Implementation of Gauss-Seidel algorithm for sparse matrices on Nvidia GPU device

Fiorentini Riccardo

12-2022

Abstract

The algorithm of Gauss-Seidel is notoriously serial and trades on iteration to solve problems like linear systems of equations. In particular, the variant to implement has a backward part where the algorithm is the same but it starts from the last row instead of the first one, flipping all the dependencies. Many attempts to find a way to parallelize this algorithm use the Jacobi method as an approximation. Consequentially it needs more cycles to find a result of the same precision. The advantage of having sparse matrixes is that computing everything in sequence is no more necessary, it is possible to find independent groups of rows computable in parallel using algorithms based on a graph-coloring approach. These analyses, therefore, have a computational cost that we can not neglect. The aim is to create a lighter program that does not need preprocessing of the data and can do the independent operations in parallel taking the same input of the serial CPU implementation.

1 Introduction

1.1 Gauss-Seidel method

The Gauss-Seidel method, as we already said before, is an iterative technique to solve linear equations systems $Ax + b = 0$ and consist in:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^k), i = 1, 2, ..., n$$

where $x_i^{(k+1)}$ is the value of the vector "x" at the index "i" after "k+1" iteration of the algorithm. It is simple to see that the value at k+1 depends on the value at iteration k and this is where the dependencies arise in the algorithm.

```
for (int i = 0; i < num_rows; i++)
    {
        float sum = x[i];
        const int row_start = row_ptr[i];
        const int row_end = row_ptr[i + 1];
        float currentDiagonal = matrixDiagonal[i];
        for (int j = row_start; j < row_end; j++)
        {
            sum -= values[j] * x[col_ind[j]];
        }
        sum += x[i] * currentDiagonal;
        x[i] = sum / currentDiagonal;
    }
```

Listing 1: Serial C++ implementation

1.2 One kernel execution waiting dependencies resolution

The first version of the algorithm assigned a thread to each row of the matrix and computed all the computable rows. If a row had dependencies the system started a loop until the data it was waiting for was finally available and could continue the processing. The big deal with this implementation was the thread alternation on the cores and the fact that the constant concurrent access to common data led the kernel never to stop.

1.3 Layering the dependencies

Since there is no way to avoid dependencies, the hypothesis to create a kernel that checks if a row is independent and computes it when possible, leaving the others for the next cycle, became the most reasonable. This method does not have high latency caused by threads management and every cycle computes in parallel all the computable rows. In contrast, the threads assigned to computed rows or rows with dependencies terminate without loading the valid computations.

2 The algorithm

2.1 Data structures

Both CPU and GPU algorithms use the CSR (Compressed Sparse Row) format to represent the sparse matrix. Since the GPU has to execute the algorithm in parallel, it also needs a vector to save new results, avoiding the risk of overwriting useful data, and a vector to save which row has changed to know if the data in the new vector are valid or not.

```
//CSR for Gauss-Seidel algorithm:
int num_rows, num_vals;
//indexes of the firs nonzero element of each row plus num_vals as last element
int row_ptr[num_rows+1];
//contain in sequence the column index of each nonzero element row by row
int col_ind[num_vals];
//every nonzero value of the matrix ordered by row and column
float values[num_vals];
//all the values on the diagonal, usefull for the division part of the method
float matrixDiagonal[num_rows];
```

Listing 2: CSR implementation

2.2 Kernel

Since the kernel executes in parallel on each thread and each thread is assigned to a row, the kernel must control if that row is computable or has dependencies. To avoid doing the same cycle more times the control is done in the same cycle that computes the row. When a dependency it is found, the thread terminates without saving. That way it minimizes the number of cycle to do in a kernel call letting all the uncomputable rows leave the resources to the computable one.

2.3 Main iteration

In the main we only have to set the variables useful to check if we have computed all the rows using a variable, shared by all threads, which becomes false if at least one thread has not finished its computation. Before each iteration, it also must be guaranteed that all threads have terminated adding "cudaDeviceSynchronize()" after the kernel call.

```
const int row = blockIdx.x*blockDim.x + threadIdx.x;
if((row<num_rows) && modified[row]==0){
    float tmp = x[row];
    const int row_start = row_ptr[row];
    const int row_end = row_ptr[row+1];
    bool done = true;
    for(int col = row_start; (col < row_end) && done; col++){
        if(col_ind[col]>=row){
            tmp -= values[col]*x[col_ind[col]];
        }else if(modified[col_ind[col]]==1){
            tmp -= values[col]*y[col_ind[col]];
        }else if(modified[col_ind[col]]==0){
                done = false;
        }
    }
    if(done){
        tmp += x[row] * matrixDiagonal[row];
        y[row] = tmp / matrixDiagonal[row];
        modified[row] = 1;
    }else{
        finish[0] = 0;
    }
}
__syncthreads();
```

Listing 3: kernel implementation in CUDA

3 Sperimental efficiency analysis

This algorithm allows parallelizing at least all the independent rows. Now, if a thread terminates before another one tries to use its computed data, the second one can proceed without problems. It is possible to say that the number of iterations of the kernel are:

$$O(longestdependencybranch)$$

considering a graph representation of the dependencies.

3.1 Trial matrix analysis

The used matrix is a lower triangular matrix, which means that in the backward part of the algorithm there are no dependencies, it will cycle the backward kernel only once. Other interesting analyses of the matrix are forward dependencies and the worst-case number of cycles in terms of the number of kernel calls. The result obtained are:

- Maximum number of kernel iteration = 91;

- Average dependencies per row = 2.092111;
- Maximum number of dependencies = 36;
- Maximum number of lines solved at once (in the worst case) = 17890330;
- Minimum number of lines solved at once (in the worst case) = 1;

Considering the matrix as a square matrix with the number of columns and rows equal to 51813503 and the number of nonzero values equal to 103565681.

3.2 Average speedup on GPU

To test the effectual improvement of the algorithm once parallelized on the GPU I tested it 100 times (compiled with "nvcc program.cu -O3 -o program") setting the number of thread per block to 1024 and the block per grid to $\lceil num_rows/1024 \rceil$ obtaining the following data:

```
With 100 execution of the algorithm the data obtained are:
Average execution time of the GPU: 0.021251
Average execution time of the CPU: 0.666331
Average execution improovment: 31.355725
Average max relative error: 0.000423
```

Figure 1: Results on: GPU = GeForce GTX 1650 and CPU = Intel core i7-9750H

We can easily see the advantage of a GPU parallelization for this algorithm but, due to the floating point approximation done by the GPU, it is essential to consider that the result is not as precise as the one obtained with the CPU. In the end, it is possible to measure how many cycles are effectively executed by the GPU:

```
Average number of forward cycles needed: 64.110000
```

Figure 2: Results on: GPU = GeForce GTX 1650 and CPU = Intel core i7-9750H

3.2.1 Compilation instruction:

The program can be compiled thanks to the **CUDA toolkit** with the comand **"nvcc project.cu -O3 -o project"** and then execute it with **"./project nameMatrix.mtx"**, the executable and the matrix must be in the same folder.