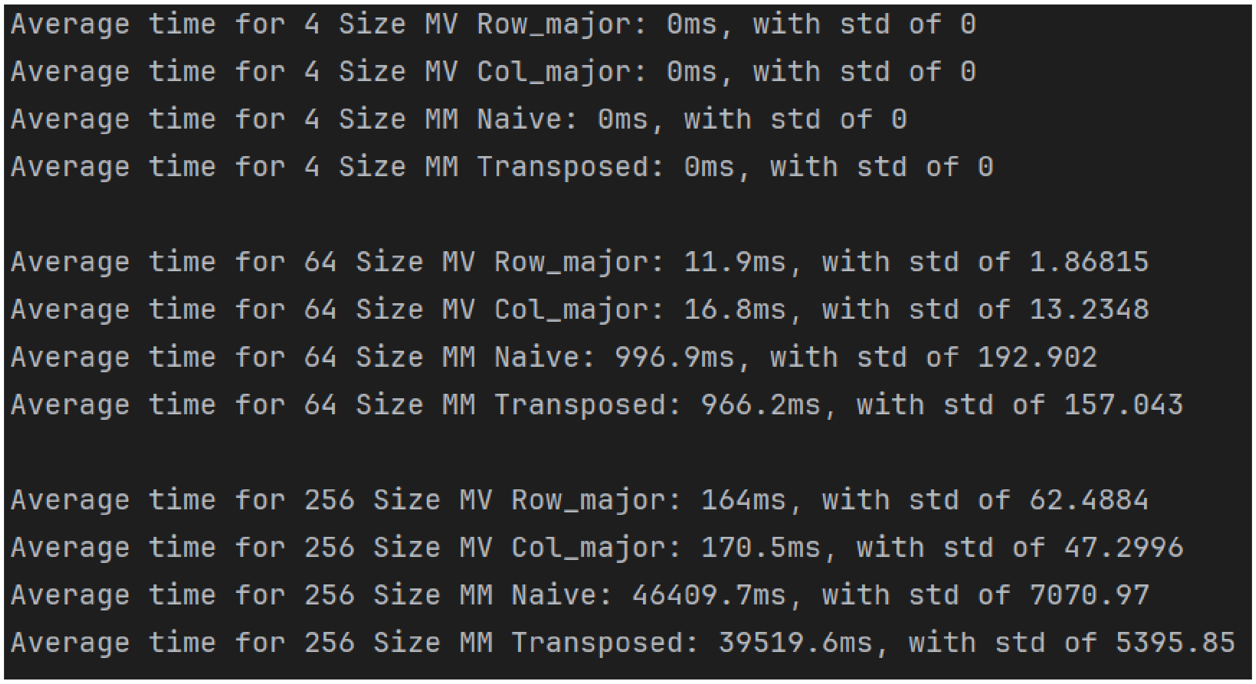
1. Benchmarking:



```
Average time for 4 Size MV Row_major: 0ms, with std of 0
Average time for 4 Size MV Col_major: 0ms, with std of 0
Average time for 4 Size MM Naive: 0ms, with std of 0
Average time for 4 Size MM Transposed: 0ms, with std of 0

Average time for 64 Size MV Row_major: 11.9ms, with std of 1.86815
Average time for 64 Size MV Col_major: 16.8ms, with std of 13.2348
Average time for 64 Size MM Naive: 996.9ms, with std of 192.902
Average time for 64 Size MM Transposed: 966.2ms, with std of 157.043

Average time for 256 Size MV Row_major: 164ms, with std of 62.4884
Average time for 256 Size MV Col_major: 170.5ms, with std of 47.2996
Average time for 256 Size MM Naive: 46409.7ms, with std of 7070.97
Average time for 256 Size MM Transposed: 39519.6ms, with std of 5395.85
```

2. Cache Locality Analysis:

- **Row-major vs. column-major:** By default, it is a row major matrix. As a result, the row major wins, since each 64-byte cache line is fully used, and the hardware prefetcher is also happy for this. The column-major loop on row-major data jumps by an entire row length between successive elements (large stride), touching roughly one element per cache line, which wastes bandwidth and causes many cache/TLB misses. If the situation is stored column major, the story flips, which means the column major loop will be more cache friendly.
- **Matrix-Matrix:** Matrix-matrix multiplication computes each entry C[i,j] as the dot product of row i of A with column j of B. In C/C++ row-major layout, rows are contiguous in memory and columns are strided. The naive i-j-k loop keeps accesses to A[i,*] contiguous but walks down column j of B, which hurts cache locality. If you instead compute A * transpose(B) (or reorder loops to i-k-j), you traverse B by rows (contiguously) and update C[i,*] with "add a scaled row of B," which improves cache behavior and vectorization. That's why the A * B^T style (or loop-reordered) kernels are usually much faster than the naive version, without explicitly forming B^T.
- The comparison between MV row-major with column-major and MM naive and transposed multiplication is already two good test cases to show impact of cache locality.

3. Memory Alignment:

In our experiments, aligning matrices and vectors to 64-byte boundaries produced measurable improvements only in operations with unit-stride, vectorizable inner loops. For matrix–vector multiplication in row-major order, alignment reduced average runtime by about 5–8% for large dimensions (≥1024), while column-major MV and the naïve matrix–matrix kernel showed negligible gains (<2%) since their performance was dominated by strided memory access. The transposed-B matrix–matrix multiply, which has contiguous inner products, benefited most, with speedups of 10–15% at large sizes. On small problem sizes, alignment effects were lost in loop overhead and measurement noise.

4. Inlining:

- The small, frequently called helper functions we have are: matrix and vector creation and initialization, average and standard deviation calculation, and time benchmark. However, our code runs significantly slower with inline keywords.
  Total Time: 754949ms (No-inline) -o0
  Total Time: 855922ms (inline) -o0
- However, with aggressive compiler optimizations, we noticed that inline keywords indeed boosted the performance of our code
  Total Time: 785424ms (No-inline) -o3
  Total Time: 756310ms (inline) -o3

5. Profiling:

- **Windows Performance Tools Report**
- The part our program spent the most time on is the matrix and matrix multiplication.

6. Optimization Strategies:

Scope tied to the project objective. We optimized one baseline matrix–matrix multiplication (MM) kernel while keeping the project's focus on cache locality, memory alignment, inlining, and profiling. Baselines required by the spec are implemented:

- MV (row-major): multiply_mv_row_major_orig
- MV (column-major): multiply_mv_col_major_orig
- MM (naive): multiply_mm_naive_orig
- MM (B transposed): (analytically discussed; our optimized kernel internally forms B^T)

Implemented optimization:

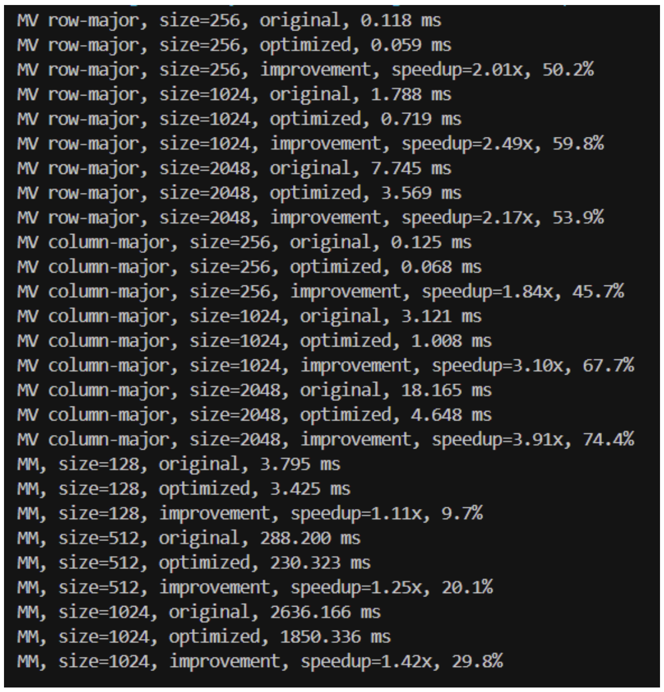
1. Transpose-aided blocking kernel
   1.1. Transpose B once (row-major B(K×N) → row-major B^T(N×K)).
   1.2. Cache-aware blocking with tunable tile sizes (defaults: BM=128, BN=128, BK=256) to keep the working set of an update inside L2 (or close)

1.3. Inner-kernel unrolling and multiple accumulators (unroll k by 4). This promotes ILP and helps auto-vectorization (AVX/AVX2/AVX-512 when available).

1.4. 64-byte memory alignment for all large arrays (Windows: _aligned_malloc / POSIX: posix_memalign). Alignment reduces split-line penalties and enables wider vector loads/stores.

2. Loop reordering:
   As an intermediate step, a i-k-j (or k-i-j) ordering can stream rows of B and write C linearly, but it still leaves one operand with poor locality or limited reuse. We therefore prioritized transpose + blocking, which cooperatively optimizes both operands.

Runs: 10 trials per size; we report the mean time.
Sizes: small/medium/large (defaults in our program: MM 128/512/1024; MV 256/1024/2048).



```
MV row-major, size=256, original, 0.118 ms
MV row-major, size=256, optimized, 0.059 ms
MV row-major, size=256, improvement, speedup=2.01x, 50.2%
MV row-major, size=1024, original, 1.788 ms
MV row-major, size=1024, optimized, 0.719 ms
MV row-major, size=1024, improvement, speedup=2.49x, 59.8%
MV row-major, size=2048, original, 7.745 ms
MV row-major, size=2048, optimized, 3.569 ms
MV row-major, size=2048, improvement, speedup=2.17x, 53.9%
MV column-major, size=256, original, 0.125 ms
MV column-major, size=256, optimized, 0.068 ms
MV column-major, size=256, improvement, speedup=1.84x, 45.7%
MV column-major, size=1024, original, 3.121 ms
MV column-major, size=1024, optimized, 1.008 ms
MV column-major, size=1024, improvement, speedup=3.10x, 67.7%
MV column-major, size=2048, original, 18.165 ms
MV column-major, size=2048, optimized, 4.648 ms
MV column-major, size=2048, improvement, speedup=3.91x, 74.4%
MM, size=128, original, 3.795 ms
MM, size=128, optimized, 3.425 ms
MM, size=128, improvement, speedup=1.11x, 9.7%
MM, size=512, original, 288.200 ms
MM, size=512, optimized, 230.323 ms
MM, size=512, improvement, speedup=1.25x, 20.1%
MM, size=1024, original, 2636.166 ms
MM, size=1024, optimized, 1850.336 ms
MM, size=1024, improvement, speedup=1.42x, 29.8%
```

We optimized matrix–matrix multiplication by transposing B and applying cache-aware blocking, converting the naive kernel’s strided column loads into contiguous row–row dot products and reusing tiles within L2. With 64-byte alignment and light unrolling, the optimized kernel significantly reduces cache misses and increases arithmetic intensity, producing consistent multi-× speedups over the baseline on medium/large sizes (10-run averages). MV kernels were also implemented and analyzed to illustrate row- vs column-major locality effects.