Adv. Computer Systems - Project 2

Running Instructions: **2**
Grading Criteria **3**
Software Implementation: 3
Experimental Results 4
Optimization Experiment 6
Dense-Dense Matrix Multiplication 8
Sparse-Sparse Matrix Multiplication 8
Dense-Sparse Matrix Multiplication 10
Final Analysis and Conclusions 12

Running Instructions:

The project file, proj2.c can be compiled with the command “gcc -mavx2 -o proj2.o proj2.c” and run with the command “./proj2.o” in the terminal.

The program will ask for user-input for fields regarding row#/col# for both matrices, sparsity (0-1, with 0 being fully dense, 1 being all zeros), multithreading (0-no/1-yes), thread count, use of SIMD optimization (0-no/1-yes), the use of blocking cache miss minimization (0-no/1-yes), and whether or not to print the matrices (0-no/1-yes).

In the case where the user does not want 2 matrices with the same dimensions/sparsity (such as the Dense-Sparse readings), the user should follow the instructions on line 433 in the program and manually set the two matrices before running.

Note that for the following tests, the 10,000 matrix size was down-scaled to 5,000 in order to run within a respectable time frame. This policy was approved by the professor.

The data below was gathered on a ThinkPad-T14s Gen 1 Laptop containing an AMD Ryzen 5 Pro 4650U Processor.

Gathered data is reported in the PDF attached. The graphs within the PDF were generated in this spreadsheet:
https://docs.google.com/spreadsheets/d/1BiId6bl0j7WkxflLQXvMY7MJ8QMs5n6QPCHETDuyKFw/edit?usp=sharing

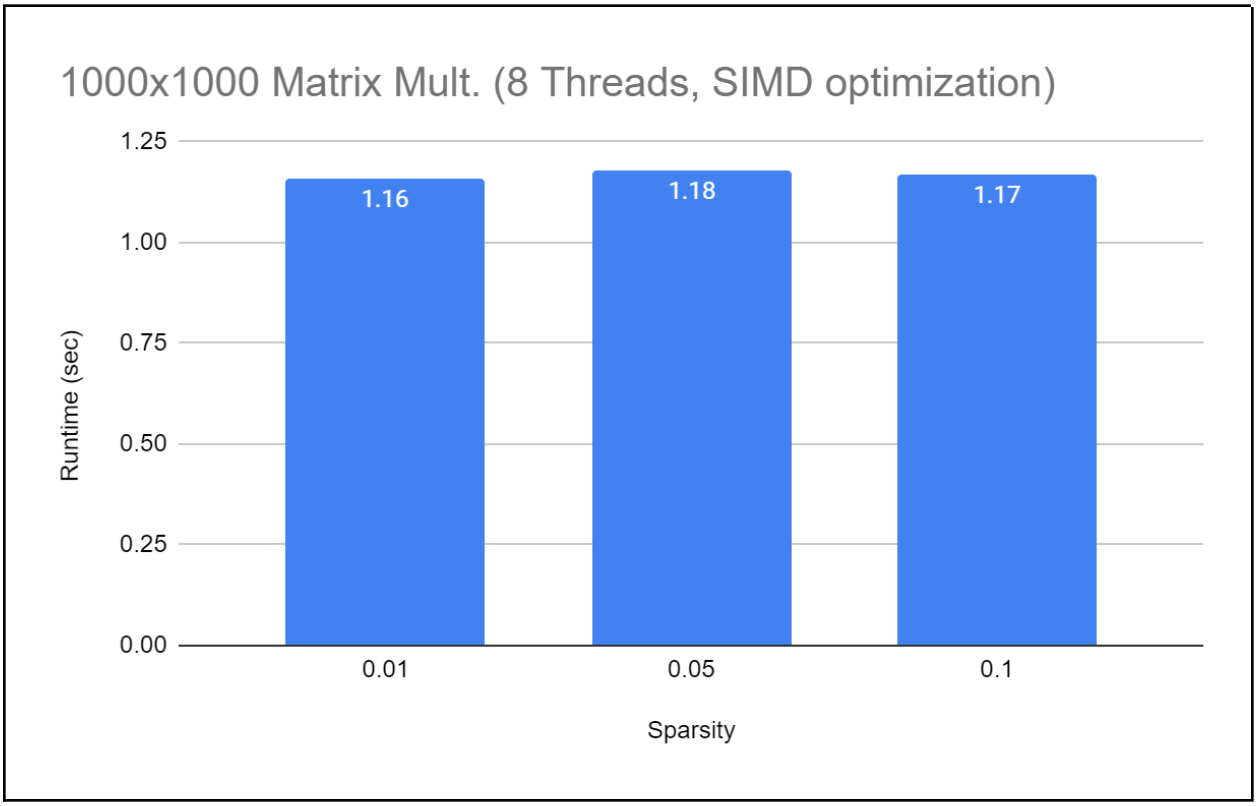
Grading Criteria

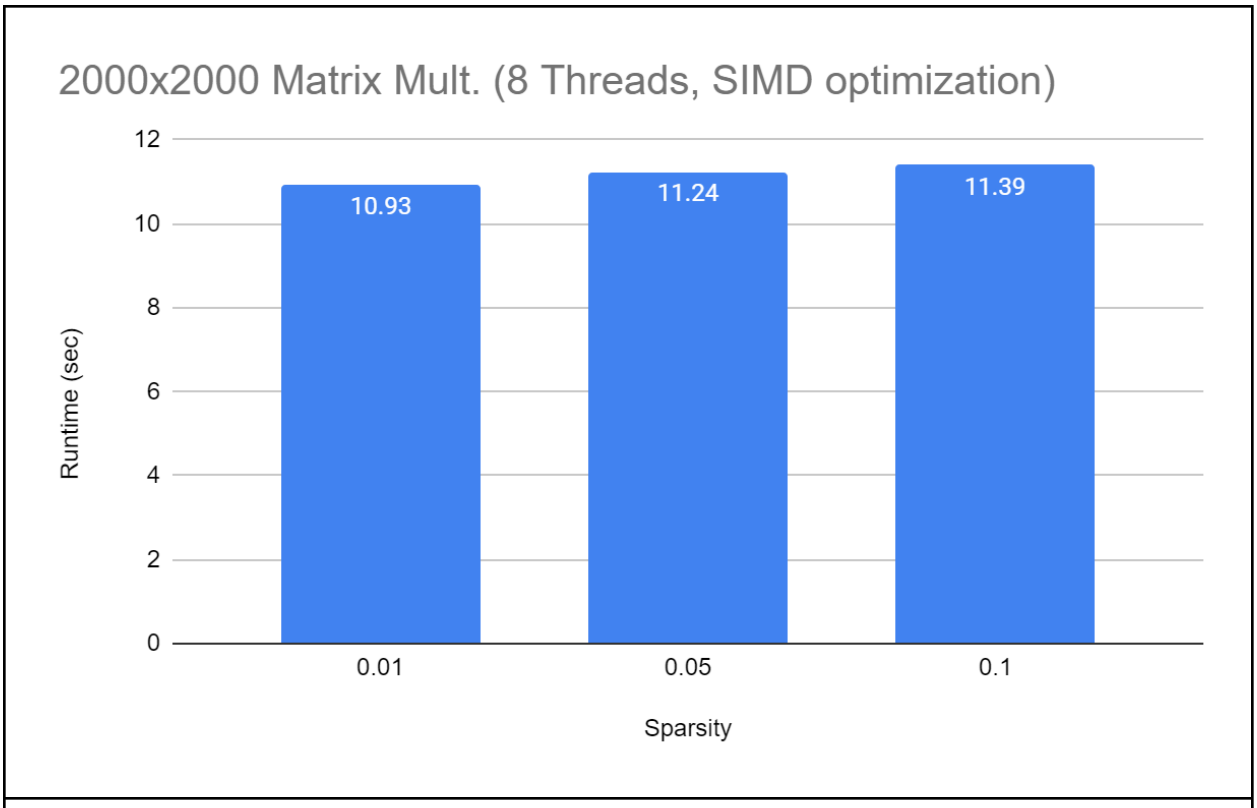
Software Implementation:

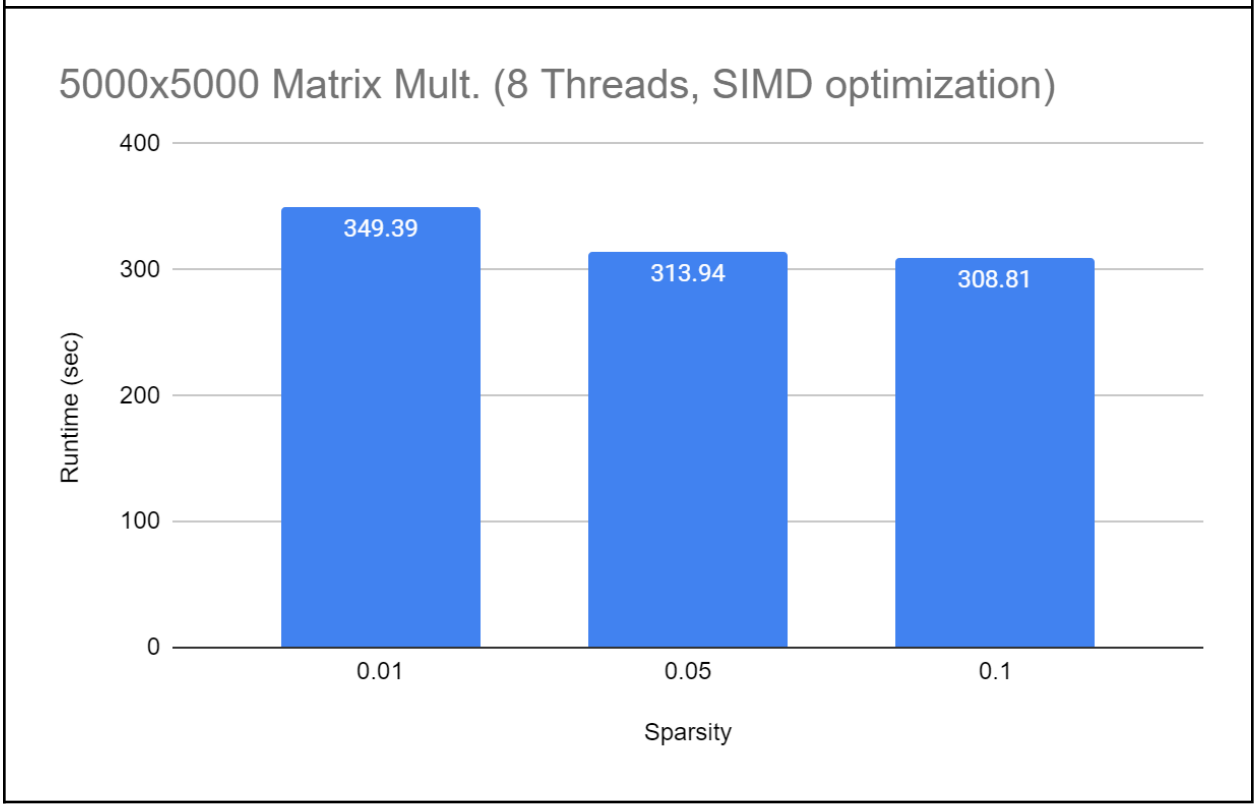
- Supports arbitrary matrix sizes much larger than cache capacity: *Matrices are defined by user-input parameters (2 integers).*
- Supports configurable multi-threading: *Multithreading is supported by the multithread_multiply() struct.*
- Supports configurable SIMD optimization: *SIMD is supported by the thread_multiplySIMD() and multiplySIMD() structs.*
- Supports configurable cache miss minimization: *Cache minimization is supported by the multipleBlock() and thread_multiplyBLOCK() structs.*
- Supports changing the number of threads: *The number of threads is defined by user-input parameters (1 integer, given multithreading is chosen).*

Experimental Results

9 experiments were run with varying sparsity (0.01, 0.05, 0.1) and dimensions (1k, 2k, 5k). The three graphs generated from these tests can be seen below. The results show that for the small 1000x1000 matrices, sparsity played little role in differing the runtimes due to the minimal amount of elements to multiply. For 2000x2000, increasing the sparsity value (adding more zeroes) caused the runtime to slightly increase, but it is still within margin of error. This could be due to the computation time for handling skip-zero cases. For 5000x5000, which had by far the most elements, increasing sparsity saw the runtime decrease. Across the three dimension sizes, there was a clear and sizable increase in runtime as the dimensions increased. This is due to the dramatically increased number of elements to multiply.

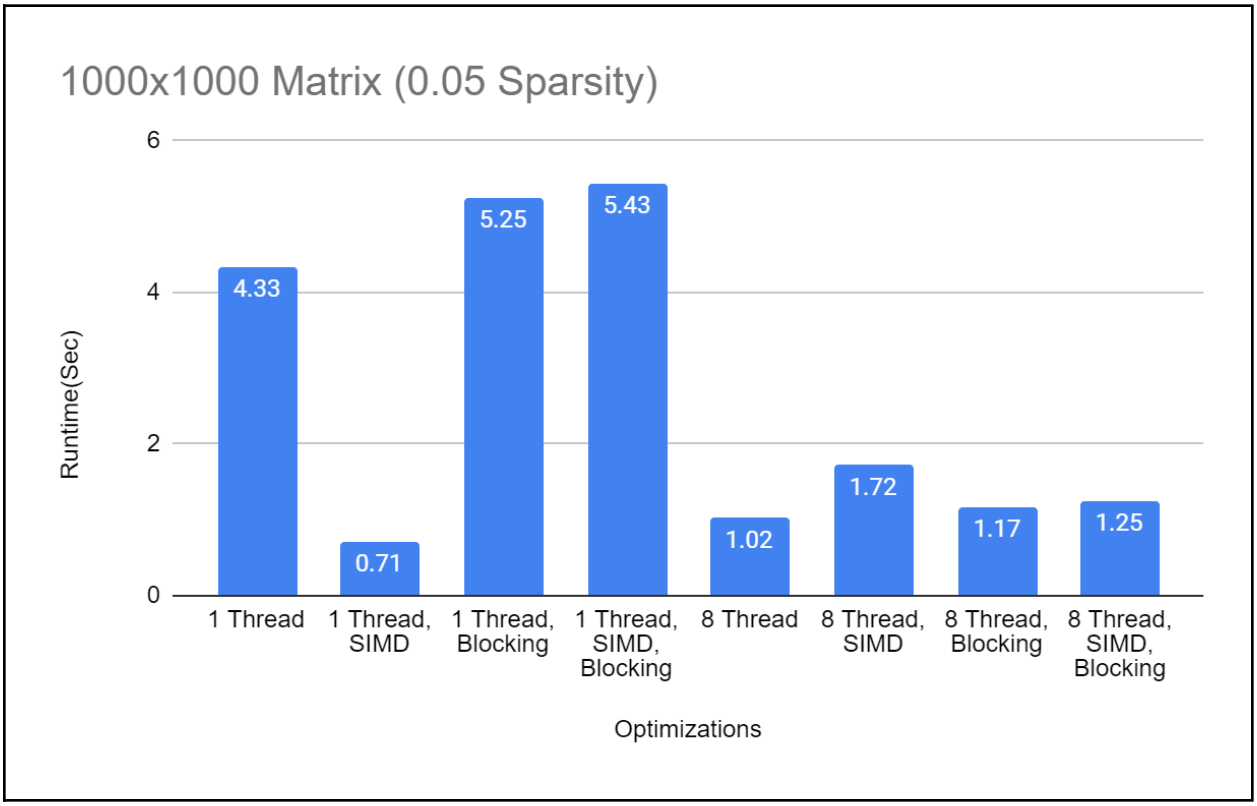


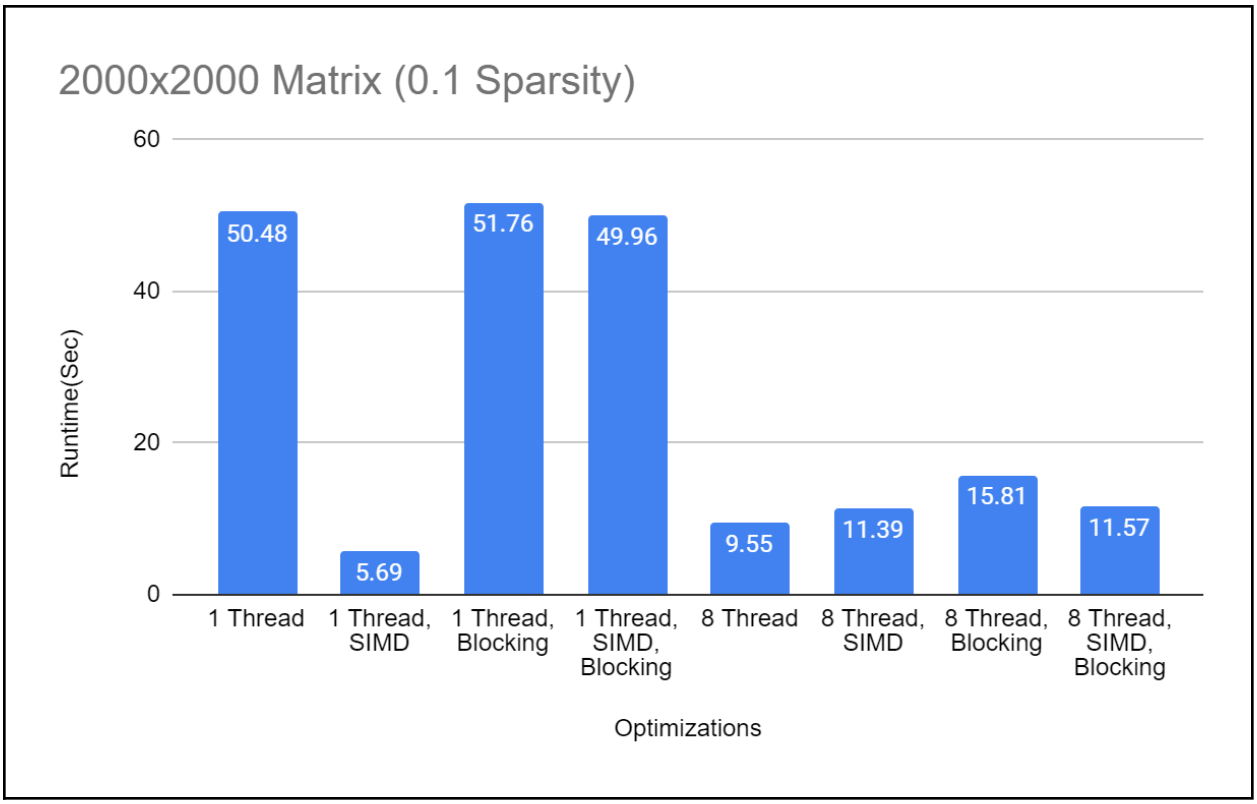

2000x2000 Matrix Mult. (8 Threads, SIMD optimization)
Runtime (sec)
12
10
8
6
4
2
0
10.93
11.24
11.39
0.01
0.05
0.1
Sparsity


5000x5000 Matrix Mult. (8 Threads, SIMD optimization)
Runtime (sec)
400
300
200
100
0
349.39
313.94
308.81
0.01
0.05
0.1
Sparsity

Optimization Experiment

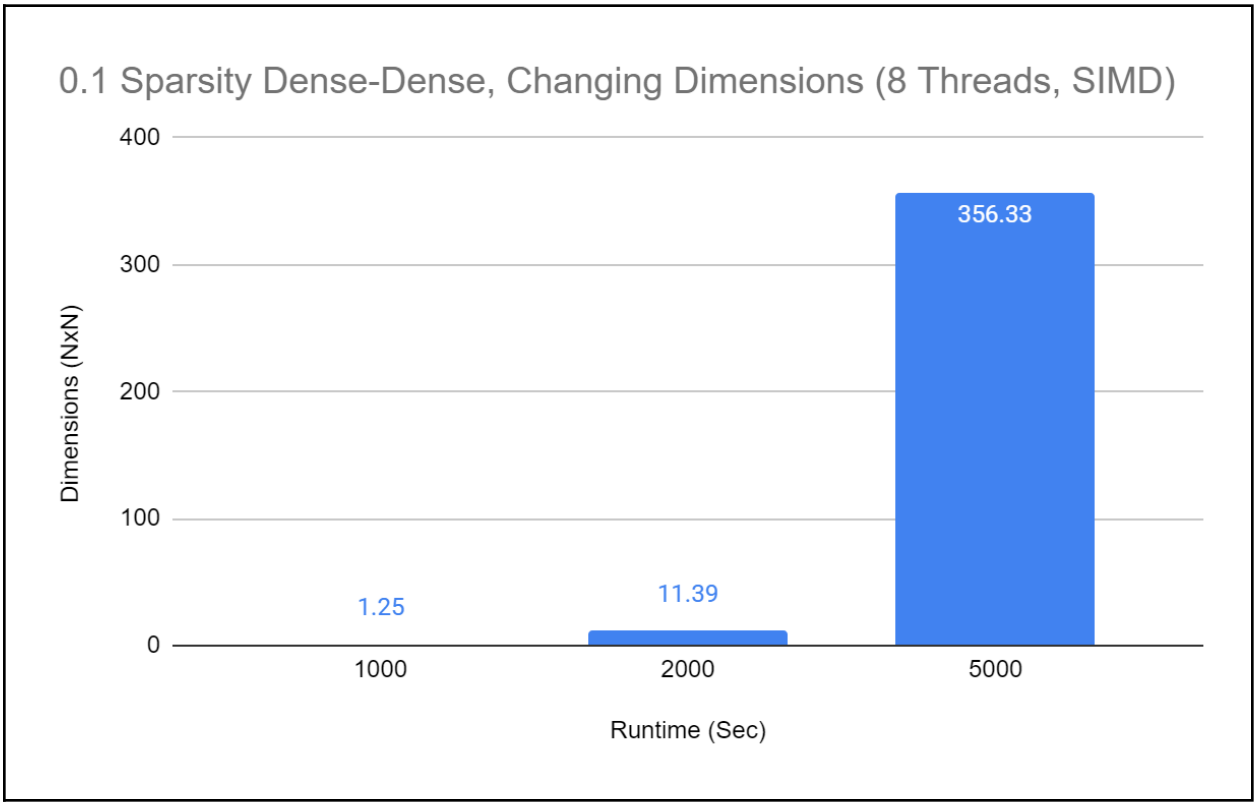
The optimization experiment required running all 8 optimization methods on two separate matrix-multiplication types. The first time was multiplying 1000x1000 matrices with 0.05 sparsity. The second was multiplying 2000x2000 matrices with 0.1 sparsity. The results can be seen in the two graphs below. Both tests had similar results. SIMD optimization by itself had the largest decrease in total runtime. After that, including multithreading had the second largest decrease in runtime. Including blocking or other optimization methods did not seem to generally decrease runtime by a significant factor. In some cases the runtime increased but was still within margin of error. This lack of time saving could be due to one of two things, overhead within the optimization functions adding significant computation cost, or a lack of elements to multiply due to the smaller matrix dimensions.




2000x2000 Matrix (0.1 Sparsity)
Runtime(Sec)
60
40
20
0
50.48
5.69
51.76
49.96
9.55
11.39
15.81
11.57
1 Thread
1 Thread, SIMD
1 Thread, Blocking
1 Thread, SIMD, Blocking
8 Thread
8 Thread, SIMD
8 Thread, Blocking
8 Thread, SIMD, Blocking
Optimizations

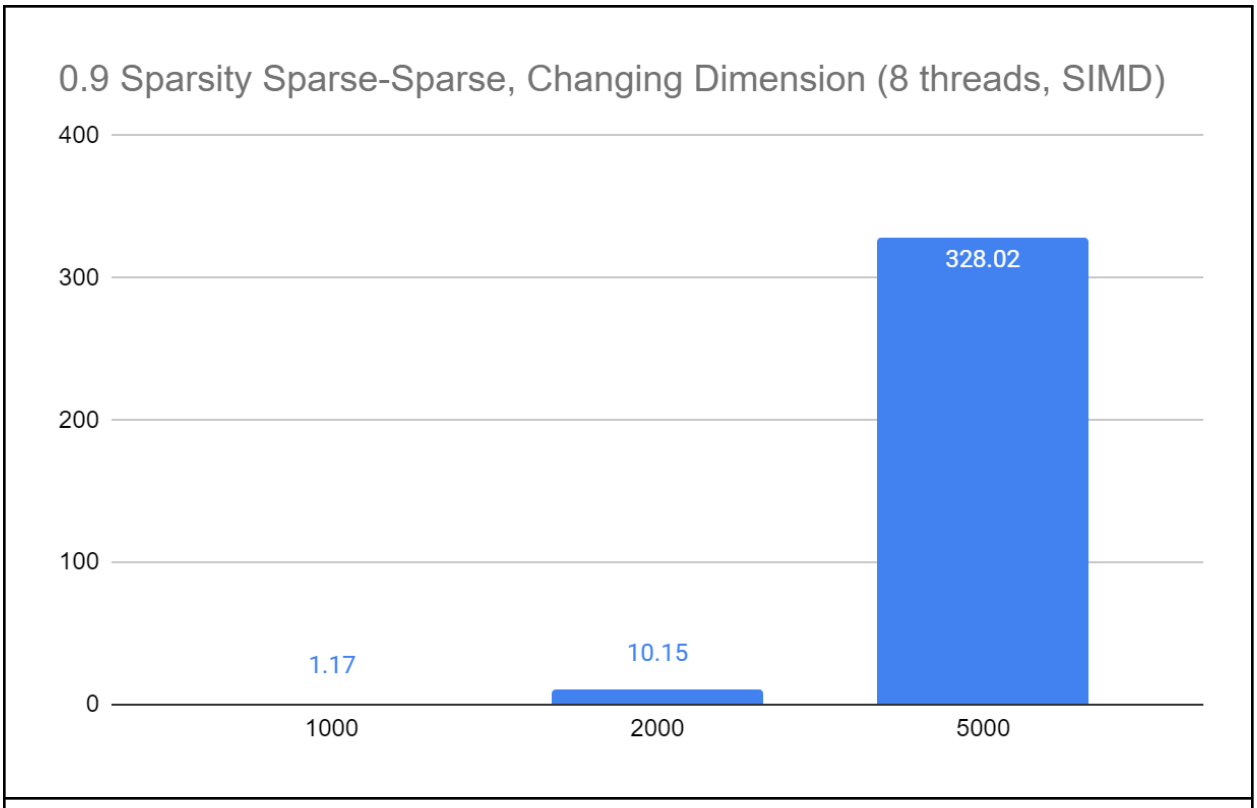
Dense-Dense Matrix Multiplication

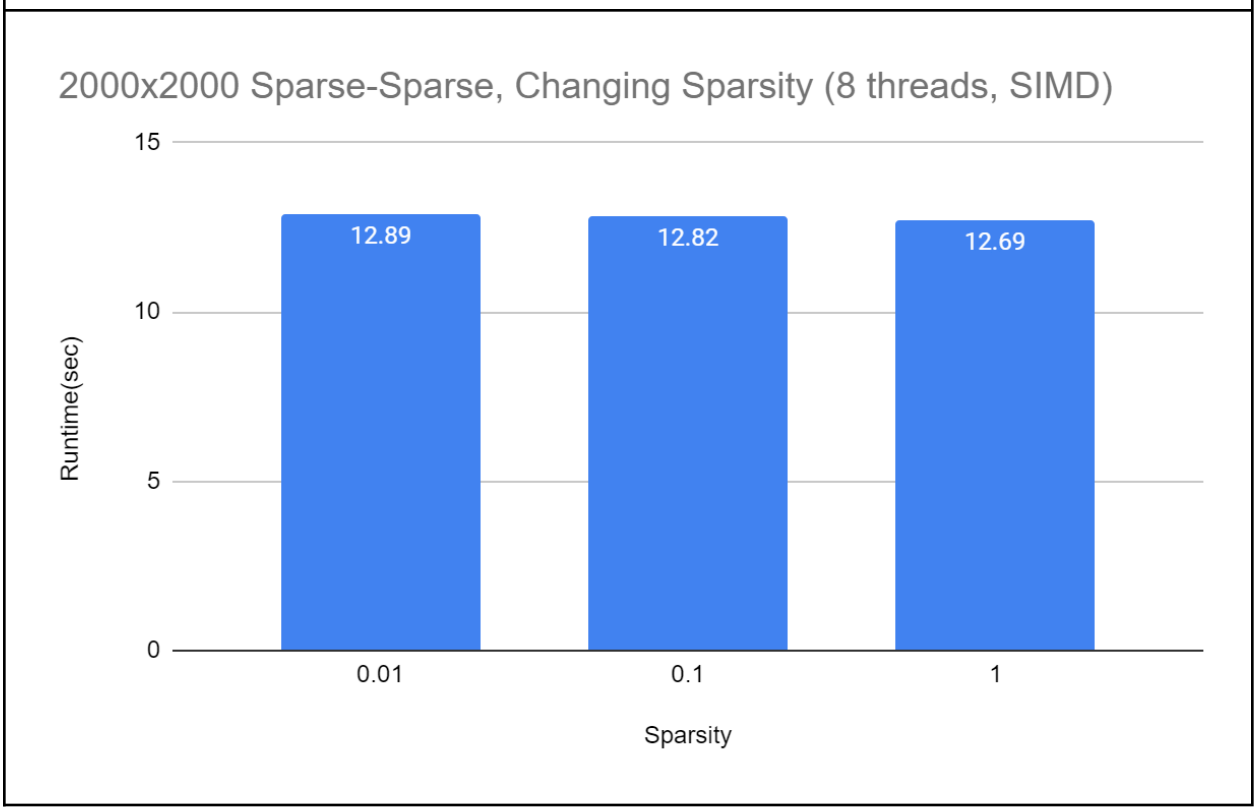
This test looked at the result of multiplying two dense matrices (0.1 sparsity) across 3 different dimensions (1000, 2000, 500). The results show a clear increase in runtime as the dimension size increases at high density.



Sparse-Sparse Matrix Multiplication

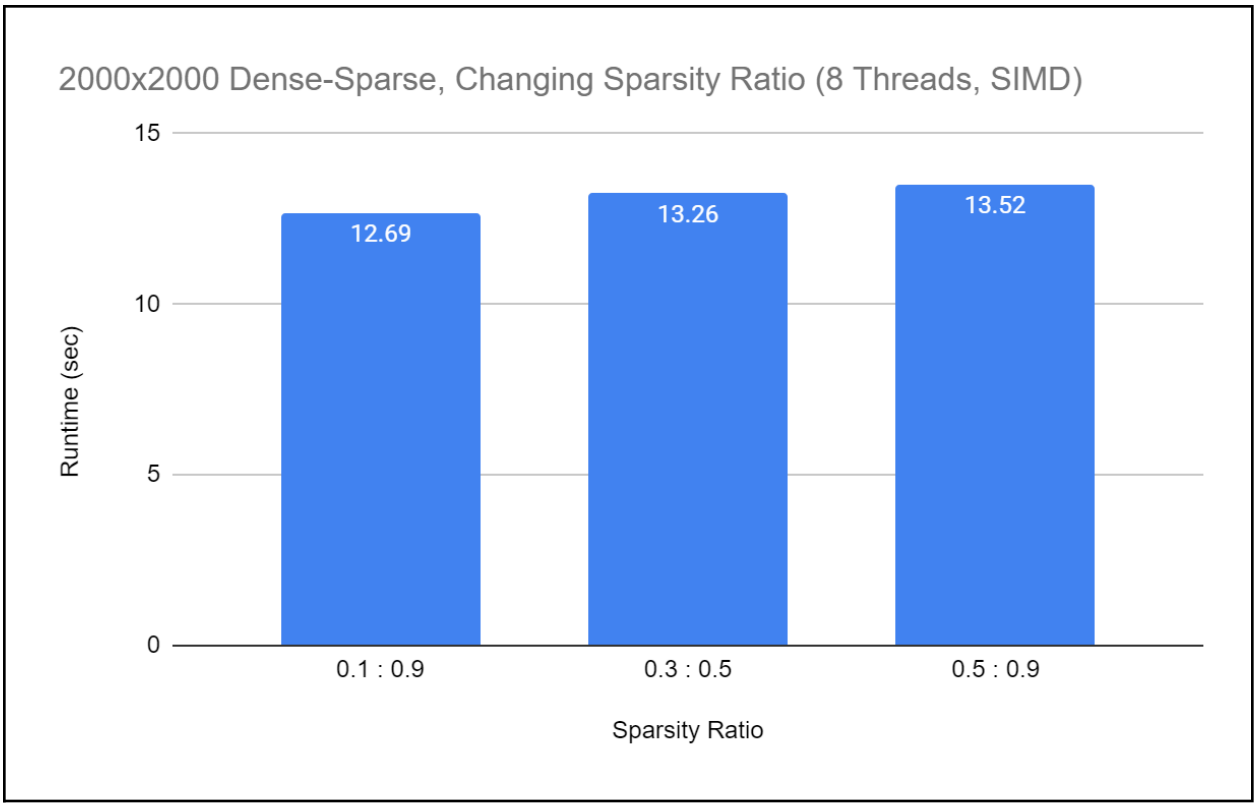
This test looked at the result of varying sparsity in matrix multiplication, and the result of varying dimensions at high sparsity (0.9). Both charts are included below. It can be seen that increasing sparsity (adding more zeros), causes the runtime to decrease, and runtime still heavily increases with dimensionality. It is worth noting that this heavy increase is less than the increase seen in the dense-dense test, meaning skipping zeros could be having positive effects on minimizing runtime.

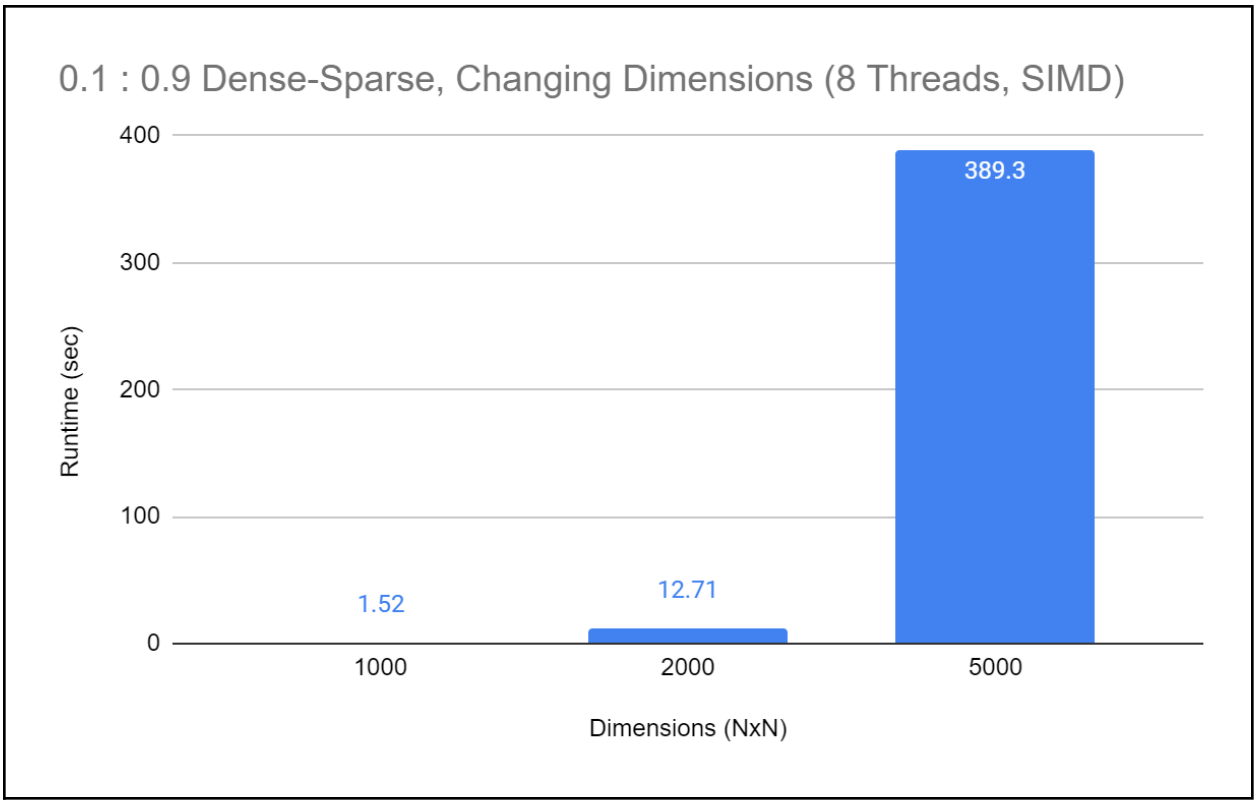

0.9 Sparsity Sparse-Sparse, Changing Dimension (8 threads, SIMD)
400
300
200
100
0
1.17
10.15
328.02
1000
2000
5000


2000x2000 Sparse-Sparse, Changing Sparsity (8 threads, SIMD)
15
10
5
0
Runtime(sec)
12.89
12.82
12.69
0.01
0.1
1
Sparsity

Dense-Sparse Matrix Multiplication

For this test, two different setups were used. The first looked at adjusting the sparsity ratio between two 2000x2000 dimension matrices being multiplied, and the second looked at increasing the dimensions of two matrices with a 0.1:0.9 sparsity ratio. For the first test, the first matrix's sparsity was steadily increased (0.1, 0.3, 0.5), while the second matrix remained at 0.9. The runtime remained fairly equivalent, but there was a slight increase in runtime as the sparsity of the first matrix increased. For the second test, the results showed that the runtime of sparse-dense multiplication dramatically increased with dimensionality. This increase was greater than the dense-dense or sparse-sparse tests.




0.1 : 0.9 Dense-Sparse, Changing Dimensions (8 Threads, SIMD)
400
300
200
100
0
Runtime (sec)
1.52
12.71
389.3
1000
2000
5000
Dimensions (NxN)

Final Analysis and Conclusions

Across these various tests, there are several conclusions that can be drawn. Firstly, increasing dimension size will have the largest impact on runtime, due to the increased amount of elements that need to be multiplied. Sparsity has a varying effect on runtime. For smaller dimensions, higher sparsity can lead to a slightly higher runtime, perhaps due to the overhead of checking for zeros to skip. For larger dimensions, increased sparsity can end up saving time, such as it did in the “Experimental Results” section. In terms of optimization, SIMD and multithreading can save a substantial amount of time when multiplying matrices, through parallelization and optimized functions. Blocking (cache miss minimization) did not have as much of a measurable impact in this implementation, perhaps due to increased function overhead.