Solving write conflicts in GPU-accelerated graph computation: a PageRank case-study

Diego Piccinotti*, Edoardo Ramalli*, Alberto Parravicini†, Rolando Brondolin†, Marco Santambrogio†
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milano, Italy
*{diego.piccinotti, edoardo.ramalli}@mail.polimi.it
†{alberto.parravicini, rolando.brondolin, marco.santambrogio}@polimi.it

***Abstract*—Graph ranking algorithms, such as PageRank, are widely used in a number of real-world applications like web search. As the size of the graphs on which these algorithms are applied gets bigger and bigger, it is necessary to devise powerful and flexible techniques to accelerate and parallelize the computation, both at software and hardware level. Leveraging GPUs is a promising direction thanks to their highly parallel computing capabilities, but execution time is often hampered by write conflicts. In this paper, we present a solution to handle write conflicts in highly parallel computations on GPU, and show how this technique can effectively be used to accelerate the computation of PageRank by a factor of 5x, with respect to a baseline in which conflicts are not handled. Our solution is implemented at software level, and doesn't require specific hardware resources.**

***Index Terms*—Graph Algorithms, GPU, Atomic Addition, PageRank**

I. Introduction

Graphs represent a powerful tool to model many real world scenarios, including social, biological and communication networks. Due to their power, great effort has been devoted to research graphs algorithms, which allow to extract informations stored (implicitly or explicitly) in the graph in a complete and efficient way. Among the different classes of graph algorithms, a very important spot is occupied by ranking algorithms. Ranking algorithms, such as PageRank or Closeness Centrality, sort the vertices of a graph based on one or more importance criteria, and find applications in fields such as web search and ranking analysis, for instance in sport players ranking [1]. As graphs can be represented as sparse matrices [2], it is common to reformulate ranking algorithms as computations on matrices. This representations enable a natural parallelization of the computation, and significantly benefit from the hardware acceleration provided by the native Single Instruction, Multiple Data (SIMD) architecture of devices such as Graphics Processing Units (GPUs).

Computation on GPUs is usually split over multiple threads, which represent a fragment of code being executed independently to perform a well defined task. Parallel matrix computation requires different threads to work on the same data simultaneously, but when threads need to concurrently update data shared among them, multiple data conflicts, such as Write-After-Write (WAW) or Write-After-Read (WAR) conflict, may arise. Furthermore, in ranking applications, rankings are usually normalized and should add up to 1. This introduces the need for floating point arithmetic operations, which lack commutative property due to machine precision errors. Non-commutativity becomes a problem when performing parallel calculations on the same data, since the summation of results on the same memory location happens in a non-repeatable order. Even worse, rounding error introduced by floating point operations accumulates over iterations and, when analyzing large graphs, can disrupt the correctness of results and prevent algorithms' termination.

However, some of the parallel summation techniques originally developed for integer arithmetic (which does not suffer from precision errors) could be applied, with some modification, to floating point numbers. We analyzed the applicability of interleaved reduction of arrays using warp atomic methods[1], which currently work for integers, and developed a solution to the accumulating error generated by reduction methods.

The contribution of this work are as follow:

- A methodology to systematically handle write conflicts at software level, without explicit thread synchronization or serialization of write operations. We show that our approach can successfully be applied to summation of partial results on the same memory location, even when dedicated hardware solutions are not present.
- We evaluate our methodology through a case-study based on PageRank, a well-known graph ranking algorithm. We show that write conflicts that would appear during the matrix multiplication phase of the computation are correctly handled, and we are able to outperform a baseline from the point of view of execution time.

The technique presented in this work could be successfully applied also to distributed systems, to sum data coming from different nodes: when designing the distribution control application, the architecture of the distributed system is often unknown before deployment, so our proposed solution, being hardware independent, could offer an alternative to checking the presence of specific hardware, and allow a better code portability.

[1] https://devblogs.nvidia.com/faster-parallel-reductions-kepler/

II. State of the art

In this work, we focus on a practical case-study, and analyse how write conflicts impact the performance of a GPU implementation of PageRank, arguably the most well known graph ranking algorithm. Developed by Larry Page and Sergej Brin, PageRank assign a score to webpages, on the assumption that information on the web can be ranked based on human interest and attention devoted to it [3]. PageRank is compared by the authors to an idealized random web surfer, and is able to efficiently compute large numbers of page rankings, showing good results in terms of scalability over the number of pages. The original formulation is based on a recursive expression, and PageRank values are computed in an iterative fashion by taking into account the ranking of the pages linked to the page it is currently computing.

Later works have introduced a matrix formulation of PageRank [4] [5] based on power iteration, which allows to speed up the computation by means of parallelization. Matrix formulations well fit SIMD-capable devices, like GPUs, combined with data compression techniques (like Compressed Sparse Row (CSR) [6]) which allow to overcome memory limitations of such devices [7].

Another approach that has been explored is the simulation of random surfing to visit the web network graph. This approach highly benefits from distribution over multiple nodes [8] and the usage of multiple GPUs as shown by Rungsawang and Manaskasemsak [9].

The de-facto standard for GPU programming is considered to be the Compute Unified Device Architecture (CUDA) framework[2]. This framework enables the use of compute capabilities of NVIDIA's graphic cards to perform general purpose computation. SIMD capabilities of graphics processors are exploited to perform fast matrix manipulation, which is core to many graph algorithms as showed by Harish et al. [10].

Parallel computing offers advantages in terms of performance but, on the other hand, during parallel computations data can be shared among threads and this causes read and write conflicts that need to be properly managed to preserve data consistency. Francy and Lipasti's work on this topic [11] highlights that GPUs do not benefit from advanced atomic operation techniques that are found on board in modern Central Processing Units (CPUs) and that specific techniques need to be designed to find a way around this performance-crucial issue.

III. Problem Description

The computation of PageRank simulates an idealized random web surfer who explore the web through web links from the page he is currently surfing. PageRank weights the links between pages by the importance of the referencing page, assuming that its PageRank value is equally split among outwards links.

The PageRank algorithm nicely fits SIMD architectures, as it can be defined with a matrix formulation[3] of form

$$\mathbf{P_{k+1}} = ((1-d)\frac{\mathbf{E}}{|\mathbf{V}|} + d \cdot \hat{\mathbf{A}}^T)\mathbf{P_k} \tag{1}$$

where $\mathbf{P_k}$ represents the PageRank vector at iteration *k* (i.e. the vector that contains the PageRank score of each vertex), $|\mathbf{V}|$ is the number of vertices/web-pages, d is the *damping factor* (which represents the probability of continuing surfing on the current link chain, instead of jumping to a random page) and $\mathbf{E}$ is a square matrix of size $|\mathbf{V}| \times |\mathbf{V}|$ filled with ones. Finally, $\hat{\mathbf{A}}$ is the *transition probability matrix*, defined as

$$\hat{\mathbf{A}} = \mathbf{D^{-1}A} \tag{2}$$

$\mathbf{A}$ is the $|\mathbf{V}|\times|\mathbf{V}|$ *adjacency matrix*, containing 1 in position $\{i, j\}$ when there is a directed edge between vertex i and vertex j, otherwise 0. $\mathbf{D}$ is a $|\mathbf{V}| \times |\mathbf{V}|$ diagonal matrix, containing for each element of the diagonal the out-degree, i.e. the number of outward edges for the corresponding vertex.

Starting from this formulation, some tweaks can be applied to simplify the computation. Due to inner product distributivity, the $\mathbf{P_{k+1}}$ expression can be factored as the sum of two contributions

$$\mathbf{P_{k+1}} = (1-d)\frac{\mathbf{E}}{|\mathbf{V}|} \cdot \mathbf{P_k} + d \cdot \hat{\mathbf{A}}^T \cdot \mathbf{P_k} \tag{3}$$

In particular, the term $(1-d)(\mathbf{E}/|\mathbf{V}|) \cdot \mathbf{P_k}$ can be rewritten by expanding the inner product $\mathbf{E} \cdot \mathbf{P_k}$ as

$$\frac{(1-d)}{|\mathbf{V}|}(1 \cdot \mathbf{P_k^{(1)}} + 1 \cdot \mathbf{P_k^{(2)}} + \cdots + 1 \cdot \mathbf{P_k^{(|\mathbf{V}|)}}) \tag{4}$$

Which, since PageRank vector sums to 1 by construction, leads to

$$\frac{(1-d)}{|\mathbf{V}|} \cdot \sum_{i=1}^{|\mathbf{V}|} \mathbf{P_k^{(i)}} = \frac{(1-d)}{|\mathbf{V}|} \tag{5}$$

where $(1-d)/|\mathbf{V}|$ represents a constant quantity, since d and $|\mathbf{V}|$ are scalar values. This allows to skip the computation of this contribution at each iteration of the algorithm, being sufficient to compute this contribution in a pre-processing step.

The most computationally intensive part of PageRank algorithm is the remaining part of the factorization eq. (3). However, graphs encountered in real-world applications have a high sparsity degree, meaning that each vertex has only a handful of in-going or out-going arcs. It is natural to represent graphs with sparse matrices, encoding only the non-zero links present in the graph instead of the full adjacency matrix. In our implementation, we leveraged a CSR representation [6]. By storing the transposed graph as a CSR matrix, it is possible access the in-neighbours of each vertex in a very efficient way, as they correspond to the rows of the matrix. As the CSR representation encodes only non-zero entries of the matrix, we avoid multiplications of zeroes of $\hat{\mathbf{A}}$ matrix.

$\hat{\mathbf{A}}$ must be a stochastic matrix to ensure the correct termination of the algorithm. To achieve this, empty rows corresponding to vertices with no out-going edges are filled with

[2] https://developer.nvidia.com/cuda-toolkit

[3] www.dsi.unive.it/~calpar/New_HPC_course/AA12-13/project12-13.pdf

$1/|\mathbf{V}|$, which represents an equal probability of jumping to any page in the graph when reaching a page not linked to any other. When transposed, such rows become empty columns. These empty columns offer a constant contribute to the result of each $\mathbf{P_{k+1}}$ element when performing the inner product $\hat{\mathbf{A}}^{\mathbf{T}} \cdot \mathbf{P_k}$, and their contribution can be computed just once at the beginning of each iteration.

To compute $d \cdot \hat{\mathbf{A}}^T \cdot \mathbf{P_k}$ we designed an initial GPU implementation, which exploited dynamic parallelism[4] (allowed since Kepler architecture on NVIDIA GPUs) using a child thread per row to compute the new values of the PageRank vector, by adding partial results. Then, each of these threads started one child thread per multiplication to perform. This approach induced heavy synchronization and control problems which couldn't be efficiently solved using CUDA synchronization capabilities, forcing the use of `cudaDeviceSynchronize()` to stall the whole GPU and wait for each single thread to complete its execution. This approach had poor execution time but presented the advantage of clearly separating the partial result of each row. To address execution time, we decided to apply warp atomic reduction techniques[1] to improve performance of all reductions in the algorithm's computation. However, this implementation provided meaningless final results due to excessive atomic pressure generated by reduction of large arrays of floating point values. The atomic pressure caused the increase of rounding error and prevented us from achieving meaningful results in the PageRank calculation, as we required a high precision degree (a convergence error below 10^{-6}, computed with L2 norm) for our intended application.

Without the possibility of speeding up, and to avoid synchronizing the whole GPU, we dropped the dynamic parallelism approach. We then proceeded to modify the data structure to keep separation of the data belonging to each row, introducing a *bookkeeping* vector with same size as the data vector and containing in each position the corresponding row of the value in the data vector. Thanks to the *bookkeeping* vector, we are able to exploit the CSR representation of the graph, and access the in-neighbours of a vertex in a simple and efficient way. The *bookkeeping* vector allows to know where to write the partial results, but it causes multiple writes to the same memory location, which arise memory conflicts. In the next section, we will detail our proposed approach, which enables our algorithm to deal with memory conflicts while preserving the computational benefits introduced by the additional *bookkeping* vector.

IV. Proposed Solution

To overcome the problem of concurrent writes, we propose a solution centered around a simple function, implemented through the CUDA API. This function allows to write a desired value on a specific memory location, while returning the previously stored value. By using this approach in a multi-threaded execution environment, we were able to recognize when a different thread would jump in and overwrite the value that was just stored, thus allowing to keep repeating the write but without any disruption of the values from other threads. This approach allowed us to avoid forcing explicit thread synchronization or the use of a semaphore-like structure.

[4]https://devblogs.nvidia.com/cuda-dynamic-parallelism-api-principles/

```
void floatAtomicAdd (float *addr, float value){
    float old = value;
    float new;
    do {
        new = atomicExch(addr);
        new += old;
    }
    while ((old = atomicExch(addr)) != 0.f);
}
```

Listing 2. The code of our implementation of the atomic add. Note that any valued which could have been stored by other threads is preserved by the instructions in the while loop

The proposed solution has been implemented in `C++11` and CUDA 9.0, and runs on NVIDIA GPUs. We remark that our implementation doesn't require explicit synchronization between participating threads, allowing for further independent computations to be started on the GPU.

We implemented a function that, when treated as a black box, behaves as an atomic addition of a value to a memory location. In other words, the user perceives the concurrent access as atomic, leaving to the software the managing of conflicts in writing the data. Our implementation exploits the `atomicExch` C library function (whose signature is shown in listing 1) to write the new data to the memory location, while returning the old data stored at the pointed address.

```
float atomicExch(float *address, float val);
```

Listing 1. Signature of the core function needed for our implementation. The function writes value at address specified and returns the previous stored value

This informs us on the presence of previously written data, which shall be stored and added to the data we want to add. The main idea is to initially write 0 to the memory location, store any previous data and sum it to the data we intend to write. Then, we perform again an `atomicExch` with the new data (sum of old data and the value we desire to add), and check if the returned value is 0. There are two possible cases: either the returned value is 0, which means we have been the last thread to write on that address and we can terminate, or it is something else, which means that another thread managed to write that location before we could perform the second `atomicExch`. In the latter case, we must start looping and repeating the above steps until a 0 is returned. Obtaining a 0 as output value means that we successfully added our desired value to the location without losing any of the other threads' contributions. The code that implements this pattern is shown in listing 2.

V. Experimental Evaluation

To verify the effectiveness of our solution in the acceleration of the PageRank computation, we measured the execution time of two variants of the algorithm. The first variant doesn't

TABLE I
EXEC. TIME OF A BASELINE VERSION OF PAGERANK AND OF OUR OPTIMIZED IMPLEMENTATION

Implementation	Exec. Time (sec.)	Variance	Num. Iterations
Baseline	30	0.339	34
Optimized	6	8.80E-4	34

Times given in seconds. Convergence with error 10^{-6}

use any mechanism to avoid write conflict, and is used as a baseline. The second variant uses the techniques presented in listing 2.

In our experiments, we simulate the computation of PageRank values on a graph created from a subset of DBpedia 2016−10^5. DBpedia contains the same information as Wikipedia, expressed using the Resource Description Framework (RDF) format, from which it is easy to create a graph that represents the links between pages in Wikipedia. Our evaluation graph contains all the pages in Wikipedia, corresponding to 12 million vertices. We consider only a subset of the edges due to memory constraints on the GPUs at our disposal, resulting in about 50 million links. The calculation was carried out using a GTX 960, which concluded the calculation in 34 iterations in 30 seconds. The time measured goes between the first CPU call to the kernel to the moment the data returns in CPU. The initial data transfer is not considered, while we consider the final transfer of PageRank values from device to host.

In the baseline implementation, to carry out the matrix calculation of the PageRank multiplication, for each row of the PageRank vector a thread is launched which in turn launches other threads that deal with the multiplication of the data that really influences (non-zero elements of the matrix) the value of that particular PageRank position. This information is easily obtained from the CSR representation. This kind of configuration creates conflicts when writing the results, giving rise to the need of synchronizing all the threads during the computation.

Our solution solves this issue: we guarantee data consistency by repeatedly trying to write in the desired memory location, taking care not to lose the work done by other threads. This allowed us to reduce the convergence time of the PageRank algorithm from the execution time of the baseline code of about 30 seconds to about 6 seconds. In both cases we use as convergence criterion a threshold of 10^{-6}, measured as the difference of L2 norm of the PageRank vector between two successive iterations. Results are summarized in table I.

We also performed an additional test aimed at stressing the concurrent write to the same memory location by increasing the number of parallel threads that perform a given operation. In this test, we artificially compute an increasing size addition operation on a single memory location, from a minimum of 2 to a maximum of 1024 threads that concurrently try to write. The total number of threads in each test is obtained

[5]https://wiki.dbpedia.org/develop/datasets/downloads-2016-10

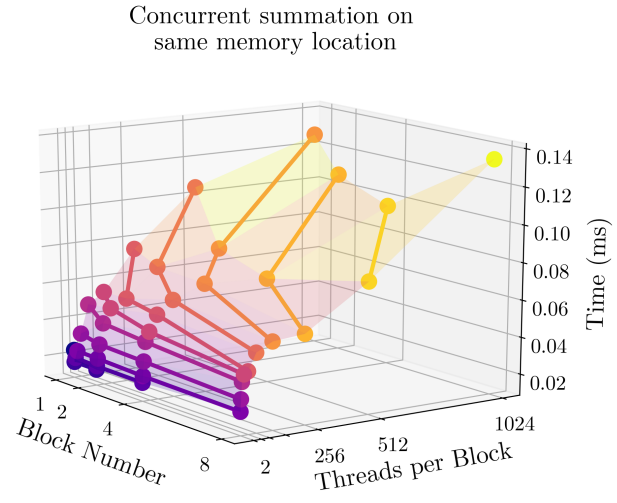

Fig. 1. Time required for concurrent summation on same memory location, as function of the number of concurrent accesses. The highlighted tridimensional lines show results for a fixed data size. Decreasing the number of blocks results in higher execution time, due to the larger number of conflicts caused by threads belonging to the same block.

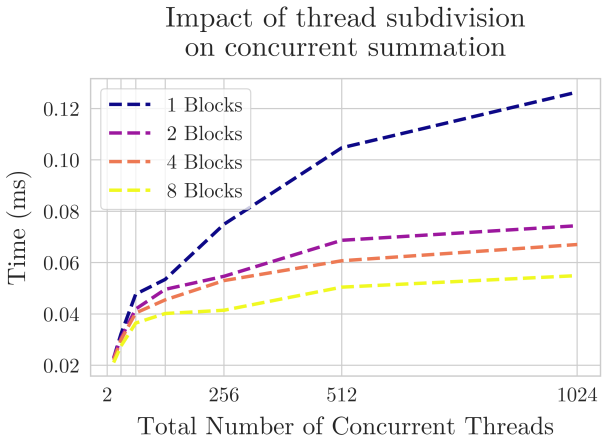

Fig. 2. Time required for concurrent summation on same memory location, as function of the number of concurrent accesses. Dividing the same number of threads across multiple blocks is highly beneficial to the execution time: 1024 threads divided across 8 blocks, instead of a single block, complete the computation more than twice as fast

by multiplying the number of blocks with the number of threads per block. Therefore a single test case starts a number of threads defined by the previous relation and measures the elapsed time between the start and ending of the computation of the thread of each task. The final execution time of a given configuration is obtained by averaging the execution time of 100 runs. The number of blocks chosen for the test ranges from 1 to 8. The number of threads per block, instead, ranges along powers of two from 2 to 1024, which is the maximum number of threads per block allowed by the GPU architecture. Results are shown in fig. 1 and fig. 2. The highlighted lines

represent computations done with a fixed input data size. It can be seen how changing the number of threads per block can have a significant impact on the overall execution time. The effect becomes more pronounced as the data size increases: using 8 blocks provides a speedup up to 3x, compared to using a single block that having roughly the same total number of concurrent threads.

VI. Conclusion

In this work, we have shown a software-based technique to prevent write conflicts in an implementation of PageRank. Our implementation is able to outperform a simple baseline in which conflicts are present, and can easily be applied to other algorithms. Even though threads try to sum the same data in an unordered way, it is unlikely to have more than a few threads trying to write to the same location, due to the sparseness of the connections in the graph. For this reason, and since our methodology does not require additional dedicated hardware solutions, we believe that it could represent a valid solution to the problem of concurrent adds in sparse graphs algorithms computation.

References

[1] L. Zack, R. Lamb, and S. Ball, "An application of googles pagerank to nfl rankings," *Involve*, vol. 4, 12 2012.

[2] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke *et al.*, "Mathematical foundations of the graphblas," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–9.

[3] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," 1998.

[4] M. Franceschet, "Pagerank: Standing on the shoulders of giants," *Communications of the ACM*, 2011.

[5] C. D. Meyer and A. N. Langville, "Deeper inside pagerank," *Internet Mathematics*, vol. 1, no. 3, pp. 335–380, 2003.

[6] W. F. Tinney and J. W. Walker, "Direct solutions of sparse network equations by optimally ordered triangular factorization," *Proceedings of the IEEE*, vol. 55, pp. 1801 – 1809, 12 1967.

[7] N. T. Duong, Q. A. P. Nguyen, A. T. Nguyen, and H.-D. Nguyen, "Parallel pagerank computation using gpus," in *Proceedings of the Third Symposium on Information and Communication Technology*, ser. SoICT '12. New York, NY, USA: ACM, 2012, pp. 223–230. [Online]. Available: http://doi.acm.org/10.1145/2350716.2350751

[8] G. P. E. U. Atish Das Sarma, Anisur Rahaman Molla, "Fast distributed pagerank computation," *Theor. Comput. Sci.*, vol. 561, no. PB, pp. 113–121, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2014.04.003

[9] B. Manaskasemsak and A. Rungsawang, "Parallel pagerank computation on a gigabit pc cluster," in *Proceedings of the 18th International Conference on Advanced Information Networking and Applications - Volume 2*, ser. AINA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 273–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977394.977506

[10] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *Proceedings of the 14th International Conference on High Performance Computing*, ser. HiPC'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 197–208. [Online]. Available: http://dl.acm.org/citation.cfm?id=1782174.1782200

[11] S. Franey and M. Lipasti, "Accelerating atomic operations on gpgpus," 04 2013, pp. 1–8.