Enterprise: Breadth-First Graph Traversal on GPUs

Hang Liu H. Howie Huang
Department of Electrical and Computer Engineering
George Washington University

ABSTRACT

The Breadth-First Search (BFS) algorithm serves as the foundation for many graph-processing applications and analytics workloads. While Graphics Processing Unit (GPU) offers massive parallelism, achieving high-performance BFS on GPUs entails efficient scheduling of a large number of GPU threads and effective utilization of GPU memory hierarchy. In this paper, we present Enterprise, a new GPU-based BFS system that combines three techniques to remove potential performance bottlenecks: (1) *streamlined GPU threads scheduling* through constructing a frontier queue without contention from concurrent threads, yet containing no duplicated frontiers and optimized for both top-down and bottom-up BFS. (2) *GPU workload balancing* that classifies the frontiers based on different out-degrees to utilize the full spectrum of GPU parallel granularity, which significantly increases thread-level parallelism; and (3) *GPU based BFS direction optimization* quantifies the effect of hub vertices on direction-switching and selectively caches a small set of critical hub vertices in the limited GPU shared memory to reduce expensive random data accesses. We have evaluated Enterprise on a large variety of graphs with different GPU devices. Enterprise achieves up to 76 billion traversed edges per second (TEPS) on a single NVIDIA Kepler K40, and up to 122 billion TEPS on two GPUs that ranks No. 45 in the Graph 500 on November 2014. Enterprise is also very energy-efficient as No. 1 in the GreenGraph 500 (small data category), delivering 446 million TEPS per watt.

1. INTRODUCTION

Breadth-First Search (BFS) algorithm serves as a building block for many analytics workloads, e.g., single source shortest path, betweenness centrality [16, 31, 32, 42] and closeness centrality [37, 40]. Notably, the Graph 500 benchmark uses BFS on power-law graph to evaluate high-performance hardware architectures and software systems that are designed to run data-intensive applications [1]. In this work, we are particularly interested in accelerating BFS traversal on power-law graphs, which can be found in a wide spectrum of applications, e.g., biomedical cells [26], WWW [9, 25] and social network [17, 27].

The traditional (top-down) BFS algorithm starts at the root of the graph and inspects the status of all of its adjacent (or neighboring) vertices. If any adjacent vertex is unvisited, the algorithm will identify it as a frontier, put it into a queue that we refer to as the *frontier queue* in this paper, and subsequently mark it visited. As the result of the inspection of the current level, the frontier queue consists of all the vertices that have just been visited and will be used for expansion at the next level. To do so, BFS iteratively selects each vertex in the frontier queue, inspects its adjacent vertices, and marks this vertex visited. The process of expansion and inspection is repeated level by level till no vertex in this graph remains unvisited. For recently proposed bottom-up BFS [10], the workflow is similar with different vertices identified as frontiers. Clearly, the frontier queue is at the heart of the BFS algorithm - at each level BFS starts with the frontier queue prepared by the inspection of the preceding level and ends with a new frontier queue that will be used for the expansion of next level.

Graphics Processing Unit (GPU) provides not only massive parallelism (in 100Ks threads) but also fast I/O (with 100s GB/s memory bandwidth), which makes it an excellent hardware platform for running the BFS algorithm. Unfortunately, although recent attempts [21, 33, 36, 24] have made remarkable progress, unleashing the full power of GPUs to achieve high-performance BFS remains extremely challenging. In this paper, we advocate that a high-performance BFS system shall carefully match the hardware aspects of GPUs through efficient management of numerous GPU streaming processors and unique memory hierarchy.

In this paper, we present Enterprise[1], a new GPU-based BFS system that tailors the BFS execution flow and data access pattern to take full advantage of high thread count and massive memory bandwidth of GPUs. Enterprise achieves up to 76 billion traversed edges per second (TEPS) on a single NVIDIA Kepler K40, and up to 122 billion TEPS and 446 million TEPS per watt on two GPUs, which ranks No. 45 and No. 1 in the Graph 500 and GreenGraph 500

SC '15, November 15-20, 2015, Austin, TX, USA
Copyright 2015 ACM 978-1-4503-3723-6/15/11 ...$15.00
DOI: http://dx.doi.org/10.1145/2807591.2807594

[1]Enterprise is the name of the first space shuttle built for NASA on 1976.

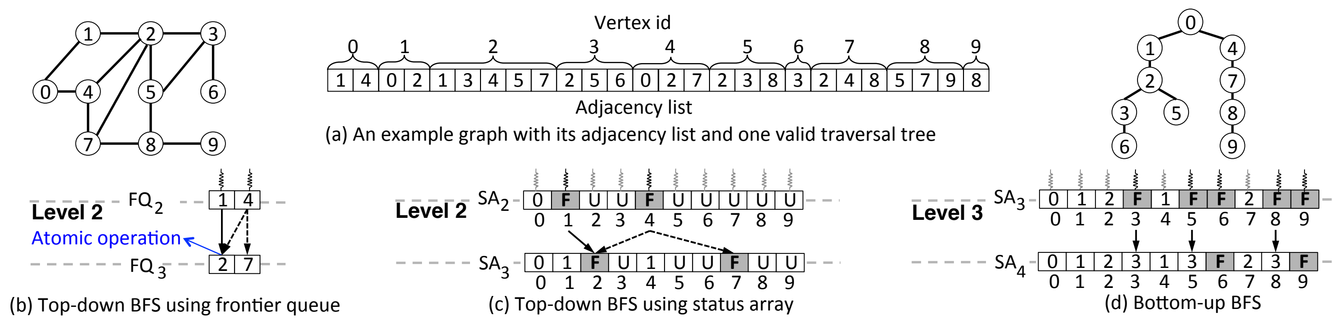

Vertex id
Adjacency list
(a) An example graph with its adjacency list and one valid traversal tree
Level 2
FQ2
FQ3
Atomic operation
(b) Top-down BFS using frontier queue
Level 2
SA2
SA3
(c) Top-down BFS using status array
Level 3
SA3
SA4
(d) Bottom-up BFS

Figure 1: (a) An example graph with its adjacency list and one valid BFS traversal tree (there may exist multiple valid BFS trees). We use this example throughout the paper. Top-down BFS using (b) the frontier queue or (c) status array, vs. (d) bottom-up BFS. The numbers in the status array represent at which level the vertex is visited. The labels of F and U represent frontier and unvisited vertex, respectively. In (c) and (d), the gray threads that are assigned to non-frontier vertices would idle with no work.

small data category in November 2014, respectively. This is achieved through the design of three novel techniques:

First, **streamlined GPU threads scheduling** is achieved through efficient frontier queue generation of two distinct steps: the scan of the status array of the graph at the current level, followed by prefix sum based frontier queue generation. When enqueueing a frontier, atomic operations are needed to ensure the uniqueness of each frontier vertex in the queue, however, for GPUs such operations can lead to expensive overhead among a large quantity of GPU threads. By breaking the queue generation into two steps, Enterprise is able to not only eliminate the need of thread synchronization by updating and accessing the status array in parallel, but also remove duplicated frontiers from the queue that avoids potentially useless work down the road. This is further combined with memory optimization to accelerate both top-down and bottom-up BFS. The evaluation shows that although it may take a small amount of time for queue generation, our GPU threads scheduling can speed up the overall BFS runtime by 37.5×.

Second, **GPU workload balancing via frontier classification**. To mitigate inter-thread workload imbalance, Enterprise classifies the frontiers based on the out-degrees (the number of edges to adjacent vertices) into a number of queues, and assigns a different number of threads to work on each queue. Specifically, Enterprise creates four different frontier queues corresponding to Thread, Warp, Cooperative Thread Array (CTA), and Grid [4]. For example, Enterprise may assign a single thread for the frontiers whose out-degree is less than 32 and a warp for those less than 256. Enterprise may even assign all threads on one GPU to a frontier in the case of extremely high out-degrees (e.g., 10^6). Prior work utilizes a fixed number of threads (typically 32 or 256), where static assignments often result in skewed workload among threads [21, 33, 23, 29]. The frontier classification greatly mitigates this imbalance, leading to additional speedup of 1.6× to 4.1× on top of the proposed GPU threads scheduling technique.

Third, **GPU-aware direction optimization** is developed in Enterprise to run bottom-up BFS efficiently on GPUs. Specifically, we propose a new parameter that uses the ratio of hub vertices in the frontier queue to determine the one-time switch from top-down to bottom-up on GPUs. This parameter is shown to be stable across different graphs, removing the need for parameter tuning as in the prior approach [10]. More importantly, Enterprise selectively caches the hub vertices in GPU shared memory to reduce the expensive random global memory access. Interestingly, this shared memory based cache with a small size of 48 KB, when caching a few thousand of critical hub vertices, can help to reduce up to 95% of global memory transactions in bottom-up BFS.

To the best of our knowledge, Enterprise is the first GPU-based BFS system that not only leverages a variety of GPU thread groups to balance irregular workloads but also employs different GPU memories to mitigate random accesses, both of which are inherent characteristics of graph traversal on power-law graph and especially challenging to optimize on modern GPUs. Enterprise can be utilized to support a number of graph algorithms such as single source shortest path, diameter detection, strongly connected component, and betweenness centrality.

The rest of the paper is organized as follows: Section 2 introduces the background on GPUs, BFS, and the graphs used in this paper. Section 3 presents the challenges of running BFS algorithms on GPUs. Section 4 describes the three techniques and their benefits. We present the overall performance and energy efficiency of Enterprise in Section 5. Section 6 discusses the related work and Section 7 concludes.

2. BACKGROUND

2.1 Breadth-First Search

Traditional (top-down) BFS algorithm performs expansion and inspection at each level, that is, from each frontier (last recently visited) vertex v, examining whether an adjacent vertex w is first-time visited. If so, v becomes the parent and w is also enqueued into the frontier queue.

The frontiers can be generated in two ways. In the first approach of Figure 1(b), known as atomic operation based frontier generation [30], two threads are dispatched at level 2 to check the adjacency lists of vertices 1 and 4 in the queue

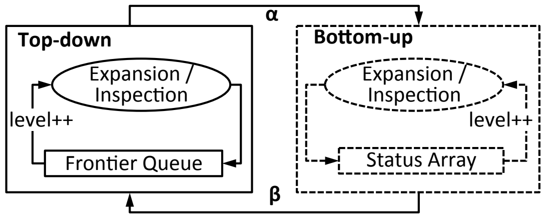


Figure 2: Hybrid (direction-optimizing) BFS.

FQ_2, and both would like to put vertex 2 into FQ_3. In this case, atomic operations (e.g., atomicCAS in CUDA [4]) are utilized to ensure that FQ_3 has no duplicated frontiers, where whichever thread that finishes first would become the parent of vertex 2. Without atomic operations, vertex 2 would be enqueued twice, resulting in redundant work at level 3.

Since inter-thread synchronization is costly on GPUs, a second approach [24, 36] uses a data structure called the Status Array (SA) to track the status of each vertex in the graph. Status array is basically a byte array indexed by the vertex ID. The status of a vertex can be *unvisited*, *frontier* or *visited* (represented by its BFS level). At every level, a thread will be assigned to each vertex, whereas only those that are working on the frontiers will perform expansion and inspection. Thus, as shown in Figure 1(c), while ten threads will be used at level 2, only two will be working on vertices 1 and 4. The advantage of this approach is that atomic operation is no longer needed - both vertices 1 and 4 can be the parent of vertex 2, and the update of the status of vertex 2 can be performed sequentially. Here, unlike the first approach whoever finishes last becomes vertex 2's parent.

Hybrid BFS is initialized with the top-down approach and switches the direction between top-down and bottom-up when the switching parameters satisfy the predefined thresholds. Figure 2 presents the workflow of hybrid (direction-optimizing) BFS. Top-down BFS aims to identify the edges that connect the frontiers and unvisited vertices, while bottom-up aims to identify those between the frontiers to visited vertices. This paper formally defines a frontier as:

Definition (Frontier) Let v be a vertex of the graph G. At level i, v becomes a frontier if

- Top-down BFS: v was visited at level $i - 1$; or
- Bottom-up BFS: v has not been visited between level 0 and $i - 1$.

Using the same example in Figure 1, at level 2, top-down selects vertices $\{1, 4\}$ as the frontiers. In comparison, bottom-up uses unvisited vertices $\{3, 5, 6, 8, 9\}$ as the frontiers at level 3. When bottom-up discovers that vertices $\{3, 5\}$ connect to a visited vertex 2, they are marked as visited with 2 as the parent. Similarly, vertex 8 is marked as visited with 7 as the parent.

The goal of direction-switching is to reduce a potentially large number of unnecessary edge checks. Hybrid BFS may switch direction twice in the process, i.e., from top-down to bottom-up and from bottom-up to top-down, each of which is associated with a parameter. In Figure 2, α is calculated as the ratio of m_u and m_f, where m_u represents the unexplored edge count, and m_f the edges to be checked from the top-down direction; and β is calculated as the ratio of n and n_f, where n represents the number of vertices in the graph and n_f the number of vertices in the frontier queue. Currently the thresholds are heuristically determined.

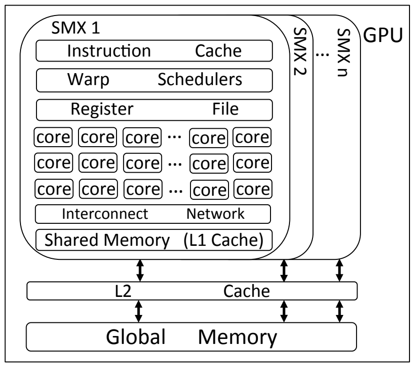


Figure 3: A simplified view of GPU architecture.

Switching from bottom-up to top-down is done in the final stages of BFS to avoid the long tail in the graphs, which we find is neither necessary nor beneficial for Enterprise. In this paper, we will show that building an efficient hybrid BFS system will require a number of GPU-aware optimizations, including a stable direction-switching parameter, hub vertex cache, as well as streamlined GPU threads scheduling and workload balancing.

2.2 General-Purpose GPUs

In this section, we will mainly explain GPU hardware, using NVIDIA Kepler K40 as an example [8]. The K40 consists of 15 Streaming Processors (SMX) each of which has 192 single-precision CUDA cores and 64 double-precision units. Each GPU **thread** runs on one CUDA core and an SMX schedules the threads in a group of 32 that is called a **Warp**. Figure 3 presents an overview of GPU architecture.

An SMX can support up to 64 warps. All the threads in a warp are executed in the single-instruction, multiple-thread fashion. But if the threads in a warp have different control paths, the warp executes all the taken branches sequentially and disables each individual thread that is not on the taken path. This so called branch divergence problem, if exists, could largely reduce SMX utilization.

Each SMX features four **Warp Schedulers** which select four warps in round-robin and issue the instructions from those that are ready for execution. The warps that are not ready due to long latency data accesses are skipped. By oversubscribing threads in each SMX, data access can be overlapped with execution.

Cooperative Thread Array (CTA), thread block, consists of multiple warps, typically 1 to 64, which can be used to run a large number of threads. And the set of all the CTAs on a GPU is referred to as a **Grid**. The number of CTAs and

Table 1: Graph Specification

Name	Abbr.	Description	# Vertices (M)	# Edges (M)	BFS Depth	Directed
Facebook	FB	Facebook user to friend connection	16.8	421	10	Y
Friendster	FR	Friendster online social network	16.8	439.2	25	Y
Gowalla	GO	Gowalla location based online social network	0.2	1.9	–	N
Hollywood	HW	Hollywood movie actor network	1.1	115	10	N
Kron-20-512	KR0	Kronecker generator	1	1073.7	6	N
Kron-21-256	KR1	Kronecker generator	2.1	1073.7	7	N
Kron-22-128	KR2	Kronecker generator	4.2	1073.7	7	N
Kron-23-64	KR3	Kronecker generator	8.4	1073.7	7	N
Kron-24-32	KR4	Kronecker generator	16.8	1073.7	8	N
LiveJournal	LJ	LiveJournal online social network	4.8	69.4	15	N
Orkut	OR	Orkut online social network	3.1	234.4	9	N
Pokec	PK	Pokec online social network	1.6	30.1	11	Y
R-MAT	RM	GTgraph: R-mat generator	2	256	6	Y
Twitter	TW	Twitter follower connection	16.8	186.4	17	Y
Wikipedia	WK	Links between Wikipedia pages in 2007	3.6	45	12	Y
Wiki-Talk	WT	Wikipedia talk network	2.4	5.0	–	Y
YouTube	YT	YouTube online social network	1.1	6.0	–	N

the number of threads in each CTA are configurable. Each thread in a CTA has a unique Thread ID and each CTA has its own CTA ID. With these built-in variables, one is able to identify each thread in a grid and schedule different threads to work on different data.

A Kernel is defined as any function that runs on GPUs. Typically, one kernel can use different **parallel granularity** (i.e., a thread, warp, CTA, or grid) by employing a certain quantity of threads. Kepler introduces Hyper-Q to support concurrent kernel execution, in other words, when several kernels are executed on the same GPU, Hyper-Q is able to schedule them to run on different SMXs in parallel to fully utilize all GPU resources.

GPU Memory Hierarchy. Each SMX has a large number of **registers**, e.g., 65,536 for each K40 SMX. Each thread can use up to 255 registers and perform four register access for each clock cycle. In addition, each SMX provides software configurable **shared memory** (L1 cache) for intra-warp and intra-CTA data communication. Each K40 SMX has 64 KB of shared memory. Different from the L1 cache on CPUs, one can allocate 16, 32, or 48 KB of the shared memory at the program runtime. Once loaded, the data in the shared memory is readable and writable to all the threads in one CTA.

GPU also has the **L2 cache** and **global memory** that are shared by all SMXs. The K40 has 1.5 MB L2 and 12 GB global memory. Each global memory access is replied with a data block that contains 32, 64 or 128 bytes based on the type. If a warp of threads happen to access the data in the same block, only one hardware access transaction is performed. By coalescing global memory accesses into fewer transactions in this way, K40 is able to achieve close to 300GB/s DRAM bandwidth.

Table 2: CPU (Xeon E7-4860) vs. GPU (K40) memory: size and access latency (in CPU and GPU cycles) [28, 4]

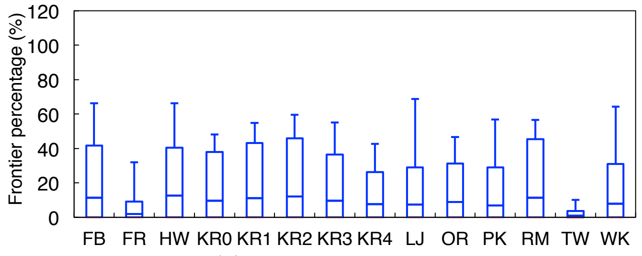
Memory	CPU		GPU		
	Size	Lat	Size	Lat	BFS Data Structures
Register	12	1	65,536	-	Status Array
L1 cache	64KB	4	64KB	-	Hub Vertex Cache
L2 cache	256KB	10	1.5MB	-	-
L3 cache	24MB	40	-	-	-
DRAM	up to 2TB	55	12GB	200 - 400	Status Array, Frontier Queue, Adjacency List

Table 2 summarizes the CPU and GPU memory hierarchies. Note that K40 has no L3 cache. We cannot find official latency numbers for register and shared memory, but our tests show that they are at least an order of magnitude faster than the global memory. In this work, we leverage the GPU support of concurrent kernels and different parallel granularity to match dynamic BFS workloads, and utilizes different GPU memory for various BFS data structures, e.g., using shared memory for hub vertices.

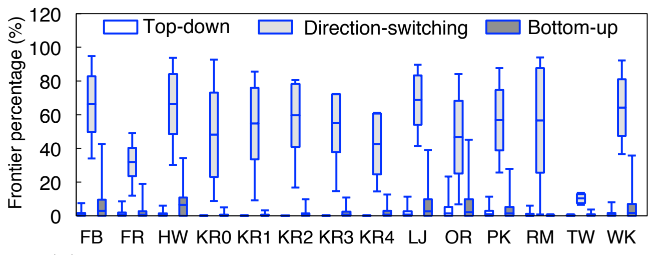
GPU Hardware Performance Counters. GPUs today can support more than 100 hardware metrics [5]. In this work, we aim to understand the kernel performance, GPU I/O throughput, and energy efficiency of our system, including the timeline of different kernels, utilization of memory load/store function unit (*ldst_fu_utilization*), percentage of stalls caused by data requests (*stall_data_request*), global memory load transactions (*gld_transactions*), IPC and power. We use two NVIDIA tools, i.e., *nvprof* and *nvvp*.

2.3 Graph Benchmarks

We use a total of 17 graphs in this paper, as summarized in Table 1, which have vertices ranging from 1 to 17 million and edges from 30 million to over 1 billion. For an undirected graph, we count each edge as two directed edges. Eleven real world graphs are included such as Facebook [19], Twitter [27], Wikipedia [7], as well as the LiveJournal, Orkut, Friendster, Pokec, YouTube, Wiki-Talk and Gowalla social network graphs from the Stanford Large Network Dataset Collection [6]. In addition, we utilize two widely used graph generators, Kronecker [1] and Recursive MATrix (R-MAT) algorithm [13] [3]. Both generators take four possibilities A, B, C and $D = 1.0 - A - B - C$. The Kronecker generator produces the Kron-*Scale*-*EdgeFactor* graphs that have 2^{scale} number of vertices with the average out-degree of *EdgeFactor*. In this work, we use (A, B, C) of (0.57, 0.19, 0.19) for Kronecker, and (0.45, 0.15, 0.15) for R-MAT graphs. It is worthy to point out that both real-world and synthetic graphs exhibit small-world characteristics - as the majority of the vertices have small out-degree and account for the small percentage of the total number of edges, there exist a number of hub vertices with high out-degree.



(a) Per-level average



(b) Top-down, direction-switching and bottom-up

Figure 4: Boxplot of percentage of frontiers (a) for per-level average and (b) for top-down, direction-switching, and bottom-up.

3. DESIGN CHALLENGES

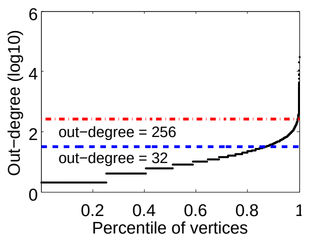
Challenge #1: Putting GPU Threads to Good Use

Eliminating the need of atomic operations on GPUs for the frontier queue generation which has been the focus of prior work [24, 33, 29, 36, 35, 30] solves only half of the problem - the queue shall consist of only the frontiers, that is, the vertices that need to be explored in next level. Using the status array for next level traversal, although avoiding atomic operations, would assign one GPU thread for each vertex, regardless whether it is a frontier [36, 24]. This inefficient approach would over-commit GPU threads because at most levels the majority of the vertices would not be a frontier. Alternatively, another work [33] generates the frontier queue with warp and historical culling, but again this approach could not completely avoid duplicated vertices across warps being enqueued. Figure 4(a) shows the boxplot of the percentage of the frontiers at each level across different graphs, where the mean and maximum percentage, as well as standard deviation, are presented. Note that the numbers here include the frontiers for both top-down and bottom-up directions. It turns out that the graphs have on average 9% frontiers per level with standard deviation of 15%. In particular, the R-MAT graph has the largest average ratio of 11% and maximum of 57%, while Twitter has the smallest average of 1% and maximum of 10.2%. If a thread were assigned to each vertex at every level, on average at least 31% of the threads would idle. Therefore, it is critical to have a queue that consists of the frontiers only, instead of wasting valuable GPU threads on those with no work to perform.

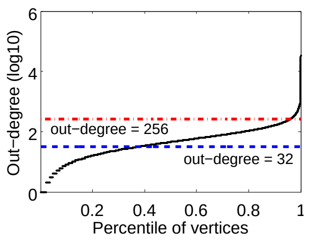
This challenge is further exacerbated by the need of direction switching between top-down and bottom-up, which generates the frontiers by focusing on two distinct sets of vertices (visited in top-down vs. unvisited in bottom-up). To illustrate this problem, we present the percentage of the frontiers by BFS traversal directions in Figure 4(b). In general, bottom-up levels have more frontiers than top-down, i.e., 1.5% vs 0.4%. In particular, the queue for the level when switching from top-down to bottom-up has most frontiers at 52% on average. Using the status array alone at this level would remain inefficient. The above observation leads us to develop Enterprise with new GPU threads scheduling that aims to prepare a frontier queue that is direction optimized for GPU memory hierarchy.

Challenge #2: Balancing Workloads Among GPU Threads

This challenge stems from that fact that large variance exists in the frontiers' out-degrees. If a frontier has more edges, the GPU thread assigned to it would naturally need to carry out more expansion and inspection. To illustrate this imbalance, we plot the CDF of the edge counts for two social networks in Figure 5, where the average out-degrees for Gowalla and Orkut are 19 and 72 respectively. In Gowalla, 86.7% and 99.5% of the vertices have fewer than 32 and 256 edges. In contrast, while Orkut has a smaller portion (37.5%) of the vertices with fewer than 32 edges, it has more (58.2%) with out-degree between 32 and 256. Furthermore, a fraction (0.5% and 4.2%) of vertices have more than 256 edges in Gowalla and Orkut with a long tail to around 30K edges.



(a) Gowalla out-degree



(b) Orkut out-degree

Figure 5: Cumulative Distribution Function (CDF) of out-degrees of vertices sorted by out-degree: (a) Gowalla (b) Orkut.

Statically assigning one fixed number of threads (e.g., a warp or CTA) is inefficient because the per-level runtime would be dominated by the threads with heavy workload. Another inefficiency may also arise from the mismatch from the thread count and the workload. For example, if one CTA were assigned to work on a frontier with fewer than 32 adjacent vertices, more than 200 threads in this CTA would have no work to do. On the other extreme, some frontiers with very high out-degrees will require more than one CTA, e.g., some graphs we examine have vertices with up to 10^6 edges. To address this challenge, Enterprise introduces a new approach of classifying frontiers based on the out-degrees and assigning an appropriate GPU parallel granularity at runtime.

Challenge #3: Making Bottom-Up BFS GPU-Aware

Implementing direction-optimizing BFS on GPUs is challenging by itself. Direction-optimizing BFS has first been proposed and implemented on multi-core CPUs in [10], but without further optimizations would not run efficiently on GPUs that can run thousands of threads but with a relatively smaller (e.g., 12GB) and slower (e.g., 200-400 cycles)

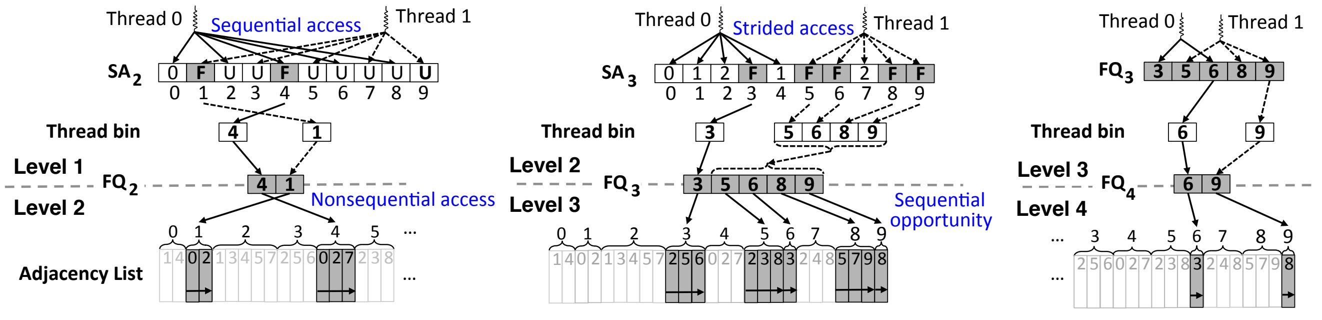
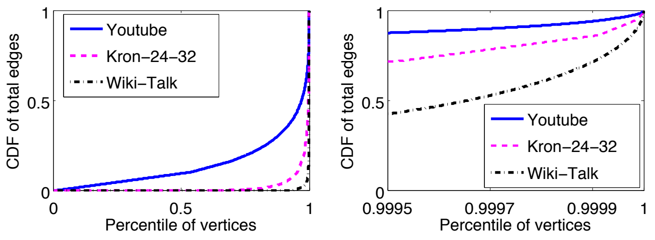


Figure 7: Streamlined GPU threads scheduling using the graph example from Figure 1, with three workflows: (a) top-down, (b) direction-switching at the explosion level, and (c) bottom-up. Sequential access means two threads access consecutive adjacent elements at each iteration. Strided access means two threads access elements in stride manner at each iteration.



(a) All range (b) Zoom into [0.9995, 1]

Figure 6: CDF of total edges in Youtube, Wiki-Talk and Kron-24-32 graphs. The vertices are sorted by out-degrees: (a) Vertices of all range (b) Zoom in range [0.9995, 1].

global memory. In contrast, as previously shown in Table 2, modern CPUs have tens of cores and threads with large L3 cache and main memory with short access latency. Fortunately, what GPU lacks on global memory is compensated by a massive number of registers and software-configurable shared memory (L1 cache), which can be utilized to accelerate memory intensive algorithms like BFS.

The CPU-based bottom-up BFS uses the status array to supply the unvisited vertices for inspection, and direction switching between top-down and bottom-up depends on the numbers of unexplored edges. In Enterprise, the GPU-based bottom-up BFS leverages a small set of highly connected vertices called hub vertices. Formally, we define a hub vertex as follows.

Definition (Hub Vertex) Let v be a vertex of the graph G. Consider v be a hub vertex if its out-degree is greater than a threshold τ.

Here τ is graph specific, e.g., in the order of 100Ks for Twitter. It is common that a few hub vertices in power-law graphs connect to a great number of vertices. Figure 6 presents both the CDF of total edges and a zoom-in view for the range of [99.95%, 100%] of the vertices. For the YouTube graph, one can see that 330 hub vertices (i.e., 0.03% of the total vertices) contribute to 10% of the total edges. Similarly, 770 hub vertices (0.005%) in Kron-24-32 produce 10% of the total edges, and 96 hub vertices (0.004%) in Wiki-Talk account for 20% of the total edges.

The most unique features of our GPU-based bottom-up BFS are: 1) Enterprise switches the direction at what we refer in this paper as the *explosion level* where a large quantity of hub vertices need to be visited. In this work, we have found that the number of hub vertices in the frontier queue can serve as a better indicator for direction switching, which can easily implemented on GPUs. And more importantly, 2) caching hub vertices turns out to be very beneficial for bottom-up BFS.

4. ENTERPRISE: GPU-BASED BFS

4.1 Streamlined GPU Threads Scheduling

Combining the power of the status array and frontier queue, Enterprise is able to produce streamlined scheduling of GPU threads through generating the frontier queue at each level with a scan of the status array. At each level, Enterprise starts with identifying the frontiers and updating the status array in a manner similar to [36, 24]. Once this step completes, Enterprise dispatches GPU threads to scan the vertices in the status array. When a frontier is found, the thread will store this vertex in its own thread bin. All the thread bins are stored in the global memory. Next, prefix sum is used to calculate the offset of each bin in the frontier queue [34, 22]. Lastly, the frontiers in each bin are copied to the queue in parallel. In all, the benefits are clear in avoiding thread synchronization (from using the array) and reducing idle threads at the next level (from using the queue). However, as we have shown, BFS direction can lead to a large disparity in the number of frontiers at each level.

To this end, Enterprise schedules GPU threads using three queue generation workflows for top-down, direction-switching, and bottom-up, to optimize the memory accesses in all cases. The status array, frontier queue and adjacency list reside in GPU global memory, and accessing the global memory randomly would only achieve a meager 3% of sequential read bandwidth. To maximize the overall performance, it is critical that we optimize the access patterns at different stages of BFS.

Top-down workflow. In this direction, Enterprise uses the GPU threads to scan the status array in an interleaved manner. For the example in Figure 7(a), two threads are dispatched at level 1: thread 0 checks the status of five vertices {0, 2, 4, 6, 8}, while thread 1 checks the others {1, 3, 5, 7, 9}. This division of work performs a sequential memory

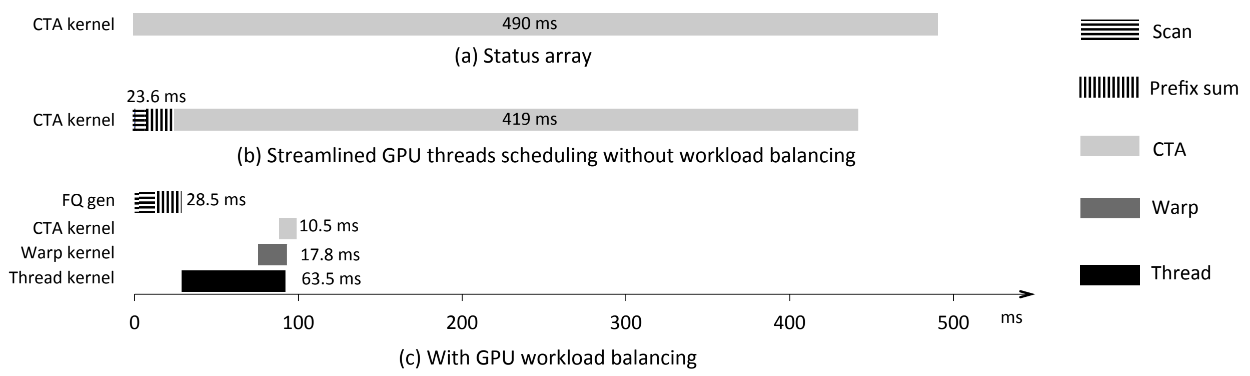


Figure 8: Execution timeline before and after streamlined GPU threads scheduling and workload balancing for the explosion level of Facebook.

access of the status array. When prefix sum is completed, threads 0 and 1 will copy their own thread bins into FQ_2 concurrently. In this case, FQ_2 stores two frontiers out-of-order as {4, 1}, which will introduce nonsequential memory access at level 2, that is, BFS accesses the adjacency list of vertex 4 before vertex 1. Fortunately, the benefit of sequential access of the status array outweighs the potential drawback of random access of the adjacency list. For top-down, adjacent vertices in the status array are unlikely to become frontiers at the same level, as there are only a small number (average 0.4%) of frontiers, as shown in Figure 4(b).

Direction-switching (explosion-level) workflow. The situation is different in this case. Here the GPU threads are allocated a certain portion of the status array to scan. Using the same example, at level 2, again two threads will be used: this time thread 0 checks the status of five vertices {0, 1, 2, 3, 4} while thread 1 checks five vertices {5, 6, 7, 8, 9}. Unlike the top-down workflow, this approach would incur strided memory access during the scan. Next, prefix sum is performed on thread bins and in this example FQ_3 consists of {3, 5, 6, 8, 9}. The performance benefit comes from that the (bottom-up) frontiers may appear in order in the queue, which in turn leads to sequential memory access at the next level. At the explosion level, chances are that adjacent vertices are all unvisited, because most are as we have shown in Figure 4(b). This workflow takes advantage of this fact to accelerate the next level traversal, e.g., at level 4, loading the adjacent list of vertices 5 and 6 are sequential adjacent memory access, and so are vertices 8 and 9.

At the explosion level, this approach will spend average 2.4× more time to scan the status array, as compared to the top-down workflow. For example, using top-down workflow would take 0.57 ms for the explosion level on Hollywood. In contrast, using direction-switching workflow will take a longer time of 0.86 ms. But this approach will improve the performance of next level traversal by average 37.6%, e.g., Hollywood runtime at the level right after the explosion decreases from 2.7 to 2 ms. When combined, because the latter step takes longer wall clock time, the overall performance achieves an average speedup of over 16% across all the graphs, with the best improvement of 33% on Facebook.

Bottom-up workflow. The key insight is that for bottom-up, the queue for the current level is always a subset of the previous queue, as the frontiers are always unvisited vertices. Instead of continuing to use the status array, we directly use the frontier queue from the preceding level to generate the queue for current level shown in Figure 7(c). This is done by simply removing the vertices that belong to current level. This approach eliminates the need of scanning the whole status array. Only a small (and fast shrinking) subset is inspected at each level. For example, at level 3, FQ_4 is created by removing vertices {3, 5, 8} from FQ_3. Our tests show this approach delivers 3% improvement across various graphs.

To summarize, this technique increases the number of GPU threads that actively work on frontiers and issue memory load/store requests, which we will see in the experiments that the utilization of memory load/store function unit increases dramatically. Using this design, the queue can still be generated very quickly from 2.2 to 53.7 ms for different graphs, which accounts for about 11% of the overall BFS traversal time, yet delivers 2× to 37.5× speedup. Figure 8 presents an execution trace of BFS execution for the explosion level of Facebook. Clearly, despite generating the frontier queue takes 23.6 ms, because a good workqueue is prepared, new threads scheduling reduces the runtime of expansion and inspection from 490 ms to 419 ms, a net saving of 46 ms.

4.2 GPU Workload Balancing

Now that Enterprise can generate a good frontier queue quickly, but the benefit would be minimal if the queue would lead to imbalanced workload. In this work, we believe that parallel granularity of GPU shall be leveraged when scheduling work from the frontier queue to ensure high thread-level parallelism. Ideally, each thread, regardless of standalone, within a warp or CTA, shall have an equal amount of work (expansion and inspection) at each level. To achieve this goal, Enterprise classifies the frontiers based on their out-degrees (potential workload) and allocates a matching parallel granularity. Enterprise focuses on the use of threads over warps or CTAs, different from prior work [23, 33]. This is motivated by the fact that the majority of the vertices in a graph have small out-degrees. For the graphs studied, the average percentage of the vertices with fewer than 32 edges is 68% and may go as high as 96% in Twitter.

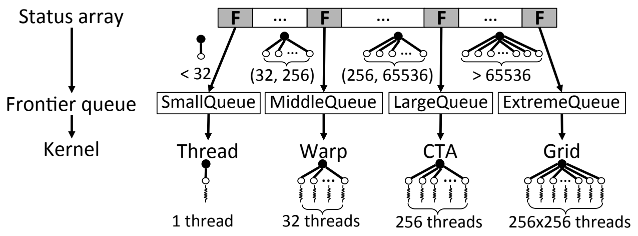


Figure 9: GPU workload balancing.

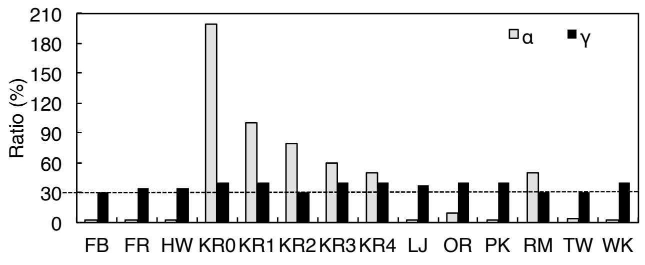


Figure 10: Comparison of direction-switching parameters.

Enterprise classifies the frontiers that are generated with the previous technique into four queues, SmallQueue, MiddleQueue, LargeQueue and ExtremeQueue, based on the out-degrees of each frontier. Specifically, the frontiers in SmallQueue have fewer than 32 edges, MiddleQueue between 32 and 256, LargeQueue between 256 and 65,536 and ExtremeQueue more than 65,536. During frontier queue generation, each thread puts the discovered frontiers into one of four thread bins according to their out-degrees. At the next level, four kernels (Thread, Warp, CTA and Grid) with different number of threads will be assigned to work on different frontier queues in order to balance the workloads among the threads, as shown in Figure 9. All kernels are executed concurrently with Hyper-Q support.

We maintain an ExtremeQueue for dealing with vertices with extremely high out-degrees. For instance, one vertex in KR2 has over 2.5 million edges. If one CTA were assigned to inspect this vertex, it would require more than 10,000 iterations. This type of vertex exists for many graphs as we have seen as long tails in Figure 5 and 6. Subsequently, expanding from these vertices would require rather long runtime, which without special handling may greatly prolong the traversal of the whole level. Using the whole grid here can considerably speed up the execution, e.g., 1.6× speedup is achieved on KR0.

In Figure 8(b) and (c), one can see the changes in runtime before and after workload balancing. Again, although this optimization adds another 5 ms of overhead to classify the frontiers, we are able to shorten the overall runtime drastically, from 419 ms to 76.5 ms. In particular, the Thread kernel takes 63.5 ms, Warp 17.8 ms, and CTA kernel 10.5 ms, where there is significant overlapping among the three kernels. In short, this technique further removes idling threads in each CTA and warp compared to prior methods, which similar to the first technique will lead to higher utilization on GPU memory units.

4.3 Hub Vertex Based Optimization

Direction-switching parameter. In this work, we have found that it is cumbersome to tune the parameter α to determine when to switch from top-down to bottom-up. Instead, as hub vertices make up a good portion at the explosion level, we propose to use the ratio of hub vertices in the frontier queue as an indicator for direction switching. We define the parameter γ formally as:

$$\gamma = \frac{F_h}{T_h} \times 100\% \qquad (1)$$

where F_h is the number of hub vertices in the frontier queue (collected per level) and T_h represents the total number of hub vertices, which can be calculated very quickly at the first level. Our experiment shows that γ is stable without the need for manual tuning. Figure 10 shows that all graphs should switch direction when $\gamma \in (30, 40)\%$, a very small range compared to α that fluctuates between 2 and 200. In this work, we set the direction-switching condition as γ being larger than 30.

Enterprise traverses on average 4 levels top-down and 8 levels bottom-up across various graphs, about one level sooner than prior method [10]. For the Kronecker graphs, using α would inspect 4% and 17% of the edges in top-down and bottom-up respectively, avoiding to visit the remaining 79% edges. Using our hub vertex based parameter γ would inspect 1% and 36% edges in top-down and bottom-up. At first glance, Enterprise would inspect more edges in total, hurting the performance. Fortunately, as we have shown, direction switching happens at the explosion level that is dominated by hub vertices, and bottom-up traversal focuses on identifying the edges connecting the frontiers to recently visited hub vertices. As a result, a good cache of hub vertices lends itself nicely for both scenarios.

Hub vertex cache. In this work, we propose to cache hub vertices in GPU shared memory during direction switching and bottom-up, which can greatly reduce the overhead of random global memory accesses. This benefit is achieved because large amount of frontiers in bottom-up are very likely to connect to hub vertices. However, as GPU shared memory is small (64 KB), we need to carefully balance the number of hub vertices cached and the occupancy of the GPU that is defined as the ratio of active warps running on one SMX and the maximum number of warps that one SMX can support theoretically (64). If a grid contains 256 × 256 threads, the full occupancy of K40 means 8 CTAs running on one streaming processor and thus each CTA only has 6 KB shared memory to construct a cache holding around 1,000 hub vertices.

Hub vertex cache (HC) is implemented in two steps. First, during the frontier queue generation, Enterprise caches the vertice IDs of those have just been visited at the preceding level and also with high out-degrees. We use a hash function to figure out which index to store each vertex ID, that is, $HC[hash(ID)] = ID$. Second, during the frontier identification, Enterprise will load the frontier's neighbors, and check whether the vertex ID of any neighbor is cached. If so, the inspection will terminate early with the cached neighbor identified as the parent for this frontier. In this case, the cache avoids accessing this neighbor's status in the global memory.

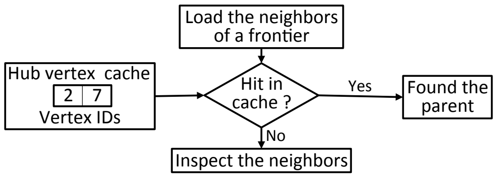


Figure 11: Hub vertex cache design, using the level 4 traversal in example graph from Figure 1.

Figure 11 presents the workflow of the hub vertex cache. In this example, Enterprise puts vertice IDs {2,7} in the hub vertex cache because these two vertices are visited in the preceding level and with the high out-degrees. At the current level, Enterprise will load the neighbors {2, 5, 6} of vertex 3 for inspection. As vertex 2 is cached, Enterprise will mark vertex 2 as the parent of vertex 3 and terminate the inspection. On the other side, if a frontier like vertex 6 does not have a cached neighbor, Enterprise will continue to inspect the statuses of its neighbors that reside in the global memory. As shown in Figure 12, the hub vertex cache is very effective on various graphs, saving 10% to 95% of global memory accesses. It is worthy to point out that caching hub vertices has limited benefit for top-down BFS, as it likely encounters very few hub vertices.

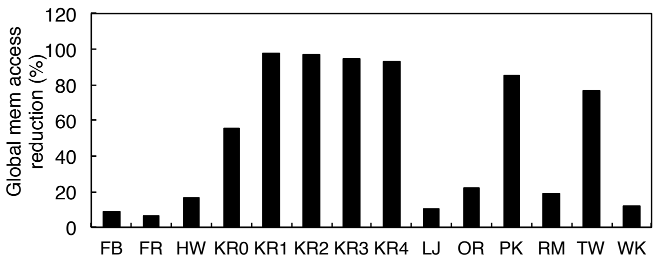


Figure 12: Global memory accesses reduced by hub cache.

4.4 Multi-GPU Enterprise

Enterprise exploits 1-D matrix partition method [11] to distribute the graphs across multiple GPUs. Specifically, each GPU is responsible for an equal number of vertices from the graph, and thus a similar number of edges. We leave the study of 2-D partition as future work. During traversal, Enterprise proceeds in three steps: (1) Each GPU identifies the current level vertices in a private status array by expanding from a private frontier queue. (2) All the GPUs communicate their private status arrays to get the global view of most recently visited vertices. In this step, each GPU uses a CUDA instruction `__ballot()` to compress the private status array into a bitwise array where a single bit is used to indicate whether one vertex is just visited. This compression reduces the size of communication data by 90%. (3) Each GPU scans the updated private status array to generate its own private frontier queue.

5. EXPERIMENTS

We have implemented Enterprise in 3,000 lines of code in C++ and CUDA. The source code is compiled with NVIDIA nvcc 5.5 and GCC 4.4.7 with the optimization flag of O3. In this work, we use three GPUs: NVIDIA Kepler K40, K20 and Fermi C2070. We perform our tests on the graphs described in Table 1. All the graphs are represented by compressed sparse row (CSR) format. The datasets that provide edge tuples are transformed into the CSR format, with the sequence of the edge tuples preserved. The majority of the graphs are sorted, e.g., Twitter and Facebook. We do not perform pre-processing such as removing duplicate edges or self-loops. All the data is represented by uint64 type, loaded into GPU's global memory. The timing starts when the search key is given to the GPU kernel and ends when the search is completed and written into the GPU memory. For each experiment, we run BFS 64 times on pseudo-randomly selected vertices and calculate the mean. The metric *traversed edges per second* (TEPS) is computed as follows: Let m be the number of directed edges traversed by the search, counting any multiple edges and self-loops, and t be the time elapsed during BFS search mentioned above. Then, TEPS is calculated by m/t.

5.1 Enterprise Performance

We implement direction-optimizing BFS with the status array approach as the baseline (BL) since atomic operation based frontier queue would be much slower. Here we use CTA to work on each vertex in the status array, which is much faster than assigning a thread or warp. Figure 13 plots the performance improvement contributed by each optimization including streamlined GPU threads scheduling (TS), GPU workload balancing (WB), hub vertex cache (HC).

The streamlined GPU threads scheduling outperforms the baseline by 2× to 37.5× across all graphs. In particular, Twitter (TW) obtains the biggest speedup from 0.04 to 1.5 billion TEPS. The reason is that the maximum frontier ratio in Twitter is only 10.2%, and on average it only has 1% frontiers at each level. Kron-20-512 (KR0) gains 2× speedup, reaching 34 billion TEPS. In general, generating the frontier queue consumes on average 11% of the BFS run time.

The GPU workload balancing technique more than doubles the traversal rate for all graphs, 2.8× on average beyond the first technique. For example, LiveJournal (LJ) achieves the biggest improvement of 4.1×, from 0.9 to 3.7 billion TEPS. For this graph, the total workload is distributed evenly so that SmallQueue contains 78% frontiers (or 22% workload), MiddleQueue has 21% frontiers (or 58% workload), LargeQueue 1% frontiers (20% workload).

The hub vertex caching technique helps improve the performance up to 55%. Both Facebook (FB) and Friendster (FR) see a small gain as they do not contain vertices with extremely high out-degree, e.g., the maximum out-degree in Facebook is 9,170. For other graphs, the improvement is more than 10%, as high as 30% to 50% for Kronecker graphs that have thousands of vertices with more than 10^5 edges. This shows that caching these hub vertices is very beneficial.

In all, Enterprise improves the TEPS of the BFS algorithm by 3.3× to 105.5×. The highest TEPS is achieved at KR0 with over 76 billion TEPS and the smallest at FR with 3.1 billion TEPS.

Comparison. Figure 14 compares Enterprise with several GPU based BFS implementations, including B40C [33], Gunrock [44], MapGraph [18] and GraphBIG [2]. We evalu-

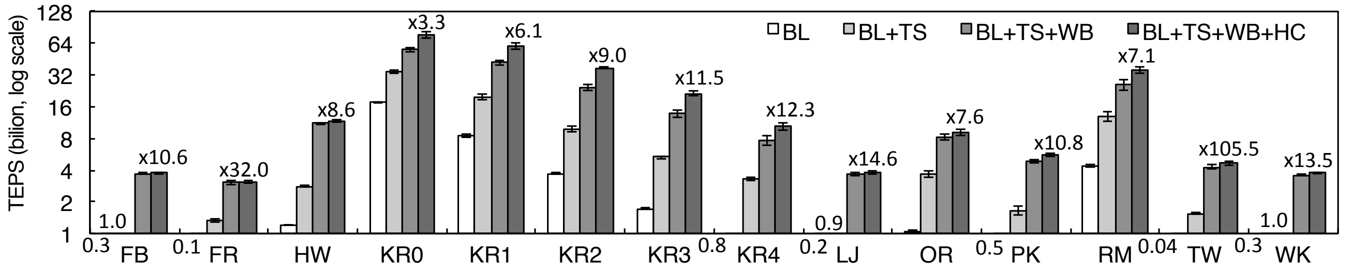


Figure 13: Enterprise performance on various graphs. Direction-optimizing BFS on GPU using the status array method serves as the baseline (**BL**). Three techniques are represented as **TS** for streamlined GPU **T**hreads **S**cheduling, **WB** for **W**orkload **B**alancing, and **HC** for **H**ub vertex **C**ache.

ate power-law graphs such as FB, TW, and KR-21-128 which has 2 million vertices with average out-degree of 128, as well as high-diameter graphs such as audikw1 [7], roadCA [6] and europe.osm [7].

For power-law graphs, Enterprise performs 4×, 5×, 9× and 74× better than B40C, Gunrock, MapGraph and GraphBIG, respectively. For high diameter graphs, Enterprise achieves 1.41 billion TEPS on average and outperforms Gunrock (0.72) 1.95×, MapGraph (0.25) 5.56×, GraphBIG (0.03) 42×. On these graphs, Enterprise delivers similar perform as B40C. It runs slightly slower on europe.osm because this graph has very small out-degrees, with the maximum out-degree of 12 and the mean 2.1.

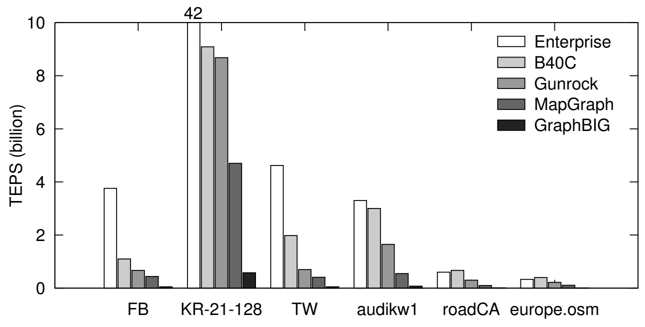


Figure 14: Performance comparison.

5.2 Enterprise Scalability

Figure 15 shows both strong and weak scalability of Enterprise. We use the largest graph from Table 1, i.e., KR4 to test the strong scalability. On 2, 4 and 8 GPUs, Enterprise achieves 15, 18 and 18.4 billion TEPS, respectively, that is, a speedup of 43%, 71% and 75%.

We evaluate weak scalability in two ways, edge scale and vertex scale. When the GPU count increases, we increase the edgeFactor – the average out-degree – with fixed vertex count, or increase the number of vertices with the constant edgeFactor. As shown in Figure 15, we achieve better scalability for edge scale, where we obtain super linear speedup, that is, 9.1×, 96 billion TEPS with 8 GPU. This is because when edgeFactor increases, the number of hub vertices increases in the graph too and the hub vertex cache will reduce more global memory accesses. On the other hand, direction-switching can possibly avoid more unnecessary edge checks.

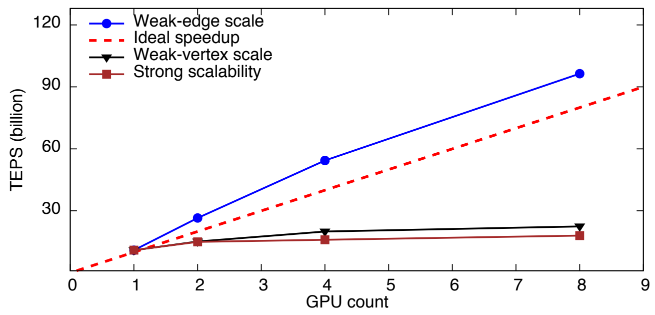


Figure 15: Strong and weak scalability of Enterprise.

5.3 Analysis of GPU Counters

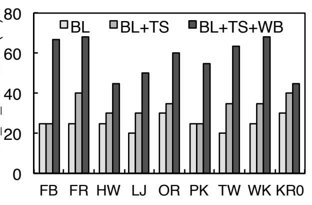
As BFS is an I/O-intensive algorithm, it is critical that GPU threads are able to access data quickly. As shown in Figure 16(a), our frontier techniques (TS and WB) significantly improve the utilization of GPU load/store function units by average 8% and 24% respectively, reaching as high as 68%. Furthermore, our hub vertex caching (HC) presented in Figure 16(b), reduces the stalls of data requests by 40%, the occurring of such events drops from 4.8% to 2.9%. This also explains the double of IPC observed on GPUs in Figure 16(c).

For comparison, we also profile [33] on Hollywood graph, which delivers 2.7 billion TEPS while consumes 40 Watts power, achieves 40% utilization of load/store unit and 0.68 IPC. On the same graph, Enterprise achieves 50% load/store unit utilization and 1.32 IPC, with 12 billion TEPS and 76 Watts power consumption.

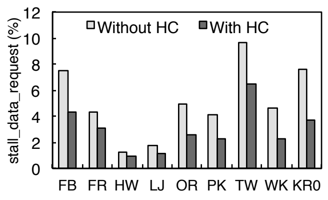
Figure 16(d) plots GPU's power consumption corresponding to different techniques. Here we only report GPU's power to understand the impact of each technique. On average, the power consumption drops from 86 to 81 Watts with our GPU threads scheduling, the biggest saving of 14.5 Watts on the Twitter graph. This comes mostly from better IO performance and fewer idle GPU threads in the system. The other two techniques (WB and HC) further reduce the power to 78 Watts.

6. RELATED WORK

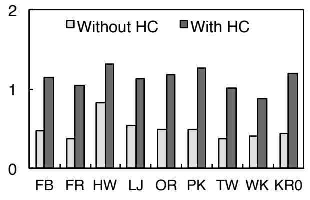
Our system Enterprise advances the state of the art in the design and implementation of graph traversal. Prior work uses either the frontier queue [30, 33] or status array [24]. Even when using both data structures, existing solutions use



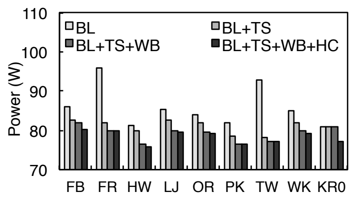
(a) Load store unit utilization



(b) Stall



(c) IPC



(d) Power

Figure 16: Microarchitecture profiling statistics of Enterprise: (a) Load/store function unit utilization (b) Stall caused by data request (c) IPC (d) GPU power consumption.

them at different directions, e.g., [36, 29] use the status array at the explosion level and the frontier queue method at other levels, and [10] uses the frontier queue for top-down and the status array for bottom-up. Enterprise utilizes both data structures throughout and delivers unprecedented performance on GPUs.

Recently several workload balance techniques have been proposed for GPUs such as task stealing [15, 12] and workload donation [41, 14]. However, this type of technique is often used in a small group of threads, and is extremely challenging to coordinate among thousands of threads as we have in this work. Instead, Enterprise targets the root of BFS workload imbalance and classifies different frontiers to mitigate the problem.

There are a number of projects [39, 20, 38] that leverage hub vertices to reduce the communication overhead, especially for distributed BFS. For example, [39] duplicate the status of hub vertices across all the machines at every level, and [20] and [38] divide hub vertices into multiple partitions and communicate in a tree-based manner. In contrast, Enterprise only enables the hub vertex cache for bottom-up levels when expansion and inspection center around hub vertices. Additionally, as GPU shared memory is limited, Enterprise updates the cache at each level with those who most likely will be visited in the following level.

Power efficiency [45] is of great importance to system design, e.g., [43] shuts down GPU streaming processors predictively to save power. Our work has shown that GPU-based graph algorithms have huge potential in delivering high performance and energy-efficiency.

7. CONCLUSION

In this work, we develop Enterprise, a new GPU-based BFS system, that produces over 70 billion TEPS on a single GPU and 122 billion TEPS on two GPUs, delivering 446 million TEPS per Watt. This is achieved by efficient management of numerous GPU streaming processors and unique memory hierarchy. As part of future work, we plan to integrate Enterprise with high-speed storage and networking devices and run on even larger graphs.

Acknowledgments

We thank anonymous reviewers for their helpful suggestions. We also thank NVIDIA for an NVIDIA Academic Partnership Award and hardware donations that facilitated this work. This material is based on research supported in part by National Science Foundation CAREER award CNS-1350766 and grant IOS-1124813.

8. REFERENCES

[1] Graph500. `http://www.graph500.org/`.

[2] GraphBIG. `https://github.com/graphbig`.

[3] GTgraph: A suite of synthetic random graph generators. `http://www.cse.psu.edu/~madduri/software/GTgraph/`.

[4] NVIDIA Corporation: CUDA C Programming Guide.

[5] NVIDIA Profiler Tools. `http://docs.nvidia.com/cuda/profiler-users-guide/`.

[6] SNAP: Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data/`.

[7] The University of Florida: Sparse Matrix Collection. `http://www.cise.ufl.edu/research/sparse/matrices/`.

[8] NVIDIA Corporation: Kepler GK110 Architecture Whitepaper. 2013.

[9] R. Albert, H. Jeong, and A.-L. Barabási. Internet: Diameter of the World-Wide Web. *Nature*, 401(6749):130–131, 1999.

[10] S. Beamer, K. Asanović, and D. Patterson. Direction-Optimizing Breadth-First Search. In *Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[11] E. G. Boman, K. D. Devine, and S. Rajamanickam. Scalable Matrix Computations on Large Scale-Free Graphs Using 2D Graph Partitioning. In *Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[12] D. Cederman and P. Tsigas. On Dynamic Load Balancing on Graphics Processors. In *Proceedings of the SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 57–64. Eurographics Association, 2008.

[13] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

[14] A. Cohen, T. Grosser, P. H. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege. Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles to Reconcile Parallelism and Locality, avoiding Divergence and Load Imbalance. In *Proceedings of Workshop on General Purpose Processing Using GPUs (GPGPU)*, 2013.

[15] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable Work Stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009.

[16] S. Dolev, Y. Elovici, and R. Puzis. Routing Betweenness Centrality. *Journal of the ACM (JACM)*, 57(4):25, 2010.

[17] D. Easley and J. Kleinberg. *Networks, Crowds, and Markets: Reasoning About A Highly Connected World*. Cambridge University Press, 2010.

[18] Z. Fu, M. Personick, and B. Thompson. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In *Proceedings of Workshop on GRAph Data management Experiences and Systems (GRADES)*, 2014.

[19] M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou. Practical Recommendations on Crawling Online Social Networks. *Journal on Selected Areas in Communications*, 29(9):1872–1892, 2011.

[20] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[21] P. Harish and P. Narayanan. Accelerating Large Graph Algorithms on the GPU using CUDA. In *High Performance Computing (HiPC)*. Springer, 2007.

[22] M. Harris, S. Sengupta, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. *GPU gems*, 2007.

[23] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA Graph Algorithms at Maximum Warp. In *Proceedings of SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[24] S. Hong, T. Oguntebi, and K. Olukotun. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In *Proceeding of the Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[25] B. A. Huberman and L. A. Adamic. Internet: Growth Dynamics of the World-Wide Web. *Nature*, 401(6749):131–131, 1999.

[26] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai, and A.-L. Barabási. The Large-scale Organization of Metabolic Networks. *Nature*, 407(6804):651–654, 2000.

[27] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, A Social Network or A News Media? In *Proceedings of International Conference on World Wide Web (WWW)*, 2010.

[28] D. Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processors. *Intel Performance Analysis Guide*, 2009.

[29] D. Li and M. Becchi. Deploying Graph Algorithms on GPUs: An Adaptive Solution. In *International Symposium on Parallel & Distributed Processing (IPDPS)*, 2013.

[30] L. Luo, M. Wong, and W.-m. Hwu. An Effective GPU Implementation of Breadth-First Search. In *Proceedings of Design Automation Conference (DAC)*, 2010.

[31] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarria-Miranda. A Faster Parallel Algorithm and Efficient Multithreaded Implementations for Evaluating Betweenness Centrality on Massive Datasets. In *International Symposium on Parallel & Distributed Processing (IPDPS)*, 2009.

[32] A. McLaughlin and D. A. Bader. Scalable and High Performance Betweenness Centrality on the GPU. In *Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[33] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU Graph Traversal. In *Proceedings of SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012.

[34] D. Merrill and A. Grimshaw. Parallel Scan for Stream Architectures. Technical report, University of Virginia, 2009.

[35] R. Nasre, M. Burtscher, and K. Pingali. Atomic-Free Irregular Computations on GPUs. In *Proceedings of Workshop on General Purpose Processor Using GPUs (GPGPU)*, 2013.

[36] R. Nasre, M. Burtscher, and K. Pingali. Data-Driven Versus Topology-Driven Irregular Computations on GPUs. In *International Symposium on Parallel & Distributed Processing (IPDPS)*, 2013.

[37] P. W. Olsen, A. G. Labouseur, and J.-H. Hwang. Efficient Top-k Closeness Centrality Search. In *International Conference on Data Engineering (ICDE)*, 2014.

[38] R. Pearce, M. Gokhale, and N. M. Amato. Scaling Techniques for Massive Scale-Free Graphs in Distributed (External) Memory. In *International Symposium on Parallel & Distributed Processing (IPDPS)*, 2013.

[39] Z. Qi, Y. Xiao, B. Shao, and H. Wang. Toward a Distance Oracle for Billion-Node Graphs. *Proceedings of the VLDB Endowment*, 7(1):61–72, 2013.

[40] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. Regularizing Graph Centrality Computations. *Journal of Parallel and Distributed Computing*, 2014.

[41] S. Tzeng, A. Patney, and J. D. Owens. Task Management for Irregular-Parallel Workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics*, 2010.

[42] V. Ufimtsev and S. Bhowmick. Application of Group Testing in Identifying High Betweenness Centrality Vertices in Complex Networks. In *Workshop on Machine Learning with Graphs, KDD*, 2013.

[43] P.-H. Wang, Y.-M. Chen, C.-L. Yang, and Y.-J. Cheng. A Predictive Shutdown Technique for GPU Shader Processors. *Computer Architecture Letters*, 2009.

[44] 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 *Proceedings of SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2015.

[45] C.-L. Yang, H.-W. Tseng, C.-C. Ho, and J.-L. Wu. Software-Controlled Cache Architecture for Energy Efficiency. *Transactions on Circuits and Systems for Video Technology*, 15(5):634–644, 2005.