Homeworks 3-4: Sparse Matrix Vector Multiplication, parallelized

Stella Greenvoss
11/29/2025 — CS431
Computation done on ix-dev and ORCA

Function	Runtime	Speedup from serial	Speedup from max
COO Serial	2.05966	1	1
COO Parallel	0.09873	20.86264	20.86264
CSR Serial	1.58293	1	1.30118
CSR Parallel	0.05230	30.26799	39.38398

Table 1: Runtime and speedup for all functions. Computation done on ix-dev.

This paper details my experience attempting to parallelize sparse matrix vector multiplication using different methodologies and different storage formats. The first section describes the process of parallelizing for the CPU using C and OpenMP, with the storage formats COO and CSR. The second section discusses parallelizing for the GPU using CUDA and C++ with the storage formats CSR and ELLPACK. In this particular paper, my focus will be more towards describing my process towards parallelizing these functions (rather than an attempt at describing solid academic findings).

Part of my difficulties in this process involved manually editing the Makefile and SLURM script to run on ORCA, the Oregon Regional Computing Accelerator. Their environment is set up differently to Talapas', so I had to copy some header files from the CUDA repository, change CUDA versions, and rewrite the SLURM script to properly load modules and submit jobs.

1 CPU Parallelism

This section will describe serial and parallel versions of SPMV using both COO and CSR.

1.1 COO and conversion to CSR

COO, or *COOrdinate* format, stores sparse matrices using three columns: row, indices, and data. Each column has a total of nnz elements, where nnz is the number of nonzero elements in the sparse matrix.

Computing the matrix-vector product serially using a matrix stored in COO is simple and runs in $O(nnz)$. Special care must be taken to ensure off-by-one safety, since the arrays stored in COO tend to be 1-indexed. As might be expected, this serial code performed the worst of the four functions. This is because much less work is being done per cycle, since a single thread must do every computation itself (assuming no compiler optimization). The data access pattern is also suboptimal, being linear for the row and col array accesses but without a clear pattern for r/w to and from the result and input, which negatively contributes to the temporal locality and thus the runtime.

The parallel version of COO-stored SPMV was also straight-forward, although there may be a more complicated implementation that performs better. Since I achieved a $20x$ speedup from the serial version, I decided that this would be sufficient for now. The algorithm is essentially the same, but with two parallel regions: one to initialize the result array and one to compute and add values to the result. To prevent simultaneous access, writing to a position in the result array in parallel must be done using an atomic pragma. It could also be done using an omp_lock, but my brief experiments with using that seemed to show that it was much slower.

Converting from COO to CSR was difficult, as parts of it had to be parallel. I had never done a parallel histogram before and had to use (and fix) my previous implementation of parallel prefix sum. For simplicity, I chose to use the simpler version of parallel prefix sum, which likely makes it overall less efficient than it could be.

1.2 CSR

CSR, or *Compressed Sparse Row* format, stores data and col arrays like COO. However, it also stores a third array, called row_ptr, which stores pointers to the first value in every row. This allows it to store rows of varying sizes in a compressed manner, removing the extraneous repeated values from COO's row array.

Implementing the serial version is relatively simple, although the relative complexity (compared to COO) made it so that reasoning about the indices takes a little extra work. Although both have the same asymptotic complexity, the CSR implementation does less work per row than the COO version. Because CSR directly stores row boundaries, the inner loop only iterates over the nonzeroes in that row. This improvement likely accounts for the small speedup (1.3x COO serial). On ix-dev, at least, the overhead from converting COO to CSR is not worth the tiny performance boost from a serial CSR SPMV implementation.

However, parallelized CSR SPMV was by far the best function, with a 30x speedup from the serial version. These performance gains were achieved via two simple OpenMP pragmas, parallelizing the initialization loop and the outer loop of the main computation. Schedule(static) seemed to achieve better performance than dynamic scheduling. The huge performance gains might be from the improved temporal locality achieved through the new data format, or as a consequence of the asymptotic improvement described in the serial section. Either way, it is clear that on a CPU with sufficient threads, CSR is the optimal storage format for computing SPMV. This should be (and certainly has been) explored further using different types of matrices.

An interesting final observation for the CPU-based part of this assignment is that I ran the executable on the large cant matrix on both ix-dev and a login node on ORCA. Ix-dev produced the behavior as seen in table 1. On ORCA, the distribution was bizarre, with parallel COO taking 2.5 seconds, almost twice as long as both serial functions. Looking at the output of lscpu on ORCA, the login node only has four CPUs with one thread per core. In contrast, ix-dev has 64, with two threads per core. Therefore, it makes perfect sense that parallel functions run slower on a node that is so ill-equipped to handle parallelism.

2 GPU Parallelism

This section will describe my experience writing CUDA kernels that compute SPMV. This was a significant challenge, made more difficult by the small architectural differences between Talapas and ORCA. The basis for comparison is in Table 2 as CSR (CPU), which is another version of parallel CSR written using OpenMP. I believe that the runtime for this is faster than in Table 1 because ORCA is better optimized for parallelism.

Function	Runtime	Speedup from CPU	Speedup from serial
CSR (CPU)	0.01170	1	111.79785
CSR (GPU)	0.00633	1.84868	206.67857
CSR (GPU, v2)	0.00608	1.92437	215.14009
ELL (GPU)	0.00405	2.89059	323.16177

Table 2: Runtime and speedup for the kernels. Speedup from serial is computed using the runtime 1.3085s, which is the time that serial CSR took on ORCA.

A key issue I encountered while programming the functions described was silent memory leakage. For a while, my executable would run perfectly fine on a small input, then crash without any proper error message *before entering main* on the real matrix. Eventually I found out about compute-sanitizer, which is a valgrind-like tool suite for CUDA programming. Running the executable with compute-sanitizer produced no errors and correct output, but without it, the crashes would still occur. It was only after enabling memcheck that I found that I was leaking hundreds of thousands of bytes in thousands of different instances. Cleaning up my code, tracking where memory is allocated and freed (including whether it was allocated on the device vs. host), and triple-checking my loop indices allowed me to tackle all of these leaks. The biggest change I ended up making to the main file was to not re-allocate memory for the matrix and input vector between different kernels. Since the function signature for allocating memory for ELL does not actually pass n, the original number of columns in the matrix, but rather k, the number of columns in the ELL data structure, we cannot know the proper amount of memory to allocate for the input vector x (which should have a length of n). So keeping the original allocations of these two values is reasonable, especially because allocating them again is simply repeated work, and they are only read and not modified.

2.1 CSR

My initial kernel code, using the CSR storage format to do SPMV, was very naive, using only one thread per row. This led to a performance gain from the CPU-based function, likely due to the larger possibility for parallelism on the GPU. However, it certainly wasn't an economical use of threads, with my code only using one thread per block.

Improving upon this, my second kernel assigned each thread in a block a portion of the row to be processed. They would then step through the row in block-sized increments, computing a local partial sum for those particular positions. A data array was used to store these partial sums, which was then reduced to a single value. This is an imperfect kernel, and could likely achieve greater performance by improving locality for the individual threads' accesses, since each of them end up reading the array in large strides (determined by block size). This implementation marginally improved upon the initial CSR GPU kernel, as can be seen in table 2.

Both kernels perform significantly better than the serial CPU-based implementation of SPMV using CSR. This degree of speedup was surprising for me; 100-200x is shocking and makes me a little concerned that some part of my calculations are incorrect. Assuming that they are correct, this reveals that a GPU-based (or highly parallelized approach running on a powerful computer) SPMV leads to incredible performance gains and should definitely be preferred.

2.2 ELL

ELLPACK (ELL) is a format that stores two 2D matrices: columns and data. The columns matrix stores the indices of the data. Both are m by k, where k is the maximum number of nonzeroes in a single row in the original matrix. When storing these data, they are represented as 1D arrays which are then indexed into as if they were still 2D.

This kernel was difficult to write, particularly due to the memory issues described above. However, the performance gains are significant, with nearly a 3x improvement from CPU-based parallel CSR and a 300x improvement from serial CSR.

ELL is known to be efficient for storing matrices whose maximum number of non-zero values in a single row is not significantly different from the typical number of non-zero values per row. If instead, the max is an outlier, say 100, while the average is 5, ELL ends up wasting a lot of space by padding out the rows. I do not know these numbers for the matrix that was provided, and would be curious to evaluate this kernel running on different matrices.

3 Conclusion

Overall, this project was rewarding and very challenging, particularly due to the unexpected time spent locating and debugging the memory errors. The best current runtime was achieved by ELL-based SPMV running on the GPU, but better results are surely possible.