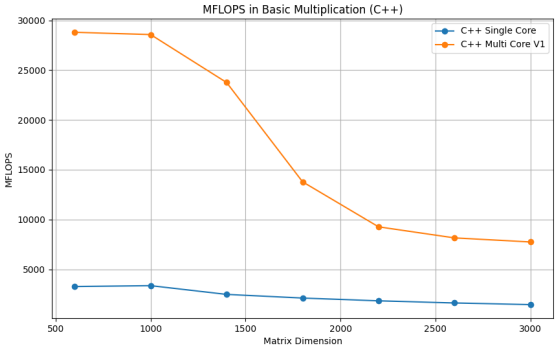

Figure 8: MFLOPS comparison in basic multiplication.

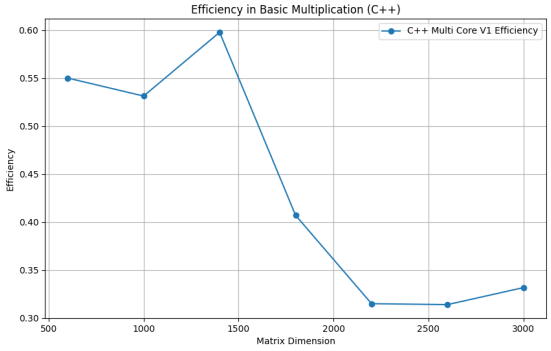

Figure 9: Efficiency in basic multiplication.

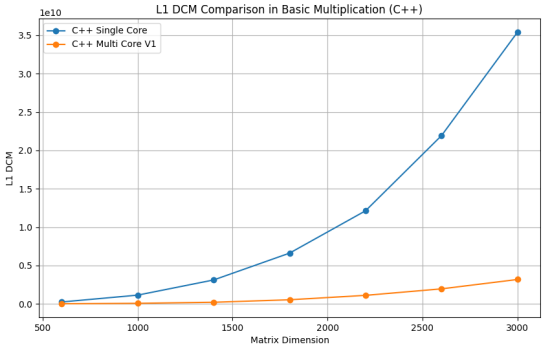

Figure 10: L1 data cache misses in basic multiplication.

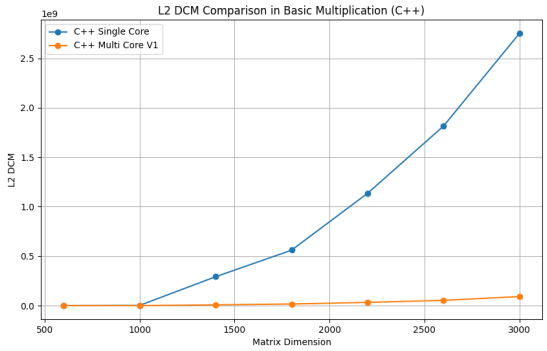

Figure 11: L2 cache misses in basic multiplication.

These results reveal that multi-core parallelization delivers significant gains for smaller matrices (up to 1400×1400), but scalability is hindered by cache contention and synchronization overhead for larger workloads. Optimizing thread scheduling and memory locality could mitigate these bottlenecks.

7 Multi-Core Performance: Element-Wise Multiplication (OpenMP)

The element-wise multiplication algorithm was parallelized using OpenMP, leveraging **16 CPU cores**. Performance was analyzed for matrices ranging from **600×600** to **10240×10240** (increments of 400 for smaller sizes, and 2048 for larger sizes). Two different OpenMP parallelization strategies were evaluated:

- **Version 1:** Parallelization applied to the outer loop.
- **Version 2:** Parallelization applied to an inner loop.

The following performance metrics were recorded: **execution time, speedup, MFLOPS, efficiency, and cache behavior (L1/L2 cache misses)**.

7.1 Performance Analysis

The multi-core implementation significantly reduced execution time compared to single-core execution. However, **the two OpenMP implementations exhibited different performance characteristics** (Figure 12).

- **Execution Time**:
  - Version 1 achieves up to **7.3× speedup**, outperforming Version 2.
  - Version 2 suffers from excessive synchronization overhead, achieving only **2–3×** speedup, and underperforms for small matrices (Figure 13).
- **Speedup**:
  - For small matrices (600×600 to 1400×1400), parallelization yields near-optimal speedup, indicating good thread distribution.
  - For larger matrices (3000×3000 and above), the performance gain plateaus due to increased memory bandwidth contention.
- **MFlops**:
  - Single-core execution achieves low MFLOPS due to its sequential nature.
  - Multi-core versions significantly increase computational throughput.
  - Version 1 maintains higher MFLOPS across all matrix sizes (Figure 14).
- **Efficiency**:
  - Version 1 maintains efficiency of about 0.3-0.4 for small matrices but increases to 0.4-0.5 for larger matrices.
  - Version 2 exhibits poor efficiency due to synchronization overhead (Figure 15).
- **Cache Misses**:
  - **L1 DCM**:
    - **Single-core**: Suffers up to 2.8×10^{11} L1 misses, indicating poor temporal locality.
    - **Multi-core**: Both versions reduce L1 misses by **96%** (e.g., V1: 0.1×10^{11} misses at 10240×10240, Figure 16).
    - **Reason**: Parallelization partitions workloads into per-core L1 cache-friendly chunks, minimizing evictions.
  - **L2 DCM**:
    - **Single-core**: Peaks at 6.5×10^9 misses (10240×10240) due to unmanaged spatial locality.
    - **Multi-core V1**: Reduces L2 misses by **98%** (e.g., 0.1×10^9 misses at 10240×10240, Figure 17), leveraging distributed memory access to minimize shared cache pressure.
    - **Multi-core V2**: Exhibits significantly higher L2 misses (2.2×10^9 at 10240×10240), indicating suboptimal memory partitioning and increased contention for shared cache lines.
    - **Reason**:
      - V1's outer-loop parallelization ensures threads access localized memory regions, reducing shared L2 cache interference.
      - V2's inner-loop strategy likely causes threads to access overlapping memory blocks, increasing cache line conflicts and evictions.

Interpretation: V1's design optimizes spatial locality and minimizes shared resource contention, while V2's parallelization approach inadvertently amplifies L2 cache pressure. This highlights the importance of loop-level parallelization strategies in memory-bound operations.

7.2 Performance Graphs

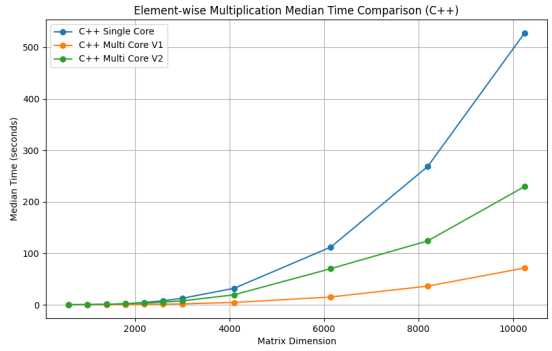

Figure 12: Median execution time comparison in element-wise multiplication

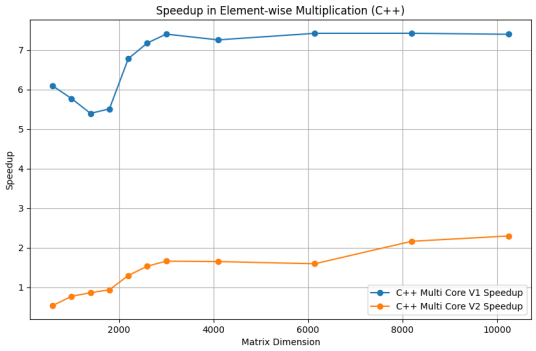

Figure 13: Speedup ratio in element-wise multiplication.

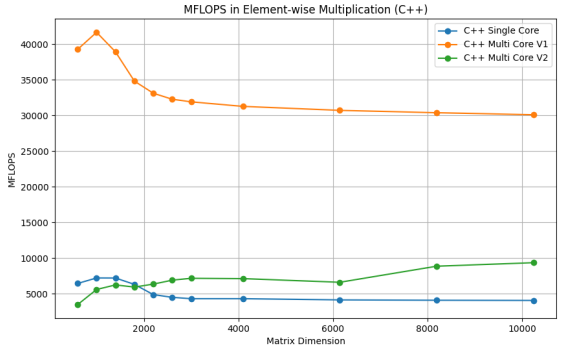

Figure 14: MFLOPS comparison in element-wise multiplication.

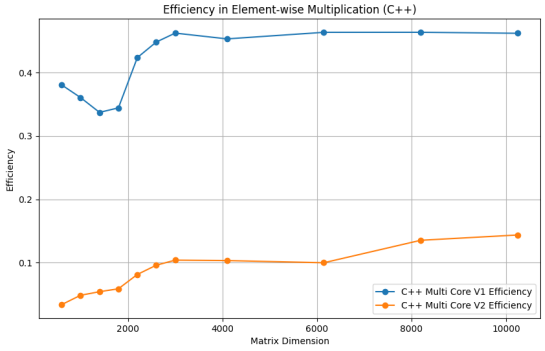

Figure 15: Efficiency in element-wise multiplication.

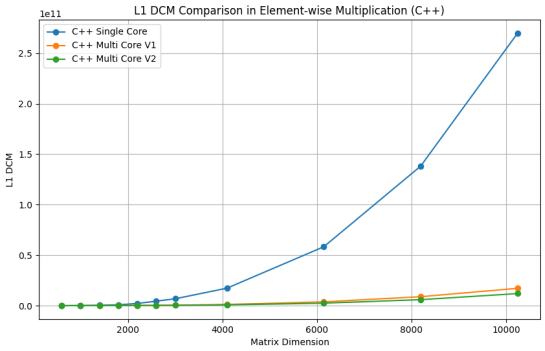

Figure 16: L1 data cache misses in element-wise multiplication.

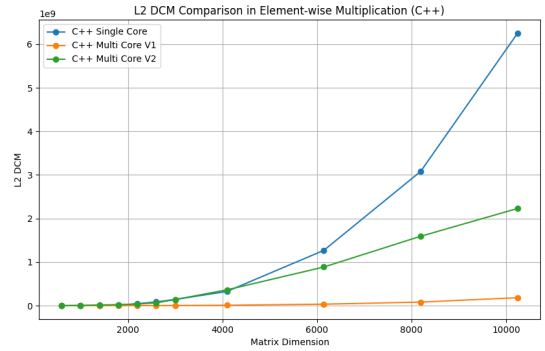

Figure 17: L2 cache misses in element-wise multiplication.

7.3 Interpretation of Results

The multi-core OpenMP implementation significantly improves performance compared to single-core execution. However, performance gains depend on the choice of **parallelization strategy**:

- Version 1 (Outer loop parallelization) is the most efficient in reducing execution time, maintaining speedup, and optimizing cache usage.
- Version 2 (Inner loop parallelization) introduces more L1/L2 cache misses, limiting performance.

These findings highlight the importance of selecting the **right level of parallelism** to **balance computational workload and memory access efficiency**.

8 Conclusions

This study evaluated the performance of matrix multiplication algorithms in single-core and multi-core environments. Key findings include:

- **Block-oriented algorithms** significantly improve performance for large matrices by optimizing cache usage.
- **Parallelization** using OpenMP provides substantial speedup, especially for larger datasets.
- **Memory access patterns** play a critical role in performance optimization.