Efficient load balancing techniques for graph traversal applications on GPUs

Federico Busato and Nicola Bombieri

Dept. Computer Science - University of Verona
`name.surname@univr.it`

Abstract. Efficiently implementing a load balancing technique in graph traversal applications for GPUs is a critical task. It is a key feature of GPU applications as it can sensibly impact on the overall application performance. Different strategies have been proposed to deal with such an issue. Nevertheless, the efficiency of each of them strongly depends on the graph characteristics and no one is the best solution for any graph. This paper presents three different balancing techniques and how they have been implemented to fully exploit the GPU architecture. It also proposes a set of support strategies that can be modularly applied to the main balancing techniques to better address the graph characteristics. The paper presents an analysis and a comparison of the three techniques and support strategies with the best solutions at the state of the art over a large dataset of representative graphs. The analysis allows statically identifying, given graph characteristics and for each of the proposed techniques, the best combination of supports, and that such a solution is more efficient than the techniques at the state of the art.

1 Introduction

Graph traversal refers to the process of visiting (i.e., checking or updating) vertices in a graph and is a core feature in many graph algorithms (e.g., BFS, SSSP, STCON). The high variability of graph characteristics over multiple dimensions such as, size, diameter, and degree distribution, makes the parallel implementation of graph traversal for GPUs a very challenging task.

Load balancing is a key aspect to face when implementing parallel graph traversal algorithms as it can strongly affect the performance of the overall application. Different solutions have been proposed to efficiently deal with such an issue during graph traversal on GPUs [9–11, 17, 18, 21]. Although they provide good results for specific graph characteristics, no one of them is flexible enough to be considered the most efficient for any input dataset. This makes each of these solutions, and in turn the higher level algorithm in which they are included, not efficient in several circumstances (in some cases, less efficient than the sequential implementation [16]).

This paper presents three different load balancing techniques for graph traversal applications on GPUs and, in particular, the key details of their architecture-oriented implementations. The paper also presents a set of features, which we call *support strategies*, which can be statically selected and modularly applied to the main balancing techniques to better address the graph characteristics.

The paper presents the results obtained by applying the different techniques to implement a common graph traversal algorithm (i.e., BFS) and how they impact on the overall performance. The analysis, which has been conducted on a large set of representative real-world and synthetic graphs, allows understanding the correlation between graph characteristics and load balancing configurations. The paper also shows how the performance of existing and widespread BFS implementations (*Gunrock* [21], *B40C* [18], and *BFS-4K* [3]) have been improved by substituting the original load balancing strategy with those presented in this paper, with and without the support strategies. The results show that the proposed solutions allow the BFS implementations to reach throughput up to 11,800 MTEPS on single GPU device, with speedups from 1x to 12.7x w.r.t. the original implementations.

The paper is organized as follows. Section 2 presents the background and related work. Section 3 presents the key details of the proposed balacing techniques and support strategies. Section 4 presents the experimental results, while Section 5 is devoted to the concluding remarks.

2 Background and related work

Different solutions for GPUs have been proposed in the last decade to improve load balancing aspects and accelerate graph traversal applications. They can be organized in three classes depending on the high-level strategy adopted to map GPU threads to graph vertices/edges.

Vertex-based mapping. Harish et al. [9] presented the first balancing solutions for BFS and SSSP applications, which target vertex parallelism to inspect every vertex in a graph at each *frontier* iteration [5]. Hong et al. [10] improved the previous approach by exploiting SIMD features of GPU architectures targeting irregular workloads for different graph applications (BFS, SSSP, and STCON). Jia et al. [11] evaluated and compared load balancing for vertex and edge parallelism to accelerate graph traversal in the context of centrality metrics (betweenness, graph, stress, and closeness). McLaughlin et al. [17] focused on the same techniques to accelerate betweenness centrality (BC) computation. All these balancing approaches do not require to maintain additional data structures, they involve very simple implementations but, on the other hand, they perform quadratic work. This makes the parallel implementations asymptotically slower than the sequential implementations. More recent research focused on efficient algorithms for linear-work graph traversal. Luo et al. [16] presented the first work-efficient BFS implementation based on single thread vertex-based mapping. Busato et al. proposed an advanced technique for BFS [3] and SSSP [4], which exploits tunable thread group size for vertex-based mapping and dynamic parallelism to process high-degree vertices.

Differently from all the approaches of this class, our first solution implements an optimized vertex-based mapping for linear-work graph traversal which relies on warp shuffle instructions and fully exploits coalesced memory accesses.

Scan-based mapping. Merrill et al. [18] presented a high-performance solution (B40C) which relies on a scan-based thread mapping for low-degree vertices and two additional techniques to handle mid-degree vertices at warp and block-level. Wang et al. [21] presented an optimized and flexible GPU graph library (Gunrock) that provides a high-level abstraction to reduce the developing effort

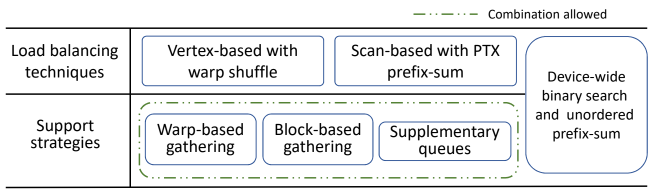

Combination allowed
Load balancing techniques
Vertex-based with warp shuffle
Scan-based with PTX prefix-sum
Device-wide binary search and unordered prefix-sum
Support strategies
Warp-based gathering
Block-based gathering
Supplementary queues

FIG. 1: *Overview of the load balancing techniques and support strategies.*

of graph primitive programming (Pagerank, SSSP, BC, etc.). The Gunrock library relies on the same thread mapping strategy adopted in B40C.
Differently from these approaches, our second solution includes an efficient scan-based technique that fully exploits the GPU shared memory and implements a low-latency PTX prefix-sum.

Binary search mapping. Bisson et al. [2] presented a BFS solution for distributed multi-node GPU platforms, which exploits a binary search algorithm to achieve perfect load balancing among all device threads. Khorasani et al. [12] presented a warp-based binary search strategy for BFS, SSSP, and PageRank. Davidson et al. [6] described and evaluated a merge-path search strategy[1] [8] at different thread hierarchy levels (i.e., warp, block, and device) in the context of SSSP. Gunrock also implements a device-wide merge-path search as an alternative load balancing technique.

Our third solution rely on a deeply revisited device-wide binary search mapping, which exploits three different and significant optimizations.

Differently from all the approaches in literature, we propose a set of strategies that can be modularly combined to support and improve any of the balancing technique.

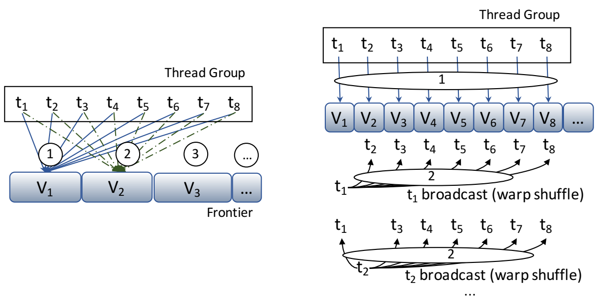
3 Load balancing techniques and support strategies

Figure 1 shows an overview of the main load balancing techniques considered and optimized in this work and the corresponding support strategies. The three main techniques (i.e., *vertex-based mapping with warp shuffle*, *scan-based mapping with PTX prefix-sum*, and mapping based on *device-wide binary search and unordered prefix-sum*) can be implemented in a mutual exclusive way in any graph traversal application to partition the workload and to map work items to the GPU threads. The support strategies can be applied singularly or combined to the selected load balancing technique.

3.1 The vertex-based mapping with warp shuffle

The *vertex-based* technique partitions the workload by directly mapping groups of threads to the edges of each frontier vertex. The left-most side of Figure 2 shows an example of the standard approach, in which the 8 threads of a thread group access to the vertex V_1 identifier in parallel and, then, each thread calculates the corresponding edge to be processed. Then, in sequence, the whole thread group moves to the other frontier vertices. The thread group size is set

[1] Merge-path search can be represented as a 2D binary search.



(a) Standard *(b) With warp shuffle*

FIG. 2: *Vertex-based mapping.*

depending on the average degree of the graphs (smaller warp sizes for graphs with lower average degrees). Nevertheless, in case of large thread group sizes, it may lead to many non-coalesced memory accesses during the frontier loading (8 accesses in the example), which in turn cause a strong loss of performance.

We propose an optimized version of such a vertex-based technique that combines *warp shuffle* instructions to the direct thread-to-edge mapping. The right-most side of Figure 2 shows the strategy main idea. Each thread accesses to a different frontier vertex and broadcasts the vertex identifier to the threads through warp shuffle. This increases memory coalescing at the cost of a minimum overhead involved by the warp-shuffle instructions. As an example, the memory accesses for the frontier loading in Figure 2(b) are reduced to 1 coalesced access.

3.2 The scan-based mapping with PTX prefix-sum

The *scan-based* load balancing technique is an alternative of the vertex-based mapping. Instead of directly mapping threads to edges, each thread organizes the own edge offsets in shared memory through scan operations. The proposed solution implements such scan operations at *warp-level* through an optimized prefix-sum[2]. Since such a procedure involves a large number of condition statements, which cause thread divergence, the proposed solution combines intrinsic *warp shuffle* instructions and *PTX instructions* [20] to implement *branch predication* (i.e., `<if(predicate) instruction>` C statements are replaced with `<@predicate instruction>` PTX instructions.

The proposed solution, thanks to the prefix sum result, allows exploiting the whole shared memory during the *frontier* propagation phase. It also adopts a warp-synchronous paradigm [19] to avoid any explicit synchronization.

A further optimization consists of a rewriting of loop iterations to exploit instruction-level parallelism (ILP). This is possible since the size of the shared memory is known at compile-time and each warp thread visits the same number of edges (except for the last iteration). The loops have been reorganized and unrolled to eliminate branches and iteration dependencies.

[2] Given an input sequence $a_1, a_2, \ldots, a_n$ the *prefix-sum* procedure computes the output as $a_1, (a_1 + a_2), \ldots, (a_1 + \ldots + a_n)$.

Algorithm 1 OPTIMIZED WARP-LEVEL BINARY SEARCH

```
Input: Sequence of values represented by the variable val
       of each thread; value to search: searched
Output: lower bound of searched
1: low = 0;
2: #pragma unroll
3: for ( $i = 1$;  $i \leq \log_2(\text{WarpSize})$;  $i{+}{+}$ ) do
4:     pos = low + (WarpSize $\gg i$); //$\gg$: compile time evaluated
5:     if (searched $\geq$ __shfl(val, pos)) then
6:         low = pos;
7: end
8: return low;
```

3.3 Device-wide binary search with unordered prefix-sum

The third load balancing technique relies on the *binary search* primitive to map the workload to the GPU threads. In the standard implementation, it provides the best load balancing in case of very irregular workloads (i.e., graphs with high standard deviation). Nevertheless, it involves a significant computation overhead, which makes the technique itself not suitable in case of regular workloads.

We propose an optimized version of the binary search that minimizes such an overhead and that fully exploits the GPU shared memory. The algorithm consists of three main steps:

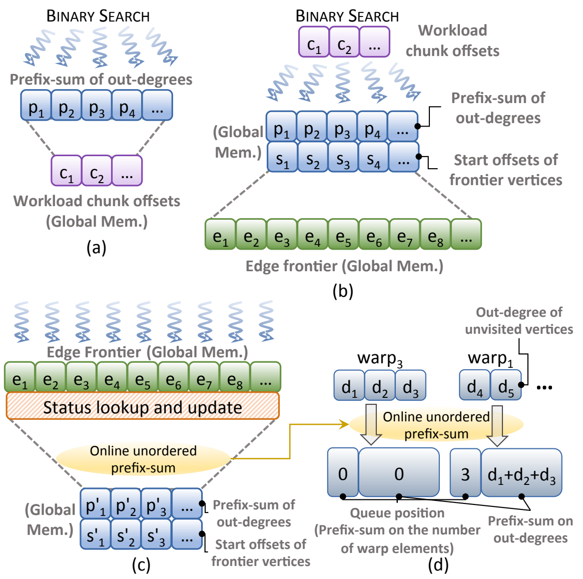
(1) It computes the prefix-sum of the out-degrees of the frontier vertices. It executes an *optimized binary search* to equally partition the workload over the thread hierarchy, i.e., at warp, block, and device-wide level.
(2) It stores and reorganizes the edge offsets in shared memory.
(3) It processes the shared memory elements in parallel.

The implementation strategy of the first step and, in particular, of the binary search over the thread hierarchy, is the key of the proposed technique performance. Algorithm 1 shows the pseudo-code of the proposed binary search *at warp-level*. The algorithm implements a variant of the standard procedure called *uniform binary search* [13], which relies on a lookup table. In our case, such a lookup table is implicit since the size of our input is a power of two. Thanks to the organization of the frontier information into shared memory, the binary search allows the following operations on the edge offsets to be performed through coalesced memory accesses. As for the *scan-based* technique, we implemented this technique by adopting the warp-synchronous paradigm to avoid barriers among warps of the same block.

The binary search *at block level* is similarly implemented to guarantee load balancing among threads of the same block.

The *device-wide* binary search guarantees equal workload among all threads of the GPU device. Given the prefix-sum of the out-degrees and the edge offsets of the frontier vertices, such a search consists of three main phases:

(A) A first kernel computes the binary search over the whole workload to uniformly partition the frontier edges among the grid blocks. Figure 3(a) shows an example, where p_i are the prefix-sum elements and c_i are the equally sized chunks of elements. The size of a workload chunk (i.e., the number of edges per chunk) is equal to the available shared memory per block.



BINARY SEARCH
Prefix-sum of out-degrees
p1 p2 p3 p4 ...
c1 c2 ...
Workload chunk offsets (Global Mem.)
(a)
BINARY SEARCH
c1 c2 ...
Workload chunk offsets
Prefix-sum of out-degrees
(Global Mem.)
p1 p2 p3 p4 ...
s1 s2 s3 s4 ...
Start offsets of frontier vertices
e1 e2 e3 e4 e5 e6 e7 e8 ...
Edge frontier (Global Mem.)
(b)
Edge Frontier (Global Mem.)
e1 e2 e3 e4 e5 e6 e7 e8 ...
Status lookup and update
Online unordered prefix-sum
(Global Mem.)
p'1 p'2 p'3 ...
s'1 s'2 s'3 ...
Prefix-sum of out-degrees
Start offsets of frontier vertices
(c)
Out-degree of unvisited vertices
warp3
warp1
d1 d2 d3
d4 d5 ...
Online unordered prefix-sum
0 0 3 d1+d2+d3
Queue position (Prefix-sum on the number of warp elements)
Prefix-sum on out-degrees
(d)

FIG. 3: *Overview of the device-wide binary search.*

(B) A second kernel applies a block-level load partition by following the steps of the binary search. Each block identifies the corresponding workload chunk by using the offsets calculated by the first kernel. This step generates the neighbour frontier starting from the edge offsets (Figure 3(b)).

(C) A third kernel generates all the information necessary to build the new frontier (Figure 3(c)). The kernel procedure executes the status lookup and update of the frontier elements, it removes previously visited vertices, and it computes an *online unordered prefix-sum*. In this particular case, the online procedure computes the prefix-sum of the out-degrees and of the number of warp elements at the same time. This allows avoiding double memory accesses to compute the prefix-sum offline through a specialized kernel procedure, which must load and store the degrees of the frontier vertices. The two informations are merged into a single value through the 64-bit `atomicAdd` instruction. We implemented a *second optimization* to discard vertices with out-degree equal to zero (for directed graphs) and equal to one (for undirected graphs) since they never contribute to the new frontier generation. Such an optimization is particularly useful in case of power-law graphs since they present a high number of *leaf* vertices (up to 20% in some instances).

The basic implementation of the device-wide binary search sets the workload chunk size proportional to the available shared memory per block. It is suitable for large frontiers, but it involves inactive threads in case of small frontiers. We implemented such a technique with a *third optimization*, which allows dynamically configuring the workload chunk size as follows:

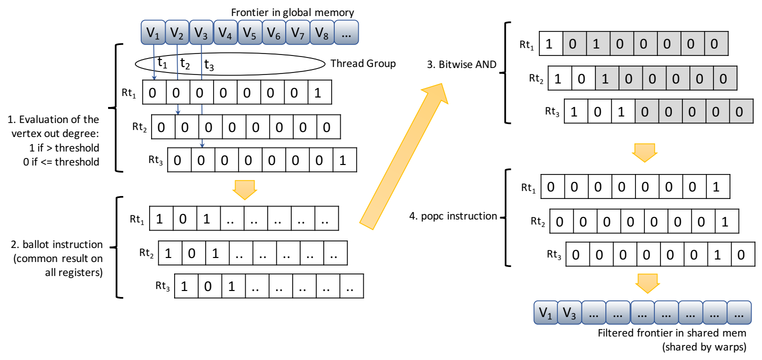


FIG. 4: *The 4-instructions binary prefix-sum*

$$\min \begin{cases} \left\lceil \dfrac{\textit{sum of out-degrees}}{\textit{\#resident threads}} \right\rceil \cdot \textit{block\_size} \\ \textit{shared\_mem\_per\_block} \end{cases}$$

The device-wide binary search is an atomic strategy. Because of its radically embedded structure, it cannot be combined with any support techniques.

3.4 Load balancing support strategies

The first support strategy is *warp-based gathering*, which aims at identifying heavy frontier vertices (i.e., vertices with out degree greater than a threshold) and, instead of mapping them to threads by following the main load balancing technique, it maps each of them to a whole warp of threads. It relies on a low latency *binary prefix-sum*, which is implemented by four hardware-implemented instructions (see Figure 4). Each warp thread saves the predicate result about the vertex out degree (1 if degree $>$ threshold, 0 otherwise) in the own register. Each register value (boolean 0 or 1) is then saved on the register bit corresponding to the thread id, in each thread register with a `ballot` instruction. After this step, all the registers of the same warp threads contain the same 32 bit value. It then computes the bitwise `and` between the register value and the thread id lower mask (e.g., the lower mask of thread 3 is `111`). Such a lower mask is efficiently obtained by reading a special register via a single PTX instruction. Finally, each thread counts the number of true values of the non-masked bits with the `popc` instruction and updates the corresponding register.

The result is a *filtered* prefix sum that gives information about all vertices of the frontier that must be processed by whole warps of threads (V_1 and V_3 in the example of Figure 4) instead of using the adopted basic mapping technique. The filtered prefix sum is stored in shared memory and, thus, it is shared by whole warps of threads, thus minimizing memory accesses. The vertices in the filtered prefix sum are processed by warps and, then, the remaining vertices of the original frontier are processed with the basic mapping approach (scan-based or vertex-based mapping).

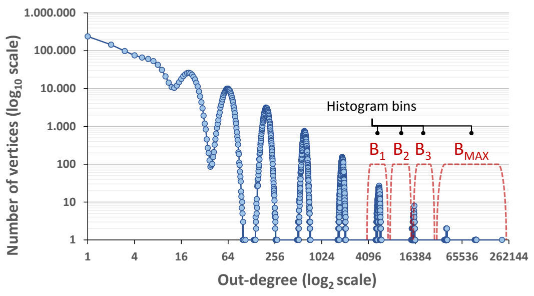


FIG. 5: *Example of supplementary queues applied to the kron_g500-logn21 graph.*

The threshold corresponds to the *warp size* (32 for the NVIDIA GPU devices adopted in this work) to fully exploit the parallelism at warp level.

The second support strategy is *block-based gathering*, which is similar to the warp-based one (with a threshold set to the *thread block size*), but with a substantial difference in the prefix-sum implementation. The block-level gathering relies on an *unordered* binary prefix-sum, which does not guarantee a strict ordering of the output while maintaining monotonic increasing values in the resulting sequence[3]. The parallel implementation of such an algorithm at block-level can take advantage of loose ordering to accelerate the computation. The unordered prefix-sum applies the same procedure of ordered variant at warp level but relies on *atomic operations* among different warps. In particular, each warp atomically updates a single value in shared memory for block-wide computation with the total sum of its values and getting back the previously stored value. Thanks to the *hardware-implemented* atomic operations in modern GPU architectures (i.e., from NVIDIA Maxwell on) the unordered binary prefix-sum allows achieving performance better than the conventional ordered scan-then-fan algorithm [22].

The third support strategy is *supplementary queues*, which is applied in graphs with maximum degree greater than half of the available device threads (e.g., 16,384 on the GeForce 980 GTX).

It aims at organizing the high degree vertices (vertices with out degree > $threshold_{SQ}$) of the frontier in different *bins*. Each bin holds vertices with sizes of the same (approximate) power of two (see the example of Figure 5). In particular, the i-th bin holds vertices with out-degree in the range $[2^{(b+i)}, 2^{(b+i+1)}]$, where 2^b identifies the base size, and b can be tuned by the user. Such a classification allows running a single kernel for the different bins, properly configured for the bin characteristics. In our tests, the total number of grid threads has been set equal to the lower bound of the bin ($2^{(b+i)}$) times the number of bin elements. This is motivated by the fact that the worst case involves at most two memory accesses among elements in consecutive queues. Finally, the bound value of the last bin is limited to the maximum number of resident device threads, since no more parallelism is possible for greater values.

[3] A possible output of the unordered prefix-sum is $a_3, (a_3+a_5), (a_3+a_5+a_1), \ldots, (a_1 + \ldots + a_n)$.

Graph	Category	U/D	V (M)	E (M)	Avg. degree	Std. deviation	Gini coeff.	Max degree	Avg. eccentricity
asia_osm	Road Network	U	12.0	25.4	2.1	0.5	0.08	9	36,626.7
europe_osm	Road Network	U	50.9	108.1	2.1	0.5	0.09	13	19,738.2
USA-road-d.USA	Road Network	U	23.9	58.3	2.4	0.9	0.21	9	6,418.6
hugebubbles-00020	Num. simulation	U	21.2	63.6	3.0	0.0	0.00	3	6,205.9
rgg_n_2_23_s0	Random Geometric	U	8.4	127.0	15.1	3.9	0.14	40	1,715.7
delaunay_n24	Structural	U	16.8	100.7	6.0	1.3	0.12	26	1,588.3
channel-500x100x100	Num. simulation	U	4.8	85.4	17.8	1.0	0.01	18	381.6
ldoor	Structural	U	1.0	47.5	49.9	11.9	0.13	78	161.4
nlpkkt160	Num. simulation	U	8.3	237.9	28.5	2.7	0.02	29	145.2
audikw_1	Structural	U	0.9	78.6	83.3	42.4	0.23	346	61.8
circuit5M	Circuit simulation	D	5.6	59.5	10.7	772.6	0.52	1,290,501	58.0
FullChip	Circuit simulation	D	3.0	26.6	8.9	23.1	0.35	2,312,481	38.3
cage15	DNA electrophoresis	D	5.2	99.2	19.2	5.7	0.17	47	37.3
indochina-2004	Social Network	D	7.4	194.1	26.2	215.8	0.74	6,985	31.0
soc-LiveJournal1	Social Network	D	4.8	69.0	14.2	36.1	0.72	20,293	14.3
soc-pokec-relationships	Social Network	U	1.6	61.2	37.5	59.5	0.62	20,518	10.2
er-fact1.5-scale23	Erdös-Rényi	U	8.4	200.6	23.9	4.9	0.12	53	7.8
hollywood-2009	Social Network	U	1.1	115.0	100.9	271.9	0.73	11,469	7.6
kron_g500-logn21	Kronecker	U	2.1	182.1	86.8	680.1	0.92	213,906	5.1

TABLE 1: *Graph dataset.*

$threshold_{SQ} = \frac{total\_device\_threads}{2}$ guarantees that at least half threads of the device are active when this technique is applied and, as confirmed by the experimental results, it avoids underutilization of threads and useless overhead.

4 Experimental Results

We conducted the analysis and the performance evaluation on a dataset of 19 graphs, which includes both real-world and synthetic graphs from different application domains. Table 1 presents the graphs and their characteristics in terms of structure (directed/undirected), number of vertices (V, in millions), edges (E, in millions), average degree, standard deviation, Gini coefficient, maximum degree, and average eccentricity (or BFS depth).

The graphs have been selected to be representative of a wide range of characteristics, including size, diameter, degree distribution (from regular to power-law). The graphs have been selected from the University of Florida Sparse Matrix Collection [7], the 10th DIMACS Challenge [1], and the SNAP dataset [15].

We ran the experiments on a NVIDIA GeForce GTX 980 device with CUDA Toolkit 7.5, AMD Phenom II X6 1055T 3GHz host processor, Ubuntu 14.04 O.S., and clang 3.6.2 host compiler with the -O3 flag. We ran all tests 100 times from random sources to obtain the average execution time t_{avg}. The traversal throughput is computed as E/t_{avg} for all tools and is expressed in MTEPS (million traversed edges per second).

To evaluate the efficiency of the proposed techniques, we measured how the performance of the best and most representative BFS implementations for GPUs at state of the art (*Gunrock* [21], *BFS-4K* [3], and *B40C* [18]) have been improved by substituting the original load balancing implementations with those presented in this paper. Figure 6 shows the results.

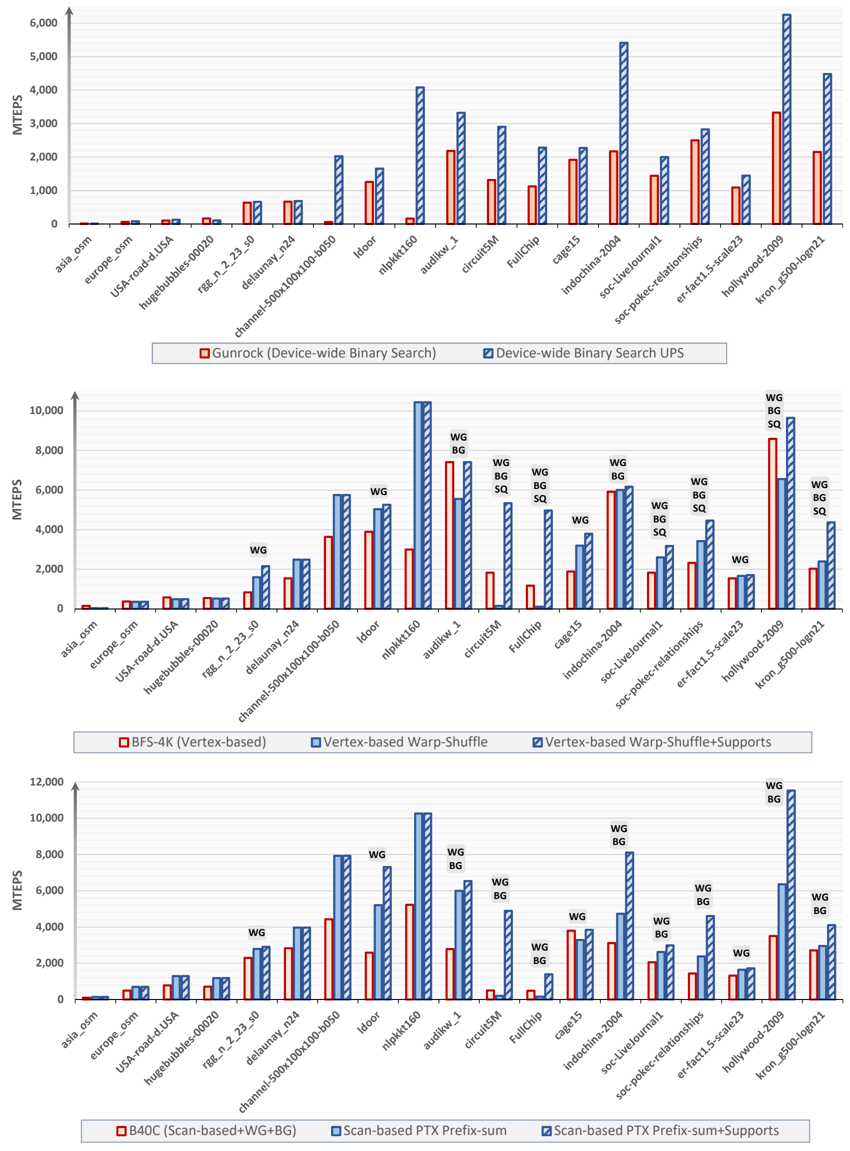


FIG. 6: *Performance comparison: Gunrock (upper-side), BFS-4K (middle-side), B40C (bottom-side). WG= warp-based gathering, BG= block-based gathering, SQ= supplementary queues.*

The upper side plot compares the performance of the original *Gunrock*, whose load balancing is set on a *device-wide binary search*, with the corresponding version in which we implemented the proposed device-wide binary search with unordered prefix-sum (UPS). As explained in Section 3.3, it was not possible to apply any support strategy due to the radically embedded structure of this load

Load balancing support	**Rules**
Warp-based gathering	max. degree > Warp size,
Block-based gathering	max. degree > Block size,
Supplementary queues	max. degree > Half device threads

Table 2: *Configuration table.*

balancing technique. However, the results show that applying only the proposed balancing technique allowed us to improve the BFS performance in all graphs, with speedups from $1x$ to $12.7x$.

Figure 6 - middle-side - compares the results of *BFS-4K*, which in the original version implements a direct vertex-based mapping strategy, with the corresponding version with *warp-shuffle*. First with no support strategies applied, and then with the supports applied in graphs satisfying the characteristics reported in Table 2. We observed that the performance of the proposed optimized technique are substantially higher than the standard implementation in most cases. In graphs with very irregular degree distribution, the original *BFS-4K* is better as it takes advantage of additional load balancing techniques, such as dynamic parallelism, to alleviate the workload unbalancing. However, the proposed strategy properly combined with the load balancing support strategies selectively enabled depending on the graph characteristics (reported on the top of the bar), provides the best results in all graphs (from $1x$ to $4.2x$, and $0.9x$ in a single case).

Finally, the bottom-side plot compares the results of the original *B40C* implementation, which relies on a scan-based mapping strategy, with the corresponding version based on *PTX prefix-sum*, without and with the proposed supports. It is important to note that the original tool also implements a support (which is always enabled) comparable to the warp- and block-level gathering proposed in this work. The results show that, thanks to the highly optimized prefix-sum and the instruction-level parallelism technique, the proposed load balancing technique (with no supports) provides performance almost always better and up to two times faster than the original *B40C* implementation. The throughput is further improved by enabling the proposed support techniques (right-most bar of the plot), thus providing speedups from $1x$ to $9.7x$. Such supporting techniques significantly contribute to the graph traversal performance thanks to their new algorithms based on binary prefix-sum, which, differently from the standard version implemented in B40C, it allows avoiding sequential iterations over high-degree vertices.

5 Conclusions

This paper presented three load balancing techniques for graph traversal applications on GPUs and the most important details of their architecture-oriented implementations. The paper presented a set support strategies that can be statically selected and modularly applied to the main balancing techniques to better address the different graph characteristics. Experimental results have been presented to show how the performance of existing and widespread BFS implementations have been improved by substituting the original load balancing strategy with those presented in this paper, with and without the support strategies.

References

1. Bader, D.A., Meyerhenke, H., Sanders, P., Wagner, D.: Graph partitioning and graph clustering, 10th DIMACS implementation challenge workshop. Contemporary Mathematics 588 (2013)
2. Bisson, M., Bernaschi, M., Mastrostefano, E.: Parallel distributed breadth first search on the Kepler architecture. IEEE Transactions on Parallel and Distributed Systems 27(7), 2091–2102 (2015)
3. Busato, F., Bombieri, N.: BFS-4K: an efficient implementation of BFS for kepler GPU architectures. IEEE Transactions on Parallel and Distributed Systems 26(7), 1826–1838 (2015)
4. Busato, F., Bombieri, N.: An Efficient Implementation of the Bellman-Ford Algorithm for Kepler GPU Architectures. IEEE Transactions on Parallel Distributed Systems 27(8), 2222–2233 (2016)
5. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. MIT press (2009)
6. Davidson, A., Baxter, S., Garland, M., Owens, J.D.: Work-efficient parallel GPU methods for single-source shortest paths. In: Proc. of IEEE IPDPS. pp. 349–359 (2014)
7. Davis, T.A., Hu, Y.: The University of Florida sparse matrix collection. ACM Transactions on Mathematical Software 38(1), 1 (2011)
8. Green, O., McColl, R., Bader, D.A.: GPU merge path: a GPU merging algorithm. In: Proc. of ACM SC. pp. 331–340 (2012)
9. Harish, P., Narayanan, P.: Accelerating large graph algorithms on the GPU using CUDA. In: Proc. of IEEE HiPC. pp. 197–208 (2007)
10. Hong, S., Kim, S.K., Oguntebi, T., Olukotun, K.: Accelerating CUDA graph algorithms at maximum warp. In: Proc. of ACM PPoPP. pp. 267–276 (2011)
11. Jia, Y., Lu, V., Hoberock, J., Garland, M., Hart, J.C.: Edge v. node parallelism for graph centrality metrics. GPU Computing Gems 2, 15–30 (2011)
12. Khorasani, F., Rowe, B., Gupta, R., Bhuyan, L.N.: Eliminating intra-warp load imbalance in irregular nested patterns via collaborative task engagement. In: Proc. of IEEE Parallel and Distributed Processing Symposium. pp. 524–533 (2016)
13. Knuth, D.E.: The art of computer programming, vol. 3. Pearson Education (1997)
14. Kunegis, J., Preusse, J.: Fairness on the web: Alternatives to the power law. In: Proc. of ACM WebSci. pp. 175–184 (2012)
15. Leskovec, J., et al.: Stanford network analysis project (2010), `http://snap.stanford.edu`
16. Luo, L., Wong, M., Hwu, W.m.: An effective GPU implementation of breadth-first search. In: Proc. of ACM/IEEE DAC. pp. 52–55 (2010)
17. McLaughlin, A., Bader, D.A.: Scalable and high performance betweenness centrality on the gpu. In: Proceedings of the IEEE International Conference for High performance computing, networking, storage and analysis. pp. 572–583 (2014)
18. Merrill, D., Garland, M., Grimshaw, A.: Scalable GPU graph traversal. In: Proc. of ACM PPoPP. pp. 117–128 (2012)
19. NVidia Corporation: Kepler Tuning Guide (2014), http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html
20. NVidia Corporation: Parallel Thread Execution ISA (2014), http://docs.nvidia.com/cuda/parallel-thread-execution/index.html
21. Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., Owens, J.D.: Gunrock: A high-performance graph processing library on the GPU. In: Proc. ACM PPoPP. pp. 265–266 (2016)
22. Wilt, N.: The cuda handbook: A comprehensive guide to gpu programming. Pearson Education (2013)