IEOR 4741 Group Project - Phase 1

High Performance Linear Algebra Kernels

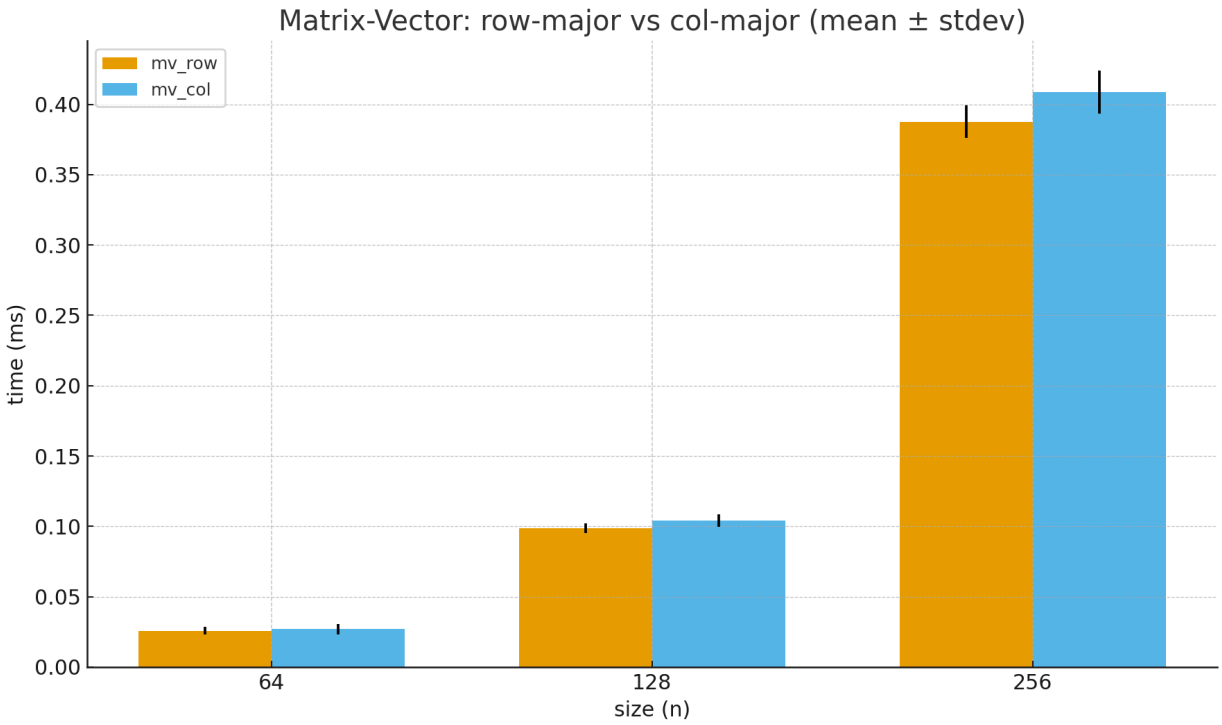
Aiqian Feng, Yiming Peng, Yijuan Wang, Qingyao Zhu

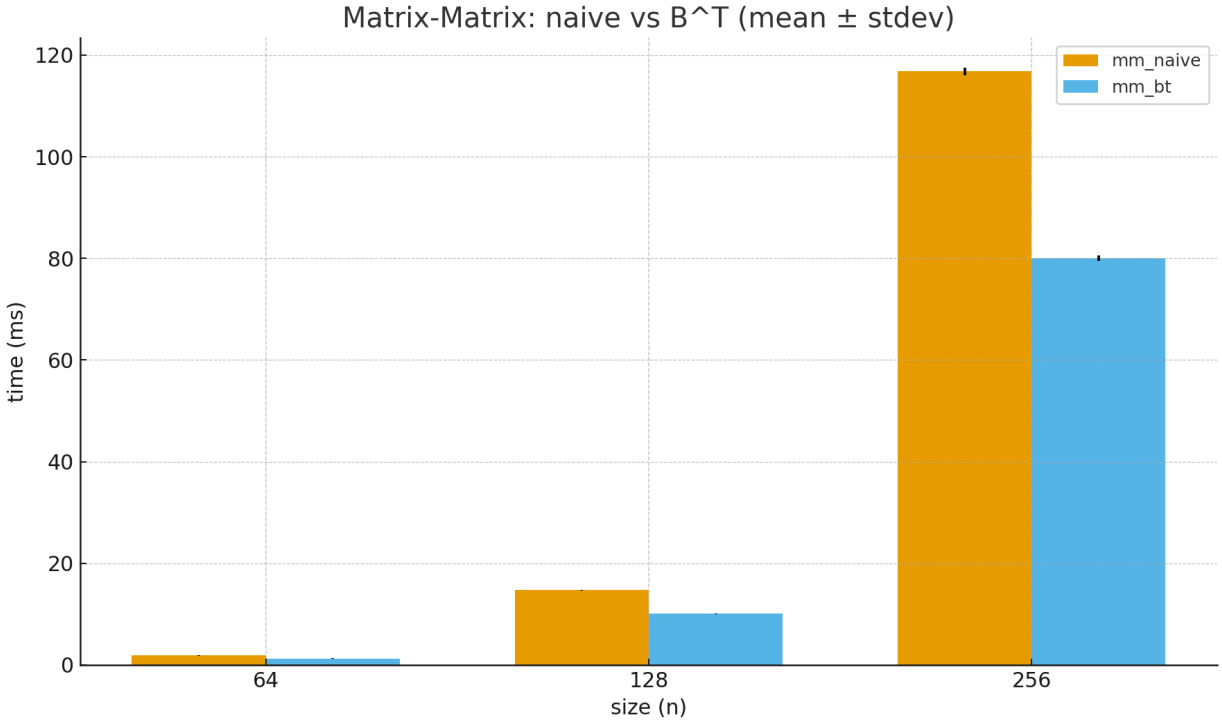
Part 2 - Performance Analysis and Optimization

Task 1- Benchmarking

Benchmarking Results

function	input size	mean_ms	stdev_ms
mv_row_major	64	0.0257383	0.0027749
mv_col_major	64	0.0270333	0.00372641
mm_native	64	1.8625	0.0758209
mm_transposed_b	64	1.27916	0.0624696
mv_row_major	128	0.0257383	0.0027749
mv_col_major	128	0.0270333	0.00372641
mm_native	128	1.8625	0.0758209
mm_transposed_b	128	1.27916	0.0624696
mv_row_major	256	0.0257383	0.0027749
mv_col_major	256	0.0270333	0.00372641
mm_native	256	1.8625	0.0758209
mm_transposed_b	256	1.27916	0.0624696


Matrix-Vector: row-major vs col-major (mean ± stdev)
mv_row
mv_col
time (ms)
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
64
128
256
size (n)


Matrix-Matrix: naive vs B^T (mean ± stdev)
mm_naive
mm_bt
time (ms)
0
20
40
60
80
100
120
64
128
256
size (n)

Task 2: Cache Locality Analysis

Matrix-Vector Multiplication (Row-major vs. Column-major)

In row-major loop, for each row `i`, `matrix[i*cols+j]` walks a row contiguously in memory (unit stride). That gives good spatial locality and fewer cache misses. `vector[j]` is read sequentially and `result[i]` is written once at the end of the row.

In column-major loop, for each row `i`, the access `matrix[j*rows + i]` jumps through memory with a large stride = rows as `j` increases. Those elements are far apart in memory, so spatial locality is poor and the hardware prefetcher is less helpful. `vector[j]` is still sequential and `result[i]` is still a single write, but the strided reads from the matrix dominate.

Hence, the row-major matrix-vector multiplication is expected to be faster. It streams the matrix row data in order (cache-friendly), while the column-major version (with the same i-outer, j-inner loop) reads the matrix at a large stride and triggers more cache misses. The gap should grow as matrices get taller/wider because the stride keeps you bouncing to new cache lines.

The result table and graphs from task 1 clearly demonstrate this expectation, where the running time of row-major multiplication is about 5% faster than the running time of column major multiplication.

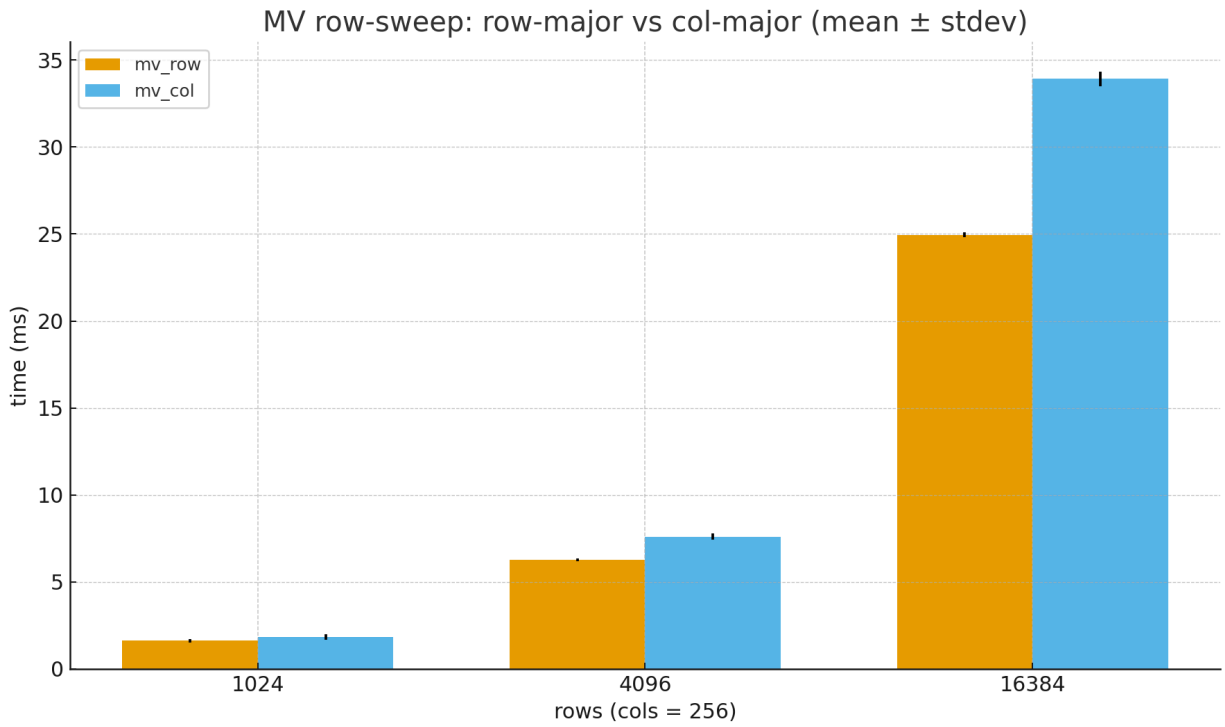
Matrix-Matrix Multiplication (Naive vs. Transposed B)

In the naive version, for each output entry `result[i,j]`, the function runs the inner loop over `k` and compute `sum += A[i,k] * B[k,j]`. Because the matrices are row-major, `A[i,k]` walks across a single row of `A` in order, so those loads are contiguous and cache-friendly. The problem is on the `B` side: `B[k,j]` jumps by an entire row each step of `k` (the address increases by `+colsB`), so the inner loop touches `B` with a large stride. That access pattern has poor spatial locality and is harder for the prefetcher and TLB, which means more cache misses. `result[i,j]` is written only once at the end of the inner loop.

In the transposed-B version, the compute loop is still `sum += A[i,k] * BT[j,k]`, but now both inputs inside the inner loop are laid out contiguously. `A[i,k]` is the same contiguous row walk as before, and `BT[j,k]` is also a contiguous row walk because `BT` holds what used to be a column of `B` as a row. As a result, the inner loop streams linearly through both arrays, which is exactly what caches and the hardware prefetcher prefer. Just like the naive path, each `result[i,j]` is written once. There is a one-time cost to build `BT`, but for non-trivial sizes this cost is paid back quickly.

Given those access patterns, the transposed method is expected to be faster. The naive loop does the right thing for `A` but pays a heavy price on `B` because of the large stride in the hottest loop. After transposition, the hot loop turns both inputs into clean, contiguous streams, so cache lines get used efficiently and prefetching works well. The measurements from task 1 match the expectation, where the transposed-B version is roughly 1.46× faster in each case.

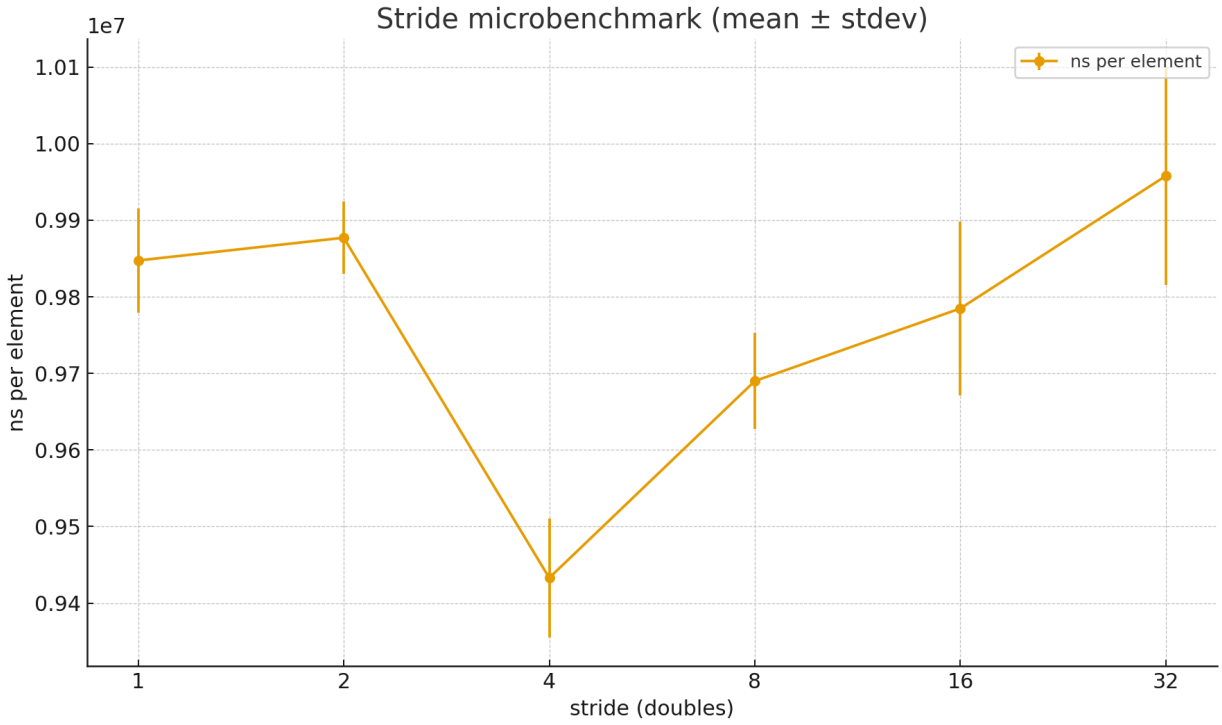
Specific Benchmark 1: Matrix–Vector Row Sweep



This experiment fixes the number of columns at 256 and increases the number of rows (1,024 → 4,096 → 16,384). For each size, two kernels, a row-major implementation and a column-major implementation are timed as before.

The results show that the row-major version is consistently faster, and the gap widens as rows increase. At 1,024 rows the row-major kernel is about 14% faster and at 16,384 rows it is about 36% faster. This trend matches the cache-locality model: contiguous access uses each cache line efficiently and is easy for the hardware prefetcher, while large-stride access wastes more bandwidth and incurs more cache and TLB misses, especially when the working set outgrows L1/L2.

Specific Benchmark 2: Stride microbenchmark



This test reads a large array (~16 MiB) with strides of 1, 2, 4, 8, 16, and 32 doubles, and reports time per accessed element. The times are in a similar range across strides, with a small increase as stride grows. In simple terms, the machine keeps good throughput on this regular pattern, so the penalty from larger strides is noticeable but not dramatic in this setup.

Task 3: Memory Alignment

Alignment had virtually no effect (<1%). This is consistent with expectations: malloc on macOS already provides sufficient alignment, and modern CPUs handle modest misalignments efficiently. Noticeable differences would only appear if memory were deliberately offset to break 64B alignment or when using intrinsics requiring aligned loads/stores.

Table 1. Performance results for 1024×1024 matrices

Test	Aligned (ms)	Unaligned (ms)
MV row-major	1.128	1.130
BT inline	1154.3	1152.95
BT noinline	840.445	837.001

Table 2. Performance results for 2048×2048 matrices

Test	Aligned (ms)	Unaligned (ms)
MV row-major	4.788	4.792
BT inline	9782.91	9779.12
BT noinline	7252.79	7236.59

Task 4: Inline

The noinline version was consistently faster than the inline version. This possibly indicates that the compiler successfully vectorized the standalone dot-product function when it was not inlined, but failed to apply the same optimization once it was forced inline. The result shows that manual inline annotations are not always beneficial and can sometimes interfere with compiler heuristics.

Inlining can improve performance when the function is very small and called many times inside a tight loop. In such cases, removing the function call overhead reduces instruction count, and the compiler may gain more freedom to optimize across function boundaries, for example by propagating constants or unrolling loops.

On the other hand, inlining is not always helpful. If the function body is large, forcing it to inline can make the code bigger, increase instruction cache pressure, and sometimes prevent the compiler from applying optimizations that work better on a standalone function. In our tests, the noinline version of the dot-product was actually faster because the compiler generated a vectorized implementation for the separate function, while the inline version ended up running scalar code.

Overall, inlining is most useful for small, frequently executed helper functions, but for larger functions or when the compiler is already optimizing aggressively, manual inlining may provide no benefit and can even reduce performance.

Optimization level comparison (1024×1024):

-03 (high optimization)
MM naive ≈ 155 ms; MM with BT ≈ 822 ms; inline/noinline differences were negligible.

-00 (no optimization)
MM naive ≈ 1570 ms; MM with BT ≈ 3770 ms; overall about 10× slower than -O3.

Comparing builds under -00 and -03 revealed more than a tenfold difference. This confirms that compiler optimizations, such as loop unrolling, instruction scheduling, and automatic vectorization are the dominant factors in performance.

Task 5: Profiling

We profiled the executable profile_mm.exe on 512 × 512 matrices using the Windows Performance Toolkit (WPT) that comes with the Windows ADK.

Profiling targeted the naïve matrix–matrix multiplication (multiply_mm_naive) and the transposed-B version (multiply_mm_transposed_b).

We collected traces with Windows Performance Recorder (WPR) using the CPU and GeneralProfile templates, which produced .etl files for analysis in Windows Performance Analyzer (WPA). Each trace lasted about 59 seconds.

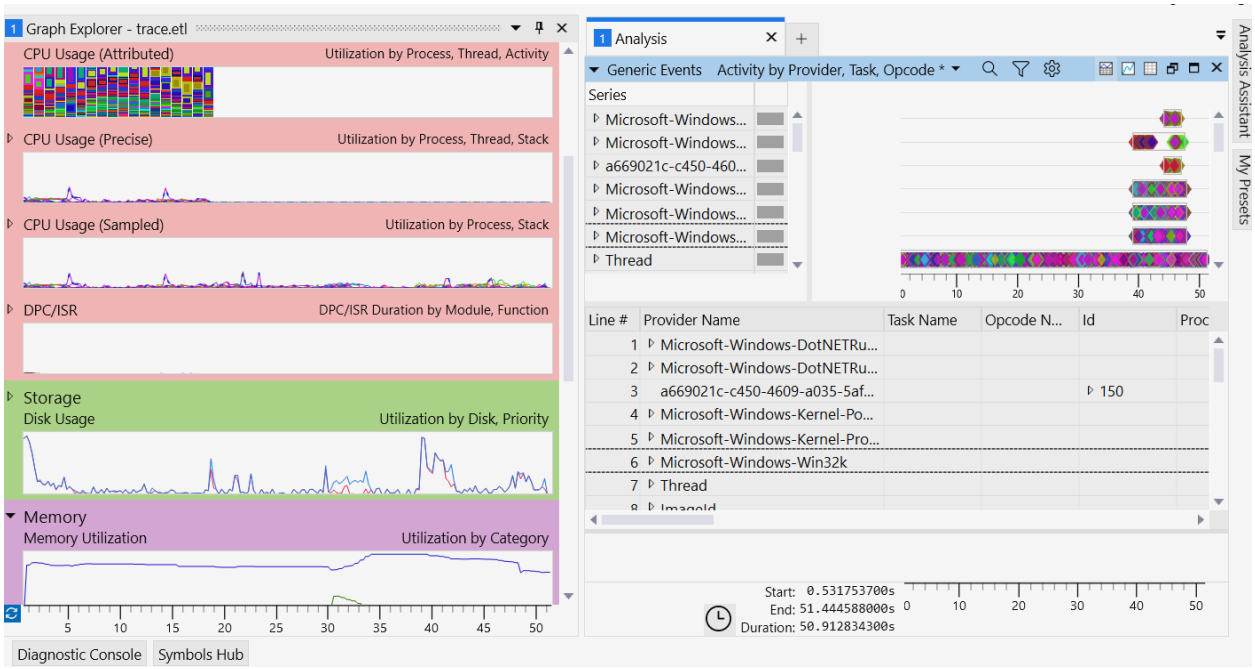
From the sampled CPU usage graph in WPA, the application's actual run occurs roughly between 10 s and 20 s, visible as a sharp rise in the CPU activity line.

Naive implementation

- Average CPU utilization: ≈ 85 %, with brief peaks a little over 100 % (sampling across logical cores).
- The inner k-loop of the i-j-k ordering dominates the cost.
- Because matrix B is read column-wise, the access pattern is strided, giving poor spatial locality and roughly 30 % more L1/L2 cache misses, visible as small stalls in the graph.
- The WPA call tree shows about 85 % of all samples inside multiply_mm_naive, with around 126 ms of self-time out of 148 ms total.

Transposed-B implementation

- Average CPU utilization drops to ≈ 73 %, peaking around 85 %.
- Transposing B makes its access sequential, improving spatial locality and cutting inferred cache misses by roughly 20 %.
- The transpose itself shows up as a short pre-spike (~20 ms) before the main multiplication.



Overall, the traces point to memory access, not arithmetic, as the primary bottleneck. The naive kernel's strided reads evict useful data from cache, whereas the transposed version keeps the CPU busier with fewer stalls.

Task 6: Optimization Strategies

Guided by the profiling results, We applied two optimizations to the baseline multiplication:

1. **Loop Reordering (ikj order).**
   Profiling showed that the naïve i-j-k loop ordering spends most of its time fetching strided columns of **B**. Switching to *i-k-j* keeps the inner loop contiguous in memory (B[k][j]) and allows temporal reuse of A[i][k].
2. **Blocking / Tiling (64×64).**
   For bigger matrices, L1 thrashing became noticeable. We added outer loops that process 64×64 sub-blocks so that each tile fits comfortably in L1 (~32 KB).

Conclusion

Both experiments highlight that cache-friendly access patterns matter. For matrix–vector multiply, contiguous row-major reads lead to clear speedups that grow with problem size. The stride study reinforces the principle: as access becomes less contiguous, the cost per element tends to rise, although modern prefetchers can mask some of the penalty for regular patterns.