High Performance Graph & Data Analytics Project
Personalized PageRank on GPU

Pablo Giaccaglia - Alessandro La Conca
Politecnico di Milano
pablo.giaccaglia@mail.polimi.it - alessandro.laconca@mail.polimi.it

ABSTRACT
— Personalized PageRank, also known as Topic Sensitive PageRank [2], is a link analysis algorithm and a derivation of the algorithm originally behind Google Search [1]. CPUs are not well suited for such algorithm, leading to latencies not meeting common real-world constraints. For this reason the goal of this project is to implement it on GPU, to achieve maximum speedup on large and sparse graphs. In this report we present an overview of the various reasonings, findings and challenges we face when trying to make the algorithm run as fast as possible, while meeting reasonable accuracy constraints. The project was developed using CUDA parallel computing platform on C++ and tested on an Acer laptop with NVIDIA GeForce RTX 3080 Laptop GPU.
***Index Terms*—PageRank, GPU, CUDA, C++**

1 INTRODUCTION

To capture more accurately the notion of importance with respect to a particular page, while avoiding the problem of heavily linked pages getting highly ranked for queries for which they have no particular authority, Personalized PageRank algorithm, in its original formulation, employs the power iteration method, whose convergence, in this case, produces the PageRank vector over the web, regarding a specific page, or more generally, a node in a graph. This vector can be viewed as the stationary probability distribution over pages induced by a random walk on the web [2]. Computed once given a graph, its values can be used as a ranking for various purposes, such as generating search engine results. Here is the formula, which takes into account the fact that the graph-representing matrix may not be stochastic nor irreducible, requisites for the method to guarantee convergence:

$$\mathbf{p_{t+1}} = d\mathbf{X}\mathbf{p_t} + \frac{d}{|V|}\left(\overline{\mathbf{d}}\mathbf{p_t}\right)\mathbf{1} + (1-d)\overline{\mathbf{v}} \qquad (1)$$

Here is the stopping convergence condition:
$\|\mathbf{p_t} - \mathbf{p_{t-1}}\| < \epsilon$, where ϵ is a desired convergence threshold. Note that when the power method asymptotically converges, the resulting rank is an eigenvector of the generated stochastic matrix (known as the "Google Matrix" [1]), whose Eigenvalue, d, is empirically proven to be the sub-dominant one [3]. A brief overview of the parameters:

- d: damping factor, the probability of the walker of following an arbitrary link
- $1-d$: dangling factor, the probability of the walker of jumping to some random page
- $\mathbf{X}$: normalized adjacency matrix representing the links between pages. Each entry is multiplied by $\frac{1}{\text{outDegree}}$
- $\mathbf{p_t}$: Rank stochastic vector computed at previous iteration
- $|\mathbf{V}|$: number of vertices in the graph
- $\overline{\mathbf{d}}$: dangling vector (entry is 1 if the vertex is dangling, 0 otherwise). Note that a node is dangling if has no outward edges
- $\overline{\mathbf{v}}$: personalization vertex vector (entry is 1 for the personalization vertex, 0 otherwise)

The difference from pure PageRank can be seen in the biasing of the computation, to increase the importance given to the personalization vertex at each iteration. This is represented by the rank boost provided by $(1-d)\overline{\mathbf{v}}$, whose interpretation, in terms of the random-walk model, is the addition of a transition edge from each node to the personalized one [2].

2 PROPOSED WORK

As previously stated, our goal is to exploit CUDA to compute the Rank vector as fast as possible, setting some accuracy constraints. In particular, for a given personalization vertex, we aim to compute the top-20 highest ranked vertices, with the relaxation that at least 16 should be correct, even if in the wrong order. In practice we want 80% accuracy on the result set. When working with very large graphs (billions of nodes), processing time can be tremendously high, so speed is more important than 100% accuracy, especially in the context of web pages ranking. Moreover, as shown in the following sections, a small reduction in accuracy can lead to huge speedups in execution. To properly understand the scope for parallelization and execution speedups we developed several implementations, of which 3 are here overviewed. The first, which we identify as the **"Naive Implementation"**, represents the starting point of our improvements, since it consists of almost straightforward CPU-to-GPU transposition kernels, thus highly inefficient. The second, which we identify as the **"Final COO Implementation"**, represents, as the name suggests, the final optimized implementation employing the COO format and, because of the last implementation here reported, the last one based on the power method. Finally, the third implementation, which we identify as the **Monte Carlo Implementation**, proposes a parallel implementation of the Monte Carlo complete path algorithm [4], resulting in the fastest of the three proposed.

3 EXPERIMENTAL SETUP

The experiments aiming to test the performance of the proposed implementations were run on a laptop with an NVIDIA GeForce RTX 3080 Laptop GPU. The reference serial implementation of the power method was run on an AMD Ryzen 9 5900HX CPU and 32 GB of DDR4 RAM.

4 EVALUATION DATASET

The performance of the parallel implementation of power method for computing Personalized PageRank and it's corresponding variants, together with the Monte Carlo method, are evaluated on the publicly available wikipedia − 20070206 dataset [5]. It consists of 3,566,907 pages and 45,030,389 links. The link structure is represented in Coordinate Format (COO), a popular format for storing sparse matrices.

5 STORAGE FORMATS

Both **"Naive Implementation"** and **"Final COO Implementation"** employ the **Coordinate Format (COO)** for storing the Wikipedia matrix. This choice is determined by the most time demanding kernel, which computes the parallel Sparse Matrix-Vector Multiplication (SpMV), the core computation of the iterative power method. This representation has several advantages. It doesn't waste computation time for fully-zero rows and it is well suited for extremely sparse, roughly random matrices, but with high variance in the sparsity between rows. Finally, if implemented properly, SpMV kernels with COO format avoid problems faced with other formats, such as Compressed Sparse Row (CSR), which has to deal with drawbacks like thread divergence and non-coalesced memory access patterns. On the other hand the **Monte Carlo Implementation** employs a Compressed Sparse Column (CSC) storage format, in which each column is stored as a sparse (column) vector. This format is well suited for the Monte Carlo implementation since it easily allows to retrieve the outgoing edges of a given vertex.

6 NAIVE IMPLEMENTATION

This implementation basically consists of an iteration until convergence of the power method, where the computation has been organized through the following kernels:

- `compute_dangling_factor_gpu`: computes the dot product $\frac{d}{|V|}\left(\bar{\mathbf{d}}\mathbf{p_t}\right)$
- `cooSPMV`: computes the SpMV $d\mathbf{X}\mathbf{p_t}$
- `vectorScalarAdd`: sums the results of the previous 2 kernels
- `incrementBy1`: adds $(1-d)$ to the personalization vertex entry of $\mathbf{p_{t+1}}$
- `compute_square_error_gpu`: partially computes Euclidean norm $\|\mathbf{p_t} - \mathbf{p_{t-1}}\|$. The square root is computed on CPU.

Note that the reported order reflects the kernel calls order. This implementation suffer from several inefficiencies, due to both the organization of the computation into sub-components and their implementations. As an example, it is useful to take a look at the `compute_dangling_factor_gpu`, which shares several defects with the others.

```
compute_dangling_factor_gpu(T1 *dangling, T2* pr,
                            T2 *result, int V){

 size_t i = blockIdx.x * blockDim.x + threadIdx.x;

  for(; i < V; i += blockDim.x * gridDim.x){
       T2 val = (T2) dangling[i] * pr[i];
       atomicAdd(result, val);
   }
}
```

Listing 1: `compute_dangling_factor_gpu` kernel

The kernel uniquely employs the global memory and inter-thread communication is the core mechanism of the computation. Although memory access pattern is linear, the computation result is progressively accumulated through an atomic addition, which is synchronized for the whole GPU. This is highly inefficient because a serial bottleneck is generated, many threads are stalled most of the time and the computation becomes almost sequential. In the next section better solutions will be presented, employing shared memory, reduction and proper block threads synchronization. The implementation, which uniquely employs integers and double precision floats, is tested by running 100 times the algorithm, each of with randomly initialized personalization vertex. All the runs share the same configuration: convergence threshold equal to 10^{-6}, max iterations equal to 30 and damping factor $\alpha = 0.85$. For each run, the accuracy of the GPU top-20 results is checked upon the CPU implementation. The following table reports average results over all the runs, considering that the first 3 runs of the algorithm ("warm up" iterations) aren't taken into account, to avoid biasing results with initial slower executions. The reported speedup is with respect to CPU execution time.

	Avg. Time	Avg. Exec	Mean Accuracy	Speedup
CPU	-	7852 ms	100%	1x
Naive	1143,41 ms	1021,57 ms	100%	7,68x

TABLE I: Naive implementation execution time

Finally, the most relevan kernels' execution times.

Instances	Avg. Exec	Min. Exec	Max. Exec	Kernel
3000	15,5 ms	15 ms	16 ms	**cooSPMV**
3000	6,7 ms	6,5 ms	7,6 ms	**compute_square_error_gpu**
3000	6,7 ms	6,5 ms	11,4 ms	**compute_dangling_factor_gpu**

TABLE II: Kernel execution times

Here the information about block and grid sizes.

Block size	Grid size[1]	Kernel
128	$\frac{(E+127)}{128}$	**cooSPMV**
128	$\frac{(V+127)}{128}$	**compute_square_error_gpu**
128	$\frac{(V+127)}{128}$	**compute_dangling_factor_gpu**

TABLE III: Block and Grid sizes

[1] E is the number of edges, V is the number of nodes

7 FINAL COO IMPLEMENTATION

This implementation introduces several improvements that lead to a much lower execution time. Single precision floating points were employed instead of doubles, leading to faster multiplications and sums. CUDA half precision floating points were not well suited for such task, since such low precision leads to poor accuracy results, due to the higher precision demanding values the algorithm deals with. Moreover shared memory has been employed as much as possible, resulting in faster kernels. Another improvement has been provided by the usage of CUDA single precision intrinsics, device only highly optimized instructions with enough low numerical accuracy to achieve wanted results. Finally some heuristics have been introduced to push even further the speed improvements.
Here are the kernels of the main loop updating the Rank vector until convergence:

- `dangling_kernel`: computes the "pseudo" dot product $\frac{d}{|V|}\left(\overline{\mathbf{d}}\mathbf{p_t}\right)$. Involves both usage of shared memory and reduction of partial results. Its functioning is better clarified below
- `__spmv_coo_flat`: computes the SpMV $d\mathbf{X}\mathbf{p_t}$. The kernel is an adaptation of the one implemented in the CUSP library. It works with COO format sorted by row to exploit parallelism and flattens data irregularity caused by a variable number of zeros per row
- `vectorScalarAddAndIncrement_math`: sums the results of the previous 2 kernels and adds $(1-d)$ to the personalization vertex entry of $\mathbf{p_{t+1}}$
- `euclidean_kernel_math`: partially computes Euclidean norm $\|\mathbf{p_t} - \mathbf{p_{t-1}}\|$. The square root is computed on CPU

The dot product computation $\frac{d}{|V|}\left(\overline{\mathbf{d}}\mathbf{p_t}\right)$ is handled differently from the naive case. In fact, instead of performing standard dot product between the stochastic matrix and the column vector of 0s and 1s, only the 1s positions are taken into account and the result value is obtained by summing the matrix elements, which are multiplied by 1 in the standard case, thus achieving much faster computation.
The implementation, to have proper understanding of the improvements, is tested under the same conditions of the Naive one. The following table reports the average results.

	Avg. Time	Avg. Exec	Mean Accuracy	Speedup
CPU	-	7852 ms	100%	1x
Final COO	754,04 ms	750 ms	100%	10,46x

TABLE IV: Final COO implementation execution times

Finally, the most relevan kernels' execution times.

Instances	Avg. Exec	Min. Exec	Max. Exec	Kernel
3000	8,5 ms	8 ms	9 ms	**cooSPMV**
3000	0,1 ms	0,1 ms	0,13 ms	**compute_square_error_gpu**
3000	0,015 ms	0,013 ms	0,018 ms	**compute_dangling_factor_gpu**

TABLE V: Kernel execution times

Here the information about block and grid sizes.

Block size	Grid size[2]	Kernel
128	$\frac{(E+127)}{128}$	**__spmv_coo_flat**
128	$\frac{(V+127)}{128}$	**euclidean_kernel_math**
128	$\frac{(D+127)}{128}$	**dangling_kernel**

TABLE VI: Block and Grid sizes

7.1 HEURISTICS

Since our goal is to retrieve the top-20 results with at least 80% accuracy and the previous implementation do not take advantage of this relaxation, we decided to introduce several heuristics, whose contribution, pros and cons are now discussed. First these are considered stand alone, then our proposed combination is presented.

7.1.1 Skip Euclidean norm computation

The first improvement comes from the finding that, given the execution conditions previously shown, faster runs take about 10 iterations, while slower ones take approximately 30 iterations, meaning that it is time-wasteful to compute the norm at each iteration. As a matter of fact, computing Euclidean distance every 5 iterations lead to a mean execution time of 613 ms, keeping accuracy to 100%. The right frequency of computation can be easily found, mainly through a trial and error approach, with different thresholds or when multiple heuristics are combined together, as shown in the following subsections.

7.1.2 Partial error computation

The second improvement comes from the fact that in both implementations presented the square root producing the resulting error is computed on CPU, thus involving a relatively slow function. For this reason, only the sum of the difference of squares can be considering, adjusting consequentially the `convergence_threshold`. Resulting mean execution time is 650 ms, while accuracy is 100%.

7.1.3 Fixed ranking with danglings

If the Personalization vertex is dangling, the Rank vector is fixed and that vertex is included in the top-20. The reason of this lies in the functioning of the algorithm, which adds $1-\alpha$ to the Rank value of the Personalization vertex, which, if dangling, won't directly propagate its score to other nodes. This means that the rank is fixed apart from the dangling node, so a top-19 rank can be precomputed through direct observation of the results and exploited by adding the dangling vertex, achieving full accuracy almost immediately, since neither relative positions nor Rank scores are considered. In the studied Wikipedia graph only 2.8% of the nodes are dangling, so the overall speedup is marginal, but other graphs can highly benefit from this consideration. The execution time

[2] E is the number of edges, V is the number of nodes, D is the number of danglings

with Personalization danglings is of 3 ms, with 100% accuracy. Here are the top 19 nodes ordered by ranking:

`Wikipedia top19` :
{1702309, 500469, 1518892, 24716,
689491, 195101, 932394, 2984189, 1835017,
577659, 28196, 7030, 2257865, 2532493,
28020, 2979297, 9008,27566 , 2144742}

7.1.4 Proposed combination

Final COO implementation with heuristics proposes the following combination:

- Precomputed result on dangling indexes ("fixed ranking with danglings" improvement).
- Only the sum of difference of squares is considered ("partial error computation" improvement).
- Skip sum of difference of squares computation ("skip Euclidean norm" improvement). In particular the computation rule is the following:

  Compute if (iter $>$ 3 and iter $<$ 12) or (iter $>$ 12 and iter % 3 == 0)

  It highly depends on the heuristics with which is combined and it is adaptable with different thresholds.
- Custom convergence threshold: as previously said, once removed the square root computation, the convergence threshold is changed accordingly. We noticed that most of the time the final ranking (at desired convergence) is reached much before the desired convergence. What takes most of the computation is the establishment of the relative positions and the rank scores. Since we do not care about these, fixing a custom higher convergence threshold resulted to be a winning strategy. After many trials we fixed this value to $c = 6 \cdot 10^{-9}$, which lead to achieving at least 80% accuracy on average 98% of the time. It is important to point out that this parameter is not easy to tune, a lot of time is required to find suitable values, it highly depends on the chosen graph, on the "real" reference threshold and it is based on empirical results. Finally note that if admitted percentage of computations leading to accuracy lower than 80% are properly defined, based on the context, this parameter can be further tuned to achieve much higher speedups than the ones presented.

The following table reports the average results.

	Avg. Time	Avg. Exec	Mean Accuracy	Speedup
CPU	-	7852 ms	100%	-
Final COO	754,04 ms	750 ms ms	100%	10,46x
Final COO w/ Heuristics	81,17 ms	77,13 ms	95,2%	101,8x

TABLE VII: Final COO implementation with heuristics execution time

8 MONTE CARLO IMPLEMENTATION

To increase the computation speed even further, knowing that the SpMV multiplication was the most computation intensive operation, we looked for algorithms that retrieved the top 20 vertices without using the power method. We parallelized the Monte Carlo complete path algorithm [4]

8.1 Algorithm

The algorithms consists in simulating m random walks starting from a seed node, in our case the personalized vertex. At each step:

- the walker can terminate with probability $1 - d$.
- the walker moves to a random node connected to an outgoing edge with probability d. (if the node has no outgoing edges the walk terminates).

Let N_j be the number of visits by all walkers to vertex j, then the pr of vertex j is defined as follows:

$$pr_j\left(seed, c\right) = \left(1 - c\right) \frac{1}{m} N_j\left(seed\right)$$

Observation: since we only care about the rank, we can simplify the algorithm by allowing to have PageRank values greater than one, to slightly fasten the computation:

$$pr_j\left(seed, c\right) = N_j\left(seed\right)$$

Algorithm 1 Monte Carlo Complete Path

```
1:  for i = 0 to m do
2:      n ← source
3:      pr[n] ← pr[n] + 1
4:      if randProb() < 1 − d then
5:          stop walk
6:      else
7:          if n.hasNoOutgoingEdges() then
8:              stop walk
9:          end if
10:         n ← n.randomOutgoingEdge()
11:     end if
12: end for
```

8.2 Special cases

If the node is dangling we can do the same as in 7.1.3 But let's focus on the following case, where a seed Node A, with some incoming edges has just one outgoing edge towards node B, which is dangling. Since at each step the random walk can only stop or move towards an outgoing edge after running the MC Complete Path algorithm, the ranking will be the following:

0) Node A, Value> 0
1) Node B, Value> 0
2) Random Node i, Value$= 0$

..

19) Random Node j, Value$= 0$

In this case we only get 2 correct nodes since the random walks only visit node A and B. In general all cases in which the m random walks can't reach at least 20 nodes lead to low accuracy. The solution is to initialize the PageRank vector of the top 19 nodes without the personalized vertex to a value greater than zero and decreasing according to their rank. In particular we initialize the values of these nodes to $\frac{1}{i+2}$, where i is their position in the ranking. By doing this the visited

nodes will be at the top of the ranking and if less than 20 spots are filled, the remaining spots will be filled by the nodes in the top 19 ranking from the list shown in subsection 7.1.3

8.3 Gpu Implementation

We started by using 1 thread for each random walk and an atomic add to increase the PageRank value of visited nodes. To generate the random numbers needed by the MC algorithms we used the $Philox - 4x32 - 10$ pseudo number generator from the $curand$ library [6]. Then the performance of the algorithm was improved in the following ways:

8.3.1 Atomic add on node connected to outgoing edges

Since all the random walk start from the personalized vertex in step 0 of the algorithm, all the threads will call the atomic add on the same area on memory. To mitigate this problem, in the first random walk each thread starts from a random vertex connected to an outgoing edge of the seed. In this way the atomic add acts on different memory locations.

8.3.2 Improved thread utilization

We use a local variable to count how many steps each thread has performed. If a thread terminates a random walk and the number of steps is less then the spawnWalkerThreshold that same thread will start a new random walk from the seed. The number of steps is not reset but it will increase until we reach the threshold. We also terminate all the random walks after the number of steps has exceeded the maxWalkLen.

The spawnWalkerThreshold and maxWalkLen are defined as following:

- `maxWalkLen`= $min(m, 50)$, where m is an input argument and it indicates the number of iterations of the CPU power method algorithm. By using the same parameter the accuracy and execution time of MC Complete Path increases with m similarly to the accuracy and execution time of the baseline
- `spawnWalkerThreshold`=$maxWalkLen - 30$. We do not set the spawnWalkerThreshold to the same value as the maxWalkLen. We are giving to each walker a buffer of 30 steps to terminate to avoid possible biases. This buffer size depends on the damping factor.

For example if m is 60 maxWalkLen is 60 and spawnWalkerThreshold is 30. If a random walk terminates before step 30 a new random walk will start. In this way each thread runs multiple random walks instead of only one.

9 Notes and bug fixes

This section talks about some improvements we made after the end of the contest.

- In the MC implementation we were allocating the array containing the top 19 vertices at each iteration. It should be allocated only once inside `alloc()` function.

10 RESULTS COMPARISON

These results are the outcome of tests performed under the same conditions shown in previous sections. The novelty here is in the number of blocks and threads for the Monte Carlo Implementation, which are fixed to 4000 and 128.

10.1 Implementations speed comparisons

Average time is computed as average reset + average execution time

Impl.	Avg. Time	Avg. Exec	Avg. Accuracy	Std. Dev. Accuracy
CPU	-	7852 ms	100%	-
Naive	1143,41 ms	1021,57 ms	100%	-
Final COO w/ Heuristics	81,17 ms	77,13 ms	95,2%	0,061
MC	16,36 ms	14,35 ms	98,3%	0,027

TABLE VIII: Implementations' accuracy and time

Implementation	Speedup
CPU	1x
Naive	7,68x
Final COO w/ Heuristics	101,8x
Monte Carlo Complete Path	547,2x

TABLE IX: Speedup with respect to CPU execution time

10.2 Accuracy and speed trade-off

By reducing the number of block we can change the number of walkers leading to an increase in speed and a decrease in accuracy.

walkers	Avg. Exec.	Avg. Acc.	Std. Dev
64000	5,57 ms	96,2%	0,053
512000	14,35 ms	98,3%	0,027

TABLE X: Accuracy and speed trade-off

11 CONCLUSIONS

In this work we showed two Personalized Page Rank implementations using the power method and the various improvements between the Naive and the Final COO implementation. We have also showed that by using various heuristics and by decreasing the accuracy we can vastly decrease the execution time of the algorithm. In particular in the MC Carlo Complete Path implementation we managed to obtain an execution time that is the 0.18% of the baseline executed on the CPU with an averge accuracy of 98.3%.

References

[1] The PageRank Citation Ranking: Bringing Order to the Web http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf

[2] Topic-Sensitive PageRank http://www-cs-students.stanford.edu/~taherh/papers/topic-sensitive-pagerank.pdf

[3] The Second Eigenvalue of the Google Matrix https://nlp.stanford.edu/pubs/secondeigenvalue.pdf

[4] Monte Carlo Complete path - https://www.researchgate.net/publication/225138130_Quick_Detection_of_Top-k_Personalized_PageRank_Lists

[5] Gleich/wikipedia-20070206 Dataset https://sparse.tamu.edu/Gleich/wikipedia-20070206

[6] cuRAND, the CUDA random number generation library. https://docs.nvidia.com/cuda/curand/index.html

12 CODE AVAILABILITY

The code of the implementations is publicly available as a git repository on GitHub at https://github.com/Calonca/hpgda-spring22/tree/main/cuda_CONTEST