Performance Evaluation of Sparse Matrix Functions

Computer Systems

Vraj Patel

2nd August 2024

1. Introduction

A **sparse matrix**, with more zero than non-zero elements, is considered sparse if non-zero elements roughly equal rows or columns. If 2/3 of elements are zeros, it's sparse [4]. This sparsity enables specialized formats and algorithms, improving performance and reducing memory needs.

In this project, I focused on three fundamental operations involving sparse matrices:

- **Addition**: The process of adding two sparse matrices element-wise.
- **Multiplication**: The multiplication of two sparse matrices. This operation is more complex than addition and can potentially result in a denser matrix.
- **Vector Multiplication**: The multiplication of a sparse matrix with a vector.

2. The code I am evaluating

My project uses a C program (also implemented using the vectorclass in C++) to create and operate on randomly generated sparse matrices, testing performance under various conditions. The program takes matrix dimensions, non-zero elements, and vector size as inputs. For instance, ./sparse_matrix 1024 1024 2000 1024 creates a 1024x1024 matrix with 2000 non-zero elements, using a 1024-sized vector for multiplication. Due to their complexity, I would not be able to code the matrix to its fullest functionality. Hence, I used a 1D array to simulate a 2D matrix structure, similar to Lab8. Elements are accessed as matrix[i * width + j].

3. Methodology (How I evaluated it)

I executed my code on a 10th gen Intel i7-1065G7 CPU with 4 cores and 8 threads, using Windows Subsystem for Linux in a virtualized environment. For perf, I used CSIL. I implemented two versions of sparse matrix operations: one with standard C code, and another with SIMD optimizations using Agner Fog's VCL in C++. I used a timing function (from Lab 8) to measure operation times for different matrix sizes. I profiled cache usage with Valgrind's Cachegrind extension and used Perf for performance metrics. The analysis focused on operation times and cache utilization. Different matrix sizes were used to test the performance in various conditions.

4. Results and Analysis

To analyze how well the code can converted to assembly by a compiler, below are the results of different optimization levels:

Optimization Level	Addition Time (ms)	Multiplication Time (ms)	Vector-Matrix Multiplication Time (ms)	Real (s) using time command (All functions)
-O3	9.80	107996.14	15.34	88.601
-O2	13.49	79785.93	21.29	88.460
-O1	10.47	100574.28	6.73	112.374
None	33.87	253413.75	14.47	278.202

The results indicate that the compiler is highly effective at translating high-level code into efficient assembly. When the optimization level is set to -O3, the addition time is the fastest among all tested levels. This suggests that the compiler is able to effectively optimize the addition operation in the high-level code to a more efficient assembly code. However, the multiplication operation is faster with the -O2 optimization level. This indicates that different optimization levels may optimize different types of operations more effectively.

Without any optimization, the performance of the operations is significantly slower showing that without the compiler's optimization capabilities, the translation from high-level code to assembly may not be as efficient. The -O1 and -O2 optimization levels provided the best balance between execution time and performance, while -O3 did not offer additional significant benefits.

Here are the things I noticed using compiler explorer (https://godbolt.org/):

- Without optimization, the assembly code is significantly longer, doesn't take advantage of advanced CPU features, and uses more stack operations (push/pop) and local variables.
- With -O1, it uses fewer stack operations and makes better use of registers. It has a better register allocation (using r12, r13, r14, r15 registers) with more efficient loop structures.
- With -O2, it reduced the loop iterations and used vectorization instructions especially in the multiplication functions. The movq instructions also reduced showing better use of registers. Bit shifts are used instead of using mul or div instructions.
- With -O3, we can see advanced loop optimizations with more sophisticated loop structures. More vectorization techniques are used with wider SIMD registers.

a. Memory Locality

Given the cache sizes of my VM, I chose test cases to target cache performance. The L1d cache, at 192 KiB, can hold 24,576 elements (192*1024/8), translating to a square matrix of approximately 156 elements per side. For the L2 cache (2 MiB), we can fit 262,144 elements, corresponding to a square matrix of about 512 elements per side. The L3 cache (8 MiB) accommodates 1,048,576 elements, allowing for a square matrix of roughly 1024 elements per side.

Below are the results that were used for performance comparisons:

Cache Level	Matrix Size	Non-Zero Elements	Addition Time (ms)	Multiplication Time (ms)	Vector-Matrix Multiplication Time (ms)
Within L1	156x156	500	0.25	38.79	0.14
Within L2	512x512	1000	3.02	915.58	0.72
Within L3	1024x1024	2000	10.12	14896.62	3.12
Exceeds L3	2048x2048	5000	33.81	243216.91	32.88

Analysis:

In the code, matrices are stored in a one-dimensional array in row-major order, meaning elements in the same row are stored next to each other in memory. Operations like addition or multiplication access elements row by row, benefiting from spatial locality. Once a row is loaded into the cache, the remaining elements can be accessed quickly. However, as the matrix size increases and exceeds the cache size, the benefits diminish. When a row no longer fits into the cache, cache misses occur, leading to slower memory accesses.

When the matrix fits within the L1 cache, operations are extremely fast, demonstrating the benefits of having data close to the CPU. As the matrix size increases to fit within the L2 cache, execution times increase noticeably: addition is about 12.08 times slower, multiplication 23.02 times slower, and vector-matrix multiplication 5.14 times slower. The L3 cache, shared across cores, has higher latency compared to L1 and L2 caches: addition is 3.35 times slower, multiplication 16.27 times slower, and vector-matrix multiplication 4.33 times slower compared to the L2 cache. When the matrix size exceeds the L3 cache, execution times increase dramatically.

Addition operations are less affected by matrix size increases and benefit more from cache locality than multiplication, which shows a dramatic rise in execution time as matrices grow due to its high memory intensity and exponential growth. Vector-matrix multiplication also increases in execution time with larger matrices but less sharply than full matrix multiplication, indicating better memory access efficiency. Accessing columns is not cache-friendly in a row-major layout because it causes many cache misses, leading to slower performance. This is likely why the multiplication operation is significantly slower than the others, especially for large matrices.

b. Cache Misses (using Cache grind)

Metric	**Value**
Total Data Refs (Dr)	197,720,663,205
D1 Cache Misses (D1mr)	8,611,998,595
D1 Miss Rate	4.4%
LL Data Misses (DLmr)	8,597,571,321
LL Miss Rate (LLd miss rate)	4.3%

The instruction cache (I1) is highly efficient, with extremely low miss rates, ensuring fast instruction fetching. The data cache (D1) has a miss rate of 4.4%, indicating that the working set exceeds its capacity, leading to frequent removals. The last level cache (LL) shows a miss rate of 1.5%, suggesting it cannot fully contain the working set, resulting in main memory accesses. High miss rates in both D1 and LL caches indicate poor spatial locality, as data access patterns are not efficiently utilizing cache line sizes. (Using Perf, The L1 cache miss rate is high at 52.07%, which suggests that the code is not optimally utilizing the L1 cache.)

Based on the results of the output, the function **multiply_sparse_matrices** had the highest instruction references and data references. It accounts for 99.88% of all instruction references and 99.87% of data references. 99.96% of all L1 data cache misses occur in this function. Despite its high data cache miss rate, the function has a good instruction cache performance, contributing to only 0.32% of L1 instruction cache misses.

c. Branch Predictability (using Perf)

The perf metrics show 8.63 billion branch instructions with 4.46 million misses (0.05% rate) over 27.5 seconds. The branch misprediction rate of 0.05% is extremely low, indicating excellent branch prediction performance.

Branch Prediction Accuracy=(Branch Instructions−Branch Misses)/Branch Instructions ×100
= ((8,630,943,240-44,458,845)/ 8,630,943,240) x 100 = 99.49%

The branch predictor's 99.49% accuracy shows its effectiveness in predicting most branches, reducing pipeline stalls, and ensuring smooth CPU execution.

Branch Penalty = 95,455,683 cycles / 4,470,941 mispredictions = **21.35 cycles/mispredictions**

This small penalty demonstrates that branch mispredictions do not significantly impact overall execution time, accounting for only a minor fraction of the total runtime. Depending on the machine's clock speed, the time penalty per branch misprediction is relatively low (around 4.36 ns on my machine).

Although the code is written efficiently using a 1D array, it does not utilize actual sparse matrix techniques due to the project's nature, so it's uncertain whether the branch misprediction will be high or low. With its full functionality, the number should be relatively higher.

d. SIMD (Single Instruction, Multiple Data) Instructions

I’m using Agner Fog’s Vector Class Library for SIMD optimizations in sparse matrix operations. It’s great for sparse matrices with many zeros, allowing efficient memory use and parallel operations for speedup. It also enables comparison with the C implementation. Below are the speedup results comparing C and SIMD times using the Linux time command (speedup = C time / SIMD time):

Matrix Size	Add. Speedup	Mult. Speedup	Vec-Mult. Speedup	Real (s)
1000x1000	1.56	3.04	0.48	1.68
2000x2000	1.32	2.30	1.33	2.46
3000x3000	4.40	3.47	2.09	3.74

The SIMD implementation consistently outperforms the standard C implementation across all matrix sizes and operations, especially in matrix multiplication. While C is slightly faster for vector multiplication with the smallest matrix size, SIMD is better with larger matrices, achieving speedups of 3x to 3.7x. SIMD also shows better memory locality and efficient use of L1/L2/L3 caches. Using perf, SIMD reduced the number of cycles, improved the instruction-per-cycle ratio, and had a lower branch misprediction rate (0.06%) compared to C (0.05%). Branch misses were significantly lower in SIMD (1.46 million) versus C (4.46 million). Overall, SIMD offers faster processing and better performance.

From these results, I **learned** that SIMD allows the CPU to perform the same operation on multiple data points simultaneously. For example, in matrix multiplication, SIMD can process multiple elements in parallel, speeding up computation and reducing the number of operations needed. By processing multiple data elements with a single instruction, SIMD decreases the total number of instructions executed, reducing overhead from instruction fetching. SIMD instructions can move and process more data per clock cycle compared to scalar instructions, reducing pipeline stalls and utilizing the vector registers more effectively. This leads to better resource utilization and potentially reduces the need for frequent memory access. With optimizations, SIMD performed even better.

5. Conclusion (What I learned about the code)

I chose to work with sparse matrices due to their relevance in exploring how performance differences with different element sizes. This project has provided me with the following learning about my code:

- Different optimization levels significantly impact performance, with -O2 and -O3 providing the best results for different operations to convert it to assembly.
- The size of the matrices relative to the cache sizes impacts the performance of sparse matrix operations, highlighting the importance of memory locality.
- Conditional logic in sparse matrix operations affects performance due to branch predictability.
- The use of SIMD instructions through the VectorClass library led to significant performance improvements, particularly for matrix multiplication. This demonstrated the power of SIMD for operations on large data sets where parallelism can be used.

Overall, this project highlighted the importance of considering hardware characteristics, compiler optimizations, and algorithm design in writing efficient code. It showed that with careful design and optimization, it is possible to significantly improve the performance of computationally intensive operations on sparse matrices.

References

[1] Fog, Agner. *VCL C++ Vector Class Library Manual*. 7 Aug. 2022.

[2] Gupta, Vasu. "Difference between Spatial Locality and Temporal Locality." *GeeksforGeeks*, 12 June 2020, www.geeksforgeeks.org/difference-between-spatial-locality-and-temporal-locality/.

[3] "How Do You Measure the Effect of Branch Misprediction?" *Stack Overflow*, stackoverflow.com/questions/2877961/how-do-you-measure-the-effect-of-branch-misprediction.

[4] Karabiber, Faith. "Sparse Matrix." *Learn Data Science*, www.learndatasci.com/glossary/sparse-matrix/#:~:text=A%20sparse%20matrix%20is%20a.

[5] "Sparse Matrix and Its Representations | Set 1 (Using Arrays and Linked Lists)." *GeeksforGeeks*, 20 July 2017, www.geeksforgeeks.org/sparse-matrix-representation/.

The logic of the C code is obtained from Geeksforgeeks (Reference 5)