Configuring Graph Traversal Applications for GPUs: Analysis and Correlation of Implementation Strategies with Graph Characteristics

Federico Busato, Nicola Bombieri
Dept. Computer Science
University of Verona - Italy
Email: name.surname@univr.it

***Abstract*—Implementing a graph traversal (GT) algorithm for GPUs is a very challenging task. It is a core primitive for many graph analysis applications and its efficiency strongly impacts on the overall application performance. Different strategies have been proposed to implement the GT algorithm by exploiting the GPU characteristics. Nevertheless, the efficiency of each of them strongly depends on the graph characteristics. This paper presents an analysis of the most important features of the parallel GT algorithm, which include frontier queue management, load balancing, duplicate removing, and synchronization during graph traversal iterations. It shows different techniques to implement each of such features for GPUs and the comparison of their performance when applied on a very large and heterogeneous set of graphs. The results allow identifying, for each feature and among different implementation techniques of them, the best configuration to address the graph characteristics. The paper finally presents how such a configuration analysis and set allow traversing graphs with throughput up to 14,000 MTEPS on single GPU devices, with speedups ranging from 1.2x to 18.5x with regard to the best parallel applications for GT on GPUs at the state of the art.**

1. Introduction

Graph traversal (GT) applications are fundamental primitives for graph exploration and analysis. The irregular nature of the problem and its high variability over multiple dimensions such as, graph size, diameter, and degree distribution, make the parallel implementation of GT for graphics processing units (GPUs) a very challenging task.

Different parallel implementations of GT for GPUs have been proposed to efficiently deal with such issues [1], [2], [3], [4]. 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 and, in some cases, less efficient than the sequential implementation [5].

This paper presents an overview of the most important *features* that characterize the parallel implementation of GT for GPUs. They include load balancing, frontier queue management, synchronization between GT iterations, and duplicate removing. For each feature, the paper compares the different *implementation techniques* at the state of the art (e.g., node-based mapping, scan-based, binary search for *load balancing* and vertex, edge, hybrid queues for the *frontier management*).

Then, the paper presents a performance analysis conducted on a very large set of widespread real-world and synthetic graphs to understand the correlation among GT features, their implementation techniques for GPUs, and graph characteristics. The results show that some widespread implementation techniques adopted to deal with specific GT features (e.g., binary search for load balancing) are the most efficient when considered singularly while not the best when combined to the others for building a complete GT application.

In addition, the analysis results show how to classify the best implementation techniques of each feature by considering average degree, Gini coefficient, and maximum degree of a given graph. To exploit such an information, the paper presents an implementation of a GT application that, given a graph, it can be statically configured to better address the graph characteristics. The paper shows the results obtained by applying such a configurable GT implementation on a set of representative and heterogeneous graphs. The proposed solution allows reaching throughput up to 14,000 MTEPS on single GPU device, with speedups ranging from 1.2x to 18.5x with regard to the best parallel GT solutions at the state of the art in all the analysed graphs.

The paper is organized as follows. Section 2 presents some background on GT for GPUs. Section 3 presents the GT features and their implementation techniques. Section 4 presents the configuration analysis. Section 5 presents the experimental results, while Section 6 is devoted to the concluding remarks.

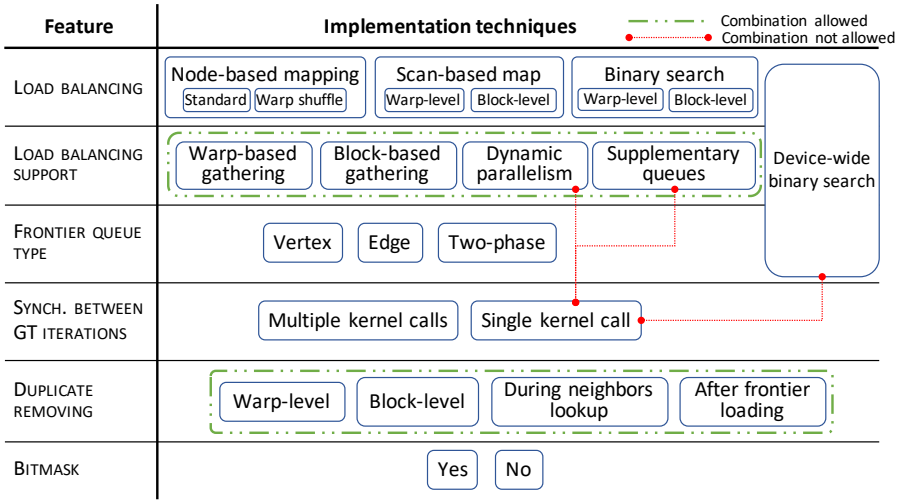

FIG. 1: *Overview of GT features and implementation techniques.*

2. Parallel graph traversal on GPUs

Graph traversal consists of visiting each reachable vertex in a graph from a given set of root vertices. Given a graph $G = (V, E)$ with a set V of vertices, a set E of edges, and a source vertex s, the parallel graph traversal explores the reachable vertices level-by-level starting from s. Algorithm 1 illustrates the main structure of such an algorithm, which, in the parallel version, is commonly based on *frontiers*. The algorithm expresses the parallelism in two `for` loops (lines 3 and 4). The first loop provides trivial parallelism by iterating over the frontier vertices. The second (nested) loop requires advanced parallelization techniques. This is mainly due to the fact that the loop aims at exploring the immediate neighbors of the frontier vertices, each of them requires a different number of iterations, and such a number depends on the out-degree of the vertices. As the corresponding sequential algorithm, the frontier-based GT algorithm shows linear work-complexity $\mathcal{O}(V + E)$.

Algorithm 1 FRONTIER-BASED PARALLEL GRAPH TRAVERSAL

Input: $G(V, E)$ graph, s source vertex
$\forall v \in V \setminus s : \ v.dist = \infty$
$s.dist = 0$
$Frontier_1 = s$
$Frontier_2 = \emptyset$

```
1:  level = 1
2:  while Frontier_1 ≠ ∅ do
3:      parallel for v ∈ Frontier_1 do
4:          parallel for u ∈ neighbor(v) do
5:              if u.dist = ∞ then
6:                  u.dist = level
7:                  INSERT(Frontier_2, u)
8:              end
9:          end
10:     end
11:     BARRIER
12:     level = level + 1
13:     SWAP(Frontier_1, Frontier_2)
14:     Frontier_2 = ∅
15: end
```

3. The GT features and their implementation techniques

Figure 1 shows the key features of a parallel GT application and the corresponding possible implementation techniques for GPUs. As underlined by the dashed green line, some techniques can be combined to implement a single feature. Others techniques are mutually exclusive, as underlined by the continuous red line.

3.1. Load balancing

The load balancing problem of the parallel GT can be addressed in four different ways. They rely on *node-based mapping*, *scan-based mapping*, *binary search*, or *device-wide binary search*.

The *node-based* technique [6] partitions the workload by directly mapping groups of threads to the edges of each frontier vertex. The thread group size can be set 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, which in turn cause a strong loss of performance.

In these cases the node-based technique can be improved by combining *warp shuffle* instructions to the direct thread-to-edge mapping. Each thread accesses to a different frontier vertex and broadcasts the vertex identifier to the threads through warp shuffle instructions.

The *scan-based* load balancing strategy [3], [4] is an alternative of node-based mapping. Instead of directly mapping threads to edges, each thread organizes the own edge offsets in shared memory through scan operations. This allows the threads to exploit data locality. A *warp-level scan* strategy can be adopted to exploit the whole shared memory during the *frontier expansion* phase and a warp-synchronous paradigm to avoid any kind of explicit synchronization. The *block-level scan* strategy follows the same steps, even though each iteration involves additional overhead for the mandatory thread synchronization through barriers.

The *binary search* technique [6] provides the best load balancing among all threads at *warp level*, at the cost of additional computation. With even more computation, such a balancing can be guaranteed at *block level*. As for the *scan-based* technique, this strategy can be implemented by adopting the warp-synchronous paradigm to avoid barriers among warps of the same block. In general, the *warp-level* binary search provides perfect load balancing only among warp threads, while the *block-level* technique guarantees uniform workload among warps of the same block.

The *device-wide binary search* strategy [4] is a special case of the previous solution and guarantees equal workload among all threads of the GPU device. It implements a revisited algorithm of the *merge-path strategy* proposed by Green et al. [7] in the context of merge-sort, which is strongly oriented to graph traversal. The device-wide binary search is an atomic strategy. Because of its radical structure, it cannot be combined with any of the other frontier queue or load balancing support techniques.

3.2. Load balancing support techniques

Warp-based gathering [3], [4] consists of collecting frontier vertices with out-degree greater than a threshold (*warp_size*) and, then, cooperatively processing their adjacency lists among warp threads.

It can be implemented through a low latency *binary prefix-sum*, which is computed on the condition upon the threshold of each thread (1 if *out-degree* $>$ *warp_size*, 0 *otherwise*). The prefix-sum result allows storing the vertices to be processed in consecutive locations of the shared memory. The *block-level gathering* [3], [4] applies the same idea at block level.

The *dynamic parallelism* [6] processes vertices with *very* large out-degrees. It provides benefits only with a fine tuning of the out-degree threshold. A wrong setting of such a value leads to a sensible overhead (caused by the dynamic kernel) that may compromise the overall application performance.

The *supplementary queues* technique [8] organizes the high degree vertices in different *bins*. Each bin holds vertices with sizes of the same (approximate) power of two. Such a classification allows running a single kernel for the different bins, properly configured for the bin characteristics.

3.3. Frontier queue types

The frontier queue represents the fundamental data structure of any work-efficient GT application. Such a queue stores either the vertices or the edges of the frontier, it allows an efficient neighbour exploration, and updates at each GT iteration. Many approaches in literature store the *vertices* to be visited at the next iteration in the frontier queue [3], [5], [6], [9], [4]. The amount of global memory accesses involved to maintain such a vertex queue is in the order of $2V$.

Two possible alternatives are the *edge queue* [3] and the *two-phase queue* [3]. The *edge queue* stores the neighbors of the vertex frontier, and it involves $2E$ memory accesses. The *two-phase queue* alternates the two representations, by involving $2V + 2E$ memory accesses.

In general, the *vertex queue* allows minimizing the global memory data movement but suffers from thread inactivity during the status lookup[1]. On the other hand, the *edge queue* involves more memory accesses, but it provides better performance for the status lookup phase. The *two-phase queue* guarantees high thread utilization at the cost of a high number of memory accesses.

3.4. Synchronization between GT iterations

Thread synchronization between GT iterations strongly impacts on performance in the exploration of large diameter graphs. Implementing the whole graph traversal in one single kernel call [6], [3] requires synchronizing all threads at device level without returning to the host. This procedures can easily lead to low symmetric multiprocessor utilization or significant overhead at each iteration if not properly implemented.

Thread synchronization can be managed through explicit *multiple kernel calls* [6], [4], i.e., one call per GT iteration. Even though it introduces overhead at each kernel call, which strongly impacts on the overall performance especially in graphs with large diameter[2], it has three advantages: (i) it allows flushing the GPU caches between kernel calls, improving the hit rate for status lookups; (ii) it provides more flexibility in the kernel configuration; (iii) it improves the device occupancy for each kernel. Indeed, differently from the global synchronization, it does not require including many different synchronization functions into a single kernel, thus limiting the usage of thread registers.

3.5. Duplicate removing

Linear-work graph traversal can cause concurrent visits of the same vertex neighbors by different threads.

1. The status lookup (e.g., updating of vertex distance) is the most expensive step of the algorithm due to the sparse memory accesses involved by the graph data structure.

2. In our GPU devices, we experimentally measured such an overhead equal to 15μs per kernel call, which leads to a 15 ms overhead in graphs with one thousand depth. This overhead is consistent with other recent GPU devices [10], [11].

This involves duplicate vertices in the frontier queue. Duplicate removing can be implemented by merging vertex and thread *ids* in a *single vector data*, store the result, and then recover the data. The duplicate vertices are progressively eliminated through multiple iterations and different hash functions in warps [6], [3], [4]. The same approach can be applied at block level and can be adopted at different phases of the GT iteration, including *during the neighbour lookup* [6] and *after the frontier loading* [3], [4].

3.6. Bitmask status lookup

The bitmap structure [3] aims at improving the efficiency of the status lookup phase. Thanks to its compact size, it generally fits in the L2 cache and allows the threads to retrieve visited/unvisited information of frontier vertices without accessing the global memory. On the other hand, the massive parallelism of GPU threads often leads to *false negatives* (unvisited vertices that actually have been already visited). This is due to the concurrent accesses of threads to the same memory locations.

4. The configuration analysis

We conducted the analysis and the performance evaluation of all the GT features and their different implementation techniques presented in Section 3 on three different graph databases: the University of Florida Sparse Matrix Collection [12], the 10th DIMACS Challenge [13], and the SNAP dataset [14].

We ran the experiments on a NVIDIA Maxwell 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. The GPU device consists of 16 SMs (2,048 SPs) capable of concurrently executing 32,768 threads. For a fair comparison, we did not perform any modification to the input graphs and we adopted the Matrix Market format (.mtx) representation. 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 the features and corresponding implementations and is expressed in MTEPS (million traversed edges per second).

It is important to note that we run all the possible combinations of the implementation techniques (which we call GT configurations) on each graph of the databases. For the sake of space, we report only the overall considerations, which are finally summarized in Table 1. For load balancing, the *node-based mapping* technique is well-suited to graphs with regular degree distribution. This is due to the fact that the direct mapping of threads to work items implemented through node-based mapping, differently from the other balancing techniques, does not uselessly introduce overhead of complex mapping computations.

Node-based mapping with *warp shuffle* provides the best throughput when such instructions involve at least half warp threads (16) and if there exists *enough parallelism* (double thread group size) during the exploration of vertex neighbours

The *scan-based* load balancing is well-suited to irregular graphs thanks to the thread cooperation implemented by such technique in shared memory.

We found that both *binary search* and *device-wide binary search* are never better than the other balancing approaches for traversing any graph. This is due to the significant overhead they introduce in the workload partition procedures. In particular, even though they are the most efficient techniques from the balancing point of view for very irregular graphs and very popular in literature [15], [16], [9], [4], we found that the best GT configurations for such graphs do not include them. It is worth noting that the *binary search* load balancing has speedups ranging from 1.1x (in regular graphs) to 9.7x (in irregular graphs) with respect to the same technique at the state of the art that does not rely on edge reorganization in shared memory and on the unordered prefix-sum.

Warp-level and *block-level gathering* provide performance benefits when applied to graphs in which the maximum degree is greater than their corresponding work sizes (warp and block size, respectively). Below these thresholds, the efficiency of the techniques are affected by thread inactivity by construction. The *supplementary queues* provide the best efficiency in graphs with maximum degree greater than half of the available device threads, which corresponds to a threshold of 16,384 on the GeForce 980 GTX. Such a threshold is motived by the fact that the device is well exploited when at least half threads are active.

We found that *dynamic parallelism* never performs better than the supplementary queues for any graph of the dataset. For this reason, it does not belong to the best GT configuration we identified for the analysed databases.

The *vertex queue* is the most efficient solution to represent the frontiers thanks to the low number of memory accesses. The *edge queue* well applies only when the available parallelism is low and in case of irregular workload as it reduces the thread inactivity. For this reason, the most efficient combination consists of adopting the *edge queue* combined with *node-based mapping* on graphs with low average degree and that present not uniform distribution.

The two-phase queue can alleviate the workload unbalancing when combined with simple techniques but, on the other hand, the high number of memory accesses strongly affects the performance. For these reasons, the two-phase queue does not belong to the best GT configuration for any graph.

The *multiple kernel calls* strategy provides the best performance in graphs with average eccentricity less than 100. After such a value, the amount of overhead

generated by each single kernel invocation eludes all the advantages provided by the strategy and makes the *single kernel call* a better alternative. Since the average eccentricity is not known a priori, we consider *average degree greater than 20 or Gini coefficient greater than 0.6* as a good heuristic for identifying graphs with average eccentricity less than 100.

We also found that the choice of the *duplicate removing* technique can be associated to the chosen load balancing technique. It reduces the redundant work in the frontier exploration by introducing additional computation. Removing duplicates *during the neighbour lookup* allows eliminating a high number of vertices, but the amount of introduced overhead completely eludes the advantages of the technique. For this reason, duplicate removing during the neighbour lookup does not belong to any *best* GT configuration. In contrast, duplicate removing *after the frontier loading* provides more efficiency at the cost of a lower number of eliminated vertices. In the same way, *block-level* removing is more effective than the *warp-level* procedure, but it requires synchronization barriers if used in conjunction with load balancing techniques that rely on shared memory. For this reason, the best combination is removing *after frontier loading* at *warp-level* with *scan* load balancing, while removing *after frontier loading* at block-level with *node-based* load balancing. Duplicate removing through *atomicCAS* operations completely eliminates duplicate vertices at the cost of a considerable overhead. It never performs better than the other duplicate removing strategies for any graph of the dataset.

The choice of the *bitmask* application depends on how the input graph has been built. We observed a 70%-99% usage of the whole bitmask in synthetic graphs after the whole graph exploration. This is due to the fact that such graph vertices are randomly enumerated and, as a consequence, neighbor vertices have not close identifiers. This prevents concurrent accesses of threads to the same memory locations during the frontier exploration. In contrast, real-world graphs, for which the vertex enumeration is implicitly done during the graph generation, present neighbor vertices with close identifiers. We measured a 20%-30% usage of the whole bitmask and a high number of *conflicts* during the frontier exploration in such non-randomly enumerated graphs. We claim that the bitmask technique is not suited to all graphs in which the vertex labeling follows the topological structure of the graph. In contrast, the bitmask provides positive speedups in synthetic graphs generated with random vertex labeling

Finally, from the large set of obtained data, we derived a *configuration table*, with the aim of matching the best (near-optimal) GT configurations and the characteristics of the graphs to be visited. To do that, we applied a *decision tree* for multiclass classification [17]. Table 1 summarizes the result of such a regression, which reports, for each GT feature, the graph characteristics considered to select the best implementation techniques.

Feature	Rules
Load balancing	- *Node-based 1*: avg. degree $<$ 5 and Gini coeff. $\geq$ 0.2 (edge queue), - *Node-based 4*: avg. degree $<$ 5 and Gini coeff. $<$ 0.2, - *Node-based 16 (with shuffle)*: avg. degree $>$ Warp size and Gini coeff. $<$ 0.3, - *Node-based 32 (with shuffle)*: avg. degree $>$ Warp size * 2 and Gini coeff. $<$ 0.3, - *Scan-based (warp-level)*: otherwise
Load balancing support	- *Warp-based gathering*: max. degree $>$ Warp size, - *Block-based gathering*: max. degree $>$ Block size, - *Supplementary queues*: max. degree $>$ Half device threads
Frontier queue type	- *edge queue*: avg. degree $<$ 5 and Gini coeff. $\geq$ 0.2 (Node-based 1), - *vertex queue*: otherwise
Synchronization between GT iterations	- *Multiple kernel calls.*: eccentricity $<$ 100 (avg. deg. $>$ 20 or Gini coeff. $\geq$ 0.6, heuristic) - *Single kernel call*: otherwise
Duplicate removing	- *After frontier load* at warp-level with *Scan-based* - *After frontier load* at block-level with *Node-based*
Bitmask	- Synthetic graphs and DNA electrophoresis

TABLE 1: *Configuration table.*

It is worth noting that average degree, Gini coefficient, and maximum degree are the necessary and sufficient information to correctly customize the GT for every analysed graphs.

Many rules are expressed in function of GPU device parameters (warp size, block size, etc.) or according to architecture-independent graph properties, such as the Gini coefficient. A fine-tuning of the GT application by considering also the characteristics of the specific GPU device is still possible, even though, by considering our preliminary results with different GPU architectures, it plays a minor role. It is part of our current and future work.

5. Experimental Results

We implemented *Helix*, a configurable version of a common GT application (i.e., Breadth-First Search). Thanks to the flexible and expressive programming model, *Helix* allows the user to switch among the different implementation techniques of each GT feature. We used the configuration constraints summarized in Table 1 to statically configure *Helix* by considering any single graph characteristics.

We run *Helix* on a set of graphs, which are representative of a wide range of characteristics, including size, diameter, degree distribution (from regular to power-law). The variability of such a dataset is essential to fully stress the proposed solution under very different input types. Table 2 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 (i.e., GT depth).

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 2: *Graph dataset.*

The Gini coefficient [18] measures the inequality among vertex degrees. It is complementary to the standard deviation information to express the graph irregularity and it is considered a clear and reliable alternative to the power-law. It is expressed in the range $[0, 1]$, where 0 indicates the maximum regularity among vertex degree (i.e., all vertices with equal degree), while 1 indicates the maximum inequality among vertex degree (i.e., there exists one vertex with degree equal to the total number of edges).

We compared, in terms of performance, *Helix* and the best and most representative BFS implementations for GPUs at the state of the art (B40C [3], Gunrock [4], BFS-4K [6]) and with a sequential CPU implementation for completeness. For all the state-of-the-art BFS implementations, we considered the best performance results obtained by trying any possible configuration (if available).

Figure 2 shows the results in terms of throughput (MTEPS) and the speedup of *Helix* over the different solutions. The results show that *Helix* is significantly faster than all the other implementations in all graphs. We observed that, in general, the throughput of the GPU implementations is strongly related to the average degree and the graph size.

All the GPU solutions provide lower throughput in the left-most graphs as these graphs do not allow the GT applications to exploit high-degree of parallelism during exploration. Despite the low average degree and the high eccentricity, *Helix* provides speedups from 3.8x to 43.5x with regard to the CPU implementation for the first four graphs of the dataset. This is due to the combination of the global synchronization strategy with an efficient load balancing technique. *Helix* has speedups ranging from 1.2x to 1.6x w.r.t. B40C for these graphs since this last implements an edge frontier queue and a slightly less efficient global synchronization. *Helix* has speedups up to 18.5x w.r.t. Gunrock as this last supports only host-device synchronization. It is worth to note that the lack of global synchronization in Gunrock translates into an execution time higher than the sequential CPU implementation for the *asia_osm* graph.

The results show a significant throughput improvement of *Helix* compared to the other GPU solutions in graphs with a very high maximum degree (*circuit5M, FullChip, kron_g500-logn21, soc-LiveJournal1, soc-pokec-relationships*). This is due to the efficient *block-level gathering* and to the *supplementary queue* technique. Indeed, B40C provides a technique to process such vertices only at block-level, BFS-4K applies the dynamic parallelism which has been proved to be less efficient than the supplementary queues, while the device-wide binary search technique implemented in Gunrock suffers from high overhead despite the perfect load balancing.

Finally, the 8-bit bitmask implemented in *Helix* significantly improves the performance for synthetic graphs (*kron_g500logn21* and *er-fact1.5-scale23*), which present full-random labelling.

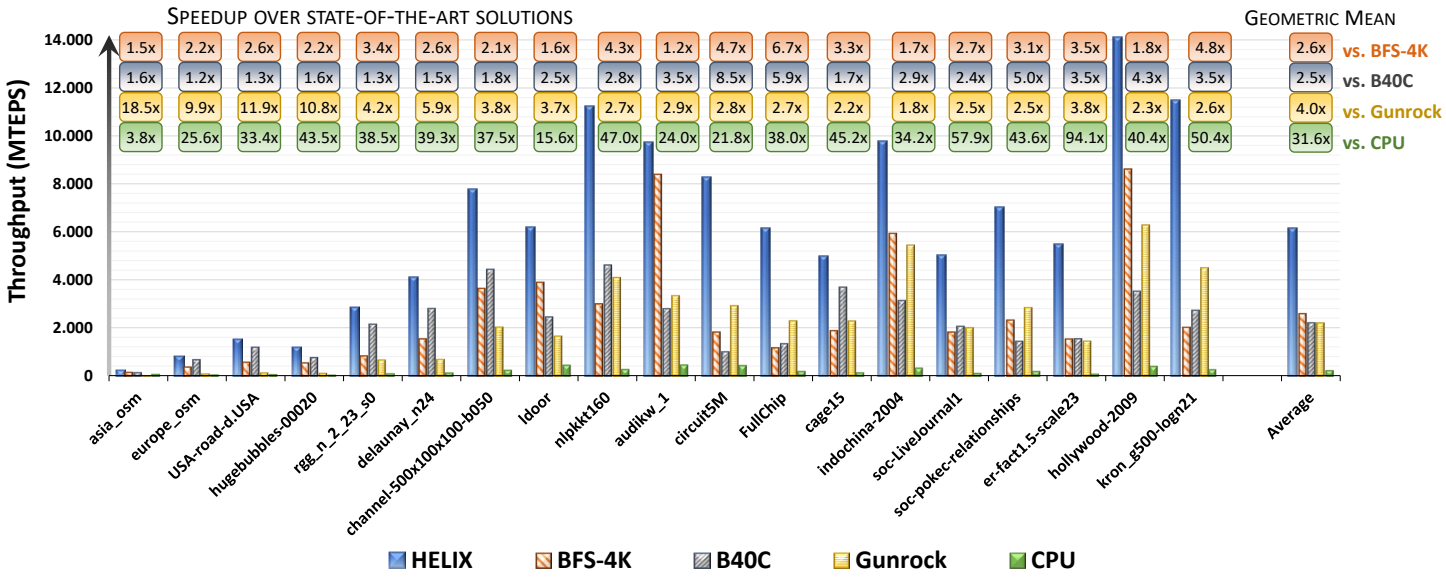

FIG. 2: *Performance comparison of Helix with the most representative implementations at the state of the art.*

6. Conclusions

This paper presented an overview of the most important features of the parallel GT algorithm. It summarized different techniques to implement each of such features for GPUs and compared their performance on a very large and heterogeneous set of graphs. The paper showed how the results of such an analysis allow identifying, for each feature and among different implementation techniques of them, the best configuration to address the characteristics of any given graph. The paper finally presented the performance of *Helix*, a GT application that implements the BFS and that is configured by exploiting the results of such an analysis. The experimental results show that *Helix* provides speedups ranging from 1.2x to 18.5x with regard to the best parallel BFS solutions for GPUs at the state of the art, with peak throughput of 14,000 MTEPS on a single GPU device.

References

[1] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proc. of IEEE HiPC*, 2007, pp. 197–208.

[2] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proc. of ACM PPoPP*, 2011, pp. 267–276.

[3] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proc. of ACM PPoPP*, 2012, pp. 117–128.

[4] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *Proc. ACM PPoPP*, 2016, pp. 265–266.

[5] L. Luo, M. Wong, and W.-m. Hwu, "An effective GPU implementation of breadth-first search," in *Proc. of ACM/IEEE DAC*, 2010, pp. 52–55.

[6] F. Busato and N. Bombieri, "BFS-4K: an efficient implementation of BFS for kepler GPU architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 7, pp. 1826–1838, 2015.

[7] O. Green, R. McColl, and D. A. Bader, "GPU merge path: a GPU merging algorithm," in *Proc. of ACM SC*, 2012, pp. 331–340.

[8] F. Busato and N. Bombieri, "Efficient load balancing techniques for graph traversal applications on GPUs," in *Proc. of Intern. European Conference on Parallel and Distributed Computing (Euro-Par)*, 2018, pp. 1–8.

[9] M. Bisson, M. Bernaschi, and E. Mastrostefano, "Parallel distributed breadth first search on the Kepler architecture," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 7, pp. 2091–2102, 2015.

[10] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 58.

[11] D. Tarjan, K. Skadron, and P. Micikevicius, "The art of performance tuning for cuda and manycore architectures," *Birds-of-a-feather session at SC*, vol. 9, 2009.

[12] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, p. 1, 2011.

[13] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "Graph partitioning and graph clustering, 10th DIMACS implementation challenge workshop," *Contemporary Mathematics*, vol. 588, 2013.

[14] J. Leskovec *et al.*, "Stanford network analysis project," 2010. [Online]. Available: http://snap.stanford.edu

[15] S. Baxter, "Modern gpu - nvidia research," 2013.

[16] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel GPU methods for single-source shortest paths," in *Proc. of IEEE IPDPS*, 2014, pp. 349–359.

[17] "Matlab: Statistics and machine learning toolbox - mathworks."

[18] J. Kunegis and J. Preusse, "Fairness on the web: Alternatives to the power law," in *Proc. of ACM WebSci*, 2012, pp. 175–184.