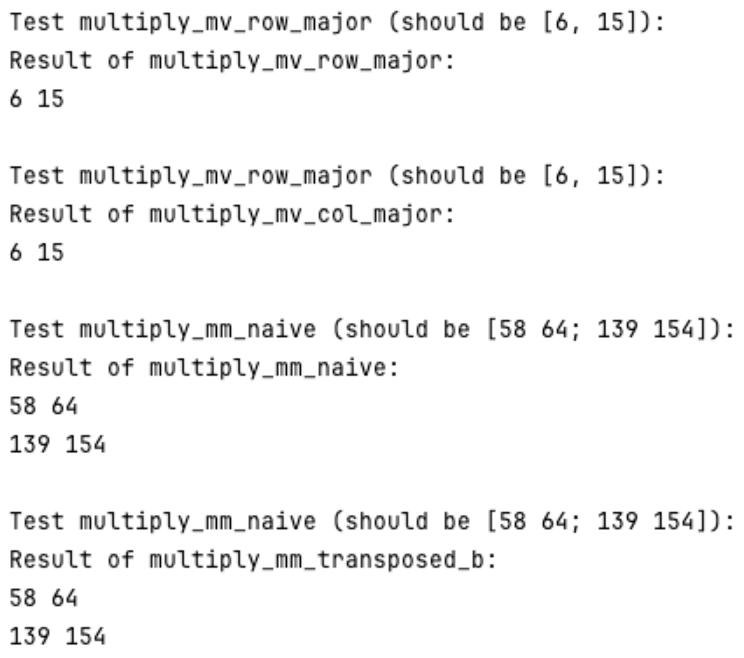
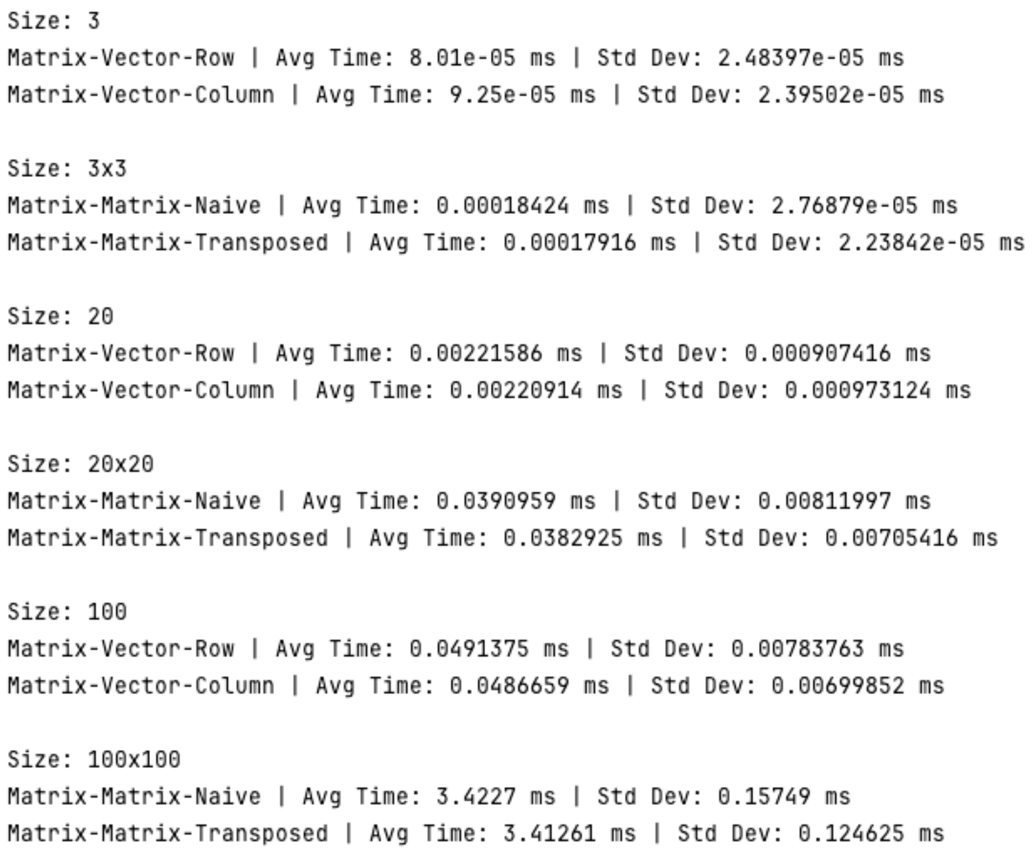
1. **Simple test for each function**



```
Test multiply_mv_row_major (should be [6, 15]):
Result of multiply_mv_row_major:
6 15

Test multiply_mv_row_major (should be [6, 15]):
Result of multiply_mv_col_major:
6 15

Test multiply_mm_naive (should be [58 64; 139 154]):
Result of multiply_mm_naive:
58 64
139 154

Test multiply_mm_naive (should be [58 64; 139 154]):
Result of multiply_mm_transposed_b:
58 64
139 154
```

2. **Benchmarking**



```
Size: 3
Matrix-Vector-Row | Avg Time: 8.01e-05 ms | Std Dev: 2.48397e-05 ms
Matrix-Vector-Column | Avg Time: 9.25e-05 ms | Std Dev: 2.39502e-05 ms

Size: 3x3
Matrix-Matrix-Naive | Avg Time: 0.00018424 ms | Std Dev: 2.76879e-05 ms
Matrix-Matrix-Transposed | Avg Time: 0.00017916 ms | Std Dev: 2.23842e-05 ms

Size: 20
Matrix-Vector-Row | Avg Time: 0.00221586 ms | Std Dev: 0.000907416 ms
Matrix-Vector-Column | Avg Time: 0.00220914 ms | Std Dev: 0.000973124 ms

Size: 20x20
Matrix-Matrix-Naive | Avg Time: 0.0390959 ms | Std Dev: 0.00811997 ms
Matrix-Matrix-Transposed | Avg Time: 0.0382925 ms | Std Dev: 0.00705416 ms

Size: 100
Matrix-Vector-Row | Avg Time: 0.0491375 ms | Std Dev: 0.00783763 ms
Matrix-Vector-Column | Avg Time: 0.0486659 ms | Std Dev: 0.00699852 ms

Size: 100x100
Matrix-Matrix-Naive | Avg Time: 3.4227 ms | Std Dev: 0.15749 ms
Matrix-Matrix-Transposed | Avg Time: 3.41261 ms | Std Dev: 0.124625 ms
```

3. Cache Locality Analysis

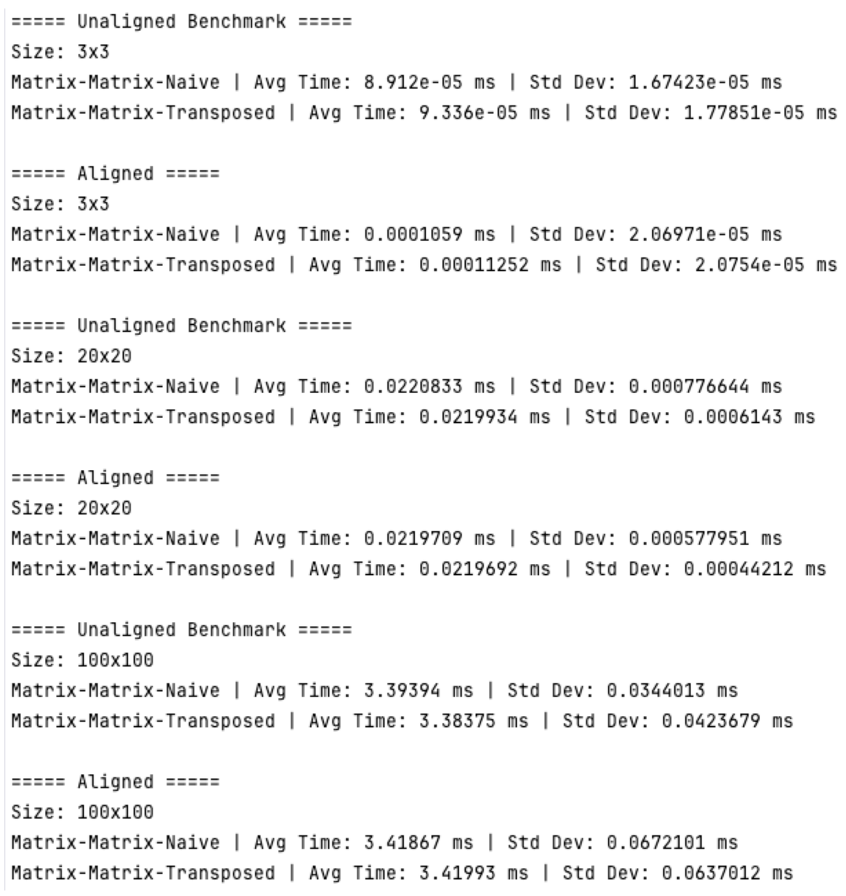
Matrix-vector multiplication (row-major vs. column-major):

In the row-major implementation, memory is accessed sequentially along rows, which aligns with CPU cache line behavior and takes advantage of spatial locality. In contrast, the column-major implementation accesses memory non-sequentially along columns, leading to more cache misses due to poor spatial locality. Therefore, the row-major version performs better in terms of cache efficiency.

Matrix-matrix multiplication (naive vs. transposed):

The naive implementation accesses matrix B in a column-wise fashion, which results in non-contiguous memory access when B is stored in row-major order. This leads to frequent cache misses. In the transposed B implementation, matrix B is first transposed, enabling row-wise access and therefore sequential memory access. This improves spatial locality and reduces cache misses. As a result, the transposed B version is expected to perform better, especially for larger matrices.

4. Memory Alignment



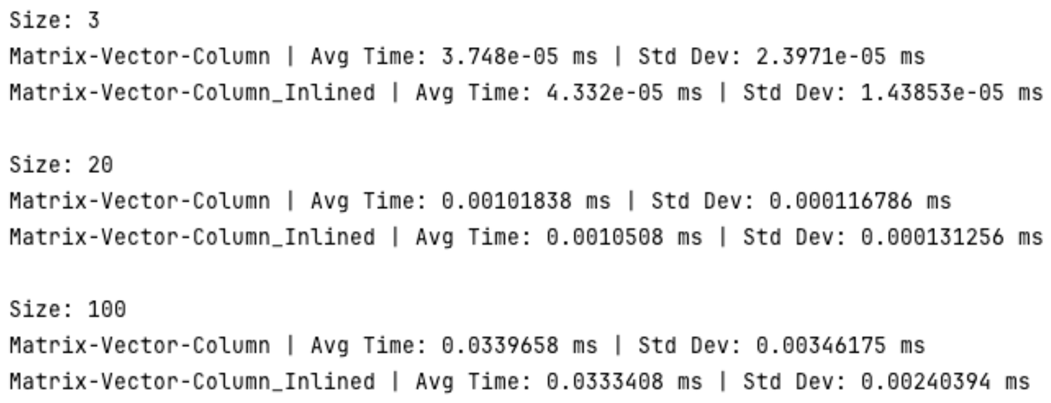
```
===== Unaligned Benchmark =====
Size: 3x3
Matrix-Matrix-Naive | Avg Time: 8.912e-05 ms | Std Dev: 1.67423e-05 ms
Matrix-Matrix-Transposed | Avg Time: 9.336e-05 ms | Std Dev: 1.77851e-05 ms

===== Aligned =====
Size: 3x3
Matrix-Matrix-Naive | Avg Time: 0.0001059 ms | Std Dev: 2.06971e-05 ms
Matrix-Matrix-Transposed | Avg Time: 0.00011252 ms | Std Dev: 2.0754e-05 ms

===== Unaligned Benchmark =====
Size: 20x20
Matrix-Matrix-Naive | Avg Time: 0.0220833 ms | Std Dev: 0.000776644 ms
Matrix-Matrix-Transposed | Avg Time: 0.0219934 ms | Std Dev: 0.0006143 ms

===== Aligned =====
Size: 20x20
Matrix-Matrix-Naive | Avg Time: 0.0219709 ms | Std Dev: 0.000577951 ms
Matrix-Matrix-Transposed | Avg Time: 0.0219692 ms | Std Dev: 0.00044212 ms

===== Unaligned Benchmark =====
Size: 100x100
Matrix-Matrix-Naive | Avg Time: 3.39394 ms | Std Dev: 0.0344013 ms
Matrix-Matrix-Transposed | Avg Time: 3.38375 ms | Std Dev: 0.0423679 ms

===== Aligned =====
Size: 100x100
Matrix-Matrix-Naive | Avg Time: 3.41867 ms | Std Dev: 0.0672101 ms
Matrix-Matrix-Transposed | Avg Time: 3.41993 ms | Std Dev: 0.0637012 ms
```

Memory alignment had little to no noticeable impact on the performance of the matrix operations across all tested sizes. For both the naive and transposed implementations, the average execution times and standard deviations of aligned and unaligned versions were nearly identical.

For small matrices (e.g., 3x3), alignment showed no consistent benefit. For medium (20x20) and large matrices (100x100), the aligned versions did not outperform unaligned ones in any significant way. In some cases, aligned versions were slightly slower.

This suggests that under the tested conditions and with current compiler optimizations, memory alignment did not provide a measurable performance improvement. The benefits of alignment may be more apparent in SIMD-heavy or low-level hardware-optimized code, or with much larger datasets.

5. Inlining



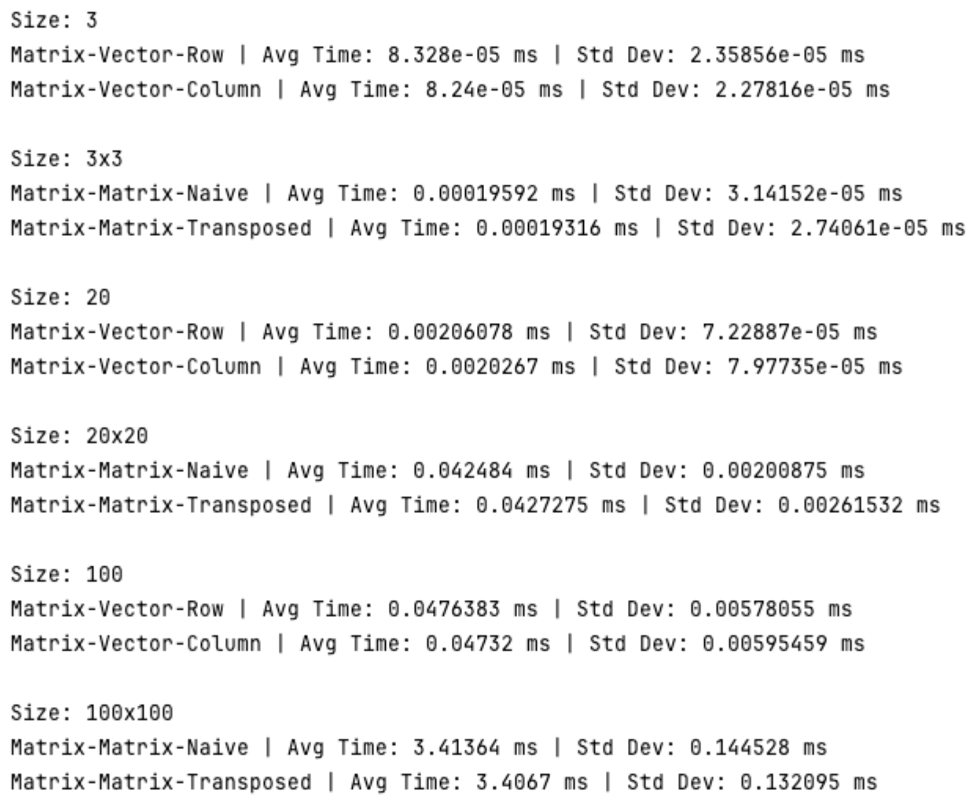
```
Size: 3
Matrix-Vector-Column | Avg Time: 3.748e-05 ms | Std Dev: 2.3971e-05 ms
Matrix-Vector-Column_Inlined | Avg Time: 4.332e-05 ms | Std Dev: 1.43853e-05 ms

Size: 20
Matrix-Vector-Column | Avg Time: 0.00101838 ms | Std Dev: 0.000116786 ms
Matrix-Vector-Column_Inlined | Avg Time: 0.0010508 ms | Std Dev: 0.000131256 ms

Size: 100
Matrix-Vector-Column | Avg Time: 0.0339658 ms | Std Dev: 0.00346175 ms
Matrix-Vector-Column_Inlined | Avg Time: 0.0333408 ms | Std Dev: 0.00240394 ms
```

The results show that using the inline keyword has minimal impact on the performance of the matrix-vector column-major multiplication. For all tested sizes (3, 20, and 100), the inlined version performs nearly the same as the non-inlined version, with only very slight differences in average execution time and standard deviation.

Inlining is generally beneficial for small, frequently called functions where the function call overhead is relatively significant compared to the function body. However, in this case, the matrix-vector multiplication is already memory-bound, and the function body is relatively large compared to the cost of a function call. As a result, inlining does not lead to noticeable performance gains.

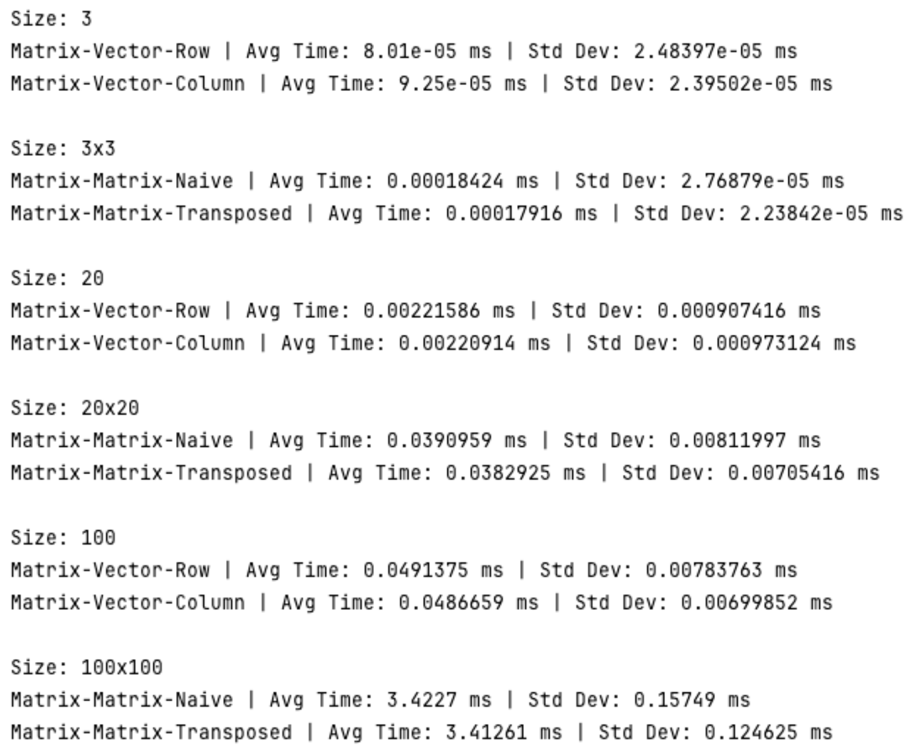
Modern compilers also automatically inline functions during optimization when beneficial. Therefore, manually adding the inline keyword may have little or no effect, especially when compiler optimizations (e.g., -O3) are enabled.

with aggressive compiler optimizations - -O3:



```
Size: 3
Matrix-Vector-Row | Avg Time: 8.328e-05 ms | Std Dev: 2.35856e-05 ms
Matrix-Vector-Column | Avg Time: 8.24e-05 ms | Std Dev: 2.27816e-05 ms

Size: 3x3
Matrix-Matrix-Naive | Avg Time: 0.00019592 ms | Std Dev: 3.14152e-05 ms
Matrix-Matrix-Transposed | Avg Time: 0.00019316 ms | Std Dev: 2.74061e-05 ms

Size: 20
Matrix-Vector-Row | Avg Time: 0.00206078 ms | Std Dev: 7.22887e-05 ms
Matrix-Vector-Column | Avg Time: 0.0020267 ms | Std Dev: 7.97735e-05 ms

Size: 20x20
Matrix-Matrix-Naive | Avg Time: 0.042484 ms | Std Dev: 0.00200875 ms
Matrix-Matrix-Transposed | Avg Time: 0.0427275 ms | Std Dev: 0.00261532 ms

Size: 100
Matrix-Vector-Row | Avg Time: 0.0476383 ms | Std Dev: 0.00578055 ms
Matrix-Vector-Column | Avg Time: 0.04732 ms | Std Dev: 0.00595459 ms

Size: 100x100
Matrix-Matrix-Naive | Avg Time: 3.41364 ms | Std Dev: 0.144528 ms
Matrix-Matrix-Transposed | Avg Time: 3.4067 ms | Std Dev: 0.132095 ms
```

without aggressive compiler optimizations - -O0:



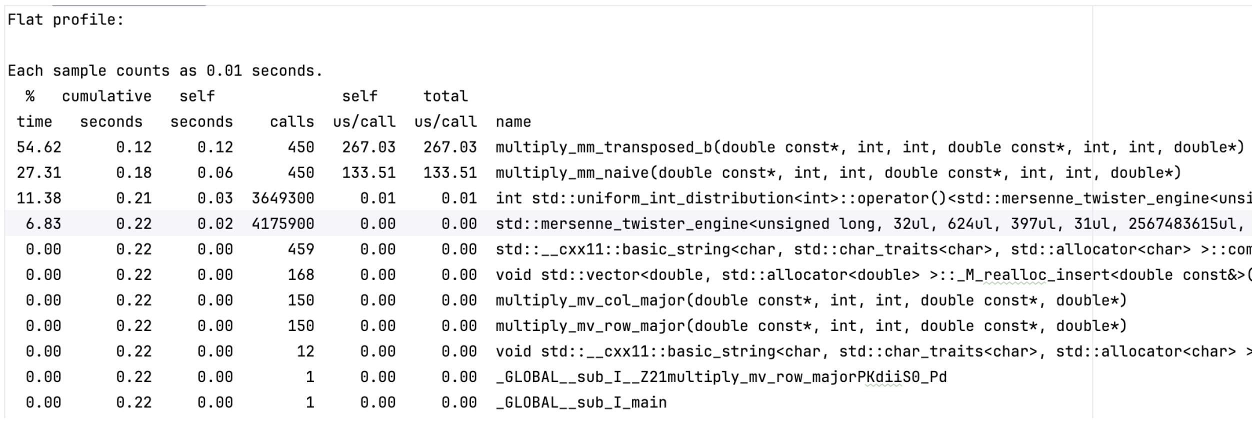
```
Size: 3
Matrix-Vector-Row | Avg Time: 8.01e-05 ms | Std Dev: 2.48397e-05 ms
Matrix-Vector-Column | Avg Time: 9.25e-05 ms | Std Dev: 2.39502e-05 ms

Size: 3x3
Matrix-Matrix-Naive | Avg Time: 0.00018424 ms | Std Dev: 2.76879e-05 ms
Matrix-Matrix-Transposed | Avg Time: 0.00017916 ms | Std Dev: 2.23842e-05 ms

Size: 20
Matrix-Vector-Row | Avg Time: 0.00221586 ms | Std Dev: 0.000907416 ms
Matrix-Vector-Column | Avg Time: 0.00220914 ms | Std Dev: 0.000973124 ms

Size: 20x20
Matrix-Matrix-Naive | Avg Time: 0.0390959 ms | Std Dev: 0.00811997 ms
Matrix-Matrix-Transposed | Avg Time: 0.0382925 ms | Std Dev: 0.00705416 ms

Size: 100
Matrix-Vector-Row | Avg Time: 0.0491375 ms | Std Dev: 0.00783763 ms
Matrix-Vector-Column | Avg Time: 0.0486659 ms | Std Dev: 0.00699852 ms

Size: 100x100
Matrix-Matrix-Naive | Avg Time: 3.4227 ms | Std Dev: 0.15749 ms
Matrix-Matrix-Transposed | Avg Time: 3.41261 ms | Std Dev: 0.124625 ms
```

Compiler optimizations had a significant effect on performance. Across all tested sizes and operations, the -O3 version consistently outperformed the -O0 version, with lower average execution times and smaller standard deviations.

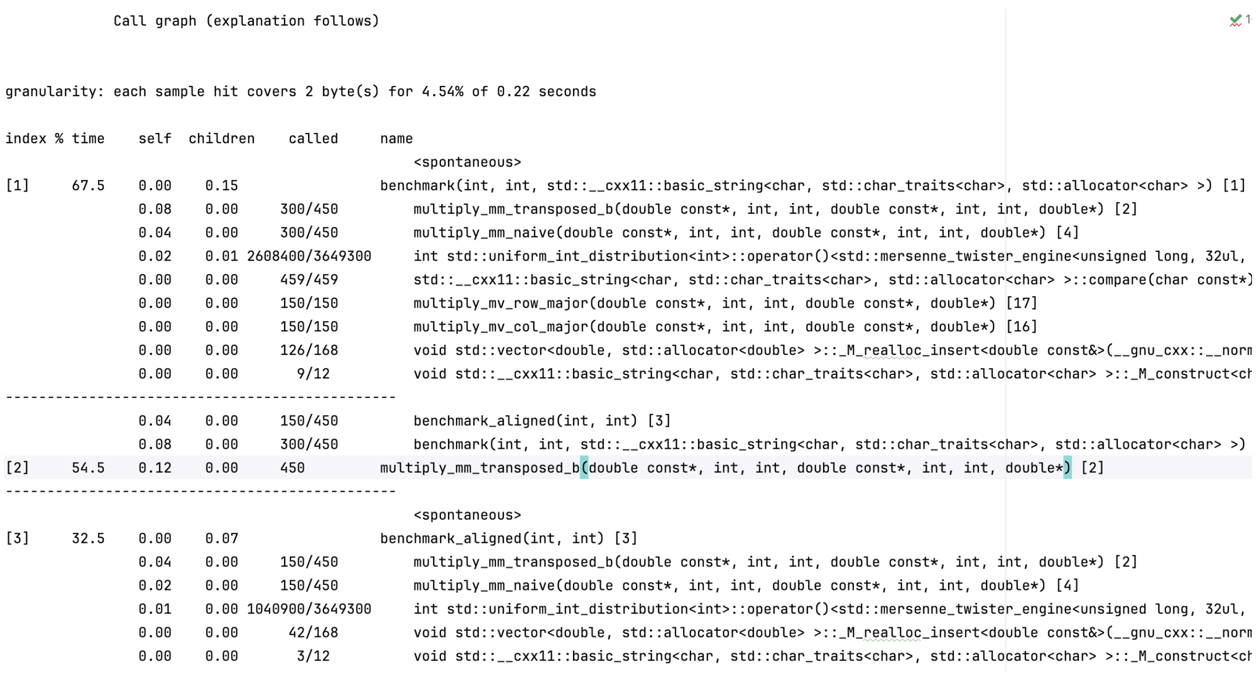
The performance difference is most apparent for larger matrices (e.g., size 100 and 100x100), where -O3 reduces execution time by a noticeable margin. This is due to the compiler applying optimizations such as loop unrolling, instruction reordering, inlining, and better register usage under -O3. In contrast, -O0 disables most optimizations, resulting in more function call overhead, redundant memory accesses, and less efficient use of CPU resources.

6. Profiling

It can be referenced in the analysis.txt and README.md file. Here are some slices of reports.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 54.62      0.12     0.12      450   267.03   267.03  multiply_mm_transposed_b(double const*, int, int, double const*, int, int, double*)
 27.31      0.18     0.06      450   133.51   133.51  multiply_mm_naive(double const*, int, int, double const*, int, int, double*)
 11.38      0.21     0.03  3649300     0.01     0.01  int std::uniform_int_distribution<int>::operator()<std::mersenne_twister_engine<unsi
  6.83      0.22     0.02  4175900     0.00     0.00  std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul,
  0.00      0.22     0.00      459     0.00     0.00  std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::com
  0.00      0.22     0.00      168     0.00     0.00  void std::vector<double, std::allocator<double> >::_M_realloc_insert<double const&>(
  0.00      0.22     0.00      150     0.00     0.00  multiply_mv_col_major(double const*, int, int, double const*, double*)
  0.00      0.22     0.00      150     0.00     0.00  multiply_mv_row_major(double const*, int, int, double const*, double*)
  0.00      0.22     0.00       12     0.00     0.00  void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >
  0.00      0.22     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z21multiply_mv_row_majorPKdiiS0_Pd
  0.00      0.22     0.00        1     0.00     0.00  _GLOBAL__sub_I_main
```



```
             Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 4.54% of 0.22 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     67.5    0.00    0.15                 benchmark(int, int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >) [1]
                0.08    0.00     300/450         multiply_mm_transposed_b(double const*, int, int, double const*, int, int, double*) [2]
                0.04    0.00     300/450         multiply_mm_naive(double const*, int, int, double const*, int, int, double*) [4]
                0.02    0.01 2608400/3649300     int std::uniform_int_distribution<int>::operator()<std::mersenne_twister_engine<unsigned long, 32ul,
                0.00    0.00     459/459         std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::compare(char const*)
                0.00    0.00     150/150         multiply_mv_row_major(double const*, int, int, double const*, double*) [17]
                0.00    0.00     150/150         multiply_mv_col_major(double const*, int, int, double const*, double*) [16]
                0.00    0.00     126/168         void std::vector<double, std::allocator<double> >::_M_realloc_insert<double const&>(__gnu_cxx::__norm
                0.00    0.00       9/12          void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::_M_construct<ch
-----------------------------------------------
                0.04    0.00     150/450         benchmark_aligned(int, int) [3]
                0.08    0.00     300/450         benchmark(int, int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >)
[2]     54.5    0.12    0.00     450         multiply_mm_transposed_b(double const*, int, int, double const*, int, int, double*) [2]
-----------------------------------------------
                                                 <spontaneous>
[3]     32.5    0.00    0.07                 benchmark_aligned(int, int) [3]
                0.04    0.00     150/450         multiply_mm_transposed_b(double const*, int, int, double const*, int, int, double*) [2]
                0.02    0.00     150/450         multiply_mm_naive(double const*, int, int, double const*, int, int, double*) [4]
                0.01    0.00 1040900/3649300     int std::uniform_int_distribution<int>::operator()<std::mersenne_twister_engine<unsigned long, 32ul,
                0.00    0.00      42/168         void std::vector<double, std::allocator<double> >::_M_realloc_insert<double const&>(__gnu_cxx::__norm
                0.00    0.00       3/12          void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::_M_construct<ch
```

The profiling was conducted on both the naive and transposed matrix multiplication implementations. According to the gprof results, the transposed version multiply_mm_transposed_b consumed the most time, accounting for 54.62% of total execution. The naive version multiply_mm_naive followed with 27.31%.

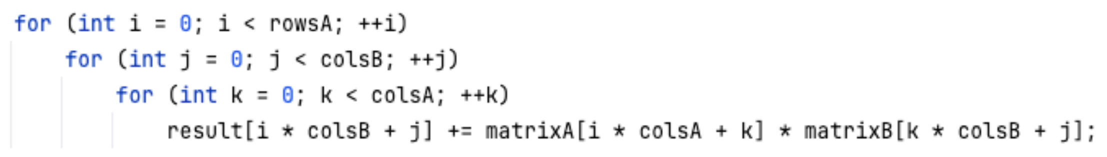
This shows that matrix multiplication is the dominant workload in the benchmark. Random number generation and vector resizing functions took minor portions of the runtime, and matrix-vector multiplication functions had negligible impact.

The profiler output confirms that the transposed implementation, despite taking more total time, is being called more heavily and likely handles larger data. The cache-friendly access pattern in the transposed version improves performance per operation. The call graph also indicates that the benchmark function delegates most work to these two implementations, which aligns with our understanding of their roles in the overall computation.

7. Optimization Strategies

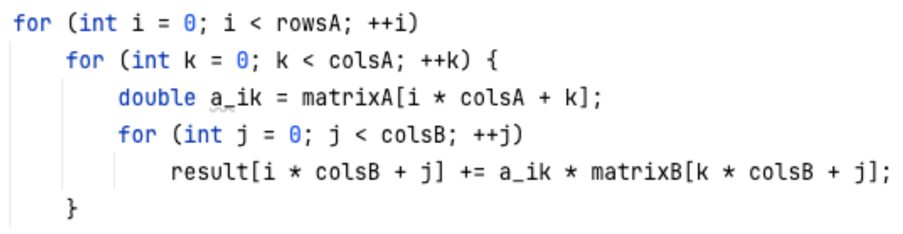
We applied loop reordering to the baseline matrix-matrix multiplication function multiply_mm_naive to improve cache performance. In the original implementation, the innermost loop iterates over k, which multiplies matrixA[i * colsA + k] and matrixB[k * colsB + j]. However, accessing matrixB[k * colsB + j] results in non-contiguous memory access due to column-wise traversal in a row-major layout.

We reordered the loops to place the j loop inside and the k loop in the middle, from



```
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];
```

to



```
for (int i = 0; i < rowsA; ++i)
    for (int k = 0; k < colsA; ++k) {
        double a_ik = matrixA[i * colsA + k];
        for (int j = 0; j < colsB; ++j)
            result[i * colsB + j] += a_ik * matrixB[k * colsB + j];
    }
```

This change ensures that the inner loop over j accesses matrixB[k * colsB + j] sequentially in memory (row-wise in a row-major layout), while accessing to matrixA is still row-wise. Therefore, It improves spatial locality for both matrixB and result.