XBFS: eXploring Runtime Optimizations for Breadth-First Search on GPUs

Anil Gaihre
UMass Lowell

Zhenlin Wu
UMass Lowell

Fan Yao
University of Central Florida

Hang Liu
UMass Lowell

ABSTRACT

Attracted by the enormous potentials of Graphics Processing Units (GPUs), an array of efforts has surged to deploy Breadth-First Search (BFS) on GPUs, which, however, often exploits the static mechanisms to address the challenges that are dynamic in nature. Such a mismatch prevents us from achieving the optimal performance for offloading graph traversal on GPUs.

To this end, we propose XBFS that leverages the runtime optimizations atop GPUs to cope with the nondeterministic characteristics of BFS with the following three techniques: First, XBFS adaptively exploits four either new or optimized frontier queue generation designs to accommodate various BFS levels that present dissimilar features. Second, inspired by the observation that the workload associated with each vertex is not proportional to its degree in bottom-up, we design three new strategies to better balance the workload. Third, XBFS introduces the first truly asynchronous bottom-up traversal which allows BFS to visit vertices for multiple levels at a single iteration with both theoretical soundness and practical benefits. Taken together, XBFS is, on average, 3.5×, 4.9×, 11.2× and 6.1× faster than the state-of-the-art Enterprise, Tigr, Gunrock on a Quadro P6000 GPU and Ligra on a 24-core Intel Xeon Platinum 8175M CPU. Note, the CPU used for Ligra is more expensive than the GPU for XBFS.

ACM Reference Format:
Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. 2019. XBFS: eXploring Runtime Optimizations for Breadth-First Search on GPUs. In *The 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19), June 22–29, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3307681.3326606

1 INTRODUCTION

BFS is the building block for a collection of graph algorithms, for example, the Strongly Connected Component (SCC) detection algorithm relies on forward and backward BFS to identify the SCCs from a directed graph [19]. Other algorithms, such as, Betweenness Centrality (BC) [27, 29] and subgraph matching [6, 16, 17] also rely heavily on BFS. Toward practical usefulness, BFS is also readily supporting a variety of applications, e.g., peer-to-peer network routing [39]. The importance of BFS is ultimately signified by the fact that Graph 500 [11], one of the most popular supercomputers ranking organizations, directly uses BFS to rank supercomputers from the world.

GPUs, with exceptional computing and memory throughput capabilities, are attractive platforms to accelerate graph traversal. In addition, GPU hardware is upgraded in a steady pace with improvements in both computing and memory capabilities. For instance, the new Volta V100 GPU [41] features 16 Tera-Floating Point Operations per Second (TFLOPS), 32 GB memory configurations, and 900 GB/s memory throughput, all of which are significant improvements over the last generation Pascal GPUs [32]. While one may concern about the *limited* memory capacity of GPUs, we find that the 32 GB memory space is adequate for a majority of the large graphs that is evaluated by the state-of-the-art distributed and external memory-based projects [25, 44].

Traditional BFS consists of the following three steps at each iteration:

(1) Load the neighbors of *frontiers* (i.e., Definition 2.1).
(2) Check the statuses of these neighbors.
(3) Store the first-time visited neighbors in frontier queue.

In order to maximize the bandwidth utilization, as well as circumvent the thread contentions on GPUs, two research problems draw particular attentions, that is, *frontier queue generation* and *workload balancing*. Specifically, first, instead of step (3) which introduces atomic operation, conventional methods [24, 42] scan the status array to generate the frontier queue. Note, status array indicates the "status" (i.e. the visited level) of each vertex. Second, to remedy the workload imbalance problem, existing work [24, 30, 42] either assigns various number of threads to vertices with respect to their degrees or uses single source shortest path (SSSP) to relax the synchronization requirement [5, 33] thus mitigates the penalty caused by workload imbalance.

1.1 Related Work and Challenges

In this section, we analyze the closely related work within the aforementioned three aspects, i.e., frontier queue generation, workload balancing and synchronous traversal, as well as outline the challenges that are observed in these directions.

Challenge I. Dissimilar BFS levels present different frontier queue generation requirements. Traditional "one frontier queue generation approach fits the entire graph traversal" philosophy falls short at several aspects. First, the hierarchical queue method [28] excels at levels with very few frontiers but will suffer from enormous space consumption and strided memory access at levels with tremendous frontiers. Second, the edge frontier filtering concept from B40C [30] and Gunrock [42] will also suffer from enormous space consumption, as well as duplicated frontiers at levels with tremendous frontiers. This also explains why Gunrock cannot deal with FR graph in Table 2. Third, the scan approach from Enterprise [24] is

HPDC '19, June 22–29, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6670-0/19/06…$15.00
https://doi.org/10.1145/3307681.3326606

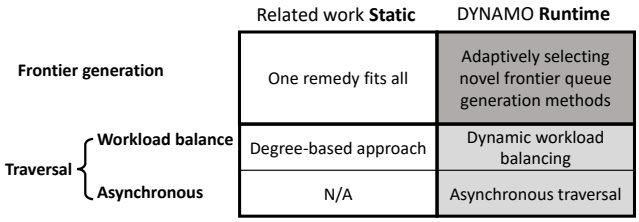


Figure 1: Related work vs. XBFS design.

designed for levels with large volume of frontiers but ends up with noticeable overhead at levels with low frontier count. It is worthy of noting that several work [15, 22] has already explored adaptive graph traversal concept, XBFS is distinct because we place adaptive concept only atop frontier queue generation – an important step for graph traversal.

Challenge II. The workload of each vertex, in bottom-up, is determined at runtime. A vast majority of existing efforts follow the rule – vertex degree indicates its associated workload. Towards that end, [14] divides thread warps into sub-warps with different number of threads to accommodate various frontiers. B40C [24, 30] extends this idea to assigning a cooperative thread array (CTA), to a frontier followed by a warp and thread. Eventually, [8, 20, 21, 31] propose to pre-calculate the workload of each frontier and divide them into segments in order to ultimately balance the workload. While the degree-workload association rule is correct in top-down BFS, the early termination in bottom-up BFS ultimately breaks this tie (as shown in Figure 9), putting all existing workload balancing optimizations in vein. GraphGrind [38] also suggests the break of this degree-workload tie for different graph algorithms but from a dissimilar point – various edges could yield different amounts of work.

Challenge III. SSSP-based asynchronous BFS tends to introduce exceeding amount of redundant work [12, 33]. Particularly, [5, 33] use SSSP algorithm [7, 40] to conduct BFS traversal thus various threads do not need synchronization. However, this design might update various vertices at multiple iterations, leading redundant vertex revisiting problem. And SIMD-X [26] finds that this redundant check is the culprit to make SSSP severely slower than BFS.

1.2 Contributions

Figure 1 depicts the radical distinctions between XBFS and the related work. XBFS identifies and addresses the problems that are faced by BFS which cannot be addressed in a static manner from existing endeavors. In particular, XBFS is, on average, 3.5×, 4.9×, 11.2× and 6.1× faster than the state-of-the-art Enterprise [24], Tigr [31] and Gunrock [42] on Quadro P6000 GPU and Ligra [35] on 24-core Intel Xeon Platinum 8175M CPU, respectively. Note, Quadro P6000 GPU [2] is cheaper than the CPU used for Ligra [18]. These speedups would not be possible without the following contributions.

First, XBFS adaptively exploits various either optimized or new frontier queue generation mechanisms to deal with the dynamics across the entire course of BFS traversal. To start with, we introduce four frontier queue generation methods, i.e., scan-free, single-scan and optimized double-scan, as well as no frontier queue generation methods. In particular, countering the traditional wisdom [24, 30, 42] which advocates to use prefix sum [13] to concatenate frontiers from various threads, XBFS finds out that atomic operation-based implementation is actually faster on recent GPUs (detailed in Figure 5). This has inspired our scan-free and single-scan designs. Further, our double-scan optimizes the design from Enterprise [24] to yield sorted frontier queue without strided memory access. Also, XBFS can surprisingly conduct one level traversal without the need of frontier queue generation. Eventually, we judiciously combine these designs based upon their fitness toward different levels of BFS (as shown in Figure 7).

Second, XBFS designs one static and two dynamic workload balance strategies to tackle the nondeterministic workload issue faced by the bottom-up BFS. This design is motivated by the novel observation that the actual workload (in bottom-up) tends to be both *small* and *not necessarily proportional to the degree of the vertex* (Figure 9) due to early termination [4]. Correspondingly, our static solution always assigns a single thread to a frontier regardless of the degree, which yields 1.4× speedup over the performance of first contribution. To further accommodate the unpredictability in the workload, our *thread-* and *warp-* centric approaches rely on atomic operation to schedule the tasks at runtime, which yields an extra 13.3% performance boost. And our dynamic workload balance brings another 59% speedup to XBFS.

Third, XBFS introduces a truly asynchronous bottom-up traversal which allows BFS to visit vertices from multiple levels at a single iteration with both theoretical soundness and practical benefits. In particular, in bottom-up BFS, the unvisited vertices of $i + 1$ level are a subset of level i, which suggests that we can actually visit unvisited vertices from multiple level at one iteration. Furthermore, on one hand, if one vertex does not have a neighbor from level $i - 1$, it surely will not be a level i vertex. On the other hand, we propose to track whether this vertex contains neighbors from level i. Once both conditions hold, we can surely visit this level $i + 1$ vertex one iteration earlier. Our evaluation shows that this design can identify 88% next level vertices, in addition to all the vertices from current level, with negligible overhead, yielding up to 20% performance gain over earlier contributions.

1.3 Paper Organization

The rest of this paper is organized as follows: Section 2 introduces the background of GPU hardware, direction-optimizing BFS and graph datasets. Section 3 overviews the framework of XBFS. Section 4 presents the detail of our adaptive frontier queue generation strategy. Section 5 talks about the runtime optimized bottom-up traversal. Section 6 presents the experimental setup and overall performance evaluations of XBFS. Section 7 concludes.

2 BACKGROUND

This section presents the essential background knowledge for XBFS. In particular, we briefly introduce the architecture of GPUs, direction optimizing BFS algorithm and the graph datasets used by this work.

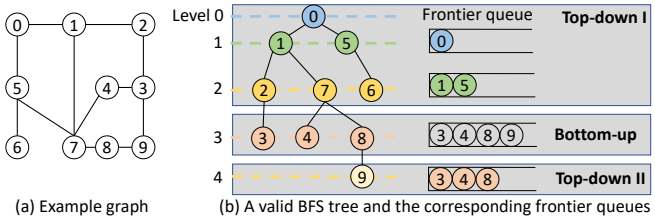


Figure 2: (a) An example toy graph used throughout this paper and (b) A valid BFS traversal tree with 0 as the source vertex from (a).

2.1 General-Purpose GPUs

The history of general-purpose GPUs dates back to 2006 when Compute Unified Device Architecture (CUDA) is initially introduced to this community. Thereafter, we observe a wave of emerging GPU microarchitectures, such as, Tesla, Fermi, Kepler, Pascal and Volta at 2007, 2009, 2013, 2016 and 2017, respectively. To the best of our knowledge, linear algebra, such as Linpack [10] and Lapack [3], inspires Tesla and Fermi. Energy efficiency is the major target of Kepler architecture. And Pascal and Volta are tailored for deep learning. Across these architecture generations, we observe a steady pace of enhancements in memory throughput, capacity and computing capability, reflecting the long-lasting merits of deploying graph traversal on GPUs. For more details regarding GPU history, we refer the readers to [43]. Below, we walk the readers through essential GPU hardware features with Pascal P100 GPU [32].

Streaming processors and threads. A P100 GPU consists of six Graphics Processing Clusters (GPCs), each of which encompasses ten Streaming Multiprocessors (SMXs). Every SMX has 64 single-precision (FP32) CUDA cores and four texture units. With 60 SMXs, P100 features a total of 3,584 single precision CUDA cores. During execution one GPU thread runs on one CUDA core and an SMX schedules a group of 32 consecutive threads, which is termed as a warp, in a Single Instruction Multiple Data (SIMD) manner. With two dispatchers of eight warp schedulers, an SMX can handle as many as 64 warps. A collection of warps formulates a Cooperative Thread Arrays (CTAs), or a block. And all CTAs, together, are called a grid.

GPU memory architecture. In P100, each SMX contains 65,536 registers and every thread can use up to 255 registers. Further, each thread block owns a fast on-chip shared memory that is available to all threads of the CTA whose shared memory retains the same lifetime. Specifically, each P100 SMX is equipped with a dedicated 64 KB shared memory and a separate L1 cache which means one SMX will always possess 64KB shared memory, avoiding shared memory/L1 space competition which exists in prior generations of GPUs. GPU also comes with a unified L2 cache and a global memory that are shared by all SMXs. Each memory controller is attached to 512 KB of L2 cache, and each High Bandwidth Memory 2 (HBM2) DRAM stack is controlled by a pair of memory controllers. The entire GPU includes a total of 4096 KB L2 cache and 12 GB global memory.

2.2 Direction-Optimizing BFS

Briefly, conventional BFS traverses a graph level by level, at each of which it checks the neighbors of all frontiers and marks those first times visited neighbors as the next level frontiers and puts them in the next frontier queue. Eventually, all the frontier queues formulate a valid BFS tree. Taking level 1 from Figure 2(b) whose frontiers are {0} as an example, BFS will check the neighbors of vertex 0, i.e., {1, 5}, and put the first time visited neighbors, that is {1, 5}, in next frontier queue. Iteratively, one will arrive at a BFS tree which could be the same as in Figure 2(b).

Recently, the direction-optimizing BFS [4] which comprises top-down and bottom-up directions prevails in graph traversal. In particular, top-down BFS is identical to traditional BFS – checking the neighbors of parent vertices to find children vertices, e.g., level 1, 2, and 4 in Figure 2(b). Whereas, bottom-up BFS checks the neighbors of unvisited vertices to find *potential* parent. For ease of illustration, we follow Enterprise [24] to define frontiers as follows.

Definition 2.1. **(Frontier)** At level i, v, which is a vertex from graph G becomes a frontier if

- Top-down BFS: v is visited at level $i-1$; or
- Bottom-up BFS: v remains unvisited until level $i-1$.

For instance, {3, 4, 8, 9} are unvisited vertices and stored in frontier queue in Figure 2(b). When working on each frontier, bottom-up BFS terminates the search once one parent vertex is found, which is called **early termination**. Using vertex 3 from Figure 2 as an example, bottom-up BFS will terminate the search once it finds 2 as its parent, leaving the remaining neighbors – {2, 4, 9} untouched.

2.3 Graph Datasets

Table 1: Graph specification.

Datasets	Abbr.	$\|V\|$	$\|E\|$	Size
DBpedia	DB	3,966,924	13,820,853	205MB
DBLP-author	DP	4,000,150	8,649,016	134MB
EnWiki	EW	21,504,191	266,769,613	7GB
Friendster	FR	65,608,357	710,869,285	11.9GB
LiveJournal	LJ	4,036,538	34,681,190	502MB
Orkut	OR	3,072,626	117,185,083	1.8GB
Trackers	TR	27,665,730	140,613,762	2GB
USpatent	US	3,774,768	16,518,947	256MB
Wiki-dbpedia	WK	18,268,992	172,183,984	2.7GB
Wiki-link	WL	12,150,976	378,142,420	6GB

Table 1 describes the ten graph datasets that are evaluated by XBFS. Note, these datasets cover a wide range of applications, such as social network (FR, OR, LJ, DP, US), webpages (WK, WL, EW), knowledge graph (DB), and advertisement (TR). Particularly, they are retrieved from either KONECT [9] or SNAP [36]. The vertex and edge counts ranges of these graphs are 3,072,626 to 65,608,357 and 8,649,016 to 710,869,285, respectively. For ease of supporting direction-optimizing traversal, we intentionally add a reverse to each edge, which doubles the edge count for every graph. Provided as edge tuples, we follow the tradition to transform them into compressed sparse row (CSR) format [24, 30, 42]. Following the same convention from Ligra [35], we use 4-byte and 8-byte to represent the vertex ID and offset (also known as begin position array) in CSR, respectively.

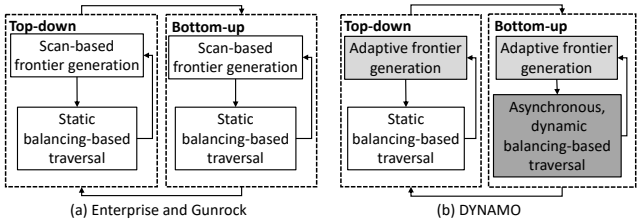


Figure 3: Workflow of state-of-the-art vs. XBFS.

3 OVERVIEW

As shown in Figure 3, XBFS advances the state-of-the-art projects [24, 42] in both frontier generation and traversal phases – shadowed boxes in Figure 3(b).

On one hand, instead of merely using a single frontier queue generation approach through the entire BFS traversal, we propose the adaptive frontier queue generation strategy to cope with the dynamics exhibited in various stages of BFS traversal. First, we revamp single-scan and double-scan, as well as introduce the new scan-free and no frontier queue generation approaches. Eventually, as shown in Section 4, XBFS judiciously alternates among these designs at runtime to arrive at the best performance.

On the other hand, we unveil our novel dynamic bottom-up optimizations. First, we introduce the novel observation that the workload of each vertex in bottom-up BFS is not necessarily proportional to its degree. Correspondingly, we design our static and thread- and warp- centric dynamic techniques as well as intra-warp working donation attempts to tackle this workload dynamics. Second, we enable the first, to the best of our knowledge, truly asynchronous traversal which can discover frontiers from more than one level with less workload than the traditional direction optimizing BFS algorithm [4], which is different from the SSSP based asynchronous traversal [33] that actually introduces more workload than the traditional BFS.

4 ADAPTIVE FRONTIER QUEUE GENERATION

This section discusses various frontier queue generation strategies. Subsequently, we unveil the judicious criteria to combine these mechanisms.

4.1 Top-Down Frontier Queue Generation

In top-down BFS, we employ three frontier queue generation approaches, i.e., scan-free, single-scan and no frontier queue generation. In particular, the former two are inspired by Observation 1 while the last one by Observation 2.

Observation 1. *On emerging GPU generations atomic operation-based frontier queue generation is faster than the mainstream prefix sum-based methods.*

Observation 1 counters the traditional wisdom that prefix sum [13, 37] is faster than atomic operation-based frontier queue generation [13, 24, 30, 34, 42]. In detail, both methods scan through a status array and put the frontiers into a queue, resembling the dotted area in Figure 4(b) – with the difference dwelling in how to enqueue the discovered frontiers. In particular, atomic operation enqueues frontiers atomically while prefix sum approach relies on prefix sum to figure out the offset for each frontier. Note, prefix sum is introduced by [30], in part, to avoid enqueuing frontiers atomically.

Figure 5 presents the speedup of atomic operation-based approach over the prefix sum-based counterpart. In this test, we randomly generate arrays at sizes of 1 - 512 million and use these two approaches to gathering frontiers. We find, on average, atomic operation-based option is 5.2×, 4.1×, 2.5× and 1.7× faster than the prefix sum based one on K80, P6000, Titan Xp and V100 GPUs, respectively.

Single-scan frontier queue generation extends the aforementioned observation to graph traversal. During traversal, each thread first loads the neighbors of the frontiers and updates the first time visited neighbors in status array. After the traversal is done, it scans through the entire status array, and puts those just updated vertices into next frontier queue with atomic operation.

Example 1. *Figure 4(b) exemplifies single-scan design with the toy graph from Figure 2 at level 2, where the frontiers are {1, 5}. Assuming the black and blue threads work on frontier 1 and 5, respectively, vertices {2, 6, 7} will have their statuses updated. Since repeated updating the status for vertex 7 will end up with a valid update, atomic operation is thus avoided here. Further, these two threads sequentially and consecutively scan the status array and atomically enqueue the frontiers into next frontier queue. Note, due to atomic operation, these frontiers may appear out of order, like {2, 7, 6} in Figure 4(b).*

Scan-free frontier queue generation goes further on exploiting atomic operation stemming from the fact that scanning through the status array (even merely a single scan) can be costly. In particular, scan-free method simultaneously generates frontier queue while conducting traversal. This method uses atomic operation to update the status of each neighbor. Once the status update succeeds which means this neighbor is indeed the first time visited, we further resort to atomic operation to enqueue this frontier into the next frontier queue. In summary, scan-free approach exploits atomic operation in two phases: status updates and frontier enqueuing.

Example 2. *Figure 4(a) explains the way scan-free method works when traversing the toy graph from Figure 2 at level 0. This method first loads the neighbors of 0 – {1, 5}, then atomically updates their statuses to 1s in status array. Eventually, this method relies on atomic operation to enqueue {1, 5} into the next frontier queue.*

Observation 2. *The frontier queue from the bottom-up traversal at level i encompasses the necessary frontiers for the top-down BFS at level $i + 1$.*

Proof. On one hand, bottom-up BFS puts the unvisited vertices in frontier queue and updates those ones which belong to current level. On the other hand, top-down level regards vertices visited in the preceding level as frontiers. As such, the frontier queue from the preceding bottom-up level consists of both unvisited vertices and the frontiers for the subsequent top-down BFS. This is also evident by level 3 and 4 from Figure 2(b). □

No frontier generation approach takes this inspiration to avoid frontier queue generation. Whereas, it is important to notice that this frontier queue also consists of unvisited vertices. We thus

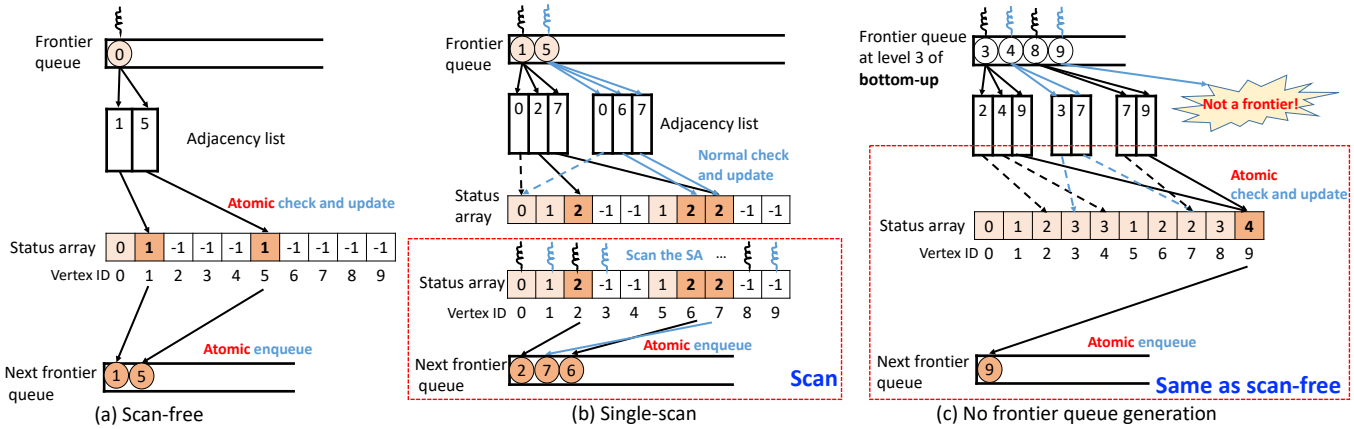


Figure 4: Traversing the toy graph from Figure 2 with various XBFS frontier queue generation approaches. In top-down the new (a) scan-free, optimized (b) single-scan and new (c) no frontier queue generation methods. Particularly, dark blue and light shadows represent next level frontiers and updated vertices, respectively.

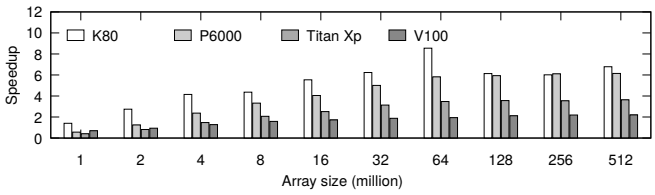


Figure 5: Speedup of atomic operation over prefix sum-based frontier queue generation on Kepler K80 GPU, Pascal GPUs (P6000 and Titan Xp) and Volta V100 GPU.

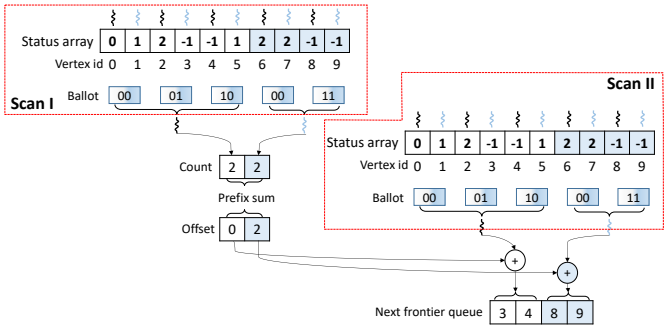


Figure 6: Double-scan based frontier generation in bottom-up of Figure 2 at level 3. Note, in bottom-up, the unvisited vertices (i.e., status equals to -1) are treated as frontiers according to Definition 2.1.

rely on a condition statement to distinct the frontiers from the unvisited vertices.

Example 3. *Figure 4(c) explains how to use the frontier queue from bottom-up to conduct top-down traversal. Basically, we will iterate through the bottom-up frontier queue but only traverse those vertices that are just visited – {3, 4, 8}. The actual traversal is identical to scan-free approach in Figure 4(a).*

It is worthy of mentioning that *no frontier queue generation method* presents two benefits. First, it avoids scanning the status array for the frontier queue. Note traditional BFS will need to scan through the entire status array to produce the frontier queue for the first top-down iteration after the switch. Second, bottom-up BFS always generates sorted frontier queue, which introduces friendly memory access [24].

4.2 Bottom-Up Frontier Queue Generation

Since bottom-up BFS will potentially explore a majority of the vertices in the graph [4], storing the frontiers (i.e., unvisited vertices) in order will result in sequential memory accesses to their neighbor lists, as suggested by Enterprise [24], thus better performance. However, to generate such a sorted frontier queue, Enterprise [24] experiences strided memory accesses, which is addressed by our double-scan method.

Double-scan frontier queue generation. Basically, this method partitions the status array into multiple segments. Note, we try to make the length of each segment evenly divisible by 32 which is the number of threads in a warp. As such, the biggest difference between different segments can be 32. Scan I counts the number of frontiers from each segment. Next, it uses prefix sum to compute the global offsets for each segment to place the frontiers. Note, we choose prefix sum for the global offset computation because we need to generate globally sorted frontiers. Scan II puts those frontiers from each segment into next frontier queue based upon the global offsets.

The most noteworthy part of double-scan is that we manage to generate a sorted frontier queue with coalesced memory access. The key toward this achievement is that even various threads from a warp are responsible for different segments, we schedule them to collaboratively scan each segment. That is, when processing a specific segment, all the threads in the same warp should focus on the same segment. Here, we use CUDA shuffle function to communicate the essential begin and end offsets information surrounding the focused segment across warp. During scan, all threads use __ballot_sync [23] to indicate whether the 32 scanned indices are frontiers. But only the thread that is responsible for this segment needs to keep, process and extract the frontiers from this returned ballot value.

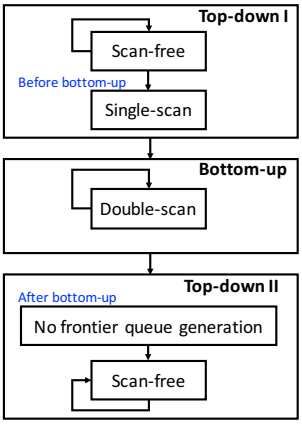


Figure 7: Adaptive flow chart in XBFS.

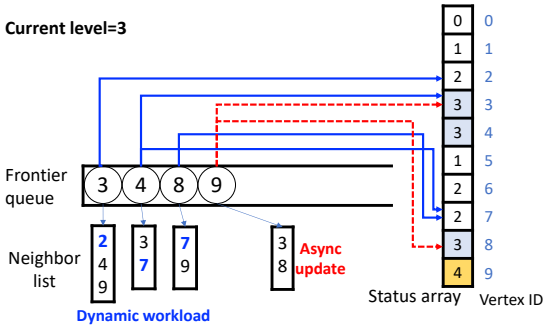


Figure 8: Example for the cause of nondeterministic workloads and the possibility of asynchronous traversal at level 3 of the graph in Figure 2.

Example 4. *Figure 6 exemplifies the flow of double-scan at level 3 of the toy graph from Figure 2. In this example, we assume a warp contains two threads – the black and blue threads. And black thread accounts for segment of indices 0 - 5 and blue thread corresponds to segment of indices 6 - 9. Note, this partition ensures each segment contain the number of indices that are divisible by two – the assumed warp size.*

Subsequently, we schedule these two threads to collaboratively scan the status array and use __ballot_sync to return whether a vertex is a frontier (i.e., '1' in Ballot array in the figure means it is a frontier). Since 0 - 5 belongs to black thread, all the ballot returned values in this range go to black thread. Similarly, the ballot returned values of 6 - 9 will go to blue threads. Afterwards, each thread uses __popc [1] to figure out the number of frontiers. Finally, it executes the prefix sum to figure out the global offset – 0 and 2 – for each thread.

Scan II is roughly similar to scan I during scanning and ballot operation, except it needs to extract and put the frontiers from the returned ballot value into the next frontier queue at correct location. In particular, the black thread puts frontiers {3, 4} into the next frontier queue starting from offset 0 while the blue thread puts {8, 9} in this queue from offset 2. Eventually, we have the sorted frontier queue {3, 4, 8, 9}.

4.3 Runtime Management

It is crystal clear that various frontier queue generation methods excel at different traversal stages, which suggests we need an adaptive approach to retain the best performance. Figure 7 plots the best adaptation design. In particular, we start with scan-free frontier queue generation and change to single-scan right before bottom-up. Bottom-up traversal relies on double-scan. The second top-down phase begins with no frontier queue generation and ends up with scan-free.

The majority of the switching criteria are straightforward – scan-free prefers small number of frontiers – thus it will be used at the initial and ending levels of BFS. Since double-scan and no frontier queue generation are directly tied to direction optimizing [4], XBFS adopts the same switch condition as direction-optimizing BFS, that is, switching from top-down to bottom-up when $\frac{edgeCheck}{|E|} \geq \alpha$ (i.e., double-scan) and back to top-down when that condition does not hold. In this paper, we use $\alpha = 0.1$.

Whereas, determining the criterion for switching to single-scan cannot hinge upon direction switching since direction switching happens after this level. In order to accurately predict this level, we introduce another parameter λ which measure the *edgeCheck* incremental ratio.

$$\lambda = \frac{edgeCheck_{current\ level}}{edgeCheck_{prior\ level}} \quad (1)$$

With λ, the condition of deciding the level before direction switching can be expressed as following:

$$\frac{edgeCheck_{current\ level} \cdot \lambda}{|E|} \geq \alpha, \quad (2)$$

Note, λ is not pre-tuned. Instead, we calculate this relative ratio from previous to current level. Since we always need to track this *edgeCheck* parameter in order to determine direction switching, the overhead of this computation is negligible.

5 RUNTIME OPTIMIZED BOTTOM-UP TRAVERSAL

While rapidly generating frontier queues only solves half of the problem for BFS, this section unveils the dynamic optimizations for *bottom-up traversal*, including dynamic workload balancing and asynchronous traversal.

5.1 Dynamic Bottom-Up Workload Balancing

5.1.1 Observations. This technique is mainly motivated by Observation 3, which suggests the mismatch between predominately used degree-based workload balancing technique and bottom-up BFS.

Observation 3. *In bottom-up BFS, the workload associated with each vertex is not necessarily proportional to its degree.*

Illustration. Thanks to *early termination* [4] which stops checking more neighbors once a valid parent is found in bottom-up, BFS can tremendously reduce the workload associated with each vertex. However, this design also breaks the correlation between degree and workload. In other words, those high degree vertices might end up with small workloads because of early termination.

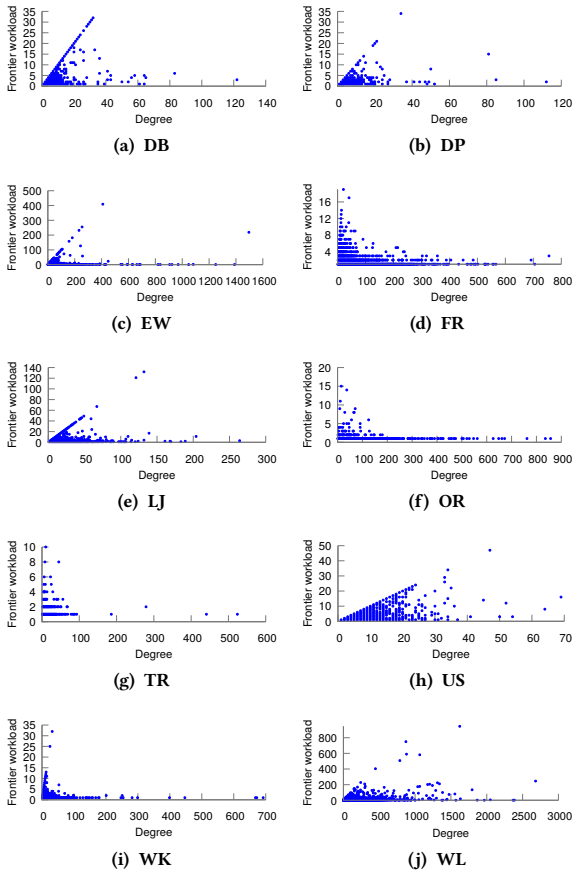


Figure 9: The frontier degree and its real workload (number of traversed edges) across all datasets.

Example 5. *Figure 8 exemplifies this situation. Vertex 3 has degree of three while vertices 4 and 8 have degrees of two, but vertices 3, 4 and 8 possesses real workloads of one, two and one, respectively, because they find their parent vertices at the first, second and first check of their neighbor lists, respectively. This example suggests that the final workload from each frontier, in bottom-up, is not proportional to its degree.*

Figure 9 further studies the vertex degree and workload relationship for all the real graphs from this paper. The following twin observations are implied:

(1) A majority of the high degree frontiers presents a small volume of workloads.
(2) For some datasets, e.g., DP, US and WL, high degree frontiers could come with a large volume of workloads.

The first observation suggests that existing GPU-based attempts [24, 30, 42] which assign a CTA and Warp of threads to frontiers with large and medium degrees, respectively might waste GPU resources. The second observation represents the unpredictability of the workload.

5.1.2 Techniques. To this end, we present three types of solutions. First, a static solution that assigns a single thread to one frontier regardless of the degree to save threads. In addition, we introduce two dynamic mechanisms, i.e., thread- and warp- centric approaches, to deal with the unpredictable amounts of workloads. Below, we describe the designs.

Algorithm 1 Thread-centric dynamic workload balancing

```
1: procedure threadCentricWB
2:     myFrontier ← frontierQueue[tid];
3:     traverse (myFrontier);
4:     while myFrontierIndex < |frontierQueue| do
5:         myFrontier ← atomicInc (myFrontierIndex, 1);
6:         traverse (myFrontier);
7:     end while
8: end procedure
```

Static solution generates a single universal frontier queue to hold all frontiers since there is no need to classify the frontiers into large, medium and small ranges, which is used by the traditional attempt [24]. Afterwards, we schedule thread tid to work on the frontiers stored in indices $blockDim.x \times gridDim.x \times p + tid$ of the frontier queue at iteration p.

Thread-centric dynamic workload balancing. To deal with the workload unpredictability, this method schedules each thread to fetch a new frontier once it finishes current one. All to be processed frontier is decided during runtime with atomic operation. Particularly, we partition the whole frontier queue into two ranges, where the first range is at size of the total number of threads. The remaining part is the second range. The first range will not need atomic operation since we can simply schedule thread tid to work on the frontier that resides at index tid from the frontier queue. For the second range, a global counter is shared across all threads to retrieve the frontiers.

Algorithm 1 sketches the proposed design with the highlighted lines as the dynamic scheduling. Each thread fetches one frontier from the frontier queue at line 5. In line 6, all threads traverse the neighbors of their own frontiers.

Warp-centric dynamic workload balancing attempts to reduce the atomic operation but with the overhead of more instructions, including a warp level shuffle instruction. Particularly, this design schedules one thread to fetch 32 new frontiers instead of one at a time. Afterwards, this thread will disseminate the fetched frontiers across the warp. It is worthy of noting that the remaining threads that are not fetching new frontiers will not move forward after finishing their own frontiers due to the SIMD nature. In this way, we can reduce the atomic operation usage by 32× but with more intra-warp communications.

Algorithm 2 explains the design of our warp-centric method, where the highlighted lines are the novel design. At line 6 in Algorithm 2, after traversal, all threads in a warp should have finished their work due to locked execution convention from SIMD. Now instead of each thread fetch its own workload, XBFS schedules the $laneId = 0$ thread to obtain the next frontier begin position for the entire warp with atomic operation (line 6 - 8). Finally, XBFS uses the __shfl_sync to share this new bulk of frontiers across the warp (lines 9).

Dynamic intra-warp working donation aims to schedule light loaded threads to help heavy ones inside the warp. Note, our warp-centric approach should already ensure the inter-warp

Algorithm 2 Warp-centric dynamic workload balancing

```
1:  procedure warpCentricWB
2:      laneId ← threadIdx.x & 31;
3:      while myFrontierIndex < |frontierQueue| do
4:          int myFrontier ← frontierQueue [myFrontierIndex];
5:          traverse (myFrontier);
6:          if laneId==0 then          ▷ Warp-based workload fetching
7:              warpFrontierBegin ← atomicAdd (counter, 32)
8:          end if
9:          myFrontierIndex ← __shfl_sync() + laneId
10:     end while
11: end procedure
```

workload balance. The main idea in this work donation design is to inform all the finished threads in the warp to together help the last unfinished thread. Note, other designs are also possible but would involve much higher overhead. After every five neighbors, we schedule all threads to execute popCount(__ballot_sync()). Once it returns 31, we will assign the workload from the unfinished thread to the remaining threads in the warp.

While doing this can help balance the workload, nontrivial overhead is introduced to check whether majority of the threads have finished. Our evaluation shows this method suffers from 1 - 5% slow down across various graphs.

5.2 Asynchronous Bottom-Up Traversal

This asynchronous bottom-up traversal allows us to discover frontiers for more than one level at each iteration, which will reduce the repeated edge checks for the same frontier in bottom-up. In particular, this design is inspired by the following novel rules:

Lemma 1. *In bottom-up BFS, the next iteration frontiers will be a subset of that in current iteration.*

Proof. Supposing $Q_{current}$ and Q_{next} denote the frontiers in current and next iterations, respectively. By the definition of the frontiers in bottom-up BFS in Section 2.2, Q consists of all unvisited vertices before the starting of this iteration. Since each iteration will certainly visit some unvisited vertices, which means $Q_{current} = Q_{next} \bigcup vertex_{current}$. Because current level should visit some frontiers, that is, $vertex_{current} \neq \emptyset$, then $Q_{next} \subset Q_{current}$. □

Lemma 2. *At level l of bottom-up, if one frontier does not have any neighbors from level $l-1$ but has neighbor from level l, this frontier belongs to level $l+1$.*

Proof. We will use proof by contradiction to illustrate this theorem. As suggested by the nature of BFS, each vertex is reached by its parent vertices, which is one level above this vertex. Therefore, in order to be visited at level l, this vertex needs to have a parent vertex which is from level $l-1$. Otherwise, it should not belong to level l. Further, because this vertex has neighbors in level l, it must be discovered at level $l+1$. □

Lemma 1 and 2 together warrant that we can discover vertices from both current and next levels at bottom-up. In particular, Lemma 1 ensures the frontiers from next iteration is a subset of current iteration and Lemma 2 guarantees the correctness of this

Algorithm 3 Asynchronous bottom-up traversal

```
1:  procedure asynchronousBottomUp
2:      while tid < |frontierQueue| do
3:          int myFrontier ← frontierQueue[tid];
4:          while nebr in adjacencyList (myFrontier) do
5:              neighborLevel=statusArray[nebr];
6:              if neighborLevel== l - 1 then
7:                  statusArray[myFrontier] = l;        ▷ Early termination
8:                  isEarlyTermination = true; break;
9:              end if
10:             if neighborLevel == l then
11:                 hasCurrNeighbor = true;
12:             end if
13:         end while
14:         if !isEarlyTermination && hasCurrNeighbor then
15:             statusArray[myFrontier] = l+1;          ▷ Find next level vertex
16:         end if
17:     end while
18: end procedure
```

asynchronous traversal, that is, we will not wrongly mark current level vertex as next level.

Algorithm 3 sketches the pseudocode of our asynchronous traversal with the highlighted part as the new design. Firstly, this algorithm guarantees that the level l vertices can be updated correctly. This is achieved via exactly following the traditional bottom-up traversal process, that is, we scan the neighbor list for a frontier (line 5 - 9 in Algorithm 3). Once one neighbor belongs to $l-1$, this algorithm will update the status of this frontier to be l and early terminate the check.

The advantage of Algorithm 3 lies in the situation when none of the neighbors of a frontier is from $l-1$. In this case, Algorithm 3 actually tracks whether any of the checked neighbor belongs to l at line 10 - 12. Note, this process of tracking involves no extra memory accesses. Once, we find a frontier does not have any level $l-1$ neighbors but has level l neighbors, we can surely mark this frontier as level $l+1$ vertex (line 14 - 16 in Algorithm 3).

Example 6. *Frontier '9' in Figure 8 is the example to aid the understanding of asynchronous traversal. When checking the neighbor statuses for vertex 9, that is, {3, 8}, we find neither neighbors is from preceding level (i.e., level 2), which assures frontier 9 does not belong to level 3. Further, since 9 has at least one neighbor from level 3, we can safely mark frontier 9 as level 4 vertex.*

Lemma 3. *The probability of level $l+1$ vertex v_s being identified at iteration l is $(1-\frac{s}{|Q|})^{N_l}$, where s is the index of vertex v_s in frontier queue Q and N_l is the number of neighbors of vertex v_s that belongs to level l.*

Proof. In bottom-up traversal, the frontier queue Q consists of three types of vertices, that is, the vertices belong to level l, $l+1$ and $level > l+1$. Here, we denote the set of vertices that belongs to level l as Q_l.

Assuming the vertex v_s is at index s of this Q and belongs to level $l+1$, the expected number of level l frontiers that have already been visited before index s is:

$$\mathbb{E}_l = \frac{|Q_l|}{|Q|} \cdot s \tag{3}$$

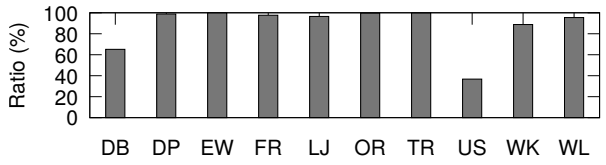


Figure 10: Ratio of the next level vertices updated at current iteration over the total number of vertices in next level.

Adopting a common estimation method, the probability of any level l vertex that has already been visited before index s is: $\frac{\mathbb{E}_l}{|Q_l|} = \frac{s}{|Q|}$.

Since v_s belongs to level $l+1$, it should have at least one neighbor that belongs to level l according to Lemma 2. Here, we assume v_s has N_l neighbors that belong to level l, the probability of v_s being identified is the same as **one or more of v_s's level l neighbors (i.e., N_l) is already visited**. We first calculate the probability of none of N_l is visited, that is:

$$\bar{P} = (1 - \frac{\mathbb{E}_l}{|Q_l|})^{N_l} = (1 - \frac{s}{|Q|})^{N_l} \quad (4)$$

Thus, the probability of v_s being visited at level i is:

$$P = 1 - \bar{P} = 1 - (1 - \frac{s}{|Q|})^{N_l} \quad (5)$$

We analyze this probability in three cases:

- When $s = 0$, we get $P = 0$ which means v_s will not be visited at level l.
- When $s = |Q|, P = 1$ which means v_s will be traversed at level l.
- When $s \in [1, |Q|)$, P depends upon the value of s and N_l. In particular, if $s \geq \frac{|Q|}{2}$, the probability of v_s being asynchronously traversed becomes 50%, 75%, ..., if it has 1, 2, ... neighbors from level l.

□

Figure 10 further evaluates the ratio of the extra vertices that are identified at the first bottom-up iteration to the total number of vertices belonging to the second level of bottom-up BFS. In most of those testing graphs, especially for DP, OR, LJ, EW, TR and FR, XBFS nearly visits all the next level vertices (>99%). Even for US, we also have at least nearly 40% extra update ratio. On average, the extra updates ratio is 88%. This result has practically exhibited the correctness of Lemma 3.

6 EVALUATION

We implement XBFS with 1,500 lines of code (LOC) in C++ and CUDA, extending a GPU-based high-performance BFS implementation - Enterprise [24]. XBFS is compiled with NVIDIA CUDA 9.2 with the optimization flag set to be O3. In this work, the experiments are performed on NVIDIA Pascal Quadro P6000 GPU (which comes with 24 GB memory) on a server with Intel(R) Core(TM) i7-8700 (3.20Hz) CPU. The server runs Ubuntu Linux 16.04 operating system with kernel version to be 4.15.0. Ligra runs on Amazon AWS server which features 24-core Intel Xeon Platinum 8175M CPUs and 384 GB memory. Note, the Quadro P6000 and Intel Xeon Platinum 8175M CPU cost ~$4,500 and ~$8,000, respectively, according to Amazon [2, 18].

Table 2: Runtime (ms) comparison between XBFS and state-of-the-art projects with maximum and minimum speedups highlighted with heavy and light shadows, respectively. Note, "OOM" stands for out of memory. error.

Datasets	XBFS	Enterprise	Tigr	Gunrock	Ligra
DB	7.9	30.2	12.9	16.4	54
DP	5.3	14	8.1	22.9	48.2
EW	19.6	88.7	74.7	145.0	71.9
FR	102.8	408	883.9	OOM	526.7
LJ	6.9	24	17.0	34.9	29.4
OR	2.83	14.2	48.7	154.1	29.8
TR	28.1	101.3	73.4	171.2	95.1
US	10.2	21.2	17.7	28.3	62.6
WK	21.4	76.9	92.1	168.5	259.3
WL	23.6	59.5	120.7	262.8	58.3
Avg. speedup	-	3.5	4.9	11.2	6.1

6.1 XBFS vs. State-of-the-art

Table 2 compares XBFS against the state-of-the-art BFS implementations, including Enterprise [24], Tigr [31], Gunrock [42], and Ligra [35]. XBFS, Enterprise, Tigr and Gunrock are tested in GPU P6000. Ligra runs on the Intel Platinum 8175M CPU. On average, XBFS presents 3.5×, 4.9×, 11.2× and 6.1× speedup over Enterprise, Tigr, Gunrock, and Ligra, respectively.

XBFS turns out to be the fastest on every graph dataset. The maximum speedup over Enterprise, Tigr and Gunrock occur in OR datasets. The speedups are 5.0 ×, 17.2 × and 54.3 respectively. The maximum speedup of 12.1 occurs on WK dataset over Ligra. For minimum speedup, we observe 2.1 × on US over Enterprise, 1.5 × on DP over Tigr, 2.1 × on DB over Gunrock and 2.5 × on WL over Ligra. Note, the performance numbers for the state-of-the-art reported in Table 2 might mismatch the ones in their manuscripts since we use different GPU/CPU platforms.

6.2 Performance Impacts of Different Techniques

Figure 11 examines the performance impact of various techniques, where **Ad, WB** and **Async** stand for adaptive frontier queue generation, dynamic workload balancing and asynchronous traversal. Here, we use the state-of-the-art Enterprise [24] as the baseline and implement our contributions, i.e., Ad, WB and Async atop the baseline for the comparison.

Adaptive frontier queue generation method, as shown in Figure 11, helps XBFS gain 2.3× speedup over baseline. In particular, the maximum and minimum are 3.5× on DB and 1.1× on WL, respectively. **Dynamic workload balancing** along with adaptive frontier queue generation produces, on average, 3.1× speedup over state-of-the-art Enterprise static workload balancing. On OR and US, we get the maximum and minimum speedup, which is 4.8× and 1.9×, respectively. **Asynchronous bottom-up traversal** together with the adaptive frontier queue generation and dynamic workload balancing, yields 3.5×, on average, performance boost over Enterprise. The maximum climb is 5.0× on OR, while US gains the minimum of 2.1× speedup.

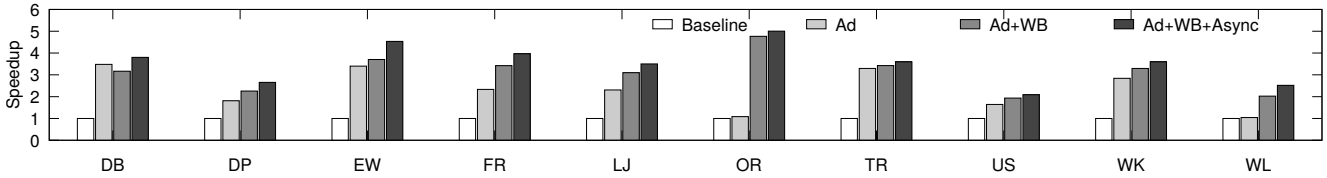
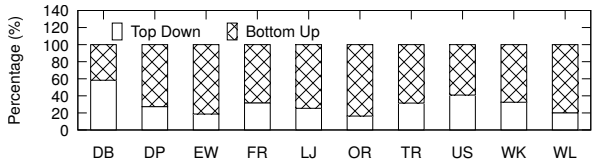
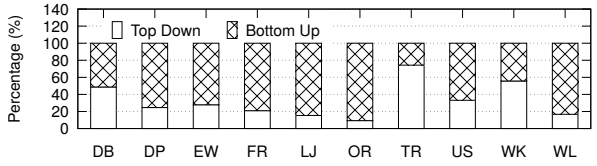


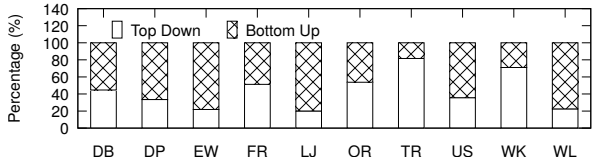
Figure 11: Performance impacts by different contributions over baseline (Enterprise). Ad: Adaptive frontier queue generation, WB: Workload balancing in bottom-up, Async: Optimized asynchronous bottom-up traversal.



(a) Baseline.



(b) After adaptive frontier generation optimization.



(c) XBFS.

Figure 12: The time consumption ratio dynamics of top-down vs. bottom-up.

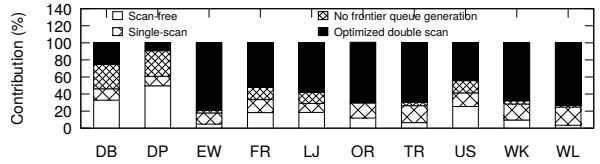


Figure 13: Contribution of different frontier queue generation algorithm in XBFS.

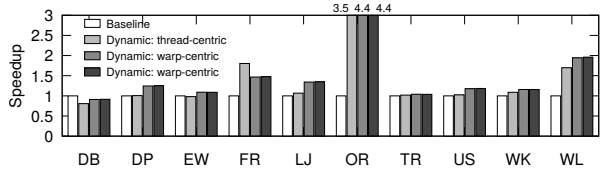


Figure 14: Impacts of various workload balancing methods.

Figure 12 further presents the time consumption ratio dynamics of top-down and bottom-up traversal with respect to these techniques. In baseline, the time consumption ratios of top-down and bottom-up are 30% and 70%, respectively. After our adaptive frontier queue generation, as shown in Figure 12 (b), top-down and bottom-up retain similar ratios as 32% and 68%, respectively. Such a slight change also matches the results we obtained in Figure 13 which suggests similar performance impacts brought by adaptive frontier queue generation to top-down (i.e., scan-free, single-scan and no frontier queue generation) and bottom-up (i.e., optimized double-scan). Eventually, our runtime optimizations which mainly target bottom-up traversal change the average ratio to 43% and 57% for top-down and bottom-up, as shown in Figure 12(c).

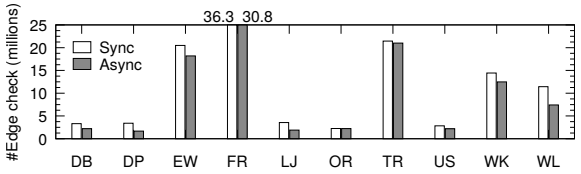


Figure 15: Number of edge checks by synchronous vs asynchronous bottom-up traversal.

6.3 Study of Various Techniques

Figures 13, 14 and 15 further study the speedup contributions from the four scan approaches in adaptive frontier queue generation; performance differences of three workload balancing methods; and edge check reduction introduced by asynchronous traversal, respectively.

As shown in Figure 13, the contribution breakdown of scan-free, single-scan, no frontier queue generation and optimized double scan are 18%, 15.7%, 11.4% and 54.9%, respectively. In particular, scan-free observes contribution for graphs with higher diameters, like DP, DB and US while optimized double scan for graphs with high volume of workloads in bottom-up, such as EW, WL and OR.

Figure 14 demonstrates the impact of three workload balancing techniques in bottom-up traversal. On average, the speedup increases over baseline with adaptive frontier queue generation implementation for static, warp-centric and thread-centric are 1.4×, 1.6× and 1.6×, respectively. Surprisingly, warp-centric turns out to be similar to thread-centric. We find the overhead of shuffle instruction is similar to atomic operation according to our profiling with nvprof and two profiling flags – inst_executed_global_atomics and inst_executed_global_reductions. Besides, warp-centric introduces slightly more instructions than thread-centric option, resulting in a similar performing warp-centric approach.

It is important to note that there is performance degradation for the proposed approaches on DB dataset. It is caused by the existence of outlier frontiers whose workloads are very high - close to their degrees. This is also evident in Figure 9.

Figure 15 plots the number of edge checks in normal synchronous and the asynchronous traversals. In particular, the edge check reduction percentage is 23.2%, ranging from 51% on DP to a minimum of 1% on OR dataset.

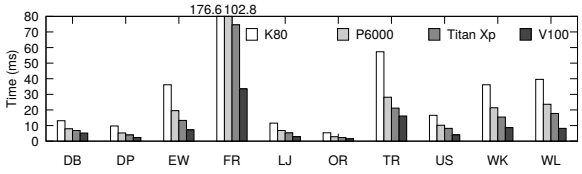


Figure 16: XBFS performance in different GPUs.

6.4 XBFS Performance on Different GPUs

Figure 16 demonstrates the performance of XBFS on K80, P6000, Titan Xp and V100 GPUs. The XBFS performance aligns with the expectation, that is, the minimum appears on K80 across all the datasets and goes on increasing along P6000, Titan Xp and V100 where V100 delivers the best performance. On average, P6000, Titan Xp and V100 are 1.8×, 2.3× and 4.1× faster than K80, respectively. Such climbing performance gains of XBFS from older to newer generations GPU also reflect the effectiveness and efficacy of various optimizations from XBFS.

7 CONCLUSION

In this work, we develop XBFS which introduces dynamic optimizations to BFS on GPUs. We adaptively use four either novel or optimized scan approaches to rapidly generate frontier queue. Further, inspired by the observation that bottom-up BFS experiences unpredictable amounts of workload, we propose the novel dynamic workload balancing method. Third, we design and implement the first truly asynchronous BFS traversal. Taken together, XBFS is, on average, 3.5×, 4.9×, 11.2× and 6.1× faster than the state-of-the-art Enterprise, Tigr, Gunrock and Ligra respectively. The first three state-of-the-arts along with XBFS are run in P6000 GPU and Ligra on 24-core Intel Xeon Platinum 8175M CPU for speedup calculations.

ACKNOWLEDGMENT

We thank the anonymous reviewers and our shepherd Ana Lucia Varbanescu for their constructive suggestions that help improve the quality of this paper, and Julian Shun to help run the Ligra experiment. We also would like to gracefully acknowledge the support from XSEDE supercomputers and Amazon AWS, as well as the NVIDIA Corporation for the donation of the Titan Xp and Quadro P6000 GPUs. This work was in part supported by NSF CRII Award No. 1850274.

REFERENCES

[1] Nvidia cuda c programming guide. *NVIDIA Corporation* (2018).
[2] Amazon. Price of Quadro P6000. Retrived from https://www.amazon.com/PNY-Quadro-P6000-Graphic-Card/dp/B01M0S2FKR?keywords=Quadro+P6000&qid=1539599742&s=Electronics&sr=1-2&ref=sr_1_2. Accessed: 2018, October 6.
[3] Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., et al. *LAPACK Users' guide*. 1999.
[4] Beamer, S., Asanović, K., and Patterson, D. Direction-optimizing breadth-first search. *SC* (2013).
[5] Ben-Nun, T., Sutton, M., Pai, S., and Pingali, K. Groute: An asynchronous multi-gpu programming model for irregular computations. In *PPoPP* (2017), ACM.
[6] Bhattarai, B., Liu, H., and Huang, H. H. Ceci: Compact embedding cluster index for scalable subgraph matching. In *SIGMOD* (2019).
[7] Blelloch, G. E., Gu, Y., Shun, J., and Sun, Y. Parallelism in randomized incremental algorithms. In *SPAA* (2016).
[8] Buluç, A., and Madduri, K. Parallel breadth-first search on distributed memory systems. In *SC* (2011).
[9] Collection, T. K. N. http://konect.uni-koblenz.de/networks/.
[10] Dongarra, J. J., Bunch, J. R., Moler, C. B., and Stewart, G. W. *LINPACK users' guide*, vol. 8. Siam, 1979.
[11] Graph500. http://www.graph500.org/.
[12] Han, W., Mawhirter, D., Wu, B., and Buland, M. Graphie: Large-scale asynchronous graph traversals on just a gpu. In *PACT* (2017), IEEE.
[13] Harris, M. Parallel prefix sum (scan) with cuda.
[14] Hong, S., Kim, S. K., Oguntebi, T., and Olukotun, K. Accelerating cuda graph algorithms at maximum warp. In *PPoPP* (2011), vol. 46, ACM, pp. 267–276.
[15] Hong, S., Oguntebi, T., and Olukotun, K. Efficient parallel graph exploration on multi-core cpu and gpu. In *PACT* (2011).
[16] Hu, Y., Liu, H., and Huang, H. H. High-performance triangle counting on gpus. In *HPEC* (2018).
[17] Hu, Y., Liu, H., and Huang, H. H. Tricore: Parallel triangle counting on gpus. In *SC* (2018).
[18] Intel. Price of Intel Xeon Platinum 8158 Processor. Retrived from https://ark.intel.com/products/120500/Intel-Xeon-Platinum-8158-Processor-24-75M-Cache-3-00-GHz-. Accessed: 2018, October 6.
[19] Ji, Y., Liu, H., and Huang, H. H. ispan: Parallel identification of strongly connected components with spanning trees. In *SC* (2018).
[20] Khorasani, F., Gupta, R., and Bhuyan, L. N. Scalable simd-efficient graph processing on gpus. In *PACT* (2015).
[21] Khorasani, F., Vora, K., Gupta, R., and Bhuyan, L. N. Cusha: vertex-centric graph processing on gpus. In *HPDC* (2014).
[22] Li, D., and Becchi, M. Deploying graph algorithms on gpus: An adaptive solution. In *IPDPS* (2013).
[23] Lin, Y., and Grover, V. Using CUDA Warp-Level Primitives. Retrived from https://devblogs.nvidia.com/using-cuda-warp-level-primitives/. Accessed: 2019, April 3.
[24] Liu, H., and Huang, H. H. Enterprise: Breadth-first graph traversal on gpus. In *SC* (2015).
[25] Liu, H., and Huang, H. H. Graphene: fine-grained io management for graph computing. In *USENIX FAST* (2017).
[26] Liu, H., and Huang, H. H. Simd-x: Programming and processing of graph algorithms on gpus. *arXiv preprint arXiv:1812.04070* (2018).
[27] Liu, H., Huang, H. H., and Hu, Y. ibfs: Concurrent breadth-first search on gpus. In *SIGMOD* (2016), ACM, pp. 403–416.
[28] Luo, L., Wong, M., and Hwu, W.-m. An effective gpu implementation of breadth-first search. In *DAC* (2010), ACM, pp. 52–55.
[29] McLaughlin, A., and Bader, D. A. Scalable and high performance betweenness centrality on the gpu. In *SC* (2014), IEEE Press, pp. 572–583.
[30] Merrill, D., Garland, M., and Grimshaw, A. Scalable gpu graph traversal. In *PPoPP* (2012).
[31] Nodehi Sabet, A. H., Qiu, J., and Zhao, Z. Tigr: Transforming irregular graphs for gpu-friendly graph processing. In *ASPLOS* (2018).
[32] Nvidia. Nvidia tesla p100 architecture whitepaper. 2016.
[33] Pearce, R., Gokhale, M., and Amato, N. M. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *SC* (2010), IEEE, pp. 1–11.
[34] Sengupta, S., Harris, M., Zhang, Y., and Owens, J. D. Scan primitives for gpu computing.
[35] Shun, J., and Blelloch, G. E. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP* (2013), ACM.
[36] SNAP: Stanford Large Network Dataset. http://snap.stanford.edu/data/.
[37] Springer, M. Breadth-first Search in CUDA. Retrived from https://m-sp.org/downloads/titech_bfs_cuda.pdf. Accessed: 2019, April 4.
[38] Sun, J., Vandierendonck, H., and Nikolopoulos, D. S. Graphgrind: addressing load imbalance of graph partitioning. In *ICS* (2017).
[39] Thampi, S. M., et al. Survey of search and replication schemes in unstructured p2p networks. *arXiv preprint arXiv:1008.1629* (2010).
[40] Tiskin, A. All-pairs shortest paths computation in the bsp model. In *ICALP* (2001).
[41] Volta, I. The world's most advanced data center gpu. *https://devblogs.nvidia.com/parallelforall/inside-volta* (2017).
[42] Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., and Owens, J. D. Gunrock: A high-performance graph processing library on the gpu. In *PPoPP* (2016).
[43] Wikipedia. Graphics Processing Units. Retrived from https://en.wikipedia.org/wiki/Graphics_processing_unit. Accessed: 2018/10/6.
[44] Zhu, X., Chen, W., Zheng, W., and Ma, X. Gemini: A computation-centric distributed graph processing system. In *OSDI* (2016).