Parallelizing the Symmetric Gauss-Seidel Smoother Algorithm for Sparse Matrices using CUDA

Simone Messina

June 2023

1 Introduction

The Symmetric Gauss-Seidel Smoother (SYMGS) is an iterative method used to solve a system of linear equations and is similar to the Jacobi Method, with the difference that only one vector is needed for the Gauss-Seidel method. In the latest iteration, the latest computed value is used if available, so the vector is overwritten, and there is no need for two different x_{old} and x_{new} vectors. The main problem with using SYMGS is that as the matrices get larger, the serial implementation of the algorithm slows down the computation. That's why we want to develop a parallel implementation of the SYMGS-based algorithm.

1.1 Matrix format

While it is preferable for SYMGS to work with strictly diagonally dominant or symmetric and positive definite matrices, the algorithm can be applied to any matrix with non-zero elements on the diagonals. However, the convergence of results is not guaranteed. We will work with Sparse Matrices, where the number of null elements is much higher than the number of non-zero elements. To optimize the algorithm further, we will store the matrix in Compressed Sparse Row Format (CSR), which saves the number of non-zero values, as well as the row and column indices of those non-zero values. This makes it easy to locate them without iterating over every useless null element of the matrix.

1.2 Symmetric Gauss-Seidel Smoother

As mentioned before, the Gauss-Seidel method is an iterative process to solve linear equations:

$$Ax = B$$

where A is an $N \times N$ matrix, and x and b are vectors of n elements:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

The standard iteration for the algorithm is:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_j^{k} \right)$$

The only vector we use to save computation results is x, which is the same vector we use as input for the computation. The advantage of using only one vector is evident: it allows us to save a significant amount of memory, especially for large problems.
However, the main drawback of this algorithm is its limited parallelizability. Parallelization refers to the ability to divide the computation into smaller, independent tasks that can be executed simultaneously. Unfortunately, due to the data dependencies and the need to ensure up-to-date values, it becomes challenging to effectively parallelize the work in this algorithm. As a result, the potential benefits of parallel processing for faster computation are not fully utilized.

1.3 Working with dependencies

A lot of elements of x are computed at the same time, but in order to compute them, they may require other elements that have not been processed yet. As a result, they use outdated values.
To address this issue, we can simply implement a kernel that checks for data dependencies and delays the computation of elements until the next iteration. To optimize the operations, a matrix is required to track which elements have been computed and which ones need to be computed later.
By implementing this approach, the additional time required for computation is minimal, while the precision of the results improves.

2 Parallel Implementation

To implement the parallelized algorithm, we will use CUDA, which allows us to write code in popular languages like C and parallelize the algorithm using the GPU.

2.1 Inputs

The matrix is stored in CSR format, so the algorithm needs to know:

- The size of the matrix and the number of non-zero elements.
- The positions of the non-zero elements.
- The values of the non-zero elements.
- The values on the main diagonal of the matrix.

Other useful data we want to provide to the algorithm are:

- Two vectors for computing and saving the results.
- A matrix to keep track of the computed and uncomputed elements.
- An integer to determine if the algorithm has reached its end or if another iteration is required.

The function call will be:

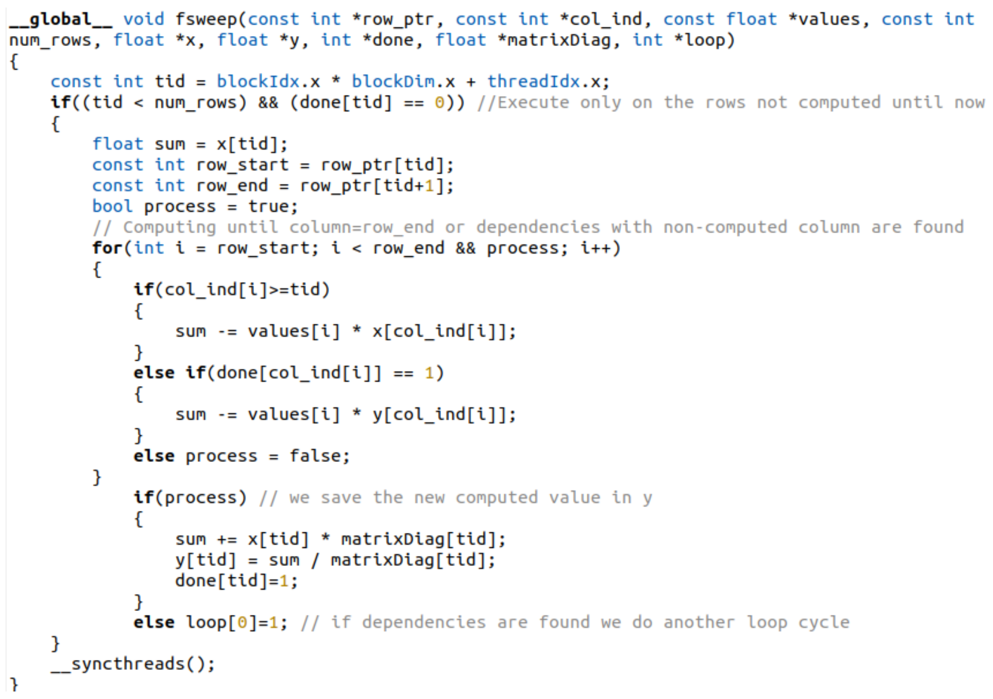
```
__global__ void kernel(const int *row_ptr, const int *col_ind, const float *values, const
int num_rows, float *x, float *y, int *done, float *matrixDiag, int *loop)
```

2.2 Kernels

There will be two complementary kernels working one after another.

The first kernel (Forward Sweep) iterates through the rows of the system, from the first row to the last row:

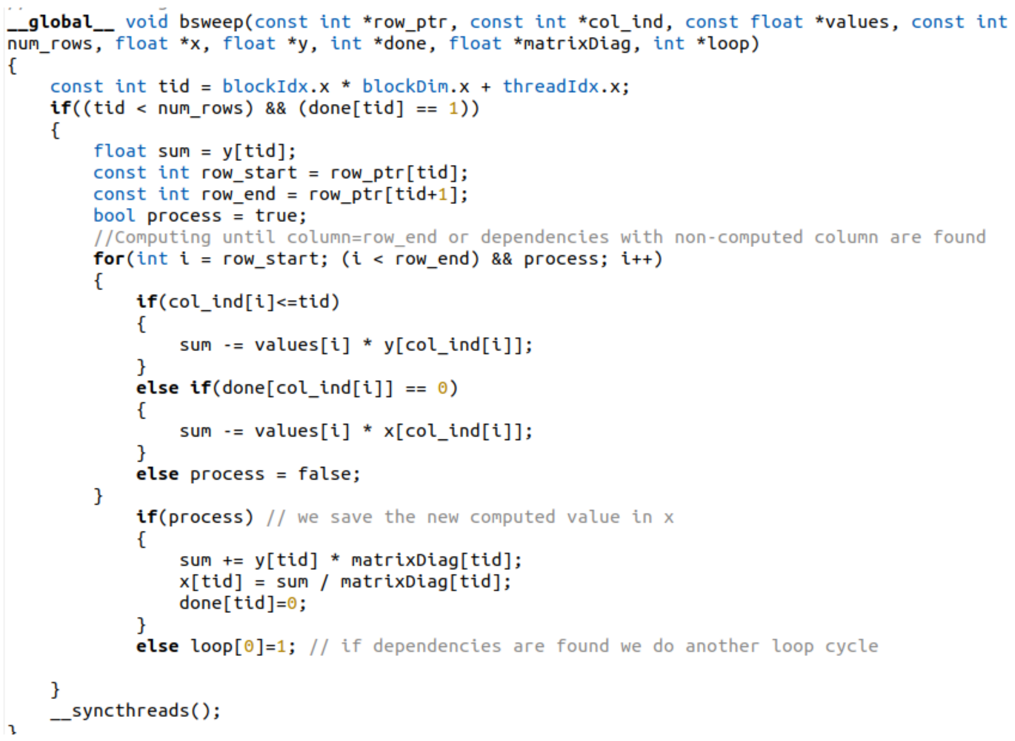
- For each row, it computes the new value of the solution variable based on the current values of the neighboring variables and the system coefficients.
- The forward sweep updates the solution variable values in a forward direction, moving from the known variables to the unknown variables.
- The purpose of the forward sweep is to update the solution approximation by incorporating information from the previously updated variables.



```
__global__ void fsweep(const int *row_ptr, const int *col_ind, const float *values, const int
num_rows, float *x, float *y, int *done, float *matrixDiag, int *loop)
{
    const int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if((tid < num_rows) && (done[tid] == 0)) //Execute only on the rows not computed until now
    {
        float sum = x[tid];
        const int row_start = row_ptr[tid];
        const int row_end = row_ptr[tid+1];
        bool process = true;
        // Computing until column=row_end or dependencies with non-computed column are found
        for(int i = row_start; i < row_end && process; i++)
        {
            if(col_ind[i]>=tid)
            {
                sum -= values[i] * x[col_ind[i]];
            }
            else if(done[col_ind[i]] == 1)
            {
                sum -= values[i] * y[col_ind[i]];
            }
            else process = false;
        }
            if(process) // we save the new computed value in y
            {
                sum += x[tid] * matrixDiag[tid];
                y[tid] = sum / matrixDiag[tid];
                done[tid]=1;
            }
            else loop[0]=1; // if dependencies are found we do another loop cycle
    }
    __syncthreads();
}
```

The second kernel (Backward Sweep) iterates through the rows of the system in reverse order, from the last row to the first row:

- For each row, it computes the new value of the solution variable based on the current values of the neighboring variables and the system coefficients.
- The backward sweep updates the solution variable values in a backward direction, moving from the known variables to the unknown variables.
- The purpose of the backward sweep is to further refine the solution approximation by incorporating information from the variables that were updated in the forward sweep.



```
__global__ void bsweep(const int *row_ptr, const int *col_ind, const float *values, const int
num_rows, float *x, float *y, int *done, float *matrixDiag, int *loop)
{
    const int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if((tid < num_rows) && (done[tid] == 1))
    {
        float sum = y[tid];
        const int row_start = row_ptr[tid];
        const int row_end = row_ptr[tid+1];
        bool process = true;
        //Computing until column=row_end or dependencies with non-computed column are found
        for(int i = row_start; (i < row_end) && process; i++)
        {
            if(col_ind[i]<=tid)
            {
                sum -= values[i] * y[col_ind[i]];
            }
            else if(done[col_ind[i]] == 0)
            {
                sum -= values[i] * x[col_ind[i]];
            }
            else process = false;
        }
            if(process) // we save the new computed value in x
            {
                sum += y[tid] * matrixDiag[tid];
                x[tid] = sum / matrixDiag[tid];
                done[tid]=0;
            }
            else loop[0]=1; // if dependencies are found we do another loop cycle

    }
    __syncthreads();
}
```

3 Efficiency of the Algorithm

Even though the dependencies between the elements of the matrix slow down the algorithm, parallelization allows us to save a significant amount of time for large matrices.

To test the algorithm, we used a trial matrix with the following characteristics:

- Number of rows: 51813503
- Number of non-zero values: 103565681
- Matrix type: Lower Triangular Matrix

The matrix is a lower triangular matrix, so in the backward sweep, no dependencies are found, and a single iteration does all the work.

3.1 Execution Time and Speed-up

The tests were performed on a machine equipped with an Intel Core i7-11800H CPU and an NVIDIA GeForce RTX 3060 Mobile GPU.

Out of 500 tests conducted:

- The average CPU runtime was 0.500061989 seconds.
- The average GPU runtime was 0.105113586 seconds.
- The speed-up obtained from using the GPU instead of the CPU was 4.75x.

Therefore, the time required for computation using the GPU is approximately one-fifth of the time needed by the CPU, resulting in a significant improvement in performance.

Due to the floating-point approximation performed by CUDA, the results obtained by the GPU may differ slightly from those obtained by the CPU. However, the error is less than 0.1%.