Part 1: Sparse Matrix Matrix Multiplication

Effects of Sampling Percentage(Sparsity)

The srPercentage determines the sparsity of the matrix in this lab. It first divides its value by 100, then pass to the samplingDense function as the sampling rate which is the function that will convert a dense matrix into sparse matrix. The sparsity of the matrix will based on the passed 1./samplingRate. For example, if the sampling percentage is 20, samplingRate = 20/100 = 0.2, the stride value then becomes 1./0.2 = 5. The function will only keep the index that can be divisible by this stride value and set the others to 0. By sweeping the sampling percentages, I found that due to floating point calculations, the stride value would actually become 4 when the sampling percentage is 20 and when the value of sampling percentage is over 50, the stride values would become 1 which it will keep the original dense matrix. The sampling percentage samples use in part 1 is shown in the graph below with the corresponding stride values.

Sampling_Percentage	Index_point_to_set
1	100
2	50
3	33
5	19
10	9
15	6
18	5
20	4
30	3
50	2
60	1

Table 1 Sampling Percentage Analysis

Optimization (Tiling + Vectorization) Strategy

I found that tilling in the Spmm multiplication doesn't improve the performance as much as the Gemm multiplication due to the irregular Access patterns of the CSR format. It is hard to predict tile sizes based on distribution of non-zero values. If the tile does not contain enough non-zero values, many cache lines will remain unused, which will lead to inefficient cache utilization. A simple T1 cache profile experiment is done with sampling percentage equal to 3. As we observe from the graph, a small cache line would result in more cpu time due to the unnecessary unrolling of zero values. And as the cache size reach 1024, the cpu time would perform stable.

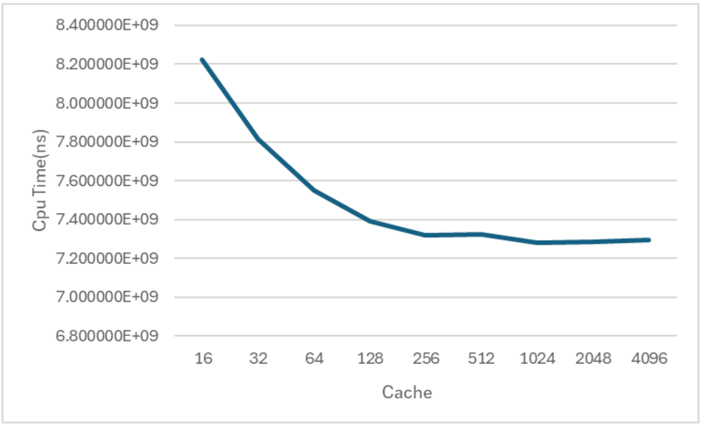

Figure 1 sweeping spmm tile size for 4k by 4k matrix

Vectorization is done by leveraging the CSR data structure which access only non-zero elements from the matrix A. Below figure illustrates the basic idea of performing vectorization. The Ap is the rowPtr which stores the index of the corresponding starting row of Ai. The vectorization is done by incrementing the Ap[i] till it reaches the next row Ap[i+1]. A vectorization size of 4x8 is chosen based on previous lab analysis by best utilizing the AVX instructions and YMM registers. And also to leverage the spatial locality, row by row matrix multiplication is used similar to lab1. For each iteration of j from below example, the Ax[j] will be extracted which represents the non-zero element in the sparse matrix A and put this value into a YMM register. The Ai[j] value corresponds the row index where needs to multiply from the matrix B for this non-zero element. By properly loading the next 4x8 elements from that Ai[j] row from matrix B then we can perform the vectorization multiplication of Ax[j] * Ai[j]. Finally, the vectorization multiplication results will be added to the corresponding C[i] index for that row.

It is also important to choose the correct vectorization boundary and handle different matrix sizes. The vectorization boundary is set to the minimum of Ap[i+1] −Ap[i] and n, ensuring that vectorization remains within the limits of each row boundary. Any remaining columns after vectorization would also be added to the specific C[i][bnd] index. This vectorization technique can best utilize the CSR data structure which only process the non-zero elements from sampling A matrix then multiply the corresponding row from matrix B. It also utilizes the spatial locality to access the matrix B row by row which we would expect to see a good performance improvement.

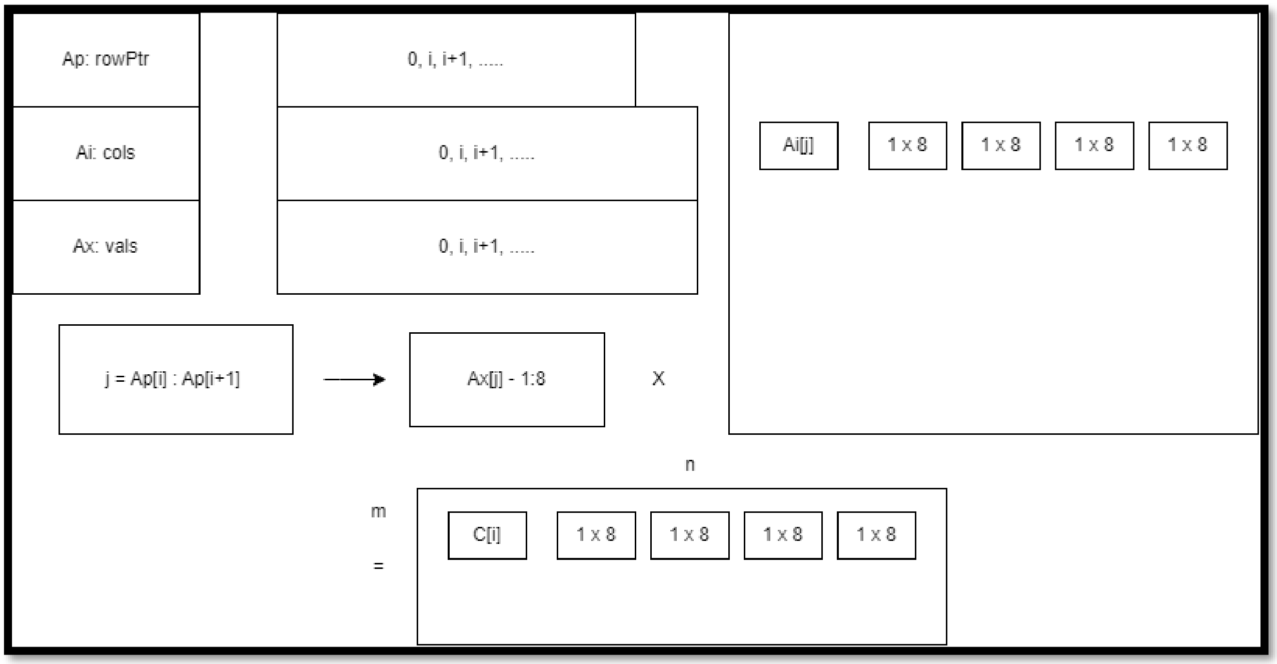

Figure 2 spmm vectorization implementation flowchart

Execution Time & Sparsity Analysis

By profiling 4 different matrix multiplication techniques over different sampling percentages. I obtained the following graph. From the graph, we can analyse that for the GEMM from lab1, the sampling percentage didn't really affect the performance. This is reasonable because in GEMM, we didn't do specific optimizations on zero elements which it will lead to the same performance by accessing each of the element from matrix A. For the baseline spmm matrix multiplication, we notice that the performance is pretty good when the sampling percentage is very low (below 5) and has a better performance than the

GEMM, but as sampling percentage increases, more non-zero elements get increased, the performance starts deceasing and became very worse for high density. An interesting observation I found that is for the even number of index points for the baseline_spmm, the even number of stride values (when sampling percentage is 18 and 30) actually perform better than the previous stride value. And it turns out that the even number of stride values have few caches misses which the way to arrange stride values can actually impact the cache utilization.

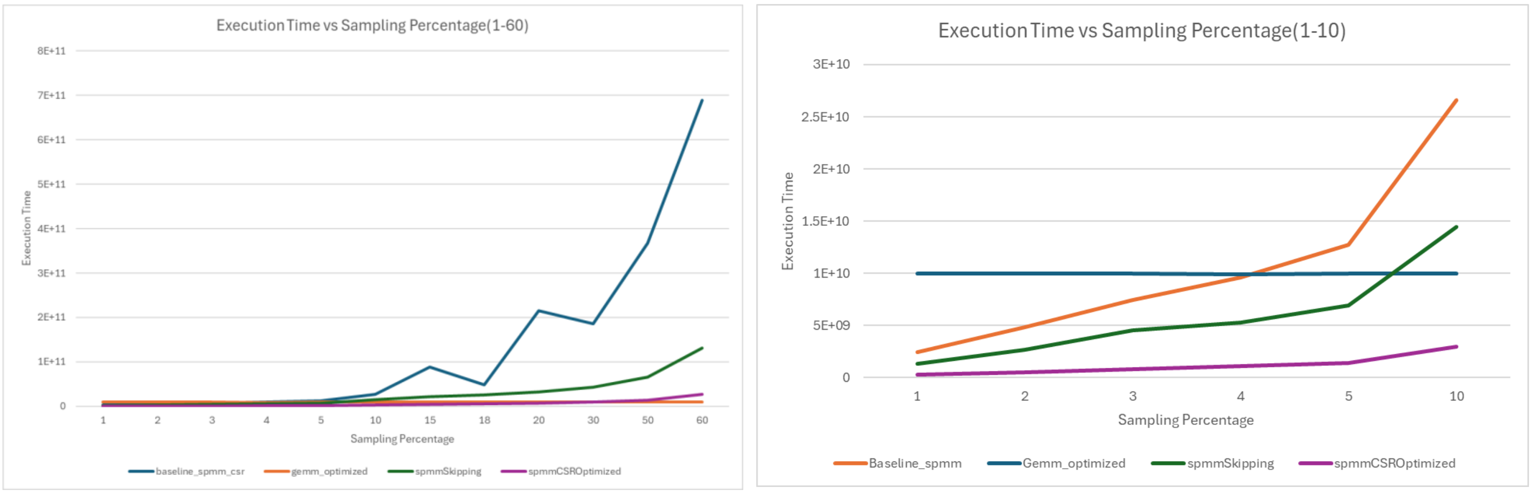

Figure 3 Execution Time vs Sampling Percentage for various matrix matrix implementations

For the spmmSkipping multiplication, I simply ignore any zero elements which it saves lots of unnecessary memory accesses on those values and we can see that from the green line the performance is a lot better compared to the baseline_spmm, and when the sparsity is low (sampling percentage < 6), the performance is better than the GEMM multiplication. Another possible way to further optimize GEMM on sparse matrix is that if there are any zero values in the matrix, for those zero values, we can simply skip the operation to avoid further memory access on each matrix.

Finally, for the spmmCSROptimized, the performance stays the best until the sampling percentage is greater than 30. Compared to the baseline Spmm multiplication, by utilizing the CSR data structure and vectorization techniques, and also performing row by row multiplication by improving the spatial locality. The performance gets a dramatic improvement when handle sparse matrix. However, it is also important to note that once the sampling percentage over 30 (stride 3), GEMM multiplication still provide the best performance. Therefore, when performing the matrix multiplication, it is important to understand the density or sparsity of the matrix to choose the best technique for the best performance.

Part 2: Sparse Matrix Vector Multiplication

Optimization (Tiling + Vectorization) Strategy

In sparse matrix-vector multiplication (SpMV), I focused on tiling only the m-dimension. Since the non zero elements in matrix A are not always going to occur consecutively, the memory access of vector B along the n dimension is non contiguous. Tiling the n-dimension would be challenging and yield minimal benefit, as the irregular access patterns limit opportunities for spatial locality. However, I can tile the m dimension of the sparse matrix as the CSR format re-writes the non-zero elements in a vector where the spatiality locality can be exploited.

Various factors will affect the "ideal" tilting size that minimizes the cache misses and reduces the execution time. The tile size will change for different sizes & sparsity of matrixes, and even when I vectorize the tiled the code. With that in mind, I decided to first vectorize the tiled SpMV code before finding the ideal tile size. I decided to look for the ideal tile size for matrixes of 4k by 4k dimension only. This is because larger matrixes will show a greater benefit from tiling and vectorization and the results can be easier to compare to see the potential improvement.

In terms of vectorizing, I know that vectorizing both the i and j loops are simply not possible. I can only vectorize the j loop (columns). Similar to gemv, vectorizing across rows would mean trying to compute multiple elements of the resultant vector simultaneously. This would require multiple partial sums for each row. The memory and storage overhead required for this implementation is not worth it. In addition, there is an irregular number of nonzero elements per row. So, it is simply not possible to vectorize across rows as this would mean handling rows of various lengths. In the code, I had to manually gather the corresponding 8 elements of vector b using the column indices and then store them in the AVX register. This is due to the non-consecutive nature of the vector b memory accesses.

To further optimize the code, I decided to unenroll the code by a factor of 4. This helps to reduce the loop overhead (condition checking, counter incrementing etc). Enrolling the loop by 4 also allows the AVX registers to compute 8 numbers per row within a single iteration. This will increase the total number of floating-point operations per loop iteration from 1 to 32 (8 elements per AVX register x 4). This allows for a greater level of parallelism which can decrease the execution time. The algorithm used for SpMVVec can be seen below:

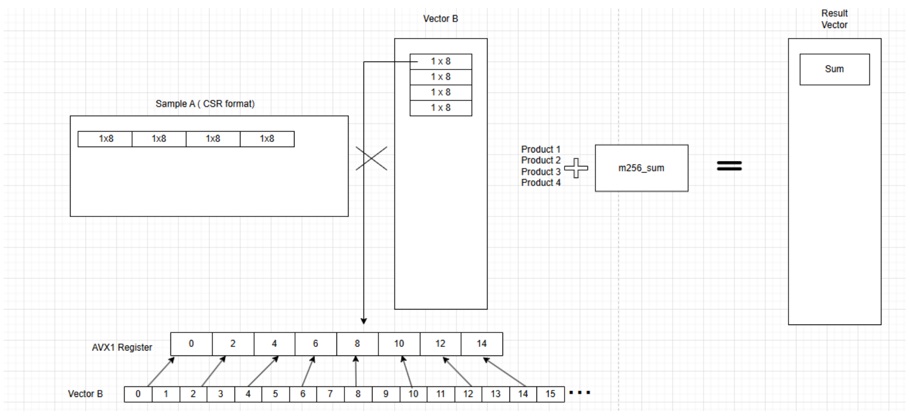

Figure 4Flowchart showing spmvec implementation

To find the ideal tile size of the tiled + vectorized SpMV code, I sweep through tiles sizes of 16 – 8000 for various sparsity levels on a 4k by 4k matrix. I noticed that the tile size has minimal impact on the CPU time for matrixes that are more than 50% dense. The reason for this will be explained more in depth in the sparsity comparison. With this in mind, I decided to focus on finding the ideal tile size for matrixes of a small density (20%). The results can be seen below:

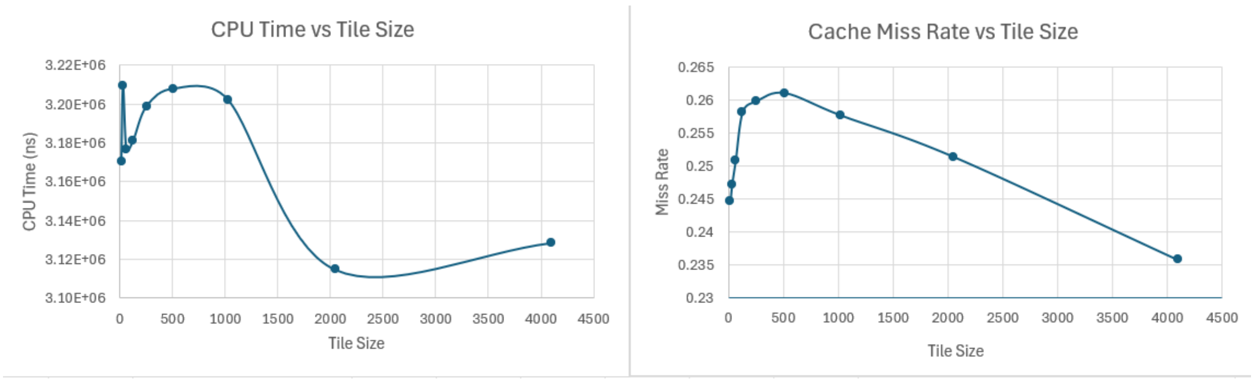

Figure 5 Tile size sweeping for a 4k by 4k matrix with a density of 20%

It is clear that larger tiles sizes (2048-4096) perform much better than smaller tiles sizes. This is because when the tile sizes are too small, there is additional loop overhead which can mitigate the benefits of tiling. Once the tile size increases to a size that maximizes the L1 cache usage, there are fewer cache misses and thus a reduce in execution time. Although not seen in this graph, if we continue to increase the tile size past 4096, the cpu time and cache miss rate will increase. For larger tile sizes, this will result in a cache overflow as data needs to be fetched from the lower level cache to L1. This is costly for execution time as lower-level access are further from the main memory so they take up significantly more time.

Execution Time & Sparsity Analysis

The effectiveness of SpMV in reducing execution time relies heavily on the matrix's sparsity as seen in the graph below:

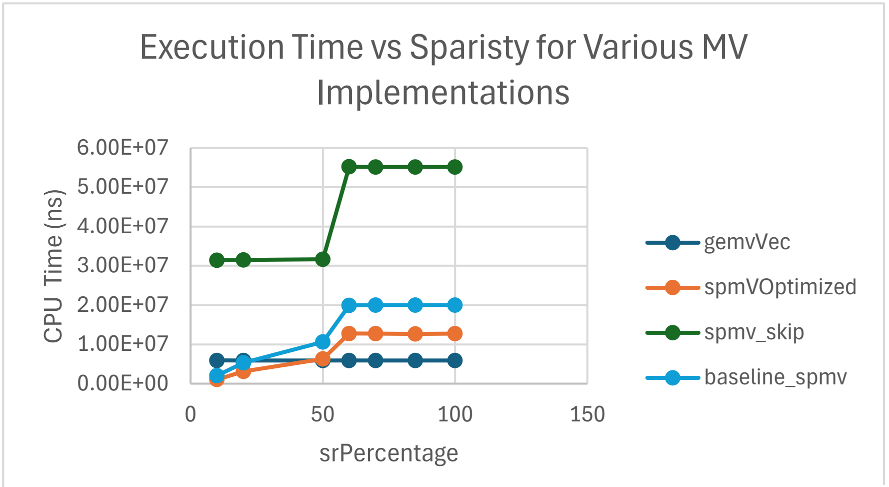

Figure 6 Execution time vs Sparsity for Various MV implementations

The main advantage of using a sparse matrix over a dense matrix is that operations involving zero elements can be skipped. When a matrix is stored in CSR format, we only store non-zero elements and their positions, allowing us to avoid unnecessary computations. If there's a zero element in matrix A, the corresponding entry in the dot product will always be zero, so we don't need to send it to the ALU for computation or waste time fetching the corresponding element from vector b. However, as the density of non-zero elements in the matrix increases, the execution time of SpMV increase for the same matrix dimensions. It can be seen that once the srPercentage is greater than 50, the gemVec will outperform the spmVecOptimzed. This is because in gemv the execution time is relatively stable regardless of density. However, in SpMV execution time increases linearly as density rises. To get a clearer picture on why this is, the graph of the miss% can be analyzed:

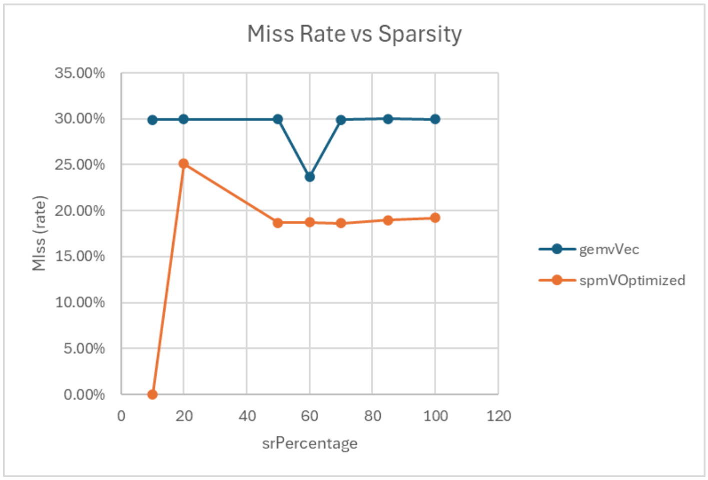

Figure 7Miss Rate vs Sparisty Graph for mv

The graph shows that for gemVec, the cache miss rate remains relatively constant at around 30%. This is expected because gemv processes the matrix as dense, accessing each element regardless of whether it's zero or non-zero. This results in consistent memory access patterns and, therefore, a stable cache miss rate. In contrast, SpMVOptimized has a lower and more variable miss rate. For very sparse matrices, the miss rate is close to 0% because only a few non-zero elements are accessed, leading to fewer overall memory accesses and, consequently, fewer cache misses. However, as matrix density increases, the miss rate rises initially. This increase occurs because more non-zero elements create a greater number of irregular memory accesses when fetching corresponding elements from vector b, disrupting spatial locality. This disruption leads to a higher rate of cache misses, cache evictions, and reloads, which ultimately slows down performance. Eventually, the cache miss rate stabilizes around 18% due to a tradeoff: as more elements become non-zero, the sparse matrix format loses effectiveness, but the increasing density improves spatial locality, slightly offsetting the negative impact on cache performance. Another reason why could be due to the stride indexing issue as described in table 1.

Although the miss rate percentage is lower in spmV than in gemV the reason why the gemV still outperforms the spmV at high densities is due to the number of memory loads/accesses required. **gemV has 3 loads**: element of matrix A, and vector b + c, while **spmV has 4 loads:** element of matrix A, column index, vector b + c. This additional memory load for the column index, combined with the irregular access patterns in the sparse format, introduces overhead and slows down execution. In addition, the sequential access pattern in gemvVec can fully exploit the SIMD instruction parallelism and it allows the hardware prefetcher to easily predict and load upcoming data in the cache. This can help decrease the execution time. In terms of other implementations, the baseline SpMV shows a similar trend to SpMVOptimized in the CPU time versus sparsity graph. However, its CPU time is consistently higher than SpMVOptimized because the optimized version incorporates vectorization and loop unrolling, which further reduce execution time. The SpMV_skip implementation consistently has the highest execution time across varying sparsity levels. This is due to its dense matrix approach, which involves additional checks in each iteration of the inner and outer loops to determine if an element in matrix A or vector b is non-zero. These checks add loop overhead. Unlike in SpMV, where zero elements can be entirely ignored, SpMV_skip still loads these zero elements to verify if they should be skipped. This increases memory load time, negating any advantage from skipping zero elements and ultimately leading to slower performance compared to other implementations.