Multithreading Google's PageRank with CUDA

Adithya Swaroop

Course: CS6023 GPU Programming

Professor: RUPESH NASRE

May 8, 2021

Abstract

We use the Internet to search for information on a daily basis. We use search engines to find out the useful information from the vast Internet. This is possible because the search engines are using a heuristic called PageRank. It is nothing but a value to a web page that is assigned by a PageRanking algorithm. This algorithm scans all possible webpages and then calculates the rank accordingly given by a formula. Search engines show results according these ranks, which stand for the popularity of the page. The lower is the rank, more popular the page is. Traditional approaches use multi-CPU architecture and this is not a very good choice due to the communication overhead and the low processing power of CPU compared to GPU. Hence, designing a PageRanking algorithm efficiently modified for parallel GPU-CPU environment that achieves higher accuracy and consumes lesser time to evaluate the PageRank exact rank vector even for large-scale webgraphs.

Contents

1 Introduction

When a search is made on the Internet using a search engine, there is first a traditional text processing part, where the aim is to find all the Web pages containing the words of the query. Due to the massive size of the Web, the number of hits is likely to be much too large to be of use. Therefore, some measure of quality is needed to filter out pages that are assumed to be less interesting. When one uses a Web search engine it is typical that the search phrase is underspecified.

Obviously Google uses an algorithm for ranking all the Web pages that agrees rather well with a common-sense quality measure. Somewhat surprisingly, the ranking procedure is based not on human judgment but on the link structure of the Web. Loosely speaking, Google assigns a high rank to a Web page if it has in-links from other pages that have a high rank. We will see that this self-referencing statement can be formulated mathematically as an eigenvalue equation for a certain matrix.

2 Web as a Directed Graph

Let all Web pages be ordered from 1 to n, and let i be a particular Web page. Then O_i will denote the set of pages that i is linked to, the outlinks. The number of outlinks is denoted $N_i = |O_i|$. The set of inlinks, denoted I_i, are the pages that have an outlink to i.

In general, a page i can be considered as more important the more inlinks it has. However, a ranking system based only on the number of inlinks is easy to manipulate. When you design a Web page i that (e.g., for commercial reasons) you would like to be seen by as many users as possible, you could simply create a large number of informationless and unimportant pages that have outlinks to i. To discourage this, one defines the rank of i so that if a highly ranked page j has an outlink to i, this adds to the importance of i in the following way: the rank of page i is a weighted sum of the ranks of the pages that have outlinks to i. The weighting is such that the rank of a page j is divided evenly among its outlinks. Translating this into mathematics, we get

$$r_i = \sum_{j \in I_i} \frac{r_j}{N_j}$$

This preliminary definition is recursive, so pageranks cannot be computed directly. Instead a fixed-point iteration might be used. Guess an initial ranking vector r^0. Then iterate

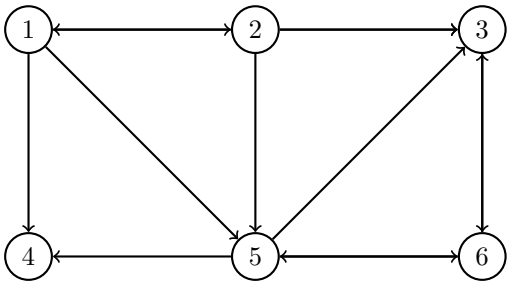
$$r_i^{(k+1)} = \sum_{j \in I_i} \frac{r_j^{(k)}}{N_j}$$

There are a few problems with such an iteration: if a page has no outlinks, then in the iteration process it accumulates rank only via its inlinks, but this rank is never distributed further. Therefore it is not clear if the iteration converges. We will come back to this problem later. More insight is gained if we reformulate

as an eigenvalue problem for a matrix representing the graph of the Internet. Let Q be a square matrix of dimension n. Define

$$Q_{ij} = \begin{cases} \frac{1}{N_j} & \text{if there is a link from j to i} \\ 0 & \text{otherwise} \end{cases}$$

This means that row i has nonzero elements in the positions that correspond to inlinks of i. Similarly, column j has nonzero elements equal to N_j in the positions that correspond to the outlinks of j, and, provided that the page has outlinks, the sum of all the elements in column j is equal to one. Suppose a page i has no outlinks, all elements in i^{th} column will be 0.
For below graph,



Corresponding matrix becomes

$$Q = \begin{bmatrix} 0 & \frac{1}{3} & 0 & 0 & 0 & 0 \\ \frac{1}{3} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{3} & 0 & 0 & \frac{1}{3} & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & 0 & \frac{1}{3} & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 & \frac{1}{3} & 0 \end{bmatrix}$$

Since page 4 has no outlinks, the corresponding column is equal to zero. Obviously, the above definition is equivalent to the scalar product of row i and the vector r, which holds the ranks of all pages. We can write the equation in matrix form,

$$\lambda r = Qr,\ \lambda = 1$$

i.e., r is an eigenvector of Q with eigenvalue $\lambda = 1$. It is now easily seen that the iteration mentioned above is equivalent to

$$\lambda r^{(k+1)} = Qr^{(k)},\ k = 0, 1...$$

which is the **power method** for computing the eigenvector. However, at this point it is not clear that pagerank is well defined, as we do not know if there exists an eigenvalue equal to 1. It turns out that the theory of Markov chains is useful in the analysis.

3 Random Walk and Markov Chains

There is a random walk interpretation of the pagerank concept. Assume that a surfer visiting a Web page chooses the next page among the outlinks with equal probability. Then the random walk induces a Markov chain . A Markov chain is a random process in which the next state is determined completely from the present state; the process has no memory. The transition matrix of the Markov chain is Q_T . (Note that we use a slightly different notation than is common in the theory of stochastic processes.) The random surfer should never get stuck. In other words, our random walk model should have no pages without outlinks. (Such a page corresponds to a zero column in Q.) Therefore the model is modified so that zero columns are replaced with a constant value in all positions. This means that there is equal probability to go to any other Internet page. Define the vectors

$$d_j = \begin{cases} 1 & \text{if } N_j = 0 \\ 0 & \text{otherwise} \end{cases}$$

for $j = 1, ..., n$, and

$$e = \begin{pmatrix} 1 \\ 1 \\ . \\ . \\ 1 \end{pmatrix} \in \mathbb{R}^n$$

The modified matrix is defined

$$P = Q + \frac{1}{n} e d^T$$

With this modification the matrix P is a **proper column-stochastic matrix**. It has nonnegative elements, and the elements of each column sum up to 1. The preceding statement can be reformulated as follows.

$$e^T P = e^T$$

So the matrix in the previous graph modifies to

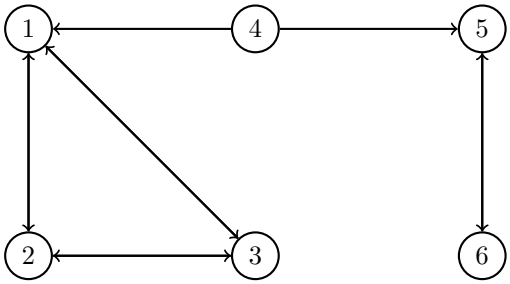
$$P = \begin{bmatrix} 0 & \frac{1}{3} & 0 & \frac{1}{6} & 0 & 0 \\ \frac{1}{3} & 0 & 0 & \frac{1}{6} & 0 & 0 \\ 0 & \frac{1}{3} & 0 & \frac{1}{6} & \frac{1}{3} & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & \frac{1}{6} & \frac{1}{3} & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{6} & 0 & \frac{1}{2} \\ 0 & 0 & 1 & \frac{1}{6} & \frac{1}{3} & 0 \end{bmatrix}$$

We would like to define the pagerank vector as a unique eigenvector of P with eigenvalue 1,

$$Pr = r.$$

The eigenvector of the transition matrix corresponds to a stationary probability distribution for the Markov chain. The element in position i, r_i, is the probability that after a large number of steps, the random walker is at Web page i. However, the existence of a unique eigenvalue with eigenvalue 1 is still not guaranteed. To ensure uniqueness, the matrix must be **irreducible**.

To illustrate the concept of reducibility, we give an example of a link graph that corresponds to a *reducible* matrix:



A random walker who has entered the left part of the link graph will never get out of it, and similarly will get stuck in the right part. The directed graph corresponding to an irreducible matrix is **strongly connected**: given any two nodes (N_i, N_j), in the graph, there exists a path leading from N_i to N_j .

Given the size of the Internet, we can be sure that the link matrix P is reducible, which means that the pagerank eigenvector of P is not well defined. To ensure irreducibility, i.e., to make it impossible for the random walker to get trapped in a subgraph, one adds, artificially, a link from every Web page to all the others. In matrix terms, this can be made by taking a convex combination of P and a rank-1 matrix,

$$A = \alpha P + (1 - \alpha)\frac{1}{n}ee^T$$

for some α satisfying $0 \leq \alpha 1$. It is easy to see that the matrix A is column-stochastic:

$$e^T A = \alpha e^T P + (1 - \alpha)\frac{1}{n}e^T ee^T = \alpha e^T + (1 - \alpha)e^T = e^T$$

The random walk interpretation of the additional rank-1 term is that in each time step the surfer visiting a page will jump to a random page with probability $1 - \alpha$ (sometimes referred to as teleportation). We now see that the pagerank vector for the matrix A is well defined.

4 Power Method for PageRank computation

We want to solve the eigenvalue problem

$$Ar = r$$

where r is normalized $||r||_1 = 1$. In this section we denote the sought eigenvector by t_1. Dealing with stochastic matrices and vectors that are probability distributions, it is natural to use the 1-norm for vectors. Due to the sparsity and the dimension of A (of the order billions), it is out of the question to compute the eigenvector using any of the standard methods on applying orthogonal transformations to the matrix. The only viable method so far is the power method. Assume that an initial approximation $r^{(0)}$ is given. The power method is given

in the following algorithm.

Algorithm 1: The power method for $Ar = \lambda r$

for *(k = 1, 2, . . . until convergence)* **do**
  $q^{(k)} = Ar^{(k-1)}$
  $r^{(k)} = q^{(k)}/||q^{(k)}||_1$
end

5 Multi-threading Power Method

C++ code for Power Method looks like this.

```
void power_method(float **graph, float *r, int n ){
    int max_iter = 1000; float eps = 0.000001;
    float* r_last = (float*) malloc(n * sizeof(float));

    for(int i = 0; i< n; ++i){
        r[i] = (1/(float)n);
    }

    while(max_iter--){
        for(int i = 0; i< n; ++i){
            r_last[i] = r[i];
        }
        for(int i = 0; i< n; ++i){
            float sum = 0.0;
            for (int j = 0; j< n; ++j){
                sum += r_last[j] * graph[i][j];
            }
            r[i] = sum;
        }

        for(int i = 0; i< n; ++i){
            r_last[i] -= r[i];
        }

        if(norm(r_last, n) < eps){
            return;
        }
    }
    return;
}
```

The complexity of this algorithm is $O(n^3)$, which will be very heavy to handle computationally for large scale web graph with 100,000 nodes. We can also observe that most of those loops in outermost while loop are iterating through every node. We can reduce the computational burden here by paralleling that operation. There is no sight of any data race, thread divergence too. Also, adjacent threads access adjacent memory locations in every case and Degree of coalescing is maximum. So, we can remove the suppress the for loops by paralleling them. Kernels after paralleling looks like this.

```
__global__ void initialize_rank(float* gpu_r, int n){
    int id = blockIdx.x * blockDim.x + threadIdx.x;

    if(id < n){
        gpu_r[id] = (1/(float)n);
    }
}

__global__ void store_rank(float* gpu_r,float* gpu_r_last, int n){
    int id = blockIdx.x * blockDim.x + threadIdx.x;

    if(id < n){
        gpu_r_last[id] = gpu_r[id];
    }
}

__global__ void matmul(float* gpu_graph, float* gpu_r, float* gpu_r_last, int n){
    int id = blockIdx.x * blockDim.x + threadIdx.x;

    if(id < n){
        float sum = 0.0;

        for (int j = 0; j< n; ++j){
            sum += gpu_r_last[j] * gpu_graph[id* n + j];
        }

        gpu_r[id] = sum;
    }
}

__global__ void rank_diff(float* gpu_r,float* gpu_r_last, int n){
    int id = blockIdx.x * blockDim.x + threadIdx.x;

    if(id < n){
        gpu_r_last[id] = abs(gpu_r_last[id] - gpu_r[id]);
    }
}
```

For sorting the final rank vector returned after power method, we can use **thrust::sort()** instead of normal **std::sort()**, even this saves us time in case of large scale network graphs.

6 Experiments

6.1 Time vs Number of nodes

No: of nodes	sequential	parallel	speedup
100	405	593	0.68
200	6556	2152	3.04
400	3462	936	3.69
800	22785	2356	9.67
1000	2972898	104831	28.35
5000	69266776	816446	84.83
10000	68519880	755148	90.73
16000	26509166	626986	42.28

```
Time taken :405.000000 for sequential implementation with 100 nodes.
Time taken :242.000000 for sequential implementation with 100 nodes.
Time taken :6556.000000 for sequential implementation with 200 nodes.
Time taken :3462.000000 for sequential implementation with 400 nodes.
Time taken :22785.000000 for sequential implementation with 800 nodes.
Time taken :2972898.000000 for sequential implementation with 1000 nodes.
Time taken :69266776.000000 for sequential implementation with 4772 nodes.
Time taken :68519880.000000 for sequential implementation with 9664 nodes.
Time taken :26509166.000000 for sequential implementation with 16062 nodes.
```

```
Time taken :593.000000 for parallel implementation with 100 nodes and 32 threads per block.
Time taken :408.000000 for parallel implementation with 100 nodes and 32 threads per block.
Time taken :2152.000000 for parallel implementation with 200 nodes and 32 threads per block.
Time taken :936.000000 for parallel implementation with 400 nodes and 32 threads per block.
Time taken :2356.000000 for parallel implementation with 800 nodes and 32 threads per block.
Time taken :104831.000000 for parallel implementation with 1000 nodes and 32 threads per block.
Time taken :816446.000000 for parallel implementation with 4772 nodes and 32 threads per block.
Time taken :755148.000000 for parallel implementation with 9664 nodes and 32 threads per block.
Time taken :626986.000000 for parallel implementation with 16062 nodes and 32 threads per block.
tcmalloc: large alloc 1364418560 bytes == 0x55fc3a780000 @  0x7fc9e2d381e7 0x55fc38032243 0x7fc9e1d
Time taken :15588785.000000 for parallel implementation with 18469 nodes and 32 threads per block.
```

We can observe that there is no trend with respect to number of nodes. Performance of PageRank algorithm also depends on how nodes are connected too. If it's a perfectly reducible matrix, algorithm converges quickly, but if that's not the case, algorithm doesn't converge even after 100 iterations.

6.2 Time vs Number of threads in a block

```
Time taken :950401.000000 for parallel implementation with 9664 nodes and 8 threads per block.
Time taken :839092.000000 for parallel implementation with 9664 nodes and 16 threads per block.
Time taken :752572.000000 for parallel implementation with 9664 nodes and 32 threads per block.
Time taken :759962.000000 for parallel implementation with 9664 nodes and 64 threads per block.
Time taken :868481.000000 for parallel implementation with 9664 nodes and 128 threads per block.
Time taken :900917.000000 for parallel implementation with 9664 nodes and 256 threads per block.
Time taken :2052449.000000 for parallel implementation with 9664 nodes and 512 threads per block.
Time taken :3698519.000000 for parallel implementation with 9664 nodes and 1024 threads per block.
```

As number of threads in a block increases, performance decreases.

6.3 Top Pages

Here are the top 3 nodes after computation of power method sequentially and parallelly.

```
[2]  %%shell
     cd /content/
     g++ sequential.cpp

     ./a.out 16000.txt
```

```
Rank 1 Node is 5075
Rank 2 Node is 9534
Rank 3 Node is 2526
Time taken :30574412.000000 for sequential implementation with 16062 nodes.
```

```
%%shell
cd /content/
nvcc parallel.cu

./a.out 16000.txt 32
```

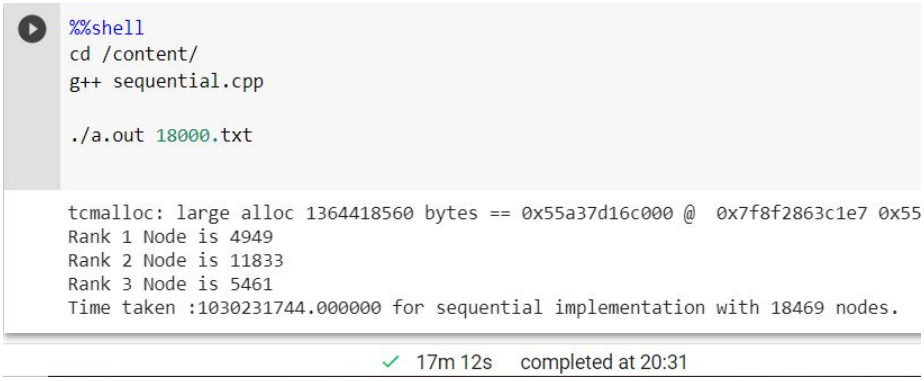
```
Rank 1 Node is 5075
Rank 2 Node is 9534
Rank 3 Node is 2526
Time taken :672219.000000 for parallel implementation with 16062 nodes.
```

7 New Idea : TweetRank

Above all experiments are conducted on web network datasets. But, now let's try them on twitter retweet network dataset.

Let us hope that just like in case of ranking pages from web graphs, we can rank best tweets for twitter search. If many users retweets an user's tweet, it may be important. And if that user retweets another tweet, it contains information which is not in his tweet, so even it is slightly important. Let us run our algorithm on that dataset.

Though we got results (next page) and a good speedup (25 s vs 17 min), we should not suppress the fact that sequential implementation took 17 min to run, which is very abnormal compared to other datasets. We can observe that it did not converge so our hypothesis of ranking **may be wrong**. Retweet network graph may not be reducible.

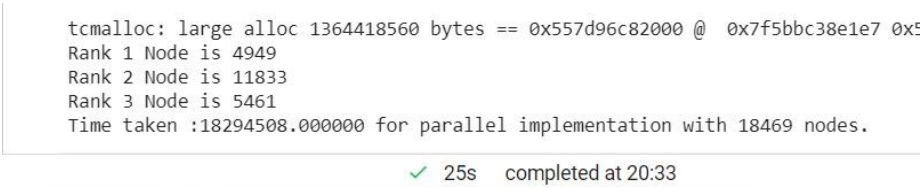


```
%%shell
cd /content/
g++ sequential.cpp

./a.out 18000.txt
```

```
tcmalloc: large alloc 1364418560 bytes == 0x55a37d16c000 @  0x7f8f2863c1e7 0x55
Rank 1 Node is 4949
Rank 2 Node is 11833
Rank 3 Node is 5461
Time taken :1030231744.000000 for sequential implementation with 18469 nodes.
```

✓ 17m 12s completed at 20:31



```
tcmalloc: large alloc 1364418560 bytes == 0x557d96c82000 @  0x7f5bbc38e1e7 0x5
Rank 1 Node is 4949
Rank 2 Node is 11833
Rank 3 Node is 5461
Time taken :18294508.000000 for parallel implementation with 18469 nodes.
```

✓ 25s completed at 20:33

8 References

- Parallel-implementation-of-PageRank
- PageRank Explained: Theory, Algorithm, and Some Experiments
- Book: Matrix Methods in Data Mining and Pattern Recognition
- Real-World Dataset
- More data
- Web network datasets
- Twitter Retweet network dataset