SPMV Task 3: Performance Analysis

Introduction

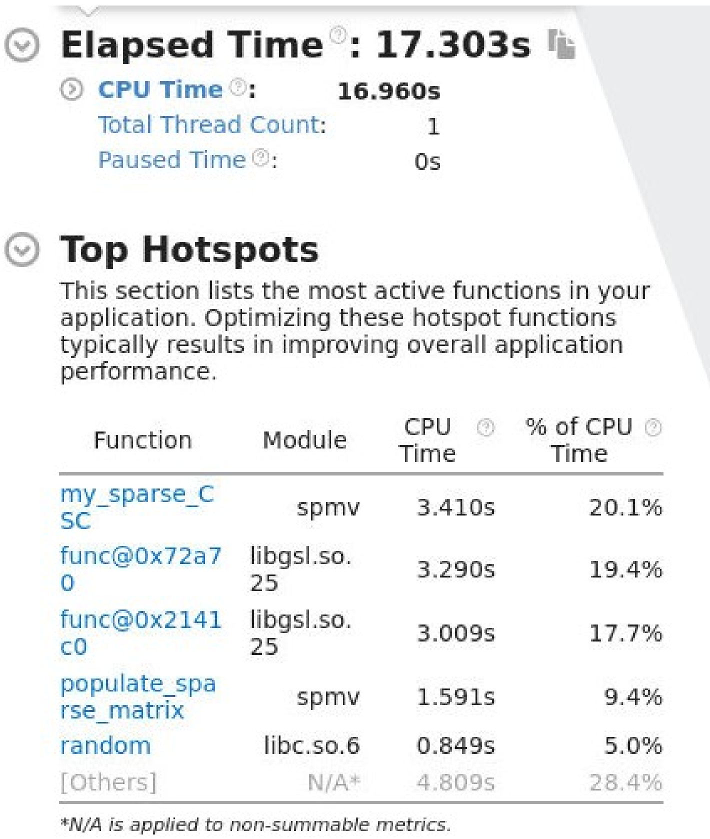
To prepare the code for analysis, I removed functions which were used with ICC (Intel C Compiler) in Task 2, as they were not required for this task. The primary objective of this work is to analyze the performance of Sparse Matrix-Vector Multiplication (SpMV) with a focus on two key aspects:

1. **Vectorization Analysis**
    - Determine if the relevant parts of the routines are being autovectorized by the compiler.
    - Identify reasons for lack of autovectorization, if applicable.
    - Explore ways to assist the compiler in achieving autovectorization.
2. **Memory and Cache Behavior Analysis**
    - Assess whether heap usage is being managed efficiently (e.g., avoiding memory leaks and ensuring proper allocation sizes).
    - Analyze the memory access pattern in the code to understand its impact on cache performance, focusing on spatial and temporal locality.

The analyses will be conducted using Intel VTune Profiler.

Hotspots Analysis

As the first step, I performed a **hotspots analysis** of my code to identify the most computationally intensive sections. This serves as a baseline for further optimization and performance tuning.



Elapsed Time: 17.303s

CPU Time:	16.960s
Total Thread Count:	1
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
my_sparse_CSC	spmv	3.410s	20.1%
func@0x72a70	libgsl.so.25	3.290s	19.4%
func@0x2141c0	libgsl.so.25	3.009s	17.7%
populate_sparse_matrix	spmv	1.591s	9.4%
random	libc.so.6	0.849s	5.0%
[Others]	N/A*	4.809s	28.4%

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

The function that consumes the most CPU time is `my_sparse_CSC`, followed by two functions from a library. Since the library functions are external, there is nothing we can optimize there.

For the `populate_sparse_matrix` function, its purpose is to populate the sparse matrix, and there is nothing significant to optimize in this routine.

```
  for (unsigned int i = 0; i < n * n; i++) {                 0.4%    59.99
    if ((rand() % 100) / 100.0 < density) {                  5.5%   929.24
      // Get a pseudorandom value between -9.99 e 9.99
```

Additionally, `random`, which is also from a library, cannot be optimized either.

Let's focus on the `my_sparse_CSC` function.

```
    // Iterate over columns for CSC format
    for (unsigned int j = 0; j < n; j++) {
        sparse[j].col = k;  // Column pointer: Start of each column in `val` and `row`
        for (unsigned int i = 0; i < n; i++) {                                          0.6%
            if (mat[i * n + j] != 0) {                                                  17.6%
                sparse[k].row = i;            // Store the row index for the non-zero element   1.6%
                sparse[k].val = mat[i * n + j]; // Store the non-zero element itself    0.2%
                k++;                                                                    0.1%
            }
        }

    }
```

It seems to consume a lot of time due to the `mat[i * n + j]` operation, where we traverse the matrix column by column instead of row by row. In C, matrices are stored in row-major order (like a flattened table), so accessing elements row by row would typically be more efficient. Unfortunately, since this function is responsible for creating the CSC (Compressed Sparse Column) format matrix, column-wise traversal is required, and we cannot change this access pattern without altering the matrix format.

Memory Consumption

Memory Consumption

Analysis Configuration | Collection Log | Summary | Bottom-up

INTEL VTUNE P

Elapsed Time: 17.616s

Allocation Size: 18.2 GB
Deallocation Size: 2.5 GB
Allocations: 68
Total Thread Count: 1
Paused Time: 0s

Top Memory-Consuming Functions

This section lists the most memory-consuming functions in your application.

Function	Memory Consumption	Allocation/Deallocation Delta	Allocations	Module
main	15.0 GB	12.9 GB	7	spmv
func@0x2bc1e0	1.2 GB	1.2 GB	9	libopenblas.so.0
gsl_spmatrix_alloc_nzmax	1.1 GB	751.7 MB	14	libgsl.so.25
func@0x214a50	859.0 MB	859.0 MB	1	libgsl.so.25
gsl_block_alloc	262.1 KB	0.0 B	2	libgsl.so.25
[Others]	83.9 KB	18.2 KB	35	N/A*

N/A is applied to non-summable metrics.

In the column "Allocation/Deallocation Delta," we can see that a significant amount of memory is not being deallocated. This indicates that we need to review the code to ensure proper deallocation of these data.

Top Memory-Consuming Functions

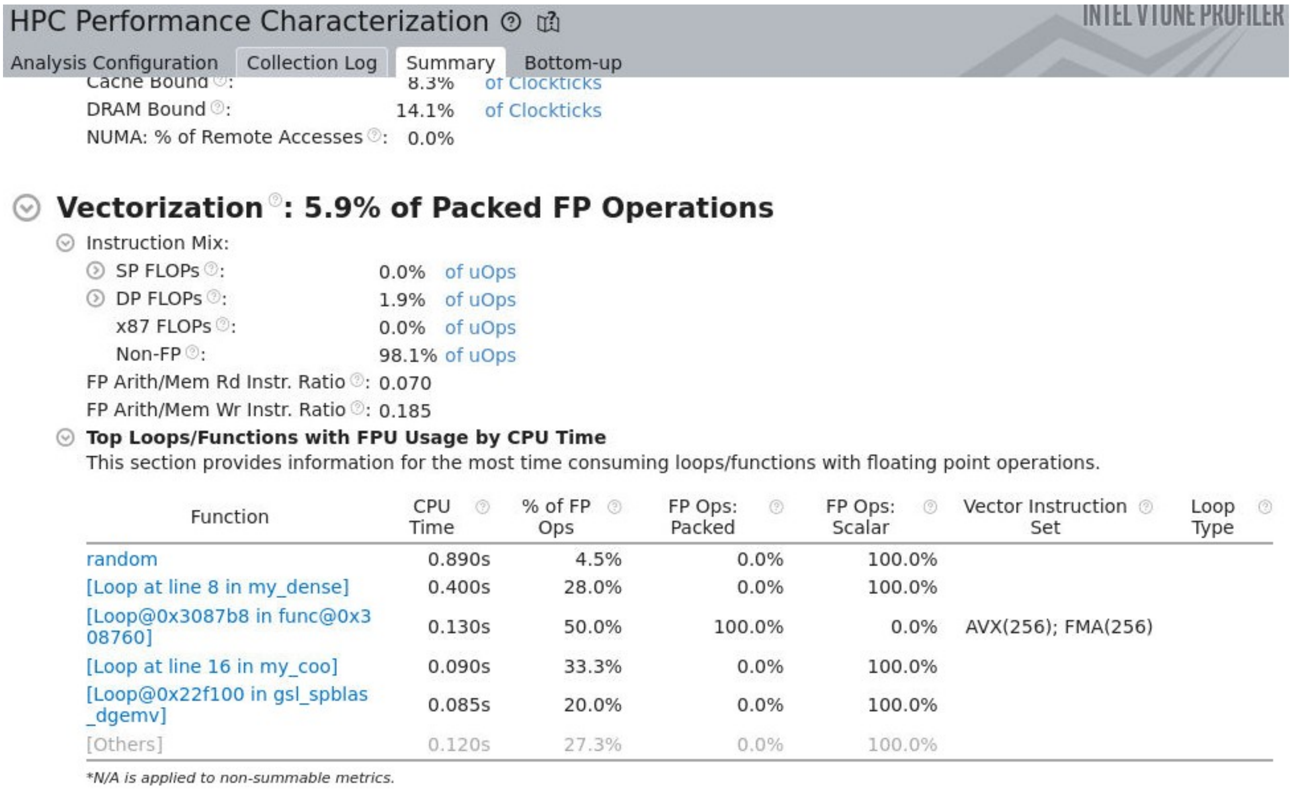
This section lists the most memory-consuming functions in your application.

Function	Memory Consumption	Allocation/Deallocation Delta	Allocations	Module
main	15.0 GB	0.0 B	7	spmv
func@0x2bc1e0	1.2 GB	1.2 GB	9	libopenblas.so.0
gsl_spmatrix_alloc_nzmax	1.1 GB	0.0 B	14	libgsl.so.25
func@0x214a50	859.0 MB	0.0 B	1	libgsl.so.25
gsl_block_alloc	262.1 KB	0.0 B	2	libgsl.so.25
[Others]	83.9 KB	18.2 KB	35	N/A*

N/A is applied to non-summable metrics.

After improving the code, almost all the issues were resolved. The missing `free()` function calls were added, addressing most of the memory leaks. However, there is still one function, `func@0x2bc1e0`, which does not deallocate memory. Unfortunately, I don’t know which function this refers to since its name doesn’t refer directly to it.

Vectorization Analysis


HPC Performance Characterization

Analysis Configuration | Collection Log | Summary | Bottom-up

INTEL VTUNE PROFILER

Cache Bound: 8.3% of Clockticks

DRAM Bound: 14.1% of Clockticks

NUMA: % of Remote Accesses: 0.0%

Vectorization: 5.9% of Packed FP Operations

Instruction Mix:

SP FLOPs: 0.0% of uOps

DP FLOPs: 1.9% of uOps

x87 FLOPs: 0.0% of uOps

Non-FP: 98.1% of uOps

FP Arith/Mem Rd Instr. Ratio: 0.070

FP Arith/Mem Wr Instr. Ratio: 0.185

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	0.890s	4.5%	0.0%	100.0%		
[Loop at line 8 in my_dense]	0.400s	28.0%	0.0%	100.0%		
[Loop@0x3087b8 in func@0x308760]	0.130s	50.0%	100.0%	0.0%	AVX(256); FMA(256)	
[Loop at line 16 in my_coo]	0.090s	33.3%	0.0%	100.0%		
[Loop@0x22f100 in gsl_spblas_dgemv]	0.085s	20.0%	0.0%	100.0%		
[Others]	0.120s	27.3%	0.0%	100.0%		

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

The vectorization efficiency is reported to be only 5.9%.

Despite using the compiler flags `-g -c -Wall -Wextra -Ofast -ftree-vectorize`, the low vectorization rate persists. The reason for this behavior remains unclear and requires further investigation. This issue was already observed in Task 2, where different optimization flags were tested to evaluate whether vectorization improved execution speed. However, with GCC, no significant change in execution time was noted.