COMPUTING MATRIX PERMANENTS AND COUNTING PERFECT MATCHINGS ON GPUS

by
BERK YAĞLIOĞLU

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
July 2021

COMPUTING MATRIX PERMANENTS AND COUNTING PERFECT MATCHINGS ON GPUS

Approved by:

Date of Approval: July 14, 2021

ABSTRACT

COMPUTING MATRIX PERMANENTS AND COUNTING PERFECT MATCHINGS ON GPUS

BERK YAĞLIOĞLU

Computer Science and Engineering M.S. THESIS, JULY 2021

Thesis Supervisor: Asst. Prof. Kamer Kaya

Keywords: permanent, perfect matching, bipartite graph, GPU, High Performance Computing, #P-complete

Permanent -just like determinant-, is an important numeric value in order to understand matrix characteristics and multiple applications of permanent exist. For example, because graphs and sparse matrices are structurally similar to each other, they can be used to show the representation of the same data. The Permanent value of a symmetric matrix that is consisted of 1s and 0, is equal to the perfect matching number of the corresponding bipartite graph which is an important information of relationship among bipartite graph's vertices.

Calculating exact value of matrix permanent is a #P-complete problem. For that reason, there is not an algorithm that works in polynomial time. The fastest algorithm that calculates an $n \times n$ matrix's permanent value has a time complexity of $O(2^{n-1}n)$. This nature of the problem makes the calculation of even considerable smaller matrices like $n > 40$ very slow. In literature, there exist studies that focuses on computing the exact permanent value in parallel with a computer or a supercomputer.

In this thesis, parallel algorithms are designed and implemented that can calculate the exact permanent value of dense and sparse matrices efficiently on multicore CPUs and multiple GPUs. Furthermore, algorithms are developed to approximate the permanent value of a given dense or sparse matrix in parallel.

ÖZET

BIRDEN FAZLA GPU ÜZERINDE PERMANENT DEĞERININ HESAPLANMASI VE MÜKEMMEL EŞLEMELERIN SAYILMASI

BERK YAĞLIOĞLU

PROGRAM ADI YÜKSEK LİSANS TEZİ, MAYIS 2021

Tez Danışmanı: Asst. Prof. Kamer Kaya

Anahtar Kelimeler: permanent, mükemmel eşleme, iki parçalı çizge, GPU, Yüksek Başarımlı Hesaplama, #P-tam

Permanent aynı determinant gibi matrislerin karakterlerini anlamaya yarayan önemli bir sayısal değerdir ve bir çok pratik uygulaması mevcuttur. Örneğin, çizgeler ve seyrek matrisler yapısal olarak birbirine benzediğinden aynı veriyi göstermek üzere kullanılabilirler ve 1 ve 0'lardan oluşan bir simetrik matrisin permanent değeri, o matrise karşılık gelen iki parçalı çizgenin mükemmel eşleme sayısına eşittir. İki parçalı çizgenin mükemmel eşleme sayısı, o çizgenin noktaları arasındaki ilişkisi adına önemli bir bilgidir.

Permanent değerini hesaplama #P-tam bir problemdir. Bu yüzden polinom zamanda çalışan bir algoritma bulunmamaktadır. Literatürdeki en hızlı algoritmanın karmaşıklığı $O(2^{n-1}n)$'dır. Problem bu yapısı gereği $n > 40$ gibi küçük denilebilecek matrislerin için bile permanentin hesaplanması oldukça yavaştır. Literatürde permanent değerini bilgisayar veya süper bilgisayar ile paralel olarak hesaplamak adına çalışmalar mevcuttur.

Bu tezde hem dolu hem seyrek matrisler için birden çok çekirdeki CPU'lar ve birden çok GPU'da çalışan paralel algoritmalar tasarlanıp geliştirilmiştir. Ayrıca dolu ve seyrek matrisler için permanent değerini yakınsayan paralel algoritmalar geliştirilmiştir.

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor Dr. Kamer Kaya for everything he taught and his support, and my family and friends for their support throughout my studies in Sabancı University.

To my family and friends

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Permanent is the sum of products of all transversals of a matrix. This immanent of the matrices has important applications in many scientific areas such as complexity theory, graph theory, game theory, and quantum computing. The exact permanent of an adjacency matrix is equivalent to the number of perfect matching for a bipartite graph, and the number of cycle covers for a directed graph. The exact permanent values are also used in computing the transition amplitude of a quantum circuit (Rudolph, 2009) and boson sampling (Aaronson & Arkhipov, 2010). Valiant has shown that, computing the exact permanent identifies the arithmetic version of complexity class NP, called VNP. (Valiant, 1979). In the same paper, Valiant proved that computing permanent is #P-complete problem. The most efficient algorithm known for calculating the exact permanent has $O(2^{n-1}n)$ complexity, therefore it takes a tremendous amount of time or even impossible to calculate the exact permanent of big matrices. Since calculating the exact permanent is a costly operation, there are numerous researches about approximating permanent in the literature. Among the approximation algorithms, approaches that use Markov chains to approximately sample from a distribution of weighted permutations and provide polynomial runtime with an acceptable error are proven to be the most successful (Huber & Law, 2008). However, despite the heavy computational cost, there is no work in the literature investigating high-performance computing techniques to deliver fast and computationally efficient algorithms in order to greatly reduce the time required for the permanent calculation and make exact calculation and approximation of big matrices plausible except (Kaya, 2019). This thesis aims to close this gap in the literature. To this end, several high-performance algorithms are implemented and analyzed for permanent calculation problems. Currently, calculating the exact permanent of 40×40 matrix takes 32497.7 seconds with the best-known algorithm on a single CPU thread. With techniques proposed in this thesis, 2702.9 seconds on 16 CPU threads, 259.5 seconds on a single GPU, and 96.4 seconds with 4 GPUs are obtained for the same calculation. The contributions of this thesis can be summarized as the following:

- High-performance permanent calculation algorithms for dense matrices on multicore CPUs and/or multiple, manycore GPUs.
- High-performance permanent calculation algorithms for sparse matrices on multicore CPUs and/or multiple, manycore GPUs.
- High-performance permanent approximation algorithms for dense matrices on multicore CPUs and/or multiple, manycore GPUs.
- High-performance permanent approximation algorithms for sparse matrices on multicore CPUs and/or multiple, manycore GPUs.
- A detailed analysis of the performance of developed algorithms, problems encountered while parallelizing the computation over different architectures, and proposed solutions.
- **SUPerman**, a publicly available HPC tool that includes every algorithm proposed in this thesis to help other researchers.

The rest of the thesis is organized as follows: In Chapter 2, background on the calculation of the exact permanent, approximation, notation used in this thesis, a literature review, and architectural details of the used hardware accelerators are given. In the chapter 3, algorithms are proposed for the exact permanent calculation for dense and sparse matrices. This chapter also provides the variations of algorithms that leverages optimized memory access patterns. In Chapter 4, high-performance versions of approximation algorithms that run on single and multiple GPU devices and their variations optimized for dense and sparse matrices are proposed. In Chapter 5, GPU capable algorithms for counting perfect matchings are proposed. In Chapter 6, performance analysis is provided for every algorithm proposed in this thesis. Chapter 7 concludes the thesis.

2. BACKGROUND AND NOTATION

2.1 Permanents and Matchings

For a given $n \times n$ matrix $\mathbf{A}$, let $a_{i,j}$ be the entry at the ith row and jth column of $\mathbf{A}$. The permanent value of $\mathbf{A}$ can be calculated using the following equality.

$$\texttt{perm}(\mathbf{A}) = \sum_{\sigma \in \mathcal{P}} \prod_{i=1}^{n} a_{i,\sigma(i)} \tag{2.1}$$

where $\mathcal{P}$ is the permutation of all the numbers within $1, 2, \ldots, n$. Due to this nature of the algorithm, it has to go over all the permutations, which leads to $\mathcal{O}(nn!)$ complexity. The Ryser algorithm (Ryser, 1963) makes use of inclusion and exclusion principle to reduce to time complexity to $\mathcal{O}(2^{n-1}n^2)$. The Ryser algorithm changes the equation of calculating permanent as follows.

$$\texttt{perm}(\mathbf{A}) = -1^n \sum_{S \subseteq \{1,2,\cdots,n\}} (-1)^{|S|} \prod_{i=1}^{n} \sum_{j \in S} a_{i,j} \tag{2.2}$$

Nijenhuis and Wilf improved this algorithm even further by processing the S sets in the order of gray code (Nijenhuis & Wilf, 1978). They reduced the complexity to the $\mathcal{O}(2^{n-1}n)$. This algorithm is the most efficient algorithm in the literature currently for dense and almost full matrices. In the rest of the thesis, this algorithm will be denoted as `Ryser`. A parallel implementation of `Ryser` on CPU will be denoted as `ParRyser`. Gray code will be denoted as `GrayCode`, and ith gray code will be denoted as $\texttt{GrayCode}_i$.

For dense matrices, most of the permutations or most of the sets in the S makes a contribution to the permanent that is different than 0. However, most of the matrices in real life applications are sparse with very few nonzero elements, and it is a complex problem to calculate the permanent value of a sparse matrix. Because, as opposed to a dense matrix, the number of permutations or sets in S that contributes to the permanent value is very low in sparse matrices. One can take into account only the sets of S that contributes to the permanent. However, determining those sets is not an easy problem. Most recently (Kaya, 2019), two algorithms `SpaRyser` and `SkipPer` are proposed that are efficient on sparse matrices, where both the algorithms are based on the `Ryser` algorithm. The `SpaRyser` algorithm makes use of compressed row and compressed column storage denoted as `CRS` and, `CCS` which are initially proposed by Mittal and Al-Kurdi for use of permanent calculation (Mittal & Al-Kurdi, 2001). In `SpaRyser` algorithm, a single iteration can detect efficiently whether the current iteration is a zero contribution iteration for the permanent or not. If there is no contribution in the current one, the algorithm skips to the next iteration by avoiding unnecessary multiplications. On the other hand, in `SkipPer`, it further skips many *consecutive* iterations with a zero contribution by a single jump when a `GrayCode` is detected yielding a zero contribution. In this thesis, their parallel implementations are designed and implemented, which will be denoted as `ParSpaRyser` and `ParSkipPer`. In `ParSkipPer`, a dynamic iteration-chunk-to-thread scheduling is applied to employ a thread with a chunk when it finishes its previous task.

There have been studies in the literature to estimate the number of perfect matchings in a bipartite graph, where a matching is *perfect* as long as every edge in the matching covers each vertex exactly once. Calculating the number of perfect matchings in a bipartite graph is equivalent to the permanent value of the binary 0-1 adjacency matrix that corresponds to the bipartite graph. The Rasmussen algorithm (Rasmussen, 1994) estimates the permanent value of a given matrix using random selection mechanism at each step. At each iteration the first row is chosen and the number of nonzero of this row is used to multiply the permanent value which is initially 1. Then, one of the nonzero in this first row is chosen and its row and column is extracted from the matrix. The algorithm continues until there is no row/column left or one of the row is having no nonzero elements, which leads to 0 as a value of the estimation. This algorithm will be denoted as `Rasmussen`. Different approaches using `Rasmussen` algorithm are also proposed (Dufossé, Kaya, Panagiotas & Uçar, 2018). They improved `Rasmussen` by eliminating edges that do not contribute perfect matching, and used a heuristic to choose the row with the least nonzero at each step. This improved version will be denoted as `Rasmussen+`. Similar to `Rasmussen`,

they proposed a variant for estimating permanent which makes use of a scaling process to scale the matrix in a doubly stochastic form at each iteration. After first row is chosen at each iteration and edges that cannot contribute to the permanent are eliminated, diagonal entries of scaling matrices are updated which then to be used in contribution to the permanent. This algorithm is denoted as `Scaling`. An improved version of this algorithm that uses the same heuristic as `Rasmussen+` to choose the row with the least nonzero at each step is also denoted as `Scaling+` in this paper.

In the most of the experiments, the input matrix whose permanent value to be calculated will be denoted as `mat`. For a matrix, the dimension n indicates that the matrix is $n \times n$ matrix, and it is denoted as `dim`. The number of nonzero elements inside a matrix is denoted as `nnz`.

2.2 Graphics Processing Units

A Graphics Processing Unit, that is called shortly GPU, is a hardware device that is used in applications containing graphics heavily. Nowadays, GPUs are also used in High Performance Computing for programs to run faster. In contrast to a CPU that is a Central Processing Unit, a GPU contains many cores that focuses on completing many tasks at once by hiding the latency using communication-computation overlap. There are multiple Processor Clusters in a GPU architecture where each consists of multiple Streaming Multiprocessors that is called an SM. An SM contains memory cache layers which are smaller than a cache in a CPU. Each SM in a GPU runs on its cores multiple blocks that consist of threads, where all the blocks together form a grid. Therefore, in terms of hierarchy in a GPU, grid is the largest (abstract) structure, and it has a dimension that indicates the number of blocks, which will be denoted as `GridDim` in this paper. Then, each block has many threads inside of it as the number of its dimension, which will be denoted as `BlockDim`. Therefore, it can be said that there will be `GridDim` $\times$ `BlockDim` many threads running in total in a GPU. Within a block, there is another form of group that is called *warp* that consists of 32 threads. The important thing to note about a warp is that it is a collection of 32 threads that are synchronized and running the same commands at the same time, such that if they encounter an **if-else** block, some will wait to enter in else block if there are other threads in the warp that enter in if block. Furthermore, threads in a warp tend to run faster when they access the same memory locations or

the memory location that lie next to each other. This last pattern is called memory coalescing.

In terms of memory hierarchy, there are three levels. A variable can be kept in the global memory, where any thread in the grid can access and update this memory. Reading an updating a global memory location is slow since it is in a place for each thread to access from different SMs. On the other hand, there is also *shared memory* which is special to each block. Therefore, a shared memory region can be allocated for each block, where only the threads within the block can access and update this memory. Read write operations to this memory region can be performed very fast. However, there is a size limitation on the memory that can be allocated for a block. There is also another form of memory that is called *registers* which is only accessible by a single thread, such that it is special to a thread. This memory region also can be accessed and updated very fast, whereas there is a limitation for this memory region too. In the paper, the GPU devices used are TITAN X (Pascal) which will be denoted as `TITAN`, and GeForce GTX 980 which will be denoted as `Gtx`. For these GPU devices, shared memory limitation is $48KB$ per block, such that there cannot be a memory region allocated more than $48KB$ for each block. In terms of limitation on registers, 65536 registers are available per block. Maximum `BlockDim` is 1024, and a single SM can take up to maximum of 2048 threads. For example, when `BlockDim` is 1024, 2 blocks can fit into a single SM.

For the code that runs on a GPU, CUDA is used, which is an application programming interface that enables running codes on a GPU. In order to start a GPU device for the experiments, a global device kernel should be created where it is able to be called from the code that runs in the host machine on a CPU. This device kernel is started by a single thread on a CPU by inputting `GridDim`, `BlockDim`, and necessary parameters that are allocated using CUDA. Initially allocated memory using CUDA lies in the global memory when the kernel starts for each thread. However, a shared memory can be created inside of the kernel and values can be copied from global memory to shared memory in the beginning of the kernel. In the end, if there are many output values to return, all output values can be written to a memory region in the global memory, which then to be copied back to host memory on a CPU using CUDA. In this thesis, global thread ID within all of the threads on a GPU is denoted as `tid`. On the other hand, a thread ID of a thread within a block is denoted as `threadId`.

3. COMPUTING MATRIX PERMANENTS ON GPUs

The most efficient algorithm for calculating matrix permanent is `Ryser` which has a run time complexity of $\mathcal{O}(2^{n-1}n)$. In this section, GPU implementations of `Ryser` and `SpaRyser` algorithms will be discussed for dense and sparse matrices, respectively.

3.1 Computing the Permanents of Dense Matrices

In the `Ryser` algorithm, the for loop which has $\mathcal{O}(2^{n-1})$ complexity can be parallelized in a data-parallel fashion. The implementation for `ParRyser` is given in Algorithm 1. The input parameters of this algorithm are the matrix mat, dimension of the matrix dim, $start$ and end variables for the iteration space which are 1 and 2^{dim-1} respectively for the total iteration space. This iteration space can be divided into chunks. When there are multiple threads, a chunk size can be chosen as the iteration size divided by the number of threads. Then, each thread needs to use the previous chunk's last `GrayCode` that is $\texttt{GrayCode}_{myStart-1}$ to calculate its initial $\mathbf{x}$, where $myStart$ is the start index for each thread in Algorithm 1. Then, the initial $\mathbf{x}$ is obtained using the columns of the matrix that corresponds to the bits of the current `GrayCode` which are 1. Afterwards, the rest of the algorithm is the same as `Ryser` for each thread with its own start, endpoints, and $\mathbf{x}$. One important thing to note is the use of $matT$, which is the transpose of the original matrix in order to iterate over columns more efficiently at each iteration.

For the GPUs, `Ryser` algorithm is implemented using CUDA. In the host machine, memory is allocated for the input variable of the device kernel and copied those input variables' initial values from host to global device memory. These input variables for the device kernel are x which is $\mathbf{x}$, $matT$ that is a transpose of the original matrix, dim that is the dimension of the matrix, p as an output list to store the

final permanent value each thread finds in its own chunk of work, and *start* and *end* positions for the start and end point of the iteration space in the current kernel call. The output list is created as the size of the number of threads, which is `GridDim` × `BlockDim`. After each thread calculates its own result for the chunk it has given, this result is stored in the index of the global thread ID on the GPU which is `tid`. After the device kernel terminates, the result list is copied back to host memory, and results are summed up in the host to obtain the permanent value of the matrix.

Algorithm 1 `ParRyser` $(mat, dim, start, end)$

1: $p \leftarrow 1$
2: **for** $i = 1 \ldots N$ **do**
3: $\quad rowSum \leftarrow 0$
4: $\quad$ **for** $j = 1 \ldots N$ **do**
5: $\quad\quad rowSum \leftarrow rowSum + mat[i,j]$
6: $\quad x[i] \leftarrow mat[i,n] - rowSum/2$
7: $\quad p \leftarrow p \times x[i]$
8: $matT \leftarrow transpose\ of\ the\ mat$
9: **for** *each thread* **do**
10: $\quad myX \leftarrow x$
11: $\quad myP \leftarrow 0$
12: $\quad myStart \leftarrow start + \texttt{threadId} \times chunkSize$
13: $\quad myEnd \leftarrow min(start + (\texttt{threadId} + 1) \times chunkSize, end)$
14: $\quad$ calculate **x** using the previous `GrayCode`
15: $\quad$ **for** $g = myStart \ldots myEnd - 1$ **do**
16: $\quad\quad j \leftarrow \log_2(\texttt{GrayCode}_g \oplus \texttt{GrayCode}_{g-1}) + 1$
17: $\quad\quad s \leftarrow 2 \times \texttt{GrayCode}_g[j] - 1$
18: $\quad\quad prod \leftarrow 1$
19: $\quad\quad$ **for** $i = 1 \ldots N$ **do**
20: $\quad\quad\quad myX[i] \leftarrow myX[i] + (s \times matT[j,i])$
21: $\quad\quad\quad prod \leftarrow prod \times myX[i]$
22: $\quad\quad myP \leftarrow myP + (-1)^g \times prod$
23: $\quad AtomicAdd(p, myP)$
24: **return** $p \times (4 \times (n \bmod 2) - 2)$

On a GPU, it is crucial to access and manipulate data fast. A warp on a GPU is the collection of 32 threads, where they all work in a synchronized manner. Therefore, threads in a warp execute the same commands at the same time. This synchronization is the reason behind condition statements being problematic in terms of performance on GPUs. It is also important to note that threads in a warp execute faster when they are accessing memory locations that are the same or close to each other. Based on this feature, a novel approach is followed in this thesis aimed to obtain faster memory access by utilizing a feature of `GrayCode`. In every iteration of `Ryser`, it makes use of `GrayCode` the difference to find the column ID of the

matrix to update the **x**. If every thread in a warp utilizes the same column at the same iteration, execution can get faster, since threads in a warp are faster when they use the same or narrow memory locations together at the same time. Luckily, one can achieve this by properly choosing the chunk size each thread is employed with. When calculating the $(i+1)$th `GrayCode`, always one bit is flipped in the ith `GrayCode`. The observation is that mostly the same bit is flipped if one move from $(i+2^k)$th gray code to $(i+2^k+1)$th where k is a positive integer. In Figure 3.1, 8-bit gray code is shown describing this condition by indicating the flipped bits. Therefore, if the chunk size each thread is employed with is chosen as the exact power of 2, each thread in a warp will operate on the same column in most of the iterations at the same time.

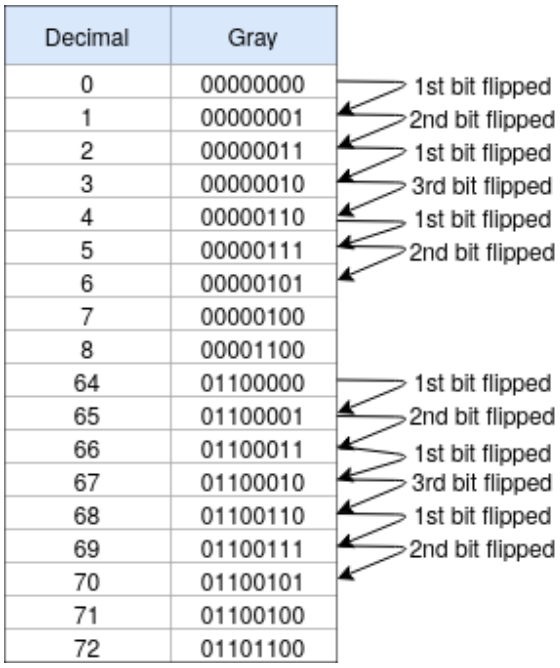


Figure 3.1 Gray codes

3.1.1 `Ryser-Gx`: keeping x in global device memory

In `Ryser-Gx`, the initial **x**, which was created on the host and copied to the global device memory, is used by each thread. For each thread to use **x** in the global device memory, there should have been a separate **x** created for each thread, since each thread will update it differently according to the portion of the iteration space they process. For this reason, **x** is allocated on the global device memory as the size of `dim` $\times$ `GridDim` $\times$ `BlockDim` and threads are able to access their own portion of the **x** using their global thread ID `tid`. In Algorithm 2, the device kernel of `Ryser-Gx`

is given. As it can be seen, $\mathbf{x}$ is utilized from global memory using x that comes as a parameter. Also, $matT$ which is the transpose of the matrix is also accessed directly using global memory accesses. When a thread wants to access the ith index of the $\mathbf{x}$, it accesses using $x[\texttt{tid} \times \texttt{dim} + i]$. Also, the ith row and jth column of the original matrix can be accessed using $matT[j \times \texttt{dim} + i]$. The rest of the algorithm is similar to the `ParRyser`.

Algorithm 2 `Ryser-Gx` $(matT, dim, x, p, start, end)$

1: $\texttt{tid} \leftarrow threadIdx.x + (blockIdx.x \times blockDim.x)$
2: $chunkSize \leftarrow (end - start)/(\texttt{GridDim} \times \texttt{BlockDim}) + 1$
3: $myStart \leftarrow start + \texttt{tid} \times chunkSize$
4: $myEnd \leftarrow min(start + (\texttt{tid} + 1) \times chunkSize, end)$
5: $myP \leftarrow 0$
6: **for** $k = 1 \ldots dim - 1$ **do**
7: **if** $(\texttt{GrayCode}_{myStart-1} >> k)\&1$ **then**
8: **for** $j \ldots dim$ **do**
9: $x[\texttt{tid} \times \texttt{dim} + j] \leftarrow x[\texttt{tid} \times \texttt{dim} + j] + matT[k \times dim + j]$
10: **for** $g = myStart \ldots myEnd - 1$ **do**
11: $j \leftarrow \log_2(\texttt{GrayCode}_g \oplus \texttt{GrayCode}_{g-1}) + 1$
12: $s \leftarrow 2 * \texttt{GrayCode}_g[j] - 1$
13: $prod \leftarrow 1$
14: **for** $i = 1 \ldots dim$ **do**
15: $x[\texttt{tid} \times \texttt{dim} + i] \leftarrow x[\texttt{tid} \times \texttt{dim} + i] + (s \times matT[j \times dim + i])$
16: $prod \leftarrow prod \times x[\texttt{tid} \times \texttt{dim} + i]$
17: $myP \leftarrow myP + (-1)^g * prod$
18: $p[\texttt{tid}] \leftarrow myP$

3.1.2 `Ryser-Rx`: keeping x on registers

In `Ryser-Rx`, the initial $\mathbf{x}$ is created in the device registers, which means locally created and copied its initial values from the global device memory which comes as input x in Algorithm 2. In the device kernel of `Ryser-Rx`, x parameter has the size of `dim` since it only contains the initial values of $\mathbf{x}$ as opposed to `Ryser-Gx`. Therefore, myX is created locally in `Ryser-Rx` as the size of `dim` and x is copied to myX before line 6 in Algorithm 2. The rest of the device kernel of `Ryser-Rx` is the same as the device kernel of `Ryser-Gx`, except the ith index of $\mathbf{x}$ will be accessed using $myX[i]$ by each thread in `Ryser-Rx`. Although GPU devices have fast memory accesses for registers, when matrix size is increased, this version has limitations due to the limited number of the registers per streaming multiprocessor(SM), since $\mathbf{x}$ has the length as the matrix dimension.

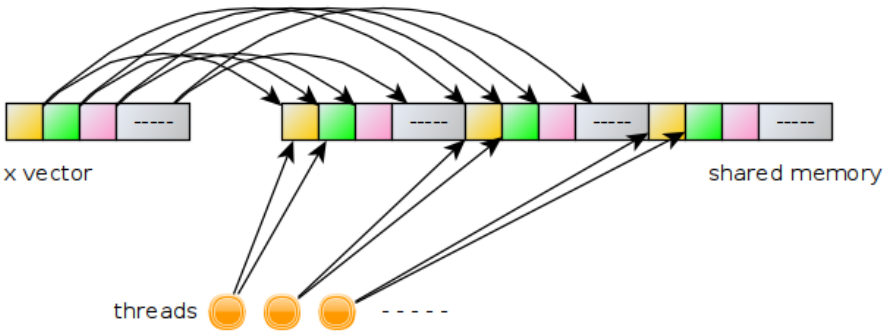


Figure 3.2 Moving **x** to shared memory

3.1.3 `Ryser-Sx`: keeping x in the shared memory

The device kernel of `Ryser-Sx` is given in Algorithm 3. In `Ryser-Sx`, shared memory is allocated for each thread's **x**, and initial parameter x in Algorithm 3 is copied into this shared memory. Allocating memory for **x** in the shared memory means that memory will be allocated in the shared memory of each block as the size of the `dim` $\times$ `BlockDim` since each thread in a block on GPU maintains a different **x**, while $matT$ and result list p is in the global device memory. After the memory is allocated in the shared memory, initial parameter x is copied by each thread in a block to the shared memory location of **x** for the specified thread, as in line 5. In `Ryser-Sx`, **x** of each thread in the shared memory is lying next to each other, as it is illustrated in Figure 3.2. Therefore, in order to access the ith index of the **x** in the shared memory, a thread in a block should use $myX[\texttt{threadId} \times \dim + i]$. Since the transpose of the matrix is in the global device memory, $matT$ is used directly to access the elements of the matrix. The rest of the algorithm is the same as `Ryser-Rx` except the memory accesses to myX. The aim to use **x** in shared memory is to utilize **x** faster by accessing elements faster. However, this approach also comes with a cost due to limitations of the shared memory available per block, which is $48KB$. This limits the number of threads to be able to use per block. Because **x** has a length of `dim` for each thread. For example, when a data type of double is used for **x**, then `dim` $\times$ `BlockDim` $\times$ 8 bytes of space is consumed per block, which should be smaller than $48KB$ shared memory. This means, when `dim` equals 36-40, only 150-170 threads can fit into one block. This means also one block can fit into one SM since the shared memory limitation is also $48KB$ per SM. However, one SM can take up to 2048 threads inside of it, and this is a must if one wants to hide the latency of the works between threads. Therefore, this is also a huge limitation of

this approach. In the experiments, **x** is used as type of float, which uses less space as 4 bytes, in order to fit more threads inside of a block. When **x** is a type of the float, it is possible to fit 305-340 threads into one block.

Algorithm 3 `Ryser-Sx` $(matT, dim, x, p, start, end)$

1: $\texttt{tid} \leftarrow threadIdx.x + (blockIdx.x \times blockDim.x)$
2: $\texttt{threadId} \leftarrow threadIdx.x$
3: $\texttt{BlockDim} \leftarrow blockDim.x$
4: **for** $k = 1 \ldots dim$ **do**
5: $\quad myX[\texttt{threadId} \times \dim + k] \leftarrow x[k]$
6: $syncthreads()$
7: $chunkSize \leftarrow (end - start)/(\texttt{GridDim} \times \texttt{BlockDim}) + 1$
8: $myStart \leftarrow start + \texttt{tid} \times chunkSize$
9: $myEnd \leftarrow min(start + (\texttt{tid} + 1) \times chunkSize, end)$
10: $myP \leftarrow 0$
11: **for** $k = 1 \ldots dim - 1$ **do**
12: $\quad$ **if** $(\texttt{GrayCode}_{myStart-1} >> k)\&1$ **then**
13: $\quad\quad$ **for** $j \ldots dim$ **do**
14: $\quad\quad\quad myX[\texttt{threadId} \times \dim + j] \leftarrow myX[\texttt{threadId} \times \dim + j] + matT[k \times dim + j]$
15: **for** $g = myStart \ldots myEnd - 1$ **do**
16: $\quad j \leftarrow \log_2(\texttt{GrayCode}_g \oplus \texttt{GrayCode}_{g-1}) + 1$
17: $\quad s \leftarrow 2 * \texttt{GrayCode}_g[j] - 1$
18: $\quad prod \leftarrow 1$
19: $\quad$ **for** $i = 1 \ldots dim$ **do**
20: $\quad\quad myX[\texttt{threadId} \times \dim + i] \leftarrow myX[\texttt{threadId} \times \dim + i] + (s \times matT[j \times dim + i])$
21: $\quad\quad prod \leftarrow prod \times myX[\texttt{threadId} \times \dim + i]$
22: $\quad myP \leftarrow myP + (-1)^g * prod$
23: $p[\texttt{tid}] \leftarrow myP$

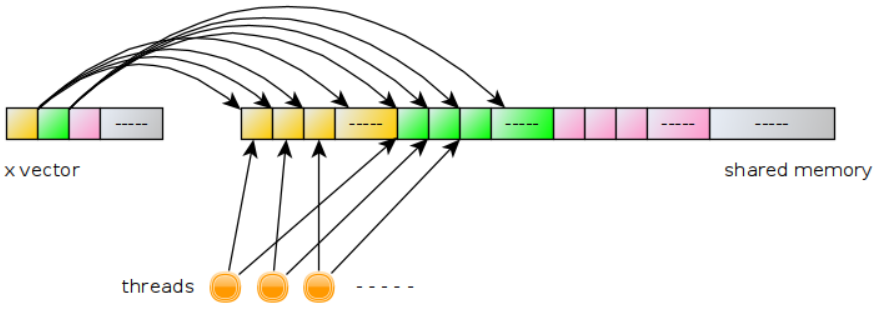


Figure 3.3 Moving **x** to shared memory with memory coalescing

3.1.4 `Ryser-SxC`: keeping x in the shared memory with memory coalescing

`Ryser-SxC` algorithm has a very similar approach as `Ryser-Sx`. Both approaches utilize faster element accesses using shared memory for $\mathbf{x}$, while they use a matrix from the global device memory accessing $matT$ in Algorithm 3. The only difference is that in `Ryser-Sx`, each thread's $\mathbf{x}$ lies next to each other in the shared memory of a block. This leads to threads in a warp accessing a wider memory region. As it was previously described, threads in a warp execute faster when they use narrower memory regions and accessing locations that are closer to each other at the same time. This is called memory coalescing and this is achieved by putting a specific index of each $\mathbf{x}$ that each thread use next to each other in the shared memory, as is illustrated in Figure 3.3. Therefore, threads in a warp are accessing a smaller area on the shared memory in every command. Therefore, in order for a thread to access the ith element of the $\mathbf{x}$, it should access to $myX[\texttt{BlockDim} \times i + \texttt{threadId}]$.

Algorithm 4 `Ryser-SxC-Sm` $(matT, dim, x, p, start, end)$

1: $\texttt{tid} \leftarrow threadIdx.x + (blockIdx.x \times blockDim.x)$
2: $\texttt{threadId} \leftarrow threadIdx.x$
3: $\texttt{BlockDim} \leftarrow blockDim.x$
4: **for** $k = 1 \ldots dim$ **do**
5: $\quad myX[\texttt{BlockDim} \times k + \texttt{threadId}] \leftarrow x[k]$
6: **for** $k = 1 \ldots (dim \times dim)/\texttt{BlockDim} + 1$ **do**
7: $\quad$ **if** $\texttt{BlockDim} \times k + \texttt{threadId} < dim \times dim$ **then**
8: $\quad\quad sharedMatT[\texttt{BlockDim} \times k + \texttt{threadId}] \leftarrow matT[\texttt{BlockDim} \times k + \texttt{threadId}]$
9: $syncthreads()$
10: $chunkSize \leftarrow (end - start)/(\texttt{GridDim} \times \texttt{BlockDim}) + 1$
11: $myStart \leftarrow start + \texttt{tid} \times chunkSize$
12: $myEnd \leftarrow min(start + (\texttt{tid} + 1) \times chunkSize, end)$
13: $myP \leftarrow 0$
14: **for** $k = 1 \ldots dim - 1$ **do**
15: $\quad$ **if** $(\texttt{GrayCode}_{myStart-1} >> k)\&1$ **then**
16: $\quad\quad$ **for** $j \ldots dim$ **do**
17: $\quad\quad\quad myX[\texttt{BlockDim} \times j + \texttt{threadId}] \leftarrow myX[\texttt{BlockDim} \times j + \texttt{threadId}] + sharedMatT[k \times dim + j]$
18: **for** $g = myStart \ldots myEnd - 1$ **do**
19: $\quad j \leftarrow \log_2(\texttt{GrayCode}_g \oplus \texttt{GrayCode}_{g-1}) + 1$
20: $\quad s \leftarrow 2 * \texttt{GrayCode}_g[j] - 1$
21: $\quad prod \leftarrow 1$
22: $\quad$ **for** $i = 1 \ldots dim$ **do**
23: $\quad\quad myX[\texttt{BlockDim} \times i + \texttt{threadId}] \leftarrow myX[\texttt{BlockDim} \times i + \texttt{threadId}] + (s \times sharedMatT[j \times dim + i])$
24: $\quad\quad prod \leftarrow prod \times myX[\texttt{BlockDim} \times i + \texttt{threadId}]$
25: $\quad myP \leftarrow myP + (-1)^g * prod$
26: $p[\texttt{tid}] \leftarrow myP$

3.1.5 `Ryser-SxC-Sm`: keeping `mat` in the shared memory in `Ryser-SxC`

In `Ryser-SxC-Sm` the algorithm, the only additional feature is that a memory region is allocated also for the matrix along with **x** in the shared memory. Therefore, it can be said that everything is the same as `Ryser-SxC` except how `Ryser-SxC-Sm` to access its matrix. It is important to keep the matrix in the shared memory since there are lots of accesses to the matrix and one can utilize fast accesses using the shared memory. Algorithm 4 shows the pseudocode of the device kernel of `Ryser-SxC-Sm` in detail. Similar to previous approaches in `Ryser-Sx` and `Ryser-SxC`, there is a shared memory limitation which is even greater since the matrix is also included in the shared memory of each block. If **x** has type off the float, when the matrix uses values with the type of integer, 265-305 threads can fit into one block when matrix size is 36-40. If the type of double is used for the values in the matrix, this will lead to even fewer threads per block, such as 230-270.

3.1.6 `Ryser-SxC-Sm-MG`: static multiple GPUs implementation

In `Ryser-SxC-Sm`, the aim was to run permanent calculation on a single GPU device. However, since iteration space can be divided into chunks for each thread, one can also divide it equally for each GPU device, and run them separately. In `Ryser-SxC-Sm-MG`, total iteration space is divided equally to the number of devices and each device calculates its own portion using the `Ryser-SxC-Sm`, whose device kernel is given under Algorithm 4. Then, in the end, all outcomes of each device are summed up to obtain a final permanent value.

3.1.7 `Ryser-SxC-Sm-MG+`: dynamic hybrid implementation

In `Ryser-SxC-Sm-MG`, it divides iteration space among multiple GPU devices. However, there is an issue in performance if one device is faster than the other one. In such a case, the faster device could finish the work earlier and wait for the other devices, since the slower ones are still working. This can be avoided with further optimization by using small chunks for each device to start with instead of dividing the iteration space as the number of devices. This approach with dividing iteration space into many small chunks is followed in `Ryser-SxC-Sm-MG+`. In a loop, after

every device finishes with its own chunk using the kernel of `Ryser-SxC-Sm`, then it will take the next chunk if there is any left. By this approach, if one device finishes its chunk earlier than others, it will continue to work until there is no chunk left. It is also important to choose the chunk size good. Because if one chooses a large space for a single chunk, the last chunk could be taken by a slow device, and it may take longer while others are free of work. On the other hand, if one chooses a very small space for a chunk, this would give an extra overhead since each device comes with a cost to initialize and copy all the results back to host memory after it is done. Therefore, it is crucial to find a medium-size chunk. The number of chunks for this algorithm is detected after manual experiments to be $2^{\texttt{dim}-29}$. In addition to the implementation of this algorithm, also a CPU kernel is added which is `ParRyser` that optionally takes a chunk and runs along with GPU devices to contribute and make the permanent calculation process faster.

3.2 Computing the Permanents of Sparse Matrices

When matrices are sparse, previous approaches begin to work inefficiently. The reason is that most of the elements inside the matrix are zero and ineffective due to them having no contribution in updating $\mathbf{x}$. However, there are still unnecessary memory accesses are being made to these elements that are zero. Therefore, a sparse data structure should be used to store the matrix. Then, the algorithm should be adjusted accordingly. For sparse matrices, appropriate data structures are `CRS` and `CCS` as they were being used in `SpaRyser` algorithm. In these data structures, there are three arrays used for each `CRS` and `CCS` as it can be seen in Figure 3.4. For an example, arrays named as *rptrs*, *columns*, *rvals* in `CRS`. *rptrs* is used to store the start location of each row in the *columns* and *rvals* arrays. *columns* array is used to store the column ID of each nonzero element in the specified row. Similarly, *rvals* array is used to store each nonzero element's value.

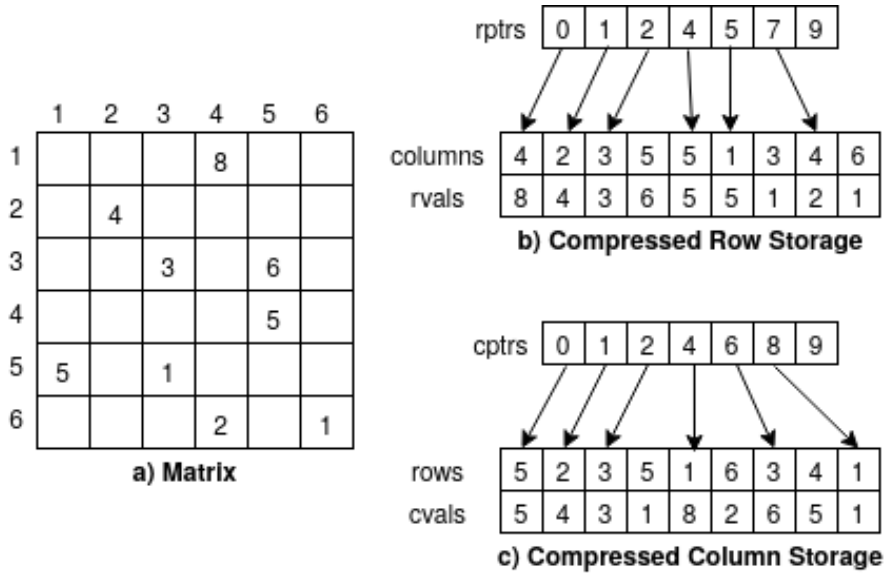


Figure 3.4 (a)A 6×6 matrix and its (b)`CRS` and (c)`CCS` representations

`SpaRyser` algorithm can be parallelized in the same manner as, `Ryser` since the original `Ryser` algorithm is being utilized. Implementation of the `ParSpaRyser` is given in Algorithm 5. As input parameters, `CCS` is given along with the matrix. Dimension of the matrix, start and end points for the iteration space are also given. The matrix itself is only used in initializing the $\mathbf{x}$. For the main loop, only `CCS` is needed, since each thread only operates on a column of the matrix at each iteration to update the $\mathbf{x}$ and contribute to the permanent.

In `SpaRyser`, using `CCS`, instead of iterating over all of the elements in a column of a matrix, `CCS` is being used only to iterate over nonzero elements. After iterating only nonzero elements updating $\mathbf{x}$, if there is any zero in $\mathbf{x}$, the iteration is skipped since there will be no contribution of the current iteration. In order to understand whether there is a zero in $\mathbf{x}$, the number of zeros are being kept from the previous iterations and updated every iteration as $\mathbf{x}$ is being updated. If the number of zeros is zero, then $\mathbf{x}$ is being iterated to contribute. This similar approach is also used in the Algorithm 5 and GPU implementations, in a slightly different manner. Instead of keeping only the number of zeros from the previous iterations, production of nonzero elements in the $\mathbf{x}$ from the previous iteration along with the number of zeros is being kept. Similarly, only nonzero elements of a selected column are iterated using `CCS`, and the production of nonzero elements and the number of zeros in $\mathbf{x}$ are both updated. In the end, if the number of zeros is zero, then previously kept production is being used to have a contribution and $\mathbf{x}$ is not going to be iterated again. `SortOrder` is also used as a preprocessing to make the more frequently processed columns are the ones with the least nonzero elements. Because in the gray code order, the first few bits frequently change while the last bits are not

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

1: $p \leftarrow 1$
2: **for** $i = 1 \ldots N$ **do**
3: $\quad rowSum \leftarrow 0$
4: $\quad$ **for** $j = 1 \ldots N$ **do**
5: $\quad\quad rowSum \leftarrow rowSum + mat[i,j]$
6: $\quad x[i] \leftarrow mat[i,n] - rowSum/2$
7: $\quad p \leftarrow p \times x[i]$
8: **for** *each thread* **do**
9: $\quad myX \leftarrow x$
10: $\quad myP \leftarrow 0$
11: $\quad myStart \leftarrow start + \texttt{threadId} \times chunkSize$
12: $\quad myEnd \leftarrow min(start + (\texttt{threadId} + 1) \times chunkSize, end)$
13: $\quad$ calculate myX using $\texttt{GrayCode}_{myStart-1}$
14: $\quad prod \leftarrow \prod_{n=1}^{\texttt{dim}} myX[i]$ for $myX[i] \neq 0$
15: $\quad zeroNum \leftarrow \sum_{n=1}^{length(myX)} myX[i]$ for $myX[i] = 0$
16: $\quad$ **for** $g = myStart \ldots myEnd - 1$ **do**
17: $\quad\quad j \leftarrow \log_2(\texttt{GrayCode}_g \oplus \texttt{GrayCode}_{g-1}) + 1$
18: $\quad\quad s \leftarrow 2 * \texttt{GrayCode}_g[j] - 1$
19: $\quad\quad$ **for** $i = cptrs[j] \ldots cptrs[j]$ **do**
20: $\quad\quad\quad$ **if** $myX[rows[i]] == 0$ **then**
21: $\quad\quad\quad\quad zeroNum \leftarrow zeroNum - 1$
22: $\quad\quad\quad\quad myX[rows[i]] \leftarrow myX[rows[i]] + s \times cvals[i]$
23: $\quad\quad\quad\quad prod \leftarrow prod \times myX[rows[i]]$
24: $\quad\quad\quad$ **else**
25: $\quad\quad\quad\quad prod \leftarrow prod / myX[rows[i]]$
26: $\quad\quad\quad\quad myX[rows[i]] \leftarrow myX[rows[i]] + s \times cvals[i]$
27: $\quad\quad\quad\quad$ **if** $myX[rows[i]] == 0$ **then**
28: $\quad\quad\quad\quad\quad zeroNum \leftarrow zeroNum + 1$
29: $\quad\quad\quad\quad$ **else**
30: $\quad\quad\quad\quad\quad prod \leftarrow prod \times myX[rows[i]]$
31: $\quad\quad$ **if** $zeroNum == 0$ **then**
32: $\quad\quad\quad myP \leftarrow myP + (-1)^g * prod$
33: $\quad AtomicAdd(p, myP)$
34: **return** $p \times (4 \times (n \bmod 2) - 2)$

changing often, that leads to first columns of the matrix being used frequently. Therefore, after `SortOrder`, the first columns will have the least nonzero elements. For parallelism, the for loop which has 2^{n-1} iterations is parallelized in a data-parallel fashion. For each algorithm in section 3.1, this approach is applied just by using `CCS` instead of the matrix data structure, and changing the iteration where $\mathbf{x}$ is updated as described in this chapter.

The same approaches in chapter 3.1 for dense matrices are followed in this chapter. However, instead of using a matrix and copying it from host memory to global or shared memory of the device, `CCS` is used, and three arrays are copied into device memory. Three arrays for, `CCS` as it was indicated in Figure 3.4, consume $2 \times \texttt{nnz} + \texttt{dim} + 1$ many elements, which are usually less than the matrix itself when the matrix is half full at most. It is also important to note that `SortOrder` is used as a preprocessing to make the calculation process faster.

3.2.1 `SpaRyser-SxC-Sm`: keeping x and `CCS` in the shared memory

In `SpaRyser-SxC-Sm`, a very similar approach as in `Ryser-SxC-Sm` is followed. There is a memory allocated in the shared memory for both $\mathbf{x}$ and the matrix. However, `CCS` is used for the representation of the matrix. Therefore, instead of allocating a memory as the number of elements in the matrix, memory is allocated only for *cptrs*, *rows*, and *cvals* which has the size of $2 \times \texttt{nnz} + \texttt{dim} + 1$. The device kernel of `SpaRyser-SxC-Sm` is given in Algorithm 6. Between lines 4-10, the shared memory locations are initialized for $\mathbf{x}$ and `CCS` with the values from the global device memory. It is also important to note that `SpaRyser-SxC-Sm` makes use of memory coalescing for $\mathbf{x}$ where the same indices of each $\mathbf{x}$ of each thread are next to each other in the shared memory. As opposed to `Ryser-SxC-Sm`, only the nonzero elements in the current column to be processed are iterated, as in lines between 22-33. While processing, the column, $zeroNum$ which is the number of zeros, and *prod* which is the production of nonzero elements in myX are updated. If there are no zeros in myX at the end, the current iteration contributes to the permanent using *prod* as in line 35. In the end, each thread writes their results back to the global memory location in p using their global thread ID `tid`, which then to be summed up to be used in calculating the final permanent result in the host. It is important to note that since less memory is used than matrix itself as opposed to `Ryser-SxC-Sm`, assuming matrix is half full at most, there will be more available space remained in the shared memory when the same `BlockDim` is used.

Algorithm 6 SpaRyser-SxC-Sm($cptrs, rows, cvals, dim, nnz, x, p, start, end$)

1: $\texttt{tid} \leftarrow threadIdx.x + (blockIdx.x \times blockDim.x)$
2: $\texttt{threadId} \leftarrow threadIdx.x$
3: $\texttt{BlockDim} \leftarrow blockDim.x$
4: **for** $k = 1 \ldots \texttt{dim}$ **do**
5: $\quad myX[\texttt{BlockDim} \times k + \texttt{threadId}] \leftarrow x[k]$
6: $\quad sharedCptrs[k] \leftarrow cptrs[k]$
7: $sharedCptrs[\texttt{dim}+1] \leftarrow cptrs[\texttt{dim}+1]$
8: **for** $k = 1 \ldots \texttt{nnz}$ **do**
9: $\quad sharedRows[k] \leftarrow rows[k]$
10: $\quad sharedCvals[k] \leftarrow cvals[k]$
11: $syncthreads()$
12: $chunkSize \leftarrow (end - start)/(\texttt{GridDim} \times \texttt{BlockDim}) + 1$
13: $myStart \leftarrow start + \texttt{tid} \times chunkSize$
14: $myEnd \leftarrow min(start + (\texttt{tid}+1) \times chunkSize, end)$
15: $myP \leftarrow 0$
16: calculate myX using $\texttt{GrayCode}_{myStart-1}$
17: $prod \leftarrow \prod_{n=1}^{\texttt{dim}} myX[i]$ for $myX[i] \neq 0$
18: $zeroNum \leftarrow \sum_{n=1}^{length(myX)} myX[i]$ for $myX[i] = 0$
19: **for** $g = myStart \ldots myEnd - 1$ **do**
20: $\quad j \leftarrow \log_2(\texttt{GrayCode}_g \oplus \texttt{GrayCode}_{g-1}) + 1$
21: $\quad s \leftarrow 2 * \texttt{GrayCode}_g[j] - 1$
22: $\quad$ **for** $i = cptrs[j] \ldots cptrs[j]$ **do**
23: $\quad\quad$ **if** $myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{threadId}] == 0$ **then**
24: $\quad\quad\quad zeroNum \leftarrow zeroNum - 1$
25: $\quad\quad\quad myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{threadId}] \leftarrow myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{threadId}] + s \times sharedCvals[i]$
26: $\quad\quad\quad prod \leftarrow prod \times myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{threadId}]$
27: $\quad\quad$ **else**
28: $\quad\quad\quad prod \leftarrow prod / myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{threadId}]$
29: $\quad\quad\quad myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{threadId}] \leftarrow myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{threadId}] + s \times sharedCvals[i]$
30: $\quad\quad\quad$ **if** $myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{threadId}] == 0$ **then**
31: $\quad\quad\quad\quad zeroNum \leftarrow zeroNum + 1$
32: $\quad\quad\quad$ **else**
33: $\quad\quad\quad\quad prod \leftarrow prod \times myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{threadId}]$
34: $\quad$ **if** $zeroNum == 0$ **then**
35: $\quad\quad myP \leftarrow p + (-1)^g * prod$
36: $p[\texttt{tid}] \leftarrow myP$

3.2.2 SpaRyser-SxC-Sm-MG+: dynamic hybrid implementation

In SpaRyser-SxC-Sm-MG+, the approach in Ryser-SxC-Sm-MG+ is followed. Iteration space is divided into small chunks and each device takes one chunk at a time to start the device kernel, which is SpaRyser-SxC-Sm. When a device finished its portion of work, a thread in the host contributes to the overall permanent using the result of the device, and starts the device kernel again if there is any chunk left within the total iteration space. Optionally, a CPU kernel can be started in the same manner, starting ParSpaRyser with the current chunk. In this way, multiple GPU devices and optionally a CPU work together collectively. However, there is still one question about how to set the chunk size. There could be extra overhead in initializing the devices each time when chunk size is small. Similarly, if chunk size is large, one device can finish its work and there is no chunk left while a slower device may continue working. The chunk size was chosen manually as $2^{\texttt{dim}-29}$ for Ryser-SxC-Sm-MG+. However, since SpaRyser-SxC-Sm-MG+ is a sparse implementation, chunks can be completed faster than the dense implementation. Therefore, chunk size is chosen larger as $2^{\texttt{dim}-30}$ in SpaRyser-SxC-Sm-MG+.

4. APPROXIMATING MATRIX PERMANENTS ON GPUs

Calculating the number of perfect matchings in a bipartite graph is equivalent to the permanent value of the bipartite adjacency matrix that corresponds to the bipartite graph. However, values of the corresponding adjacency matrix indicates whether an edge exists or not at each index, such that this adjacency matrix is composed of binary values where 1 shows there is an edge between vertices. Therefore, one can estimate the permanent value of a binary matrix using approximate value of the number of perfect matchings for the corresponding bipartite graph. `Rasmussen` and `Scaling` algorithms are examples which estimates number of perfect matching of a graph.

It is very important to have as many experiments as possible to obtain more accurate permanent value estimations in the approximation algorithms. As there are more experiments, there will be more time to wait for the final result. Since each experiment is independent of each other, one can parallelize the experiments that are being run. In this paper, both `Rasmussen` and `Scaling` algorithms are used along with their improved version `Rasmussen+` and `Scaling+` for the approximation algorithms implemented on GPU.

4.1 Approximating the Permanents of Dense Matrices

Experiments for the approximation algorithms can be parallelized as the number of available threads on CPU. Each thread can calculate the value of the current experiment and add the final value atomically to a global variable. After obtaining the sum of each experiment, it can be divided by the number of experiments to get the mean of experiments, which is the estimation for the permanent value in a binary matrix. The parallel versions of `Rasmussen`, `Rasmussen+`, `Scaling`, and `Scaling+` will be denoted as `ParRasmussen`, `ParRasmussen+`, `ParScaling`, and `ParScaling+`

respectively.

4.1.1 `RasmussenGpu`: Implementation of `Rasmussen` on GPU

Similar approach on a CPU is followed on a single GPU. In Algorithm 7, the device kernel of `Rasmussen+Gpu` can be seen, which is slightly different than the kernel of `RasmussenGpu`. The inputs for the device kernel are the matrix *mat*, the dimension of the matrix *dim*, a random number *rand* which then to be multiplied by global thread ID `tid` on GPU to be used as a seed for random number generation, and the result list p for the output of each experiment which then will be copied back to host memory to take their average. Memory is allocated in the shared memory as the size of the matrix to copy the matrix itself from global device memory to shared memory of each block, as it can be seen in line 5 of the Algorithm 7. As mentioned before, shared memory limitation is $48KB$. Moving the matrix to the shared memory means that there will be $\texttt{dim}^2 \times 4$ bytes of memory is allocated when the type of the matrix is integer. In order to conform to shared memory restrictions, the maximum dimension of the matrix will be approximately 110. There will be no limitation while setting the `BlockDim` since shared memory being used does not depend on the number of threads in a block. In order to keep the track of the extracted columns in each step, an array of integers is being kept in the registers. However, due to memory limitations on registers, the size of this array is also limited. For this reason, bitwise operations have been used to understand whether a column is extracted or not. In each index of the array, there is an integer element with 32 bits, which is enough to keep track of extracted columns of a matrix with a dimension of 32. As matrix dimension gets bigger, other indices' bits will be checked and updated for the next columns, as it is described in the Figure 4.1 for rows. So, it is enough to create an array of size equals to $\lceil \texttt{dim}/32 \rceil$ in overall to keep the track of extracted columns. For the dense algorithm `RasmussenGpu`, the memory limitations on registers will not be exceeded due to shared memory limitation will be reached first. When the matrix dimension is 110, the size of the $colExtracted$ will be 4 which consumes four registers. In `RasmussenGpu`, there is no need of creating an extra variable to keep track of the number of rows since each row will be utilized in order. Therefore, in `RasmussenGpu`, instead of finding the row with the least nonzero in line 11 as in Algorithm 7, it takes the row which is equal to the iteration ID k. Then, using the nonzero number in row where column of a nonzero was not extracted, $perm$ value is multiplied by it. After randomly choosing a nonzero in the row, this nonzero element's column col will be extracted by flipping the necessary bit in $colExtracted$

as in line 16. The $\vee$ in line 16 is **bitwise or** operator. After all the iterations are completed by each thread, results are written back to output list p to the index of global thread ID `tid` of each thread. At the end when kernel is completed, the output list p is copied back to host memory to calculate the mean of the results as the final estimation value.

Algorithm 7 `Rasmussen+Gpu` $(mat, dim, rand, p)$

1: `tid` $\leftarrow$ global thread id
2: `threadId` $\leftarrow$ thread id within block
3: **for** $k = 1 \ldots (\texttt{dim} \times \texttt{dim})/\texttt{BlockDim} + 1$ **do**
4: **if** $\texttt{BlockDim} \times k + \texttt{threadId} < \texttt{dim} \times \texttt{dim}$ **then**
5: $sharedMat[\texttt{BlockDim} \times k + \texttt{threadId}] \leftarrow mat[\texttt{BlockDim} \times k + \texttt{threadId}]$
6: synch threads inside a block
7: $perm \leftarrow 1$
8: $rowExtracted[\lceil \texttt{dim}/32 \rceil]$
9: $colExtracted[\lceil \texttt{dim}/32 \rceil]$
10: **for** $k = 1 \ldots \texttt{dim}$ **do**
11: $row \leftarrow$ row with the least number of nonzero
12: `nnz` $\leftarrow$ nonzero number of row
13: $perm \leftarrow perm \times \texttt{nnz}$
14: $col \leftarrow$ column of the randomly chosen nonzero in row
15: $rowExtracted[\lceil row/32 \rceil] \leftarrow rowExtracted[\lceil col/32 \rceil] \ \vee \ (1 << (row \ \% \ 32))$
16: $colExtracted[\lceil col/32 \rceil] \leftarrow colExtracted[\lceil col/32 \rceil] \ \vee \ (1 << (col \ \% \ 32))$
17: $p[\texttt{tid}] \leftarrow perm$

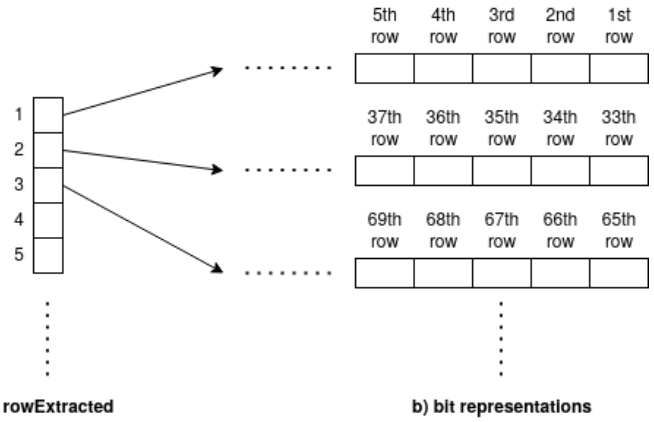


Figure 4.1 (a) array to keep track of extracted rows with (b) binary representation of each index

4.1.2 `Rasmussen+Gpu`: Implementation of `Rasmussen+` on GPU

Similar to `RasmussenGpu`, this algorithm follows everything almost the same on GPU except how to choose the row at each step. As for `Rasmussen+` algorithm, rows were chosen according to their increasing order of nonzero elements. So, this approach is followed in `Rasmussen+Gpu`, whose device kernel can be seen in Algorithm 7. The same shared memory restrictions exist in this algorithm as in `RasmussenGpu` since the every element of the matrix is copied to the shared memory. Additionally, in `Rasmussen+Gpu`, *rowExtracted* array is created along with the *colExtracted*. Because, since rows are not chosen in their initial order at each step, one should be aware of which rows are extracted while choosing the row. On line 11, it iterates through each unextracted row to find the row with the least nonzero elements, where each nonzero element's column has not been extracted yet. Furthermore, the extracted row in the current step is taken into account on line 15, whose logic behind was illustrated in Figure 4.1. It is important to note that since there are two lists to keep the track of extracted rows and columns separately, there will be more registers taken by each thread. However, this still does not have any effect since shared memory limitation comes before this, where one can only store matrix with a dimension up to 110 since `Rasmussen+Gpu` is a dense algorithm, and it copies all the elements of the matrix to shared memory.

4.1.3 `Rasmussen+MGpu`: Hybrid implementation of `Rasmussen+`

In `Rasmussen+Gpu`, the number of perfect matching, which means permanent value for binary matrices, are estimated accurately and efficiently. As the number of experiments are increased, one can obtain even more accurate, permanent results. However, this would increase the runtime for the estimation. In order to decrease the runtime further, hybrid solutions are possible since every experiment is independent of each other. In `Rasmussen+MGpu`, multiple threads in the host start device kernels on different GPU devices. Along with the GPU devices, parallel version of `Rasmussen+` on a CPU, `ParRasmussen+`, is also able to be started in parallel. The device kernel that is called for each GPU device is shown in Algorithm 7, which is `Rasmussen+Gpu`. Here, instead of dividing experiments equally and distribute workload to devices equally, a chunking mechanism has been used which makes each device start with some number of experiments as one chunk's size. After a device is done with its work, it checks whether the total number of experiments aimed to run are reached or not. If there needs to be more experiments to be run, the device starts running again with the next chunk until the number of experiments in total satisfies. In the end, all the results' average are taken as the final permanent value

estimation. It is important to choose the chunk size right. Since it makes use of an approximation algorithm, it does not take long, as it was the case in computing exact permanent values. Furthermore, each kernel call costs some more time, which may cause a hybrid solution takes more time than a single GPU approach. For this reason, each kernel call is regulated to take approximately one million experiments by setting both `GridDim` and `BlockDim` to 1024 since each thread calculates the result of a single experiment. While GPU devices calculate around one million experiments every time they start, `ParRasmussen+` runs only for $50,000$ experiments since this algorithm runs slower than a single GPU device.

4.1.4 `ScalingGpu`: Implementation of `Scaling` on GPU

For `ScalingGpu`, the algorithm makes use of the logic behind `Scaling` algorithm. The input parameters for this algorithm is the same as the `Scaling+Gpu` shown in Algorithm 8, which are matrix *mat*, dimension of the matrix *dim*, random number *rand* that is to be multiplied with global thread ID `tid` to create a seed for random number generation, result list p, two arrays R and C to store the entries of diagonal scaling matrices, `SInterval` and `STime` that are used to indicate scaling intervals and number of time to scale respectively. Firstly, matrix itself copied to the shared memory. The R and C arrays are kept in the global memory due to shared memory limitations. Because, each thread will be maintaining different arrays since each one may be choosing different row and column to extract at the same iteration due to randomness that would lead their R and C differ. If one tries to put each of them to shared memory, then $2 \times \texttt{dim} \times \texttt{BlockDim} \times 4$ bytes of memory should be allocated in a single block, assuming these arrays are composed of integer elements. This is not a good approach since shared memory limitation is $48KB$ and this limits the `BlockDim` to a maximum of 153 threads for a simple matrix with 40 as a dimension. As the dimension gets bigger, this even limits further. That's why R and C are allocated in the global memory before the device kernel call, and each array's size is $\texttt{dim} \times \texttt{BlockDim} \times \texttt{GridDim}$, such that each thread are able to access its own portion of arrays using its global thread ID `tid`. For instance, a thread should access to index of $\texttt{tid} \times \texttt{dim} + i$ if it is going to access to ith index of R or C. Before starting iterating rows, R and C are updated by each thread to its initial value, which is 1. Also, *colExtracted* array is created on registers to keep track of extracted columns, as it can be seen in line 12 of Algorithm 8. The only difference in `ScalingGpu` than Algorithm 8, there is no need to keep the track of extracted rows since it chooses rows according to their initial order instead of choosing the one

with the least number of nonzero elements. Afterwards, scaling algorithm is applied to normalize the matrix before choosing a nonzero within a row in order to increase the possibility of choosing the nonzero that contributes a perfect matching. Scale algorithm is called every `SInterval` steps and it updates the diagonal entries of scaling matrices and stores them in R and C every time it scales. Scaling algorithm can be seen in Algorithm 9 in detail. At each iteration of scaling, firstly the columns are balanced, and then the rows are balanced. While balancing the columns and rows, if one encounters a column or row without a nonzero within the unextracted elements, then scaling algorithm returns. Because there will be no need for scaling in such a case, since estimation will return 0 for the current experiment. After scaling, $R[i] * sharedMat[i \times \texttt{dim} + j] * C[j]$ is used in order to access the scaled entry at the ith row and jth column of the matrix. Then, one unextracted nonzero is chosen in the current row with the specified probability p_j for each nonzero in line 17, and the permanent value $perm$ is updated using the previously calculated p_j where j is randomly chosen column col. At final, $colExtracted$ is also updated to keep track of the column being extracted.

Algorithm 8 `Scaling+Gpu` $(mat, dim, rand, p, R, C, \texttt{SInterval}, \texttt{STime})$

1: `tid` $\leftarrow$ global thread id
2: `threadId` $\leftarrow$ thread id within block
3: **for** $k = 1 \ldots (\texttt{dim} \times \texttt{dim})/\texttt{BlockDim} + 1$ **do**
4: **if** $\texttt{BlockDim} \times k + \texttt{threadId} < \texttt{dim} \times \texttt{dim}$ **then**
5: $sharedMat[\texttt{BlockDim} \times k + \texttt{threadId}] \leftarrow \texttt{mat}[\texttt{BlockDim} \times k + \texttt{threadId}]$
6: synch threads inside a block
7: **for** $k = 1 \ldots \texttt{dim}$ **do**
8: $R[\texttt{tid} \times \texttt{dim} + k] \leftarrow 1$
9: $C[\texttt{tid} \times \texttt{dim} + k] \leftarrow 1$
10: $perm \leftarrow 1$
11: $rowExtracted[\lceil \texttt{dim}/32 \rceil]$
12: $colExtracted[\lceil \texttt{dim}/32 \rceil]$
13: **for** $k = 1 \ldots \texttt{dim}$ **do**
14: $row \leftarrow$ row with the least number of nonzero
15: **if** $(k-1)$ % $\texttt{SInterval} = 0$ **then**
16: SCALE($sharedMat, \texttt{dim}, rowExtracted, colExtracted, R, C, \texttt{STime}$)
17: $col \leftarrow$ column j of the randomly chosen element in row, where each column j have the probability $p_j = \frac{R[row] \times sharedMat[row \times \texttt{dim} + j] \times C[j]}{\sum_{k \in S} R[row] \times sharedMat[row \times \texttt{dim} + k] \times C[k]}$, where S is set of unextracted columns.
18: $perm = perm/p_j$
19: $rowExtracted[\lceil row/32 \rceil] \leftarrow rowExtracted[\lceil col/32 \rceil] \ \vee \ (1 << (row \ \% \ 32))$
20: $colExtracted[\lceil col/32 \rceil] \leftarrow colExtracted[\lceil col/32 \rceil] \ \vee \ (1 << (col \ \% \ 32))$
21: $p[tid] \leftarrow perm$

Algorithm 9 SCALE($sharedMat$,$rowExtracted$,$colExtracted$, $R, C,$ STime)

```
1: for k = 1...STime do
2:     for j = 1...dim do
3:         if j^th column is not extracted then
4:             colSum ← 0
5:             for i = 1...dim do
6:                 if i^th row is not extracted then
7:                     colSum ← colSum + R[tid × dim + i] × sharedMat[i × dim + j]
8:             if colSum = 0 then
9:                 return
10:            C[tid × dim + j] ← 1/colSum
11:    for i = 1...dim do
12:        if i^th row is not extracted then
13:            rowSum ← 0
14:            for j = 1...dim do
15:                if j^th column is not extracted then
16:                    rowSum ← rowSum + C[tid × dim + j] × sharedMat[i × dim +
   j]
17:            if rowSum = 0 then
18:                return
19:            R[tid × dim + i] ← 1/rowSum
```

4.1.5 `Scaling+Gpu`: Implementation of `Scaling+` on GPU

This algorithm is mostly the same as `ScalingGpu` except it chooses the rows according to their increasing number of nonzero elements at each step, as it was the case in `Scaling+`. Device kernel of `Scaling+Gpu` is given in Algorithm 6. As it can be seen, *rowExtracted* is also created to keep the track of the extracted rows. The remaining algorithm is the same as `ScalingGpu`. It is also important to note that in this version, there will be more registers will be used. However, this does not cause a problem since shared memory limitation avoids matrices larger than 110 as `dim`.

4.1.6 `Scaling+MGpu`: Hybrid implementation of `Scaling+`

Similar to `Rasmussen+MGpu`, this algorithm is also aimed to run on multiple GPU devices and optionally on CPU to better estimate permanent value by running more experiment in a less amount of time. Each thread in the host is making use of the device kernel of `Scaling+Gpu` for each device, whereas `ParScaling+` is used for a CPU which is the parallel version of `Scaling+` algorithm on a CPU.

In `Scaling+MGpu`, again chunking mechanism is used where each device kernel is called with `GridDim` and `BlockDim` equal to 1024. Hence, in the experiments of this thesis, each kernel call is executed around one million times. On the other hand, `ParScaling+` is called $50,000$ times each time it is called since it is working much slower than `Scaling+Gpu`. The algorithm terminates when the expected number of experiments is reached, and a final estimation permanent value is calculated by taking the average of all the experiments.

4.2 Approximating the Permanents of Sparse Matrices

For the sparse implementations of the approximation algorithms for sparse matrices, the only difference in each algorithm is the data structure being used to represent the matrix. As a data structure, `CCS` and `CRS` are used as it was previously described and shown in Figure 3.4. Sparse implementations of `ParRasmussen`, `ParRasmussen+`, `ParScaling`, and `ParScaling+` utilizing `CRS` and `CCS` will be denoted as `ParRasmussenS`, `ParRasmussen+S`, `ParScalingS`, and `ParScaling+S` respectively.

4.2.1 `RasmussenGpuS`: Sparse implementation of `Rasmussen` on GPU

In `RasmussenGpuS`, the implementation is the same as, `RasmussenGpu` except it makes use of `CRS` instead of the matrix itself. In `RasmussenGpuS`, `CRS` will be used while finding nnz and, col as in line 12 and line 14 in Algorithm 7. Furthermore, since this algorithm is an approximation algorithm to estimate number of perfect matchings, which corresponds to permanent value of a binary 0-1 matrix, $rvals$ of `CRS` is not used. The only used arrays of `CRS` are $rptrs$, and $cols$. Because, this data structure only stores the nonzero elements and all of the nonzero elements are known to be 1. Therefore, only $rptrs$ and $cols$ are copied to the shared memory, which leads to allocation of $(\texttt{nnz}+\texttt{dim}+1)\times 4$ bytes of memory in the shared memory since $rptrs$ and $cols$ are a type of integer. That means, if the number of the nonzero elements are few in a matrix, one can estimate the permanent value of matrices with huge dimensions without getting restricted by the shared memory limitations. For instance, if memory consumption of $rptrs$ was ignored, which is $(\texttt{dim}+1)\times 4$

bytes, there could be 12288 nonzero elements inside of an input matrix. Since the shared memory limitation is not an issue when there are few nonzero elements in the matrix, the larger issue could be limitation on registers. For each block, there are 65536 registers are available. In `RasmussenGpuS`, `BlockDim` is set to 1024, and `GridDim` is set to number of experiments divided by the `BlockDim`. Since there are 1024 threads in a block, each thread is able to make use of 64 registers. Therefore, since only *colExtracted* is used in `RasmussenGpuS`, the size of this array could go up to around 40-45 to consume that many registers among 64. Because there are also locally created variables exist in the device kernel. If one takes the size of the *colExtracted* as 40, then extracted columns of a matrix with a dimension of $32 \times 40 = 1280$ can be tracked if there are few nonzero elements in the matrix and the shared memory limitation is not reached.

4.2.2 `Rasmussen+GpuS`: Sparse implementation of `Rasmussen+` on GPU

In `Rasmussen+GpuS`, the exact implementation of `Rasmussen+Gpu` is followed only changing the data structure to use in storing matrix which are `CRS` and `CCS`. Similar to `RasmussenGpuS`, the only used arrays are *rptrs* and *cols* of `CRS`. The only difference is additionally *rowExtracted* array is used to keep the track of unextracted rows, as in line 11 of Algorithm 7. Therefore, sizes of *colExtracted* and *rowExtracted* are less than the size of *colExtracted* in, `RasmussenGpuS` since they consume more registers. So, sizes of *rowExtracted* and *colExtracted* can take values up to, 20-25 since the rest of the registers are consumed by the locally created variable in the device kernel. That also means that a matrix with a dimension of 32×20 will be able to be processed when sizes of *rowExtracted* and *colExtracted* are 20, and the shared memory limitations are not exceeded.

4.2.3 `Rasmussen+MGpuS`: Hybrid sparse implementation of `Rasmussen+`

In `Rasmussen+MGpuS`, each thread in the host are able to call the device kernel of `Rasmussen+GpuS` simultaneously for separate GPU devices. Optionally, an additional thread can start another permanent estimation algorithm that is `ParRasmussen+S`. Here, the chunk sizes are chosen as the same as `Rasmussen+MGpu` where device kernels are started by setting both `BlockDim` and `GridDim` to 1024

which runs as many experiments as the number of threads, which is `BlockDim` × `GridDim`. On the other hand, if additionally `ParRasmussen+S` is called, the number of experiments to start with is set to $50,000$ since it runs slower than GPU. The algorithm continue to work until expected number of experiments is reached.

4.2.4 `ScalingGpuS`: Sparse implementation of `Scaling` on GPU

In `ScalingGpuS`, the approach in `ScalingGpu` is followed, where the only difference is that it makes use of `CRS` and `CCS` instead of the matrix. Here, both `CRS` and `CCS` data structures are needed. Because in the scaling process given in Algorithm 9, which is called by the device kernel of `ScalingGpuS` in some of the iterations, first columns are balanced, and then the rows are balanced where `CCS` and `CRS` are used respectively. Furthermore, `CRS` is used in line 17 of Algorithm 8 while choosing the *col* randomly with a probability. For both of `CRS` and `CCS`, there is no need to store *rvals* and *cvals* for a binary matrix. Since a memory region in the shared memory is created for *rptrs*, *cols*, *cptrs*, and *rows*, $2 \times (\texttt{nnz} + \texttt{dim} + 1) \times 4$ bytes of memory is allocated in the shared memory. For instance, if memory consumption of *rptrs* and *cptrs* were ignored, which is $2 \times (\texttt{dim} + 1) \times 4$ bytes, there could be 6144 nonzero elements inside of an input matrix. Therefore, in terms of the shared limitation, `ScalingGpuS` is not limited if there are few nonzero elements in the matrix regardless of its dimension. Also, since this algorithm is the sparse version of `ScalingGpu`, only columns should be tracked whether they are extracted or not since rows are processed in order. In terms of register limitations, the maximum length of $colExtracted$ could be 40-45 since there are also locally created variables in the device kernel which consumes some registers and the maximum register available for each thread is 64. Therefore, similar to `RasmussenGpuS`, it can be said that if the size of $colExtarcted$ is chosen as 40, one can process a matrix with a dimension of $32 \times 40 = 1280$ while the shared memory limitation is not reached.

4.2.5 `Scaling+GpuS`: Sparse implementation of `Scaling+` on GPU

In `Scaling+GpuS`, the approach in `Scaling+` is followed by making use of `CRS` and `CCS` instead of the matrix itself. `Scaling+GpuS` is also very similar to, `ScalingGpuS` except it chooses rows according to the one with the least nonzero elements at

each iteration. Therefore, `CRS` is utilized in line 14 of Algorithm 8 in addition to `ScalingGpuS`. Furthermore, the same shared memory limitation is valid here as in `ScalingGpuS`. On the other hand, in terms of register limitation, since there are two arrays as $rowExtracted$ and $colExtracted$ as in `Scaling+Gpu`, the sizes of the arrays could take lower values than the size of $colExtracted$ in `ScalingGpuS`. Therefore, each can take up to 20-25 as the size since there are also locally created variables that consumes registers. That means, if sizes of $rowExtracted$ and $colExtracted$ are 20, a matrix with a dimension of $32 \times 20 = 640$ will be able to be processed when the shared memory limitations are not exceeded.

4.2.6 `Scaling+MGpuS`: Hybrid sparse implementation of `Scaling+`

In `Scaling+MGpuS`, each thread in the host are able to call the device kernel of `Scaling+GpuS` at the same time for different GPU devices. Optionally, an extra thread starts `ParScaling+S` to estimate the permanent and contribute to the permanent estimation process further. Similar to `Scaling+MGpu`, `BlockDim` and `GridDim` are set to 1024 to obtain 1024×1024 experiments in each device kernel call, whereas `ParScaling+S` is called for $50,000$ experiments. When one of the kernels or `ParScaling+S` completes its job, then they are started again with their chunk as the number of experiments if total expected number of experiments is not reached yet. At the end, all the permanent estimations' average is taken to be the final permanent estimation.

5. COUNTING PERFECT MATCHINGS ON GPUs

`SkipPer` algorithm can be parallelized in data parallel fashion by dividing iteration space for each thread. Implementation of `ParSkipPer` is given in Algorithm 10. The input parameters for this algorithm is matrix *mat* that is to be used only initializing **x**, arrays for `CRS` and `CCS` to exploit sparsity using sparse data structure, dimension of the matrix *dim*, and start and end point of the iteration space. In the first for loop on line 2, **x** is set to its initial values. Then, each thread calculates its own portion using *myStart* and *myEnd* by utilizing dynamic scheduling, where these portions are dynamically given to each thread as small chunks. In `SkipPer`, some of the iterations are explicitly passed with a single jump if they do not contribute to the permanent due to **x** having zero at some index. In order to keep track of the last `GrayCode` after the jump, *prevG* is created on line 13 which is initially zero. At each iteration, since there is a possibility of a jump, one should look all the columns in the difference between the current $\texttt{GrayCode}_g$ and previous $\texttt{GrayCode}_{prevG}$. Then, this difference is used in updating *myX* which is *xvector* of each thread. At the end, after contributing to the permanent, if there is no contribution, next g is calculated with a single jump using $next(g)$

$$next(g) = \begin{cases} g+1, & \text{if } myX[i] \neq 0,\ for\ 1 \leq i \leq \texttt{dim} \\ max(g^i : myX[i] = 0), & \text{otherwise} \end{cases} \tag{5.1}$$

where g^i is the first iteration after g that processes a column having nonzero in the ith row, such that $myX[i]$ will be set something different than zero in this iteration.

For the implementations on GPU, the same approach in `ParSkipPer` is followed, where the only difference is related to memory accesses to the matrix and to **x** at each iteration.

Algorithm 10 `ParSkipPer` $(mat, rptrs, cols, cptrs, rows, cvals, dim, start, end)$

```
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:     prevG ← 0
14:     g ← myStart
15:     while g < myEnd do
16:         grayDiff ← GrayCode_g ⊕ GrayCode_prevG
17:         for each grayDiff[j] = 1 do
18:             if GrayCode_g[j] = 1 then
19:                 for i = cptrs[j]...cptrs[j] do
20:                     myX[rows[i]] ← myX[rows[i]] + cvals[rows[i]]
21:             else
22:                 for i = cptrs[j]...cptrs[j] do
23:                     myX[rows[i]] ← myX[rows[i]] − cvals[rows[i]]
24:         prod ← 1
25:         for i...dim do
26:             prod ← prod × myX[i]
27:         myP ← myP + (−1)^g × prod
28:         prevG ← g
29:         g ← next(g)
30:     AtomicAdd(p, myP)
31: return p × (4 × (n mod 2) − 2)
```

5.1 SkipPer-SxC-Sm: keeping x, CRS, and CCS in the shared memory

In SkipPer-SxC-Sm, approach in SkipPer is followed on GPU. The device kernel of SkipPer-SxC-Sm is given in Algorithm 11. The input parameters are CCS, CRS, dimension of the matrix dim, number of nonzero elements nnz, $\mathbf{x}$, result list p to store the result of each thread, and start and end points of the iteration space. Shared memory is allocated for the $\mathbf{x}$, CCS, and CRS to utilize faster accesses to these elements. There is also applied memory coalescing for $\mathbf{x}$, where the same indices of $\mathbf{x}$ of each thread are next to each other in memory and one thread can reach the ith index of $\mathbf{x}$ using $myX[\texttt{BlockDim} \times i + \texttt{threadId}]$. For CCS, 3 arrays $sharedCptrs$, $sharedRows$, $sharedCvals$ are created in shared memory set to their initial values, whereas for CRS, only 2 arrays $sharedRptrs$, $sharedCols$ are created since $rvals$ of CRS is not utilized anywhere in the device kernel. After the shared memory locations are set, threads are synced before moving on to the iteration space. Afterwards, the logic is the same as ParSkipPer. Each thread determines their start and end points that are $myStart$, and, $myEnd$ respectively. Since there is a jump in the iterations to skip iterations that does not contribute to the permanent, previous GrayCode is kept using, $prevG$ which is initially zero as in line 19. In the current iteration, each thread makes use of the difference in GrayCode which is $\texttt{GrayCode}_g \oplus \texttt{GrayCode}_{prevG}$ to find out columns to process to update the $\mathbf{x}$. Then, if there is a zero in $\mathbf{x}$, the current iteration will not contribute to the permanent and one can skip iterations using $next(g)$ as described previously. While applying $next(g)$, $sharedRptrs$ and $sharedCols$ are also utilized. At the end, each thread writes their results back to p using their global thread ID tid which then copied back to host and summed up while calculating overall permanent result.

In overall, total shared memory allocated is $\texttt{dim} \times \texttt{BlockDim} \times 4$ bytes for $\mathbf{x}$, and $((2 \times \texttt{nnz} + \texttt{dim} + 1) + (\texttt{nnz} + \texttt{dim} + 1)) \times 4$ bytes for CCS and CRS with data type of integer or float. Therefore, in order to satisfy $48KB$ shared memory limitation, the following should hold when the data type is integer or float.

(5.2)
$$(\texttt{dim} \times \texttt{BlockDim} \times 4\ bytes) + ((3 \times \texttt{nnz} + 2 \times \texttt{dim} + 2) \times 4\ bytes) \leq 48 \times 1024\ bytes$$

That means for an integer or float matrices with dimension of 40, with density values of $0.20, 0.30, 0.40$, the maximum BlockDim can take values 281, 269, 257 respectively.

It is also important to note that in SkipPer-SxC-Sm, it is not possible to apply dynamic scheduling for thread where each thread starts with a small portion of the iteration space and takes new ones as they are done with the old portion in a single GPU device, as dynamic scheduling was the case in ParSkipPer.

5.2 SkipPer-SxC-Sm-MG+: dynamic hybrid implementation

In SkipPer-SxC-Sm-MG+, device kernel of SkipPer-SxC-Sm is utilized by multiple GPU devices at the same time, similar to SpaRyser-SxC-Sm-MG+. Iteration space is divided into small chunks, where each thread can take one chunk at a time and starts a device kernel of SkipPer-SxC-Sm on a single GPU device. When the device kernel is completed and results are copied back to host memory, the responsible thread in the host can use the results to contribute to the overall permanent result, and then starts a device kernel again if there is any chunk left. Optionally, one extra thread can start permanent calculation on a CPU using ParSkipPer along with the GPU devices to finish the chunks and make the overall process faster. Therefore, in SkipPer-SxC-Sm-MG+, multiple GPU devices and optionally a CPU are able to work together collectively, and a synonym of dynamic scheduling is actually applied since each device takes small chunks at a time until all of them finishes. Similar to SpaRyser-SxC-Sm-MG+, one should pick the size of the chunks properly in SkipPer-SxC-Sm-MG+ since large chunk size can lead to longer running time when slower device takes the last chunk, whereas small chunk size can lead to many chunks where initializing each device kernel comes with a cost of memory allocation and copies. Therefore, chunk size is chosen as the same chunk size as in, SpaRyser-SxC-Sm-MG+ which is $2^{\texttt{dim}-30}$ since they are both algorithms for sparse matrices.

Algorithm 11 `SkipPer-SxC-Sm` $(cptrs, rows, cvals, rptrs, cols, dim, nnz, x, p, start, end)$

1: $\texttt{tid} \leftarrow threadIdx.x + (blockIdx.x \times blockDim.x)$
2: $\texttt{threadId} \leftarrow threadIdx.x$
3: $\texttt{BlockDim} \leftarrow blockDim.x$
4: **for** $k = 1 \ldots dim$ **do**
5: $\quad myX[\texttt{BlockDim} \times k + \texttt{threadId}] \leftarrow x[k]$
6: $\quad sharedRptrs[k] \leftarrow rptrs[k]$
7: $\quad sharedCptrs[k] \leftarrow cptrs[k]$
8: $sharedRptrs[\texttt{dim}+1] \leftarrow rptrs[\texttt{dim}+1]$
9: $sharedCptrs[\texttt{dim}+1] \leftarrow cptrs[\texttt{dim}+1]$
10: **for** $k = 1 \ldots nnz$ **do**
11: $\quad sharedCols[k] \leftarrow cols[k]$
12: $\quad sharedRows[k] \leftarrow rows[k]$
13: $\quad sharedCvals[k] \leftarrow cvals[k]$
14: $syncthreads()$
15: $chunkSize \leftarrow (end - start)/(\texttt{GridDim} \times \texttt{BlockDim}) + 1$
16: $myStart \leftarrow start + \texttt{tid} \times chunkSize$
17: $myEnd \leftarrow min(start + (\texttt{tid}+1) \times chunkSize, end)$
18: $myP \leftarrow 0$
19: $prevG \leftarrow 0$
20: $g \leftarrow myStart$
21: **while** $g < myEnd$ **do**
22: $\quad grayDiff \leftarrow \texttt{GrayCode}_g \oplus \texttt{GrayCode}_{prevG}$
23: $\quad$ **for** *each* $grayDiff[j] = 1$ **do**
24: $\quad\quad$ **if** $\texttt{GrayCode}_g[j] = 1$ **then**
25: $\quad\quad\quad$ **for** $i = sharedCptrs[j] \ldots sharedCptrs[j]$ **do**
26: $\quad\quad\quad\quad myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{threadId}] \leftarrow myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{threadId}] + sharedCvals[rows[i]]$
27: $\quad\quad$ **else**
28: $\quad\quad\quad$ **for** $i = sharedCptrs[j] \ldots sharedCptrs[j]$ **do**
29: $\quad\quad\quad\quad myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{threadId}] \leftarrow myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{threadId}] - sharedCvals[rows[i]]$
30: $\quad prod \leftarrow 1$
31: $\quad$ **for** $i \ldots \texttt{dim}$ **do**
32: $\quad\quad prod \leftarrow prod \times myX[\texttt{BlockDim} \times i + \texttt{threadId}]$
33: $\quad myP \leftarrow myP + (-1)^g \times prod$
34: $\quad prevG \leftarrow g$
35: $\quad g \leftarrow next(g)$
36: $p[\texttt{tid}] \leftarrow myP$

6. EXPERIMENTAL RESULTS

In order to compare and analyze the performance of the algorithms described in the previous chapters, many experiments have been conducted for the matrices and graphs. The experiments are performed on a server equipped with two 8-core Intel Xeon E5-2620v4 sockets running on 2.10GHz and 192 GB memory. Hence, there exist 16 cores in total. For the GPU devices, there have been four GPUs that have been used in the experiments. Two of them were `TITAN X (Pascal)`, the other two were `GeForce GTX 980` The OS running on the server is `Ubuntu 20.04.2 LTS` with `Linux 4.4.0-66` generic kernel. The algorithms are implemented in `C++` and compiled with `NVCC 9.3.0` with `-O3` as the optimization flag, `-Xcompiler` and `-fopenmp` as the command line arguments. Multi-threaded CPU parallelization is obtained with `OpenMP` pragmas. On the other hand, parallelization on GPUs is obtained by calling device kernels from the host to make each thread on the GPU device start running.

6.1 Experiment Settings

For each variation of `Ryser`, `SpaRyser` and `SkipPer` algorithms on GPU, the `BlockDim` is set to `256`, and the `GridDim` is set to `2048`. On the other hand, `16` threads have been used in the parallel versions of `Ryser`, `SpaRyser` and `SkipPer` on CPU. For the multi gpu algorithms `Ryser-SxC-Sm-MG`, `Ryser-SxC-Sm-MG+`, `SpaRyser-SxC-Sm-MG+` and `SkipPer-SxC-Sm-MG+`, experiments have been conducted on two `TITAN`, and two `TITAN` + two `Gtx` separately. Additionally, some experiments of `Ryser-SxC-Sm-MG+` are conducted on a CPU with 8 threads along with two `TITAN` + two `Gtx`.

For the approximation algorithms `Rasmussen`, and `Scaling`, there have been various settings for each experiment along with their parallel implementations on a CPU.

For both variants of `Rasmussen` and `Scaling`, one setting which is the number of experiments of the algorithm is set in the range of `10k to 1m` separately. Additionally, variants of `Scaling` had two other settings. One of them was the threshold for scale intervals, which is denoted as `SInterval`, which indicates how many iterations should be passed for each scaling process. This setting has been set to `1`, and `5` for the experiments. The other setting is the number of times to scale at each scaling process. This setting is denoted as `STime` and it has been set to `5` in the experiments.

6.2 Experiments on Matrices

The synthetic matrices are produced where each entry of an $\texttt{dim} \times \texttt{dim}$ matrix is chosen to be nonzero or not independently using a probability. For the randomness, default *rand*() library of G++ is used. Each entry is chosen to be a nonzero with the probability of $(rand()\ \%\ 100) < (100 \times density)$ where *density* is the density value that indicates how full the matrix is. Also, the value of the entries is chosen randomly using *rand*() in the range of $[0, 5]$. For the experiments, the dimension of the matrix is chosen within the numbers of $32, 34, 36, 38, 40$ and density values are chosen within the range of $0.10, 0.20, \ldots, 0.80$. For each combination of dimension and density of the matrix, 5 samples are produced. In the experiments, in order to find the runtime of a single algorithm, 5 samples have been run, and their average has been taken as the runtime value.

6.2.1 Exact Permanent Computation

6.2.1.1 Experiments with dense matrices

The execution time results of the parallel dense algorithm on a CPU and all the dense algorithms that run on a single GPU is given in Table 6.1 for a 40×40 matrix in different density values. As it can be seen, almost all the GPU algorithms except `Ryser-Gx` yield faster execution than `ParRyser` with 16 threads. The rea-

Table 6.1 Execution times (in secs) of the algorithms on a CPU and a single GPU for dense matrices with various density values for a matrix with dimension of 40.

Density	ParRyser Threads = 16	Ryser-Gx BlockDim = 256 GridDim = 2048	Ryser-Rx BlockDim = 256 GridDim = 2048	Ryser-Sx BlockDim = 256 GridDim = 2048	Ryser-SxC BlockDim = 256 GridDim = 2048	Ryser-SxC-Sm BlockDim = 256 GridDim = 2048
0.6	2696.20	12338	900.42	862.42	859.11	259.51
0.7	2703.32	12298	897.43	857.91	854.38	259.44
0.8	2690.99	12312	900.69	858.24	858.65	259.46

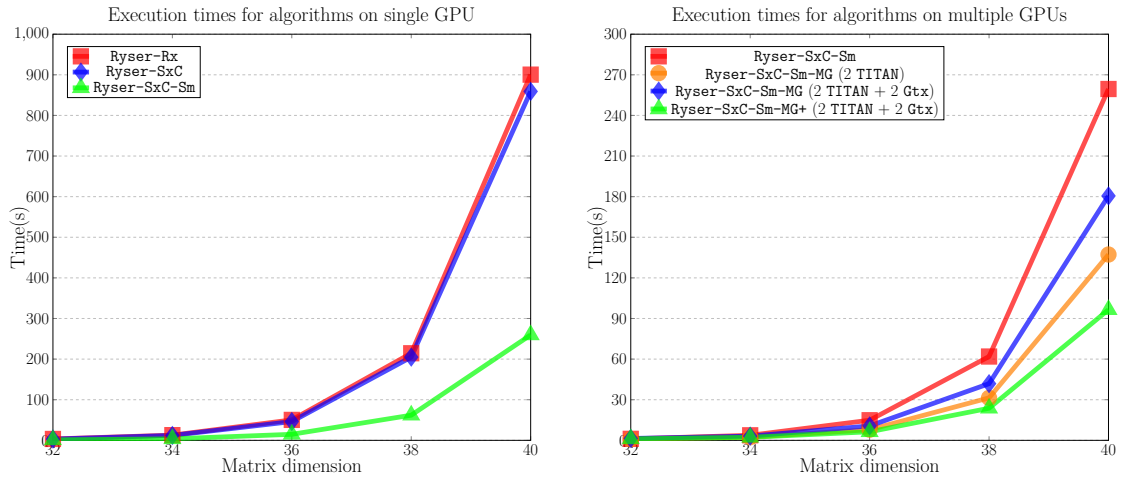
son Ryser-Gx is even slower than ParRyser is that it stores **x** in the global device memory. However, there are lots of accesses to **x** by each thread, and creating it in the registers or moving it to the shared memory helps execution to be much faster. When **x** is created on registers by each thread in Ryser-Rx, it can be seen that there is a huge improvement. However, there is even more improvement in Ryser-Sx when **x** in the shared memory. For a comparison between Ryser-Sx and Ryser-SxC, although there is a very little improvement in Ryser-SxC when there is a memory coalescing, this improvement is almost negligible since it is not much. It can be also said that a great amount of improvement is obtained when the matrix is also moved to the shared memory along with the **x** as the results of Ryser-SxC-Sm indicates. In order for better visualization, execution times for some of the algorithms that run on a single GPU can be seen in Figure 6.1 for different matrix dimensions. It is also important to note that the execution times do not change at all depending on the density of the matrix. The reason is these algorithms in Table 6.1 are dense algorithms that process the whole matrix and do not care about sparsity.

Execution times of the dense algorithms that can run on multiple GPUs can be seen in Table 6.2. The first algorithm in the table is actually Ryser-SxC-Sm that runs on a single GPU, in order to compare the improvements of multi GPU algorithms. As it can be seen, when Ryser-SxC-Sm-MG runs with two TITAN, execution is almost two times faster than Ryser-SxC-Sm which runs in a single TITAN. When the number of devices is increased by two more with two Gtx, the execution is slower for Ryser-SxC-Sm-MG with four devices than two TITAN as it can be seen in the table. The reason is that Ryser-SxC-Sm-MG divides iteration space statically as the number of GPU devices. However, TITAN can run approximately three times faster than Gtx. This means when two TITAN are finished with their chunk, they will be remained free of work until two Gtx completes their job. That's why four devices work slower than two TITAN in Ryser-SxC-Sm-MG. On the other hand, the execution became even faster than Ryser-SxC-Sm-MG with two TITAN when two TITAN and two Gtx are used in Ryser-SxC-Sm-MG+. Because in Ryser-SxC-Sm-MG+, each device dynamically takes a small chunk from iteration space at a time, and they keep working until there is no chunk left. In the experiments, Ryser-SxC-Sm-MG+ yields the best result with two TITAN and two Gtx. At the end of the Table 6.2, there is

Table 6.2 Execution times (in secs) of the algorithms on multiple GPUs for dense matrices with various density values for a matrix with dimension of 40.

Density	Ryser-SxC-Sm BlockDim = 256 GridDim = 2048	Ryser-SxC-Sm-MG BlockDim = 256 GridDim = 2048 2 TITAN	Ryser-SxC-Sm-MG BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx	Ryser-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx	Ryser-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx 8 Threads
0.60	259.51	137.25	180.55	96.48	97.09
0.70	259.44	137.26	180.61	96.52	97.00
0.80	259.46	137.27	180.54	96.43	96.93

Figure 6.1 Execution times (in secs) of the algorithms on a GPU for dense matrices with density of 0.60.



also shown the execution times of `Ryser-SxC-Sm-MG+` with four GPU devices along with a running CPU for the permanent calculation with 8 threads. Although there is an additional CPU work in this algorithm, it does not give any better result than using only four GPU devices. Furthermore, it yields slightly worse performance, which is very small difference and negligible. The reason is that when the CPU runs in `Ryser-SxC-Sm-MG+`, its performance is way slower than a single GPU device and even negligible. Therefore, when a CPU runs along with the GPU devices, it does not contribute at all. The comparison between the dense multi GPU algorithms is also illustrated in Figure 6.1 for different sizes of matrices. Again, the density does not affect the execution time since these are dense algorithms and the whole matrix is processed regardless of its sparsity.

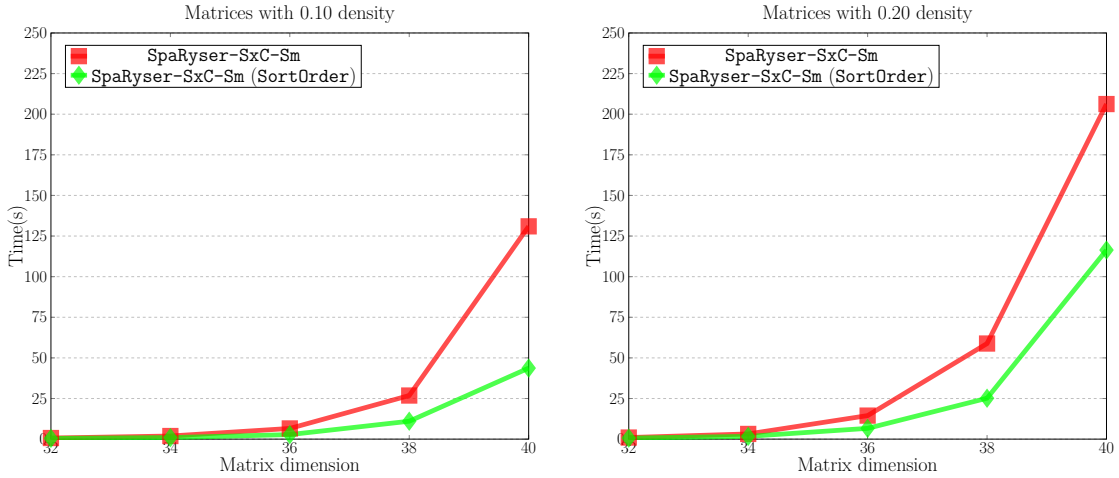
6.2.1.2 Experiments with sparse matrices

In the experiments for sparse matrices, variants of `SpaRyser` and `SkipPer` have been run on a CPU, a single GPU, and multiple GPUs. Before comparing every

Table 6.3 Execution times (in secs) of the variants of SpaRyser with and without using SortOrder for a matrix with dimension of 40.

	With SortOrder			Without SortOrder		
	ParSpaRyser Threads = 16	SpaRyser-SxC-Sm BlockDim = 256 GridDim = 2048	SpaRyser-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx	ParSpaRyser Threads = 16	SpaRyser-SxC-Sm BlockDim = 256 GridDim = 2048	SpaRyser-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx
Density	SortOrder	SortOrder	SortOrder			
0.10	360.57	43.71	16.71	981.75	130.97	48.67
0.20	783.37	116.38	43.16	1810.66	206.25	78.15
0.30	1593.18	211.31	77.93	2424.53	355.14	131.33
0.40	2598.11	312.16	114.97	4528.42	456.84	169.49

Figure 6.2 Execution times (in secs) of SpaRyser-SxC-Sm for sparse matrices with and without SortOrder.



algorithm to each other, the effect of preprocessing is tested, which are SortOrder for variants of SpaRyser, and SkipOrder for variants of SkipPer.

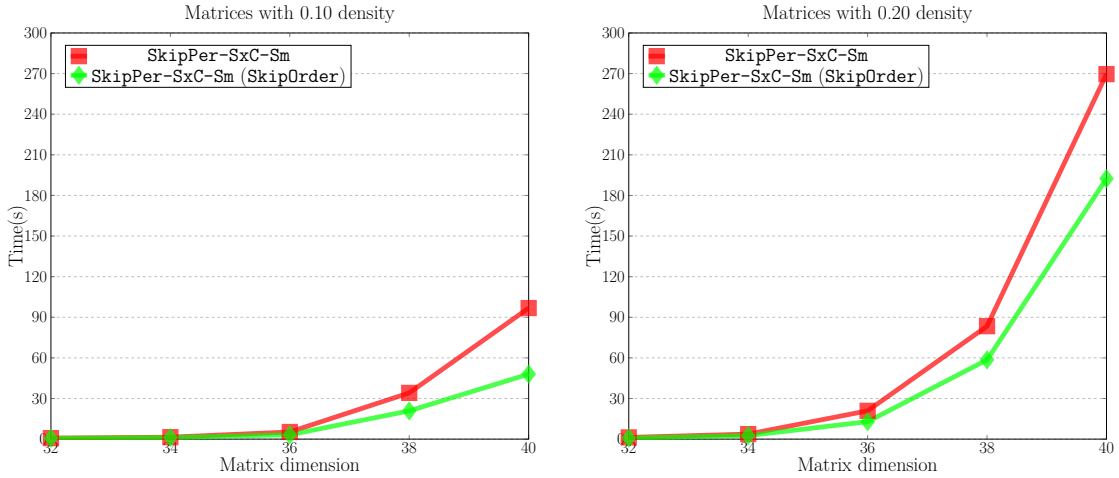
In Table 6.3, execution of three algorithms is shown with and without SortOrder, where algorithms are ParSpaRyser which runs on a CPU with 16 threads, SpaRyser-SxC-Sm which runs on a single GPU, and SpaRyser-SxC-Sm-MG+ which runs on multiple GPUs. As it can be seen, all the three algorithms, yield better performance when SortOrder is applied as a preprocessing for all the density values in $0.10, 0.20, 0.30, 0.40$. Furthermore, the ratio of the increase in performances is very similar to each other for each algorithm. It can be said that the increase in performance is usually less when the density of the matrix is higher, such that the most increase in the performance obtained when the density of the matrix is 0.10. The increase with SortOrder is illustrated in Figure 6.2 for the density values of $0.10, 0.20$ with different matrix dimensions.

In Table 6.4, execution of three algorithms is shown with and without SkipOrder, where algorithms are ParSkipPer that runs on a CPU with 16 threads, SkipPer-SxC-Sm that runs on a single GPU, and SkipPer-SxC-Sm-MG+ that runs on multiple GPUs. All three algorithms yield better performance when SkipOrder

Table 6.4 Execution times (in secs) of the variants of SkipPer with and without using SkipOrder for a matrix with dimension of 40.

	With SkipOrder			Without SkipOrder		
	ParSkipPer Threads = 16	SkipPer-SxC-Sm BlockDim = 256 GridDim = 2048	SkipPer-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx	ParSkipPer Threads = 16	SkipPer-SxC-Sm BlockDim = 256 GridDim = 2048	SkipPer-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx
Density	SkipOrder	SkipOrder	SkipOrder			
0.10	261.06	48.01	22.59	440.99	96.82	28.86
0.20	717.56	192.40	76.48	799.58	269.62	117.15
0.30	1279.66	363.98	167.53	1885.42	516.54	224.65
0.40	2110.20	489.19	213.91	3786.77	603.29	242.03

Figure 6.3 Execution times (in secs) of SkipPer-SxC-Sm for sparse matrices with and without SkipOrder.



is applied as a preprocessing. Regarding the ratio of the increase in performance for the algorithms on a single or multiple GPUs, it sometimes differs unlike the similarity in SortOrder for SpaRyser variants. For example, for a matrix with a density of 0.10, there is almost two times increase in the performance of SkipPer-SxC-Sm, whereas SkipPer-SxC-Sm-MG+ has around 1.27 times increase when SkipOrder is applied. Also, When the density is higher, it can be argued that the ratio of increase in performance is lower. However, it is not so clear as it was the case in SortOrder for SpaRyser variants. The increase with SkipOrder is illustrated in Figure 6.3 for the density values of $0.10, 0.20$ with different matrix dimensions.

In the Table 6.5, execution times of ParRyser, ParSpaRyser, and ParSkipPer algo-

Table 6.5 Execution times (in secs) of the algorithms on a CPU for generic and binary integer matrices with matrix dimension is 40.

	Generic Matrices			Binary Matrices		
	ParRyser Threads = 16	ParSpaRyser Threads = 16 SortOrder	ParSkipPer Threads = 16 SkipOrder	ParRyser Threads = 16	ParSpaRyser Threads = 16 SortOrder	ParSkipPer Threads = 16 SkipOrder
Density						
0.1	2685.65	360.57	261.06	2688.95	296.12	1.06
0.2	2709.29	783.37	717.56	2703.33	719.44	19.27
0.3	2703.84	1593.18	1279.66	2705.20	1682.33	155.24
0.4	2756.97	2598.11	2110.20	2741.17	2832.78	1123.64

Table 6.6 Execution times (in secs) of the algorithms on a GPU for generic and binary integer matrices with matrix dimension is 40.

	Generic Matrices			Binary Matrices		
	Ryser-SxC-Sm BlockDim = 256 GridDim = 2048	SpaRyser-SxC-Sm BlockDim = 256 GridDim = 2048 SortOrder	SkipPer-SxC-Sm BlockDim = 256 GridDim = 2048 SkipOrder	Ryser-SxC-Sm BlockDim = 256 GridDim = 2048	SpaRyser-SxC-Sm BlockDim = 256 GridDim = 2048 SortOrder	SkipPer-SxC-Sm BlockDim = 256 GridDim = 2048 SkipOrder
Density						
0.10	259.34	43.71	48.01	259.29	43.05	0.94
0.20	259.35	116.38	192.40	259.33	116.79	22.68
0.30	259.35	211.31	363.98	259.37	215.14	131.23
0.40	260.83	312.16	489.19	259.35	323.94	273.65

rithms are shown for both generic and binary sparse matrices. For generic matrices, both `ParSpaRyser`, and `ParSkipPer` yield better performance than `ParRyser` on a CPU with 16 threads for the density values of $0.10, 0.20, 0.30, 0.40$. As the density of the matrix is higher, the increase in performance is lower for `ParSpaRyser` and `ParSkipPer` since sparse data structures are used and the possibility of a big jump in iterations is higher for `ParSkipPer` when the matrix is more sparse. Furthermore, although the dense algorithm `Ryser` yields worse performance in all density values than the sparse algorithms, the execution times are very close to each other when the density is 0.40. It can be also said that `ParSkipPer` yields slightly better performance results than `ParSpaRyser` on generic matrices. On the other hand, for the binary matrices, the only big difference is observed for `ParSkipPer`. Here, the execution time of `ParRyser` does not change at all. For `ParSpaRyser`, the execution times differ slightly, but it is not changing always in the favor of the performance. In `ParSkipPer`, the execution times are improved remarkably in binary matrices. The reason is that the logic behind `SkipPer` is to skip as many iterations as possible when there is a zero inside the elements of $\mathbf{x}$. At each iteration, an element is either added to or subtracted from the ith element of $\mathbf{x}$, and when that element is always one or zero, it is more possible $\mathbf{x}$ remains with a zero and there will be a jump within iterations. Therefore, there is a huge improvement in `ParSkipPer` for binary matrices than the generic ones.

In Table 6.6, the comparison for the execution times of `Ryser-SxC-Sm`, `SpaRyser-SxC-Sm`, and `SkipPer-SxC-Sm` is shown for generic and binary matrices. For generic matrices, `SpaRyser` yields better performance than `Ryser-SxC-Sm` up to a density value of 0.30. As it can be seen, `SpaRyser-SxC-Sm` is slower when the matrix has 0.40 density. For `SkipPer-SxC-Sm`, it is even slower than `Ryser-SxC-Sm` when the density of the matrix is 0.30 and upward for generic matrices. Therefore, it can be said that the improvement of the sparse algorithms on a single GPU is not as much the improvements on a CPU. The reason behind this is that there is a synchronization of threads in a warp on a GPU, such that each thread in a warp runs the same command at the same time that comes with a cost when they are about

Figure 6.4 Execution times (in secs) of the algorithms on a GPU for sparse matrices with density of 0.10.

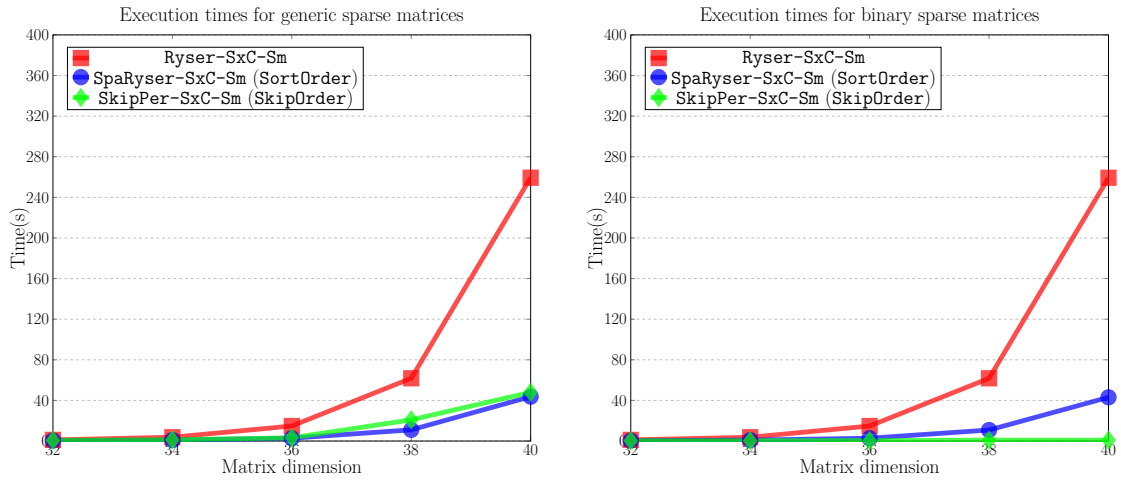


Figure 6.5 Execution times (in secs) of the algorithms on a GPU for sparse matrices with density of 0.20.

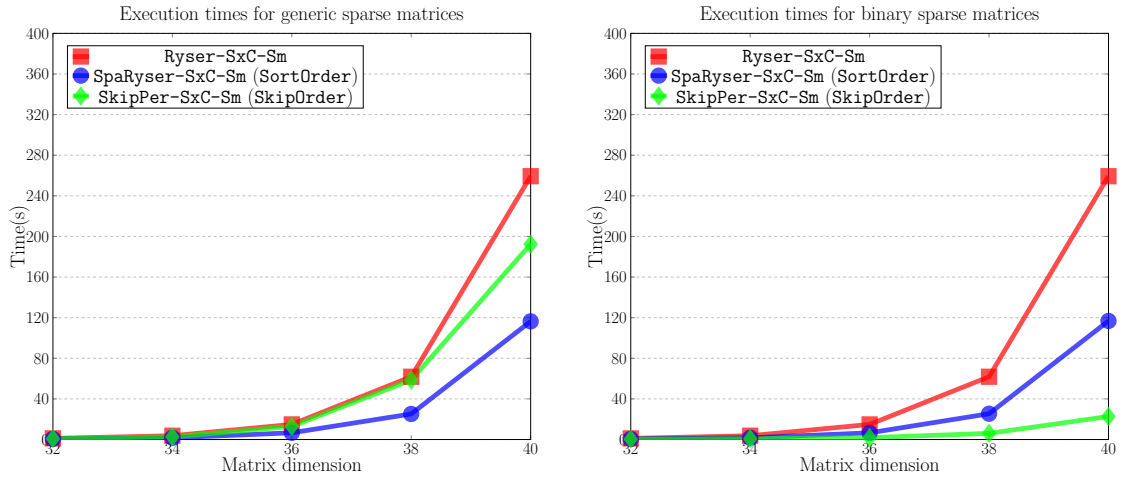


Figure 6.6 Execution times (in secs) of the algorithms on a GPU for sparse matrices with density of 0.30.

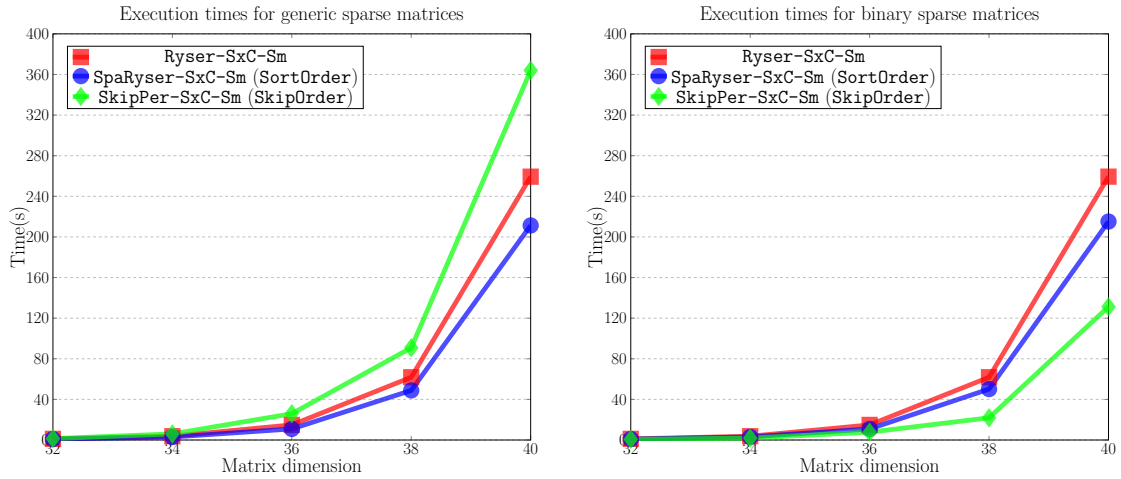


Table 6.7 Execution times (in secs) of the algorithms on multiple GPUs for generic and binary integer matrices with matrix dimension is 40.

	Generic Matrices		
	Ryser-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx	SpaRyser-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx	SkipPer-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx
Density		SortOrder	SkipOrder
0.10	96.54	16.71	22.59
0.20	96.55	43.16	76.48
0.30	96.53	77.93	167.53
0.40	96.70	114.97	213.91
	Binary Matrices		
	Ryser-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx	SpaRyser-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx	SkipPer-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx
Density		SortOrder	SkipOrder
0.10	96.55	16.44	1.81
0.20	96.58	43.51	9.67
0.30	96.55	78.47	64.37
0.40	96.65	119.35	106.78

to run different commands. As it can be seen in the table, SkipPer-SxC-Sm is also slower than SpaRyser-SxC-Sm for generic matrices, which was again the opposite on a CPU. Again the synchronization on a GPU is the reason behind this since each thread may jump different number of iterations and some finish earlier while others are still working. Therefore, it can be said that the synchronization creates larger concern for variants of SkipPer on a GPU. On the other hand, SkipPer-SxC-Sm is running faster than SpaRyser-SxC-Sm for binary matrices with all the density values. However, both algorithms still are not able to over-perform Ryser-SxC-Sm when the density is 0.40. So, SpaRyser-SxC-Sm and SkipPer-SxC-Sm make an improvement for performance up to density value of 0.30 inclusive. For illustration, the execution times of Ryser-SxC-Sm, SpaRyser-SxC-Sm, and SkipPer-SxC-Sm are given for matrices of different dimensions with $0.10, 0.20, 0.30$ density values in Figure 6.4, 6.5, 6.6 respectively.

In Table 6.7, the execution times are given for Ryser-SxC-Sm-MG+, SpaRyser-SxC-Sm-MG+, and SkipPer-SxC-Sm-MG+ for both generic and binary matrices using four GPU devices (two TITAN + two Gtx). The results are very similar to the results in Table 6.6 for algorithms that run on a single GPU. That means SpaRyser-SxC-Sm-MG+ runs faster than SkipPer-SxC-Sm-MG+ for generic matrices. SpaRyser-SxC-Sm-MG+ also yields faster execution than Ryser-SxC-Sm-MG+ up to 0.30 density inclusive, whereas SkipPer-SxC-Sm-MG+ runs faster than Ryser-SxC-Sm-MG+ up to 0.20 density inclusive. On the other hand, for binary matrices, SkipPer-SxC-Sm-MG+ runs faster than SpaRyser-SxC-Sm-MG+ in all density values. However, they both fail to over perform Ryser-SxC-Sm-MG+ when the density is 0.40. The execution times of these algorithms are illustrated for matrices of different dimensions with $0.10, 0.20, 0.30$ density values in Figure 6.7,

Figure 6.7 Execution times (in secs) of the algorithms on multiple GPUs for sparse matrices with density of 0.10.

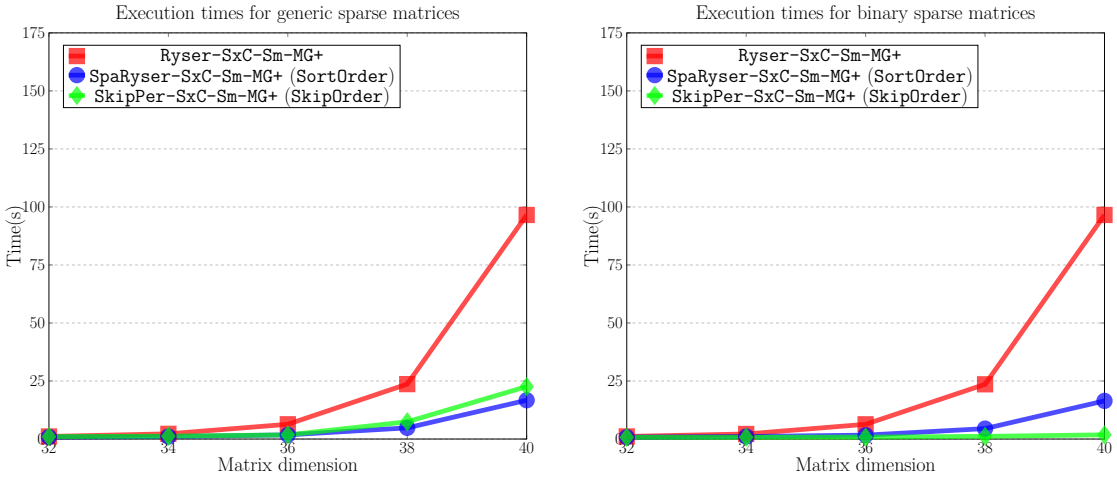


Figure 6.8 Execution times (in secs) of the algorithms on multiple GPUs for sparse matrices with density of 0.20.

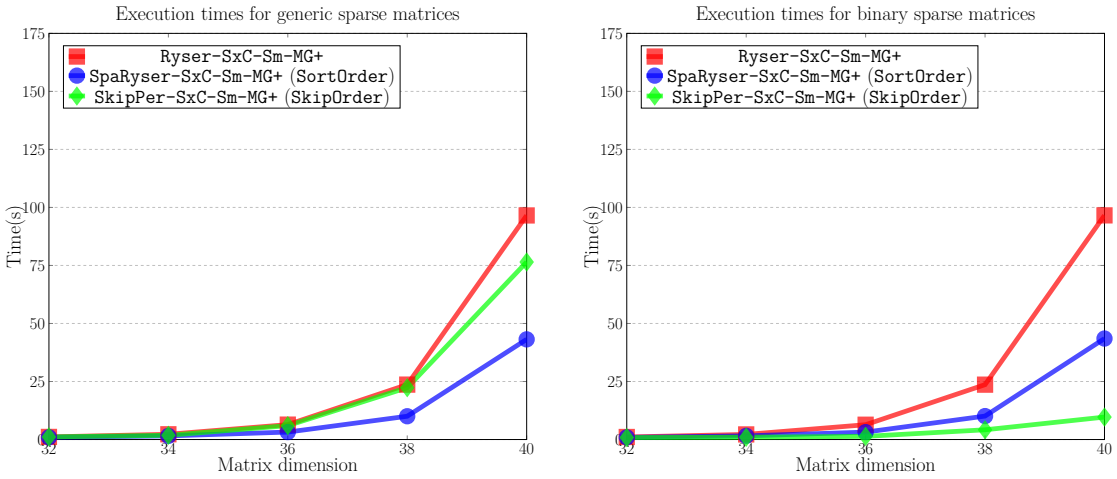


Figure 6.9 Execution times (in secs) of the algorithms on multiple GPUs for sparse matrices with density of 0.30.

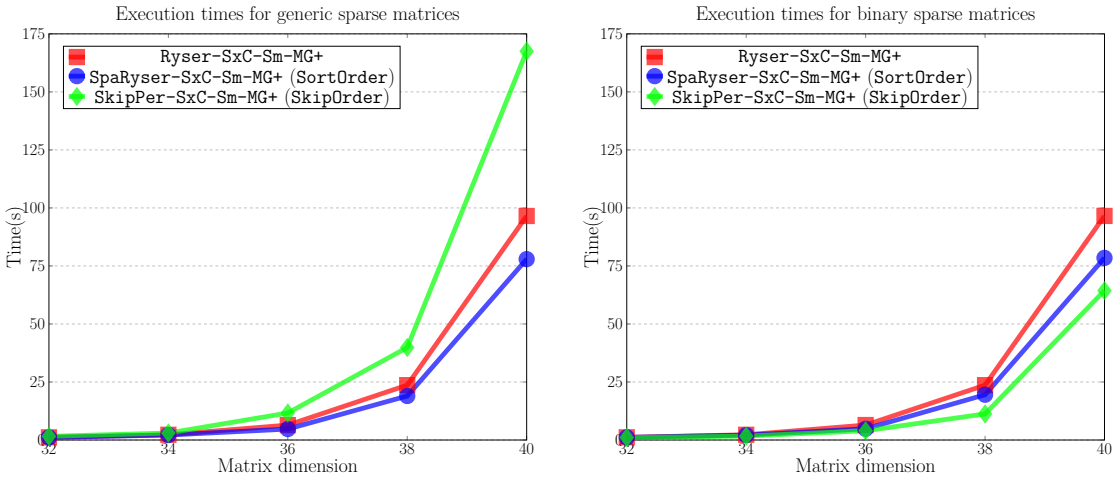


Table 6.8 Execution times (in secs) of the algorithms for dense matrices with integer, float, and double data types.

	integer		
	ParRyser Threads = 16	Ryser-SxC-Sm BlockDim = 256 GridDim = 2048	Ryser-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048
Density			2 TITAN + 2 Gtx
0.60	2696.21	259.51	96.48
0.70	2703.32	259.44	96.52
0.80	2690.99	259.46	96.43
	float		
	ParRyser Threads = 16	Ryser-SxC-Sm BlockDim = 256 GridDim = 2048	Ryser-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048
Density			2 TITAN + 2 Gtx
0.60	2132.18	260.83	96.68
0.70	2126.06	260.92	96.80
0.80	2126.69	260.08	96.52
	double		
	ParRyser Threads = 16	Ryser-SxC-Sm BlockDim = 128 GridDim = 2048	Ryser-SxC-Sm-MG+ BlockDim = 128 GridDim = 2048
Density			2 TITAN + 2 Gtx
0.60	2843.94	586.10	217.07
0.70	2854.14	586.45	217.26
0.80	2838.29	582.70	217.07

6.8, 6.9 respectively.

6.2.2 Single vs. Double Precision

The performance of the variants of `Ryser` has been tested for matrices with different data types, such as integer, float, and double. In Table 6.8, the performance results of the algorithms are shown for different data types. First of all, there is no difference in any of the data types when the density of the matrix changes, since they are all dense algorithms. On a CPU, `ParRyser` has the fastest execution times when the data type of the matrix is float, whereas the slowest execution is obtained when the data type is double. On the other hand, for `Ryser-SxC-Sm` and `Ryser-SxC-Sm-MG+` on a GPU, execution times are almost the same when the data type of the matrix is integer or float. However, when the data type of the matrix is double, the execution times of `Ryser-SxC-Sm` and `Ryser-SxC-Sm-MG+` become much slower. The reason behind this is that `BlockDim` being used is 128 when the data type of double is used. Because `x` makes use of $\texttt{BlockDim} \times \texttt{dim} \times 8$ *bytes* of memory which should be smaller than $48KB$. Therefore, it is not possible to use 256 for `BlockDim` in `Ryser-SxC-Sm` and `Ryser-SxC-Sm-MG+` when the matrix dimension is 40 and the data type is double. The illustration of the execution times for `Ryser-SxC-Sm` and `Ryser-SxC-Sm-MG+` is given in Figure 6.10 for different data types and different

Figure 6.10 Execution times (in secs) of the algorithms for dense matrices with different data types with density of 0.60.

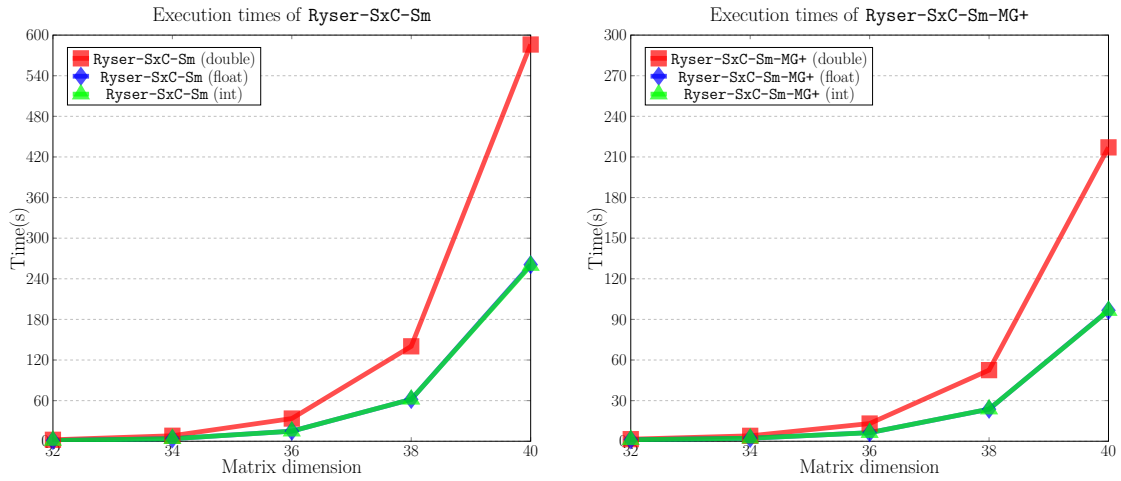


Table 6.9 Execution times (in secs) of variants of `Rasmussen` for a dense matrix with dimension of 40 when number of experiments is one million.

Density	`ParRasmussen`	`ParRasmussen+`	`RasmussenGpu`	`Rasmussen+Gpu`
`0.8`	18.89	21.86	0.32	0.43

matrix dimensions.

6.2.3 Approximate Permanent Computation

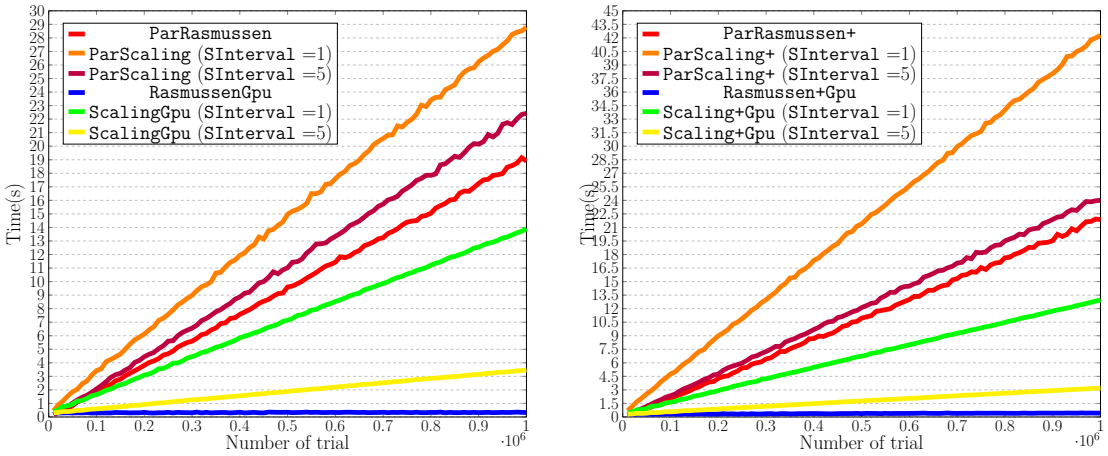
6.2.3.1 Experiments with dense matrices

In the Table 6.9, the execution times are given for the implementations of `Rasmussen` and `Rasmussen+` when there are one million experiments. As it can be seen, `ParRasmussen+` is slightly slower than `ParRasmussen` since `ParRasmussen+` makes an extra iteration over the matrix to find the row with the least nonzero elements. On the other hand, algorithms on a single GPU which are `RasmussenGpu` and `Rasmussen+Gpu` are much faster than the algorithms running on a CPU for one million experiments. Furthermore, `RasmussenGpu` and `Rasmussen+Gpu` yield execution times both under one second. It can be also said that `Rasmussen+Gpu` is slightly slower than `RasmussenGpu`. However, this difference can be neglected since the execution times on a GPU are very fast, under one second.

Table 6.10 Execution times (in secs) of variants of `Scaling` for a dense matrix with dimension of 40 when number of experiments is one million.

Density	ParScaling SInterval = 5 STime = 5	ParScaling+ SInterval = 5 STime = 5	ScalingGpu SInterval = 5 STime = 5	Scaling+Gpu SInterval = 5 STime = 5
0.8	22.43	24.00	3.46	3.20
Density	**ParScaling SInterval = 1 STime = 5**	**ParScaling+ SInterval = 1 STime = 5**	**ScalingGpu SInterval = 1 STime = 5**	**Scaling+Gpu SInterval = 1 STime = 5**
0.8	28.79	42.27	13.89	12.96

Figure 6.11 Execution times (in secs) of the approximation algorithms for dense matrices with density of 0.80.



In Table 6.10, the execution times of `ParScaling`, `ParScaling+`, `ScalingGpu`, and `Scaling+Gpu` are given. While comparing the execution times, two different settings are used where the difference is the value of `SInterval` where it is set to 1 and 5 separately. For `ParScaling` and `ParScaling+`, the performance is faster when `SInterval` is 5 since there is less scaling process. It can be also said that for both settings of `SInterval`, `ParScaling+` is slower than `ParScaling`. On the other hand, for the GPU implementations, `ScalingGpu` and `Scaling+Gpu` are much faster than `ParScaling` and `ParScaling+` for both settings of `SInterval`. Within the GPU algorithms, they are also executing faster when `SInterval` is 5 since there is less time to run the scaling algorithm. However, for the comparison among `ScalingGpu` and `Scaling+Gpu`, there is almost no difference for any setting of `SInterval` as opposed to the comparison between `ParScaling` and `ParScaling+`.

In Figure 6.11, the illustration of execution times for different number of experiments can be seen for variants of `Rasmussen` and `Scaling` in one side, and variants of `Rasmussen+` and `Scaling+` in the other side.

In Table 6.11, the execution times for the hybrid implementations are given for ten million experiments. As it can be seen in the table, when `Rasmussen+MGpu` was run for ten million experiments with four GPU devices, almost the same execution

Table 6.11 Execution times (in secs) of the hybrid implementations for a dense matrix with dimension of 40 when number of experiments is around ten million.

Density	Rasmussen+Gpu	Scaling+Gpu SInterval = 5 STime = 5	Scaling+Gpu SInterval = 1 STime = 5
0.80	1.41	29.14	124.59
Density	Rasmussen+MGpu 2 TITAN + 2 Gtx	Scaling+MGpu SInterval = 5 STime = 5 2 TITAN + 2 Gtx	Scaling+MGpu SInterval = 1 STime = 5 2 TITAN + 2 Gtx
0.80	1.45	16.60	68.14
Density	Rasmussen+MGpu 2 TITAN + 2 Gtx Threads = 8	Scaling+MGpu SInterval = 5 STime = 5 2 TITAN +2 Gtx Threads = 8	Scaling+MGpu SInterval = 1 STime = 5 2 TITAN + 2 Gtx Threads = 8
0.80	1.65	16.43	68.11

time is obtained as the execution time of `Rasmussen+Gpu` with one GPU device. The reason is that `Rasmussen+Gpu` algorithm is already capable of running ten million experiments around 1 second. Adding multiple GPUs and dividing ten million experiments among those GPUs does not contribute much since there is also an overhead of starting a device kernel, allocation of the shared memory, etc. On the other hand, for the hybrid implementations for `Scaling+`, there is an improvement since `Scaling+Gpu` is already taking long for ten million experiments and overhead for starting a kernel does not really matter. `Scaling+MGpu` almost doubled the performance for both values of `SInterval` as 5 and 1. It is also important to note that $2x$ performance is obtained using four GPU devices instead of one due to the difference of the GPU devices. `Gtx` is working slower than `TITAN` and this causes an overhead at the end if `Gtx` takes the last chunk. Finally, all the hybrid implementations are tested when the CPU is also benefited during calculations. However, when CPU is benefited, there is no improvement in neither `Rasmussen+MGpu` nor `Scaling+MGpu`. Because a single GPU device works very fast compared to the CPU, such that the work that is completed by the CPU becomes negligible.

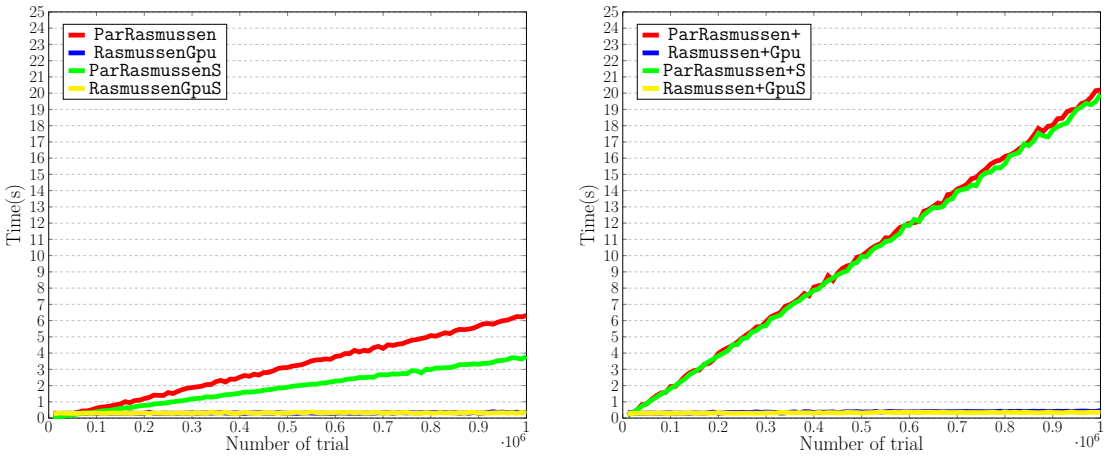
6.2.3.2 Experiments with sparse matrices

In Table 6.12, the comparison of the execution times of dense and sparse algorithms of the variants of `Rasmussen` and `Rasmussen+` is given. First of all, it can be said that the execution time of all the dense algorithms increases as the density increases. The reason is that there are more zero permanents within the results when the matrix is sparser that leads to termination of the experiment earlier. It can be also said that the decrease in performance is higher in `ParRasmussen` compared to

Table 6.12 Execution times (in secs) of variants of `Rasmussen` and `Rasmussen+` for sparse matrices with dimension of 40 when number of experiments is one million.

Density	ParRasmussen	ParRasmussen+	RasmussenGpu	Rasmussen+Gpu
0.10	6.31	20.20	0.34	0.46
0.20	13.06	21.73	0.33	0.44
0.30	16.68	22.88	0.32	0.46
0.40	18.90	22.47	0.34	0.43
Density	ParRasmussenS	ParRasmussen+S	RasmussenGpuS	Rasmussen+GpuS
0.10	3.80	19.92	0.31	0.35
0.20	7.90	21.44	0.34	0.36
0.30	10.88	22.56	0.32	0.40
0.40	12.99	22.63	0.32	0.40

Figure 6.12 Execution times (in secs) of variants of `Rasmussen` and `Rasmussen+` for sparse matrices with density of 0.10.



`ParRasmussen+` as the density gets higher. Because `ParRasmussen+` makes use of heuristic to find the row with the least nonzero elements and this leads to less zero permanents as the results of the experiments. It is hard to tell anything about `RasmussenGpu` and `Rasmussen+Gpu` since they are so much faster than the parallel implementations on the CPU and both of them are under one second. Therefore, there does not seem any decrease in performance as density gets higher, but it can be said that `RasmussenGpu` runs faster than `Rasmussen+Gpu`. When it comes to sparse implementations, as it can be seen `ParRasmussenS` improved runtime for all the density values in $0.10, 0.20, 0.30, 0.40$ compared to `ParRasmussen` where the most improvement is obtained when the density is 0.10. It can be also said that `ParRasmussen+S` slightly improves `ParRasmussen+` when the density is 0.10, but it is not so clear and there is almost no improvement for the other density values. On the other hand, it is again hard to say anything about `RasmussenGpuS` and `Rasmussen+GpuS` whether they improved the dense implementations or not since they are also very fast with an execution time under one second. Figures 6.12, 6.13 give an illustration of the comparison of the execution times between dense and sparse algorithms for the variants of `Rasmussen` and `Rasmussen+` separately for different density values.

Figure 6.13 Execution times (in secs) of variants of `Rasmussen` and `Rasmussen+` for sparse matrices with density of 0.20.

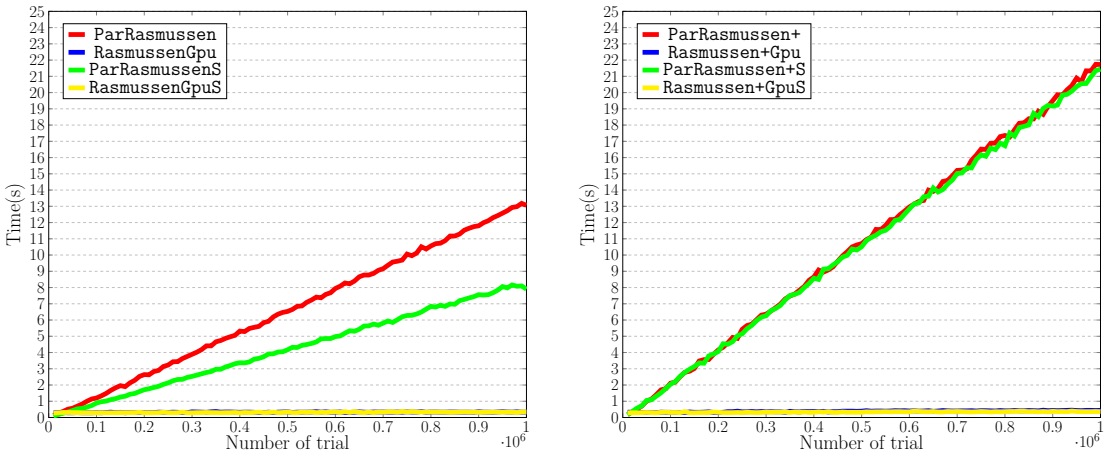


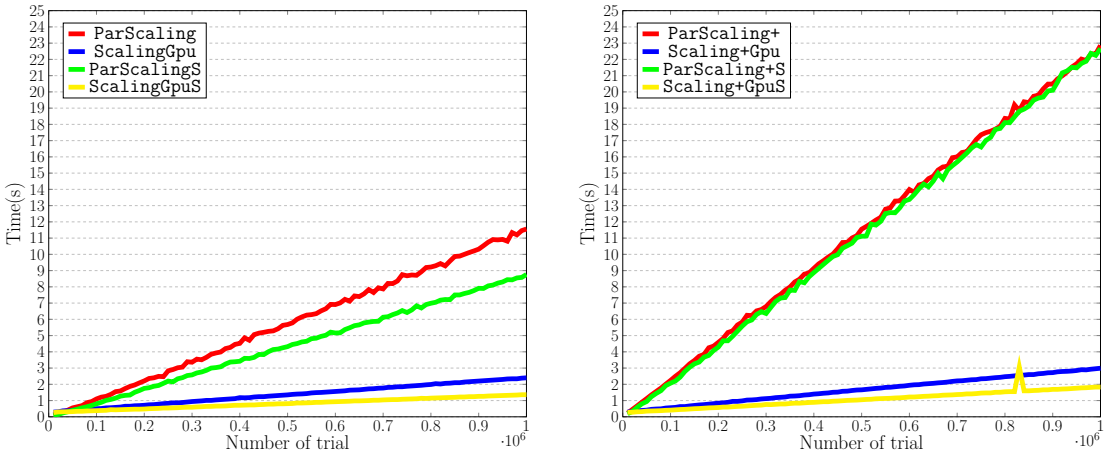
Table 6.13 Execution times (in secs) of variants of `Scaling` and `Scaling+` for sparse matrices with dimension of 40 when `SInterval` $= 5$ and number of experiments is one million.

Density	ParScaling SInterval $= 5$ STime $= 5$	ParScaling+ SInterval $= 5$ STime $= 5$	ScalingGpu SInterval $= 5$ STime $= 5$	Scaling+Gpu SInterval $= 5$ STime $= 5$
0.10	11.56	22.86	2.40	2.99
0.20	19.16	23.54	3.16	3.08
0.30	20.94	24.16	3.27	3.12
0.40	22.53	24.00	3.33	3.14
Density	ParScalingS SInterval $= 5$ STime $= 5$	ParScaling+S SInterval $= 5$ STime $= 5$	ScalingGpuS SInterval $= 5$ STime $= 5$	Scaling+GpuS SInterval $= 5$ STime $= 5$
0.10	8.76	22.67	1.36	1.83
0.20	15.34	23.90	2.08	2.15
0.30	17.62	24.25	2.30	2.38
0.40	19.26	24.28	2.52	2.59

Table 6.14 Execution times (in secs) of variants of `Scaling` and `Scaling+` for sparse matrices with dimension of 40 when `SInterval` $= 1$ and number of experiments is one million.

Density	ParScaling SInterval $= 1$ STime $= 5$	ParScaling+ SInterval $= 1$ STime $= 5$	ScalingGpu SInterval $= 1$ STime $= 5$	Scaling+Gpu SInterval $= 1$ STime $= 5$
0.10	23.36	42.87	11.48	12.50
0.20	28.00	42.20	13.46	12.76
0.30	28.43	42.79	13.71	12.89
0.40	29.17	42.64	13.77	12.88
Density	ParScalingS SInterval $= 1$ STime $= 5$	ParScaling+S SInterval $= 1$ STime $= 5$	ScalingGpuS SInterval $= 1$ STime $= 5$	Scaling+GpuS SInterval $= 1$ STime $= 5$
0.10	16.53	23.85	5.89	7.60
0.20	22.21	25.26	8.76	9.21
0.30	23.45	30.52	9.44	9.93
0.40	24.22	37.60	10.21	10.63

Figure 6.14 Execution times (in secs) of variants of `Scaling` and `Scaling+` when `SInterval` = 5 for sparse matrices with density of 0.10.



In Table 6.13 and Table 6.14, comparison of the execution times of dense and sparse algorithms of the variants of `Scaling` and `ScalingGpu` is given when `SInterval` is 5 and 1 respectively. For both settings, the tables are very similar in terms of comparison between the dense and sparse algorithms. First of all, it can be said that `ParScaling` and `ScalingGpu` have better performance when the density is low due to having more zero permanent results and terminating experiments earlier, whereas runtime of `ParScaling+` and `Scaling+Gpu` do not change at all as the density changes due to the heuristic they use. When `SInterval` is 5, `ParScalingS` improves `ParScaling` for each density value in the table. It can be said that `ParScalingS` improves the most when the density is 0.10, whereas `ParScaling+S` performs almost the same as `ParScaling+`. For the GPU algorithms, `ScalingGpuS` and `Scaling+GpuS` run faster than their dense versions for all the density values. They also over perform the most when the density value is 0.10. On the other hand, when `SInterval` is 1, all the sparse implementations `ParScalingS`, `ParScaling+S`, `ScalingGpuS`, and `Scaling+GpuS` over perform their dense implementations. Furthermore, the ratio of the increase in the performance is little higher than the performance when the `SInterval` is 5. The reason of this when `SInterval` is 1, there is a scaling operation at each iteration which means iterating over the matrix and sparse algorithms can exploit the sparsity of the matrix more using `CCS` and `CRS` when there are more iterations over the matrix. Figure 6.14, 6.15, 6.16, and 6.17 give an illustration of comparison of the execution times between the dense and sparse algorithms for the variants of `Scaling` and `Scaling+` separately for different density values and different values of `SInterval` setting.

In Table 6.15, the sparse implementations of the hybrid approximation algorithms are compared with their dense implementations for the density values of $0.10, 0.20, 0.30, 0.40$. As it can be seen, for both `Rasmussen+MGpuS` and

Figure 6.15 Execution times (in secs) of variants of `Scaling` and `Scaling+` when `SInterval` = 5 for sparse matrices with density of 0.20.

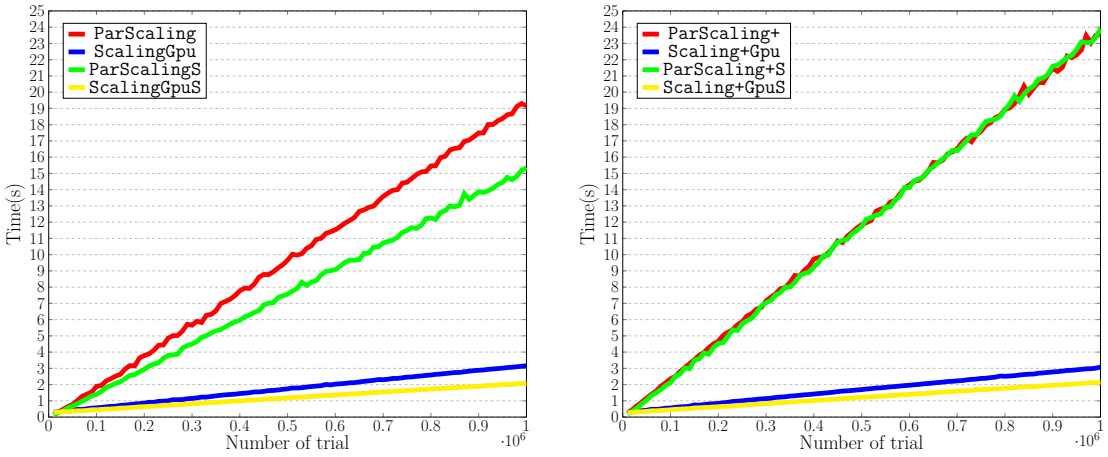


Figure 6.16 Execution times (in secs) of variants of `Scaling` and `Scaling+` when `SInterval` = 1 for sparse matrices with density of 0.10.

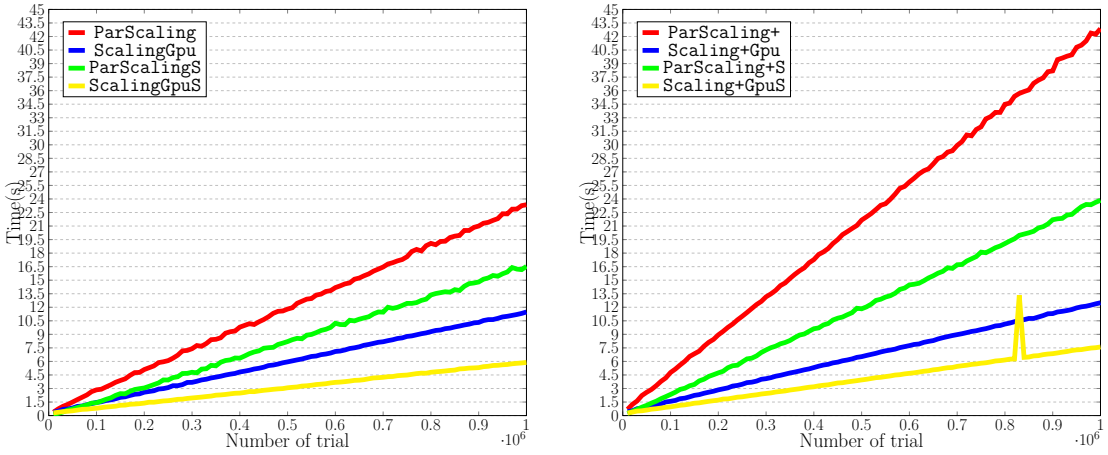


Figure 6.17 Execution times (in secs) of variants of `Scaling` and `Scaling+` when `SInterval` = 1 for sparse matrices with density of 0.20.

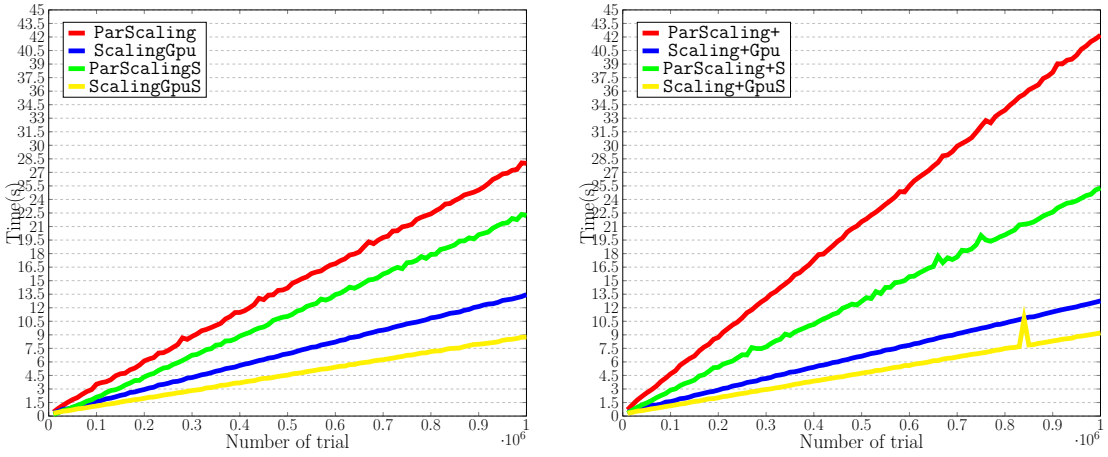
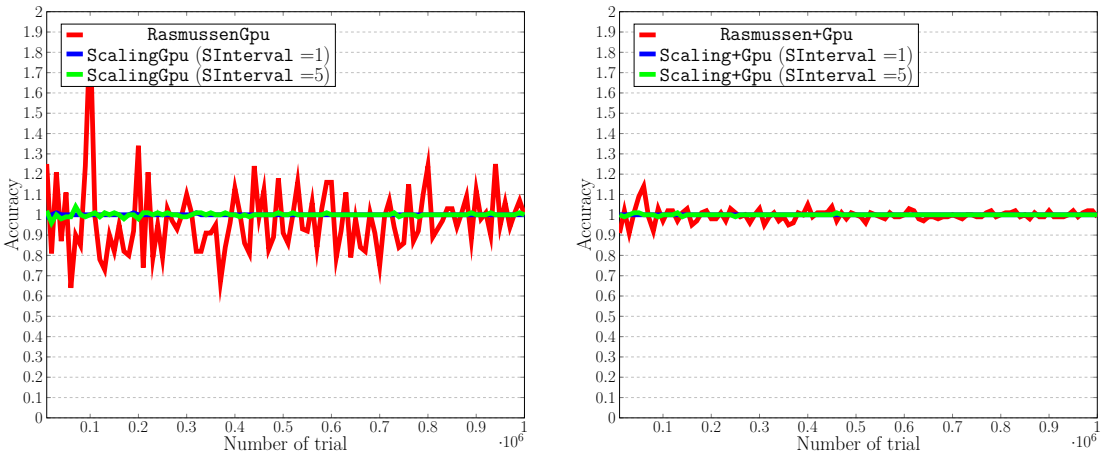


Table 6.15 Execution times (in secs) of the sparse hybrid implementations for sparse matrices with dimension of 40 when number of experiments is ten million.

Density	Rasmussen+MGpu 2 TITAN + 2 Gtx	Scaling+MGpu SInterval = 5 2 TITAN + 2 Gtx	Scaling+MGpu SInterval = 1 2 TITAN + 2 Gtx	Rasmussen+MGpuS 2 TITAN + 2 Gtx	Scaling+MGpuS SInterval = 5 2 TITAN +2 Gtx	Scaling+MGpuS SInterval = 1 2 TITAN + 2 Gtx
0.10	1.38	15.25	64.60	0.94	5.78	24.00
0.20	1.46	14.94	65.06	1.03	7.46	32.06
0.30	1.46	15.70	67.66	1.05	9.62	38.00
0.40	1.34	16.12	67.81	1.09	10.31	42.97

Figure 6.18 Accuracy of the algorithms for matrices with density of 0.20.



`Scaling+MGpuS` improved `Rasmussen+MGpu` and `Scaling+MGpu` respectively for the given density values. For `Rasmussen+MGpuS`, it can be said that it improved the performance the most when the density is 0.10, and the lowest improvement when the density is 0.40 since sparse data structures exploit the sparsity and make the algorithm run faster. Additionally, there are more experiments that return 0 and terminates when the matrix is sparser. The same situation is also valid for `Scaling+MGpuS`. In terms of improvement regarding the dense implementation, the only difference between `Rasmussen+MGpuS` and `Scaling+MGpuS` is the improvement ratio of `Scaling+MGpuS` is higher than `Rasmussen+MGpuS`. The reason is that `Rasmussen+MGpu` already runs around 1 second and it is hard to improve this execution time much since there are also overheads in starting device kernels for each GPU. Also, regarding `Scaling+MGpuS`, both of the time when `SInterval` is 1 and 5, improvement ratios are mostly the same.

6.2.3.3 Accuracy

Accuracy results of the discussed algorithms in the previous chapter, which are the variants of `Rasmussen`, `Rasmussen+`, `Scaling`, and `Scaling+`, are given in this

Figure 6.19 Accuracy of the algorithms for matrices with density of 0.30.

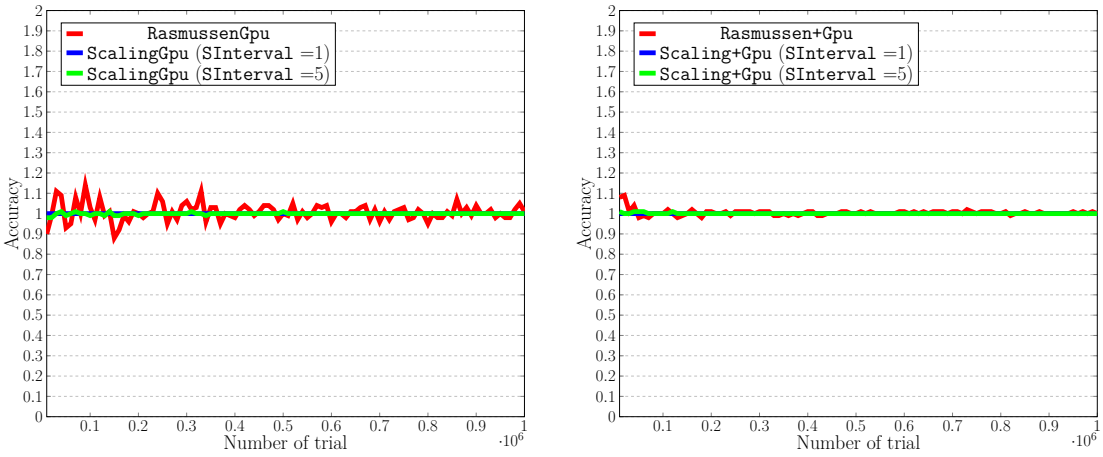
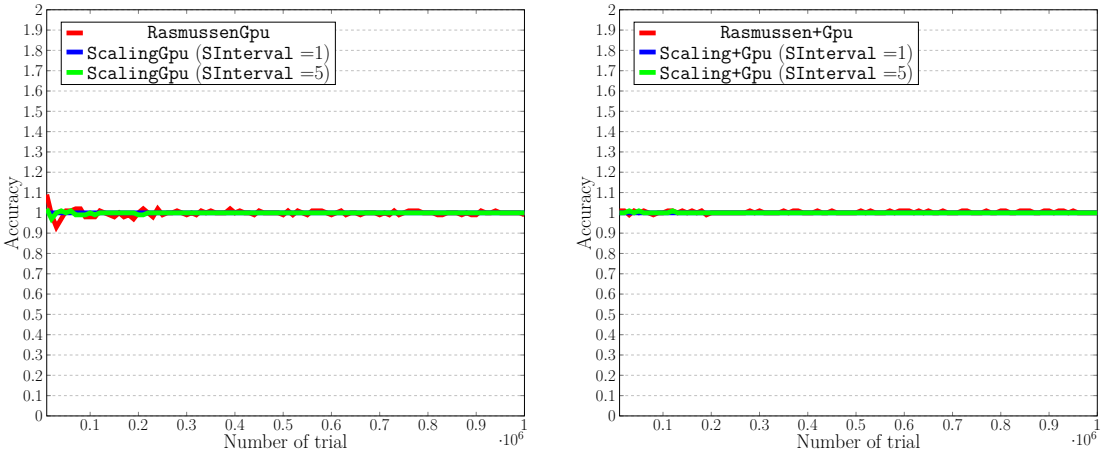


Figure 6.20 Accuracy of the algorithms for matrices with density of 0.40.

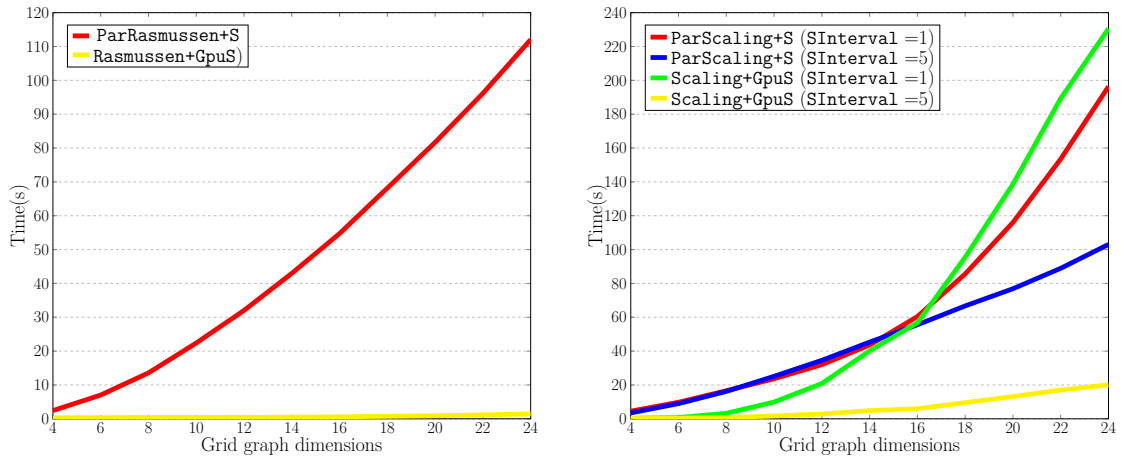


chapter. Accuracy ratio results are calculated by finding the ratio between the estimated value and the actual permanent value. Figure 6.18, 6.19, and 6.20 give the accuracy results of the algorithms when the density of the matrix is $0.20, 0.30, 0.40$ respectively. It can be said that for all the algorithms that are variants of `Rasmussen` and `Scaling` and for all the density values, the best estimation is obtained using variants of `Scaling` with `SInterval` is 1 since scaling is applied in this version the most. Then the second best is variants of `Scaling` with `SInterval` is 5 which is also estimating better than the variants of `Rasmussen`, that means scaling improves the accuracy of the estimation. Similarly, the same order in terms of accuracy is applied to the variants of `Rasmussen+` and `Scaling+`. When it comes to compare variants of `Rasmussen` and `Scaling` with variants of `Rasmussen+` and `Scaling+`, the heuristic used in `Rasmussen+` and `Scaling+` increases the accuracy of the estimation. As it can be seen in Figure 6.18, `RasmussenGpu` has some large oscillations in terms of accuracy, while `Rasmussen+Gpu` has fewer oscillations. The same applies to the comparison between `ScalingGpu` and `Scaling+Gpu` with `SInterval` is 5. However, the difference between the accuracy of `ScalingGpu` and `Scaling+Gpu` is less than the difference between the accuracy of `RasmussenGpu` and, `Rasmussen+Gpu` as it can be seen. When `SInterval` is set to 1, the results are more accurate like a straight line, as it can be seen in the figure. It is also important to note that as the density gets higher, the accuracy of each algorithm progressively increases. For instance, in Figure 6.20 when accuracy results are shown for a matrix with a density of 0.40, almost every algorithm on the right figure yields very accurate results. On the other hand, on the left figure for `RasmussenGpu`, and `ScalingGpu` with `SInterval` is 5, there are still minor oscillations when the number of experiments are low.

6.3 Experiment on Graphs

The experiments on graphs are made on grid graphs for approximating permanent value of the corresponding matrix using approximation algorithms. Grid graphs are chosen since their corresponding matrices' dimensions are very big and number of perfect matching of them can be calculated. Because, for other matrices that do not represent a grid graph and have very large dimension values, it is almost impossible to calculate the permanent value or the number of perfect matching. Therefore, grid graphs can be used to measure the performance of the approximation algorithms for very large matrices.

Figure 6.21 Execution times (in secs) of the variants of `Rasmussen+` and `Scaling+` for grid graphs.



In Figure 6.21, the execution times are given for sparse approximation algorithms that run on grid graphs for one million experiments. In the figure, grid graph dimension means that it represent $n \times n$ grid graph when the dimension is n. Also, only sparse algorithms can be used since they use `CCS` and `CRS` where memory is not allocated for all the elements of the matrix by taking into account only the nonzero elements, such that the shared memory limitations are not exceeded. Because, for a sample 24×24 grid graph, the dimension of the corresponding matrix is $(24 \times 24)/2 = 288$. As it can be seen, `Rasmussen+GpuS` runs much faster than `ParRasmussen+S` by around $100x$ when grid graph is 24×24, and number of experiments is one million. So it can be also said that as the grid graph gets bigger, the improvement of `Rasmussen+GpuS` get larger compared to `ParRasmussen+S`. On the other hand, again for 24×24 grid graph and one million experiment, a `Scaling+GpuS` run around $5\times$ faster than `ParScalingS` when `SInterval` is 5. However, when `SInterval` is 1, although `Scaling+GpuS` runs faster than `ParScaling+S` up to 16×16 grid graph, `ParScaling+S` starts to over perform `Scaling+GpuS` as the grid graph gets larger than 16×16. Therefore, it can be said that the level of exploitation of sparsity increases faster on a CPU than on a GPU as the sparsity of the matrix increases. Because, as the grid graph gets larger, the matrix gets sparser.

For the accuracy on grid graphs, the accuracy ratios can be seen in Table 6.16 for one million experiments. As it can be seen, while all the algorithms estimate the permanent value very close up to 12×12 grid graphs, `Rasmussen+GpuS` starts estimating badly for larger grid graphs. On the other hand, although `Scaling+GpuS` also estimates with a high error for 24×24 grid graph when `SInterval` is 5, it estimates very accurately when `SInterval` is 1 for all the grid graphs in the experiments.

Table 6.16 Accuracy of the approximation algorithms on grid graphs when number of experiments is one million

Algorithms	6×6	8×8	10×10	12×12	14×14
`Rasmussen+GpuS`	1.00x	1.01x	1.00x	1.09x	1.38x
`Scaling+GpuS (SInterval = 1)`	1.00x	1.00x	1.00x	1.00x	1.00x
`Scaling+GpuS (SInterval = 5)`	1.00x	1.00x	1.01x	1.02x	1.02x
Algorithms	16×16	18×18	20×20	22×22	24×24
`Rasmussen+GpuS`	0.43x	0.01x	19.32x	0.11x	0.03x
`Scaling+GpuS (SInterval = 1)`	1.00x	1.00x	1.01x	0.99x	1.00x
`Scaling+GpuS (SInterval = 5)`	1.00x	0.95x	1.72x	0.65x	0.14x

6.4 Threats to Validity

Execution times were compared for the exact permanent calculation algorithms on the GPU for matrices with data types of integer, float, and double in chapter 6.2.2. However, when the calculated permanent values of different versions of `Ryser` on the GPU are compared, there have been some permanent values observed that differ from each other for the same matrix when the matrix has a data type of float or double. This is due to the matrix of float and double elements, where each element has six numbers after the floating point. In the results, mostly up to around $10-15\%$ of error is observed for both float and double matrices. It is also important to note that, the results for integer matrices with different algorithms have always been consistent.

7. CONCLUSION

In this work, parallel algorithms that run on a CPU, on a GPU, and on multiple GPUs and a CPU in a hybrid manner are proposed for the exact permanent calculation of both dense and sparse matrices. The algorithms make use of `Ryser`, `SpaRyser`, and `SkipPer` algorithms on a GPU. In the literature, a parallel algorithm that runs with 16 threads on a CPU for a 40×40 matrix with a density of 0.50, the speedup is around 12× compared to the original `Ryser` algorithm in a single core. In the proposed approach that has the best performance for the same matrix, which is `Ryser-SxC-Sm` that copies $\mathbf{x}$ and the matrix to the shared memory with memory coalescing and runs in the single GPU, around 125× speedup is obtained compared to original `Ryser` in a single core on the CPU. Furthermore, using four GPUs that run collectively to calculate the permanent value in `Ryser-SxC-Sm-MG+`, around 338× speedup is obtained. It has also been shown that contribution of the CPU is negligible in hybrid proposed solutions. On the other hand, for sparse 40×40 binary matrices with a density of 0.10, the speedup obtained on the CPU using 16 threads using `ParSpaRyser` and `ParSkipPer` is around 9× and 2500× respectively compared to the parallel version of `Ryser` that is `ParRyser`. For the proposed algorithms `SpaRyser-SxC-Sm` and `SkipPer-SxC-Sm`, the speedup is around 6× and 275× respectively for the same matrix compared to `Ryser-SxC-Sm`. Furthermore, for a generic matrix, the speedup is around 7× and 10× for `ParSpaRyser` and `ParSkipPer`, whereas the speedup is around 6× and 5× for `SpaRyser-SxC-Sm` and `SkipPer-SxC-Sm`. Therefore, a speedup is obtained for sparse matrices on the GPU, but the speedup on the GPU is not as much as the speedup of the parallel algorithms on the CPU. Specially, while speedup of `SpaRyser` on the GPU is similar to the speedup on the CPU, there is more difference in the speedup of `SkipPer` on the GPU compared to the speedup on the CPU. Because, there is a skipping in `GrayCode` in `SkipPer`, which makes each thread's execution differ in a way that is disadvantage of the synchronization on a GPU. That's also the reason behind `SkipPer` on the GPU runs worse than `SpaRyser` on the GPU for generic matrices.

For estimating the number of perfect matchings on graphs, which is the permanent

value of the corresponding matrix, algorithms are proposed that run on a CPU, on a single GPU or multiple GPUs and a CPU in a hybrid manner by making use of `Rasmussen` and `Scaling` algorithms. Using the most accurate algorithm on the GPU, accuracy ratio of 0.999971 is obtained in 13.79 seconds, by having one million experiments for an 80% full 40×40 matrix. Again, for one million experiments for `Rasmussen`, around $60\times$ speedup is obtained on the GPU compared to the parallel version with 16 threads on the CPU. For the improved version, that is `Rasmussen+`, the same speedup is around $50\times$. On the other hand, for the same type of matrix, around $6\times$ and $2\times$ speedup is obtained in `Scaling` on the GPU compared to parallel version on the CPU with 16 threads when `SInterval` is 5 and 1 respectively. For the improved version, that is `Scaling+`, the same speedup is around $7\times$ and $3\times$. There is a way more speedup for `Rasmussen` algorithm compared to `Scaling`. Because, in `Scaling` algorithm, the part where the matrix is scaled is in disadvantage of the synchronization since it iterates `STime` many times and differently for each thread on a GPU. For `Scaling` algorithm, the speedup is also lower when `SInterval` is 1 which proves that scaling of the matrix lowers the performance on the GPU more than on the CPU. It has been also shown that using a hybrid algorithm that make use of four GPUs, the speedup is around $2\times$ in `Scaling+MGpu` compared to `Scaling+Gpu`. For sparse matrices, with the most accurate algorithm on the GPU, accuracy ratio of 0.998347 is obtained in 7.60 seconds by having one million experiments for an 10% full 40×40 matrix. For the same matrix, when the sparse parallel implementation is tested on CPU for one million experiments, there is around $2\times$ speedup is in `ParRasmussenS` compared to `ParRasmussen`. However, there is not any clear speedup is observed in `RasmussenGpuS` compared to `RasmussenGpu`. Because, both `RasmussenGpu` and `RasmussenGpuS` terminates very fast, such that most of the execution time comes from the overhead in kernel calls. For the sparse implementations of `Scaling` and `Scaling+`, the speedup on the CPU compared to their dense versions on the CPU is around 1.5-$2\times$, whereas the speedup is also 1.5-$2\times$ on the GPU compared to their dense versions on the GPU for a 40×40 matrix with a density value of 0.10. Therefore, for the sparse versions of approximation algorithms on the CPU and the GPU, while sparsity was not able to be exploited for `Rasmussen` and `Rasmussen+` on the GPU, sparsity was able to be exploited on the GPU for `Scaling` and `Scaling+` at the same level as the exploitation of sparsity on the CPU.

BIBLIOGRAPHY

Aaronson, S. & Arkhipov, A. (2010). The computational complexity of linear optics.

Dufossé, F., Kaya, K., Panagiotas, I., & Uçar, B. (2018). Approximation algorithms for maximum matchings in undirected graphs. In *Proceedings of the 8th SIAM Workshop on Combinatorial Scientific Computing*.

Huber, M. & Law, J. (2008). Fast approximation of the permanent for very dense problems. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '08, (pp. 681–689)., USA. Society for Industrial and Applied Mathematics.

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

Mittal, R. & Al-Kurdi, A. (2001). Efficient computation of the permanent of a sparse matrix. *International Journal of Computer Mathematics*, *77*(2), 189–199.

Nijenhuis, A. & Wilf, H. S. (1978). *Combinatorial Algorithms*. Academic Press.

Rasmussen, L. E. (1994). Approximating the permanent: A simple approach. *Random Struct. Algor.*, *5*(2), 349–361.

Rudolph, T. (2009). Simple encoding of a quantum circuit amplitude as a matrix permanent. *Physical Review A*, *80*(5).

Ryser, H. (1963). *Combinatorial Mathematics*. Mathematical Association of America.

Valiant, L. G. (1979). Completeness classes in algebra. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, STOC '79, (pp. 249–261)., New York, NY, USA. Association for Computing Machinery.