POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Project Report

PROJECT REPORT FOR
COMPUTER SCIENCE ENGINEERING - HIGH PERFORMANCE COMPUTING: GRAPH AND DATA ANALYTICS

Daniele Francesco Antonio Ferrazzo, 10608810
Francesco Fadini, 10573718

Academic year:
2021-2022

Abstract: Given as input a graph G and a vertex belonging to G, we are asked to compute the top-20 highest ranked vertices according to the Personalized PageRank (PPR) algorithm. In particular, the goal of this project is to accelerate the algorithm using a GPU and making the implementation as fast as possible. The requirement for a successful result is to find at least 16 of the correct top-20 vertices, even if in the wrong order.

Key-words: Personalized PageRank, GPU, top-k, algorithm, acceleration

1. Introduction

Computing graph algorithms for graphs with millions of nodes and edges on a CPU does not scale well and the time required to execute those algorithms is simply too long. A possible way to overcome this limitation is to exploit the intrinsic parallelization paradigm of a GPU's architecture to improve the performance of a graph algorithm by drastically reducing the time required for its execution.
Our problem consists in, given a large graph and one of its vertices, computing the top-20 highest ranked vertices resulting by running the Personalized PageRank (PPR) algorithm on the given vertex. The goal is to solve this problem with at least 80% accuracy and in the least time possible. In particular, it means we are required to retrieve at least 16 of the correct top-20 vertices, even if they are in the wrong order. The speed of our code is the main evaluation criteria.
To test our implementation, we will be using the Gleich/wikipedia-20070206 dataset, which consists of more than 3.5 million vertices and 45 million edges, and a GTX 1070 GPU with Nvidia architecture.

2. Implementation

We decided to approach the problem by starting from the CPU code in the `personalized_pagerank.cuh` file, in particular in the `personalized_pagerank_cpu` function. Our solution takes inspiration from this code, which solves the top-20 PPR problem using the CPU only, and parallelizes the corresponding functions on GPU to obtain better performances.
The core of our solution is coded in the `personalized_pagerank.cu` file and is structured in an incremental approach. We propose a total of 4 implementations, all of which correctly solve the problem, and each implementation aims at reducing the GPU execution time of the previous by possibly modifying it and stacking additional features on top it.

In the remainder of this section we will illustrate in detail each implementation, what we changed from the previous one and the reasons that brought us to do so.

2.1. Implementation 0

The first implementation is the most basic of the four and is coded in the `PersonalizedPageRank::ppr_0` function. The computation of the PPR makes use of four kernels (see below) which perform the same operations of the corresponding CPU functions, but in a parallelized way.
The kernels used are the following:

- `spmv_coo_gpu` – handles the computation of the matrix-vector product with a matrix in COO format. The COO format enables us to have a high coalesced access to the data, the data is decomposed so as to have that each thread computes the intermediate product for a non-zero element of the matrix. The intermediate results are then added with atomic operations in order to ensure consistency of the data and avoid race conditions.
- `dot_product_gpu` – handles the computation of the `dangling_factor` through the dot product between the PageRank values of the previous iteration and the dangling bitmap (1 if vertex is dangling, 0 otherwise). The data is decomposed in order to have that each thread computes the product between two elements from the two vectors, then each thread uses an atomic operation to sum up the final result.
- `axpb_personalized_gpu` – handles the computation of the PageRank vector for the current iteration. The data is decomposed in order to have that each thread computes the value of the PageRank for a vertex of the input matrix.
- `euclidean_distance_gpu` – handles the computation of the euclidean distance between the current iteration of the PageRank vector and the previous iteration of the PageRank vector. The data is decomposed in order to have that each thread computes the intermediate distance between two elements of the two vectors, then each thread uses an atomic operation to sum up the final result.

All of the previous kernel uses grid-stride loops to be more flexible, scalable and debuggable. Even with this simple implementation, which makes use of basic GPU parallelization patterns, we are able to improve performances by 10x with respect to the CPU code.

2.2. Implementation 1

This implementation is coded in the `PersonalizedPageRank::ppr_1` function.
In the previous implementation we are not making use of the shared memory available at block level and thus not taking advantage of the faster accesses this memory offers with respect to GPU global memory. Moreover in the `dot_product_gpu` and `euclidean_distance_gpu` kernels we are computing the final result by means of the `atomicAdd` function, which degrades performance as it is synchronized for the whole GPU.
For this reason, we improved the two above mentioned kernels by incorporating the use of shared memory, in combination with a reduction technique. The modified kernels are described below, while the rest of the implementation is unchanged with respect to the previous.

- `dot_product_gpu_with_reduction` – the main difference from the previous kernel implementation is that the computation is divided into multiple steps:
  1. each thread computes the product between two elements from the two vectors and save the result in a shared memory array
  2. a parallel reduction over the shared memory array for each block is computed
  3. the first thread of each block uses atomic operations to sum up the intermediate parallel reduction result to obtain the final result
- `euclidean_distance_gpu_with_reduction` – same structure as above

With these modifications we are able to achieve 1.5x and 15x performance increase on the previous and CPU implementations respectively.

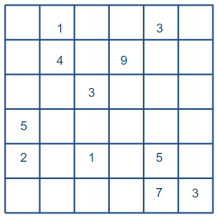
2.3. Implementation 2

This implementation is coded in the `PersonalizedPageRank::ppr_2` function.
By utilizing the CUDA command line profiling tool `nvprof`, we noticed that 90% of GPU execution time was taken up by the single kernel `spmv_coo_gpu`. Therefore, we focused on ways to more efficiently compute the

sparse matrix - vector product, also consulting the best approaches presented in literature. Our choice was the solution presented by Hoang-Vu Dang and Bertil Schmidt in their paper [1], which proposes a format for sparse matrix representation called Sliced COO and an efficient algorithm to compute SpMV on a GPU.

2.3.1. The Sliced COO Format

The main idea of the Sliced COO format is to divide a matrix into slices of consecutive rows to improve performances when computing the product between a sparse matrix and a vector. Similarly to COO, the Sliced COO format stores, for non zero elements of the matrix, the column indices, row indices and values for the corresponding row-column pairs. In addition, it stores the indices corresponding to the index of the first row of each slice of the matrix. Then, differently from COO, it orders column indices within each slice, so that entries of the same column are stored contiguously thus improving memory access performances.



SCOO
c_index = {1, 1, 2, 3, 4, 0, 0, 2, 4, 4, 5}
r_index = {0, 1, 2, 1, 0, 3, 4, 4, 4, 5, 5}
value = {1, 4, 5, 9, 3, 5, 2, 1, 5, 7, 3}
index = {0, 5}

COO
c_index = {1, 4, 1, 3, 2, 0, 0, 2, 4, 4, 5}
r_index = {0, 0, 1, 1, 2, 3, 4, 4, 4, 5, 5}
value = {1, 3, 4, 9, 3, 5, 2, 1, 5, 7, 3}

Figure 1: Comparison of SCOO with slice size = 3 and COO

To be able to work with an SCOO formatted sparse matrix, we implemented a dedicated function called `coo_to_scoo` [2] which performs the conversion from COO format to SCOO. The workload of this conversion is allocated outside of the `PersonalizedPageRank::execute` scope, so as to not consider it in the total GPU execution time.

2.3.2. SpMV SCOO Kernel

The only difference with respect to the previous implementation is in the following kernel: `spmv_scoo_gpu`.
This kernel implementation is based on the proposed version by Hoang-Vu Dang and Bertil Schmidt[1]. It exploits the main benefits of the SCOO format: more coalesced access to the input vector, thanks to the non-decreasing order of the column indexes, and the use of shared memory to store intermediate results of the product, thanks to the division of the input matrix into multiple slices. We launch as many blocks as the number of slices and assign one block to work on one slice, for each block we have an amount of shared memory equal to the slice_size * lane_size.
We can divide the overall computation in four main steps:

1. each thread is assigned to a lane, lane are used to reduce the possibility of serialization between threads when using atomic operation
2. each thread saves the partial value of the product computed for the assigned row in the corresponding shared memory lane
3. a parallel reduction is performed over each lane to sum up all the intermediate values per row
4. the final result per row is stored in the output array

Performance improvements result in a 1.7x speedup on the previous implementation and a 26x speedup on the CPU implementation.

2.4. Implementation 3

This implementation is coded in the `PersonalizedPageRank::ppr_3` function.
After having explored various data parallelization improvements in the previous implementations, for our fourth and last one we shifted our attention to task parallelization by extending the previous implementation with CUDA Streams. As a first step we switched from using `cudaMemset` to `cudaMemsetAsync` for device parameter initialization and in particular we decided to execute each call in a dedicated Stream. Moreover, since no data

dependency is present between the two kernels to compute the sparse matrix vector product and the dot product, we decided to execute them in two different Streams to try to exploit as much task parallelism as possible. Finally, in addition to the previous implementation, we slightly modified the kernel for the computation of the dot product and introduced an additional utility kernel, as described below:

- `dot_product_gpu_with_global_reduction` - the main difference from the previous kernel implementation is that the final step of the reduction is no longer performed through the use of an atomic operation. Instead, the reduction results of each block are saved in a global memory array.
- `reduce_global` - this kernel takes as input the output of the `dot_product_gpu_with_global_reduction` kernel and performs a reduction over the global memory array producing as output the final result of the dot product.

No evident performance improvement was obtained on the previous implementation. With the help of NVIDIA's Visual Profiler tool, we noticed that the overlap between kernels in different Streams was minimum, probably due to the lack of available GPU resources.

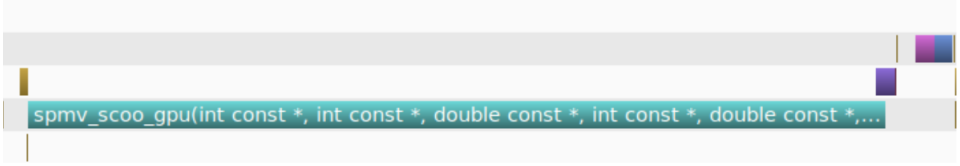


Figure 2: NVIDIA's Visual Profiler output

3. Results

In this section we will collect the results obtained by running our four implementations on the Gleich/wikipedia-20070206 dataset, and how they compare against the CPU implementation. All the following results are obtained with a NVIDIA GTX 1070 GPU with compute capability 6.1.

3.1. Results With Default Parameters

In the table below are summarized the comparisons between the GPU and the CPU execution times, obtained with 100 iterations, block size 512, grid size computed at runtime and the remaining parameters leaved as default.

	Avg Exec. Time	**Accuracy**[1]	**Speed Up**[2]
CPU	12000 ms	100%	
ppr_0	1204 ms	99.95%	10x
ppr_1	812 ms	99.95%	15x
ppr_2	472 ms	99.95%	26x
ppr_3	465 ms	99.95%	26x

We can see that already from **Implementation 0**, which makes use of basic GPU parallelization patterns the performance increase on the CPU implementation is an order of magnitude higher.
By introducing a reduction technique in **Implementation 1**, we were then able to further reduce execution time down by 1.5x against Implementation 0 and 15x against the CPU implementation.
Next, in **Implementation 2** we aimed at accelerating the `spmv_coo_gpu` kernel, which was by far the slowest component of the code. With the introduction of the SCOO format and a customized kernel for computing the sparse matrix vector product, we pushed GPU execution time down to 472ms, which resulted in a 1.7x improvement on Implementation 2 and a 26x speedup on the CPU implementation.
Finally, in **Implementation 3**, we tried adding task parallelization and a global reduction mechanism. However, we were not able to obtain notable improvements and GPU execution time was essentially unchanged with

[1] Average accuracy on 100 iterations
[2] SpeedUp compared to the CPU execution

respect to the previous implementation.

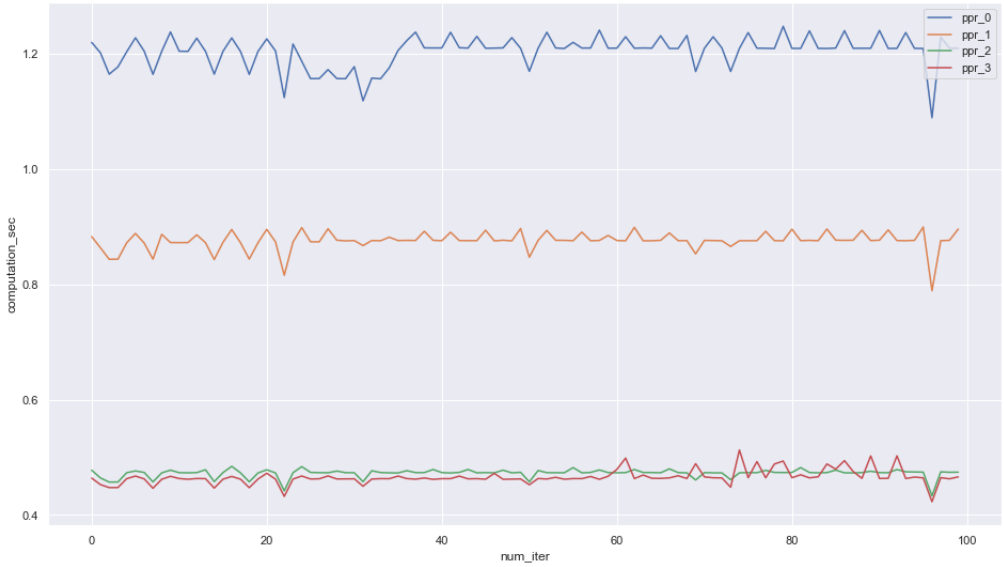


Figure 3: Results over 100 iterations of the four implementations

In Figure 3 are displayed the execution times of each implementation over 100 iterations using the parameter configuration mentioned above. It is clear to see that there is an incremental reduction in execution times until Implementation 2. Implementation 3 is then not able to further improve performance.

3.2. Results Without Default Parameters

In the table below we show the comparison between the CPU implementation, Implementation 3 with parameter configuration described in the section 3.1 and Implementation 3 with tuning of the convergence threshold parameter, which regulates when the algorithm is considered to have reached convergence on the newly computed PPR values. The result shown here is the one obtained with a convergence threshold of 10^{-3} and is our best result with a mean accuracy of 98% and a total speedup on the CPU implementation of 98x.

	Avg Exec. Time	**Accuracy**[1]	**Speed Up**[2]
CPU	12000 ms	100%	
ppr_3	465 ms	99.95%	26x
ppr_3_tuned	122 ms	98.19%	98x

References

[1] Hoang-Vu Dang and Bertil Schmidt. Cuda-enabled sparse matrix–vector multiplication on gpus using atomic operations. *Parallel Computing*, 39(11):737–750, 2013.

[2] Fitriyani Fitriyani. Sliced coordinate list implementation analysis on sparse matrix-vector multiplication using compute unified device architecture. *International Journal on Information and Communication Technology (IJoICT)*, 2:13–22.

[1] Average accuracy on 100 iterations
[2] SpeedUp compared to the CPU execution