Report for Advanced Computing in Finance – Phase 1
High-Performance Linear Algebra Kernels

Yangge Xu

Introduction

In Part 1, I implemented and tested four core matrix operations: Matrix-Vector Multiplication (Row-Major), Matrix-Vector Multiplication (Column-Major), Matrix-Matrix Multiplication (Naive), and Matrix-Matrix Multiplication (Transposed B). Each function was designed to follow the required memory layout and pointer-based logic.

In this report, I will further analyze the performance of these implementations and apply optimization strategies. By focusing on aspects like cache locality, blocking, memory alignment, and compiler optimization, I aim to significantly improve the efficiency of these linear algebra kernels. Benchmark results will be collected and analyzed across multiple input sizes to support the findings.

Benchmarking

To evaluate the performance of the implemented matrix functions, I ran each function on three different matrix sizes representing small (64×64), medium (128×128), and large (256×256) workloads. For matrix-vector multiplication, the vector length matched the number of columns. Each configuration was executed 1000 times, and I recorded the average runtime (mean) and standard deviation (std dev) for each.

Due to hardware limitations, I used 1000 runs as a balance between accuracy and feasibility. More extensive testing on larger matrices could yield more stable results, but the current setup is sufficient to illustrate trends in performance.

Below is the summary of the benchmarking results:

These results show that matrix-vector operations are extremely efficient, remaining well below 1 ms even at the largest size. Matrix-matrix operations, as expected, scale more significantly due to their cubic complexity. Notably, the multiply_mm_transposed_b function performs consistently better than the naive version, especially as size increases, due to better cache utilization.

Size	Function	Mean (ms)	Std Dev (ms)
64×64	multiply_mv_row_major	0.0332	0.0109
64×64	multiply_mv_col_major	0.0275	0.0218
64×64	multiply_mm_naive	0.8919	0.2679
64×64	multiply_mm_transposed_b	0.7633	0.2343
128×128	multiply_mv_row_major	0.0453	0.0178
128×128	multiply_mv_col_major	0.0495	0.02
128×128	multiply_mm_naive	6.2803	1.1603
128×128	multiply_mm_transposed_b	6.0109	1.291
256×256	multiply_mv_row_major	0.2291	0.1795
256×256	multiply_mv_col_major	0.2033	0.0683
256×256	multiply_mm_naive	50.7387	7.5485
256×256	multiply_mm_transposed_b	48.1055	7.1193

Cache Locality

Cache locality plays a major role in the performance of matrix operations, especially as the problem size grows. In the matrix-vector multiplication functions, the row-major version accesses elements row by row, which aligns with how data is laid out in row-major order. This means memory is accessed sequentially, making efficient use of the CPU's cache lines. When the processor fetches one element, nearby elements are likely to be loaded into cache at the same time, resulting in fewer cache misses and faster performance.

On the other hand, the column-major version of matrix-vector multiplication accesses elements column by column. Since the matrix is still stored in row-major order, this approach causes jumps across rows in memory, leading to strided access. These memory jumps reduce spatial locality and increase cache misses, making this version less efficient, especially as matrix sizes increase.

For matrix-matrix multiplication, the naive version accesses matrix B by columns in the innermost loop. This is problematic because B is also stored in row-major order, and accessing it column-wise means the processor must frequently jump across memory locations. This results in poor cache utilization and degraded performance. In contrast, the transposed-B version preprocesses B into a new matrix B_T, where elements are arranged so that they can be accessed row by row. During multiplication, both A and B_T are traversed in a cache-friendly manner, minimizing memory jumps and making full use of the loaded cache lines.

As observed from the benchmark results, the row-major and transposed-B versions consistently perform better than the column-major and naive versions. Their runtimes remain relatively stable even as the matrix size increases, while the less cache-friendly implementations suffer more noticeable slowdowns and greater variance. This comparison highlights how critical memory access patterns are to performance in high-performance computing.

Memory Alignment

To evaluate the impact of memory alignment on performance, I tested all matrix operations using aligned_alloc (or _aligned_malloc on Windows) to ensure that matrices and vectors were aligned to 64-byte boundaries. This alignment matches common cache line sizes and can improve data loading efficiency by reducing unaligned memory access penalties.

For comparison, I reused the same test conditions from the previous section, where the default allocation method was used. The results here represent the same 1000-run average and standard deviation format, using aligned memory instead. By comparing these results to the original benchmarks in the first table, we can directly assess whether alignment leads to consistent runtime improvements.

Even though the overall trend is consistent with earlier benchmarks, we can see some minor improvements in the average runtime when alignment is used, especially for small and medium matrix sizes. For example, at 64×64, all four aligned implementations show slightly faster execution times than their non-aligned versions.

However, the performance gains from alignment appear to be modest and not uniform across all sizes. This suggests that while alignment does reduce overhead in memory loading, its impact may be overshadowed by other factors like cache locality and loop ordering, especially in larger matrices where memory throughput becomes more critical.

Overall, memory alignment is a safe and low-effort optimization, especially when combined with other improvements such as blocking or transposition. It ensures consistent performance and better compatibility with low-level hardware optimizations.

Size	Function	Mean (ms)	Std Dev (ms)
64×64	multiply_mv_row_major	0.0300	0.0053
64×64	multiply_mv_col_major	0.0215	0.0077
64×64	multiply_mm_naive	0.8687	0.2886
64×64	multiply_mm_transposed_b	0.7254	0.2123
128×128	multiply_mv_row_major	0.0459	0.0167
128×128	multiply_mv_col_major	0.0532	0.0262
128×128	multiply_mm_naive	6.6976	1.1864
128×128	multiply_mm_transposed_b	5.9101	1.0808
256×256	multiply_mv_row_major	0.1654	0.0233
256×256	multiply_mv_col_major	0.1933	0.0741
256×256	multiply_mm_naive	55.4259	8.8825
256×256	multiply_mm_transposed_b	50.2075	8.3319

Inlining and Compiler Optimization

To examine the impact of compiler optimizations on performance, I re-ran the benchmarking code from Part 2.1 using the -O3 optimization level. By default, the compiler runs with -O0, which applies no optimizations. In contrast, -O3 enables aggressive optimizations including automatic inlining, loop unrolling, vectorization, and other low-level transformations that can significantly reduce runtime.

In this project, I did not use any explicitly defined helper functions that would benefit from manual inline directives. Therefore, this experiment focuses purely on the effect of the compiler's built-in optimization strategies.

From these results, it is clear that enabling -O3 brings substantial performance gains. Across all test sizes, matrix-vector operations now complete in a fraction of the time compared to -O0. The multiply_mv_col_major function, in particular, benefits significantly, with runtimes dropping below 0.01 ms even at 256×256.

Matrix-matrix operations also benefit from optimization, though the effect varies. The naive implementation improves markedly, especially at larger sizes. Interestingly, the multiply_mm_transposed_b function performs worse than the naive version at 128×128 and 256×256 under -O3, possibly due to the optimizer making assumptions about memory locality or loop patterns that don't favor the manually transposed access.

Overall, these results highlight the powerful impact of compiler optimization levels on runtime performance. Even without manually marking functions as inline, -O3 allows the compiler to apply aggressive inlining and other techniques automatically, reinforcing the importance of compiling performance-sensitive code with the right flags.

Size	**Function**	**Mean (ms)**	**Std Dev (ms)**
64×64	multiply_mv_row_major	0.0077	0.006
64×64	multiply_mv_col_major	0.0043	0.0013
64×64	multiply_mm_naive	0.2168	0.0598
64×64	multiply_mm_transposed_b	0.1956	0.074
128×128	multiply_mv_row_major	0.0121	0.0038
128×128	multiply_mv_col_major	0.0073	0.0039
128×128	multiply_mm_naive	0.7741	0.1985
128×128	multiply_mm_transposed_b	1.7464	0.4927
256×256	multiply_mv_row_major	0.0656	0.0215
256×256	multiply_mv_col_major	0.0201	0.0076
256×256	multiply_mm_naive	4.9363	1.0363
256×256	multiply_mm_transposed_b	16.9897	2.5215

Optimization Strategies
For optimization, I have enabled the -O3 compiler flag, which applies aggressive optimization techniques such as automatic inlining, loop unrolling, and vectorization. I did not use any explicit helper functions, so there was no need for manual inline directives—compiler-level inlining was left to -O3.

Additionally, I used 64-byte memory alignment for all matrices and vectors through _aligned_malloc. This is intended to align memory access with typical CPU cache line boundaries, reducing misaligned memory penalties.

Beyond these compiler and system-level optimizations, I also attempted a manual algorithmic optimization based on the matrix-matrix blocking strategy described on this reference page. The optimization introduces a tiling approach, where matrix multiplication is performed on smaller sub-blocks within the matrices to maximize cache reuse and minimize cache eviction.

However, despite the theoretical benefits, this blocked implementation did not outperform the basic multiply_mm_naive or multiply_mm_transposed_b functions in my tests. I suspect this is due to multiple factors:

- The block size used (e.g., 64×64) may not be optimal for my system's cache hierarchy.
- The increased loop nesting adds overhead for small-to-medium matrices.
- The compiler under -O3 may already apply low-level loop optimizations that interfere with or duplicate the effect of my tiling.

I did not perform detailed profiling in this phase, so I cannot conclude where exactly the bottleneck lies. It's possible that under larger problem sizes, or with hand-tuned parameters, this optimization could show better performance.