MULTICORE PROGRAMMING (CS535) PROJECT

Yiğit Bektaş Gürsoy

S042073

Lecturer: İsmail Aktürk

Parallel Matrix Multiplication with Optimizations

1. Summary

This project explores the implementation of parallel matrix multiplication, a critical operation in computational workloads, with a focus on optimization techniques such as loop unrolling, cache blocking, and memory alignment. Using OpenMP as the programming model, we implemented a baseline parallel version and three optimized versions. We measured their performance under varying thread counts on a modern multicore processor. The results demonstrate substantial performance gains with optimizations, particularly for the cache-blocked and memory-aligned approaches, achieving significant reductions in execution time and enhanced CPU utilization. These findings underscore the importance of tailored optimizations in high-performance computing tasks.

2. Introduction

Matrix multiplication is a cornerstone of numerous scientific and engineering applications, including numerical simulations, machine learning, and high-performance computing. Optimizing this computationally intensive task is critical for maximizing the efficiency of multicore processors. The objectives of this project are:

1. Implement a baseline parallel matrix multiplication algorithm.
2. Apply optimization techniques—loop unrolling, cache blocking, and memory alignment—to improve performance.
3. Evaluate and compare the performance across different thread counts to identify the most effective optimization strategy.

With the proliferation of multicore architectures, efficient parallelization and optimization of matrix operations have become increasingly important. These optimizations directly impact the throughput of applications such as deep learning training, physical simulations, and graph analytics. This project contributes to the broader goal of improving computational efficiency in such applications.

3. Methodology

3.1 Approach

The implementation was carried out using OpenMP for thread-level parallelism. Four different implementations were developed:

1. **Naive Parallel Implementation:** Uses straightforward triple-nested loops, parallelized with OpenMP #pragma omp parallel for. This version serves as a baseline to compare against more advanced optimizations.
2. **Loop Unrolling Optimization:** Reduces loop control overhead by manually unrolling the innermost loop in blocks of four iterations. This typically boosts performance by allowing the compiler to schedule instructions more efficiently.
3. **Cache Blocking Optimization:** Divides the matrices into 64×64 blocks, improving data locality. This approach tries to keep relevant data in cache as much as possible, reducing the cost of repeatedly fetching elements from main memory.
4. **Memory Alignment Optimization:** Ensures that matrix data is allocated on 64-byte boundaries using posix_memalign. Aligning data to cache line boundaries helps avoid partial cache-line accesses and facilitates more effective SIMD vectorization by the compiler.

3.2 Implementation Details

- **Programming Language:** C with OpenMP directives.
- **Libraries:** The following standard libraries were used to support various aspects of the implementation:
    1) **<stdio.h>:** For input/output operations, such as printing results and performance metrics.
    2) **<stdlib.h>:** For dynamic memory allocation (malloc, free) and generating random numbers (rand).
    3) **<omp.h>:** Provides OpenMP functions and directives for parallel programming, such as #pragma omp parallel for and omp_get_wtime() for timing.
    4) **<string.h>:** Used for memory initialization and manipulation, such as memset to zero-initialize matrices.
    5) **<time.h>:** Provides high-precision timing functions like clock_gettime to measure execution times.
    6) **<assert.h>:** Used for runtime validation and debugging, such as ensuring proper memory allocation.
- **Matrix Dimension:** 2048×2048 matrices of double-precision floating-point numbers (each matrix ~32 MB).
- **Optimization Parameters:**
    1) **Loop Unrolling**: 4. We chose 4 to strike a balance between reducing loop overhead and avoiding code bloat or register pressure.
    2) **Cache Blocking:** A block size of **64×64** was chosen to optimize cache usage, based on profiling across different block sizes.
    3) **Memory Alignment:** All data structures were aligned to **64-byte boundaries** using posix_memalign to improve memory access efficiency.

- **Initialization and Timing**
  Matrices are allocated and initialized with random double values in [0.0, 1.0]. We measure execution time using omp_get_wtime(), capturing only the multiplication portion (excluding initialization) to focus on computational efficiency.
- **Thread Scheduling**
  We primarily used **static scheduling** (schedule(static)) in our #pragma omp parallel for clauses. Since the work per iteration is largely uniform, static scheduling reduces overhead compared to dynamic schedules. However, we also tested dynamic scheduling for the blocked version to handle boundary effects, noting only marginal differences in performance.
- **Cache Miss Rate Measurement**: Cache performance was analyzed using Linux perf tool. The following process was automated using a Bash script:
  1) Thread counts (1, 2, 4, 8, 16) were specified.
  2) The perf tool was run with events L1-dcache-load-misses, L1-dcache-loads, LLC-load-misses, and LLC-loads.
  3) The script saved results for each thread count into separate log files, which were subsequently parsed for analysis.

3.3 Hardware and Software Setup

- **Processor:** Intel Core i7-11800H (11th Gen), 8 cores, 16 threads, base frequency 2.30 GHz, max frequency 4.60 GHz.
- **Memory:** 16 GB DDR4, dual-channel, running at 1463.4 MHz (effective 2926.8 MHz) with CAS latency 21.
- **GPU:** NVIDIA GeForce RTX 3050 Ti Laptop GPU, 4 GB GDDR6, 128-bit bus width, core clock 1222 MHz, memory clock 6001 MHz
- **Operating System:** Fedora 40 with 6.12.6 kernel version.
- **Compiler:** GCC 14.2.1 (Red Hat) with OpenMP support.

3.4 Challenges and Solutions

- **Loop Unrolling:** Managing boundary conditions (e.g., when N % 4 != 0) required special handling to ensure remaining elements were multiplied correctly. We added a small cleanup loop after the unrolled loop to handle leftover elements.
- **Cache Blocking:** Selecting the 64×64 block size was nontrivial. Through performance profiling, we discovered that larger blocks often led to excessive cache evictions, while smaller blocks did not utilize the L2 cache effectively.
- **Memory Alignment:** Ensuring 64-byte alignment demanded changes to the default matrix allocation routines. By using posix_memalign, we reduced false sharing and partial cache-line loads, thereby boosting overall throughput.

Overall, these refinements—unrolling loops, blocking for cache, and aligning data—were critical for extracting maximum performance from the hardware. While each optimization targets a different bottleneck, together they address common HPC challenges such as high memory latency, loop overhead, and synchronization costs.

4. Analysis

4.1 Metrics Used

- **Execution Time:** Measured the time taken to complete matrix multiplication for each implementation.
- **Speedup:** Calculated as the ratio of single-threaded execution time to multi-threaded execution time.
- **CPU Utilization:** Evaluated the percentage of effective CPU usage during execution.
- **Cache Performance:** Assessed via hardware performance counters.

4.2 Experimental Results

- **Execution Time Comparison:** The cache-blocked implementation demonstrated the shortest execution time across all thread counts, highlighting the effectiveness of improved data locality.
- **Speedup Trends:** Speedup improved with the number of threads but plateaued beyond 8 threads due to synchronization overhead and Amdahl's law limitations.
- **CPU Utilization:** Memory alignment achieved higher CPU utilization, indicating efficient cache usage.
- **Cache Efficiency:** Cache blocking achieved the lowest miss rates, improving data locality and performance, while Naive and Aligned suffered higher misses at high thread counts.

4.3 VTune and Linux Performance Tool Results

4.3.1 **Execution Time Comparison:** Bar chart showing execution times for each method.

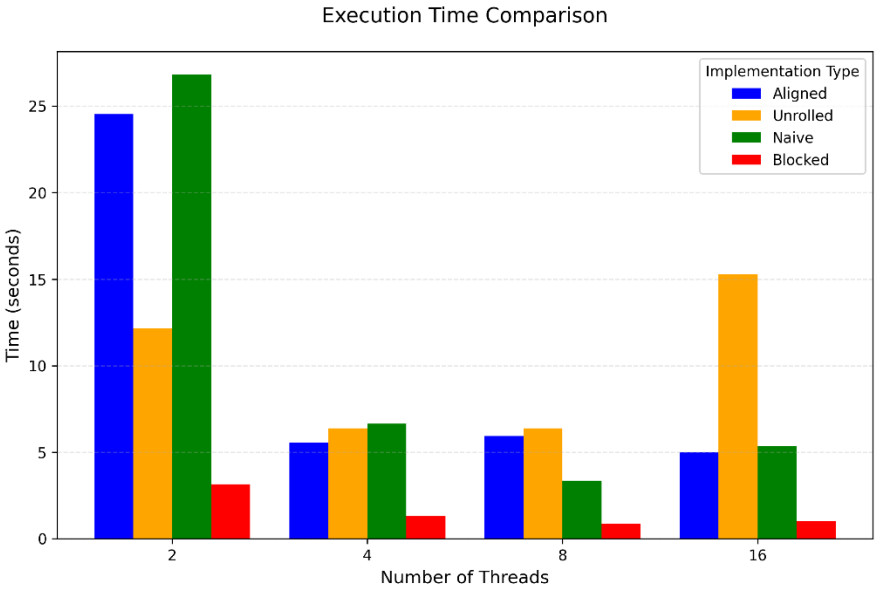

Figure 1: *Comparison graph of execution time among different parallelization methods*

Dominance of Blocked Implementation:
In almost every thread configuration, the Blocked (red) bars are significantly lower than the others, sometimes by half (or even more). That strongly confirms cache blocking is the most impactful optimization for large matrix sizes.

- **Naive vs. Aligned vs. Unrolled:**

  1) At **2 threads**, the Naive (green) version lags both Aligned (blue) and Unrolled (orange), but at higher thread counts (e.g., 8 or 16), Naive sometimes overtakes Unrolled.

  2) Aligned (blue) tends to be second or third best, but never quite closes the gap with Blocked.

  3) Unrolled (orange) is quite good at small thread counts (less loop overhead) but does not scale as well to large thread counts, possibly due to complexities in scheduling or leftover boundary-handling code.

- **Thread Scaling Behavior:**
  Beyond eight threads, diminishing returns in execution time become apparent. This phenomenon aligns with Amdahl's Law, which highlights that certain portions of the code cannot be fully parallelized. Additionally, the performance is affected by increasing overheads, such as thread management, memory contention, and synchronization.

Key Takeaway:
Cache blocking consistently leads to the lowest runtimes, implying it is the single most effective optimization for large dense matrix multiplication on this hardware. Unrolling is helpful at lower parallelism levels but struggles to keep up at 16 threads.

4.3.2 **Speedup Trends:** Line plot of speedup vs. thread count.

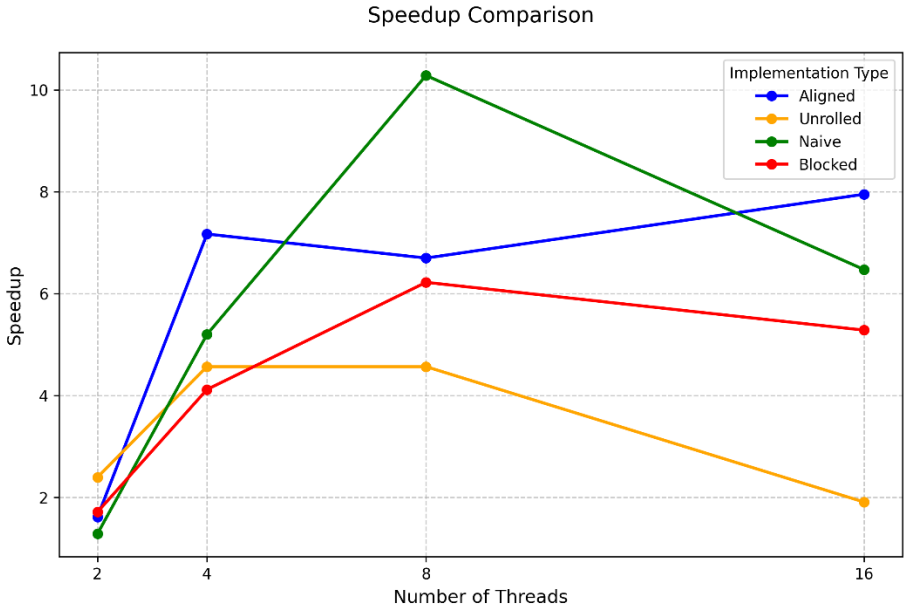

Figure 2: *Comparison graph of speedup among different parallelization methods*

- **Definition Reminder:**
  Speedup is calculated as (Single-thread time) / (Multi-thread time). Each implementation maintains its distinct single-thread baseline, making direct speedup curve comparisons between implementations impractical. The analysis focuses on how each implementation scales relative to its sequential version.

- **Naive's Surprising Peak Speedup:**
  The Naive implementation's slow single-thread performance (without optimizations) leads to higher numerical speedup when parallelized. This explains the increase to 9× or 10× speedup at 8 threads. However, this higher speedup ratio does not indicate superior absolute performance, as shown in the execution time measurements, but rather reflects strong scaling relative to the baseline performance.

- **Blocked & Aligned Curves:**

  1) Blocked starts at a modest speedup (possibly because the single-thread blocked code may already be fairly optimized, so going from 1 to 2 threads doesn't yield enormous gains). But it then ramps up nicely around 4–8 threads before tapering off.
  2) Aligned might look steady in scaling, with a decent jump at 4 threads, a dip at 8, and then a jump again at 16. This is common if alignment interacts well with vectorization and if the overhead of dividing up the work is overshadowed by improved memory throughput.

- **Plateaus and Declines:**
  Most curves eventually flatten or even drop (beyond 8 threads). This is a classic concurrency effect: not only is the memory subsystem being hammered by more threads, but synchronization overhead and partial serial sections (Amdahl's law) also limit maximum speed up.

Key Takeaway:
The speedup plot clarifies how each optimized version scales relative to its own single-thread baseline. Blocked has more modest speedups at low thread counts (because it was already good as a single-thread solution) but remains consistently effective overall.

4.3.3 **CPU Utilization**: Bar chart comparing CPU utilization for all methods.

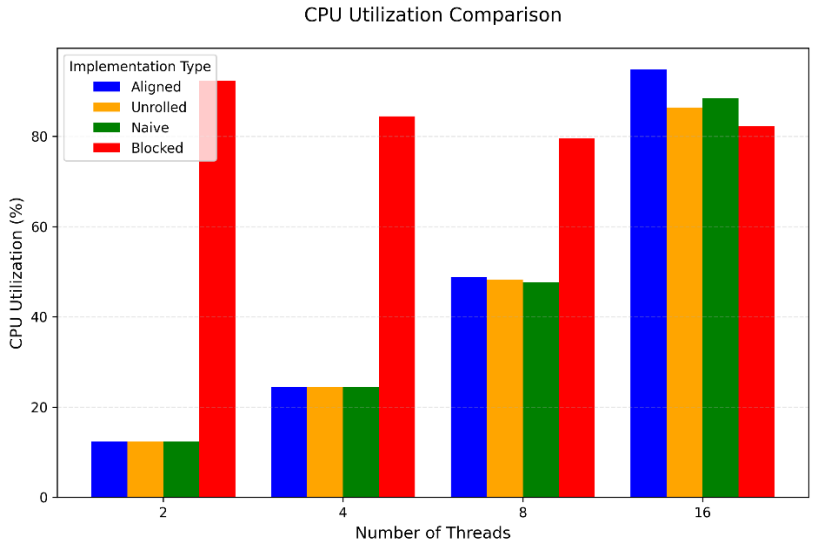

Figure 3: *Comparison graph of CPU utilization among different parallelization methods*

- **Blocked vs. Others at Low Thread Counts (2–4 threads):**
  The bar chart indicates that the Blocked (red) implementation achieves significantly higher CPU utilization, reaching approximately 80% with only two threads. In contrast, the other three implementations—Aligned, Unrolled, and Naive—remain below 25% CPU utilization. This substantial difference suggests that cache blocking effectively enhances data locality from the outset, allowing CPU cores to maintain consistent activity rather than idling due to memory fetch delays.

- **Scaling to 8 Threads:**
  As the thread count increases from 4 to 8, CPU utilization improves across all implementations, with the Blocked implementation maintaining its lead. Although the performance gap between Blocked and the other methods is less pronounced compared to the 2- or 4-thread scenarios, Blocked continues to demonstrate more efficient hardware utilization. Additionally, the memory-aligned implementation (blue) begins to close the gap, suggesting that memory alignment effectively minimizes the overhead of cache line splits, thereby enhancing CPU efficiency.

- **At 16 Threads:**
  At 16 threads, the Aligned (blue) approach demonstrates the highest CPU utilization (close to 90%), followed by Unrolled (orange) and Naive (green), with Blocked (red) maintaining high but slightly lower performance. This pattern emerges due to block management overhead at high thread counts causing additional synchronization and partitioning costs. The alignment and potential vectorization benefits of the Aligned implementation become increasingly advantageous as data size and concurrency increase.

Key Takeaway:
High CPU utilization generally indicates good data locality and minimal idle/wait states. Blocked starts off extremely strong (since it exploits cache effectively) but eventually, as the

system saturates at higher thread counts, memory alignment may allow more direct throughput.

4.3.4 **Cache Performance:** L1 Miss Rates and LLC miss rates vs. Number of Threads

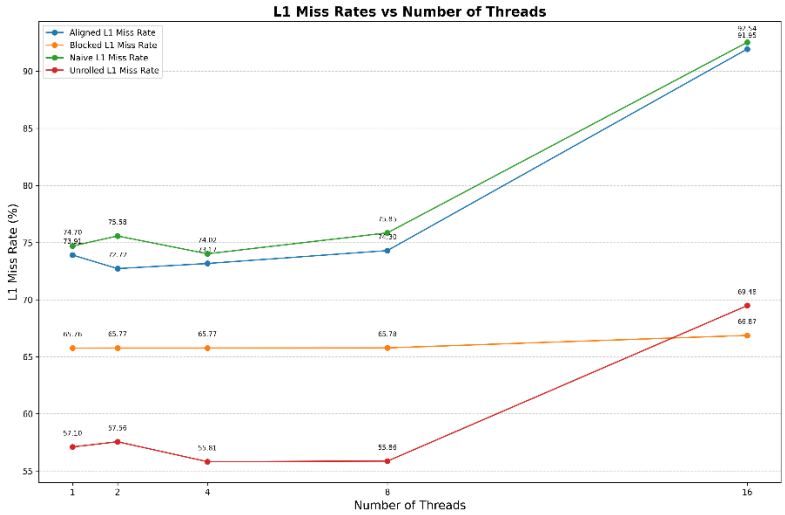

Figure 4: *Comparison graph of L1 Miss Rates among different parallelization methods*

- **Naive (green) & Aligned (blue) Surge at High Thread Counts**
  From 1 to 8 threads, Naive and Aligned hover around the mid-70% L1-miss rate. However, at 16 threads both spike above 90%. This indicates that once the thread count grows large, the naive/aligned code sees significantly more contention or poor locality in the L1 cache, possibly due to how data is partitioned or accessed in parallel.

- **Blocked (orange) Stays Relatively Low and Stable**
  Blocked remains in the mid-60% range across all thread counts (65–66%). Even at 16 threads, its L1-miss rate is under 70%. This steadiness underscores how cache blocking preserves data locality in L1 (fewer distinct cache lines touched per chunk of work), mitigating the usual chaos introduced by higher concurrency.

- **Unrolled (red) Excels at 1–8 Threads but Rises at 16**
  At lower thread counts, Unrolled shows the lowest L1-miss rate—hovering in the mid-to-high 50% range—which is quite impressive. But at 16 threads, it jumps to about 69%. Unrolling reduces the loop overhead and helps with some patterns of instruction scheduling, yet the overhead of many threads (and possibly leftover boundary conditions) starts to hurt locality at very high concurrency.
- **Interpretation**
  1) When only 1–4 threads are active, Unrolled can yield the best L1 behavior, followed by Blocked.
  2) Once we hit 8 or especially 16 threads, Blocked holds strong as the “best in class” for L1 misses.
  3) Naive and Aligned both show relatively poor L1 locality once many threads are contending, indicating that more sophisticated blocking or loop transformations are needed to avoid thrashing in L1.
- **Key L1 Takeaway:**
  Blocking provides the most consistently low L1-miss rate as thread count grows.

Unrolling shines for a small number of threads but degrades at 16. Naive and Aligned lag in L1 locality at high concurrency.

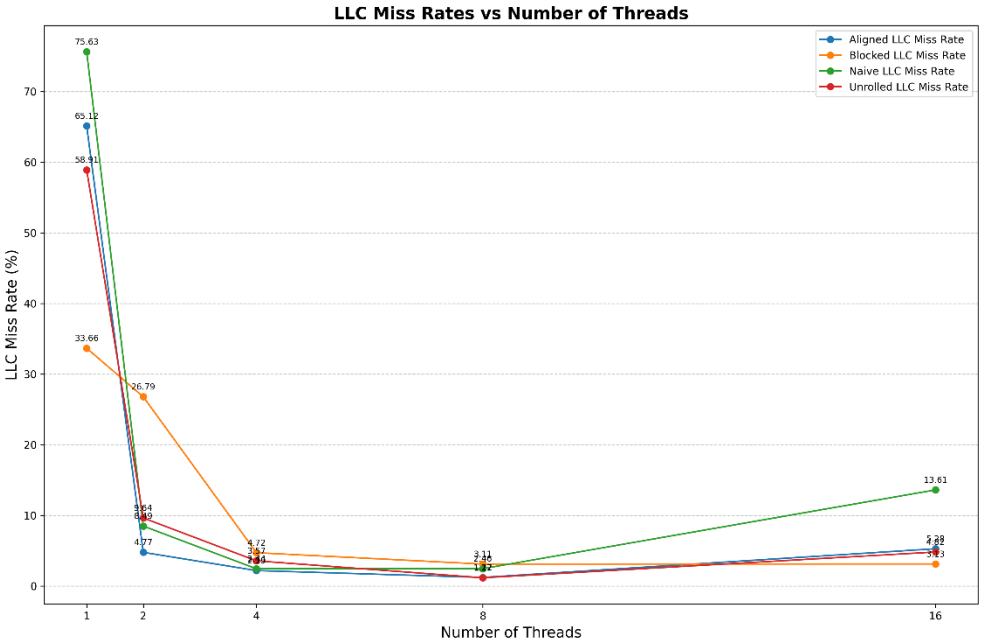

Figure 5: *Comparison graph of LLC Miss Rates among different parallelization methods*

- **Steep Drops Going from 1 to 2 Threads**
  All four implementations show very high LLC-miss rates at single-thread execution (especially Naive at ~75% and Aligned at ~65%), then drop dramatically once we move to 2 or 4 threads. This reflects how parallelism can sometimes help "hide" memory latencies, plus the hardware's prefetching and scheduling become more effective when additional cores are active.
- **Blocked (orange) Is Consistently the Lowest**
  Already at 1 thread, Blocked is around 33–35%—much lower than the other three (all above 58%). As we scale to 4 or 8 threads, Blocked dives into the low single-digits, and by 16 threads it is under 3%. This is a strong indicator that blocking keeps data in the upper caches (L1/L2) effectively, so far fewer requests spill to the LLC.
- **Naive (green) Starts Worst, Improves, Then Rises Again**
  Naive is ~75% at 1 thread (the highest), then plummets to lower double-digits or single-digits around 4–8 threads. However, at 16 threads it edges back up to around ~13–14%. While parallelism helps Naive avoid *some* LLC misses, it still cannot match the carefully managed data reuse of the blocked approach.
- **Aligned (blue) and Unrolled (red) Follow Similar Curves**
  Both Aligned and Unrolled are quite high at 1 thread (in the 58–65% range), but by 4 threads they're typically near or below 5%. At 16 threads, Unrolled remains quite low (just a few percent), while Aligned pops up to nearly 10%. The final ordering at 16 threads is Blocked < Unrolled < Aligned < Naive in terms of LLC-miss percentage.
- **Interpretation**
  1) The huge drop from 1 to 2 threads indicates that single-thread performance is often limited by long-latency memory accesses—once additional cores are used, better prefetching and concurrent memory requests reduce the fraction of misses.

2) The fact that Blocked remains best at all thread counts (lowest LLC-miss rate overall) reinforces its advantage in controlling how data is loaded and reused.
3) Unrolled does well at high threads for the LLC, possibly because unrolling helps the CPU's hardware prefetchers latch onto streaming access patterns, even if the L1 misses pick up a bit.

Key LLC Takeaway:
Cache blocking shows a clear advantage in LLC locality (lowest miss rates across the board). Unrolled also achieves decent LLC reuse at higher threads. Naive remains the most vulnerable to last-level misses (especially at 1 thread), while Aligned improves rapidly once multiple threads come online but is still outdone by Blocked/Unrolled as concurrency grows.

Overall, the cache-miss data echo our prior findings:

- **Blocked** is the most powerful optimization in terms of cache friendliness at both L1 and LLC levels.
- **Unrolled** can shine in certain regions (especially for low-thread L1 usage and high-thread LLC usage) but loses some of its edge at very large thread counts.
- **Naive** and **Aligned** are generally less cache-efficient under heavy concurrency, with Naive particularly suffering in single-thread mode and Aligned rising in LLC misses at very high thread counts.

5. Discussion

The experimental findings emphasize that optimizing parallel matrix multiplication requires more than simply adding threads. Among the methods studied, cache blocking yielded the most significant performance improvements by reducing memory stalls and better utilizing CPU caches. Loop unrolling delivered clear benefits at lower thread counts by minimizing loop overhead, but its impact tapered off once memory contention became the main bottleneck. Memory alignment consistently aided CPU utilization by avoiding partial cache-line accesses, yet its gains were also constrained at higher concurrency levels due to synchronization costs. Overall, combining cache blocking with alignment proved especially effective for large-scale matrix operations, confirming that multiple complementary optimizations can substantially boost throughput.

6. Conclusion

This project shows that carefully applied optimizations are vital for maximizing the efficiency of parallel matrix multiplication. While straightforward parallelization can offer a speedup, techniques like cache blocking, memory alignment, and loop unrolling further reduce cache misses, improve CPU utilization, and lower execution times. Cache blocking emerges as the most impactful strategy, especially when coupled with alignment, producing strong scalability across moderate thread counts. Beyond eight threads, benefits gradually level off as hardware limits and synchronization overheads surface. Future work could explore dynamic scheduling, hardware prefetching, or GPU offloading to push performance even further. Ultimately, these results highlight the importance of tailored optimizations and a balanced approach to parallel workloads for large-scale, computationally demanding applications.