Computing Sparse Matrix Permanents with OpenMP, CUDA and MPI

İlhan Yavuz İskurt
Bahadır Yazıcı

Understanding Matrix Permanents

Definition: The permanent of an *n x n* matrix *A* is a function similar to the determinant, but without the alternating signs.

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

where P_n is the set of all permutations of { 1, 2, …, *n* }

Applications:

- **Quantum Computing**: Used in boson sampling to analyze the efficiency of quantum computers.
- **Bioinformatics**: Helps in computing genotype probability distributions for DNA profiling.
- **Graph Theory**: Used to count perfect matchings in bipartite graphs.

Understanding Matrix Permanents

Challenges in Computing Permanents:

- **Complexity:** Computing the permanent is a #P-complete problem, meaning it is computationally intensive and difficult to solve for large matrices.

- **Dense Matrices:** For dense matrices, even the most efficient algorithms have exponential time complexity, making them impractical for large n.

Challenges in Computing Permanents of Sparse Matrices

Sparse Matrices:

- **Definition**: A sparse matrix is one in which most of the elements are zero.

$$\begin{bmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 4 & 0 & 0 & 5 \\ 0 & 6 & 0 & 0 \end{bmatrix}$$

Why Sparse Matrices?

- **Real-world Data:** Many real-world datasets are inherently sparse, such as social networks, biological data, and web graphs.
- **Storage Efficiency:** Sparse matrices require less memory and storage space, making computations more efficient.

Challenges in Computing Permanents of Sparse Matrices

Specific Challenges:

- **Redundant Computations**: Traditional algorithms do not leverage the sparsity, leading to unnecessary calculations involving zero elements.
- **Computational Overhead**: Managing and processing non-zero elements efficiently while skipping zeros without re-evaluating the entire matrix is complex.

Goal

- **Optimization**: Develop algorithms that specifically target and optimize the computation of matrix permanents for sparse matrices.

Gray Code Optimization in Ryser’s Algorithm

Original Ryser’s Algorithm:

- Uses the inclusion/exclusion principle.
- Computes the permanent by iterating over subsets of columns.
- Each iteration recalculates row sums, leading to inefficiencies.

Gray Code Logic:

- A binary sequence where only one bit changes at each step.
- Example:
  - Binary: 000, 001, 010, 011, 100, 101, 110, 111.
  - Gray Code: 000, 001, 011, 010, 110, 111, 101, 100.

Gray Code Optimization in Ryser's Algorithm

Efficiency Gains:

- Constant Time Updates:
  - Only one bit changes per iteration, allowing the algorithm to update the row sums in constant time (O(1)).
  - No need to recompute row sums from scratch.
- Inclusion/Exclusion Principle:
  - Efficiently keeps track of which columns are included or excluded by flipping the corresponding bit in the Gray code.
  - Instead of recalculating row sums, adjust the sum by adding or subtracting the value of the changed bit's column.

Efficient Algorithms for Sparse Matrix Permanents

SpaRyser Algorithm

Overview: An optimized version of Ryser's algorithm tailored for sparse matrices.

Key Features:

- **Sparsity Exploitation**: Uses Compressed Row Storage (CRS) and Compressed Column Storage (CCS) to efficiently handle non-zero entries.
- **Vector Updates**: Reduces unnecessary computations by maintaining a count of zeros in the vector.
- **Sorting Technique**: Applies a sorting method (SortOrd) to prioritize columns with fewer non-zero elements.

Efficient Algorithms for Sparse Matrix Permanents

SkipPer Algorithm

Overview: An advanced parallel algorithm that further optimizes sparse matrix permanent computation.

Key Features:

- **Gray Code Skipping**: Skips entire blocks of iterations that do not contribute to the permanent.
- **Dynamic Scheduling**: Uses dynamic scheduling to balance computational loads across threads.
- **Preprocessing (SkipOrd)**: Orders the matrix to maximize skipping efficiency during execution.

Performance Metrics of SpaRyser and SkipPer

Key Performance Metrics

- **Execution Time**: Measures how long the algorithm takes to compute the permanent.
- **Speedup**: The ratio of execution time between the new algorithm and a baseline (e.g., Ryser’s algorithm).
- **Scalability**: How well the algorithm performs as the size of the matrix increases or as the number of processing threads increases.
- **Efficiency**: The algorithm’s ability to exploit sparsity and parallelism effectively.

Performance Metrics of SpaRyser and SkipPer

Why These Metrics Matter

- **Execution Time**: Critical for applications needing real-time or near-real-time computations.
- **Speedup**: Demonstrates the practical improvement over existing methods.
- **Scalability**: Ensures the algorithm remains efficient for large datasets and high-performance computing environments.
- **Efficiency**: Indicates how well the algorithm minimizes redundant computations and utilizes available resources.