Introduction

Functional Overview

Upon launching the program, users are prompted to define the characteristics of two matrices, A and B, by specifying the number of rows and columns for each matrix. The program will then ask for the desired sparsity levels for both matrices. Sparsity refers to the proportion of zero elements within a matrix, which significantly influences the efficiency of matrix operations.

After the matrix specifications are complete, the user will select the type of multiplication operation they wish to perform. Options include:

- **Dense x Dense**: Multiplying two fully populated matrices.
- **Dense x Sparse**: Multiplying a dense matrix with a sparse matrix.
- **Sparse x Sparse**: Multiplying two sparse matrices.

Following the selection of the operation type, users can opt to apply optimization techniques to enhance performance. If the user indicates that they would like to utilize optimization, the program will provide timing results and specialized performance metrics based on the chosen optimization strategy.

If the user declines to use optimizations, they will be prompted to run experimental tests that assess various configurations and gather performance data. This experimental mode allows you to test a customized version of the multithreading optimization.

In summary, the user can construct matrices of their choice while allowing them to select the desired operation and apply various optimization techniques. Whether you choose to utilize performance tests or opt out, you will still have access to the experimental testing options.

Moreover, the program features an experimental mode that allows for custom configurations, including multi-threading. Users can specify the number of threads to utilize, along with the matrix dimensions and sparsity levels. This flexibility ensures that you can tailor your testing environment to suit your specific needs, providing a comprehensive understanding of how different factors influence matrix multiplication performance.

Software Implementation

The implementation of our software project is structured to efficiently handle matrix operations, performance optimization, and experimental techniques, particularly focusing on multithreading and SIMD (Single Instruction, Multiple Data) optimizations. To facilitate a clear understanding of the benefits and limitations of each optimization strategy, I isolated the optimization techniques and their corresponding performance tests. This approach allows for easier evaluation of the contributions each method makes to overall performance.

Source Files (.cpp):

- **cache_optimization.cpp**: Implements algorithms and functions aimed at optimizing cache usage through techniques such as blocking, enhancing overall performance in matrix operations.
- **experimental_multithreading.cpp**: Contains code for experimental implementations of multithreading techniques, leveraging parallel processing to improve the efficiency of matrix computations.
- **experimental_results.cpp**: Responsible for collecting and presenting results from various experimental runs, allowing for a comprehensive analysis of performance metrics.
- **main.cpp**: The entry point of the application, which initializes the program and orchestrates calls to functions from other files to execute tasks.
- **matrix.cpp**: Implements core functionalities related to matrix creation, manipulation, and multiplication, forming the backbone of matrix operations in the project.
- **multithreading.cpp**: Contains the implementation of multithreading techniques, utilizing concurrent processing to optimize matrix operation performance.
- **optimization_results.cpp**: Focused on presenting results from optimization experiments, this file provides insights into the effectiveness of implemented strategies.
- **performance_cache.cpp**: Measures and analyzes performance metrics specifically associated with cache optimization techniques, helping to identify areas for improvement.
- **performance_multithreading.cpp**: Evaluates the performance of the implemented multithreading techniques, offering insights into their effectiveness in speeding up matrix operations.
- **performance_simd.cpp**: Analyzes performance metrics for SIMD optimizations, highlighting the advantages of executing operations on multiple data points simultaneously.
- **performance_test.cpp**: This is for the default test which user wants no optimization.
- **simd.cpp**: Implements SIMD operations, enabling high-speed processing for large data sets, such as matrices, through parallel execution.

Header Files (.hpp):

- **cache_optimization.hpp**: Declares functions and classes used in `cache_optimization.cpp`, providing necessary interfaces for other components.
- **experimental_multithreading.hpp**: Serves as the interface for the experimental multithreading implementations, outlining functions and classes to be utilized in other source files.
- **matrix.hpp**: Declares functions and classes related to matrix operations, facilitating interaction with matrix objects across the project.
- **multithreading.hpp**: Provides declarations for multithreading functions or classes, enabling their use throughout the application.
- **simd.hpp**: Declares functions and classes related to SIMD operations, allowing for seamless integration of SIMD techniques in the project.

Makefile:

The Makefile is a critical component of the project, used by the `make` build automation tool to compile and link the application.

Project Focus

By isolating the optimization techniques cache optimization through blocking, SIMD, and multithreading I have created a structured approach that clearly demonstrates their respective benefits and shortfalls. This modular design enables focused analysis, allowing us to quantify performance improvements and identify any limitations inherent in each method. Overall, my project aims to optimize matrix operations through these techniques, ensuring reliable and efficient computation for applications in scientific computing, machine learning, and graphics processing.

Experimental Results

Experimental Results

Matrix Size	Sparsity	Dense-Dense Time (s)	Dense-Sparse Time (s)	Sparse-Sparse Time (s)	Cache Misses	Peak Memory (KB)
1000x1000	100%	10.0043	4.04466	3.20495	11,532,401	46,592
1000x1000	1%	9.68484	12.9854	14.1727	1,552,916,367	46,620
1000x1000	0.1%	10.1583	13.4563	14.0227	1,562,207,629	46,620
2000x2000	100%	86.1724	68.0895	26.7041	43,931,868	163,892
2000x2000	1%	87.0066	115.8	145.156	14,255,005,669	163,892
2000x2000	0.1%	86.6994	111.515	118.194	14,232,745,312	163,892
3000x3000	100%	292.537	273.809	90.6638	201,538,274	359,208
3000x3000	1%	296.816	394.017	471.082	61,787,202,958	359,224
3000x3000	0.1%	295.303	387.573	453.162	62,611,607,132	359,224

Optimization Results

Performance Testing Results

Size	Sparsity A	Sparsity B	Multithreading (s)	SIMD (s)	Cache Opt. (s)
(500x500)	0.01	0.01	0.360545	4.28258	2.63382
(500x500)	0.01	0.10	0.348306	3.19623	1.29654
(500x500)	0.01	0.50	0.269144	1.98647	1.27942
(500x500)	0.01	1.00	0.272285	2.02340	1.27603
(500x500)	0.10	0.01	0.269825	2.01547	1.31233
(500x500)	0.10	0.10	0.317893	2.02176	1.28699
(500x500)	0.10	0.50	0.316316	2.01932	1.30240
(500x500)	0.10	1.00	0.282250	2.01875	1.28796
(500x500)	0.50	0.01	0.320787	2.02429	1.29595
(500x500)	0.50	0.10	0.316998	2.01488	1.29176
(500x500)	0.50	0.50	0.298507	2.01407	1.29089
(500x500)	0.50	1.00	0.316375	2.02507	1.29636
(500x500)	1.00	0.01	0.286372	2.01819	1.29277

(700x700)	0.01	0.10	0.738272	5.50093	3.58538
(700x700)	0.01	0.50	0.780772	5.52039	3.60749
(700x700)	0.01	1.00	0.866841	5.63668	3.61859
(700x700)	0.10	0.01	0.735325	5.68434	3.58090
(700x700)	0.10	0.10	0.812846	5.48599	3.57938
(700x700)	0.10	0.50	0.882775	5.55139	3.63020
(700x700)	0.10	1.00	0.825516	5.48889	3.57580
(700x700)	0.50	0.01	0.879976	5.66547	3.65641
(800x800)	0.01	0.01	1.215563	7.73282	5.03254
(800x800)	0.01	0.10	1.267893	7.54321	5.12189
(800x800)	0.01	0.50	1.198765	7.88763	5.15642
(800x800)	0.01	1.00	1.182345	7.94512	5.27689
(800x800)	0.10	0.01	1.243215	7.65234	5.06254

Analysis of Experimental Results

Based on the experimental results, which explored different matrix sizes and levels of sparsity, I found several key insights into the performance characteristics of matrix operations. The tests were designed to assess dense-dense, sparse-sparse, and dense-sparse matrix multiplications under various conditions. Below, I present detailed observations segmented by the type of multiplication.

Dense-Dense Matrix Multiplication Results

Set of Experiments:

1. **Matrix Size: 1000x1000**
    - Time: 10.0043 s
    - Cache Misses: 11,532,401
    - Peak Memory: 46,592 KB
2. **Matrix Size: 2000x2000**
    - Time: 86.1724 s
    - Cache Misses: 43,931,868
    - Peak Memory: 163,892 KB
3. **Matrix Size: 3000x3000**
    - Time: 292.537 s
    - Cache Misses: 201,538,274

- Peak Memory: 359,208 KB

Discussion of Performance Changes with Matrix Size: The results show a significant increase in time taken for dense-dense matrix multiplication as matrix size increases. The execution time escalated from approximately 10 seconds for a 1000x1000 matrix to over 292 seconds for a 3000x3000 matrix. This pattern reflects the cubic growth in time complexity typical of dense matrix operations due to the increased number of operations required.

Cache misses also demonstrate a dramatic increase with larger matrices, suggesting that larger data sets exceed cache capacities more frequently, leading to performance decline. Peak memory usage rises from 46,592 KB to 359,208 KB as matrix size increases, reflecting the greater amount of data being processed.

Sparse-Sparse Matrix Multiplication Results

Set of Experiments (Varying Sparsity, Size Constant at 1000x1000):

1. **Sparsity: 100%**
   - Time: 3.20495 s
   - Cache Misses: 11,532,401
2. **Sparsity: 1%**
   - Time: 14.1727 s
   - Cache Misses: 1,552,916,367
3. **Sparsity: 0.1%**
   - Time: 14.0227 s
   - Cache Misses: 1,562,207,629

Discussion of Performance Changes with Matrix Sparsity: As the sparsity of the matrix decreases from 100% to 0.1%, the time taken for sparse-sparse matrix multiplication increases. The dense matrix (100% sparsity) required only 3.20495 seconds, while both the 1% and 0.1% sparsity matrices required significantly longer times (around 14 seconds). This trend show that sparser matrices utilize cache memory more effectively, while denser matrices suffer from cache inefficiencies.

Set of Experiments (Varying Size, Sparsity Constant at 100%):

1. **Matrix Size: 1000x1000**
   - Time: 3.20495 s
   - Cache Misses: 11,532,401
2. **Matrix Size: 2000x2000**
   - Time: 26.7041 s
   - Cache Misses: 43,931,868

3. **Matrix Size: 3000x3000**
    - Time: 90.6638 s
    - Cache Misses: 201,538,274

Discussion of Performance Changes with Matrix Size: The increase in time for sparse-sparse multiplication with larger matrix sizes remains significant but less pronounced compared to dense matrices. For instance, a 1000x1000 matrix takes around 3.2 seconds, while a 3000x3000 matrix takes approximately 90.7 seconds. The overall trend show that as the size of the sparse matrix increases, time and cache misses rise, though the impact is more gradual due to the inherent efficiency of processing sparse data.

Dense-Sparse Matrix Multiplication Results

Set of Experiments (Varying Sparsity, Size Constant at 1000x1000):

1. **Sparsity: 100%**
    - Time: 4.04466 s
    - Cache Misses: 11,532,401
2. **Sparsity: 1%**
    - Time: 12.9854 s
    - Cache Misses: 1,552,916,367
3. **Sparsity: 0.1%**
    - Time: 13.4563 s
    - Cache Misses: 1,562,207,629

Discussion of Performance Changes with Matrix Sparsity: In dense-sparse matrix multiplication, as sparsity decreases from 100% to 0.1%, the computation time increases substantially. The time taken grows from about 4 seconds for a fully dense matrix to over 13 seconds for the sparsest matrix. This trend show that as the number of non-zero elements decreases, the inefficiencies in processing a sparse matrix begin to significantly affect performance, especially when interacting with dense matrices.

Set of Experiments (Varying Size, Sparsity Constant at 100%):

1. **Matrix Size: 1000x1000**
    - Time: 4.04466 s
    - Cache Misses: 11,532,401
2. **Matrix Size: 2000x2000**
    - Time: 68.0895 s
    - Cache Misses: 43,931,868
3. **Matrix Size: 3000x3000**

- Time: 273.809 s
- Cache Misses: 201,538,274

Discussion of Performance Changes with Matrix Size: Similar to dense-dense multiplication, dense-sparse matrix multiplication shows significant performance decline as matrix size increases. The time required escalates dramatically from approximately 4 seconds for a 1000x1000 matrix to over 273 seconds for a 3000x3000 matrix, confirming the cubic time complexity characteristic of matrix operations. Cache misses follow suit, showing that larger matrices exceed cache capacities, resulting in performance drops.

Analysis of Performance Testing Results

The data indicates significant variations in execution time for matrix multiplication based on the sparsity levels and matrix sizes. The results can be summarized as follows:

1. **Impact of Sparsity**:
   - As sparsity increases (from 0.01 to 1.00), the execution times for the SIMD and Cache Optimization techniques generally show a decrease in computation time, especially for larger matrices.
   - For example, in the (500x500) matrix tests, the SIMD execution time decreases from 4.28 seconds (sparsity A = 0.01, sparsity B = 0.01) to 1.99 seconds (sparsity A = 1.00, sparsity B = 1.00), indicating that higher sparsity leads to more efficient processing with SIMD.
2. **Matrix Size Effect**:
   - Larger matrices (700x700 and 800x800) consistently exhibit higher execution times across all optimizations compared to the (500x500) matrices. For instance, the (800x800) matrix with sparsity levels 0.01 takes over 1.2 seconds with multithreading and nearly 7.73 seconds with SIMD.
   - This suggests that while optimizations are beneficial, they cannot fully offset the inherent increase in computational complexity that accompanies larger matrix sizes.
3. **Multithreading Performance**:
   - The multithreading approach tends to provide competitive execution times, especially at lower sparsity levels. For instance, in the (500x500) matrix with sparsity levels 0.01 and 0.10, multithreading consistently outperforms SIMD in execution time.
   - As sparsity increases, the performance gap narrows, suggesting that while multithreading is effective, SIMD and cache optimization techniques become more efficient in handling sparse matrices.
4. **Cache Optimization Benefits**:
   - The cache optimization method consistently results in faster execution times across different sparsities. For example, for the (500x500) matrix with sparsity

levels (0.50, 1.00), the cache optimization takes approximately 1.27 seconds, compared to 1.98 seconds for SIMD, illustrating its effectiveness in improving performance.
    - This suggests that improving memory access patterns can have a pronounced effect, especially for operations involving large and sparse datasets.

Optimization Impact

In summary, here are the findings regarding which optimization methods have the most impact on performance across different sparsity and size configurations:

- **Sparsity Levels**:
    - **Higher Sparsity**: SIMD and cache optimization techniques significantly reduce computation time. The more sparsity in the matrix, the more advantageous these optimizations become.
    - **Lower Sparsity**: Multithreading shows superior performance at lower sparsity levels due to the ability to parallelize computations effectively.
- **Matrix Sizes**:
    - **Larger Matrices**: Cache optimization consistently provides benefits over SIMD and multithreading, indicating that optimizing memory access is critical for handling larger datasets effectively.
    - **Smaller Matrices**: Multithreading may outperform other techniques due to reduced overhead in managing threads and maintaining computational efficiency.