SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

CUDA Implementation of SYMGS

Final project GPU101

Mattia Callegari, 957170

Advisor:
Ing. Beatrice Branchini

Academic year:
2022-2023

Abstract: The Symmetric Gauss Seidel algorithm is an iterative method used to solve a system of linear equations. The goal of the project was to achieve better performance by rewriting the CPU algorithm of SYMGS in CUDA. The new CUDA implementation takes advantage of the high number of threads available in a GPU to divide the workload. The result is a significantly faster execution, with a speedup of over 4x on the systems tested.

Key-words: CUDA, SYMGS, HPC

1. CPU Implementation

The element based formula that describes a forward pass of the Gauss Seidel Method is

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

The computation of $x^{(k+1)}$ uses the elements already calculated in the rows before. This is done in top-to-bottom for the forward pass and bottom-to-top for the backward pass. This dependency between rows is at the heart of the problem when it comes to implement it in parallel. The CPU code provided implements this algorithm for sparse matrices stored in CSR format.

```
//Forward Sweep
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;
}
```

Code 1: Forward sweep CPU

2. GPU Implementation

The implementation overcomes the problem of the dependencies between rows by using an auxiliary array that keeps track of which rows have completed the computation and have updated the corresponding value. An additional array is needed to store the new values without overwriting the old ones that may be needed. In the forward pass, a row can be computed if each value in the row either

1. is over the main diagonal
2. has the corresponding value in the solution array updated and ready to be used

If one of the two condition is met, the value is used in the sum for the row and the thread can continue its computation. If all values in a row meet the conditions, the row can be marked as done and the new solution array can be updated by subtracting the partial sum computed, as seen in the CPU implementation.
The auxiliary array also gets updated and the corresponding value for the rows sees its flag raised: now if other rows needs the value for their computation they can access it in the new copy of the array.
Otherwise if one of the values isn't ready, the computation for that row stops, as it needs a value not yet computed. The GPU kernel gets called over and over again until all threads have updated values and all rows get computed.

```
    int row_index = threadIdx.x + blockDim.x * blockIdx.x;
    if(row_index >= num_rows || updated[row_index]) return;

    float sum = old_x[row_index];
    int row_start = row_ptr[row_index];
    int row_stop = row_ptr[row_index + 1];
    float currDiag = matrixDiagonal[row_index];
    bool row_ready = true;

    for(int i = row_start;
        i < row_stop;
        i++)
    {
        if(!row_ready) break;                      // Row isn't ready, abort all row calcs
        if(col_ind[i] < 0) continue;               // Out of bound value to be handled
        if(col_ind[i] >= row_index)                // If the value is above the main diag, it has no dep
            sum -= values[i] * old_x[col_ind[i]];
        else if (updated[col_ind[i]])              // If dep has already been updated, use it
            sum -= values[i] * new_x[col_ind[i]];
        else
            row_ready = false;                     // Otherwise lower the row ready flag
    }
    if (row_ready)                                 // If row ready is raised after whole row, row has finished
    {
        sum += old_x[row_index] * currDiag;        // Remove diagonal contribution
        new_x[row_index] = sum / currDiag;         // Update x value
        updated[row_index] = true;                 // Raise the updated value of the row
    }
    else updated[num_rows] = false;                // Otherwise set done to false (new iteration is needed)
```

Code 2: Forward sweep device kernel

The number of iteration needed is strictly dependent on the form of the matrix and its sparsity rate. An upper triangular matrix has no dependencies and thus need only one iteration in the forward sweep.
A sparser matrix has a probability of dependencies between rows very low, thus needing a low number of iterations.
In order to check if all threads are done, an additional value at the end of the auxiliary array is reserved.
The last position value gets raised at each call and lowered if at least one of the threads stops.
The host code then checks if the value is still up (computation completed) or down (new iteration needed).

```
do
    {
        CHECK(cudaMemset(d_updated + num_rows, 1, sizeof(bool)));       // Set done -> 1
        parallel_symgs_fw<<<dimGrid, dimBlock>>>(...)    // Kernel call
        CHECK_KERNELCALL();
        CHECK(cudaMemcpy(&done, d_updated+num_rows, sizeof(bool), cudaMemcpyDefault));
    } while (!done);
```

Code 3: Host kernel caller

3. Results

All tests regarding the CUDA implementation have been performed on the following hardware

- CPU: Intel Core™ i5-6600K Processor @ 4.1GHz
- GPU: NVIDIA GeForce GTX 1660 SUPER

An important note has to be made about the input: the given matrix for testing purposes was lower triangular and thus needed only one iteration in the backward sweep (which took around 11 ms to complete). This is a favorable scenario for the GPU algorithm. Never the less, even if doubled the time for the forward pass the GPU performs the task in less than half the CPU time. The test conducted didn't account for the memory transfer to the device (which could be easily avoided by reading the input directly into the device memory). The fastest execution time was reached with 128 threads per block with an SM occupancy of 87%.
The CPU implementation has been tested on the newer and considerably faster Apple M1 chip: a significant speedup over the Intel i5 is achieved (over 20%).
The GPU implementation manages to beat both the i5 and the M1 chip. The execution time on the GPU is 4 to 5 times faster than on the CPU counterparts.

Threads per block	Execution time
64	0.1478598118
128	0.1093568802 s
256	0.1242611408 s
512	0.1266338825 s
1024	0.1224470139 s

Chip	Execution time
Intel i5	0.5828771591
Apple M1	0.4618160725

Table 1: GPU and CPU results

The main bottleneck of the application resides in the low memory throughput (starting at 30% for the first iteration and getting as low as 7%). Each thread works independently, not allowing the use of shared memory. More important, thread in the same warp may perform different branches of the kernel: one may stop blocked by a dependency while others may continue in the computation. This problem doesn't allow for better optimization: tools like shared memory and parallel reduction would strongly help performance as seen in algorithms like SpMV with similar memory access pattern, but are unusable due to the presence of dependencies.

4. Conclusions

The CUDA implementation finishes one iteration of the Symmetric Gauss-Seidel algorithm considerably faster on my personal machine. The measured improvement over the CPU implementation shows the power of parallelization, even for iterative algorithm with dependencies such as SYMGS. A key role in the performance is played by the sparsity of the input matrix: for denser matrices a different implementation may be needed.

4.1. Further work

The memory occupancy can be improved by removing the auxiliary array and checking directly if the new array has already been updated. A special value is needed and some complications arise in the code: the memory gains are small compared to the storing of the whole input matrix making it a not worthwile endeavor. Possible further optimization may be achieved analyzing the profiling data: a possible lead could be working on the memory access pattern of the matrix. A different format for storing sparse matrices (e.g CSC, ELLPACK, ...) could achieve better results.