SABANCI UNIVERSITY

CS406/531: Parallel Computing

Spring 2023-2024

Term Project Final Report

Computing Sparse Matrix Permanents with OpenMP and CUDA

Utku Alkan & Ekin Nalbantoğlu

Introduction

This project involves the design, implementation, and performance evaluation of a parallel computing solution to compute sparse matrix permanents using OpenMP and CUDA. This report includes the design and implementation progress, algorithm complexity, levels of parallelism exploited, profiling of existing code, and theoretical speed-up estimates.

Implementation

To calculate the permanent of a matrix, our first approach revolved around implementing the Ryser's formula (Lint & Wilson, 2001). However, exploiting Ryser's formula in its basic form was highly lacking in efficiency; therefore, we moved to other solutions which take sparsity of the matrix into account. After the literature revision, two algorithms were found to be of interest, SkipPer and SpaRyser. These approaches are designed for the efficient computation of sparse matrix permanents (Kaya, 2019). To enhance the performance further, we implemented the parallel versions of these algorithms as described in the work of Yağlıoğlu, which significantly improved the efficiency of the permanent calculations(2021). As the authors'

findings point out, the SpaRyser algorithm prevails as the superior candidate for the studied project. The pseudocode of the algorithm can be seen in Figure 1.

For the preprocessing stage, we first read the matrix file and convert it to Compressed Column Storage (CCS) format. Afterwards, “SortOrd” algorithm is applied, sorting the matrix columns in ascending order with respect to the number of nonzero values in the columns, as proposed in Yağlıoğlu's thesis. This sorting method significantly reduced the time it takes to calculate the permanent (see Figure 2).

Afterwards, we parallelized the algorithm with OpenMP. However, after executing some of the matrices, we have seen that some threads finish earlier than others and have to wait. This imbalance reduces the exploited parallelism and we did not want any core to waste their computational power. Therefore, we divided the load into chunks and integrated dynamic and guided scheduling. Given the structure of SpaRyser, there is no room for different scheduling approaches, whereas on the Skipper counterpart, the execution is constructed to accommodate such scheduling policies. However, modifying the chunk sizes with respect to customly created chunks instead of number of threads allows the developers to partition the job of a single thread into multiple job packets, such that these packets can be allocated in dynamic/guided scheduling scenarios. This hybrid take on the original SpaRyser algorithm allowed a more efficient load balancing among the threads. To decide the scheduling type, chunk size of the scheduling, and chunk size of the load division; all three parameters are swept in the possible execution space with appropriate step sizes and the taken time is measured, as it is the only figure of merit. The decided scheduling type is dynamic with chunk size 1 and chunk size of the load division is

8192. However, it is important to note that, sweeping experiment with the chunk size of the load division was found to be inconclusive, therefore based on small differences, 8192 is selected manually. Thanks to the scheduling, overall performance improved with utilizing all 16 cores within the execution period.

Algorithm 5 `ParSpaRyser` $(mat, cptrs, rows, cvals, dim, start, end)$

```
1: p ← 1
2: for i = 1...N do
3:     rowSum ← 0
4:     for j = 1...N do
5:         rowSum ← rowSum + mat[i, j]
6:     x[i] ← mat[i, n] − rowSum/2
7:     p ← p × x[i]
8: for each thread do
9:     myX ← x
10:    myP ← 0
11:    myStart ← start + threadId × chunkSize
12:    myEnd ← min(start + (threadId + 1) × chunkSize, end)
13:    calculate myX using GrayCode_{myStart−1}
14:    prod ← ∏_{n=1}^{dim} myX[i] for myX[i] ≠ 0
15:    zeroNum ← Σ_{n=1}^{length(myX)} myX[i] for myX[i] = 0
16:    for g = myStart...myEnd − 1 do
17:        j ← log2(GrayCode_g ⊕ GrayCode_{g−1}) + 1
18:        s ← 2 ∗ GrayCode_g[j] − 1
19:        for i = cptrs[j]...cptrs[j] do
20:            if myX[rows[i]] == 0 then
21:                zeroNum ← zeroNum − 1
22:                myX[rows[i]] ← myX[rows[i]] + s × cvals[i]
23:                prod ← prod × myX[rows[i]]
24:            else
25:                prod ← prod/myX[rows[i]]
26:                myX[rows[i]] ← myX[rows[i]] + s × cvals[i]
27:                if myX[rows[i]] == 0 then
28:                    zeroNum ← zeroNum + 1
29:                else
30:                    prod ← prod × myX[rows[i]]
31:        if zeroNum == 0 then
32:            myP ← myP + (−1)^g ∗ prod
33:    AtomicAdd(p, myP)
34: return p × (4 × (n mod 2) − 2)
```

Fig 1. ParSpaRyser Pseudocode (Yağlıoğlu, 2021)

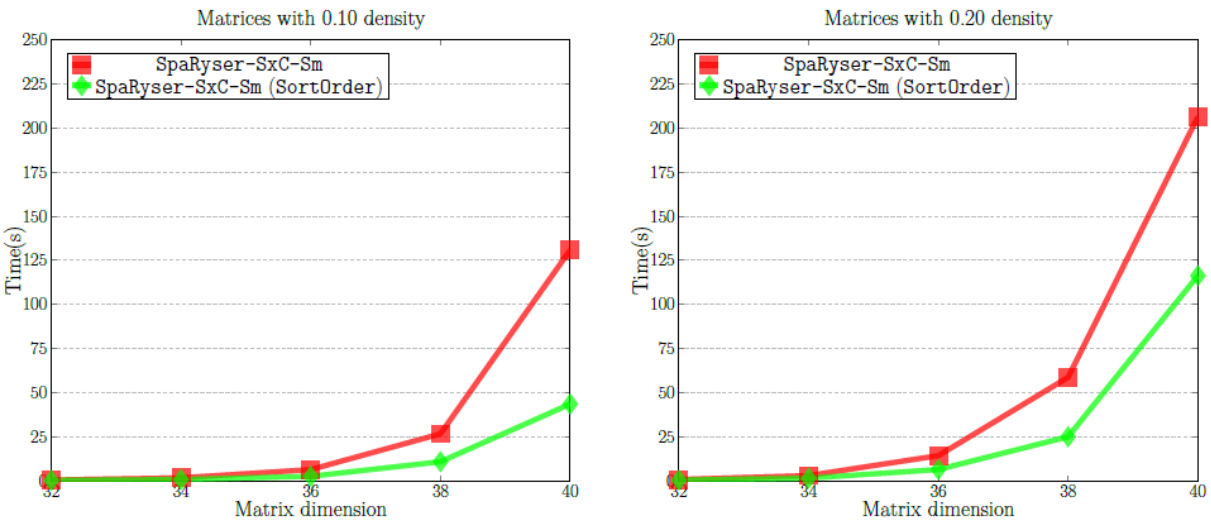

Fig 2. Execution time with and without column sorting (Yağlıoğlu, 2021)

For the GPU implementation, we adapted the parallelization from OpenMP to CUDA. To decide the number of blocks and block size in the kernel, we again tuned them by sweeping many possible values. In Figure 3, the execution time alteration with different numbers of blocks and block sizes can be traced. Determined sizes are as follows:

$$Grid\ Size = 16384$$

$$Block\ Size = 256$$

Furthermore, in the server there were 4 GPUs so we wanted to utilize them as well. First we extended the algorithm to 2 GPUs. Since these two GPUs are the same, we gave the same loads to each. Overall, using two GPUs provided approximately two times speedup in big matrices and 1.5x speedup in smaller matrices. Lastly, we extended the algorithm to 4 GPUs. However, since the newly introduced two GPUs are approximately three times slower, we loaded them with ⅓ load compared to the first two GPUs, employing load balancing. This further improved the performance especially in bigger matrices. As an example, in a 39x39 matrix, 1 GPU executed

the calculation in 47.3387 seconds while 2 GPUs took 23.5226 seconds. When 4 GPUs are used, execution time decreases to 20 seconds.

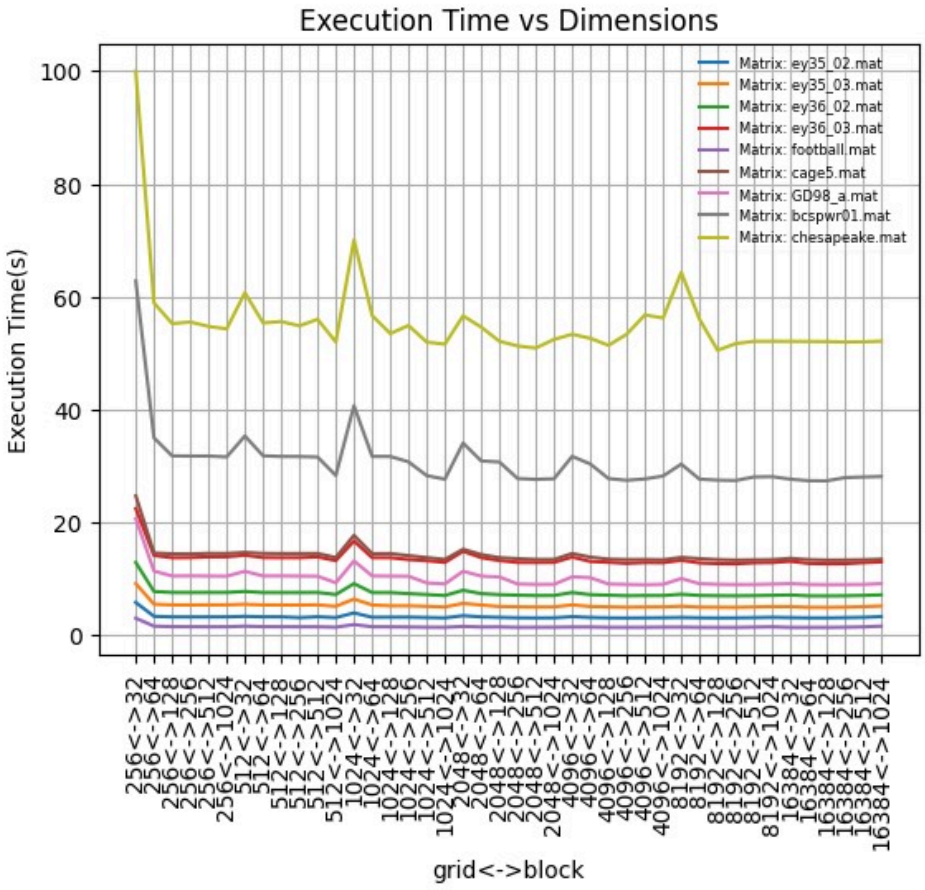

Fig 3. Execution Time (y-axis), Grid and Block Size (x-axis)

Further Improvements

In binary matrices, SkipPer algorithm is much faster than SpaRyser. Therefore, checking if the matrix is binary and utilizing SkipPer accordingly could improve the efficiency. One other improvement can be using CPU and GPU together with distributing some load into CPU cores as well when executing our CUDA implementation. This will definitely decrease the execution

time; however, it is important to calculate the correct throughput of the CPU execution compared to GPU cores and balance the load accordingly.

References

Kaya, K. (2019). Parallel algorithms for computing sparse matrix permanents. Turkish Journal of Electrical Engineering and Computer Sciences, 27, 4284–4297.

Lint, J. H. van, & Wilson, R. M. (2001). A course in Combinatorics. Cambridge University Press.

Yağlıoğlu, B. (2021). Computing matrix permanents and counting perfect matchings on GPUs [Master's thesis]. https://research.sabanciuniv.edu/id/eprint/42490/1/10364196.pdf