Linnéuniversitetet
Kalmar Växjö

Programming assignment 1

Author: Razan Kngo, Rula Nayf, Ferzend Shekhow
Examiner: Morgan Ericsson
Course code: 4DT906

1. Introduction

Matrix multiplication is a fundamental operation in computational mathematics and scientific computing. Multiplying two matrices seems simple: three nested loops and a dot product, and you're done. But behind this simplicity lies the inefficiency, computers process the data by using cache, small but fast memory units. Caches read and store data in a row-wise when using row-major order, which is common in C and C++. This means that accessing data sequentially along rows is fast and efficient, as multiple values can be loaded into the cache in a single fetch. When matrix multiplication is implemented using the naive approach, it suffers from poor cache utilization, because of accessing matrix B column-wise. Each access may result in a cache miss, which then requires a new cache load, causing a slowdown in computation.The variation between the base implementation and the optimized version can significantly impact the performance and transform hours of waiting for results into quick responses in seconds.

In this assignment, we explore cache optimization techniques such as loop reordering, blocking (tiling), and OpenMP-based parallelization to improve matrix multiplication performance. The standard time complexity for matrix multiplication is $O(N^3)$. All methods discussed in this assignment focus on minimizing actual runtime while keeping this asymptotic complexity unchanged.

NOTE: The problem in this assignment is performed on the following system:

- **Device:** MacBook Air (13-inch, 2017)
- **Processor:** 1.8 GHz Dual-Core Intel Core i5
- **RAM:** 8 GB 1600 MHz DDR3
- **Cache:**
    - L1 Cache: 64 KB per core
    - L2 Cache: 256 KB per core
    - L3 Cache: 3 MB shared

Problem 1– Naive vs Optimized Loop Order

The first task is to implement the standard three-loop matrix multiplication. The purpose of the first problem is to calculate the product C= A*B, where A, B, and C are N*N matrices. This implementation follows the loop order ijk, where *i* iterates rowwise, *j* iterates columnwise, and *k* is used to multiply rows of A by columns of B. This method is limited because matrix B is read columnwise into a row-major stored matrix, leading to many cache misses. This negatively affects performance, especially for large matrices.

Then we improved the performance by changing the loop order from ijk to ikj, which means that matrix B is read row by row instead of column by column. This allows us to make better

use of the cache, since row-major access in a row-major matrix is much faster. In addition, we save A[i][k] in a register variable (r) to avoid reading the same data multiple times. In the results, the optimized version gave a significantly shorter execution time than the naive one.

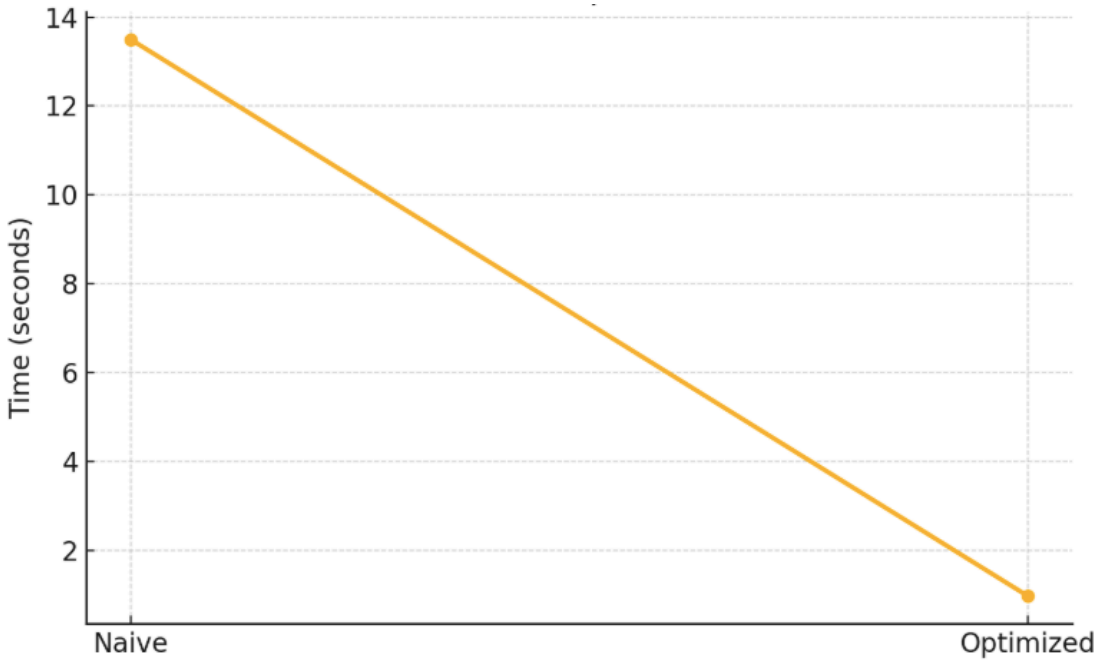

Figure 1: Shows the execution time comparison between ijk and ikj.

Problem 2 – Improvement with Blocking (Tiling)

In the second problem, we improve the performance by using blocking, where the matrices are divided into smaller blocks of size Block_Size * Block_Size. The purpose of blocking is to improve cache locality by processing small parts of the matrix at a time, which allows the data to remain in the cache during the calculation instead of being fetched again and again from RAM. This is done by looping over the blocks instead of entire rows and columns, we reduce the amount of cache misses and improve the execution time, especially for large matrices.

We test different block sizes and observe that smaller block sizes like 8 or 16 yield the best performance. However, as the block size increases beyond a certain point, such as 64, performance starts to degrade. This is because the CPU cache has a fixed capacity, and when a block becomes too large to fit entirely in the cache, the benefits of data locality diminish, and not all the needed data can be held at once.

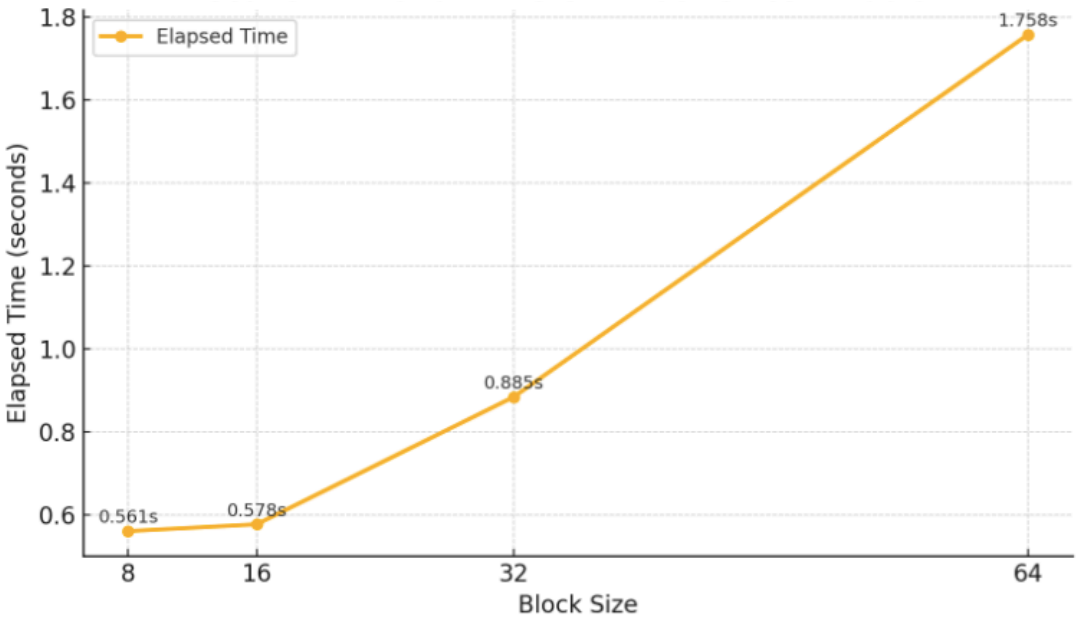

Figure 2: Shows the execution time for different block sizes in the blocked matrix multiplication.

Problem 3 – Parallel Block Multiplication with OpenMP

In the third problem, we built on the blocking from the previous task and improved the performance further by adding parallelization with OpenMP.

The goal is to utilize multiple CPU cores simultaneously to speed up the calculation. We still use blocking, where the matrices are divided into smaller blocks of size Block_Size × Block_Size. We then use OpenMP with the directive ***#pragma omp parallel for collapse(2)***, which means that the block loop over iBlock and jBlock is run in parallel across multiple threads. This allows the program to take advantage of modern multi-core CPUs and reduces the overall execution time.
The collapse(n) clause in OpenMP merges n nested loops into a single loop to increase the number of iterations available for parallel execution. This can be especially useful when the outer loop alone does not provide enough work to fully utilize all CPU threads. However, in some cases, different collapse values can result in similar performance. This typically happens when the total number of iterations is already large enough to keep all threads busy, when the workload is evenly distributed, or when the block and matrix sizes are sufficient to achieve good parallelism. In such situations, increasing the collapse level adds little benefit and may introduce some overhead.

- ➔ The program tests different:
  - block sizes: 8, 16, 32, 64
  - number of threads: 1, 2, 4, 8
- ➔ In practice we see that:
  - Small block sizes of 8 or 16 give the best performance.

- More threads reduce execution time, but the improvement decreases after 4–8 threads.

Comparison of matrix multiplication performance:

Threads	Block Size	Time (ijk)	Time (ikj)
1	8	1.48147 s	0.926113 s
1	16	1.33549 s	0.655997 s
1	32	1.70829 s	0.5139 s
1	64	1.92682 s	0.541611 s
2	8	0.710181 s	0.515033 s
2	16	0.707753 s	0.42405 s
2	32	1.20806 s	0.360542 s
2	64	1.55516 s	0.651438 s
4	8	0.811268 s	0.525063 s
4	16	0.796863 s	0.350751 s
4	32	0.87329 s	0.301115 s
4	64	1.10545 s	0.282381 s
8	8	0.673661 s	0.496994 s
8	16	0.794236 s	0.343849 s
8	32	0.861109 s	0.313352 s
8	64	1.30774 s	0.259675 s

Table 1: Execution times for ijk and ikj with different threads.