Table of contents

Introduction:

The goal of Task 3 is to profile and optimize the performance of the Sparse Matrix-Vector Multiplication (SpMV) implementation, following the benchmarking done in Task 2. This task focuses on identifying performance bottlenecks in the code and improving its efficiency, especially through the application of vectorization and better memory management practices.

To begin, I have used the **GCC compiler** with specific optimization flags to compile the code. The key flags used were:

- **-Ofast**: This optimization flag enables all optimizations available in **-O3** (such as inlining, loop unrolling, and vectorization) and includes some additional approximations for performance improvement. These optimizations are designed to enhance the execution speed of the code, though they may come at the cost of precision in some cases.
- **-ftree-vectorize**: This flag enables **autovectorization**, a process where the compiler automatically converts certain loops into SIMD (Single Instruction, Multiple Data) instructions. SIMD instructions allow the CPU to process multiple data points simultaneously, which can significantly accelerate the execution of parallelizable tasks, such as SpMV.

I have used exclusively **Intel VTune** to gain insights into performance bottlenecks and areas for improvement.

In this report, I will present an analysis of various performance aspects, including:

1. **Vectorization**: Assessing whether the compiler automatically vectorizes the critical parts of the code and exploring ways to assist the compiler in achieving better vectorization if necessary.
2. **Cache Behavior**: Examining how the program's access patterns affect cache performance.
3. **Memory Behavior**: Examining how the program handles memory allocation, deallocation, and access patterns. I will investigate whether the memory is managed efficiently, whether there are any memory leaks.

Finally, the report will evaluate the **performance improvements** achieved through these optimizations. I will measure the speedup obtained from the changes and discuss the effectiveness of the optimizations.

Analyse vectorization :

Vectorization : 6.1% of Packed FP Operations

Instruction Mix:

SP FLOPs:	0.0% of uOps
DP FLOPs:	1.6% of uOps
x87 FLOPs:	0.0% of uOps
Non-FP:	98.4% of uOps
FP Arith/Mem Rd Instr. Ratio:	0.080
FP Arith/Mem Wr Instr. Ratio:	0.144

Top Loops/Functions with FPU Usage by CPU Time

This section provides information for the most time consuming loops/functions with floating point operations.

Function	CPU Time	% of FP Ops	FP Ops: Packed	FP Ops: Scalar	Vector Instruction Set	Loop Type
__random	1.750s	1.0%	0.0%	100.0%		
[Loop@0x1892e88 in func@0x1892e30]	0.110s	20.3%	100.0%	0.0%	AVX(256); FMA(256)	
rand	0.098s	1.4%	0.0%	100.0%		
[Loop at line 65 in my_sparse_CSR]	0s	7.2%	0.0%	100.0%		
[Loop at line 45 in my_sparse_COO]	0s	3.6%	0.0%	100.0%		
[Others]	0s	30.4%	0.0%	100.0%		

*N/A is applied to non-summable metrics.

We can see that there are multiple loops with FP Ops Scalar, meaning that there is autovectorization being used with them.

- random is from the library, therefore I can't improve it in any way.
- loop@0x1892e88 in func@0x1892e30 is from libopenblas, therefore I can't improve it in any way even though it hasn't been vectorized.
- rand is from the library therefore I can't improve it in any way.

Successfully vectorized loops:

- Loop at line 65 in **my_sparse_CSR**:

```
60 int my_sparse_CSR(CSR *csr, double vec[], double result[]) {
61 // Go through all rows
62 for (unsigned int i = 0; i < csr->size_row_offsets - 1; i++) {
63     result[i] = 0.0;
64     // Go through all columns of each row
65     for (unsigned int j = csr->row_offsets[i]; j < csr->row_offsets[i + 1]; j++) {

66     result[i] += csr->values[j] * vec[csr->column_indices[j]];
67     }
68 }
69 return 0;
```

Loop at line 45 in **my_sparse_COO**:

```
41 int my_sparse_COO(COO *coo, double vec[], double result[]) {
42 for (unsigned int i = 0; i < coo->size_indices; i++) {
43     result[i] = 0.0;
44 }
45 for (unsigned int i = 0; i < coo->size_values; i++) {
46     result[coo->row_indices[i]] += coo->values[i] * vec[coo->column_indices[i]];
47 }
48 return 0;
```

The autovectorization of the compiler is successful.

Analyse cache behavior :

In **my_CSC**, the following code uses a column-major order, whereas C is a row-major language.

```
for (unsigned int j = 0; j < n; j++) {
    csc.column_offsets[j] = buffer;
    for (unsigned int i = 0; i < n; i++) {                      0.3%    0.056s
        if (mat[i * n + j] != 0) {                              19.2%   3.184s
            buffer++;
        }
    }
}
buffer = 0;
for (unsigned int j = 0; j < n; j++) {
    for (unsigned int i = 0; i < n; i++) {                      0.3%    0.056s
        if (mat[i * n + j] != 0) {                              19.6%   3.246s
            csc.row_indices[buffer] = i;                        0.7%    0.108s
            csc.values[buffer] = mat[i * n + j];                0.5%    0.088s
            buffer++;                                           0.2%    0.032s
        }
    }
}
```

By changing the code to use a row-major order, we can better exploit spatial locality which should improve the performance. This is the updated code :

```
    for (unsigned int j = 0; j < n; j++) {
        csc.column_offsets[j] = buffer;
        for (unsigned int i = 0; i < n; i++) {                  0.2%    0.032s
            if (mat[j + i * n] != 0) {// Row major              19.3%   3.148s
                buffer++;
            }
        }
    }
    buffer = 0;
    for (unsigned int j = 0; j < n; j++) {
        for (unsigned int i = 0; i < n; i++) {                  0.6%    0.104s
            if (mat[j + i * n] != 0) { // Row major             18.8%   3.064s
                csc.row_indices[buffer] = i;                    0.8%    0.132s
                csc.values[buffer] = mat[j + i * n];            0.9%    0.140s
                buffer++;                                       0.2%    0.040s
            }
        }
    }
    csc.column_offsets[n] = buffer;
    return csc;
}
```

It doesn't seem that this has improved much, which means that the compiler likely optimizes the code already.

Analyze memory behavior:

We observe on the following graph that a significant amount of memory was allocated but not released afterward.

Function / Function Stack	Allocation/Deallocation Delta ▼	Allocation Size	Deallocation Size	Allocations	Module
▶ func@0x39b9f0	939.5 MB	939.5 MB		7	libopenblas.so.0
▶ convert_dense_to_COO	429.4 MB	429.4 MB		3	spmv
▶ convert_dense_to_CSR	322.1 MB	322.1 MB		3	spmv
▶ convert_dense_to_CSC	322.1 MB	322.1 MB		3	spmv
▶ main	134.2 MB	2.3 GB	2.1 GB	5	spmv
▶ printf	4.1 KB	4.1 KB		1	spmv
▶ func@0x231c0	4 KB	4 KB		8	libgfortran.so.5
▶ func@0x24030	1.5 KB	1.5 KB		6	libgcc_s.so.1
▶ func@0x23130	1.4 KB	1.4 KB		6	libgfortran.so.5
▶ __register_frame	576 B	576 B		12	libgcc_s.so.1

For **func@0x39b9f0**, no improvements are possible because it belongs to CBLAS.

Source Line ▲	Source	Allocation Size: Total	Allocation Size: Self	Deallocation Size: Total
102				
103	`cblas_dgemv(CblasRowMajor, CblasNoTrans, size, size, 1.0, mat, size, vec, 1, 0.0, refsol, 1);`	134.2 MB	134.2 MB	

However, I can improve the memory management for the **convert_dense_to_COO**, **convert_dense_to_CSR**, and **convert_dense_to_CSC** functions. It appears that the **COO**, **CSR**, and **CSC** structures were not being properly released in **spmv.c.** By ensuring these structures are properly freed, I achieved the following result:

Function / Function Stack	Allocation/Deallocation Delta ▼	Allocation Size	Deallocation Size	Allocations	Module
▶ func@0x39b9f0	939.5 MB	939.5 MB		7	libopenblas.so.0
▶ main	134.2 MB	2.3 GB	2.1 GB	5	spmv
▶ printf	4.1 KB	4.1 KB		1	spmv
▶ func@0x231c0	4 KB	4 KB		8	libgfortran.so.5
▶ func@0x24030	1.5 KB	1.5 KB		6	libgcc_s.so.1
▶ func@0x23130	1.4 KB	1.4 KB		6	libgfortran.so.5
▶ __register_frame	576 B	576 B		12	libgcc_s.so.1
▶ convert_dense_to_CSR	0 B	322.1 MB	322.1 MB	3	spmv
▶ convert_dense_to_CSC	0 B	322.1 MB	322.1 MB	3	spmv
▶ convert_dense_to_COO	0 B	429.4 MB	429.4 MB	3	spmv

That's a lot better, after these changes, there are no memory leaks remaining that I can address.

Speepdup and Conclusion

I got the following results:

	Before Optimization	After Optimization	Speed Up
Time CBLAS(ms)	109,25	110	0,9931818182
Time Matrix-Vector product(ms)	295,5	293,75	1,005957447
TIme COO(ms)	62	61,75	1,004048583
TIme CSR(ms)	30,75	27,5	1,118181818
Time CSC(ms)	26,75	26	1,028846154
Total Time(ms)	**524,25**	**519**	**1,010115607**

Limited improvement was observed, as the compiler already optimizes effectively.

This is likely due to the use of the **-ftree-vectorize** and **-Ofast** compiler flags, which significantly enhance performance. While these optimizations may slightly reduce accuracy, this trade-off is acceptable for the sparse matrix multiplication task, where precision is less critical than execution speed.