Breadth-First Search on Dynamic Graphs using Dynamic Parallelism on the GPU

Dominik Tödling
Graz University of Technology
Graz, Austria
dominik.toedling@student.tugraz.at

Martin Winter
Graz University of Technology
Graz, Austria
martin.winter@icg.tugraz.at

Markus Steinberger
Graz University of Technology
Graz, Austria
steinberger@icg.tugraz.at

***Abstract*—Breadth-First Search is an important basis for many different graph-based algorithms with applications ranging from peer-to-peer networking to garbage collection. However, the performance of different approaches depends strongly on the type of graph. In this paper, we present an efficient algorithm that performs well on a variety of different graphs. As part of this, we look into utilizing dynamic parallelism in order to both reduce overhead from latency between the CPU and GPU, as well as speed up the algorithm itself. Lastly, integrate the algorithm with the *faimGraph* framework for dynamic graphs and examine the relative performance to a Compressed-Sparse-Row data structure. We show that our algorithm can be well adapted to the dynamic setting and outperforms another competing dynamic graph framework on our test set.**

***Index Terms*—Breadth-first search, GPU, graphs, dynamic parallelism**

I. Introduction

Breadth-First Search (BFS) is a strategy for traversing graphs and can be used as a basis for solving various graph problems, such as single-source shortest path or finding connected components. It starts at a single node and proceeds to explore all other nodes in the graph in order of distance from the first node. Thus, first the starting node's neighbors are explored, then the neighbors' neighbors, and so on. The typical single-threaded algorithm uses a so-called *frontier queue* to remember which nodes to explore next. In each step it takes one item from the queue, searches its neighbors for any undiscovered nodes, and adds those to the end of the queue.

As the underlying graph domains are growing in size, holding tens of millions of vertices and millions to even billions of edges, the need for massively parallel hardware like the graphics processing unit (GPU) arises. Since this hardware is now in frequent use and also comparatively inexpensive, the GPU fits this problem domain perfectly. Additionally, since clock speed has hit the so-called *power wall* [17] while the transistor count keeps growing, a significant speedup can only be expected by exploiting the parallelism inherent in such applications. Achieving good performance on modern, massively parallel hardware like the GPU can be challenging. This is especially true when dealing with graphs with a wide ranging degree distribution, as naive approaches fail to balance the workload accordingly. Furthermore, the non-coalesced memory access pattern and the low arithmetic load are a challenging problem. As a result, BFS is the first benchmark in the Graph500 [10] list of the HPC graph community.

This paper presents the main challenges of parallel BFS and presents a complete algorithm that is competitive with other recent implementations. We combine and build upon approaches from previous work to arrive at our solutions for work efficiency and workload distribution and also investigate the efficacy of using dynamic parallelism to split the uneven workload across threads. Finally, we integrate the final algorithms with the dynamic graph framework *faimGraph* and examine the changes required to do so, as well as their performance impact. The algorithms described here are implemented using CUDA and may use a Compressed-Sparse-Row (CSR) data structure or the page-based adjacency layout of *faimGraph*. A graph's CSR representation consists of three arrays: One contains all the graph's edges, another the weights of all edges, and one more containing an offset into the edge array for each node. This means all outgoing edges of a node are always stored consecutively in memory, which is important to achieve efficiency on the GPU. As BFS does not take edge weights into account, only the edge and offset array are used. *faimGraph* stores its adjacency data on pages linked together into a linked-list of pages, resulting in consecutive memory accesses within pages but incurring some overhead due to the page traversal.

We show that our approach adapts well to both the static as well as the dynamic data structure, staying within a range of $5 - 10\%$. Our usage of dynamic parallelism, efficient frontier queues locally and globally as well as our classification scheme produces great results, outperforming a competing dynamic graph framework for a variety of problem domains.

II. Related Work

Related work on algorithms on graph data structures for the GPU can be roughly categorized into static (not supporting dynamic graph updates) and dynamic graph libraries as well as GPU-adapted implementations of different algorithms for BFS.

A. Static Graph Frameworks on the GPU

There exist a number of different static graph libraries on the GPU: *nvGraph* [14] (NVIDIA Graph Analytics library), offers implementations of widely-used algorithms, supporting

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

billion-edge graphs (using an NVIDIA Tesla M40 with 24 GB). *BlazeGraph* [18] presents a high-performance graph database built on its own domain-specific language *DASL*. *BelRed* [4] offer a library of software building blocks, addressing the challenges and manual effort required to set up graph applications. *GasCL* [3] presents a vertex-centric graph model, supporting the "think-like-a-vertex" programming model, built using Open Compute Language (OpenCL). *Gunrock* [19] is a CUDA framework for graph processing, building on highly optimized operators, trying to achieve a balance between applicability and performance.

B. Dynamic Graph Frameworks on the GPU

As a lot of problem domains build on highly volatile data sets, changing vertices as well as edges, a few notable dynamic graph frameworks were introduced in recent years. The first dynamic graph framework introduced was *cuSTINGER* [8]. *cuSTINGER* is a GPU-adaptation of *STINGER* [7] and its internal memory manager. Adjacencies are managed as individual arrays, enabling efficient memory access within an adjacency but requiring individual allocation procedures to increase/decrease a current allocation state per adjacency. Furthermore, memory cannot be efficiently reused within the system. *aimGraph* [21] removes this restriction by shifting the memory management to the GPU, requiring only a single allocation on the host and managing memory using a page-based allocation scheme. This allows for very efficient updates directly on the GPU, but introduces some page traversal overhead and memory is not reusable as well. *Hornet* [2] lifts this limitation by limiting the length of an adjacency to a power of two and managing such blocks in auxiliary data structures on the CPU, enabling efficient reuse of freed up blocks of memory. The adjacency itself is stored in an array-like format. *GPMA* [15] is a novel dynamic graph framework building on an adapted version of a Packed Memory Array, supporting efficient stream updates with implicit sorting. The data structure is allocated with a single allocation, but additional effort is required to maintain the data structure after updates and traversal is hampered by non-contiguous memory. *faimGraph* [20] is the newest addition to dynamic graph frameworks and continues with the efforts of *aimGraph* by enabling fully-dynamic updates, efficient memory-reuse directly on the GPU as well as algorithmic validation using Static Triangle Counting and PageRank. This allows for efficient updates as well as coalesced memory access within pages, but introduces a bit of overhead due to the page traversal required.

C. BFS Implementations on the GPU

BFS was first demonstrated on the GPU by Harish and Narayanan [9] in an exploration of using CUDA to accelerate common graph algorithms. Their implementation traverses the graph in levels, maintaining an array for visited status, one for frontier status, and one more for distance from the starting node. In each iteration each vertex is then assigned a thread, which checks its frontier status and updates the distance value for all its neighbors if it is in the frontier. Deng et al. [6] later showed a BFS algorithm based on their implementation of Sparse Matrix-Vector Multiplication which outperformed current GPU algorithms, but both of these algorithms performed more than the asymptotically optimal amount of work.

To solve this, Luo et al. [12] first demonstrated a multi-tier approach to constructing a frontier queue on the GPU, where queues are first assembled on a warp-level, then block-level, and finally on a global level. Once such a queue is finished, it can be used to examine only the current frontier nodes and their corresponding edges in each iteration. In their approach, warp-level queues require atomics, however, due to the fact that the hardware they were working with could only schedule 8 threads at a time, while a warp consists of 32, they were able to organize these accesses in such a way that simultaneously scheduled threads did not collide with one another. Once the warp-level queues are constructed, they then use a single thread to calculate the offset of the 8 warp-level queues that make up a block-level queue. Using those offsets each warp copies its queue into the block-level queue. Finally, a single *atomic increment* on a global queue pointer is used to reserve space for each block-level queue before copying.

They also introduced a strategy of hierarchical kernel management, where a different synchronization strategy is used depending on the frontier size. For frontiers up to the block size they use the simple block-level synchronization provided by CUDA, as well as maintain the frontier queue entirely in shared memory. Once that size is exceeded, they switch to a different strategy described by Xiao and Feng [22], which allows synchronization between blocks as long as there is a maximum of one block per multiprocessor. Only then is the synchronization provided by separate kernel launches used.

Later work done by Merrill et al. [13] shows a more comprehensive approach, which tackles both vertex- and edge-level parallelism, as well as performing an asymptotically optimal amount of work and being multi-GPU compatible. Their algorithm maintains an explicit vertex queue, as well as an edge queue. Instead of inspecting the neighbors of the current frontier-vertices immediately, they instead aggregate them into a global edge-queue. This queue is then filtered to remove previously-visited and duplicate vertices before either being immediately expanded again or placed into a global vertex queue depending on the frontier size. They also demonstrate the effectiveness of *prefix sum* as a way of determining per-thread offsets when building a global queue structure, which they use for both their vertex and edge queue.

In the gathering part of the algorithm (expanding the vertex queue into an edge queue), they start by assigning one vertex to each thread, but then have all threads with large nodes vie for control over the entire block by writing to the same address before synchronizing. The last thread to perform the write then has its vertex explored. This is repeated on the warp level for nodes larger than the warp width. Finally, adjacencies of small nodes are shared within each block by their assigned threads, copying them into shared memory, before jointly checking them. This results in efficient utilization of all threads.

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

A different algorithm was introduced by Liu and Huang [11] which also incorporates *bottom-up BFS* to save on the amount of edges that need to be traversed [1]. During the top-down and direction switching phase of their algorithm they do not produce a new frontier queue each iteration to use as input for the next, but rather scan the status array to generate the frontier queue at the beginning of each iteration. In the bottom-up phase they instead simply place all vertices that did not have any visited neighbors into the queue for the next iteration. This is possible because with bottom-up BFS the next frontier queue is always a subset of the current queue. Their approach to edge-level parallelization is also different, relying on classification of vertices based on their number of neighbors to determine how many threads to assign to each. They use four separate frontier queues to represent the classes, assigning either one thread, warp, block, or the entire grid to work on each node in a queue.

Lastly, Zhang et al. [23] investigate using dynamic parallelism to implement BFS. They use two kernels in their implementation, an outer kernel that iterated over all vertices to check their frontier status, and an inner kernel to iterate over the adjacency of a single vertex. This results in a very simple implementation which they can then expand with various experimental improvements. Their final algorithm utilizes the warp-level cooperation scheme described by Merrill et al. [13] and shows comparable performance to an implementation of Merrill's algorithm up to scale 19 on the Graph500 benchmark [10], but falls off sharply at larger scales.

III. Parallelizing BFS

When parallelizing BFS, there are two main points to consider: Work efficiency and workload distribution. As creating and maintaining a complex data structure on the GPU is often difficult, a naive implementation might forego an explicit frontier queue and instead simply check the frontier status of each node in each iteration. This can work well enough for shallow graphs, however, for large-diameter graphs it means that each vertex is examined many times. The other challenge becomes apparent when considering how many threads to assign to each adjacency. In a naive algorithm, one thread might handle one complete adjacency, however, this means that the largest node in each depth step slows down the entire iteration as everything has to wait for a single thread to finish before the next iteration can start. This section presents solutions to both of these problems, introducing an explicit frontier queue to keep track of the current search frontier as well as more efficient workload distribution.

A. The Frontier Queue

In order to produce a work-efficient algorithm, an explicit frontier queue is required so that only the nodes currently in the frontier can be examined.

As described in section II-C, Luo et al. [12] first demonstrated an approach to generate such a structure, however, our work uses an approach closer to the one described by Merrill et al. [13]. Each thread keeps track of discovered vertices in its registers and an *exclusive prefix-sum* is used to calculate the block-level offsets for each thread. The last thread in each block then reserves space in the global queue with an *atomic addition*, before each thread copies its thread queue directly into the global queue, as shown in Figure 1. This means only a single *atomic operation* is required per block and the discovered nodes are only copied once. There is also an additional cost, however, as it is now important to avoid duplicates when discovering new nodes. In order to achieve this, we replace the look-up in the depth array for a vertex's status with an *atomic compare-and-swap*. As these accesses are typically distributed randomly across the graph, the chance of multiple threads wanting to access the same node at the same time in this manner is low, meaning little to no performance impact on average. This process allows each iteration to produce a list of vertices the next iteration can then examine, eliminating unnecessary frontier status checks.

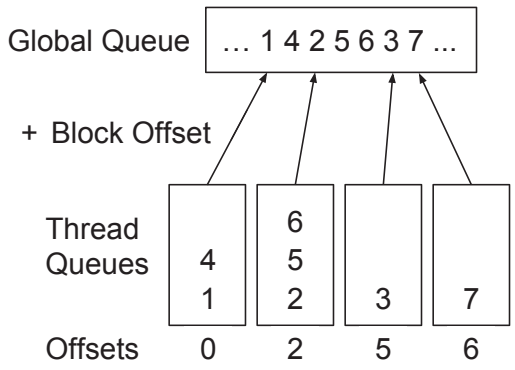

Fig. 1. Diagram of individual thread frontiers and how they are arranged into the global queue.

B. Distributing Workload

With an explicit frontier queue in place, it now becomes important to distribute the work of checking edges for undiscovered vertices evenly across threads. Not only is it necessary for good overall performance, but using the process described above, each thread can only check so many edges before it runs out of registers to store new vertices in.

Our first attempt at tackling this involved a technique called *dynamic parallelism*. In a CUDA program, kernels are typically launched from the host, that is, the CPU, however using dynamic parallelism, new kernels can be launched from within other kernels. This enables an approach where, whenever a thread is assigned to check a large adjacency, it delegates the work to a new kernel rather than checking it itself. To determine whether to launch a new kernel the adjacency size is compared to a fixed threshold value, beyond which a separate kernel is used. In theory, this is a relatively simple way to dynamically and fairly distribute workload, however, in practice, this results in very inconsistent results due to the large overhead involved in kernel launches. Compared to a naive approach it performs better on a variety of graphs but comes with its own weaknesses. Most importantly, for

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

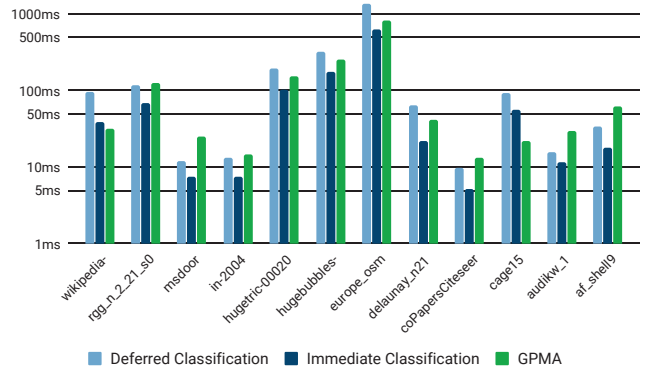

Fig. 2. Comparison between deferred and immediate classification and the algorithm implemented by the *GPMA* framework.

graphs that have a consistently high out-degree the kernel launches become the dominant performance bottleneck and the algorithm slows down considerably. In these cases the performance also becomes quite erratic, subsequent test runs often varying in timing by a factor of 2 or more, most likely due to the irregular scheduling pattern for the increased number of kernel launches. Clearly, a more refined scheme is required for how to distribute work across threads evenly. We chose to go with an approach similar to the one proposed by Liu and Huang [11], where frontier nodes are classified into different queues based on their size. Each of these queues is then handled by a different kernel with different parameters for how many threads to assign to each node. Similar to Liu and Huang, we sort vertices into one of four classes: Small (one thread), medium (one warp), large (one block), or huge (the entire grid). The precise thresholds are determined by the number of edges an individual thread should check. 4 worked well for for most graphs in our experiments, however some performed better with another value, sometimes by a quite considerable factor. The measurements presented here were all made with a value of 4 edges per thread. A significant difference to the aforementioned approach is that we perform all the steps directly on the GPU using dynamic parallelism, foregoing any CPU intervention.

For the small, medium, and large classes the number of threads launched per vertex is constant and equal to the threshold value, however for the huge class the necessary number of threads varies depending on the size of the node. It is possible to simply launch a separate kernel for each huge node and calculate the appropriate grid size for each, however for certain types of graphs - such as power law graphs - this can result in many thousands of kernel launches within a single iteration. To avoid this we decided to base the number of threads per huge vertex on the largest vertex currently in the huge frontier. A parallel reduction algorithm is applied to the sizes of these vertices to find the maximum and then a single kernel is launched with a configuration based on that value. This results in some unnecessary threads when the range of sizes within the huge frontier is large, however it still provides better performance than launching individual kernels.

In the algorithm presented by Liu and Huang, they rebuild the frontier queue by processing the complete status array in each iteration and so do the classification then. In our work we implemented two different versions, one with deferred classification as a separate pre-processing step between iterations, and one where classification takes place immediately upon discovering a new vertex. For deferred classification this means a total of five frontier queues are necessary: one has new vertices pushed onto it for the next iteration, and the other four are filled in the classification step. Each of those is then passed to a separate kernel for processing.

Immediate classification requires a total of eight queues as each class has one queue that is currently being processed and one that is being constructed for the next iteration. This also means that each thread has to keep four separate thread-level queues in its registers, further limiting the number of edges each thread can explore. As a further optimization we keep a set of shared variables within each block that signals whether any vertices of a certain size were discovered and only perform the queue-filling process for each class where this flag is set.

Figure 2 shows a comparison between these two approaches, as well as the implementation provided by the *GPMA* framework for working with dynamic graphs [15]. Their algorithm is based on the work done by Merrill et al. [13] and performs well on a wide variety of graphs. Due to the dynamic data structure of *GPMA* there is some overhead involved in traversing an adjacency that is not present using a CSR format, however the comparison serves well to show how different approaches show varying performance on different workloads.

The plot also shows quite clearly that immediate classification performs much better, often outperforming deferred classification by a factor of 2 or more with otherwise similar performance characteristics. All measurements shown in the graph were made with the same configuration for the number of edges per thread, however the ideal number varies from graph to graph and, for some, tweaking this parameter can lead to significant speedups.

IV. Integrating BFS with *faimGraph*

faimGraph is a fully-dynamic framework, supporting both vertex as well as edge updates efficiently. Edges are stored on pages linked together in a linked list, as can be seen in Figure 3. For this work, only the destination of an edge is relevant and all other edge data is omitted. This storage format is the biggest difference compared to competing approaches, which enables faster update rates directly on the device, but introduces additional challenges to algorithms such as BFS. Array-like adjacencies, like CSR, store all edges in a single array, where edges within a vertex adjacency reside in contiguous memory and adjacent threads can be served efficiently with coalesced memory access.

In order to achieve similar efficiency on a dynamic data structure with a page-based adjacency management, a few

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

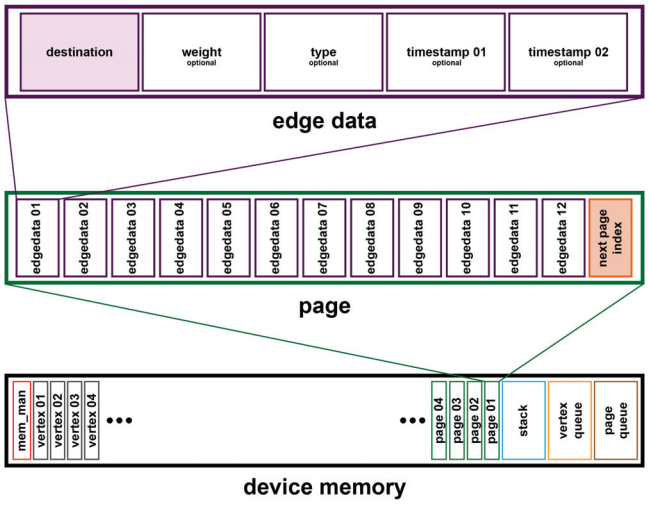

Fig. 3. Edges within *faimGraph* are stored on pages linked together by indices.

Name	Nodes	Edges	Depth
wikipedia-20070206	3,566,907	45,030,389	460
rgg_n_2_21_s0	2,097,152	28,975,990	1147
msdoor	415,863	19,173,163	127
in-2004	1,325,741	16,917,053	47
hugetric-00020	7,122,792	21,361,554	2800
hugebubbles-00020	21,198,119	63,580,358	4500
europe_osm	50,912,018	108,109,320	17346
delaunay_n21	2,097,152	12,582,816	564
coPapersCiteseer	434,102	32,073,440	26
cage15	5,154,859	99,199,551	500
audikw_1	943,695	77,651,847	55
af_shell9	504,855	17,588,845	472

TABLE I
GRAPHS USED TO EVALUATE PERFORMANCE OF DIFFERENT ALGORITHMS. SHOWS NUMBER OF NODES, EDGES, AND ITERATIONS REQUIRED TO COMPLETELY TRAVERSE THE GRAPH STARTING AT NODE 0.

adjustments to the algorithms are required. Exploring an adjacency with a single thread works nearly identical to array traversal. The framework provides edge data iterators which can be incremented until the last edge is reached, automatically resolving the page traversal in the process. Since the page size is fixed for an instance of *faimGraph*, ideally a single page can be read by threads within a warp at the same time. Hence, each work group (consisting of a warp) first calculates the page its target edge is located on, before one thread performs the page traversal to this page. Only then will the threads within a warp resolve the edge data iterator to its corresponding edge. This results in coalesced memory access for threads within a warp, with the same memory access patterns as with CSR. Nonetheless, some overhead is introduced by the need to first traverse the page lists to access the correct page, as can be seen in Figure 4. This overhead depends on the sparsity diversity within the graph and is on average around 10% for our test set, dependent on the average out-degree within the graph.

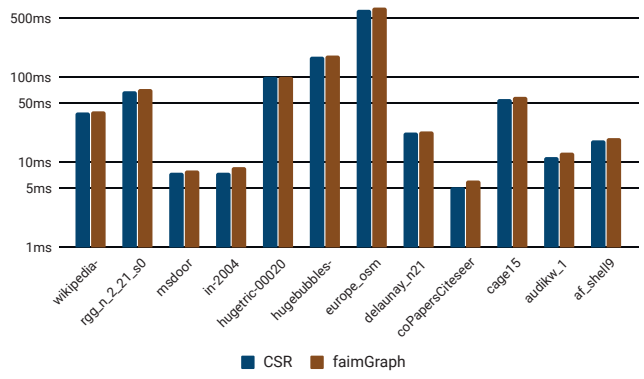

Fig. 4. Performance comparison between the classification version running on CSR and *faimGraph*.

V. EVALUATION

Table I shows the test set used to evaluate the performance of our algorithm. All graphs were taken from the SuiteSparse Matrix Collection [5] and the chosen graphs cover a wide range of sizes, diameters, and densities.

Figure 5 shows performance measurements for both classification versions for CSR and *faimGraph*, measured on an NVIDIA GTX 2080 Ti. Overall, our algorithm performs well for many different types of graphs, slowing down only for very large numbers of vertices in a single iteration. While the algorithm implemented by *GPMA* only ever launches a bounded number of threads we simply calculate the required number of threads based on the number of vertices in each class. This means for very large frontier queues the added scheduling overhead of handling so many threads slows down our algorithm. The graphs *wikipedia-20070206* and *cage15* are examples of this, the former having just a few iterations with hundreds of thousands of medium vertices, thousands of large vertices, and several hundred huge vertices. In total this results in well over 20 million threads launched, the largest portion of which are produced by the medium frontier.

Our experiments also showed that performance was also improved by running the main loop directly on the GPU, utilizing *dynamic parallelism*. This saves having to copy back the necessary information to determine whether the last iteration discovered any new vertices and is as simple as launching a one-thread kernel to handle managing the iterations. For a naive algorithm this speedup ranged from a few percent to up to 2 times faster in our measurements. This same approach was investigated before by Zhang et al. [23], however they found a slowdown of up to 44%. It is unclear why this is the

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

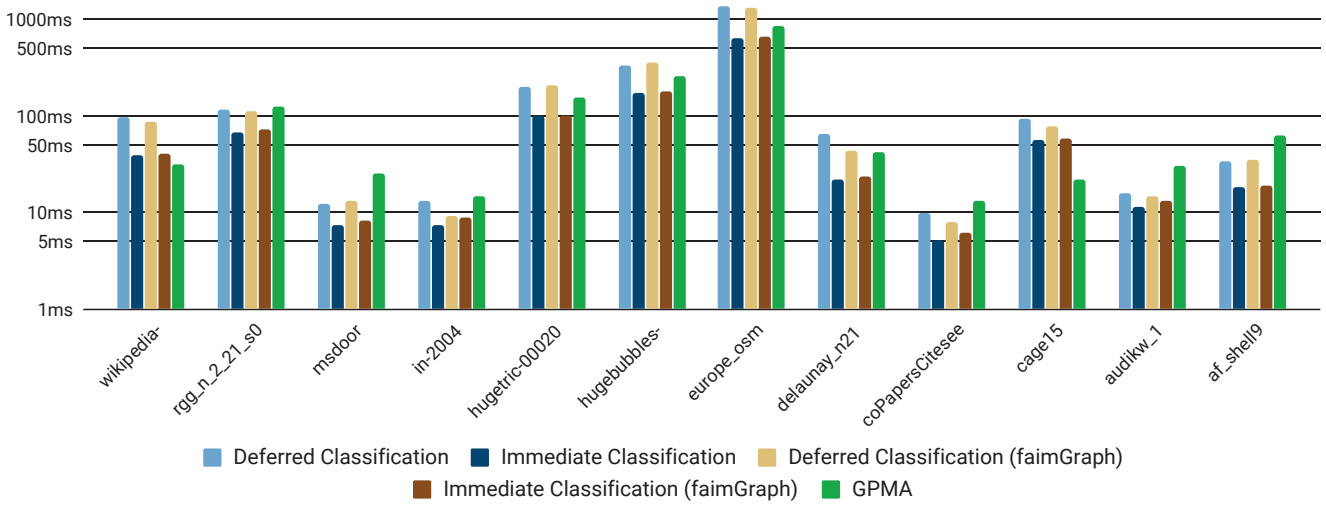

Fig. 5. Comparison of all variants described in this paper, as well as the algorithm implemented by *GPMA*.

case, however our results remained consistent across multiple devices and architectures.

A differentiating factor not mentioned so far is the memory footprint of each version. The naive version only requires an additional array with one entry for each node, while the dynamic parallelism version requires the same status array plus space for its frontier queues, one that is currently being processed and another that is being filled. As a single iteration could potentially have effectively the entire graph as its frontier, each queue is also allocated enough space for every node. The deferred classification version requires five queues in total, one for each of the four classes, as well as the queue for the next iteration. Immediate classification further increases this to eight queues, as each class requires a queue for current frontier vertices and those for the next iteration. This is still typically less than that of approaches maintaining an explicit edge queue, such as the one implemented by the *GPMA* framework. In our test set with 12 graphs from different application domains (including street networks, sparse matrices, citation graphs, triangulation graphs and more), our implementation outperforms the competing dynamic graph framework *GPMA* in 10 out of 12 cases. On average, the speed-up of *faimGraph* to *GPMA* is 78%, ranging from 0.37 to 3.1 times.

VI. Conclusion

We present an efficient BFS algorithm capable of outperforming competing implementations, covering solutions to both work efficiency and work distribution. Our classification-based approach to work distribution was adapted to immediately add newly discovered vertices to a frontier queue, which proved to be more efficient than performing classification in a separate step. Dynamic parallelism was examined for its ability to reduce unnecessary latency when running the main loop on the GPU and we discussed how using it purely for work distribution leads to mixed results, requiring more sophisticated classification for competitive performance. Finally, all algorithms were integrated with the *faimGraph* framework and the overhead introduced by its paged data structure found to be comparatively low, outperforming a BFS implementation on a competing dynamic graph framework, *GPMA*.

A. Future Work

While the final algorithm described here shows competitive performance on a wide variety of graphs, there are still several possible enhancements. While our approach can effectively skip costly round trips to the GPU between iterations of the BFS, avoiding the iteration-like processing altogether may increase performance further. To this end, a dynamic scheduling framework could help to naturally advance through the graph [16]. Another possible enhancement would be incorporating the idea of bottom-up Breadth-First Search demonstrated by Beamer et al. [1] and adapted for the GPU by Liu and Huang [11]. A hybrid approach that switches from top-down BFS to bottom-up BFS once a switching criterion is met can reduce the number of edges that need to be checked. As the algorithms were integrated with *faimGraph*, a natural next step would be to investigate partially updating a previous BFS result after a change to the graph. This could include tying the BFS implementation closer to the framework and observing changes to the adjacencies, which can further guide the exploration phase later on.

Acknowledgment

This research was supported by the German Research Foundation (DFG) grant STE 2565/1-1, and the Austrian Science Fund (FWF) grant I 3007. The GPU for this research was donated by NVIDIA Corporation.

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

REFERENCES

[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[2] F. Busato, O. Green, N. Bombieri, and David A. Bader. HORNET: An efficient data structure for dynamic sparse graphs and matrices on GPUs. In *2018 IEEE High Performance Extreme Computing Conference (HPEC '18)*. Georgia Institute of Technology, 2018.

[3] S. Che. GASCL: A vertex-centric graph model for GPUs. In *2014 IEEE High Performance Embedded Computing Conference (HPEC '14)*, 2014.

[4] S. Che, B. M. Beckmann, and S. K. Reinhardt. BelRedbel: Constructing GPGPU graph applications with software building blocks. In *2014 IEEE High Performance Embedded Computing Conference (HPEC '14)*, 2014.

[5] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[6] Y. Steve Deng, B. David Wang, and S. Mu. Taming irregular EDA applications on GPUs. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 539–546. ACM, 2009.

[7] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. STINGER: High performance data structure for streaming graphs. In *2012 IEEE High Performance Extreme Computing Conference (HPEC '12)*. Georgia Institute of Technology, 2012.

[8] O. Green and David A. Bader. cuSTINGER: Supporting dynamic graph algorithms for GPUs. In *2016 IEEE High Performance Extreme Computing Conference (HPEC '16)*. Georgia Institute of Technology, 2016.

[9] P. Harish and P. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *International Conference on High-Performance Computing*, pages 197–208. Springer, 2007.

[10] The Graph500 List. Graph500 BFS lists. https://graph500.org/?page_id=514. Accessed: 2019-02-03.

[11] H. Liu and H. H. Huang. Enterprise: Breadth-first graph traversal on GPUs. In *2015 SC-International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[12] L. Luo, M. Wong, and W. Hwu. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, pages 52–55. ACM, 2010.

[13] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.

[14] NVIDIA. nvGraph. https://developer.nvidia.com/nvgraph, 2016. Accessed 2107-05-12.

[15] M. Sha, Y. Li, B. He, and K. Tan. Accelerating dynamic graph analytics on GPUs. *Proceedings of the VLDB Endowment*, 11(1):107–120, 2017.

[16] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. Whippletree: Task-based scheduling of dynamic workloads on the gpu. *ACM Trans. Graph.*, 33(6):228:1–228:11, November 2014.

[17] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, pages 202–210, 2005.

[18] LLC SYSTAP. BlazeGraph. https://www.blazegraph.com/, 2017. Accessed 2017-05-01.

[19] 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 *ACM SIGPLAN Notices, vol. 50*, 2015.

[20] M. Winter, D. Mlakar, R. Zayer, H. Seidel, and M. Steinberger. faimGraph: high performance management of fully-dynamic graphs under tight memory constraints on the GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 60. IEEE Press, 2018.

[21] M. Winter, R. Zayer, and M. Steinberger. Autonomous, independent management of dynamic graphs on GPUs". In *2017 IEEE High Performance Extreme Computing Conference (HPEC '17)*. University of Technology, Graz, 2017.

[22] S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[23] P. Zhang, E. Holk, J. Matty, S. Misurda, M. Zalewski, J. Chu, S. McMillan, and A. Lumsdaine. Dynamic parallelism for simple and efficient GPU graph algorithms. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, page 11. ACM, 2015.

978-1-7281-5020-8/19/$31.00 ©2019 IEEE