Project 2

Part 1: Program use

To experiment on different settings, please go to <config.h> where you can

1. Change the operating type (D-D, D-S, S-S)
2. Change the matrix size. Note all matrices are spare matrices
3. Change the Sparsity of the matrix. (only apply to sparse matrix)
4. Enable/Disable each optimization technique
5. Change the number of threads (only if multi-threading is enabled)

Run the main.cpp and the time elapsed will be displayed.

Part 2: Optimization Techniques

In this part we will explore the following optimization techniques for the following:

A-n. Multi-threading using n threads. (Default to 4)

B. x86 SIMD instructions

C. Minimize cache miss rate

The initial result is shown below

10,000: 517.572

Time elapsed(in second) for Dense-Dense Matrix Multiplication of difference sizes

Techniques Enabled	1000x1000	Percent	2,000x2,000	Percent
None	5.65216	100%	69.3461	100%
Multi-threading (4)	1.5246	27%	17.948	25%
SIMD instructions	1.97902	35%	21.4854	31%
Minimize cache miss rate	4.8017	85%	44.8121	65%
ALL	0.45031	8%	3.49873	5%

Matrix size of 10,000x10,000 is only tested with all 3 techniques enabled since it will take an extremely long time without them.

Assume the time used with no optimization techniques is T.

1. Multi-threading

Time elapsed(in second) for Dense-Dense Matrix Multiplication of difference sizes

Threads	1000x1000	Percent	2,000x2,000	Percent
1	5.65216	100%	69.3461	100%
2	2.96595	52%	36.7226	52%
4	1.5246	27%	17.948	25%
6	1.04773	18%	11.8386	17%
8	0.8772	15%	9.5771	14%

Theoretically the time with N threads is T/N. But because of the overhead of setting up threading, the actual time will be slightly larger than the ideal value. In the results, we can observe a 1%-3% increase from the ideal percentage. Also the effect is unlikely to vary by matrix size (computational load).

2. SIMD instructions

In this experiment using SIMD instructions reduce the time of the program by 60%, which indicates that SIMD optimizations effectively harness the parallelism inherent in dense-dense matrix operations, leading to substantial performance improvements.

This result shows what can be achieved through vectorization, guiding further optimizations and highlighting the potential benefits of leveraging SIMD-like methods in other computationally intensive tasks.

3. Minimize cache miss rate

In this experiment, I change the pattern of memory access by simply transposing the matrix before multiplication. This action reduces the time consumed by 15%. Transposing the second matrix (B) lowers cache misses by accessing B^t row-wise, thus enhancing data throughput.

4. Overall Effect

Enabling all 3 techniques gives a 90%+ reduction in the final result, which matches the product of each individual speedup. This outcome suggests that each optimization independently contributes to performance improvements with little interfering with one another.

SIMD, transposition, and multi-threading address different aspects of performance:

SIMD optimizes data-level parallelism.

Transposition improves memory access patterns.

Multi-threading enhances task-level parallelism.

Since they operate on distinct performance facets, their effects compound multiplicatively.

5. Size effect

The time complexity for regular complexity is O(N^3). So theoretically an increasing the matrix size by *A* (one-dimension) will increase the time used by *A^3* times.

Matrix Size	All Enabled	Ratio	None Enabled	Ratio
1,000 x 1,000	0.45031	1	5.65216	1
2,000 x 2,000	3.49873	7.8	69.3461	12
4,000 x 4,000	27.8468	62	537.241	95
10,000 x 10,000	451.429	1002	/	

It is notable that when optimization is enabled, the result aligns well with the theoretical expectations. When the optimization is not enabled, the ratio tend to be much higher than the theoretical expectations. One explanation could be that theoretical models abstract away constant factors and lower-order terms, but in practice, these can have a significant impact, especially for moderately sized matrices. Non-optimized code typically has higher constant factors due to the reasons mentioned above, causing the runtime to scale worse than expected.

Part 2 Dense-Sparse / Sparse-Sparse Multiplication

Matrix Sparsity = 10%

Matrix Size	Dense-Dense	Dense-Dense*	Dense-Sparse	Sparse-Sparse
1,000	5.65216	0.45031	1.1419	0.242244
2,000	69.3461	3.49873	10.0765	1.43657
4,000	537.241	27.8468	72.2395	9.05895

10,000	/	451.429	/	

Matrix Sparsity = 1%

Matrix Size	Dense-Dense	Dense-Dense*	Dense-Sparse	Sparse-Sparse
1,000	5.65216	0.45031	0.127724	0.0446074
2,000	69.3461	3.49873	0.961658	0.215529
4,000	537.241	27.8468	7.43441	1.13014
10,000	/	451.429	113.553	10.6255

Matrix Sparsity = 0.5%

Matrix Size	Dense-Dense	Dense-Dense*	Dense-Sparse	Sparse-Sparse
1,000	5.65216	0.45031	0.0726832	0.0372472
2,000	69.3461	3.49873	0.525072	0.155804
4,000	537.241	27.8468	3.82596	0.700096
10,000	/	451.429	80.4879	6.00463

Matrix Sparsity = 0.1%

Matrix Size	Dense-Dense	Dense-Dense*	Dense-Sparse	Sparse-Sparse
1,000	5.65216	0.45031	0.031296	0.034589
2,000	69.3461	3.49873	0.168323	0.134773
4,000	537.241	27.8468	1.02992	0.537902
10,000	/	451.429	13.0885	3.45389

*: Optimized

Although the Sparsity (S) does not affect the time complexity but it introduce a constant factor S that can still significantly reduce the time consumption. Although not as obvious dense-dense multiplication, from the result we can still roughly observe a cubic-like trend in time consumption with changing size and the same sparsity and type.

With varying matrix sparsity and the same matrix size/type, we can observe the linear reduction effect.

Although the optimal sparsity threshold for compression depends on various factors, A Sparsity of 1% or less is generally suitable to consider compression.