Computing Sparse Matrix Permanents with OpenMP and CUDA

Kaan Bilgili, Görkem Yar, İsmail Berat Düzenli

June 2024

1 Introduction

In this project, we are asked to design and implement parallel algorithms in CPU and GPU for computing the exact permanent of the sparse matrices.

Permanent. The permanent of a square matrix (nxn matrix) is defined as:

$$\text{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}$$

where the summation is done on the set of all permutations of numbers 1 to n, the computation of the permanent resembles the computation of the determinant, as only the signs on the summation differ. Another difference between the determinant and the permanent is that while the determinant can be computed in $O(N^3)$ time [1], computation of the permanent takes exponential time and it remains as an NP-hard problem[2].

In the following sections, we will talk about state-of-the-art algorithms for permanent calculation. Later we will introduce the Ryser's and SkipOrd algorithms and how they can be parallelized. We will also give a performance analysis for these algorithms. Then, we will explain the improvements in GPU with their results and speed up. & Finally, we will conclude with the Discussion & Future Work section.

2 Literature Survey

The time of computation of the permanent is exponentially bounded. As such, there have been various studies that try to make faster algorithms for the exact computation of the permanent [3][4]. There are also studies that approximate the permanent[5].

In 1969, using the inclusion-exclusion formula Ryser developed a general exact algorithm for permanent calculation [6]. This algorithm is still one of the fastest algorithms for permanent calculation and it's time complexity is $O(n^2 2^n)$. Another algorithm developed by mathematicians is the Balasubramanian–Bax–Franklin–Glynn algorithm [7]. This algorithm has time complexity of $O(2^n n)$, which is an improvement over Ryser's algorithm.

There are various studies that tries to improve and specialize Ryser's algorithm. One of these is the Lundow & Markstörm Greedy Partition [4]. They parallelized the algorithm for general matrices and worked on improvements for sparse matrices. The algorithm is designed to be used for an efficient Boson Sampling implementation.

Another study that tries to improve Ryser's algorithm is the work by Kamer Kaya [3]. In this research, two parallel algorithms are presented: one focuses on enhancing Ryser's algorithm for computing the permanents of sparse matrices, and the other introduces a novel row-ordering method.

3 Ryser's Algorithm

Here is the mathematical representation of Ryser's method to calculate permanent of a matrix.

$$\text{per}(A) = (-1)^n \sum_{J \subseteq N} (-1)^{|J|} \prod_{i=1}^{n} \sum_{j \in J} a_{i,j}$$

Below is the pseudocode of the Ryser's algorithm that is implemented.

Algorithm 1 Compute the Permanent of a Matrix

Require: M (an $N \times N$ input matrix)
Ensure: Perm(M)
1: $rowSum \leftarrow$ array of size N
2: $pMultiplied \leftarrow 1$
3: **for** $r \leftarrow 1$ **to** N **do**
4: $\quad sum \leftarrow 0$
5: $\quad$ **for** $c \leftarrow 1$ **to** N **do**
6: $\quad\quad sum \leftarrow sum + M[r, c]$
7: $\quad$ **end for**
8: $\quad rowSum[r] \leftarrow M[r][N] - \frac{sum}{2}$
9: **end for**
10: $pMultiplied \leftarrow$ Multiplicate($rowSum$)
11: **for** $g \leftarrow 1$ **to** $2^{(N-1)}$ **do**
12: $\quad jking \leftarrow \log_2(\text{ToGray}(g) \oplus \text{ToGray}(g-1))$
13: $\quad s \leftarrow 2 \cdot (\text{ToGray}(g-1)[jking]) - 1$
14: $\quad prod \leftarrow 1$
15: $\quad$ **for** $i \leftarrow 1$ **to** N **do**
16: $\quad\quad rowSum[i] \leftarrow rowSum[i] + (s \cdot M[i][jking])$
17: $\quad\quad prod \leftarrow prod \cdot rowSum[i]$
18: $\quad$ **end for**
19: $\quad pMultiplied \leftarrow pMultiplied + (-1)^g \cdot prod$
20: **end for**
21: **return** $pMultiplied \cdot (4 \cdot (N \mod 2) - 2)$

3.1 Complexity

Complexity of the Rysers algorithm is $O(n2^n)$ which can be calculated by looking at two nested loops. Loop 1(11) is called 2^{n-1} times and inside loop (ii) is called n times. The first loop at the beginning and multiplication of rowSum have n computations but complexity is overwhelmed by nested second loop resulting in $O(n2^n)$ time complexity.

4 Parallel Sparse Ryser's Algorithm

```
1  double parSpaRyser(double* matrix, int* rptrs, int* colids, double* rvals, int* cptrs
       , int* rowids, double* cvals, int N, int t){
2      double* x = new double[N];
3      double result = 1;
4
5      for (int i = 0; i < N; i++){
6          double sum = 0;
7          for (int ptr = rptrs[i]; ptr < rptrs[i+1]; ptr++){
8              sum += rvals[ptr];
9          }
10         x[i] = (double)matrix[(i+1)*N - 1] - (((double)sum)/2.0);
11         result *= x[i];
12     }
```

```
    long end = 1l << (long)(N - 1);
    long chunk = (end + t - 1) / t;
    #pragma omp parallel num_threads(t)
    {
        double my_x[N];
        for (int i = 0; i < N; i++){
            my_x[i] = x[i];
        }
        long thread_id = omp_get_thread_num();
        long my_start = thread_id * chunk;
        if (thread_id == 0) my_start++;
        long my_end = min(end, (thread_id+1)*chunk);

        long my_g = binaryToGray(my_start-1);
        long g_c = my_g;
        long ptr = 0;
        while (g_c > 0){
            if (g_c & 1l){
                for (int j = cptrs[ptr]; j < cptrs[ptr+1]; j++){
                    my_x[rowids[j]] += cvals[j];
                }
            }
            g_c = g_c >> 1;
            ptr++;
        }

        double my_result = 0;
        double prod = 1;
        int nzeros = 0;
        for (int i = 0; i < N; i++){
            if (my_x[i] != 0){
                prod *= my_x[i];
            }else{
                nzeros++;
            }
        }

        long b = my_start;
        long my_g_pre = my_g;
        long row_idx;
        double div;
        while (b < my_end){
            my_g = binaryToGray(b);
            long xorVAR = my_g ^ my_g_pre;
            long jking = __builtin_ctzl(xorVAR);
            double s = 2 * getBit(my_g, jking) - 1l;

            div = 1;
            for (int k = cptrs[jking]; k < cptrs[jking+1]; k++){
                row_idx = rowids[k];
                if (my_x[row_idx] == 0){
                    nzeros--;
                    my_x[row_idx] += s * cvals[k];
                    prod *= my_x[row_idx];
                }else{
                    div *= my_x[row_idx];
                    my_x[row_idx] += s * cvals[k];
                    if (my_x[row_idx] == 0){
                        nzeros++;
                    }else{
                        prod *= my_x[row_idx];
                    }
                }
            }
            prod /= div;

```

```
79              if (nzeros == 0){
80                  double sign = b & 1 ? -1 : 1;
81                  my_result += sign * prod;
82              }
83              my_g_pre = my_g;
84              b++;
85          }
86          #pragma omp atomic
87          result += my_result;
88      }
89
90      return result * (4 * (N%2) - 2);
91  }
```

4.1 Optimizations

4.1.1 Finding index of the last bit that contains 1:

Using a built-in function instead of using log2 for faster calculation. The change from log to _built_ctzl (__ffsll on GPU) on line 57 improved the algorithm 1.3 times.

4.1.2 Aggregate division instead of iterative division, lines: 68, 77

Dividing can take up to 30 clock cycles. If we divide and wait for the result to be written into the prod. It would cost many clock cycles. Instead of dividing at each iteration, multiplying the divisors and dividing at the end once achieves a speed-up of 1.6 times.

Table 1: Performance Results of Division Optimization with CPU

Matrix Name	Without Builtin and division Optimizations	division at every iteration	aggregate division
ey35_02	33.44	22.80	15.81
ey35_03	42.37	37.51	23.07
ey36_02	68.84	52.39	34.19
ey36_03	115.22	97.73	52.86
football	20.703	11.71	10.08
cage5	128.67	104.32	71.53
chesapeake	580.48	390.36	292.85

With using builtin_ctzl function instead of log2 we improved the performance by 1.2-1.3 times. In addition to that, aggregate division optimization improved it by another 1.6-1.7 times speed up.

4.1.3 Parallelization

The parallelization of the ParSpaRyser Algorithm is accomplished by partitioning the loop (line 15) iterations for the different threads. We choose coarse-grained parallelism since the number of tasks is large (2^n) and tasks are independent and equal in terms of the amount of the computation. Also, with this approach, we are able to reduce synchronization overhead and load balancing overhead for the threads. As a result, τ threads partition the 2^n iterations evenly. Each thread executes the following region of the loop:

- Loopvariant = 2^{n-1}
- Size = Loopvariant/τ
- start = size * id, end = size * (id + 1)

Each thread will execute the region between the start and end iterations.

4.1.4 Avoiding Interactions Optimizations: Reduction at the end, lines: 39, 87, figure 2

Instead of OpenMP reducing the result, every thread initializes and computes my_result. After threads compute their assigned chunks their result is aggregated to the final result at master.

5 SkipPer

```
1  double SkipPerOptimized(double* matrix, int* rptrs, int* colids, double* rvals, int*
       cptrs, int* rowids, double* cvals, int N, const int t){
2      double* x = new double[N];
3      for(int i = 0; i < N; i++){
4          double sum = 0;
5          for(int intp = rptrs[i]; intp < rptrs[i+1]; intp++){
6              sum += rvals[intp];
7          }
8          x[i] = (double)matrix[(i+1)*N - 1] - (((double)sum)/2.0);
9      }
10     double result = 1;
11     for(int i = 0; i < N; i++){
12         result *= x[i];
13     }
14     long loopVariant = pow(2ll, (long)(N - 1));
15     long loopDivision = loopVariant / DIVIDER;
16     long loopSize = loopDivision / t;
17     #pragma omp parallel num_threads(t)
18     {
19         double my_x[N];
20         for (int i = 0; i < N; i++){
21             my_x[i] = x[i];
22         }
23         double my_result = 0.0, s;
24         int id = omp_get_thread_num();
25         long gpre = binaryToGray(0);
26         for (int u = 0; u < DIVIDER; u++){
27             long tmp = loopDivision * u, b = id * loopSize + tmp;
28             if (id == 0 && u == 0) b+=1;
29             long g = binaryToGray(b);
30             long limit = (id != t-1) ? ((long)(id + 1))*loopSize + tmp: loopDivision
                   *(u+1);
31             limit = (id == t-1) && (u == DIVIDER -1) ? loopVariant : limit;
32
33             for (; b < limit;){
34                 long grdiff = g ^ gpre, j = 0;
35                 while (grdiff > 0){
36                     if (grdiff & 1){
37                         s = 2 * getBit(g, j) - 1;
38                         for(int ptr = cptrs[j]; ptr < cptrs[j+1]; ptr++){
39                             my_x[rowids[ptr]] += (s * cvals[ptr]);
40                     }}
41                     grdiff = grdiff >> 1;
42                     j++;
43                 }
44                 double prod = 1;
45                 for(int i = 0; i < N; i++){
46                     prod *= my_x[i];
47                 }
48                 long s = b & 1 ? -1 : 1;
49                 my_result += s * prod;
50                 gpre = g;
51                 if (prod == 0){
52                     b = nextgOptimized(b, my_x, matrix, N, t);
53                 }else{
54                     b++;
```

```
55                  }
56                  g = binaryToGray(b);
57          }}
58          #pragma omp atomic
59          result += my_result;
60      }
61      return result * (4 * (N%2) - 2);
62  }
```

This algorithm is for sparse 0-1 matrices and it exploits sparsity by skipping iterations. A next(.) function is defined as: Find the next gray code that would contribute to the calculation of the permanent. So, with the next(.) function it is possible to only calculate the products that would contribute to the result.

SkipOrd. An ordering algorithm can be used to further enhance SkipPer algorithm's performance [3]. SkipOrd algorithm creates a matrix structure such that the matrix entries are nearer to the diagonal in the upper triangular part.

5.1 Skipper Optimizations

5.1.1 Memory Access Optimizations: Dividing access to x, lines: 19-22, figure 1

Instead of creating matrix of x's in master thread where there are n identical rows of x and every thread accessing to a single row assigned to itself - every thread creates their own copy of x after threads started to run.

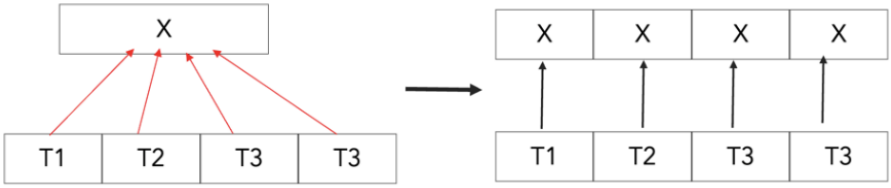

Figure 1: Multiple x arrays

5.1.2 Avoiding Interactions Optimizations: Reduction at the end, lines: 23, 59, figure 2

Instead of OpenMP reducing the result, every thread initializes and computes my_result. After threads compute their assigned chunks their result is aggregated to the final result at master.

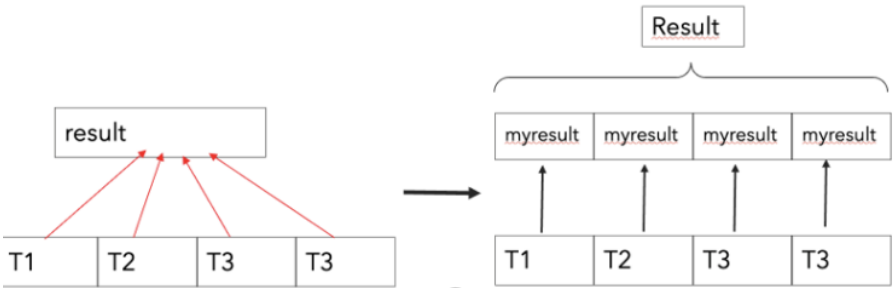

Figure 2: Result

5.1.3 Only necessary bits, lines: 35-43, figure 3

Instead of iterating over every bit in grdiff to find which rows will be included or not included now grdiff is shifted to the right and checked least significant bit until leftmost bit that has value 1 is computed. Thanks to this operation instead of 64 (number of bits in grdiff) iterations, at most n iterations are made.

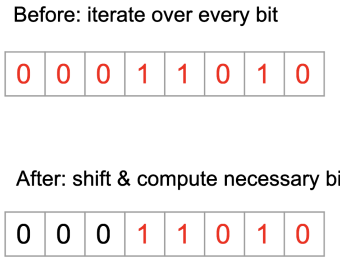

Figure 3: Result

5.1.4 Condition check beforehand, lines: 53-58

In the previous version, nextg function was called for every iteration and repeatedly checked if there exists a 0 in a given iteration's combination. However, if there was any then the production value would be 0. Therefore we checked the condition of production = 0 which indicates the existence of 0. If production is not 0 then there aren't 0s and nextg is not called and just calculated using b+1.

5.1.5 Load Balancing & Task Parallelism Optimizations: Dividing skipper jumps, figure 4

Instead of assigning iterations to threads as a single block, we first divided iterations to k chunks (=16) and inside these chunks iterations are assigned to all threads again. As a result skip operations are divided to threads more balanced and every threads working duration became closer to each other and shorter than maximum of previous version which defined total computation time previously.

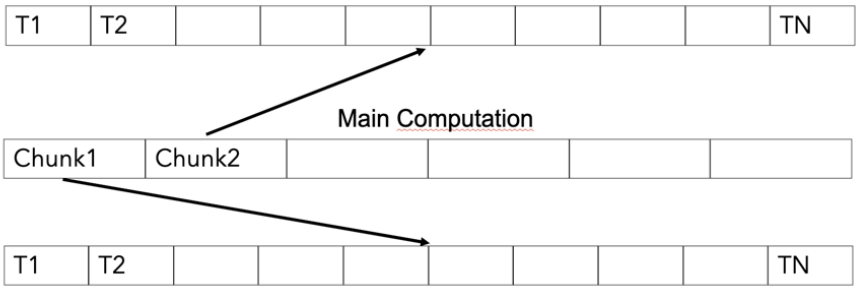

Figure 4: Result

5.2 Complexity

The complexity of the SkipPer algorithm can be calculated as such: The first nested loop is at most called n^2 times. The outer while loop on the second nested loop is called at most 2^{n-1} times and the inner nested for loops are called at most n^2 times. Therefore, the SkipPer algorithm has a time complexity of $O(2^{n-1}n^2)$. The SkipOrd algorithm works in $O(n^3)$ time, which is rather trivial.

For the implementations, we used C++ with OpenMP.

6 Parallelization in GPU

Parallelization of the two algorithms is implemented similarly. The following optimizations are implemented for the both SkipPer and ParSpaRyser algorithms. However, the performance of the ParSpaRyser algorithm in GPU dominates the SkipPer algorithm. To this end, the performance results that are given in the following sections are the results of ParSpaRyser algorithm.

6.1 Shared Memory

We transferred the compressed matrix to shared block memory to access values faster. In this way, instead of every thread accessing the memory from the same region, each block has its copy to access from.

- row ids, shared
- column pointers, shared
- column values (copied) shared

6.2 Coalescing

In CPU every thread creates a my_x, in GPU we create a united x of length #blockdim * $\|x\|$ in shared block memory, and this is filled in the following way: thread j puts the data in the indices j + i*block_dim, where i represents the index of the element in original x array and block_dim is the number of threads in the block.

6.3 Parallelization & Load Balancing

Threads on the GPU work the same as threads on the CPUs. They all have an iteration range that is assigned to them according to their block and thread IDs. They iterate over this range and save their results in a local my_result variable. Each thread in a block atomically writes its partial result into an output result array based on its block IDs. After the GPU computation is finalized, in the CPU we reduce this array into a single result. With this approach, we are able to avoid interactions between threads as much as possible and divide the work among them equally.

6.4 Weak and Strong Scaling:

To check the thread overhead that is a result of the parallelization. We run our ParSpaRyser algorithm with the following block sizes: 128, 256, 512, and 2048 (Using Single GPU). In each performance test threads per block are 128 (Weak and Strong Scaling Test).

Table 2: Performance Results - Weak, Strong Scaling

Matrix Name - Block Count	128	256	512	2048
ey35_02	3.69	3.40	3.20	2.90
ey35_03	4.93	4.25	4.10	3.80
ey36_02	7.56	6.61	6.55	5.57
ey36_03	11.62	9.78	9.70	8.93
football	2.72	2.48	2.30	1.98
cage5	14.70	12.15	12.23	11.10
chesapeake	61.09	51.16	51.04	47.07

In terms of weak and strong scaling our GPU algorithm did not perform ideal with a high number of threads. When we double the number of threads, the speedup is not reduced to half (Strong Scaling). If we examine the performance of ey36_02 with block size 256 and the performance of ey35_02, one can detect their performance are not the same even though we increased the total number of threads by 2 and doubled the task size. As a result, our algorithm does not perform ideal for weak scaling in a high number of threads.

This situation may stem from copying the compressed matrix form for each block. As the block size increased the total number of copied data increased as well.

6.5 Throughput

Throughput = Total-Units-of-Work-Done/Execution-Time

Matrix Name -	Second	Sparcity	Throughput
ey35_02	2.90	0.2	4.14E+11
ey35_03	3.80	0.3	3.16E+11
ey36_02	5.57	0.2	4.44E+11
ey36_03	8.93	0.3	2.77E+11
football	1.98	0.19	6.07E+11
cage5	11.10	0.17	4.68E+11
chesapeake	47.07	0.22	4.55E+11

The calculations of the throughput done in the following way: $2^N * N/$ execution_time The $2^N * N$ part comes from the complexity of the complexity of the Ryser's algorithm. Throughput decreases as the density increases. This is expected since ParSpaRyser algorithm is performing better in sparser matrices.

7 Final Results

The following table contains the best performance results for each matrix. The CPU results of the football, bcspwr01 and chesapeake (these matrices are 0-1 matrices) matrices are obtained from the SkipPer algorithm. Additionally, GPU results of the bcspwr01 is also obtained from the GPU version of the SkipPer algorithm. Also, the results of the GD98_a matrix (its permanent is 0) obtained via preprocessing not by either of the algorithms. All of the other results come from the ParSpaRyser algorithm.

Table 3: Performance Results

Matrix Name	CPU 16 Threads	GPU Single	GPU Dual	Results
ey35_02	15.0989	2.89567	1.86377	1.26E+15
ey35_03	23.0691	3.80129	2.39203	8.27E+19
ey36_02	34.1885	5.91286	3.47106	2.00E+15
ey36_03	52.859	8.92634	5.11796	3.76E+21
football	5.67711	1.97858	1.59633	5.39E+17
cage5	75.7038	11.0969	6.30624	5.15E-08
GD98_a	0	0	0	0
bcspwr01	0.179731	0.998974	0.923233	3.76E+08
chesapeake	186.224	47.3387	24.2358	1.32E+13

8 Discussion & Future Work

In our performance testing, we get a speed-up of 6-8 in our GPU algorithms compared to the CPU version. The double GPU version gets another 2 times speedup. Based on the performance of the other groups, we performed as the best group in the matrices with double entries, namely ey35_02, ey35_03, ey36_02, ey36_03; with the exception of the cage5 matrix. In the integer and 0-1 matrices our group performed good results as well. We have some best results in some of the test cases on those as well.

As future work, one needs to implement the zero matrix detection algorithm in a better and more robust manner. Also, for the 0-1 matrices one may skip the memory access since the nonzero entries of the algorithms are always 1. We did not implement this since we wanted more general algorithms as much as possible.

References

[1] Volker Strassen. Gaussian elimination is not optimal. *Numer. Math. 13, 354–356*, 1969.

[2] L.G. Valiant. The complexity of computing the permanent. *Numer. Math. 13, 354–356*, 1977.

[3] Kamer Kaya. Parallel algorithms for computing sparse matrix permanents. *Turkish Journal of Electrical Engineering and Computer Sciences*, 2019.

[4] P.H. Lundow and K. Markström. Efficient computation of permanents, with applications to boson sampling and random matrices. *Journal of Computational Physics*, 2020.

[5] MARK JERRUM, ALISTAIR SINCLAIR, and ERIC VIGODA. A polynomial-time approximation algorithm for thepermanent of a matrix with nonnegative entries. *Journal of the ACM, Vol. 51, No. 4, July 2004, pp. 671–697*, 2004.

[6] Herbert John Ryser. *Combinatorial Mathematics*. Mathematical Association of America, USA, 1963.

[7] Wikipedia. Computing the permanent — Wikipedia, the free encyclopedia, 2024. [Online; accessed 04-04-2024].