Pritom Paul 10/12/2024

ACS- Project 2: Matrix-Matrix Multiplication

Overview:

The 2nd project from my Advanced Computer System class (ECSE 4320) includes the main idea of Matrix-matrix multiplication. The objective is to build a program in C++ that carries out high-speed dense/sparse matrix-matrix multiplication by explicitly utilizing:

1. No optimization
2. Multiple threads
3. x86 SIMD instructions
4. Minimize cache miss rate via restructuring data access patterns or data compression (Cache optimization)
5. All optimization techniques together

These concepts can be helpful for numerous real-life applications, e.g., machine learning, computer vision, signal processing, and scientific computing. As such, my code for the optimizations is not the best but is optimized enough to work from matrix size from 100 to 2000 while some optimizations can work with 5000 or 10000 matrices. The output from this experimental data will be the execution time (in seconds).

Structure/Installation/Usage of Code:

The way how I structure my code in the MMM.cpp include the necessary libraries, global variables that is used as a default options, functions for each of the 5 optimizations used as explained from the overview, and the main function that contains its arguments checks, printing statement of what optimization your using along with the size, sparsity, and timing.

In order to start compile the program, you must use this command:

```
g++ -o matrix_mult.out MMM.cpp -march=native -O2 -pthread
```

The reason is that you have to ensure you use the correct flags to enable AVX2 optimizations (for the x84 SIMD instructions), as it is specified as the architecture explicitly.

When executing the code from your WSL or any of ur command prompt applications, in order to use different optimizations on matrix multiplications performance, here are the structure for the argument commands to use:

./matrix_mult.out --size # --sparsity # --optimization

- --size #: define any size of the square matrix, this is mandatory argument you have to use
- --sparsity #: define any percentage of the sparsity of the matrix. This is an add-on parameter to have your matrix as a sparse matrix
- --optimization: **DON'T** type optimization as you have your optional optimization you can use
    - Leave it blank means you set the thread = 1 and uses no optimizations
    - --mult_threads #: enable mult_threading optimizations along with setting whatever number of threads you want to use
    - --simd: enable the simd instructions optimizations
    - --cache: enable cache_optionimcations

Matrix size and Sparsity Experimental Data:

The following data below was obtained by running tests using no optimizations only, but we have shown 3 cases of different matrix size and different sparsity to demonstrate the performance.

The data in the table below was obtained by running: ./matrix_mult.out --size # --sparsity #

Matrix size & Sparsity	Time (in seconds)
Matrix Size: 500x500, **Sparsity:** 0.1%	0.941219
Matrix Size: 500x500, **Sparsity:** 1%	0.954555
Matrix Size: 500x500, **Sparsity:** 10%	0.952239
Matrix Size: 1000x1000, **Sparsity:** 0.1%	8.43344
Matrix Size: 1000x1000, **Sparsity:** 1%	8.44614
Matrix Size: 1000x1000, **Sparsity:** 10%	8.25905
Matrix Size: 2000x2000, **Sparsity:** 0.1%	122.999
Matrix Size: 2000x2000, **Sparsity:** 1%	125.303
Matrix Size: 2000x2000, **Sparsity:** 10%	125.628

Optimization Experiment:

In this section, we will present the performance of each of the matrix-matrix multiplication optimization used by the --optimization flag when executing the code in the terminal.

No optimization:

The data below can be run by: ./matrix_mult.out --size # --sparsity #

Matrix Size	Time (in seconds)
100	0.00715978
250	0.114206
500	0.945089
750	3.27899
1000	8.32149
1250	21.145
1500	31.7145
2000	122.081
2250	162.235

Multi-threading

For the multi-threading section, the overall optimization comparison will be using 4 threads as our case. However, solely for this section I will demonstration all the experimental data using (2, 4, 8, 16) threads individually:

2 threads:
The data below can be run by: ./matrix_mult.out --size # --mult_threads 2

Matrix Size	Time (in seconds)
100	0.000816906
250	0.00688822
500	0.0551302

750	0.197309
1000	0.493587
1250	1.025
1500	1.99375
2000	9.19821
2250	21.9193
5000	473.463

4 threads:
The data below can be run by: ./matrix_mult.out --size # --mult_threads 4

Matrix Size	Time (in seconds)
100	0.000376605
250	0.00377515
500	0.0344287
750	0.112686
1000	0.29103
1250	0.742425
1500	1.26404
2000	5.40789
2250	11.7451
5000	267.211

8 threads:
The data below can be run by: ./matrix_mult.out --size # --mult_threads 8

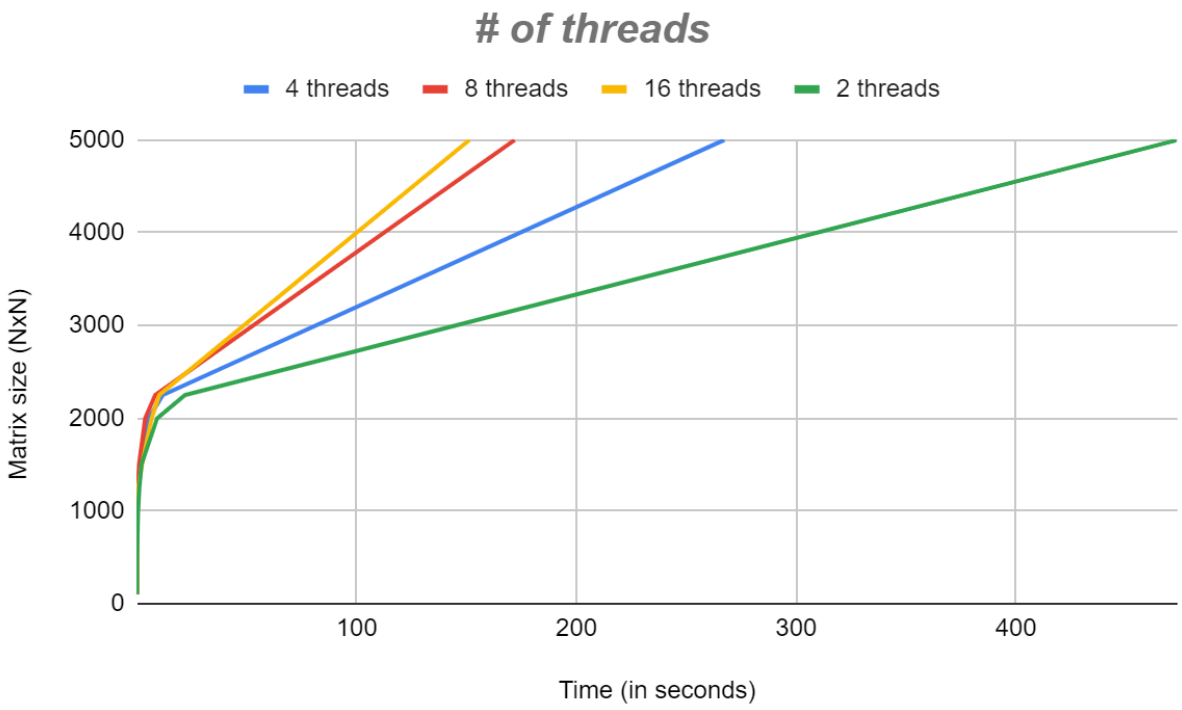
Matrix Size	Time (in seconds)
100	0.00072932
250	0.0026978
500	0.0224965

750	0.0803511
1000	0.204671
1250	0.516033
1500	0.992502
2000	3.75361
2250	8.37807
5000	171.84

16 threads:
The data below can be run by: ./matrix_mult.out --size # --mult_threads 16

Matrix Size	Time (in seconds)
100	0.00125269
250	0.00272667
500	0.0218295
750	0.0692902
1000	0.184547
1250	0.574981
1500	2.27046
2000	6.68064
2250	10.1837
5000	151.35

Analysis of the Multi-threads



The graph is all of the data recorded from the different number of threads being used as the experiment. As seen from the graph and all of the tables, the best performance is using the 16 threads as predicted. Multithreading is a program execution that allows for multiple threads to be created within a process, executing independently but concurrently sharing process resources. From this definition having a higher number of threads will increase the performance time for the program to run. Having multiple threads working at once will result in faster time, as you are multitasking 16 tasks at once. As a result my data proves multithreading will become faster as you have more threads, hence 16 threads is the fastest in this experiment.

(x86) SIMD Instruction:

As a remainder when compiling the program, make sure you run this:

```
g++ -o matrix_mult.out MMM.cpp -march=native -O2 -pthread
```

The data below can be run by: ./matrix_mult.out --size # --simd

Matrix Size	Time (in seconds)
100	0.000613477
250	0.00549543
500	0.0269474
750	0.0873653

1000	0.114119
1250	0.321107
1500	0.60621
2000	1.22345
2250	2.03386
5000	20.9607
10000	309.395

Cache_optimization:

The data below can be run by: ./matrix_mult.out --size # --cache

Matrix Size	Time (in seconds)
100	0.000567917
250	0.00876635
500	0.0684486
750	0.230353
1000	0.568853
1250	1.17233
1500	1.91992
2000	4.97786
2250	6.80408
5000	83.5577
10000	650.733

All optimization together:

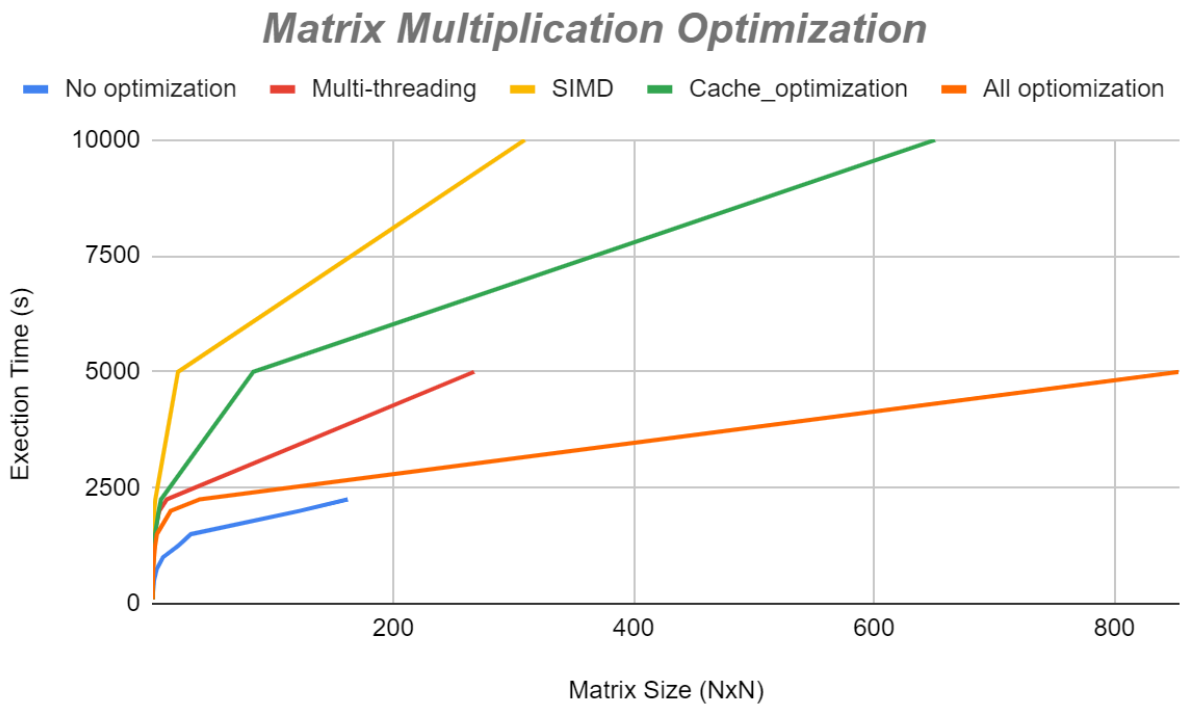
The data below can be run by: ./matrix_mult.out --size # --all

Matrix Size	Time (in seconds)

100	0.000705249
250	0.0132459
500	0.109481
750	0.396642
1000	0.948884
1250	1.82516
1500	3.35297
2000	14.8077
2250	39.0151
5000	854.074

Comparison and Analysis:

The graph below shows the execution time for matrix multiplication based on different sized matrices and different types of optimizations.



As shown from the graph and the data, the results are worse when there are no optimizations. It was surprising to me that when having all the optimization running, it was the 2nd worst optimizations to use compared to all the individual optimization techniques. This is probably due to the data that even if you incorporate all the best possible optimizations, if they can't work well together, then it can't be the best algorithm to use. Multi-threading has some performance speed, but not compared to SIMD and cache optimization. This makes sense because every column access into the second matrix results in a cache miss, which happens N times for each element in the resulting matrix. The top performance algorithm is the SIMD instructions, such as the x86 architectures (e.g., AVX, SSE), allow you to operate on multiple data points simultaneously. In addition, using SIMD intrinsics will be the most advantageous compared to the other as it allows faster matrix multiplications by parallelizing operations at the instruction level.

Dense-Dense matrix multiplication:

(This is a copy from the no optimization table its the same thing to be considered Dense*Dense)

Matrix Size	Time (in seconds)
100	0.00715978
250	0.114206
500	0.945089
750	3.27899
1000	8.32149
1250	21.145
1500	31.7145
2000	122.081
2250	162.235

As shown from the table, the more matrix size you increase, the longer the speed it will take for the program to finish. This makes sense as having a large amount of data in any kind of array rather than its arrays, list, vectors, or matrix, you have to transversal each of the elements of both matrices to get your results, hence leading to more time needed to complete the program.

Sparse-Sparse matrix multiplication:

For this section, we will now use the --sparsity # flag to make our matrices to be a sparse-matrix and compare the results from matrix size vs. sparsity if we make one of them constant when running over various optimization functions.

Keeping size constant, different sparsities:

No optimization		
Matrix Size	Sparsity (%)	Time (s)
1000	0.1	0.915371
1000	1	0.929117
1000	10	0.91726

Multi-threading (4 threads)		
Matrix Size	Sparsity (%)	Time (s)
1000	0.1	0.320442
1000	1	0.293184
1000	10	0.324202

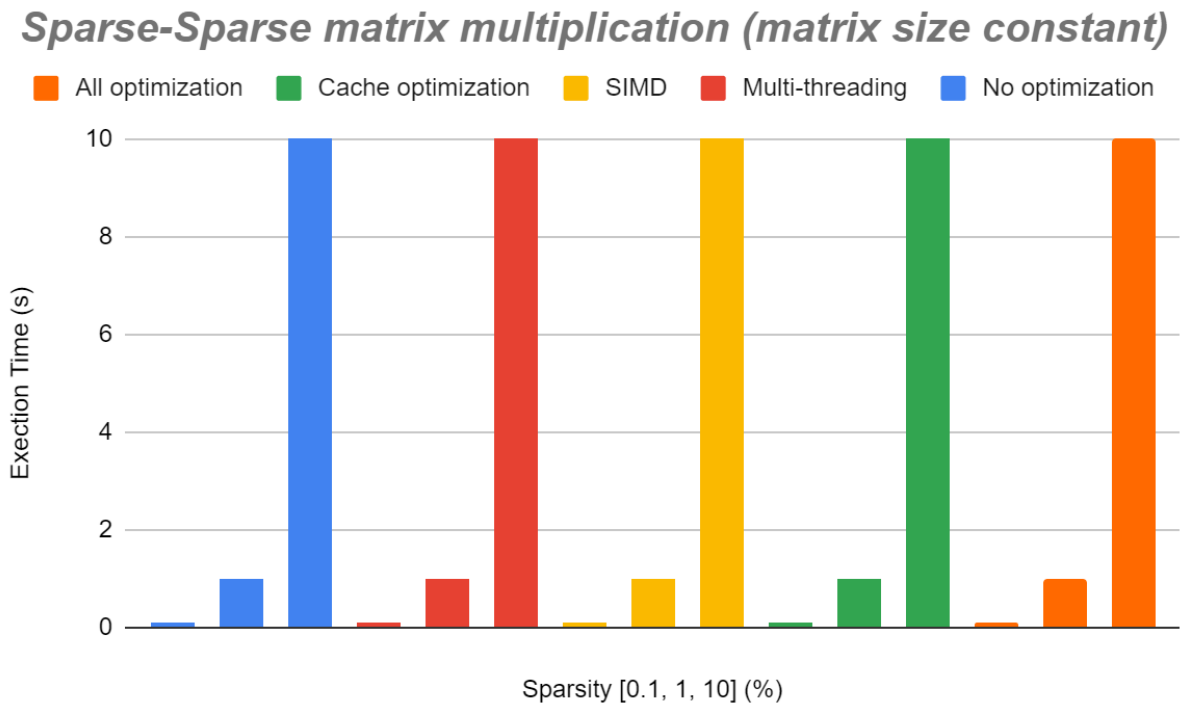
SIMD		
Matrix Size	Sparsity (%)	Time (s)
1000	0.1	0.114124
1000	1	0.114102
1000	10	0.112224

Cache optimization		
Matrix Size	Sparsity (%)	Time (s)
1000	0.1	0.57167

1000	1	0.606611
1000	10	0.565383

All Optimization		
Matrix Size	Sparsity (%)	Time (s)
1000	0.1	0.138638
1000	1	0.14415
1000	10	0.14224

The graph below is the chart version of all of the optimizations used for this experiment.



Analysis:

Comparing the data table and the graph, we can see that the optimization that has the best performance to worst performance are:

SIMD > All opt. > cache opt. > multi-thread > No opt.

It kinda surprised me that All optimization was the 2nd best optimization algorithm to use, so maybe this is due to the fact that sparsity allows only some data to be used

while the rest of the elements have no values. It makes sense that SIMD is the best performance as shown from the Optimization Experiment.

No Optimization				
Dense-Dense matrix multiplication		Sparse-Sparse matrix multiplication (constant size)		
Matrix Size	Time (s)	Matrix Size	Sparsity (%)	Time (s)
1000	8.32149	1000	10	0.91726

Comparing the from both Dense-Dense matrix and Sparse-Sparse matrix multiplication when both have a matrix size of 1000, we can see that having sparsity will drastically reduce the speed to completion. As such, having sparsity will result in having better performance.

Different size, Keep sparsities constant:

For this experiment, we will continue to be doing Sparse-Sparse matrix multiplication, however we will now keep sparsity constant while changing the matrix size.

No optimization		
Matrix Size	Sparsity (%)	Time (s)
500	10	0.10993
1000	10	0.990546
2000	10	15.1973

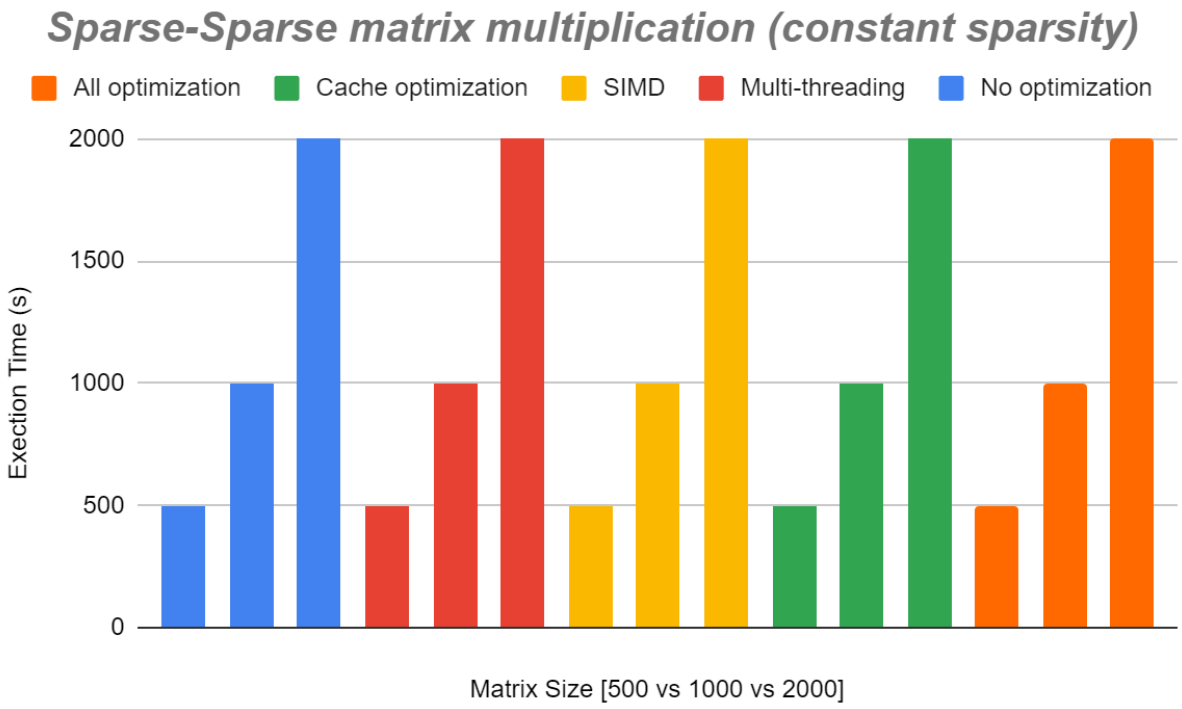
Multi-threading (4 threads)		
Matrix Size	Sparsity (%)	Time (s)
500	10	0.029289
1000	10	0.315544
2000	10	5.42642

SIMD		
Matrix Size	Sparsity (%)	Time (s)
500	10	0.0248119
1000	10	0.112846
2000	10	1.22376

Cache Optimization		
Matrix Size	Sparsity (%)	Time (s)
500	10	0.0732897
1000	10	0.576636
2000	10	4.75051

All optimization		
Matrix Size	Sparsity (%)	Time (s)
500	10	0.0264492
1000	10	0.140358
2000	10	0.995043

The graph below is the chart version of all of the optimizations used for this experiment.



Analysis:

Comparing the data table and the graph, we can see that the optimization that has the best performance to worst performance are:

All opt. > SIMD > cache opt. > multi-thread > No opt.

It kinda surprised me that All optimization did slightly better than SIMD optimization.

Comparing each of the optimizations used when we keep Matrix size constant vs Sparsity constant, we can see that there's an increase in time consumption for the Sparsity constant. This makes sense as for the sparsity matrix, only some of the elements in the matrix will be non-zero values. So if you increase the sparsity %, you will have more values that are non-zero values, however if you increase the matrix size, that matrix will still contain more non-zero values, which results in having more time to complete the matrix multiplication. As a result, changing the sparsity will lead to better performances.

Dense-Sparse matrix multiplication:

In this experiment, we will now have a situation where one matrix will be a Dense matrix and the other will be a Sparse matrix. We will multiply them together using various optimization algorithms and change what's our independent variable for the matrix sizes and sparsity.

```
269     //std::vector<std::vector<float>> B = createMatrix(size, sparsity);
270
271     //uncomment this block if you are planning to do a Dense-Sparsity matrix multiplication and manunally changes the size and
        sparsity, otherwise uncomment above and comment this matrix
272     std::vector<std::vector<float>> B = createMatrix(size, 10.0);
```

Note, from this line of code, you have to manually change what the size and sparsity it will be. Since re-coding it to be able to work with dense-sparse will take too much time, it is just easier to manually change it as it says. So make sure you comment line 269 and uncomment line 272 when using Dense-Sparse Matrix multiplication. Otherwise, if you want to revert back to the original, comment 272 and uncomment 269.

Keeping size constant, different sparsity:

The data below has been collected from running: ./matrix_mult.out --size 1000

No optimization		
Matrix Size	Sparsity (%) for Matrix B	Time (s)
1000	0.1	0.922368
1000	1	0.912018
1000	10	0.92091

The data below has been collected from running: ./matrix_mult.out --size 1000 --mult_threads 4

Multi-threading (4 threads)		
Matrix Size	Sparsity (%) for Matrix B	Time (s)
1000	0.1	0.313711
1000	1	0.317127
1000	10	0.28822

The data below has been collected from running: ./matrix_mult.out --size 1000 --simd

SIMD

Matrix Size	Sparsity (%) for Matrix B	Time (s)
1000	0.1	0.114466
1000	1	0.11511
1000	10	0.115435

The data below has been collected from running: ./matrix_mult.out --size 1000 --cache

Cache optimization		
Matrix Size	Sparsity (%) for Matrix B	Time (s)
1000	0.1	0.573008
1000	1	0.576328
1000	10	0.560224

The data below has been collected from running: ./matrix_mult.out --size 1000 --all

All Optimization		
Matrix Size	Sparsity (%) for Matrix B	Time (s)
1000	0.1	0.141539
1000	1	0.137205
1000	10	0.134273

Analysis:

From looking at the various data tables, no matter what optimization algorithm you use, changing the sparsity % for matrix B while keeping the same matrix size will result in barely changing the performance. As such, you would only expect to have a slight increase/decrease by a few decimals as it is almost consistent. Comparing this data to the Sparse-Sparse matrix multiplication on the section of “Keeping size constant, different sparsities”, we can see that they are almost the same.

Different size, Keep sparsities constant:

For this experiment, we will continue to be doing Dense-Sparse matrix multiplication, however we will now keep sparsity constant while changing the matrix size of Matrix B.

```
269     //std::vector<std::vector<float>> B = createMatrix(size, sparsity);
270
271     //uncomment this block if you are planning to do a Dense-Sparsity matrix multiplication and manunally changes the size and
        sparsity, otherwise uncomment above and comment this matrix
272     std::vector<std::vector<float>> B = createMatrix(size, 10.0);
```

The data below has been collected from running: ./matrix_mult.out --size #

No optimization		
Matrix Size for Matrix B	Sparsity (%)	Time (s)
500	10	0.108602
1000	10	0.937935
2000	10	14.3494

The data below has been collected from running: ./matrix_mult.out --size # --mult_threads 4

Multi-threading (4 threads)		
Matrix Size for Matrix B	Sparsity (%)	Time (s)
500	10	0.0299151
1000	10	0.310713
2000	10	5.74608

The data below has been collected from running: ./matrix_mult.out --size # --simd

SIMD		
Matrix Size for Matrix B	Sparsity (%)	Time (s)
500	10	0.0166179
1000	10	0.126857
2000	10	1.25332

The data below has been collected from running: ./matrix_mult.out --size # --cache

Cache Optimization		
Matrix Size for Matrix B	Sparsity (%)	Time (s)
500	10	0.0681929

1000	10	0.560325
2000	10	4.66133

The data below has been collected from running: ./matrix_mult.out --size # --all

All optimization		
Matrix Size for Matrix B	Sparsity (%)	Time (s)
500	10	0.0224962
1000	10	0.134366
2000	10	1.09141

Analysis:

From looking at the various data tables, we can see that the execution time is increasing as we increase the matrix size. This does make sense, as the more elements you have in your matrix, the longer the execution time it will take. Comparing this data to the Sparse-Sparse matrix multiplication on the section of “Different size, Keep sparsities constant”, we can see that they are almost the same timing.

Dense vs Sparse Matrix Multiplication Final analysis:

Comparing the results with each other (Dense-Dense vs. Sparse-Sparse vs. Dense-Sparse), I can say that Sparse-Sparse = Dense-Sparse > Dense-Dense. Dense-Dense will take too much time to run the program as the matrix multiplication function has to transverse each element since all of the size is filled with values unlike Sparse-matrix as only few elements of the size are filled with values.

Final Analysis and Conclusions:

From running these multiple experiments, I can say that SIMD instructions are the best optimization algorithm for matrix-matrix multiplication. Comparing when using matrix size vs. sparsity %, sparsity % will result in better performances as I have already explained on why it's better due to only some elements in the size being non-zero while the rest is empty. Therefore, having a sparse matrix is better to have for data management in the matrix-matrix multiplication as it has better performance as shown from my data.