Dense/Sparse Matrix-Matrix Multiplication Report

Yunhua Fang

October 2024

Introduction

The objective of this design project is to implement a C/C++ module that carries out high-speed dense/sparse matrix-matrix multiplication by explicitly utilizing (i) multiple threads, (ii) x86 SIMD instructions, and/or (iii) techniques to minimize cache miss rate via restructuring data access patterns or matrix compression.

Matrix-matrix multiplication is one of the most important data processing kernels in numerous real-life applications, e.g., machine learning, computer vision, signal processing, and scientific computing. This project aims to gain hands-on experience in multi-thread programming, SIMD programming, and cache access optimization. It will develop a deeper understanding of the importance of exploiting task/data-level parallelism and minimizing cache miss rate.

The implementation is able to support configurable matrix sizes that can be much larger than the on-chip cache capacity. Moreover, the implementation allows users to individually turn on/off the three optimization techniques (i.e., multi-threading, SIMD, and cache miss minimization) and configure the thread number so that users can easily observe the effect of any combination of these three optimization techniques.

Methods

The Dense/Sparse Matrix-Matrix Multiplication Program is designed to multiply two matrices, Matrix A and Matrix B, which can be either dense or

sparse. The program supports multiple optimization modes including multi-threading, SIMD, and matrix compression to enhance performance based on the characteristics of the input matrices.

Our matrix generation function generates matrix in $n \times n$ size (n is defined by the users). The integer elements in the matrix are randomly generated between 0 and 10. We tried to set a sparse matrix with $O(n)$ elements that are non-zero, but the optimizations are not obvious in the performance of sparse matrix multiplications. Hence, in the experiment, we define the matrix with 0.1% sparsity (0.1% elements in the matrix is 0) as a dense matrix and the matrix with 1% sparsity (1% elements in the matrix is 0) as a sparse matrix. The program generates two matrices based on the user's input command and then multiplies them by following the configuration. The generated matrix supports both ELLPACK for dense matrices and CSR (Compressed Sparse Row) formats for sparse matrices. ELLPACK is optimized for storing dense matrices where each row has a uniform number of non-zero elements. CSR is tailored for sparse matrices with irregular distributions of non-zero elements.

The basic multiply function is a fundamental implementation designed to perform matrix-matrix multiplication between two input matrices, matA and matB. This function is adept at handling both dense and sparse matrices by leveraging the ELLPACK format for efficient storage and access of non-zero elements. The multiplication process begins by iterating through each row of matA. For every non-zero element in a given row of matA, the function accesses the corresponding row in matB using the ELLPACK structure, which facilitates rapid retrieval of non-zero values and their associated column indices. To accumulate the results of these multiplications, an unordered_map is employed, mapping column indices to their summed product values. This ensures that only relevant non-zero computations contribute to the final result, thereby optimizing performance for sparse data scenarios. After processing all necessary elements, the function identifies the maximum number of non-zero elements present in any row of the resultant matrix, which is crucial for configuring the ELLPACK structures (values and colIndices) of the output matrix. By systematically organizing and storing the computed values, the multiply function efficiently generates the product matrix, balancing both computational accuracy and memory utilization. This approach provides a clear and effective method for matrix multiplication, particularly suited for applications where matrix sparsity can be exploited to enhance performance.

The project introduces multi-threading into the multiplication process when

enabling multi-threading, utilizing multiple CPU cores to parallelize the computation across different rows of Matrix A. By distributing the workload, the program significantly reduces the overall execution time, especially for large matrices, as multiple rows are processed concurrently. This parallelism is achieved through efficient thread management, ensuring that each thread operates independently without causing data races or synchronization bottlenecks.

On the other hand, we integrates SIMD (Single Instruction, Multiple Data) vectorization techniques using AVX2 intrinsics. SIMD allows the function to perform the same operation on multiple data points simultaneously within a single CPU instruction cycle. In SIMD version, this is manifested through the simultaneous loading, multiplication, and accumulation of multiple non-zero elements from Matrix B. By processing eight integers in parallel, it can accelerate the computation within each row, maximizing data throughput and minimizing processing latency.

We apply an optimized approach to matrix-matrix multiplication, enhancing the foundational multiply function by incorporating critical performance improvements tailored for both dense and sparse matrices. Unlike the basic multiply method, which relies on direct element access and an unordered_map for accumulating results, Optimized version leverages the CSR format to efficiently manage sparse data. This optimization begins by ensuring that both input matrices, matA and matB, are compressed into CSR format, facilitating rapid access to non-zero elements and minimizing memory overhead. The function meticulously iterates through each row of matA, and for every non-zero element within that row, it accesses the corresponding row in matB using the CSR structure. This method eliminates redundant zero computations and significantly reduces cache misses by streamlining memory access patterns.

Results

To obtain different performances under different configurations, the project will discover the multiplication of two matrices with $1,000 \times 1,000$ and $10,000 \times 10,000$. The size of the matrix exceeds the cache size, which leverages our various optimizations. For the configuration, we present the latency performance of multiplication under situations: no optimization, multi-threading, SIMD, matrix compression, and all of them together in Table 1

and Table 2, and throughput performance in Table 3 and Table 4. We use "m" to represent multi-threading enabled, "s" to represent SIMD enabled, and "o" to represent matrix compression enabled. For multi-threading, we apply 12 threads to run the program. We also record the performance of multi-threading + SIMD, multi-threading + matrix compression, SIMD + matrix compression in the test_result.txt. Dense-Dense, Dense-Sparse, and Sparse-Sparse matrix multiplications are also applied to each case of tests.

The data we used in latency tables are recorded in test_result(latency).txt. From the data, we can observe that optimizations play important roles in reducing latency. The performance is obvious on dense-dense matrix multiplication. When the program enables all optimization, the latency reduces 98% compared to the original time. Moreover, multiplication will take more benefits from SIMD and matrix compression. When the matrix gets sparser, the effectiveness of optimizations reduces. Dense-Dense matrix multiplication can leverage those optimizations. Matrix compression and optimized data access are the most effective on sparse matrix multiplication.

	Dense-Dense	Dense-Sparse	Sparse-Sparse
no opt	2.08	0.230	0.045
m	0.625	0.075	0.035
s	0.197	0.035	0.033
o	0.037	0.038	0.032
m&s&o	0.036	0.036	0.033

Table 1: Latency (s) of $1,000 \times 1,000$ Matrix Multiplication

	Dense-Dense	Dense-Sparse	Sparse-Sparse
no opt	2138.97	155.272	50.987
m	672.494	71.501	50.867
s	529.313	49.997	49.742
o	50.7472	51.626	49.567
m&s&o	51.3274	51.163	50.386

Table 2: Latency (s) of $10,000 \times 10,000$ Matrix Multiplication

In order to transparently view the effectiveness of optimizations, we also recorded the throughput (operations per second) in the multiplication. Without any optimization, the throughput is relatively low, especially for Dense-Dense and Sparse-Sparse multiplications. Multi-threading dramatically increases throughput, especially for Dense-Dense matrices, where it more than

quadruples the operations per second for both matrix sizes. SIMD optimization also contributes to higher throughput, but its impact is less than multi-threading, especially for Dense-Dense matrices. However, in combination with multi-threading, SIMD provides an additional boost. Matrix compression enhances the throughput primarily for Sparse-Sparse and Dense-Sparse cases, where compression is beneficial due to the reduced number of non-zero operations. When all optimizations are applied together, the throughput is maximized, showing the best performance across all matrix types, particularly for Dense-Dense multiplication. For single optimization enabled, SIMD provides the highest throughput in the process. The throughput gains around 12 times of the one without optimization.

	Dense-Dense	Dense-Sparse	Sparse-Sparse
no opt	8.82207e+08	9.01414e+08	8.9702e+08
m	4.03211e+09	3.92984e+09	3.93084e+09
s	1.00864e+10	8.97711e+09	9.37163e+09
o	1.9614e+09	1.96614e+09	1.99247e+09
m&s&o	1.48596e+10	1.46237e+10	1.47749e+10

Table 3: Throughput (ops/s) of $1,000 \times 1,000$ Matrix Multiplication

	Dense-Dense	Dense-Sparse	Sparse-Sparse
no opt	8.89299e+08	4.73437e+08	4.73859e+08
m	3.33103e+09	3.1027e+09	3.07218e+09
s	2.01567e+09	2.01348e+09	3.09481e+09
o	1.11837e+09	1.11599e+09	1.83268e+09
m&s&o	8.23388e+09	8.02698e+09	1.08804e+10

Table 4: Throughput (ops/s) of $10,000 \times 10,000$ Matrix Multiplication

Conclusion

The performance tests on matrix multiplication demonstrate that applying optimization techniques such as multi-threading, SIMD, and matrix compression significantly enhances both latency and throughput across various matrix types (Dense-Dense, Dense-Sparse, and Sparse-Sparse) and matrix sizes. Multi-threading consistently provides the largest performance boost, especially for Dense-Dense multiplications, where the computational load is

highest. SIMD further improves performance by leveraging data-level parallelism, and matrix compression is particularly effective for sparse matrices, reducing the number of operations needed. When all optimizations are combined (multi-threading, SIMD, and matrix compression), they yield the best results, drastically reducing computation time and maximizing throughput. Overall, the combination of these techniques proves essential for handling large-scale matrix multiplications efficiently, with the optimizations being most impactful for dense matrices in terms of speed and for sparse matrices in terms of efficiency.

Matrix compression techniques play a pivotal role in optimizing both memory usage and computational performance, particularly when dealing with large-scale matrices that exhibit significant sparsity. Sparsity, defined as the proportion of zero elements within a matrix, serves as a critical indicator for determining the applicability and effectiveness of compression strategies. Generally, matrices with sparsity levels exceeding 70% benefit substantially from compression, yielding notable reductions in memory footprint and enhancing processing speeds. In scenarios where sparsity surpasses 90%, compression becomes almost indispensable, enabling efficient storage and swift computational operations that would otherwise be prohibitively resource-intensive.

However, the decision to employ matrix compression should not be based solely on sparsity levels. Factors such as matrix size, the nature of computational operations, hardware architecture, and the specific sparsity patterns within the matrix must also be meticulously considered. Additionally, while high sparsity generally favors compression, matrices with lower sparsity (below 50%) may not experience significant benefits and could even incur additional computational overhead, rendering dense storage formats more advantageous in such cases.

In summary, leveraging matrix compression is a highly effective strategy for managing sparse matrices, provided that the sparsity levels and operational requirements align with the strengths of available compression formats. By judiciously applying multi-threading, SIMD, and matrix compression, significant enhancements in both memory efficiency and computational performance can be achieved, thereby facilitating the efficient handling of large-scale and complex matrix operations in diverse scientific and engineering applications.