# Phase 1

* Group: 7
* Group members: Scott Turro, Kyle Parran

# 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:

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:

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:

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.

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. 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.