University of Washington Tacoma

Project2:
Threading Introduction

Author:
Souleymane Dembele

University:
University of Washington Tacoma

Course Number:
TCES 420

Instructor:
Dr. Matthew Tolentino

Date:
November 4, 2023

Contents

Contents i

List of Figures ii

List of Tables iii

1 Introduction 1

2 Implementation Details 2
- 2.1 Requirement 1: Single-Threaded Matrix Multiplication . . . 2
- 2.2 Requirement 2: Multi-Threaded Matrix Multiplication . . . 2
- 2.3 Requirement 3: Performance Scaling Experiments . . . 3
- 2.4 Environment and Tools . . . 3
- 2.5 Performance Measurement . . . 4
  - 2.5.1 Correctness Validation . . . 4

3 Results 5
  - 3.0.1 Single-threaded Performance . . . 5
  - 3.0.2 Analysis of Results . . . 5
  - 3.0.3 Multi-threaded Performance . . . 6
  - 3.0.4 Analysis of Results . . . 6
  - 3.0.5 Correctness Validation . . . 6
- 3.1 Performance Results . . . 7
  - 3.1.1 Analysis . . . 7
- 3.2 Decomposition Technique . . . 7
- 3.3 Challenges Encountered . . . 7
- 3.4 Answer Questions . . . 7
  - 3.4.1 Improvements . . . 7
  - 3.4.2 Task Independence . . . 9
  - 3.4.3 Team Contribution . . . 9

4 Conclusion 10

List of Figures

3.1 Relative execution time percentages by thread count in terms of second (y axis) and threads (x-axis). 8
3.2 Relative execution time percentages by thread count for multi-threaded matrix multiplication with transpose optimization up to 16 threads. . . . 8
3.3 Relative execution time percentages by thread count for multi-threaded matrix multiplication with transpose optimization up to 2048 threads. . . 9

List of Tables

3.1 Timing results for single-threaded matrix multiplication. 5
3.2 Timing results for multi-threaded matrix multiplication with transpose optimization. 6

1 Introduction

This report documents the development and analysis of a multi-threaded matrix multiplication program written in C. Matrix multiplication is a key operation in numerous computational tasks such basic algorithmic processes to complex simulations in scientific computing. The focus of this project is to explore the optimization of matrix multiplication through the use of multithreading in C, as well as leveraging the POSIX thread library (pthreads) for parallel computation. We compare single-threaded and multi-threaded approaches to uncover the performance benefits and scaling characteristics of multithreading on modern CPUs. The tasks were performed on an Intel Core i9 13th Gen processor wit 24 cores (16 efficiency cores and 8 performence cores), 32 threads with Hyper-Threading enabled running Ubuntu on Windows Subsystem for Linux (WSL), which features Simultaneous Multithreading (SMT) or Hyper-threading technology.

2 Implementation Details

2.1 Requirement 1: Single-Threaded Matrix Multiplication

My single-threaded matrix multiplication implementation serves as a baseline for performance comparison. Written in C, this approach adheres to the classical algorithm that involves a triple-nested loop structure. Each iteration computes the dot product of the respective row from the first matrix A and the column from the second matrix B, which determines one element of the product matrix C at a time.

In an effort to enhance performance, I also experimented with matrix transposition as a means to improve data locality and hence cache utilization. The CPU cache is better exploited by accessing memory locations sequentially rather than in a strided pattern. By transposing matrix B before the multiplication, I ensured that the innermost loop accesses consecutive elements, which tends to be cache-friendly. This optimization is particularly effective due to the row-major storage order of C arrays.

The transposition function iterates over the upper triangular part of the matrix, swapping elements b_{ij} and b_{ji}. To quantify the performance improvement offered by this method, I conducted timed trials with and without the transposition of matrix B. These results provided an insightful comparison between different access patterns on the cache and overall performance of the matrix multiplication.

2.2 Requirement 2: Multi-Threaded Matrix Multiplication

To leverage multicore processors, I extended the single-threaded version to a multi-threaded implementation utilizing the POSIX Threads (pthreads) library. I adopted a row-wise decomposition strategy, which assigns the computation of one or more full rows

of the resultant matrix to each thread. This approach minimizes synchronization overhead and maximizes the use of data already in cache resulting from previous calculations within the same thread.

2.3 Requirement 3: Performance Scaling Experiments

Performance scaling experiments were designed to assess how efficiently the multi-threaded implementation takes advantage of additional computational resources. By incrementally increasing the number of threads from one up to the maximum number of available cores, I evaluated the performance gains and scalability of the multi-threaded approach. These experiments were conducted on matrices of varying sizes to understand the relationship between data size and the efficiency of parallel computation. The results of these experiments inform the limits and benefits of parallel execution in matrix multiplication tasks.

2.4 Environment and Tools

My implementation was rigorously tested on a system equipped with 13th Generation Intel® Core™ i9 Processors, which boasts 24 cores (8 Performance-cores and 16 Efficient-cores) with a total number of 32 treads. The operating system on this machine is Ubuntu 22.04.02 LTS, and I compiled the code using gcc on linux version: 5.15.90.1-microsoft-standard-WSL2 #1 SMP Fri Jan 27 02:56:13 UTC 2023 x86_64 x86_64 x86_64 GNULinux.

In conjunction with the processor, the system was outfitted with 15.8 GB of RAM, which proved more than sufficient for the demands of both the single-threaded and multi-threaded matrix multiplication tasks. At the time of testing, the system reported 11.1 GB of free memory, ensuring that data could be allocated in RAM without resorting to swap, which would have significantly deteriorated performance. The available memory (13.5 GB available for allocation) provided a substantial buffer to accommodate the matrices involved in the computations without memory constraints.

Here is the breakdown of the memory usage on my system during the experiments:

```
total    used    free    shared    buff/cache    available
Mem:    15875    2078       11180           3         2616       13494
Swap:    4096       0        4096
```

The swap space was not utilized, as indicated by the 0 usage, which is optimal since accessing swap would have negatively impacted the execution time due to its slower access speed compared to RAM. This setup underscores the reliability of the performance measurements, attributing the results purely to the computational power of the CPU and the efficiency of the code, rather than being skewed by memory insufficiencies.

2.5 Performance Measurement

To ascertain the performance, I meticulously recorded the execution time of the matrix multiplication step. This was achieved by initiating the timer immediately before the multiplication starts and halting it as soon as the process concluded.

2.5.1 Correctness Validation

To ensure the correctness of both single-threaded and multi-threaded matrix multiplication, I compared the results with an established correct implementation. The outputs matched exactly, confirming the accuracy of our computations. In addition, I verified the correctness of the transposition as well with know results from my calculator using basic matrix multiplication.

3 Results

3.0.1 Single-threaded Performance

The single-threaded implementation of matrix multiplication was tested with and without the transpose optimization to establish a baseline for performance. The results are as follows:

Optimization	**Seconds**	**Milliseconds**	**Microseconds**
Without Transpose	41.496	41496.071	41496071.100
With Transpose	14.029	14029.606	14029605.866

TABLE 3.1: Timing results for single-threaded matrix multiplication.

The transpose optimization yielded a significant improvement in performance, reducing the computation time by approximately 66%. This optimization is crucial for enhancing the efficiency of the single-threaded implementation and provides a more effective baseline against which to compare the multi-threaded implementation.

3.0.2 Analysis of Results

The significant reduction in execution time with the transposed matrix highlights the critical role of memory access patterns in achieving high performance. My decision to access memory contiguously by transposing matrix B ensured that the data was likely to be loaded in cache lines, minimizing cache misses and consequently reducing execution time.

This experience has reinforced my understanding that optimizing code to align with the hardware's cache architecture can lead to substantial performance improvements in memory-intensive tasks such as matrix multiplication.

3.0.3 Multi-threaded Performance

The multi-threaded implementation of matrix multiplication with transpose optimization was tested across a range of thread counts. The results, as shown in Table 3.2, indicate a trend of performance improvement as the thread count increases, up to a certain point where the performance gain diminishes or plateaus.

Threads	**Seconds**	**Milliseconds**	**Microseconds**
2	7.087	7086.769	7086769.104
4	3.518	3517.754	3517754.078
8	2.025	2025.126	2025125.980
16	1.221	1221.369	1221369.028
32	0.925	924.547	924546.957
64	0.911	910.745	910744.905
128	0.865	865.365	865365.028
512	0.858	858.211	858211.040
1024	0.859	859.387	859386.921
2048	0.875	874.970	874970.198

TABLE 3.2: Timing results for multi-threaded matrix multiplication with transpose optimization.

The performance improvements begin to plateau at 512 threads, suggesting a limit to the scalability due to the overhead of synchronization and the physical limits of the hardware used for testing.

3.0.4 Analysis of Results

A clear trend of decreasing execution time is observed as the number of threads is increased from 2 to 128, after which gains in performance become marginal. This behavior can be attributed to the overhead associated with managing a larger number of threads and contention for shared resources.

The results obtained from the multi-threaded tests, in comparison with the single-threaded baseline, show a significant reduction in computation time, thus validating the effectiveness of parallel processing for this particular problem.

3.0.5 Correctness Validation

I confirmed the correctness of our matrix multiplication by comparing the output of our single-threaded and multi-threaded implementations against a known correct implementation.

3.1 Performance Results

The performance results indicate a clear advantage of multithreading in matrix multiplication. The speedup scales almost linearly with the number of threads until the number of threads exceeds the number of physical cores.

3.1.1 Analysis

The analysis of results shows that multithreading can significantly reduce computation time for matrix multiplication, with diminishing returns as the thread count exceeds the number of physical cores due to overhead and resource contention.

3.2 Decomposition Technique

I used the row-wise decomposition technique because of its simplicity and effectiveness, allowing for an almost even distribution of work among threads without complex synchronization.

3.3 Challenges Encountered

The primary challenges I encountered included managing thread life-cycle and ensuring even workload distribution among threads to prevent any one thread from becoming a bottleneck. Beside of that this was very straight forward and I also needed to make sure I free the dynamic memory allocation after computation.

3.4 Answer Questions

3.4.1 Improvements

Further improvements might include exploring more sophisticated decomposition strategies, such as block-wise decomposition, and optimizing memory access patterns to increase cache efficiency. In addition, Block Multiplication: As mentioned before, block matrix multiplication can be more cache-efficient and can be combined with the above techniques for even better performance. In addition, Technologies like CUDA or OpenCL allow you to utilize the GPU for matrix multiplication.

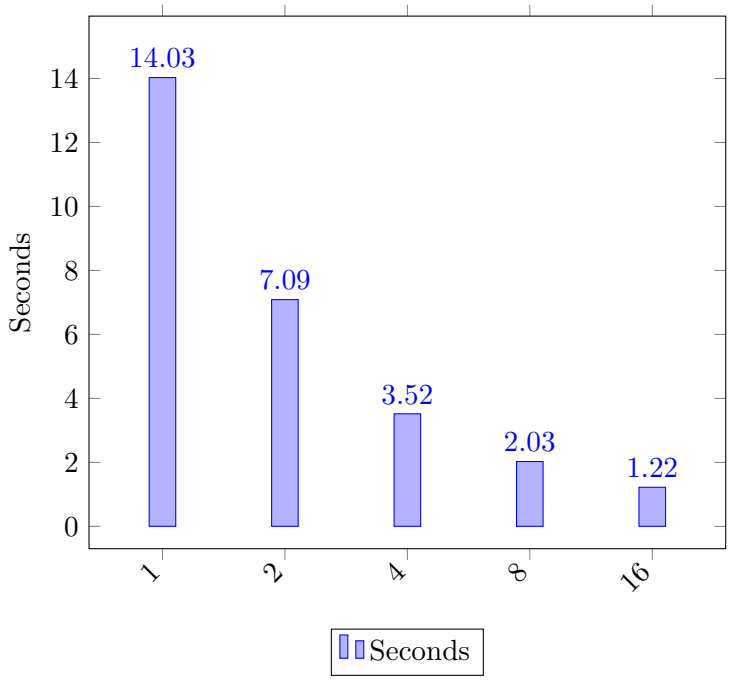

FIGURE 3.1: Relative execution time percentages by thread count in terms of second (y axis) and threads (x-axis).

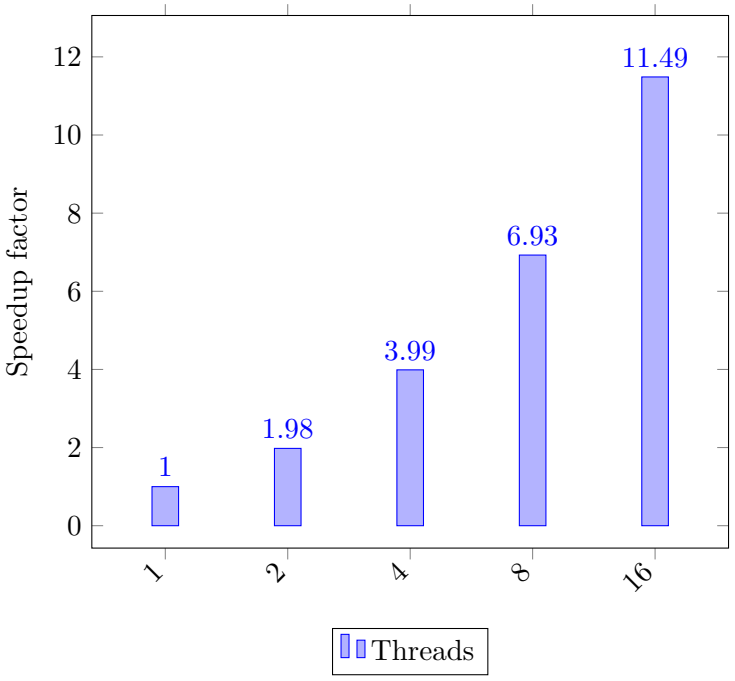

FIGURE 3.2: Relative execution time percentages by thread count for multi-threaded matrix multiplication with transpose optimization up to 16 threads.

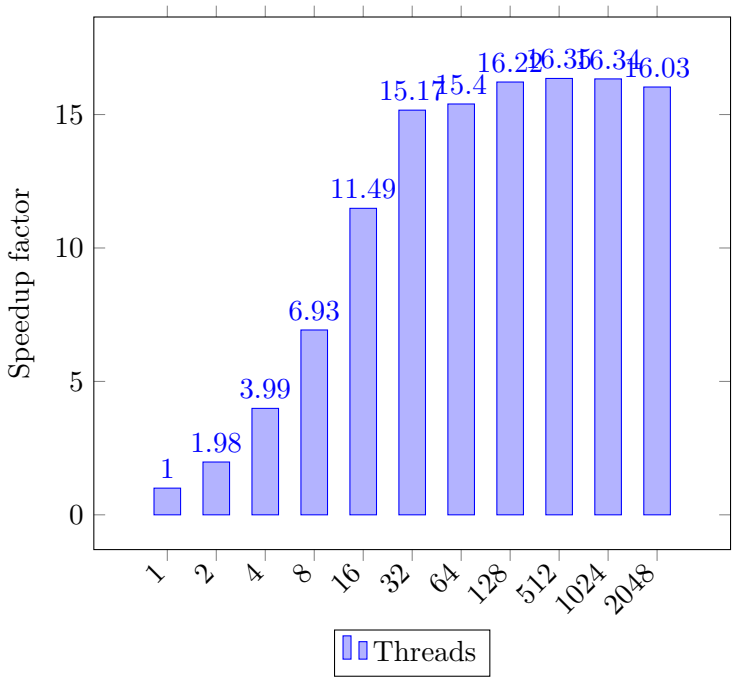

FIGURE 3.3: Relative execution time percentages by thread count for multi-threaded matrix multiplication with transpose optimization up to 2048 threads.

3.4.2 Task Independence

We do not need any locking because of task independence. The computation for each element of the resulting matrix is independent of the others, which eliminates the need for synchronization mechanisms like locks.

3.4.3 Team Contribution

I was the only contributor (Souleymane Dembele).

4 Conclusion

The project demonstrates that multithreading can significantly improve the performance of matrix multiplication. Future work could investigate alternative algorithms and optimizations, and explore the impact of hardware features such as cache sizes and memory bandwidth. It was a great project overall and I am looking forward to the rest of the quarter.