Computing Sparse Matrix Permanents with OpenMP, CUDA and MPI

In our study, we employed two algorithms to compute the permanents of sparse matrices: SpaRyser with SortOrder and SkipPer with SkipOrder. The SpaRyser algorithm, enhanced with SortOrder, is versatile and applicable to all types of matrices. SortOrder optimizes matrix arrangement by prioritizing columns with a higher number of nonzero elements, thus streamlining the computation process.

SkipPer, on the other hand, is specifically designed for matrices with binary values (1s and 0s) and is paired with the SkipOrder. SkipOrder is a dynamic sorting algorithm that optimizes column arrangements based on their degrees, aiming to maximize the skipping of zero-contributing traversals. This optimization significantly enhances SkipPer's performance by reducing unnecessary computations.

SpaRyser exploits matrix sparsity to increase efficiency. It uses the Compressed Column Storage (CCS) representation, allowing it to skip zero entries during updates, thereby reducing the number of floating-point operations. This algorithm also incorporates a technique to avoid unnecessary multiplications by maintaining a count of zero entries and skipping product computations when zero values are detected.

SkipPer further improves efficiency by using Gray code skipping, a method that identifies and skips zero-contributing iterations. This approach is guided by the 'next' function, which calculates the earliest iteration that can make a previously zero vector entry nonzero. By dynamically updating the degree counts of columns during preprocessing, SkipOrder ensures that SkipPer processes columns with fewer nonzero entries more frequently, enhancing its performance.

Parallelization of SkipPer is achieved through coarse-grain parallelism, dividing the iteration space among multiple threads. Dynamic scheduling further balances the load, ensuring efficient use of computational resources. Experiments demonstrate that SkipPer, particularly with SkipOrder, offers substantial speedups compared to traditional algorithms, especially for highly sparse matrices.

Overall, the combination of SpaRyser and SkipPer, with their respective order optimization techniques, provides a robust framework for efficient computation of matrix permanents, leveraging both sparsity and parallelism to achieve significant performance improvements.

CPU Parallelization (OpenMP)

To fully utilize the CPU, specifically leveraging 16 threads, we implemented OpenMP for shared-memory multiprocessing. OpenMP is an API that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran. By integrating OpenMP into our algorithms, we enabled parallel processing capabilities that significantly enhance performance.

In the SpaRyser algorithm, OpenMP was used to parallelize the computation of the permanent by dividing the work among multiple threads. The algorithm restructures the matrix to prioritize columns with more nonzero elements, optimizing the overall computational efficiency.

Similarly, the SkipPer algorithm benefits from OpenMP through dynamic scheduling and load balancing. The SkipPer algorithm dynamically skips zero-contributing iterations, significantly reducing the computational load. Dynamic scheduling was particularly crucial for the SkipPer algorithm due to the behavior of the next function, which can skip many iterations. With static scheduling, some threads would end up with significantly less work than others, leading to poor thread utilization and suboptimal performance. By using dynamic scheduling, we ensured that all threads remained effectively utilized, balancing the computational load and maximizing the efficiency of parallel processing.

By integrating OpenMP into both algorithms and employing dynamic scheduling for SkipPer, we achieved substantial speedups, making the computation of matrix permanents more efficient and scalable on multicore processors. This approach demonstrates the power of parallel computing in handling complex mathematical operations, particularly in the context of sparse matrices.

GPU Parallelization (CUDA)

To further enhance the computational efficiency of our algorithms, we utilized CUDA for GPU parallelization. CUDA, a parallel computing platform and programming model developed by NVIDIA, allows for significant acceleration of computations by leveraging the power of NVIDIA GPUs.

In our implementation, the SpaRyser and SkipPer algorithms were adapted for GPU execution using CUDA. The core idea was to distribute the computational workload across the many cores of the GPU, taking advantage of their ability to handle thousands of threads simultaneously.

For the SpaRyser algorithm, CUDA was employed to parallelize the computation of matrix permanents by distributing the work of updating and multiplying nonzero elements across the GPU threads. The following steps outline the key components of the GPU implementation:

1. **Shared Memory Utilization**: Key matrix components such as column pointers (*cptrs*), row indices (*rows*), and values (*cvals*) were loaded into shared memory. This allows for faster access times compared to global memory, enhancing performance.
2. **Thread Distribution**: Each thread in the GPU was responsible for a portion of the workload. The threads were organized into blocks, with each block handling a specific segment of the matrix. This division ensures that the workload is evenly distributed and that the GPU resources are fully utilized.
3. **Synchronization**: Synchronization points (*__syncthreads()*) were used to ensure that all threads within a block completed their memory loading before proceeding with computations. This prevents race conditions and ensures data consistency.
4. **Chunk Processing**: The total number of iterations required for the permanent computation was divided into chunks, with each thread processing its designated chunk. This approach helps in managing the large number of iterations efficiently.

The SkipPer algorithm was also adapted for GPU execution with similar principles. Dynamic scheduling and efficient memory management were crucial to handle the dynamic nature of the SkipOrder function and the skipping of zero-contributing iterations.

1. **Dynamic Scheduling**: CUDA's dynamic parallelism features were leveraged to handle the dynamic nature of the next function, which skips multiple iterations. This prevents load imbalance and ensures that all GPU threads remain busy.
2. **Efficient Memory Management**: The SkipPer algorithm also utilized shared memory for frequently accessed data, minimizing the latency associated with global memory accesses.

Despite the significant performance improvements achieved through GPU parallelization, there is a scalability issue with matrices that have dimensions larger than 40x40 and contain more than 400 nonzero elements. This problem arises from the block sizes of the CUDA cores. As the matrix size and number of nonzero elements increase, the computational demands grow exponentially, and the fixed block sizes of CUDA cores become a limiting factor. This leads to potential performance bottlenecks even on powerful GPU architectures.

By employing CUDA for GPU parallelization, both SpaRyser and SkipPer algorithms achieved substantial performance improvements. The GPU's ability to handle massive parallelism allowed us to scale the computation of matrix permanents efficiently, making our approach highly effective for large and sparse matrices within certain limits. However, addressing the scalability issue for extremely large matrices remains an area for future research and optimization, particularly focusing on overcoming the limitations imposed by CUDA core block sizes.

Our Benchmarks from Leaderboard

Matrix (Times, seconds)	CPU (16 threads)	Single GPU
ey35_02	23.2047	3.44631
ey35_03	44.1529	5.50122
ey36_02	63.0541	7.86358
ey36_03	106.973	14.1624
football	15.9668	6.48959
cage5	105.717	15.0138
GD98_a	61.9248	9.65921
bcspwr01	0.180901	0.245205
chesapeake	108.844095	20.2256

Permanent	CPU (16 threads)	Single GPU
ey35_02	1.26E+15	1.26E+15
ey35_03	8.27E+19	8.27E+19
ey36_02	2.00E+15	2.00E+15
ey36_03	3.76E+21	3.67E+21
football	5.39E+17	5.39E+17
cage5	5.15E-08	5.15E-08
GD98_a	0	0
bcspwr01	3.76E+08	3.76E+08
chesapeake	1.32E+13	1.32E+13

Running the OpenMP Version

1. **Compilation**:
   - The OpenMP-enabled executable is compiled using a Makefile. To compile the code, open your terminal and navigate to the directory containing the Makefile.
   - Execute the following command: “make”
   - This command will create a binary named run in the build directory.
2. **Execution**:
   - To run the compiled binary, use the following command: “./build/run <matrix_file>”
   - Replace <matrix_file> with the path to your matrix file.
3. **Output**:
   - After the binary finishes running, it will return two numbers:
     1. The calculated permanent of the given matrix.
     2. The execution time in seconds.

Running the OpenMP Version

4. **Compilation**:
    - The CUDA-enabled executable can be compiled with a single command. Open your terminal and navigate to the directory containing the final.cu file.
    - Execute the following command: “nvcc final.cu -O3 -o run_gpu”
    - This command will create a binary named run_gpu.
5. **Execution**:
    - To run the compiled CUDA binary, use the following command: “./run_gpu <matrix_file>”
    - Replace <matrix_file> with the path to your matrix file.
6. **Output**:
    - Similar to the OpenMP version, after the binary finishes running, it will return two numbers:
        3. The calculated permanent of the given matrix.
        4. The execution time in seconds.

İlhan Yavuz İskurt 31112
Bahadır Yazıcı 30643