# 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:

Present your benchmarking results in a clear table or graph in your report.

# 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?

# 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)

# Profiling:

* Linux: gprof and perf.
  + gprof: Compile your code with the -pg flag. Run the executable. Analyze the generated gmon.out file using gprof your\_program gmon.out. Understand the flat profile and call graph.
  + perf: Use commands like perf stat ./your\_program to get summary statistics or perf record -g ./your\_program followed by perf report for call graph analysis.
* macOS: Instruments (part of Xcode).
  + Compile with debug symbols (-g). Open Instruments, choose the "Time Profiler" template, and select your executable as the target. Record the execution and analyze the "Call Tree" and "Top Functions" views. Alternatively, use the command-line tool xcrun xctrace.
* Windows: Visual Studio Performance Profiler, Windows Performance Toolkit (WPT), or Cygwin with gprof.
  + Visual Studio: Build in Release (with optional debug info). Use the Performance Profiler (Debug -> Performance Profiler or Alt+F2) and select the "CPU Usage" tool. Analyze the summary, functions, and call tree views.
  + WPT: Install the Windows ADK. Use Windows Performance Recorder (WPR) to collect a trace while running your application, and then analyze the trace using Windows Performance Analyzer (WPA), focusing on CPU usage graphs and call stacks.
  + Cygwin (Optional): If you have Cygwin installed (a Linux-like environment for Windows), you can compile your code using g++ -pg your\_program.cpp -o your\_program within the Cygwin terminal. Run the executable (./your\_program). Then, use gprof your\_program gmon.out to analyze the profiling data. Understand the flat profile and call graph as you would on Linux.

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.

l

# 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).