Breadth-First Search using Dynamic Parallelism on the GPU

Dominik Tödling*
Supervised by: Martin Winter†

Institute of Computer Graphics and Vision
Graz University of Technology, Austria

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, three algorithms of varying complexity are implemented using the CUDA Programming Model for the GPU and are compared to one another on a variety of different, sparse 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, the same three algorithms are then integrated with the *faimGraph* framework for dynamic graphs and the relative performance to a Compressed-Sparse-Row data structure is examined. We show that our algorithm can be well adapted to the dynamic setting and outperforms another competing dynamic graph framework on our test set.

Keywords: GPU Programming, Breadth-First Search, Dynamic Parallelism

1 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. This results in every node and edge connected to the starting node being examined at least once.

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 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 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 [1] list of the high-performance-computing graph community.

This paper presents three BFS algorithms implemented using CUDA. We start with a naive version and discuss its major weaknesses before looking into how how to solve them. We combine approaches from previous papers 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 all three algorithms with the dynamic graph framework *faimGraph* and examine the changes required to do so, as well as their performance impact.

The CUDA Programming Model allows writing efficient GPU code close to the hardware in a manner very similar to traditional C++ code. The main functions to be executed on the device are so-called *kernels*, which are executed by many threads in parallel. When launching a kernel, a configuration for its *grid* is supplied, specifying how many blocks of threads and how many threads per block to launch, allowing millions of threads to be started simultaneously. Threads within the same block can be easily synchronized and share a faster kind of memory with one another, called *shared memory*. During execution, threads are also grouped into *warps* where all threads in the same warp always execute the same instructions in step.

The algorithms described here were all implemented for a Compressed-Sparse-Row (CSR) data structure. 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

*dominik.toedling@student.tugraz.at
†martin.winter@icg.tugraz.at

not take edge weights into account, only the edge and offset array are used.

2 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.

2.1 Static Graph Frameworks on the GPU

There exist a number of different static graph libraries on the GPU: *nvGraph* [15] (NVIDIA Graph Analytics library), offers implementations of widely-used algorithms, supporting 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* [5] offer a library of software building blocks, addressing the challenges and manual effort required to set up graph applications. *GasCL* [4] 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.

2.2 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* [9]. *cuSTINGER* is a GPU-adaptation of *STINGER* [8] 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* [3] 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* [16] 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.

2.3 BFS Implementations on the GPU

BFS was first demonstrated on the GPU by Harish and Narayanan [10] 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. [7] 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. [13] 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 it there.

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 once that size is exceeded is the synchronization provided by separate kernel launches used.

Later work done by Merrill et al. [14] 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, the 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 for all vertex sizes.

A different algorithm was introduced by Liu and Huang [12] which also incorporates *bottom-up BFS* to save on the amount of edges that need to be traversed [2]. 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, during which all unvisited vertices are examined for already visited neighbors, 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. [14] and shows comparable performance to an implementation of Merrill's algorithm up to scale 19 on the Graph500 benchmark [1], but falls off sharply at larger scales.

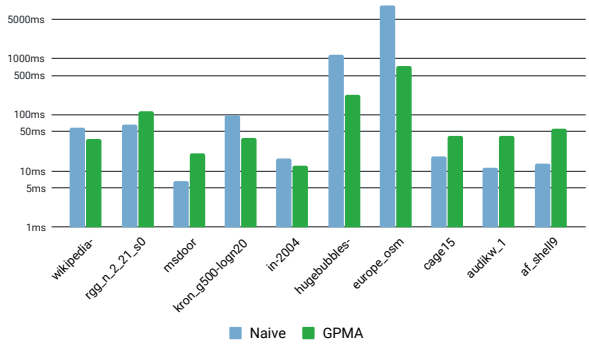

Figure 1: Performance comparison between the naive version of the BFS algorithm described and the algorithm implemented by the *GPMA* framework.

3 Parallelizing BFS

Sequential implementations of BFS are rather trivial but not worth of consideration for larger graphs, due to the inherent benefits of using a parallel approach. Given a certain root vertex, its corresponding adjacency is traversed, adding these vertices to an auxiliary data structure for the next iteration while noting the current depth. This process is repeated for all vertices in the auxiliary data structure, whereas each vertex can be added to this structure just once (to avoid cycles in the search). The algorithm is done once no new vertices can be added to the auxiliary data structure.

When parallelizing BFS, there are two possible areas to tackle: Exploring multiple vertices in parallel and exploring multiple neighbors of a single vertex in parallel. The initial algorithm presented here only takes advantage of the former. In the interest of simplicity of implementation it also foregoes an explicit frontier queue, as managing such data structures on the GPU is non-trivial. The only supporting data structure used is a status array which contains one entry for each vertex indicating at which depth, that is distance from the starting node, it was first discovered. The algorithm is then split into separate iterations, each of which performs a single depth step into the graph.

In each iteration one thread is launched for every node in the graph. Each thread then checks if its assigned node is currently marked as a frontier in the status array and only then proceeds to explore the neighbors of that node. When it finds an undiscovered neighbor, it simply marks it as a frontier for the next iteration by writing the next depth value into its status array entry. As all threads potentially discovering the same node in this step would write the same value to the status array, no synchronization between them is necessary.

Table 1 lists the graphs used to compare the perfor-

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
kron_g500-logn20	1,048,576	89,239,674	6
in-2004	1,325,741	16,917,053	47
hugebubbles-00020	21,198,119	63,580,358	4500
europe_osm	50,912,018	108,109,320	17346
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 1: The graphs used as a test set to evaluate performance of different algorithms. Shows number of nodes, edges, and iterations required to completely traverse the graph starting at node 0.

mance of different algorithms in this paper. All graphs were taken from the *SuiteSparse Matrix Collection* [6]. Figure 1 shows a comparison between the naive algorithm just described and the algorithm implemented by the *GPMA* [16] framework for working with dynamic graphs on the GPU, which is based on the work done by Merrill et al. [14]. As the *GPMA* algorithm is based on a dynamic data structure while the CSR structure used here is purely static, there is some overhead not represented in our measurements, however, it serves to highlight strengths and weaknesses of the algorithms described here.

As can be seen in Figure 1, our algorithm already performs quite well for certain kinds of workloads. Namely, graphs with a low diameter and a relatively uniformly low out-degree perform well, as the overhead of launching unnecessary threads is made up for by the low amount of work each thread has to perform in order to discover a single node. It slows down significantly, however, as soon as one of those conditions is not fulfilled. As each node is always only explored by a single thread, having a single large node in an iteration can result in all other threads waiting for one thread to finish its work. This is apparent in its performance on the *kron_g500-lgn20* graph, which is a so-called *kronecker graph* [11]. These graphs have comparable properties to real-world power-law graphs, which for this algorithm means it contains a few very large nodes. This graph in particular is of degree 20, meaning it contains 2^{20} vertices.

The most drastic difference can be seen for very large-diameter graphs such as *europe_osm*, which is a graph of the European street network. Its 50 million vertices are spread over 17346 iterations and each vertex only has a few neighbors each. In these cases the vast majority of threads launched in each iteration are unnecessary, resulting in a massive slowdown.

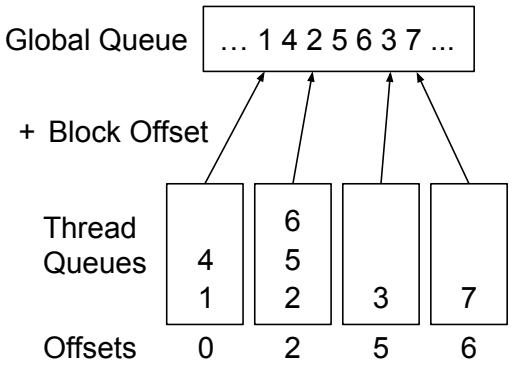

Figure 2: Diagram of individual thread frontiers and how they are arranged into the global queue.

4 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 2.3, Luo et al. [13] first demonstrated an approach to generate such a structure, however, our work uses an approach closer to the one described by Merrill et al. [14]. 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 2. 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.

5 Dynamic Parallelism

In CUDA programs, kernels are typically launched from the host, that is, the CPU, however, dynamic parallelism allows launching kernels from within other kernels. This allows for a simple improvement to the algorithm, where the main loop can be run directly on the GPU, rather than on the host. This saves extra latency introduced in each iteration by having to copy back the flag indicating whether any new nodes were discovered in the last iteration. For most graphs in the test set this resulted in roughly a 10-20% speedup, however, for some graphs it improved performance by up to a factor of 2. Zhang et al. [23] also ex-

perimented with using *dynamic parallelism*, however, they reported a slowdown of 1 to 44% in their measurements. It is not clear why this is the case, but our results were consistent across two devices with different hardware and compute capability (5.2 and 6.1 respectively).

We also attempted to use dynamic parallelism to remedy the second weakness of the initial algorithm, namely that it only ever explores a single node's neighbors with a single thread. This means that a single large node in an iteration can dominate that iteration's processing time, slowing down overall performance considerably. In order to deal with this we set a static threshold above which a thread would launch a sub-kernel to explore the node, rather than exploring it directly. When doing this, it is important to consider how to distribute the workload across the available threads. If threads within a warp access successive memory locations, the GPU can *coalesce* these memory accesses and serve them with potentially only a single memory transaction. To achieve such a pattern, each thread simply uses its *unique thread ID* (assigned sequentially, starting at 0) to calculate its starting offset into the adjacency array. In each step it then reads and processes an edge, before incrementing that offset by the number of threads working on the adjacency until all edges have been explored.

Putting this together with the frontier queue described in Section 4 results in the performance numbers shown in Figure 3. Most importantly, this algorithm shows comparable performance to the *GPMA* reference for the large-diameter graphs from before. However, it also shows slowdowns for others, due to its own weakness: The number of kernels launched can potentially become very large, resulting in large driver overhead. For graphs with a large median out-degree, it introduces a trade-off between the amount of work performed and the number of kernels launched. Since a thread's discovered nodes are stored in its registers, this threshold can not always be set high enough to avoid a crippling number of kernel launches. It is also important to note that this algorithm's performance is very inconsistent, varying by up to 2 times between consecutive runs, most likely due to different scheduling patterns by the hardware scheduler for a larger number of kernel launches.

6 Distributing Workload

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 [12], 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. In this work, either one thread, one warp, or one block is assigned to each node. As can be seen in Figure 4, this improves performance across the board while also not suffering from

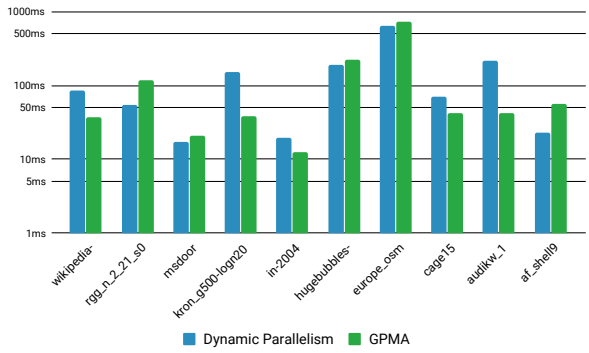

Figure 3: Performance measurements for the dynamic parallelism version of the algorithm.

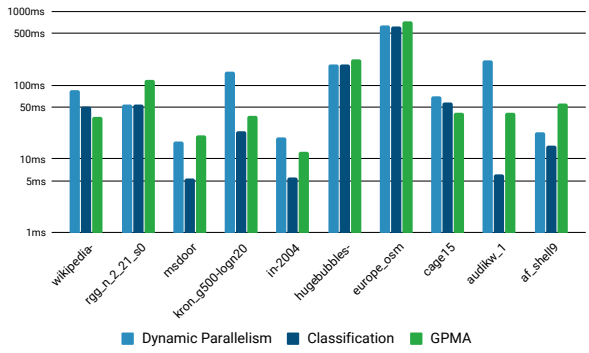

Figure 4: Performance measurements for the classification version of the algorithm.

the inconsistency of the dynamic parallelism-based algorithm. In particular, it can now handle all varieties of degree distributions without causing significant slowdown. Moreover, with this algorithm it is possible to tweak the number of edges assigned to each thread and the number of threads per block to better fit the type of graph being processed. With some experimentation this can yield an additional 10-20% speedup.

7 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 5. 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, as described in section 5.

In order to achieve similar efficiency on a dynamic data structure with a page-based adjacency management, a few 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 6. This overhead depends on the overall sparsity diversity within the graph and is on average around 10% for our test set, dependent on the average out-degree within the graph.

8 Evaluation

All performance measurements so far have only considered the processing time itself, leaving out the allocation of supporting structures as well as copying back the result from GPU to CPU memory. In our measurements, allocations made up a negligible fraction of total time spent and supporting structures can also be reused between runs if memory requirements have not increased. Figure 7, however, shows that simply copying back the resulting depth array can make up a substantial fraction of the total time

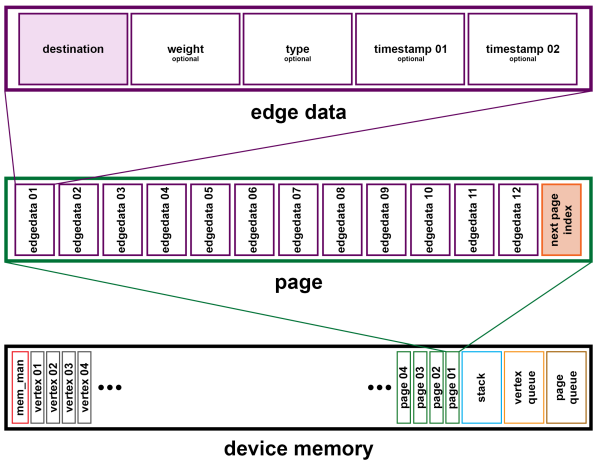

Figure 5: Edges within *faimGraph* are stored on pages linked together by indices.

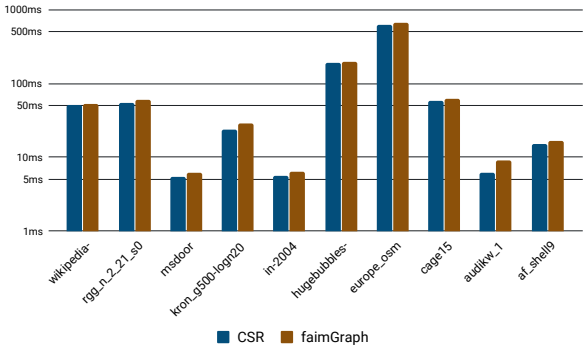

Figure 6: Performance comparison between the classification version running on CSR and *faimGraph*.

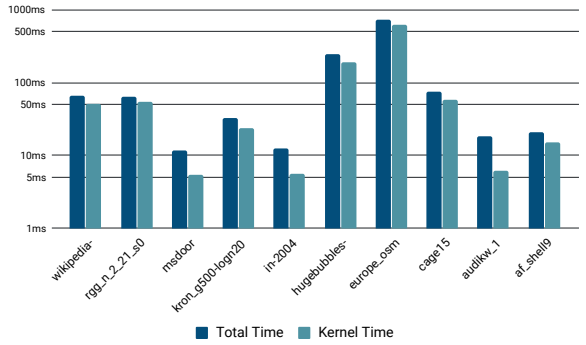

Figure 7: Comparison of total time used to allocate supporting structures, perform BFS, and copy the result from GPU memory back to CPU memory versus the time spent in the BFS kernel itself.

used, strongly incentivizing continued processing on the GPU in order to save this step.

Figure 8 shows the relative performance of all variants of BFS described in this paper, both using CSR and *faimGraph*, as well as the implementation provided by the *GPMA* framework and a simple, single-threaded CPU implementation as described in Section 1. It shows quite well how BFS' performance is very dependent on the type of graph and even very simple algorithms can outperform other, more sophisticated ones if the workload suits them better. For example, the *af_shell9* data set favors the naive implementation above all others, and even the CPU implementation stays competitive.

Furthermore, the impact of the adaption to *faimGraph* can be seen for all variants of the algorithm, although its effect is comparatively small and explained by the additional overhead of traversing the page-based data structure. The impact is generally higher for the naive and dynamic parallelism variants, where adjacencies are predominantly explored by a single thread.

A differentiating factor not mentioned so far is the memory footprint of each version. The naive version only re-

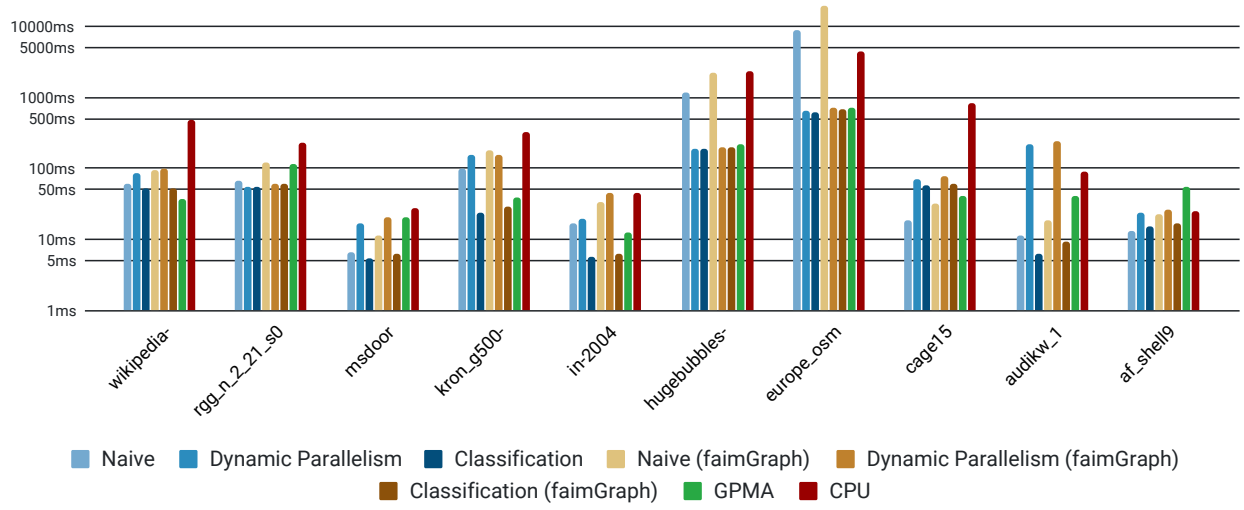

Figure 8: Comparison of all variants described in this paper, as well as the algorithm implemented by *GPMA* and a simple CPU implementation.

quires 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 classification version requires four queues in total, one for each of the three classes, as well as the queue for the next iteration. It would also be possible to consolidate the classification step into the exploration of the neighbors itself, however, this would increase the requirement to six total queues as each class would require its own working and next-iteration queue. This is still typically less than that of approaches maintaining an explicit edge queue, such as the one implemented by the *GPMA* framework.

9 Conclusion

In this paper, three different algorithms are presented in order to demonstrate the main challenges faced when adapting Breadth-First Search for the GPU. The initial algorithm performs poorly for large diameter graphs as well as graphs with consistently large out-degrees and was gradually improved to result in an algorithm capable of competitive performance across a variety of graphs. The importance of a frontier queue was demonstrated and approaches as to how to build one in parallel were discussed.

Dynamic parallelism was examined for its ability to reduce unnecessary latency when running the main loop on the CPU and it was demonstrated how attempting to use it to deal with large nodes can lead to mixed results. A classification-based approach to parallelizing individual adjacencies was adapted from Liu and Huang [12], and its effectiveness investigated. 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*.

9.1 Future Work

While the final algorithm described here shows competitive performance on a wide variety of graphs, there are still several possible enhancements. Concerning the performance of the algorithm itself, incorporating the idea of bottom-up Breadth-First Search demonstrated by Beamer et al. [2] and adapted for the GPU by Liu and Huang [12] can lead to significant speedups for certain types of graphs. It can save having to explore a significant chunk of a graph's edges by not searching all neighbors of the current frontier for undiscovered nodes, but searching the neighbors of undiscovered nodes for already discovered nodes. A hybrid approach starting with conventional bottom-down BFS and switching to bottom-up BFS once the frontier size reaches a certain threshold can potentially run several times faster than simply using the bottom-down approach.

As the algorithms were integrated with *faimGraph*, a natural next step would be investigating partially updating a previous Breadth-First Search 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, possibly reducing the amount of work to be done.

References

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

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

[3] 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.

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

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 133–145. Springer, 2005.

[12] 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.

[13] 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.

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

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

[16] 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.

[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.