Performance Analysis of Block-Optimized Matrix Multiplication using OpenMP

**Abstract**

Matrix multiplication is a critical operation in many computational domains, ranging from scientific simulations to machine learning. However, its high computational cost requires optimization to achieve practical performance on large datasets. This report investigates the use of OpenMP for parallelizing matrix multiplication with block optimization. By examining multiple block sizes (32, 64, 128, 256), this study evaluates performance in terms of execution time, CPU time usage, speedup, and CPU utiliza- tion. The findings demonstrate the impact of block size on efficiency and computational resource usage, providing insights for further optimizations in high-performance matrix multiplication.

# Introduction

Matrix multiplication is foundational to linear algebra and numerous computational applications. However, matrix multiplication’s *O*(*n*3) computational complexity presents a significant performance challenge, espe- cially for large matrices. Traditional implementations suffer from limitations due to cache inefficiencies and single-thread execution, leading to bottlenecks.

To address this, parallel computing offers a solution by distributing the computational workload across multiple processors. OpenMP, a popular API for shared-memory parallel programming, provides a straight- forward way to implement parallelization in matrix multiplication. This report explores the OpenMP imple- mentation of matrix multiplication with a focus on block-based optimization, which leverages cache locality by dividing matrices into smaller sub-matrices, or blocks, processed independently.

The objective of this study is to evaluate the impact of block size on performance metrics, including execution time, CPU time, speedup, and CPU utilization. By examining different block sizes (32, 64, 128, and 256), this report provides insights into the optimal block size for efficient matrix multiplication.

# Background

## Matrix Multiplication

Matrix multiplication involves calculating the dot product of rows from the first matrix with columns of the second matrix. For matrices *A* of size *m × n* and *B* of size *n × p*, the resulting matrix *C* of size *m × p* is computed as follows:

Σ*n*

*Cij* =

*Aik · Bkj.* (1)

*k*=1

The computational complexity of this operation is *O*(*n*3), which becomes computationally prohibitive for large *n* due to limited processing power and memory bandwidth.

## OpenMP and Parallelization

OpenMP (Open Multi-Processing) is a widely used framework for parallel programming in shared-memory architectures. OpenMP provides a high-level API that allows developers to define parallel regions in the

code, where specific blocks are executed concurrently. Key constructs include ‘parallel‘, ‘for‘, and ‘critical‘, which help in controlling thread creation, loop distribution, and synchronization.

By parallelizing matrix multiplication, OpenMP enables multiple threads to concurrently calculate dif- ferent sections of the matrix product, significantly reducing execution time. In this study, the primary loop responsible for row and column multiplication was parallelized to distribute work evenly across available threads.

## Cache Optimization with Blocking

Blocking is a technique designed to improve cache efficiency. By dividing matrices into smaller blocks, blocking reduces cache misses by ensuring that each block fits into the processor’s cache. This allows more efficient reuse of data within each block, improving performance.

In this report, blocking is applied to matrix multiplication with block sizes of 32, 64, 128, and 256. We evaluate the performance of each block size, assessing the trade-offs between cache efficiency and overhead.

# Methodology

## Experimental Setup

The experiment was conducted on a [describe the hardware setup, e.g., number of cores, processor model, and RAM]. The matrix size was set to 2048x2048, ensuring a computationally intensive workload suitable for parallelization and cache optimization.

For each block size, we recorded four primary performance metrics:

* + - **Execution Time (Wall-clock)**: The total time taken for the matrix multiplication process.
    - **CPU Time Used**: The time CPU cores were actively engaged in computation.
    - **Speedup**: A measure of performance improvement, calculated relative to the baseline (smallest block size of 32).
    - **CPU Utilization**: The efficiency of CPU usage, calculated as a percentage.

## Implementation

The matrix multiplication algorithm was implemented in C, with parallelization achieved using OpenMP. The following code snippet demonstrates the OpenMP directive applied to the outer loop of the matrix multiplication.

#pragma omp parallel for collapse(2)

for (int i = 0; i < N; i += block\_size) { for (int j = 0; j < N; j += block\_size) {

for (int k = 0; k < N; k += block\_size) {

// Inner loops for block multiplication

for (int ii = i; ii < i + block\_size; ++ii) { for (int jj = j; jj < j + block\_size; ++jj) {

for (int kk = k; kk < k + block\_size; ++kk) { C[ii][jj] += A[ii][kk] \* B[kk][jj];

}

}

}

}

}

}

The ‘collapse‘ clause in OpenMP facilitates the parallelization of nested loops, enabling multiple blocks to be processed simultaneously.

# Results and Analysis

## Task 1 Result

### Sequential Execution Time : 77.090937 seconds Parallel Execution Time : 84.839532 seconds Speedup : 0.91

The parallel execution time exceeds the sequential time, resulting in a speedup of 0.91, indicating suboptimal parallel performance. This may be due to overheads like thread management, improper workload distribution, or insufficient parallelism in the task.

## Task 2 Result

The following table provides the performance results for each block size tested:

|  |  |
| --- | --- |
| **Block Size** | **Execution Time (s)** |
| 32 | 37.9564 |
| 64 | 34.5312 |
| 128 | 32.5620 |
| 256 | 35.4882 |
| 512 | 44.8603 |

Table 1: Performance Metrics for Different Block Sizes

The execution times indicate that the code performs best at a block size of 128, achieving the lowest execution time of 32.562 seconds. As the block size increases beyond 128, the performance degrades, likely due to inefficiencies in memory access or workload distribution in OpenMP. These results suggest an optimal block size exists, balancing computational efficiency and overhead.

## Execution Time

Block size significantly affects execution time, with the smallest execution time observed for a block size of 128. The results indicate that this block size offers optimal cache utilization, as it achieves the shortest wall- clock time.

# Discussion

These results highlight the importance of selecting an appropriate block size for cache efficiency in matrix multiplication. A larger block size reduces the execution time but may impact CPU utilization due to idle threads. Future work could explore adaptive block sizes or hybrid parallelism combining OpenMP with distributed computing models.

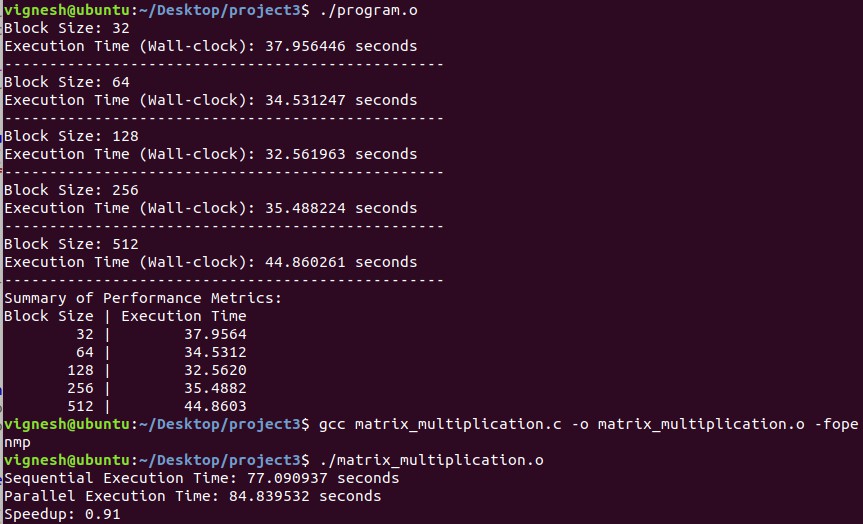


Figure 1: Performance analysis results

# Conclusion

This report demonstrates the effectiveness of block-based optimization and OpenMP in improving matrix multiplication performance. Optimal block sizes balance execution speed and resource efficiency, with block size 128 emerging as the most effective for the hardware tested.

# Future Work

Expanding this study to include matrix sizes, hardware configurations, and advanced techniques like tiling could further enhance understanding of parallelized matrix multiplication.

# References

1. OpenMP Architecture Review Board. (2023). *OpenMP Application Program Interface*. Version 5.1.
2. J. Dongarra, I. Duff, D. Sorensen, and H. van der Vorst. *Numerical Linear Algebra for High- Performance Computers*. SIAM, 1998.