1. Benchmarking

We use `std::chrono` and framework below to measure the execution time of each baseline function.

```
start = std::chrono::high_resolution_clock::now();
// Tested Function
end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double, std::milli> duration_col_major = end - start;
```

Then we test the baseline functions with various matrix and vector sizes 5 times each and calculate the average execution time and standard deviation.

```
enum class MatrixSize {
    TINY = 2,
    TINT2 = 4,
    SMALL1 = 64,
    SMALL2 = 128,
    MEDIUM = 256,
    MEDIUM2 = 512,
    LARGE = 1024,
    LARGE2 = 2048,
    ENORMOUS = 4096,
};
```

Which means that we test the multiplication of 2x2, 4x4,..., 4096x4096 matrices and corresponding vectors and matrices.

Below are the table of our benchmarking results.

Function	Size	Average Execution Time	Standard Deviation
multiply_mv_row_major 	2x2	0.0001 ms	0.0000 ms
	4x4	0.0002 ms	0.0000 ms
	64x64	0.0095 ms	0.0001 ms
	128x128	0.0375 ms	0.0001 ms
	256x256	0.1532 ms	0.0001 ms
	512x512	0.6206 ms	0.0110 ms

Function	Size	Average Execution Time	Standard Deviation
	1024x1024	2.4983 ms	0.0549 ms
	2048x2048	9.9698 ms	0.0767 ms
	4096x4096	39.6474 ms	0.1309 ms
multiply_mv_col_major 	2x2	0.0002 ms	0.0001 ms
	4x4	0.0002 ms	0.0000 ms
	64x64	0.0113 ms	0.0020 ms
	128x128	0.0416 ms	0.0017 ms
	256x256	0.1831 ms	0.0211 ms
	512x512	0.7451 ms	0.0636 ms
	1024x1024	2.8106 ms	0.0330 ms
	2048x2048	18.6358 ms	0.5270 ms
	4096x4096	74.1854 ms	2.5556 ms
multiply_mm_naive 	2x2	0.0001 ms	0.0000 ms
	4x4	0.0003 ms	0.0000 ms
	64x64	0.6235 ms	0.0119 ms
	128x128	5.0767 ms	0.1116 ms
	256x256	42.5578 ms	0.4478 ms
	512x512	408.4492 ms	6.4710 ms
	1024x1024	3288.0140 ms	70.1005 ms
	2048x2048	36751.3400 ms	1114.6630 ms
	4096x4096	291757.0000 ms	3700.2682 ms
multiply_mm_transposed_b 	2x2	0.0002 ms	0.0001 ms
	4x4	0.0004 ms	0.0001 ms
	64x64	0.6012 ms	0.0145 ms
	128x128	4.8967 ms	0.0469 ms

Function	Size	Average Execution Time	Standard Deviation
	256x256	39.2244 ms	0.2317 ms
	512x512	320.4752 ms	1.0389 ms
	1024x1024	2595.6020 ms	10.3757 ms
	2048x2048	34871.5400 ms	252.2200 ms
	4096x4096	274581.0000 ms	4116.7000 ms

2. Cache Locality Analysis On common CPUs (64B cache lines), the Row-Major version is usually faster because it accesses the matrix in a sequential streaming manner (high hit rate and hardware prefetcher friendly); while the Column-Major version uses large strides in this loop order and has poor spatial locality, especially when the rows are large.

Naive (B is row-major, not transposed) When computing C[i][j] = Σ_k A[i][k] * B[k][j], the inner k increment will access B with a stride length = colsB, resulting in large strides across rows and poor spatial locality.

Transposing B (using B^T row-major order) B[k][j] = (B^T)[j][k]. If we fetch a row using the row pointer b_row = &B^T[j][0], and then sequentially access b_row[k] using the inner k, we achieve a sequential streaming read of the entire row of B^T, significantly improving cache hit rate and forming a "double-stream" dot product with the sequential access of A[i][:].

We compare the execution time between two ways multipling the matrices, the result below clearly indicate what we find.

Function	Size	Average Execution Time	Standard Deviation
multiply_mm_naive 	2×2	0.0001 ms	0.0000 ms
	4×4	0.0003 ms	0.0000 ms
	64x64	0.6235 ms	0.0119 ms
	128x128	5.0767 ms	0.1116 ms
	256x256	42.5578 ms	0.4478 ms
	512x512	408.4492 ms	6.4710 ms

Function	Size	Average Execution Time	Standard Deviation
	1024x1024	3288.0140 ms	70.1005 ms
	2048x2048	36751.3400 ms	1114.6630 ms
	4096x4096	291757.0000 ms	3700.2682 ms
multiply_mm_transposed_b 	2x2	0.0002 ms	0.0001 ms
	4x4	0.0004 ms	0.0001 ms
	64x64	0.6012 ms	0.0145 ms
	128x128	4.8967 ms	0.0469 ms
	256x256	39.2244 ms	0.2317 ms
	512x512	320.4752 ms	1.0389 ms
	1024x1024	2595.6020 ms	10.3757 ms
	2048x2048	34871.5400 ms	252.2200 ms
	4096x4096	274581.0000 ms	4116.7000 ms

3. Memory Alignment

Since aligned data fits within a single cache line and matches the CPU's natural word boundaries, it avoids split accesses, improves cache usage, and enables faster load/store operations. We used `posix_memalign` to makte the matrices aligned on a 64-byte boundary:

```
double *matrix_row;
posix_memalign((void **)&matrix_row, 64, MATRIXSIZEROW * MATRIXSIZECOL *
sizeof(double));
```

Performance results for baseline and 64-byte aligned matrices, obtained on an Apple M1 CPU with * -O3 optimization, are listed below:

Function	Size	Baseline	Alignment
multiply_mv_row_major 	2x2	4.2e-05 ms	0 ms
	4x4	4.1e-05 ms	4.1e-05 ms

Function	Size	Baseline	Alignment
	64x64	0.001625 ms	0.001583 ms
	128x128	0.012333 ms	0.008375 ms
	256x256	0.046334 ms	0.036667 ms
	512x512	0.217584 ms	0.20325 ms
	1024x1024	1.35088 ms	1.03171 ms
	2048x2048	5.81337 ms	5.85262 ms
	4096x4096	58.7503 ms	22.726 ms
multiply_mv_col_major 	2x2	4.1e-05 ms	0 ms
	4x4	4.1e-05 ms	4.2e-05 ms
	64x64	0.002833 ms	0.002917 ms
	128x128	0.026375 ms	0.015458 ms
	256x256	0.109209 ms	0.086417 ms
	512x512	0.41 ms	0.368292 ms
	1024x1024	2.31646 ms	2.06183 ms
	2048x2048	14.2085 ms	15.0496 ms
	4096x4096	68.2399 ms	66.2803 ms
multiply_mm_naive 	2x2	4.2e-05 ms	0.000166 ms
	4x4	0.000125 ms	8.3e-05 ms
	64x64	0.256625 ms	0.172542 ms
	128x128	2.72613 ms	1.90821 ms
	256x256	21.1951 ms	18.4953 ms
	512x512	165.245 ms	162.63 ms
	1024x1024	1573.42 ms	1713.18 ms
	2048x2048	231686 ms	192517 ms
	4096x4096	3.33726e+06 ms	3.09257e+06 ms

Function	Size	Baseline	Alignment
multiply_mm_transposed_b 	2x2	8.3e-05 ms	8.3e-05 ms
	4x4	4.1e-05 ms	0.000125 ms
	64x64	0.220167 ms	0.156 ms
	128x128	2.12433 ms	1.61696 ms
	256x256	16.2154 ms	15.4394 ms
	512x512	147.497 ms	149.695 ms
	1024x1024	1340.64 ms	1936.79 ms
	2048x2048	24421.6 ms	24516.4 ms
	4096x4096	253332 ms	248731 ms

The result shows that in most cases, memory alignment improves performance.

4. Inlining

In our code, we don't use small, frequently called helper functions, thus we skip the experiment with the use of the inline keyword.

Then, we use two ways to experiment on compiling the code with and without aggressive compiler optimizations.

```
g++ -O0 -g Test_Program.cpp Original_Linear_Operation.cpp -o
Test_Program.exe
```

```
g++ -O3 -g Test_Program.cpp Original_Linear_Operation.cpp -o
Test_Program.exe
```

Or change the `args` in `tasks.json` in vscode.

```
"args": [
    "-fdiagnostics-color=always",
    "-fopenmp",
    "-O3",
    "-g",
    "${fileDirname}\\*.cpp",
```

```
        "-o",
        "${fileDirname}\\${fileBasenameNoExtension}.exe"
    ],
```

```
    "args": [
        "-fdiagnostics-color=always",
        "-fopenmp",
        "-O3",
        "-g",
        "${fileDirname}\\*.cpp",
        "-o",
        "${fileDirname}\\${fileBasenameNoExtension}.exe"
    ],
```

Both two methods can change compiler optimizations and below are our benchmarking results.

Function	Size	Compiler Optimization	Execution Time
multiply_mv_row_major 	2×2	O0	0.0002 ms
		O3	0.0001 ms
	4×4	O0	0.0002 ms
		O3	0.0001 ms
	64x64	O0	0.0096 ms
		O3	0.0016 ms
	128x128	O0	0.0372 ms
		O3	0.0075 ms
	256x256	O0	0.1521 ms
		O3	0.0339 ms
	512x512	O0	0.6494 ms
		O3	0.1537 ms
	1024x1024	O0	2.5006 ms
		O3	0.6135 ms

Function	Size	Compiler Optimization	Execution Time
	2048x2048	O0	10.6520 ms
		O3	2.8979 ms
	4096x4096	O0	39.3285 ms
		O3	11.7290 ms
multiply_mv_col_major 	2x2	O0	0.0001 ms
		O3	0.0001 ms
	4x4	O0	0.0002 ms
		O3	0.0001 ms
	64x64	O0	0.0125 ms
		O3	0.0043 ms
	128x128	O0	0.0469 ms
		O3	0.0167 ms
	256x256	O0	0.1939 ms
		O3	0.1029 ms
	512x512	O0	0.7717 ms
		O3	0.9228 ms
	1024x1024	O0	2.914 ms
		O3	3.6954 ms
	2048x2048	O0	19.0277 ms
		O3	20.5780 ms
	4096x4096	O0	73.9502 ms
		O3	85.1583 ms
multiply_mm_naive 	2x2	O0	0.0001 ms
		O3	0.0001 ms
	4x4	O0	0.0003 ms

Function	Size	Compiler Optimization	Execution Time
		O3	0.0002 ms
	64x64	O0	0.6017 ms
		O3	0.1014 ms
	128x128	O0	5.0028 ms
		O3	1.2346 ms
	256x256	O0	42.0114 ms
		O3	25.4073 ms
	512x512	O0	401.1180 ms
		O3	513.2000 ms
	1024x1024	O0	3365.67 ms
		O3	3751.2500 ms
	2048x2048	O0	36642.6000 ms
		O3	40643.0000 ms
	4096x4096	O0	293795.0000ms
		O3	367138.0000 ms
multiply_mm_transposed_b 	2×2	O0	0.0002 ms
		O3	0.0001 ms
	4×4	O0	0.0004 ms
		O3	0.0001 ms
	64x64	O0	0.5924 ms
		O3	0.0973 ms
	128x128	O0	4.9857 ms
		O3	1.2785 ms
	256x256	O0	39.0641 ms

Function	Size	Compiler Optimization	Execution Time
		O3	25.9043 ms
	512x512	O0	323.898 ms
		O3	452.5820 ms
	1024x1024	O0	2647.81 ms
		O3	3641.8400 ms
	2048x2048	O0	34388.8000 ms
		O3	39239.2000 ms
	4096x4096	O0	272516.0000ms
		O3	349148.0000 ms

It can be observed that for small and medium matrices, the effect of compiler optimization is significant. The execution of O3 is much lower than that of O0 for large sizes. However when the size grow large in matrices multiplication, compiler optimization O3 behave even worse than O0 in some cases.

When short and frequently called functions are inlined, execution efficiency can be significantly improved. However, inlining complex and lengthy functions may instead lead to slower performance. From the perspective of assembly language, if inline is not used, the compiler will generate a call instruction to invoke the function and a ret instruction at the end to return the result. Both call and ret introduce additional overhead. Therefore, for short and frequently called functions, repeated use of call and ret can greatly increase execution time. By using inline, the small function body is directly substituted into the assembly code, eliminating the overhead of call and ret and thus improving efficiency. On the other hand, for complex and lengthy functions, inlining may cause code size expansion, potentially slowing down instruction caching and leading to reduced performance.

5. Profiling

TODO

6. Optimization Strategies

Our optimization strategy mainly consists of the following three points.

1. Compiler Optimization

   We observed that for lots of matrix multiplication operations, using O3 compiler optimization performs better. Therefore, for optimization purposes, we use O3 and modify the `args` parameter in tasks.json accordingly.

2. Parallelization

   We found that during the execution of the code, the CPU usage was only about 10%, and the computational resources were not being fully utilized. Therefore, we use `#pragma omp parallel for` to implement parallel operations simply to improve computation speed.

Benchmark our optimized version against the baseline, the results are displayed as table below. Both two methods can change compiler optimizations and below are our benchmarking results.

Function	**Size**	**Baseline / Optimized**	**Execution Time**
`<b>`multiply_mv_row_major `</b>`	2×2	Baseline	0.0001 ms
		Optimized	21.6644 ms
	4×4	Baseline	0.0001 ms
		Optimized	0.1598 ms
	64x64	Baseline	0.0096 ms
		Optimized	0.1892 ms
	128x128	Baseline	0.0376 ms
		Optimized	0.1849 ms
	256x256	Baseline	0.1522 ms
		Optimized	0.1745 ms
	512x512	Baseline	0.6194 ms
		Optimized	0.1600 ms
	1024x1024	Baseline	2.5921 ms

Function	Size	Baseline / Optimized	Execution Time
		Optimized	0.3867 ms
	2048x2048	Baseline	10.0876 ms
		Optimized	1.2032 ms
	4096x4096	Baseline	39.8755 ms
		Optimized	4.2755 ms
multiply_mv_col_major 	2x2	Baseline	0.0002 ms
		Optimized	0.1905 ms
	4x4	Baseline	0.0002 ms
		Optimized	0.0.1725 ms
	64x64	Baseline	0.0097 ms
		Optimized	0.1620 ms
	128x128	Baseline	0.0406 ms
		Optimized	0.1933 ms
	256x256	Baseline	0.1761 ms
		Optimized	0.1796 ms
	512x512	Baseline	0.7327 ms
		Optimized	0.1824 ms
	1024x1024	Baseline	2.8174 ms
		Optimized	0.3821 ms
	2048x2048	Baseline	18.0493 ms
		Optimized	1.4861 ms
	4096x4096	Baseline	74.3040 ms
		Optimized	6.7518 ms
multiply_mm_naive 	2x2	Baseline	0.0002 ms
		Optimized	0.1721 ms

Function	Size	Baseline / Optimized	Execution Time
	4×4	Baseline	0.0002 ms
		Optimized	0.2079 ms
	64x64	Baseline	0.6142 ms
		Optimized	0.1572 ms
	128x128	Baseline	5.0943 ms
		Optimized	0.2692 ms
	256x256	Baseline	41.8711 ms
		Optimized	1.9909 ms
	512x512	Baseline	397.0830 ms
		Optimized	19.3134 ms
	1024x1024	Baseline	3381.0400 ms
		Optimized	145.4390 ms
	2048x2048	Baseline	36201.4000 ms
		Optimized	1595.9000 ms
	4096x4096	Baseline	298681.0000 ms
		Optimized	17449.7000 ms
multiply_mm_transposed_b 	2×2	Baseline	0.0002 ms
		Optimized	0.1687 ms
	4×4	Baseline	0.0004 ms
		Optimized	0.1627 ms
	64x64	Baseline	0.5866 ms
		Optimized	0.1646 ms
	128x128	Baseline	4.9525 ms

Function	Size	Baseline / Optimized	Execution Time
		Optimized	0.2286 ms
	256x256	Baseline	39.3873 ms
		Optimized	1.3576 ms
	512x512	Baseline	321.8550 ms
		Optimized	17.3057 ms
	1024x1024	Baseline	2593.5800 ms
		Optimized	135.4340 ms
	2048x2048	Baseline	34766.5000 ms
		Optimized	1465.6300 ms
	4096x4096	Baseline	274581.0000 ms
		Optimized	15968.7000 ms