# Objective

To implement and optimize fundamental linear algebra operations (matrix-vector multiplication and matrix-matrix multiplication) in C++, focusing on performance considerations such as cache locality, memory alignment, and the impact of compiler optimizations like inlining. Teams will analyze the performance of their implementations using benchmarking and profiling tools.

# Benchmarking:

A graph of different types of benchmarking

AI-generated content may be incorrect.

As seen above, the difference between the unoptimized and optimized functions execution time grows at a rapid pace when the matrix size increases. The most significant improvement relative to the unoptimized version seems to be the naïve matrix multiplication, whereas the transposed matrix multiplication only sees minimal improvements with our optimizations.

A table with numbers and symbols

AI-generated content may be incorrect.

The optimized row major performed the best on-average, which makes sense as that is more efficient than the column-major function as explained earlier, and it is only multiplying a vector by a matrix instead of a matrix by a matrix. The transposed optimization almost always performs worse than the naïve optimization in both mean and standard deviation, but it does manage to have a smaller standard deviation at a matrix size 1024. Overall, these results were fairly interesting, but are close to what we would expect.

# Cache Locality Analysis:

For the matrix-vector multiplication implementations (row-major vs. column-major), analyze the cache access patterns. Explain which implementation is expected to perform better and why, considering cache locality.

For the matrix-matrix multiplication implementations (naive vs. transposed B), analyze how the memory access patterns differ and how the transposed\_b approach might improve cache utilization.

Design and run specific benchmark cases that highlight the impact of cache locality. For example, compare performance with different strides of data access.

# Memory Alignment:

Investigate the impact of memory alignment on the performance of your matrix operations.

Modify your memory allocation to ensure that the matrices and vectors are aligned to a specific boundary (e.g., 64 bytes) using techniques like custom allocators or platform-specific alignment functions (you can also use an array).

Benchmark the aligned versions against the unaligned versions and report your findings. Did alignment provide a noticeable performance improvement? Under what conditions?

A graph of a comparison between a row of rows

AI-generated content may be incorrect.

In the provided scenario, the aligned version performed worse than the unaligned version. I think the overhead of aligned\_malloc, outweighed the benefit of the alignment, as the matrix size increases though, the aligned function starts to perform closer to the unaligned version. Our compilers likely do a fairly good job of managing and optimizing the memory access patters without alignment and that it why we are seeing a performance decrease.

# Inlining:

Experiment with the use of the inline keyword for small, frequently called helper functions within your matrix operations (if any).

Compile your code with and without aggressive compiler optimizations (e.g., -O0 vs. -O3 in GCC/Clang, /Od vs. /O2 in MSVC).

Analyze how compiler optimizations and the inline keyword affect the performance. Discuss when inlining is likely to be beneficial and when it might not be (you can study the assembly code)

Using the inline keyword contributed to some of the notable optimizations in our project. When a function is marked as inline, the compiler incorporates the function's body directly into the calling code at compile time, removing the overhead associated with function calls. Inlining is particularly beneficial for small functions that are called very frequently, as the repeated call overhead can significantly affect performance.

However, inlining is less effective or even detrimental for functions that are large, complex, or rarely called. For instance, functions triggered only by rare or specific errors would generally not benefit from inlining, as it could unnecessarily increase the executable's size and negatively impact instruction cache efficiency without providing meaningful performance gains.

When we implemented optimizations there was a significant increase in performance especially with the larger matrices. This is because the compiler now automatically inlines some functions as well as optimizing memory access, register usage, and loop unrolling.

# Profiling:

A screenshot of a computer

AI-generated content may be incorrect.

Trivially, the most time is spent dotting the two matrices.

Profile the execution of your benchmarked code for at least one of the matrix multiplication implementations (both naive and transposed B).

Identify the parts of the code where the program spends the most time.

Analyze the profiler output (flat profile, call graph, or relevant views) and relate it to your understanding of the algorithms and their cache behavior. Include screenshots or relevant excerpts from the profiler output in your report.

# Optimization Strategies (Team Brainstorming and Implementation):

Based on your analysis of cache locality, memory alignment, and profiling results, brainstorm and implement at least one significant optimization to one of your baseline matrix multiplication functions. This could involve:

Loop reordering (for better cache locality).

Blocking/tiling (for improved cache reuse).

Other relevant optimization techniques discussed in class or found through research.

Clearly document the optimization you implemented and the reasoning behind it.

Benchmark your optimized version against the baseline and report the performance improvement (if any).