1. **Simple test for each function**

A screenshot of a computer code

Description automatically generated

1. **Benchmarking**

A screenshot of a computer

Description automatically generated

1. **Cache Locality Analysis**

Matrix-vector multiplication (row-major vs. column-major):

In the row-major implementation, memory is accessed sequentially along rows, which aligns with CPU cache line behavior and takes advantage of spatial locality. In contrast, the column-major implementation accesses memory non-sequentially along columns, leading to more cache misses due to poor spatial locality. Therefore, the row-major version performs better in terms of cache efficiency.

Matrix-matrix multiplication (naive vs. transposed):

The naive implementation accesses matrix B in a column-wise fashion, which results in non-contiguous memory access when B is stored in row-major order. This leads to frequent cache misses. In the transposed B implementation, matrix B is first transposed, enabling row-wise access and therefore sequential memory access. This improves spatial locality and reduces cache misses. As a result, the transposed B version is expected to perform better, especially for larger matrices.

1. **Memory Alignment**

A screenshot of a computer

Description automatically generated

Memory alignment had little to no noticeable impact on the performance of the matrix operations across all tested sizes. For both the naive and transposed implementations, the average execution times and standard deviations of aligned and unaligned versions were nearly identical.

For small matrices (e.g., 3x3), alignment showed no consistent benefit. For medium (20x20) and large matrices (100x100), the aligned versions did not outperform unaligned ones in any significant way. In some cases, aligned versions were slightly slower.

This suggests that under the tested conditions and with current compiler optimizations, memory alignment did not provide a measurable performance improvement. The benefits of alignment may be more apparent in SIMD-heavy or low-level hardware-optimized code, or with much larger datasets.

1. **Inlining**

**A screenshot of a computer

Description automatically generated**

The results show that using the inline keyword has minimal impact on the performance of the matrix-vector column-major multiplication. For all tested sizes (3, 20, and 100), the inlined version performs nearly the same as the non-inlined version, with only very slight differences in average execution time and standard deviation.

Inlining is generally beneficial for small, frequently called functions where the function call overhead is relatively significant compared to the function body. However, in this case, the matrix-vector multiplication is already memory-bound, and the function body is relatively large compared to the cost of a function call. As a result, inlining does not lead to noticeable performance gains.

Modern compilers also automatically inline functions during optimization when beneficial. Therefore, manually adding the inline keyword may have little or no effect, especially when compiler optimizations (e.g., -O3) are enabled.

**with aggressive compiler optimizations - -O3:**

A screenshot of a computer program

Description automatically generated

**without aggressive compiler optimizations - -O0:**

A screenshot of a computer

Description automatically generated

Compiler optimizations had a significant effect on performance. Across all tested sizes and operations, the -O3 version consistently outperformed the -O0 version, with lower average execution times and smaller standard deviations.

The performance difference is most apparent for larger matrices (e.g., size 100 and 100x100), where -O3 reduces execution time by a noticeable margin. This is due to the compiler applying optimizations such as loop unrolling, instruction reordering, inlining, and better register usage under -O3. In contrast, -O0 disables most optimizations, resulting in more function call overhead, redundant memory accesses, and less efficient use of CPU resources.

1. **Profiling**

It can be referenced in the analysis.txt and README.md file. Here are some slices of reports.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

The profiling was conducted on both the naive and transposed matrix multiplication implementations. According to the gprof results, the transposed version multiply\_mm\_transposed\_b consumed the most time, accounting for 54.62% of total execution. The naive version multiply\_mm\_naive followed with 27.31%.

This shows that matrix multiplication is the dominant workload in the benchmark. Random number generation and vector resizing functions took minor portions of the runtime, and matrix-vector multiplication functions had negligible impact.

The profiler output confirms that the transposed implementation, despite taking more total time, is being called more heavily and likely handles larger data. The cache-friendly access pattern in the transposed version improves performance per operation. The call graph also indicates that the benchmark function delegates most work to these two implementations, which aligns with our understanding of their roles in the overall computation.

1. **Optimization Strategies**

We applied loop reordering to the baseline matrix-matrix multiplication function multiply\_mm\_naive to improve cache performance. In the original implementation, the innermost loop iterates over k, which multiplies matrixA[i \* colsA + k] and matrixB[k \* colsB + j]. However, accessing matrixB[k \* colsB + j] results in non-contiguous memory access due to column-wise traversal in a row-major layout.

We reordered the loops to place the j loop inside and the k loop in the middle, from

A close-up of a white background

Description automatically generated

to

A white background with black text

Description automatically generated

This change ensures that the inner loop over j accesses matrixB[k \* colsB + j] sequentially in memory (row-wise in a row-major layout), while accessing to matrixA is still row-wise. Therefore, It improves spatial locality for both matrixB and result.