POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Personalized Page Rank GPU acceleration

Report document
Computer Science and Engineering - Ingegneria Informatica

Ottavia Belotti, 10657411
Riccardo Andrea Izzo, 10599996

Advisor:
Guido Walter Di Donato

Co-advisor:
Prof. M. D. Santambrogio

Academic year:
2021-2022

Abstract: The goal of the project is to implement and adapt the notorious link analysis algorithm PageRank for a high parallelized architecture such as the one of graphics processing units (GPU) to obtain a substantial speed-up in execution over very large (millions of nodes) and sparse graphs. This report includes a brief introduction to the PageRank algorithm itself and how it has been handled in its CUDA transposition. We present the format of choice to store the graph and some tweaks took in place to speed up even further the execution on GPU. Lastly, in Section 3 we provide some results of the tests carried out to assess the better performance of the final product.

Key-words: PageRank, GPU, CUDA, Sparse Graph

1. Introduction

The standard formulation of Personalized Page Rank algorithm [2] consists in computing the page rank score for each page step-by-step, with the updating dynamic rule in Equation (1).

$$\mathbf{p_{t+1}} = d\mathbf{X}\mathbf{p_t} + \frac{d}{|V|}(\bar{\mathbf{d}}\mathbf{p_t}) + (1-d)\bar{\mathbf{v}} \tag{1}$$

Equation (1) takes as parameters:

- d: damping factor (also called *alpha*)
- $\mathbf{X}$: adjacency matrix representing the links between pages
- $\mathbf{p_t}$: vector of PageRank scores at the previous step
- $|V|$: number of vertices in the graph
- $\bar{\mathbf{d}}$: dangling vector (1 if the vertex is dangling)
- $\bar{\mathbf{v}}$: personalization vertex vector (1 if vertex is personalization one)

Starting from Equation (1), the same computation has been achieved on GPU, dividing the formula into mainly three sub-components, four if considering the convergence check of the formula. Each component (i.e. GPU kernel) is described in detail in Section 2.3, but their tasks are cited here too, in order of execution:

1. Basic update of the PageRank score: $\mathbf{X}\mathbf{p_t}$
2. Dangling contribution: $\bar{\mathbf{d}}\mathbf{p_t}$
3. Final $\mathbf{p_{t+1}}$ with personalization vertex and damping contribution
4. Convergence check: stop the iteration of the formula because, on average, the PageRank scores have converged to their asymptotic values

The CUDA transposed algorithm for GPU computation proposed in the following sections kicks off from the above formulation of the PageRank.

2. Optimizations

In this section, all the optimizations implemented in the final code are presented. Some of these are more technical choices related to the particular working paradigm of CUDA language and GPU's architecture, while the last one (Section 2.5) consists in a trick to speed up the convergence of the problem.

2.1. Matrix storage format

The **Coordinate Format (COO)** is the format of choice for the algorithm thanks to its speed in accessing its elements. Since Personalized PageRank is supposed to deal with a problem which is naturally represented by a sparse graph, COO is relatively compressed as a format (i.e. better than a plain adjacency matrix), yet still rich enough to embed the absence of an underlying pattern in the graph (e.g. Diagonal format). The choice of COO over more compressed formats such as CSR and other more complex representations has been dictated by the fact that the former is more suited for the kind of access pattern that high parallelized GPU kernels have to sustain, especially for the SpMV computation [1]. For instance, while taking up less space, accessing elements stored in a CSR matrix would cause threads within a block to diverge and divergence over such a heavy task is bound to hinder the overall performance of the whole kernel call.

2.2. Computational precision

The starting CPU algorithm provides data structures dedicated to store the intermediate and final scores with double precision. However, such a precision has been deemed unnecessary via empirical validation. So the switch from double precision to single precision inside the kernels has resulted in faster computations, namely on multiplications and sums.
On a side note, in some cases half precision has been tried out, unfortunately with poor results in the score accuracy since the type is not rich enough for the kind of values that have to be deal with, hence its usage has been abandoned.

2.3. Kernel organization and CUDA streams

The execution consists of a loop computing the four basic steps to reach the updated PageRank scores. The loop stops whenever the algorithm converges to the true PageRank scores or it reaches the maximum allowed iterations (e.g. 30 iterations). Here are presented the kernels computing the tasks and their order of execution is shown in Figure 1:

- `spmv_coo`: handle the matrix-vector multiplication with a matrix in COO format, it uses a grid-stride loop to perform the dot product. It runs in parallel with `compute_dangling_factor_gpu` on the `spmv_stream` stream.
- `compute_dangling_factor_gpu`: compute the dangling factor, it makes use of shared memory and parallel reduction. It runs in parallel with `spmv_coo` kernel on the `dangling_factor_stream` stream.
- `axpb_personalized_gpu`: finalize the computation of the page rank vector by considering the damping factor and the dangling vector as reported in Figure 1.
- `euclidean_distance_gpu`: compute the euclidean distance between the previous page rank vector and the current one, useful to check when to stop the computation because the result is converging.

CUDA streams allow us to enable task parallelism (TLP) and not only data parallelism (DLP). In this case the matrix-vector multiplication and the computation of the dangling factor are independent tasks with no data dependencies and so they can be parallelized.

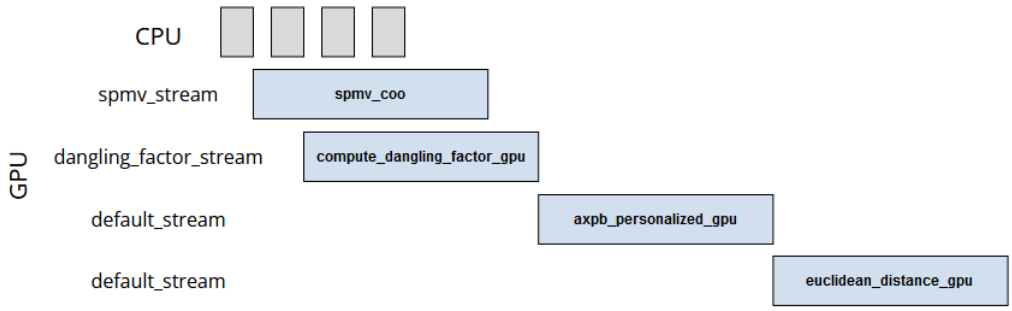

Figure 1: Kernel execution model

2.4. Parallel reduction

Both `compute_dangling_factor_gpu` and `euclidean_distance_gpu` kernels take advantage of the parallel reduction, an important data parallel primitive in CUDA. The version implemented use sequential addressing with coalesced memory accesses, this allows us to avoid divergent branching and bank conflicts. Finally to handle the fact that the number of threads decreases during the execution the last warp is unrolled with the function `warp_reduce()`.

2.5. Non-uniform PageRank score vector initialization

Given the entity of the problem, it's pretty clear how all the pages that are not linked by any other one are never going to end up in top-20, since the score of each page mainly builds up on either a great quantity of "pointers" or fewer "quality pointers"[1]. Having no kind of pointers (i.e. vertices with in-degree = 0) will results in a PageRank score which might as well be zero. However, the initial condition of the PageRank score vector $\mathbf{p_0}$ is canonically the following:

$$p_0(v) = \frac{1}{|V|} \qquad \forall v \in V \tag{2}$$

It can be seen that $\mathbf{p_0}$ is a stochastic vector, in fact $\sum_{i \in V} p_t(i) = 1 \ \forall t$. Doing an initialization like the one proposed in Equation (2) is fair to all the pages, but it doesn't take into account the knowledge that we have on how low in the ranking certain pages will end up for sure. So we propose another initialization based on a *penalty weight* α. In the final algorithm we tuned $\alpha = 0.1$. From Equation (3), $\mathbf{p_0}$ is still a stochastic vector.

$$\begin{aligned} \alpha &= 0.1 \\ \text{effective\_vertices} &= \{v \in V | \ \text{in\_degree}(v) > 0\} \\ \text{weighted\_sum} &= |\text{effective\_vertices}| \cdot (1-\alpha) + (|V| - |\text{effective\_vertices}|) \cdot \alpha \\ p_0(v) &= \begin{cases} \alpha \cdot \frac{1}{\text{weighted\_sum}} & if \ v \notin \text{effective\_vertices} \\ (1-\alpha) \cdot \frac{1}{\text{weighted\_sum}} & otherwise \end{cases} \end{aligned} \tag{3}$$

The new initialization method doesn't result in a drastic change on the overall performance, but once in a while, it makes so that the algorithm converges a couple of iterations before the implementation with canonical initialization of PageRank score vector. In fact, given the same personalization vertex and running the Personalized PageRank for 100 times, it can be seen in Table 2 that, without this initialization (Equation (3)), the overall number of iterations needed for the algorithm to converge is slightly more.

3. Results and Performance

The main metric used to evaluate the performance of the code is execution speed. Hence the code has been tested on two architectures:

- NVIDIA GeForce GTX 1050, 6.1 compute capability
- NVIDIA GeForce GTX 1080, 6.1 compute capability

[1] We define "pointer" a page that links another page in the graph. Similarly, we refer to a *very important* page (with a high PageRank score) linking another page as a "quality pointer".

Table 1 presents some statistic data collected during testing on the graph `wikipedia-20070206`[2]. The benchmark consisted in running 100 times the algorithms. For each run, the accuracy of the GPU retrieved top-20 pages is checked upon the provided CPU code. Hence the data in the table is to be considered as "averaged" over the 100 runs. Statistic data don't take into consideration the first 3 runs of the algorithm, called the "warm-up" time for GPU to obtain more fair and asymptotically realistic results. Furthermore, a starting personalization vertex is randomly chosen at each run.
From now on, "GPU-basic" and "GPU-optimized" respectively refer to a plain CPU-to-GPU CUDA transposition of the computation and the final optimized version of the GPU code built up from GPU-basic with all the adjustments presented in Section 2.

	Initialization	Avg. Reset	Avg. Execution	Mean accuracy	Speed-up[3]
CPU	-	-	19662.2 ms	100%	–
GPU-basic	170 ms	30.715 ms	2171.76 ms	99.95%	x9.05
GPU-opt.	172 ms	31.629 ms	1865.55 ms	99.95%	x10.54

Table 1: Performance over 100 runs of each implementation on GTX 1050 – 256 threads per block

All tests are performed with the basic configuration that includes:

- Convergence threshold: 10^{-6}
- Max iteration cap: 30
- Damping factor: $\alpha = 0.85$

However, we tested that lowering the precision with the convergence threshold down to 10^{-4} still achieves great accuracy (i.e. mean accuracy 99.75% over 100 runs) and a drastic speed-up of the execution (i.e. x19.52 speed-up with respect to CPU execution time).
Table 2 shows the bandwidth used by each kernel in the final implementation

	Avg. Execution	Avg. #iterations	Speed-up[4]
GPU-basic	2171.76 ms	29.85	–
GPU-optimized	1865.55 ms	29.21	x1.16

Table 2: Average bandwidth usage and speed-up on GTX 1050 – 256 threads per block

4. Conclusions

In this work, we have proposed a parallel implementation of the Personalized PageRank algorithm using CUDA programming language in order to accelerate the computation. We have also implemented a non-uniform vector initialization that penalize the spam web pages giving more importance to the ones strongly linked. We performed experiments on CPU and GPU with different datasets to examine the speed up. The outcomes presented in the tables in Section 3 show that the speed of computing PageRank scores with the proposed approach gives a considerably greater performance with respect to the CPU based algorithm.

[2] `https://sparse.tamu.edu/Gleich/wikipedia-20070206`
[3] Speed-up with respect to CPU execution time
[4] Speed-up with respect to GPU_basic execution time

References

[1] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. pages 3–5.

[2] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine.