HW1 Team HW

Jonathan Ran, Amy Wang, Victor Qian, Robert Xing

April 2025

1 Performance Analysis and Optimization

1.1 Benchmarking

Refer to Table 1, 2, 3 below.

1.2 Cache Locality Analysis

The two matrix-vector multiplication functions follow two kinds of access patterns: row-major and column-major. If we assume matrix is stored in row-major order, the row-major function was expected to perform better than the column-major function, since in C++ vectors and arrays are stored in row-major order in memory, so when a value from the matrix/vector is pulled into the cache line, it also pulls in its neighbors in memory, leading to more cache hits when that subsequent data is accessed. When trying to access this data in column-major order, the cache then has to repeatedly pull in non-contiguous memory, leading to more cache misses.

In a similar way, the two matrix-matrix multiplication functions show row-major vs. column-major access patterns. The transposed_B function makes better use of cache locality (spatial locality) by accessing members of B in row-major order. The naive function is forced to access B in column-major order due to the order of how matrix multiplication is computed, but by first transposing B, we are able to transform the data we want to access into row-major order, thus we are able to follow the contiguously stored data in memory and reduce cache misses, all while computing the same result as the naive function.

For the matrix multiplication functions, we see a difference in performance consistent with our expectations. On both large and small matrices, the transposed_B version performed noticeably better than the naive version - the following includes some of the benchmarks we used:

Table 1: Benchmark Results (Debug Build, -O0 -g, 10 runs per test)

Function	**RowsA**	**ColsA**	**ColsB**	**Avg Time (ms)**	**Std Dev (ms)**
multiply_mv_row_major	10	10	1	0.0003	0.0000
multiply_mv_col_major	10	10	1	0.0002	0.0000
multiply_mm_naive	10	10	10	0.0027	0.0001
multiply_mm_transposed_b	10	10	10	0.0027	0.0001
multiply_mm_t_b_noinline	10	10	10	0.0027	0.0001
multiply_mm_optimized	10	10	10	0.0019	0.0001
multiply_mm_opti_noinline	10	10	10	0.0019	0.0001
multiply_mv_row_major	100	100	1	0.0357	0.0015
multiply_mv_col_major	100	100	1	0.0160	0.0001
multiply_mm_naive	100	100	100	3.3485	0.1160
multiply_mm_transposed_b	100	100	100	3.2853	0.0769
multiply_mm_t_b_noinline	100	100	100	3.3300	0.0940
multiply_mm_optimized	100	100	100	1.5857	0.0831
multiply_mm_opti_noinline	100	100	100	1.5885	0.0545
multiply_mv_row_major	500	500	1	0.8596	0.0160
multiply_mv_col_major	500	500	1	0.3878	0.0211
multiply_mm_naive	500	500	500	431.8305	5.9091
multiply_mm_transposed_b	500	500	500	429.8905	4.1479
multiply_mm_t_b_noinline	500	500	500	427.2438	2.6250
multiply_mm_optimized	500	500	500	192.2985	0.5645
multiply_mm_opti_noinline	500	500	500	191.6056	0.5545
multiply_mv_row_major	1000	1000	1	3.5955	0.1583
multiply_mv_col_major	1000	1000	1	1.5142	0.0907
multiply_mm_naive	1000	1000	1000	3757.3031	176.4267
multiply_mm_transposed_b	1000	1000	1000	3641.0933	93.8409
multiply_mm_t_b_noinline	1000	1000	1000	3604.0539	83.7356
multiply_mm_optimized	1000	1000	1000	1568.6388	17.5043
multiply_mm_opti_noinline	1000	1000	1000	1591.7105	50.6970
multiply_mv_row_major	200	50	1	0.0322	0.0039
multiply_mv_col_major	200	50	1	0.0151	0.0001
multiply_mm_naive	200	50	100	3.4844	0.3006
multiply_mm_transposed_b	200	50	100	3.1823	0.0984
multiply_mm_t_b_noinline	200	50	100	3.0881	0.0239
multiply_mm_optimized	200	50	100	1.5786	0.0214
multiply_mm_opti_noinline	200	50	100	1.5718	0.0159
multiply_mv_row_major	50	300	1	0.0539	0.0060
multiply_mv_col_major	50	300	1	0.0234	0.0003
multiply_mm_naive	50	300	80	4.3242	0.0940
multiply_mm_transposed_b	50	300	80	4.2476	0.0638
multiply_mm_t_b_noinline	50	300	80	4.2771	0.0951
multiply_mm_optimized	50	300	80	1.8409	0.0183
multiply_mm_opti_noinline	50	300	80	1.8413	0.0199

Table 2: Benchmark Results (Profile Build, -O2 -g -pg, 10 runs per test)

Function	**RowsA**	**ColsA**	**ColsB**	**Avg Time (ms)**	**Std Dev (ms)**
multiply_mv_row_major	10	10	1	0.0001	0.0000
multiply_mv_col_major	10	10	1	0.0001	0.0000
multiply_mm_naive	10	10	10	0.0005	0.0000
multiply_mm_transposed_b	10	10	10	0.0003	0.0000
multiply_mm_t_b_noinline	10	10	10	0.0003	0.0001
multiply_mm_optimized	10	10	10	0.0004	0.0001
multiply_mm_opti_noinline	10	10	10	0.0004	0.0001
multiply_mv_row_major	100	100	1	0.0081	0.0001
multiply_mv_col_major	100	100	1	0.0015	0.0001
multiply_mm_naive	100	100	100	0.7886	0.0154
multiply_mm_transposed_b	100	100	100	0.4232	0.0075
multiply_mm_t_b_noinline	100	100	100	0.4253	0.0180
multiply_mm_optimized	100	100	100	0.1484	0.0011
multiply_mm_opti_noinline	100	100	100	0.1487	0.0008
multiply_mv_row_major	500	500	1	0.2630	0.0036
multiply_mv_col_major	500	500	1	0.0366	0.0003
multiply_mm_naive	500	500	500	125.0937	0.8307
multiply_mm_transposed_b	500	500	500	87.0432	4.9377
multiply_mm_t_b_noinline	500	500	500	84.4113	1.1973
multiply_mm_optimized	500	500	500	17.0234	1.5674
multiply_mm_opti_noinline	500	500	500	16.1949	0.2501
multiply_mv_row_major	1000	1000	1	1.0843	0.0298
multiply_mv_col_major	1000	1000	1	0.1398	0.0012
multiply_mm_naive	1000	1000	1000	1078.4031	3.6118
multiply_mm_transposed_b	1000	1000	1000	777.5210	18.8140
multiply_mm_t_b_noinline	1000	1000	1000	804.8560	30.1812
multiply_mm_optimized	1000	1000	1000	132.3471	4.9753
multiply_mm_opti_noinline	1000	1000	1000	135.5599	6.8742
multiply_mv_row_major	200	50	1	0.0068	0.0003
multiply_mv_col_major	200	50	1	0.0016	0.0001
multiply_mm_naive	200	50	100	0.5126	0.0137
multiply_mm_transposed_b	200	50	100	0.2964	0.0052
multiply_mm_t_b_noinline	200	50	100	0.2990	0.0100
multiply_mm_optimized	200	50	100	0.1411	0.0042
multiply_mm_opti_noinline	200	50	100	0.1408	0.0049
multiply_mv_row_major	50	300	1	0.0176	0.0059
multiply_mv_col_major	50	300	1	0.0029	0.0002
multiply_mm_naive	50	300	80	1.1067	0.0283
multiply_mm_transposed_b	50	300	80	0.6710	0.0162
multiply_mm_t_b_noinline	50	300	80	0.6714	0.0147
multiply_mm_optimized	50	300	80	0.1873	0.0070
multiply_mm_opti_noinline	50	300	80	0.1864	0.0062

Table 3: Benchmark Results (Release Build, -O3, 10 runs per test)

Function	**RowsA**	**ColsA**	**ColsB**	**Avg Time (ms)**	**Std Dev (ms)**
multiply_mv_row_major	10	10	1	0.0001	0.0001
multiply_mv_col_major	10	10	1	0.0001	0.0000
multiply_mm_naive	10	10	10	0.0012	0.0001
multiply_mm_transposed_b	10	10	10	0.0006	0.0001
multiply_mm_t_b_noinline	10	10	10	0.0006	0.0001
multiply_mm_optimized	10	10	10	0.0008	0.0002
multiply_mm_opti_noinline	10	10	10	0.0007	0.0001
multiply_mv_row_major	100	100	1	0.0138	0.0003
multiply_mv_col_major	100	100	1	0.0027	0.0002
multiply_mm_naive	100	100	100	1.1875	0.0839
multiply_mm_transposed_b	100	100	100	0.5953	0.0140
multiply_mm_t_b_noinline	100	100	100	0.5469	0.0062
multiply_mm_optimized	100	100	100	0.1942	0.0008
multiply_mm_opti_noinline	100	100	100	0.1867	0.0061
multiply_mv_row_major	500	500	1	0.3176	0.0024
multiply_mv_col_major	500	500	1	0.0445	0.0002
multiply_mm_naive	500	500	500	125.6976	1.3677
multiply_mm_transposed_b	500	500	500	83.7917	0.5218
multiply_mm_t_b_noinline	500	500	500	90.8010	13.4104
multiply_mm_optimized	500	500	500	15.8863	0.2130
multiply_mm_opti_noinline	500	500	500	15.7589	0.0498
multiply_mv_row_major	1000	1000	1	1.1204	0.0472
multiply_mv_col_major	1000	1000	1	0.1439	0.0012
multiply_mm_naive	1000	1000	1000	1087.1628	8.5991
multiply_mm_transposed_b	1000	1000	1000	766.8323	9.0496
multiply_mm_t_b_noinline	1000	1000	1000	763.5551	1.5490
multiply_mm_optimized	1000	1000	1000	125.2984	1.1272
multiply_mm_opti_noinline	1000	1000	1000	124.4580	0.4070
multiply_mv_row_major	200	50	1	0.0068	0.0002
multiply_mv_col_major	200	50	1	0.0016	0.0002
multiply_mm_naive	200	50	100	0.4957	0.0120
multiply_mm_transposed_b	200	50	100	0.3129	0.0123
multiply_mm_t_b_noinline	200	50	100	0.3060	0.0188
multiply_mm_optimized	200	50	100	0.1437	0.0088
multiply_mm_opti_noinline	200	50	100	0.1395	0.0078
multiply_mv_row_major	50	300	1	0.0158	0.0025
multiply_mv_col_major	50	300	1	0.0029	0.0002
multiply_mm_naive	50	300	80	1.1049	0.0199
multiply_mm_transposed_b	50	300	80	0.6840	0.0483
multiply_mm_t_b_noinline	50	300	80	0.6521	0.0115
multiply_mm_optimized	50	300	80	0.1828	0.0128
multiply_mm_opti_noinline	50	300	80	0.1846	0.0065

As can be seen in the tables above, for the matrix-vector multiplication functions, we found that the column-major version performed better. This is due to our assumption that the matrix is stored in column-major order. In this case, the input matrices follow a memory layout where the columns are contiguous, so we are still able to take advantage of spatial locality by accessing columns sequentially as if they were rows. Essentially, our data access pattern is the same, we instead just interpret the data differently. Furthermore, we are able to take advantage of temporal locality, as vector[i] is accessed repeatedly in the inner loop for the same i.

1.3 Memory Alignment

It is inconclusive whether memory alignment (to a 64-bit boundary, using a custom allocator in order to retain `vector` usage) provided a benefit compared to the base approach when executed under the `-O3` compiler flag. Data for larger dimensions is as follows. Note that the performance differences are small compared to the deviations in performance.

Matrix Multiplication 500×500

Function	Aligned (ms)	Unaligned (ms)	Aligned StdDev	Unaligned StdDev
multiply_mm_naive	107.2347	109.5766	0.5782	6.2465
multiply_mm_optimized	13.9766	13.9001	0.0742	0.3492
multiply_mm_transposed_b	66.6385	66.7389	0.3419	0.3529

Matrix Multiplication 1000×1000

Function	Aligned (ms)	Unaligned (ms)	Aligned StdDev	Unaligned StdDev
multiply_mm_naive	928.2855	930.1535	3.3217	2.8221
multiply_mm_optimized	111.0750	111.1938	0.8867	0.7822
multiply_mm_transposed_b	611.8406	611.4092	1.6306	1.4030

1.4 Inlining

There's dramatic differences between the Debug build (-O0) and the optimized builds (Profile -O2, Release -O3). Enabling optimizations (-O2 or -O3) yields roughly a 3-4x speedup for the naive algorithm and over a 10-12x speedup for the optimized algorithm compared to the unoptimized (-O0) debug build. The difference between the Profile (-O2) and Release (-O3) builds is generally much smaller than the jump from -O0. For the optimized functions, -O3 often provides a small but consistent edge over -O2. -O3 enables more aggressive optimizations that might yield better performance, but can sometimes increase code size or compilation time, and rarely result in slightly slower code if heuristics guess wrong.

If we consider the Release (-O3) build (where the compiler aggressively optimizes), we see that explicitly inlining has minimal impact on performance, and sometimes not inlining is slightly faster, although the difference is small and well within the standard deviation. Similarly for the Debug (-O0) build (where the compiler performs few transformations), the small variations between inlining vs. not inlining observed fall within typical measurement noise and standard deviations. For these specific, computationally intensive matrix multiplication loops, preventing the compiler from inlining the core calculation function did not negatively impact performance and sometimes yielded a marginal benefit in the -O3 build.

Inlining is likely beneficial for small functions like getters, setters, or basic calculations. It eliminates call overhead for functions that are called frequently inside inner loops. Inlining is not helpful for large functions as it significantly increases the overall code size; the executed code might not fit in the cache, which causes cache misses.

1.5 Profiling

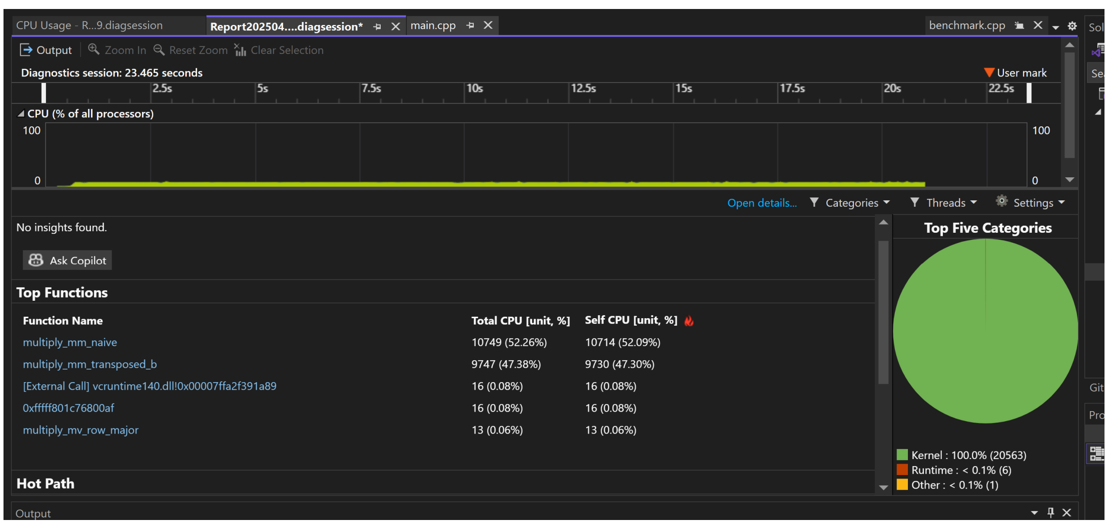


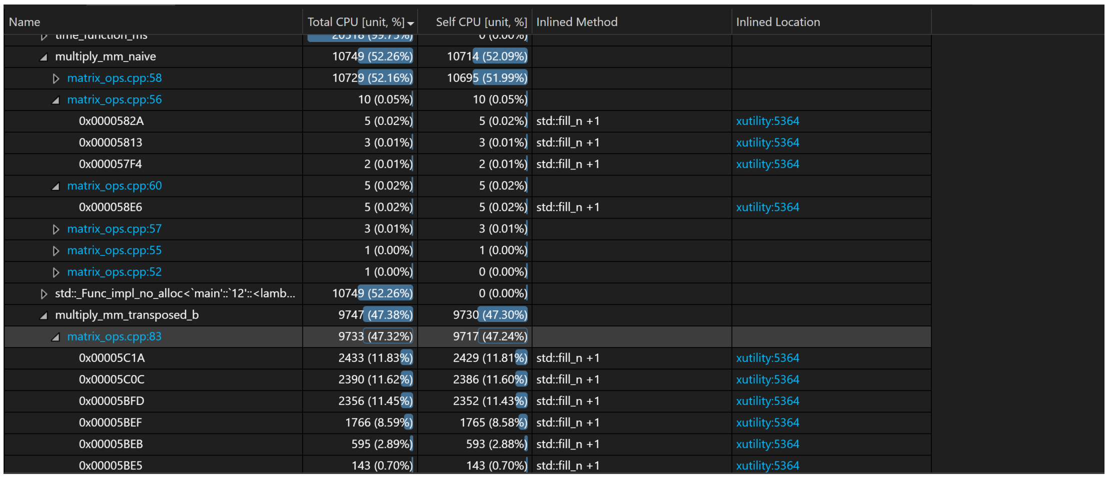
Figure 1: top functions

By profiling both the naive function and the transposed function, we can see that the naive function is significantly slower than the transposed version. It's due to the fact that the way we access matrix B will create a lot of cache misses on the line 58

```
sum += matrixA[i*colsA + k] * matrixB[k*colsB + j];.
```

To access the matrix B, we need to jump around in memory for each iteration of k, because we're traversing a column of matrixB (which is stored row-major).

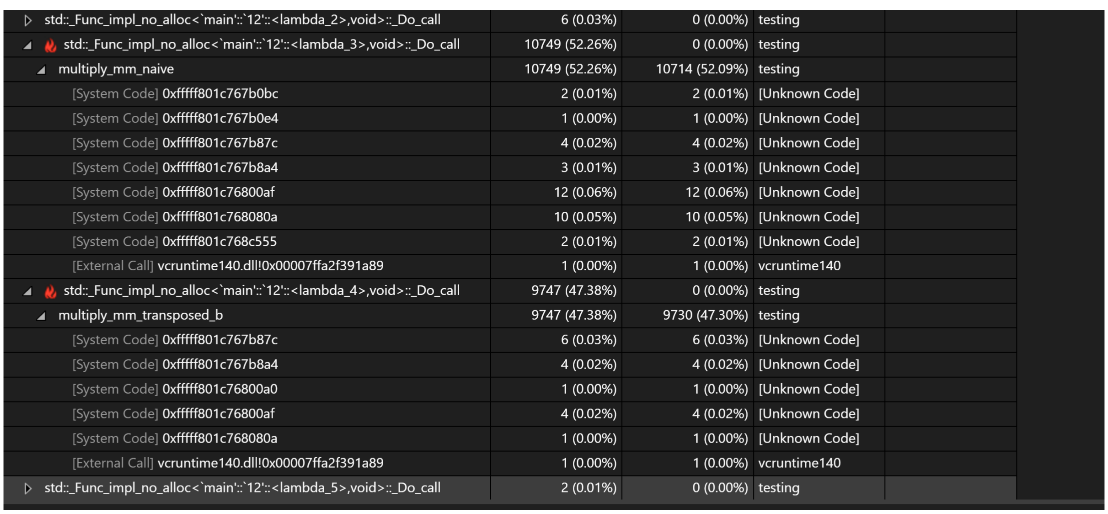
This causes cache lines to be loaded and discarded frequently.



Name	Total CPU [unit, %]	Self CPU [unit, %]	Inlined Method	Inlined Location
multiply_mm_naive	10749 (52.26%)	10714 (52.09%)		
matrix_ops.cpp:58	10729 (52.16%)	10695 (51.99%)		
matrix_ops.cpp:56	10 (0.05%)	10 (0.05%)		
0x0000582A	5 (0.02%)	5 (0.02%)	std::fill_n +1	xutility:5364
0x00005813	3 (0.01%)	3 (0.01%)	std::fill_n +1	xutility:5364
0x000057F4	2 (0.01%)	2 (0.01%)	std::fill_n +1	xutility:5364
matrix_ops.cpp:60	5 (0.02%)	5 (0.02%)		
0x000058E6	5 (0.02%)	5 (0.02%)	std::fill_n +1	xutility:5364
matrix_ops.cpp:57	3 (0.01%)	3 (0.01%)		
matrix_ops.cpp:55	1 (0.00%)	1 (0.00%)		
matrix_ops.cpp:52	1 (0.00%)	0 (0.00%)		
std::_Func_impl_no_alloc<`main'::`12'::<lamb...	10749 (52.26%)	0 (0.00%)		
multiply_mm_transposed_b	9747 (47.38%)	9730 (47.30%)		
matrix_ops.cpp:83	9733 (47.32%)	9717 (47.24%)		
0x00005C1A	2433 (11.83%)	2429 (11.81%)	std::fill_n +1	xutility:5364
0x00005C0C	2390 (11.62%)	2386 (11.60%)	std::fill_n +1	xutility:5364
0x00005BFD	2356 (11.45%)	2352 (11.43%)	std::fill_n +1	xutility:5364
0x00005BEF	1766 (8.59%)	1765 (8.58%)	std::fill_n +1	xutility:5364
0x00005BEB	595 (2.89%)	593 (2.88%)	std::fill_n +1	xutility:5364
0x00005BE5	143 (0.70%)	143 (0.70%)	std::fill_n +1	xutility:5364

Figure 2: cpu time breakdown

The CPU time breakdown also gives us the same conclusion as there's more cpu time spent on the naive version.



std::_Func_impl_no_alloc<`main'::`12'::<lambda_2>,void>::_Do_call	6 (0.03%)	0 (0.00%)	testing
std::_Func_impl_no_alloc<`main'::`12'::<lambda_3>,void>::_Do_call	10749 (52.26%)	0 (0.00%)	testing
multiply_mm_naive	10749 (52.26%)	10714 (52.09%)	testing
[System Code] 0xfffff801c767b0bc	2 (0.01%)	2 (0.01%)	[Unknown Code]
[System Code] 0xfffff801c767b0e4	1 (0.00%)	1 (0.00%)	[Unknown Code]
[System Code] 0xfffff801c767b87c	4 (0.02%)	4 (0.02%)	[Unknown Code]
[System Code] 0xfffff801c767b8a4	3 (0.01%)	3 (0.01%)	[Unknown Code]
[System Code] 0xfffff801c76800af	12 (0.06%)	12 (0.06%)	[Unknown Code]
[System Code] 0xfffff801c768080a	10 (0.05%)	10 (0.05%)	[Unknown Code]
[System Code] 0xfffff801c768c555	2 (0.01%)	2 (0.01%)	[Unknown Code]
[External Call] vcruntime140.dll!0x00007ffa2f391a89	1 (0.00%)	1 (0.00%)	vcruntime140
std::_Func_impl_no_alloc<`main'::`12'::<lambda_4>,void>::_Do_call	9747 (47.38%)	0 (0.00%)	testing
multiply_mm_transposed_b	9747 (47.38%)	9730 (47.30%)	testing
[System Code] 0xfffff801c767b87c	6 (0.03%)	6 (0.03%)	[Unknown Code]
[System Code] 0xfffff801c767b8a4	4 (0.02%)	4 (0.02%)	[Unknown Code]
[System Code] 0xfffff801c76800a0	1 (0.00%)	1 (0.00%)	[Unknown Code]
[System Code] 0xfffff801c76800af	4 (0.02%)	4 (0.02%)	[Unknown Code]
[System Code] 0xfffff801c768080a	1 (0.00%)	1 (0.00%)	[Unknown Code]
[External Call] vcruntime140.dll!0x00007ffa2f391a89	1 (0.00%)	1 (0.00%)	vcruntime140
std::_Func_impl_no_alloc<`main'::`12'::<lambda_5>,void>::_Do_call	2 (0.01%)	0 (0.00%)	testing

Figure 3: call tree

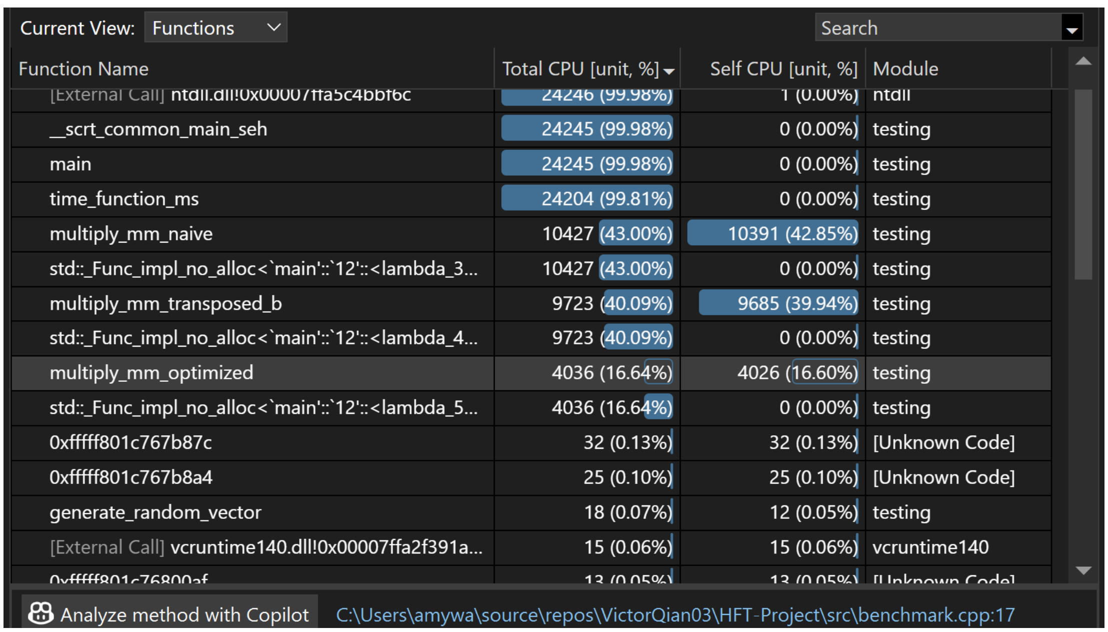
Taking a closer look at the cpu time breakdown, transposed version having lower execution times and fewer sample counts, that supports the idea that accessing a transposed (and thus contiguous) version of matrix B is more efficient.

1.6 Optimizing Strategies

Based on our analysis, we implemented a blocking optimization to improve cache locality and reuse in our matrix multiplication function. In the optimized version, the matrices are partitioned into smaller sub-blocks of size 64, which are chosen to fit better into the processor cache.

We also implemented loop reordering. Within each block, the loops are reordered to iterate in an i, k, j order. This reordering allows each element from matrix A (accessed as a_{ik}) to be loaded once and then reused across the innermost loop that iterates over the j dimension. By doing so, the algorithm minimizes the number of times data must be loaded from main memory, thereby reducing cache misses. Moreover, the innermost loop accesses contiguous memory locations in matrix B and updates contiguous regions of the result matrix, which further enhances spatial locality. This combination of blocking and loop reordering significantly speeds up the matrix multiplication by leveraging the memory hierarchy more effectively.

The following is our profiling for the optimized function.



Current View: Functions   Search

Function Name	Total CPU [unit, %]	Self CPU [unit, %]	Module
[External Call] ntdll.dll!0x00007ffa5c4bbf6c	24246 (99.98%)	1 (0.00%)	ntdll
__scrt_common_main_seh	24245 (99.98%)	0 (0.00%)	testing
main	24245 (99.98%)	0 (0.00%)	testing
time_function_ms	24204 (99.81%)	0 (0.00%)	testing
multiply_mm_naive	10427 (43.00%)	10391 (42.85%)	testing
std::_Func_impl_no_alloc<`main'::`12'::<lambda_3...	10427 (43.00%)	0 (0.00%)	testing
multiply_mm_transposed_b	9723 (40.09%)	9685 (39.94%)	testing
std::_Func_impl_no_alloc<`main'::`12'::<lambda_4...	9723 (40.09%)	0 (0.00%)	testing
multiply_mm_optimized	4036 (16.64%)	4026 (16.60%)	testing
std::_Func_impl_no_alloc<`main'::`12'::<lambda_5...	4036 (16.64%)	0 (0.00%)	testing
0xfffff801c767b87c	32 (0.13%)	32 (0.13%)	[Unknown Code]
0xfffff801c767b8a4	25 (0.10%)	25 (0.10%)	[Unknown Code]
generate_random_vector	18 (0.07%)	12 (0.05%)	testing
[External Call] vcruntime140.dll!0x00007ffa2f391a...	15 (0.06%)	15 (0.06%)	vcruntime140
0xfffff801c76800af	13 (0.05%)	13 (0.05%)	[Unknown Code]

Analyze method with Copilot   C:\Users\amywa\source\repos\VictorQian03\HFT-Project\src\benchmark.cpp:17

Figure 4: Optimized Version cpu time