CS406-HW2
Sparse Matrix - Dense Vector Multiplication (SpMV)

Görkem Yar (Student)

May 2024

1 Introduction

The objective of this assignment is to improve the performance of the SpMV operation. To this end, the matrix preprocessing and different representation types for the sparse matrix are allowed. Also, the keypoint to improve the SpMV operation using multithreading by implementing OpenMP primitives. I created three different approaches to improve the SpMV operation, namely CRS-Separated-Dense-Rows, CRS-RCM, and CRS-Dense-Row-Col. I got the best performance in the CRS-Separated-Dense-Rows algorithm. The implementations and performance of the all algorithms will be explained in the coming sections.

For all of my implementation models, after the preprocessing has been completed, I ordered my matrix in CRS format. CRS model matrix representation is especially useful for distributing different row-vector multiplications in different threads.

After all of the algorithms are introduced, I will explain the performance of my algorithms in the results & discussion section further.

2 Matrix-Metrics

Since we are required to improve the performance of the given matrix, I examine the properties such as the number of non-zeros, average row density, and the existence of dense rows.

- The matrix has dimensions 2987012 x 2987012.
- The number of non-zeros in the matrix is 26621990.
- The average row density is 8.91.
- The number of rows consists more than 1000 non-zeros is 6.
- The number of rows consists more than 2000000 non-zeros is 2.
- Also, the position of nonzero entries is diagonally symmetric, however, values are not symmetric. As a result, the matrix is not symmetric.

3 CRS-Separated-Dense-Rows

This algorithm has a preprocessing step that detects the dense rows. Dense rows consist of more than 1000 non-zeros. 1000 is a predefined value that can be changed, however the experiments conducted with the density metric 1000.
The logic behind this algorithm as follows:

- Total number of multiplications is 26621990.
- In the best case for 16 thread, each thread will do 26621990 / 16 = 1663874 multiplication operations.

- As in the Matrix-Metrics section mentioned, there are rows with more than 2000000 non-zero elements. If a single thread runs these rows, the job distribution among the threads would be uneven.

The pseudocode for the algorithm as follows:

```
void iterative_spmv_par_dense(int* crs_ptrs, int* crs_colids, double* crs_vals, const
    unordered_set<int> &dense, double* in, double* &out) {
    int t = omp_get_max_threads();
    int pad = (64/sizeof(int));
    double* tmp_thread = new double[t*64];

    #pragma omp parallel for schedule(static, 512)
    for(int i = 0; i < NumRows; i++) {
        if (crs_ptrs[i+1] - crs_ptrs[i] < DensityMetric){
            out[i] = 0;
            for(int  p = crs_ptrs[i]; p < crs_ptrs[i+1]; p++) {
                out[i] += crs_vals[p] * in[crs_colids[p]];
            }
        }
    }

    for(const auto &i: dense) {
        for (int j = 0; j < t; j++){
            tmp_thread[j*pad] = 0;
        }
        #pragma omp parallel for schedule(static, (crs_ptrs[i+1]-crs_ptrs[i]) / (t))
        for(int k = crs_ptrs[i]; k < crs_ptrs[i+1]; k++) {
            tmp_thread[omp_get_thread_num()*pad] += crs_vals[k] * in[crs_colids[k]];
        }
        out[i] = 0;
        for (int j = 0; j < t; j++){
            out[i] += tmp_thread[j*pad];
        }
    }
}
```

Listing 1: CRS-Separated-Dense-Rows

Input: Matrix in CRS format, Dense Row information, Input vector
Output: SpMV result vector.

For the sparse rows, approximately the work is equal and distributed randomly in the matrix. To this end, I decided to use static scheduling with a chunk size of 512. This would lead to approximately 5834 jobs. I again used static scheduling for the dense rows since the only job is multiplication and there is no need for dynamic or guided scheduling.

In the dense rows, since the multiplication for a single row is divided among the threads, I created a temporary array to avoid conflicts in writing the results. I have adjusted (padded) the size of the temporary array to match the L1 cache line size to reduce the performance degradation caused by false sharing. False sharing occurs when multiple threads on different processor cores modify data that resides on the same cache line, leading to unnecessary data transfers and cache invalidations. By aligning each element with a cache line, we ensure that modifications made by one thread do not interfere with the cache lines accessed by other threads. L1 cache line size is 64 bytes in the device that the experiments are conducted.

Further Discussion: To reduce thread overhead that comes from the thread creation and joining, one might consider reducing the two #pragma's in the given algorithm to one #pragma (one thread creation into a single parallel region). The algorithm was not that straightforward since the second #pragma does parallelize a single row while the first #pragma runs thread in different rows. Also, the new algorithm might not run faster because of the new metrics. So, I did not implement the algorithm in that way.

Important Note Regarding Correctness: Since this algorithm and the next ones run some rows in parallel the results obtained by them differ from the sequential execution because of the accuracy of double precision point operations. To check correctness, the results obtained from parallel execution normalized with the sequential results and checked whether the first 6th floating point numbers differ.

4 CRS-RCM

As a preprocessing, this algorithm uses the Reverse Cuthill-McKee (RCM) algorithm to reorder the matrix since the positions of the non-zero elements are symmetric. The RCM algorithm reduces the bandwidth of each row in the graph. The bandwidth is the distance of the farthest point from the diagonal entry of that row. The reordering can improve the performance of cache since the elements that are needed for the computations reside in the closer regions. As a result, it improves the spatial locality. Also, reordering can lead to more balanced data that can improve the load balancing between threads.

Note: I get the implementation of the RCM algorithm from the University of South Carolina's code libraries.

However, because of the dense rows, the RCM algorithm cannot create the reordering permutation. To this end, I created a copy matrix that does not have dense rows. Then the reordering permutation is applied to this new copy matrix without the dense rows. In the CRS-RCM algorithm computation is first applied to the reordered matrix by RCM then the dense rows are separately handled.

```
1  void iterative_spmv_par_rcm(int* crs_ptrs, int* crs_colids, double* crs_vals, int* perm,
       int dense_count, int* dense_row, int* dense_colid, double* dense_val, int*
       dense_perm, double* in, double* &out) {
2      int t = omp_get_max_threads();
3      int pad = (64 / sizeof(int));
4      double* tmp_thread = new double[t * pad];
5
6      #pragma omp parallel for schedule(static, 512)
7      for(int i = 0; i < NumRows; i++) {
8          for(int p = crs_ptrs[i]; p < crs_ptrs[i+1]; p++) {
9              out[perm[i]] += crs_vals[p] * in[perm[crs_colids[p]]];
10         }
11     }
12
13     for(int i = NumRows - dense_count; i < NumRows; i++) {
14         for (int j = 0; j < t; j++) {
15             tmp_thread[j * pad] = 0;
16         }
17         #pragma omp parallel for schedule(static, (dense_row[i+1] - dense_row[i]) / (t))
18         for(int k = dense_row[i]; k < dense_row[i+1]; k++) {
19             tmp_thread[omp_get_thread_num() * pad] += dense_val[k] * in[dense_colid[k]];
20         }
21         out[dense_perm[i]] = 0;
22         for (int j = 0; j < t; j++) {
23             out[dense_perm[i]] += tmp_thread[j * pad];
24         }
25     }
26 }
```

Listing 2: CRS-RCM

Input: Reordered matrix in CRS format, Dense rows in CRS format, Input vector, Permutation vector
Output: SpMV result vector.

Since the reordered rows are sparse and approximately the work is equal, I decided to use static scheduling with a chunk size of 512. This would lead to approximately 5834 jobs. The dense rows are again calculated like the CRS-Separated-Dense-Rows algorithm. The padding optimization for the dense rows is again exercised in this algorithm as well.

Unfortunately, this algorithm's performance wise could not compete with the CRS-Separated-Dense-Rows algorithm. One of the reasons that negatively affects the performance of this algorithm is the perm factor in the 8'th line (especially the "in[perm[crs_colids[p]]]" part).

5 CRS-Separated-Dense-Row-Col

So far I tried to optimize the performance using the matrics metric that I found. The CRS-Separated-Dense-Rows algorithm utilizes the computation of the dense rows, and the CRS-RCM algorithm exploits the symmetry of the non-zero entries. The CRS-Separated-Dense-Row-Col algorithm combines both of these metrics into a single method. Since the matrix is symmetric the columns corresponding to dense rows are also dense, and some of the indices in the input vector are excessively used especially the ones that have more than 2000000 elements. The idea is to do the operations for these columns separately at the beginning since the read operation for the input vector mainly will come from the cache and we will make fewer cache misses regarding these IDs.

For the dense columns, I also used the CRS matrix format. As a result, I have 3 CRS to represent the matrix. One for the dense columns, one for the dense rows, and the other for the remaining non-zeros. I used extra NumRows*sizeof(int) of memory to represent the crs_ptr_col.

```
void iterative_spmv_par_col(int* crs_ptrs, int* crs_colids, double* crs_vals, int*
    crs_ptr_col, int* crs_colid_col, double* crs_val_col, const unordered_set<int> &
    dense, int* crs_ptrs_new, int* crs_colids_new, double* crs_values_new, double* in,
    double* out) {
    int t = omp_get_max_threads();
    int pad = (64 / sizeof(int));
    double* tmp_thread = new double[t * pad];

    #pragma omp parallel
    {
        #pragma omp for schedule(static, 4096)
        for(int i = 0; i < NumRows; i++) {
            out[i] = 0;
            for(int p = crs_ptr_col[i]; p < crs_ptr_col[i+1]; p++) {
                out[i] += crs_val_col[p] * in[crs_colid_col[p]];
            }
        }

        #pragma omp for schedule(static, 512)
        for(int i = 0; i < NumRows; i++) {
            if (crs_ptrs[i+1] - crs_ptrs[i] < DensityMetric) {
                for(int p = crs_ptrs[i]; p < crs_ptrs[i+1]; p++) {
                    out[i] += crs_vals[p] * in[crs_colids[p]];
                }
            }
        }
    }
    for(const auto &i : dense) {
        for (int j = 0; j < t; j++) {
            tmp_thread[j * pad] = 0;
        }
        #pragma omp parallel for schedule(static, (crs_ptrs[i+1] - crs_ptrs[i]) / (t))
        for(int k = crs_ptrs[i]; k < crs_ptrs[i+1]; k++) {
            tmp_thread[omp_get_thread_num() * pad] += crs_vals[k] * in[crs_colids[k]];
        }
        for (int j = 0; j < t; j++) {
            out[i] += tmp_thread[j * pad];
        }
    }
}
```

Listing 3: CRS-Separated-Dense-Row-Col

Input: Matrix in 2 CRS format (one for dense columns, one for the other non-zeros), Dense Row information, Input vector
Output: SpMV result vector.

For the dense columns that are represented in CRS format, approximately workload for each row is even. As a result, I used static scheduling with a chunk size of 4096. For the sparse and dense rows, an approach similar to the CRS-Separated-Dense-Rows is conducted. The padding optimization for the dense rows is again exercised in this algorithm as well.

6 Results & Discussion

Results for SpMV with 10 iterations:

Algorithm	Thread Count	Seconds
Sequential	N/A	0.494328
CRS	1	0.481914
Separated	2	0.303812
Dense	4	0.192940
Rows	8	0.146536
	16	0.135613
CRS	1	0.990851
RCM	2	0.553387
	4	0.447598
	8	0.284717
	16	0.219806
CRS	1	0.590780
Separated	2	0.339869
Dense	4	0.228930
Row	8	0.161551
Col	16	0.162437

Results for SpMV with 100 iterations:

Algorithm	Thread Count	Seconds
Sequential	N/A	4.640102
CRS	1	4.808781
Separated	2	3.381575
Dense	4	2.191177
Rows	8	1.149991
	16	0.965991
CRS	1	10.916075
RCM	2	8.347444
	4	4.440239
	8	2.673968
	16	1.855637
CRS	1	6.320096
Separated	2	3.897012
Dense	4	2.109845
Row	8	1.436897
Col	16	1.353684

Algorithm	Thread Count	Seconds
Sequential	N/A	4.763828
CRS	1	5.645028
Separated	2	3.313380
Dense	4	1.789170
Rows	8	1.191925
	16	1.352894
CRS	1	14.110535
RCM	2	8.669302
	4	4.566285
	8	2.675905
	16	1.987123
CRS	1	5.783017
Separated	2	3.749216
Dense	4	2.253673
Row	8	1.659290
Col	16	1.543882

The performance differents among the tested algorithms can largely be attributed to their handling of matrix structure, particularly the dense rows and columns.

CRS-Separated-Dense-Rows: Optimizes load balancing by separately solving dense rows and improving overall performance.
CRS-RCM: While theoretically advantageous due to reduced bandwidth and improved spatial locality, the practical implementation faced challenges, especially in managing permutations.
CRS-Separated-Dense-Row-Col: Attempted to merge the benefits of the previous two algorithms, showing improved performance over CRS-RCM but still lagging behind CRS-Separated-Dense-Rows, possibly due to overheads from handling both dense rows and columns.

CRS-Separated-Dense-Rows algorithm almost has a speedup of 4 times with 16 threads compared to Sequential implementation. It is the best algorithm of all and creates the best results almost every time. The GFLOP values for the algorithms:

- Sequential = 2*26621990*10 / 0.494328 = 1.077GFLOPS

- CRS-Separated-Dense-Rows 1 thread = 2*26621990*10 / 0.481914 = 1.104GFLOPS
- CRS-Separated-Dense-Rows 2 thread = 2*26621990*10 / 0.303812 = 1.752GFLOPS
- CRS-Separated-Dense-Rows 4 thread = 2*26621990*10 / 0.192940 = 2.759GFLOPS
- CRS-Separated-Dense-Rows 8 thread = 2*26621990*10 / 0.146536 = 3.633GFLOPS
- CRS-Separated-Dense-Rows 16 thread = 2*26621990*10 / 0.135613 = 3.926GFLOPS
- CRS-RCM 1 thread = 2*26621990*10 / 0.990851 = 0.537 GFLOPS
- CRS-RCM 2 thread = 2*26621990*10 / 0.553387 = 0.962 GFLOPS
- CRS-RCM 4 thread = 2*26621990*10 / 0.447598 = 1.189 GFLOPS
- CRS-RCM 8 thread = 2*26621990*10 / 0.284717 = 1.870 GFLOPS
- CRS-RCM 16 thread = 2*26621990*10 / 0.219806 = 2.422 GFLOPS
- CRS-Separated-Dense-Row-Col 1 thread = 2*26621990*10 / 0.590780 = 0.901GFLOPS
- CRS-Separated-Dense-Row-Col 2 thread = 2*26621990*10 / 0.339869 = 1.566GFLOPS
- CRS-Separated-Dense-Row-Col 4 thread = 2*26621990*10 / 0.228930 = 2.325GFLOPS
- CRS-Separated-Dense-Row-Col 8 thread = 2*26621990*10 / 0.161551 = 3.295GFLOPS
- CRS-Separated-Dense-Row-Col 16 thread = 2*26621990*10 / 0.162437 = 3.277GFLOPS

7 References

- RCM source from University of South Carolina link