# 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.

# Cache Locality Analysis:

The row major implementation is expected to perform better. This is due to the contingency of row data when it is in row major form. When performing matrix-vec multiplication, you need to dot the top row A[0][:] with the vector V[:]. If your matrix is in row-major form, the elements will be next to each other, leading to cache hits and speeding up your calculation.

The matrix-matrix functions are a bit different. When you perform matrix-matrix, you typically dot the top row A[0][:] with first column B[:][0]. But implemented this way would lead to non-contiguous access of the second matrix. If the B matrix’s transpose is stored in memory, this would lead to better performance.

A graph with a line

AI-generated content may be incorrect.If matrix B is stored as-is, a slight improvement can be made to access the matrix in a contiguous pattern. We can re-arrange the loops such that matrixB is accessed more efficiently.

Unoptimized Code

for(int i = 0; i < rowsA; ++i)

for(int j = 0; j < colsB; ++j)

for(int k = 0; k < colsA; ++k)

result[i \* colsB + j] += matrixA[i \* colsA + k] \* matrixB[k \* colsB + j];

Optimized Code

for(int i = 0; i < rowsA; ++i)

for(int k = 0; k < colsA; ++k)

for(int j = 0; j < colsB; ++j)

result[i \* colsB + j] += matrixA[i \* colsA + k] \* matrixB[k \* colsB + j];

A graph with a line and a blue line

AI-generated content may be incorrect.A similar approach can be done for multiply\_mv\_col\_major

Unoptimized Code

for (int j = 0; j < cols; ++j)

for (int i = 0; i < rows; ++i)

result[i] += matrix[j \* rows + i] \* vector[j];

Optimized Code

for (int i = 0; i < rows; ++i)

for (int j = 0; j < cols; ++j)

result[i] += matrix[j \* rows + i] \* vector[j];

# 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:

A screenshot of a computer

AI-generated content may be incorrect.

Trivially, the most time is spent dotting the two matrices. But it’s hard to see which part of this line is consuming the most time.

A computer screen with text and numbers

AI-generated content may be incorrect.

We expanded the loop to see which part consumed the most time. However, this lead to counter intuitive results. Instead, the profiler sees that sum+=valTot; is waiting for the previous lines to finish and thus contributes the most time to this line.

Since matrix A is being accessed contiguously, and matrixB is not, it is highly likely that grabbing the data from matrixB is consuming the most time. By moving the j loop inside, we can access matrixB in a contiguous pattern, leading to better code performance. The rest of the logic is addition and multiplication which can also be sped up. We can use pointers, so the indexes aren’t calculated in every loop.

# A graph with different colored lines AI-generated content may be incorrect.Optimization Strategies (Team Brainstorming and Implementation):

We decide to focus on the matrix-matrix multiplication where each one is in row-major form. This is a very common operation so it’s good to focus on this one. From our analysis this function suffered from non-contiguous memory access in matrix B.

The first improvement was made during our analysis of Cache Locality. By simply reordering the loops, we able to access matrixB in a contiguous pattern. This was a significant improvement. The next improvement was to unroll the innermost loop. This allows the code to perform multiple multiplications and additions in a single loop iteration. This led to our biggest improvement. The final improvement was to use pointers to access the data. This allows us to avoid calculating the indexes in every loop. This led to a small improvement.

Overall, our optimizations significantly improved the matrix-matrix multiplication function. We reordering the loops for better cache locality, unrolled the innermost loop for vectorized operations, and used pointers to reduce index calculation.