PARALLELIZATION OF THE SPMV ALGORITHM USING CUDA C++

GPU 101 PiA project report

Luca Guffanti 955414

Abstract

The SPMV (SParse Matrix-Vector multiplication) algorithm is an essential kernel used in many areas of scientific computing: from numerical analysis, to algorithms on graphs, to solving sparse linear systems of equations. The introduction of parallelized high performance computing architectures, such as GPUs, offers us a new way to improve the performance of the algorithm, as we can exploit these innovative architectures' full potential.

1 Introduction

Given a sparse square matrix and a dense column vector the objective of the project was to enhance the performance of the computation using a NVIDIA GPU, programming it with the CUDA C++ programming language. The solution had to consist of a parallel version of the SPMV algorithm that, run on a GPU, would enhance the performance of the algorithm if compared with its sequential equivalent, run on a CPU.

2 Analysis of the workload

2.1 Input matrix

I was given a 51813503×51813503 sparse matrix, containing 103565681 values (Sparsity ≈ 1) and a 51813503× 1 column vector that was randomly generated at run-time. The expected result was a 51813503×1 column vector containing the result of the multiplication.

For reasons of memory occupation, the matrix was given in the CSR (Compressed Sparse Row) format: reading the matrix yielded three (3) vectors.

`values`: array containing the actual elements of the matrix.

`row_ptr`: array containing the index of the first element (found in `values`) of every row.

`col_ind`: array containing the column index of the correspondent element in `values`.

The matrix, therefore, couldn't be accessed in the usual way, and the implementation of the algorithm reflected the choice of format.

The Compressed Sparse Row format is one of many other compressed descriptions of sparse matrices. These formats are extremely useful because they enable users not to care about elements that are equal to 0, as the information they contain regards non-zero elements of the matrices and their coordinates.

2.2 Code to be parallelized

I was directly given the following sequential code used for calculating the product on CPU. This code wasn't just parallelized and left unused, as it was used to calculate the product, against which the behaviour of the parallel code was checked.

```
void spmv_csr_sw(const int *row_ptr, const int *col_ind,
                 const float *values, const int num_rows,
                 const float *x, float *y) {

    for (int i = 0; i < num_rows; i++) {
        float dotProduct = 0;
        const int row_start = row_ptr[i];
        const int row_end = row_ptr[i + 1];

        for (int j = row_start; j < row_end; j++) {
            dotProduct += values[j] * x[col_ind[j]];
        }

        y[i] = dotProduct;
    }
}
```

It's possible to see that the sequential implementation of the SPMV simply consists in an iterated dot product calculated between every row of the matrix and the column vector, `x`.

3 Pursuing parallel code

The parallelization was obtained trying to exploit the parallel architecture at its maximum: the threads were indirectly and uniquely associated to sets of rows, where every row satisfied the following conditions.

A thread with ID `single_thread_id` was associated to a row with index `row_index` if and only if

- `row_index` was less than the number of rows.
- `row_index` $=$ `single_thread_id` $+$ `i` $\cdot$ `num_threads`, `i` $\in N$

Every thread iterated through every row it was associated to (the row whose index was the thread ID and all the rows whose indices were equal to the thread ID plus a multiple of the number of threads) and computed the correspondent scalar product, storing the result in the `y` array.

The implemented kernel expects the following parameters.

`const int *row_ptr`: as said, it contains the index (in `values`) of the first element of every row.

`const int *col_ind`: as said, it contains the column index corresponding to a given value.

`const float *values`: as said, it contains the values of the matrix.

`const int num_rows`: number of rows.

`const float *d_x`: column vector that multiplies the sparse matrix.

`float *y`: result vector. It contains all the scalar products.

`const int num_threads`: number of active threads.

3.1 Code of the CUDA kernel

```
__global__ void spmv_csr_gpu(const int *row_ptr, const int *col_ind,
                             const float *values, const int num_rows,
                             const float *d_x, float *y,
                             const int num_threads) {

    int single_thread_id = threadIdx.x + blockIdx.x * blockDim.x;
```

```
    for(int index = single_thread_id; index < num_rows; index = index + num_threads) {

        float dot_product = 0;
        int row_start = row_ptr[index];
        int row_end = row_ptr[index+1];

        for(int j = row_start; j < row_end; j++) {
            dot_product += values[j] * d_x[col_ind[j]];
        }
        y[index] = dot_product;
    }
}
```

4 Results

4.1 A look a the code

The GPU and CPU code are very similar, as the parallelization isn't obtained by deeply modifying the given code, but by evenly distributing the workload amongst all threads. The nature of the executed operations doesn't change as every thread computes products and sums just like how the CPU did (at least from an abstract point of view), but it's the use of different multiprocessor that are capable of running concurrently hundreds of threads, all executing the same code with different data. that guarantees the improvement of the performances. The GPU code, parametric in the number of active threads, can in fact behave like the CPU code if the number of threads running is forced to be 1. Please note that threads are executed in warps (groups of 32 threads), and trying to execute one thread actually causes a single warp to be executed.

4.2 Quantitative results

The code was tested on a system with a NVIDIA GTX 1650 GPU with 4 GB of RAM and a Intel core i7-9750H CPU. In all the tests the matrix was the same and the number of threads running increased.

ID	THREADS	CPU TIME [s]	GPU TIME [s]	IMPROVEMENT
1	256	0.7268	0.4294	71%
2	512	0.7070	0.2404	66%
3	1024	0.7742	0.1605	79%
4	2048	0.7588	0.1213	84%
5	4096	1.006	0.1658	84%

Table 1: Results of the tests

The analysis of the results of the tests confirms that the parallelization of the algorithm was effective: I registered the maximum improvement in performance with 2048 and 4096 threads running and, considering the decreasing tendency of the GPU times, it's possible to induce that, until a lower bound is met, the times will probably continue to decrease, as the number of threads increases.

4.3 A note on the results

Due to differences in the hardware implementation of the multiplication operation in the CPU and in the GPU, the result array calculated on the GPU is actually just a very precise approximation of the actual result obtained with the CPU. `y` contains values that don't exactly match the values in the result array calculated on the CPU. I defined an acceptability threshold (0.003) within which differences in the values can be considered caused by the differences in architectures computing floating point operations.

5 Other possible implementations

The implementation is effective as it reaches the wanted enhancement of performance but, nonetheless, there may be other different techniques that further enhance the algorithm, that were not implemented in the scope of this project.

- A first enhancement may consist in the calculation of the products in a row done by a single thread, with a subsequent parallel reduction of every single row.
- A second enhancement regards the assignment of entire **warps of threads** to the rows. In this way it may be possible to exploit the small number of values per row to achieve a parallelization regarding both the calculation of the single partial products and the reduction of the row into its sum, obtaining even shorter times.

6 Compiling & Running the code

In order to compile and run CUDA code (`.cu` extension) it's necessary to have access to a NVIDIA GPU and to have installed the CUDA toolkit found at the following link `https://developer.nvidia.com/cuda-downloads`. You can find the test matrix at the following link
`https://tinyurl.com/gpumatrix`.

Once the system is correctly setup clone this repository and in a Linux terminal, within the scope of the directory, run

```
$ make
```

After the compilation is completed you can run the code with the following command

```
$ ./spmv/spmv-gpu path_to_test_matrix num_of_threads
```

7 Conclusion

In conclusion it's possible to improve the SPMV by means of parallelization: even a simple modification to the code shows how powerful these high performance architectures can be. It's important to notice, however, as explained in Section 4.3, that differences in hardware architectures used for computing certain operations may give divergent results. This property, if left uncontrolled, may cause some problems.