A Decentralized Frontier Queue for Improving Scalability of Breadth-First-Search on GPUs

Chou-Ying Hsieh[1], Po-Hsiu Cheng[2], Chia-Ming Chang[1] and Sy-Yen Kuo[1]

[1]Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan
[2]Graduate Institute of Electronic Engineering, National Taiwan University, Taipei, Taiwan
{f07921043, r10943151, r10921101, sykuo}@ntu.edu.tw

***Abstract*—Breath-first-search (BFS) algorithm is the fundamental building block of broad applications from the electronic design automation (EDA) field to social network analysis. With the targeting data set size growing considerable, researchers have turned to developing parallel BFS (PBFS) algorithms and accelerating them with graph processing units (GPUs). The frontier queue, the core idea among state-of-the-art designs of PBFS, opens the door to neighbor visiting parallelism. However, the traditional centralized frontier queue in PBFS suffers from a dramatic collision and explosive growth of memory space when excessive threads simultaneously operate on it. Therefore, we identify the challenges of current frontier queue implementations. To solve these challenges, we proposed the decentralized frontier queue (DFQ), which separates a centralized queue into multiple tiny sub-queues for scattering the atomic operation collision on these queues. We also developed the novel overflow-free enqueue and asynchronous sub-queue drain methods to avoid dramatic growing size of the frontier queue and the overflow issue on the naive sub-queue design. In our experiments, we showed that our design could have better scalability and grain averagely 1.04x speedup on the execution in the selected benchmark suit with considerable memory space efficiency.**

***Index Terms*—breadth-first-search, parallel computing, GPU, scalability, queue**

I. Introduction

The graph can model massive physical phenomena. From modeling integrated circuit [1], human brain [2] to social network [3], the graph can represent almost anything around us. How efficient a graph traversal is can determine how fast we can analyze and get information from the graph. Breadth-first search (BFS), as the fundamental algorithm for exploring nodes in a graph plays an essential role in mining different graph properties, for example the graph pattern matching (GPM), the strongly connected component (SCC) detection, and the betweenness centrality (BC) analysis. In addition, Graph 500, which is the prestigious international supercomputer ranking organization, uses the BFS execution time to rank a supercomputer. The action emphasizes the importance of the BFS algorithm again.

With the development of graphic processing unit (GPU), the specialized SIMT(Single-Instruction-Multiple-Data) machine, more and more algorithms, including convolution neural networks and wireless communication, have adopted on the GPU to utilize its massively parallel computing. The GPU provides remarkable computing power and memory capabilities, which brings significant performance enhancement on these applications. The BFS algorithm has also evolved to different parallel BFS (PBFS) algorithms on GPUs [4]–[9].

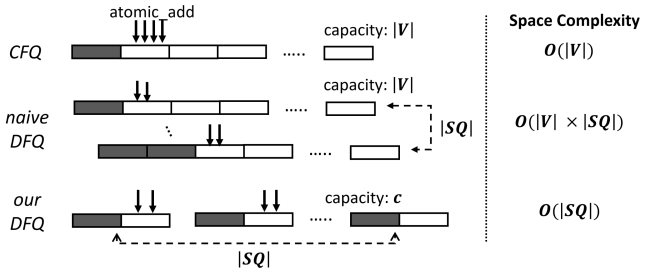

Fig. 1: The memory consumption across different frontier queue implementations.

The traditional BFS starts from a source node. At first *visits* the source's neighbors connected to the source node are called *frontiers*. These frontiers then become the next level's source nodes. Once there are no unvisited nodes, the algorithm finishes the graph traversal. Generally, the GPU can parallelly find frontiers, called *frontier generation*. Suppose we can collect these frontiers in a shared data structure, usually a frontier queue. In that case, the GPU can further assign multiple threads to visit a frontier's neighbor nodes, called *neighbor visiting*. The frontier queue acts as the bridge between frontier generation and neighbor visiting parallelism. However, the centralized frontier queue (CFQ) is a double-edged sword. As the CUDA core count increases rapidly in the GPU, the contention of enqueue and dequeue on the CFQ will worsen in the future, significantly decreasing the scalability of PBFS.

To address the future concern on the contention of CFQ, we proposed a decentralized frontier queue (DFQ). (*https://github.com/NTUDDSNLab/DFQ-BFS*) The main idea is to distribute the massive atomic operations to different sub-queues, shown in Figure 1. In addition, we observe that the naive DFQ consumes excessive memory space. Hence, we proposed two novel methods *overflow-free enqueue* and *asynchronous queue* drain to maintain our queue size at the constant level. The overflow-free enqueue method avoids the sub-queue from overflow, while the asynchronous queue drain allows asynchronous frontier visiting between warps (the basic compute unit in a GPU) when the potential overflow detects. We can decouple the frontier queue size from the graph vertex

978-3-9819263-7-8/DATE23/© 2023 EDAA

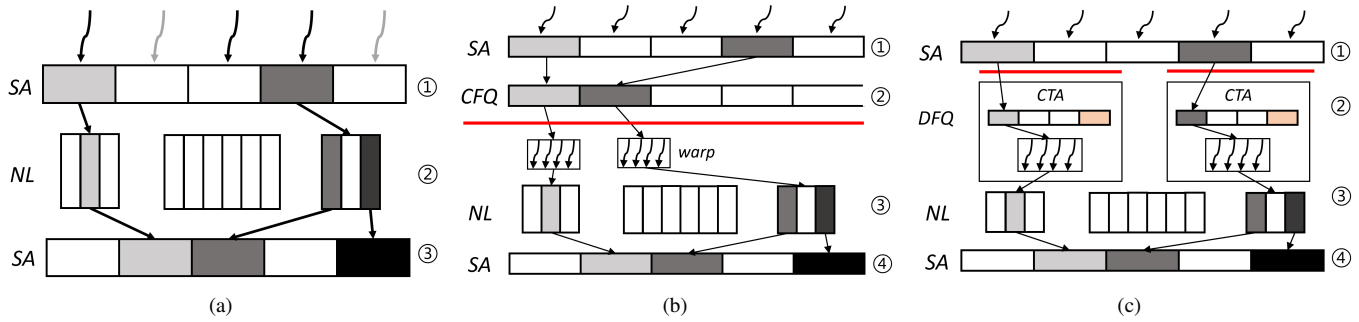

Fig. 2: The execution flow of different PBFS on GPUs (a) without the the frontier queue support. (b) with the traditional centralized frontier queue support. (c) with our decentralized frontier queue support. The SA is the status array; NL means the neighbor list; FQ is the frontier queue; DFQ is the decentralized frontier queue. The red line represents the synchronized barrier and its length identifies different computing level synchronization.

number with these two methods. We significantly reduce the memory usage to $c \times |SQ|$, whose complexity is $O(|SQ|)$. The $|SQ|$ is the number of sub-queue.

We summarize our contributions as follows:

- DFQ is the first frontier queue design for PBFS on GPUs to alleviate contention with the minimum and constant memory consumption.
- DFQ has better scalability than the traditional centralized frontier queue and *hierarchical queue*, the state-of-the-art PBFS sub-queue implementation on GPUs.
- DFQ can utilize the fast accessing latency of shared memory because of the constant consumption of sub-queue memory.

We organize the rest of this paper as follows: Section II introduces the background and related works. In Section III, we highlight two main challenges of CFQ. We will discuss our DFQ implementation in Section IV. Section V presents the evaluation of our design compared to the state-of-the-arts. Section VI concludes.

II. Background

A. Architecture and Execution Model on GPUs

Generally, a GPU consists of hierarchical computing units and memory. A GPU often accommodates several *streaming processors* (SM). All SMs can access the same memory called the *global memory*. A SM comprises multiple *cooperative thread arrays* (CTAs). An additional scratchpad memory called *shared memory* can serve as the data cache for each CTA. According to previous studies, the accessing speed of shared memory is roughly 100 times faster than the global memory [10]. Within each CTA, there are multiple subgroups of threads called *warps*. A warp is the primary computing unit where the threads share the same program counter (PC); threads in a warp will execute the same instruction. Because of the single instruction, multiple threads (SIMT) programming model of GPUs, every thread in a GPU has the same code path. Once a warp has early finished its work or reached the synchronization point, it will idle until the longest executing warp completes.

B. Frontier Queue in Parallel Breadth-First-Search

Harish et.al [4] proposed CUDA_BFS, which is the first research to adapt BFS with Nvidia GPU. It used three arrays: frontier, visited, and cost array, to store the information of each vertex and each level's frontiers. CUDA_BFS parallelly generates frontiers by scanning the status array (Figure 2(a) ①), Moreover, it parallelly visits the scanned frontiers' neighbor list for updating the status array.(Figure 2(a) ②③) Hierarchical queue [1] pointed out that a frontier queue can dramatically reduce the computational complexity of finding frontiers by only checking the last-level frontiers' neighbors, which is called frontier propagation or scan-free method [8]. The frontier queue gathers the next level frontier visited by all the threads once, which can know exactly how many frontiers are in this level and compress them to a continuously allocated memory range. (Figure 2(b) ②) By doing this frontier gathering, the frontier queue provides another parallelizing opportunity to visit frontiers. Hence, the succeeding research [7], [8] working on parallel BFS has leveraged the frontier queue for the frontier generation until now.

Figure 2 (a) and (b) illustrate the difference with and without the frontier support in a GPU. Without the frontier queue, after a thread scans the status array and finds the next level's frontier vertex, this thread has to visit all the neighbors of the vertex by itself iteratively. With the frontier queue support, on the other hand, because the frontier queue collects all frontiers in a continuous space previously, all threads can be synchronized and rearranged, which enables GPU to assign multiple threads for visiting a frontier vertex. Based on this observation, Hong et al.al [6] proposed a virtual warp-centric approach to assign a warp to parallelly visit a frontier vertex, which gained up to 9x speedup compared to thread-centric BFS implementation on GPU.

C. Related Works

Hierarchical queue [1] is the most related work to our DFQ. It first pointed out that the contention on the frontier queue caused severe performance degradation. It allowed a warp first to append frontiers to the lower level queue and gathered all

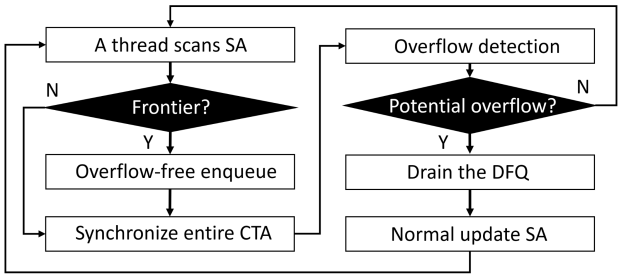

Fig. 3: The execution flow chart of our DFQ used in PBFS.

frontiers by batch uploading from low-level queues to high-level ones to avoid load imbalance among each sub-queue. Our DFQ can avoid the copy data procedure but maintain the workload balance. We will compare the hierarchical queue to our DFQ design in Section V.

There is still other research aiming to improve the utilization of different aspects of the PBFS on the GPU. Beamer et.al [11] found that their bottom-up BFS algorithm can remarkably reduce the waste of edge visits and proposed directed-optimizing BFS to switch from top-down to bottom-up BFS dynamically. Gaihre et.al [8] extended this concept to the GPU and made adaptive scanning optimizations on the status array. In addition, there were several studies [7]–[9] assigning different thread counts by the frontier's fan-out degree for improving the utilization in neighbor visiting. The above research is orthogonal to our decentralized frontier queue design.

Merrill et.al [5] claimed that the atomic operation which serializes the instruction is expensive for GPUs. They proposed an efficient prefix-sum to calculate the scatter offsets for the de/enqueue. However, as the core count in a GPU becomes considerable, the synchronization overhead of prefix-sum grows dramatically; hence the modern PBFS designs used the atomic operation for the enqueue method. [7]–[9]

III. Challenges of Frontier Queue Design in BFS on GPUs

A. Intensive Accessing Contention on Centralized Frontier Queue

The centralized frontier queue enjoys several advantages, including ease of maintenance, enough allocating space, and workload balance. However, those advantages are declining. Since we can easily foresee that the core number in a GPU will keep increasing through observing history, the dramatic growth of collision on the centralized frontier queue will become the scalable bottleneck in PBFS algorithms [1]. We can observe that the number of the core has grown extremely fast of GPUs. Notably, the newest RTX 4090 Ti might equip up to 18,434 CUDA cores, which boosts a 71% higher core count than the last version RTX 3090 Ti. The expanding hardware parallelism does not directly transform into the speedup since not all the parts in PBFS can be fully parallelized, which degrades the scalability. Significantly, over 10,000 threads could push frontiers to the same address simultaneously. Although some previous research has used the atomic fetch-and-add operation instead of the compare-and-swap to avoid the retry time of serialization. [12] The most advanced atomic fetch-and-add operation needs to be serialized, which takes about 10,000 cycles to increment the same address or value. [13]

B. Inflexible Memory Management of Sub-queues

One concept to reduce the collision is to use a set of sub-queues instead of a single queue that serializes all operations. Different threads can parallelly append their frontiers on each sub-queue. The naive approach shown in Figure 1 requires excessive $|SQ|$ times memory space, where $|SQ|$ denotes the number of sub-queue. Hierarchical queue [1] utilizes this concept to implement multi-level queues in both global and shared memory. It allows the thread first to append frontiers to its lower queues, and then aggregates all the frontiers to the global queue. The hierarchical queue can reduce the space complexity from $O(|V| \times |SQ|)$ to $O(|V| \times |H|)$, where $|V|$ is the graph vertex number and $|H|$ means the hierarchy level. However, the small capacity of the shared memory limits what graph the hierarchical queue can run, thus it can only run the graph with the small outgoing degree to avoid the lower queue overflow.

If a warp wants to append two frontiers to the queue whose size and capacity are 2 and 3, respectively. There is one frontier that has no space to place. Since there are no dynamic memory allocation or exception handling mechanisms in GPUs, this overflow causes the segment fault in GPUs. Furthermore, the frontier propagation method in the hierarchical queue requires the lower queues to aggregate their frontier to higher queues to ensure all the threads will not visit repeated frontiers. This aggregation takes additional synchronization and memory usage, which generates considerable overhead.

IV. Decentralized Frontier Queue

A. Overview

Figure 2(c) shows architecture of DFQ in the GPU, and Figure 3 is the execution flow chart of our DFQ design. Figure 4 is the pseudo code snippet of its kernel function. Fundamentally, DFQ separated the traditional centralized frontier queue into multiple decentralized sub-queues with a buffer (the red parts in Figure 2(c)) for each to avoid overflow. We gave each CTA a sub-queue to accommodate its frontiers. We propose two novel sub-queue operations, *overflow-free enqueue* and *asynchronous sub-queue drain*, to ensure no overflow happens when threads want to append frontiers to the DFQ. Besides, with these two optimizations, we can maintain the DFQ in slight synchronized overhead and extremely low memory space consumption.

B. Overflow-free Enqueue

We used the following equation to detect the overflow before assigning a warp to scan the status array and append frontiers to the corresponding sub-queue.

$$atomic\_add(size, stride) <= capacity \quad (1)$$

The *size* and *capacity* stand for the current size and maximum capacity of the sub-queue, respectively. The *stride* represents

```
1  __global__ void CUDA_BFS_KERNEL():
2      /* Scanning */
3      foreach v in graph:
4          /* Overflow checking */
5          if(size >= capacity - stride):
6              foreach ftr in DFQ:
7                  VISIT(ftr, 32) /* Drain */
8              __syncthreads()
9              if(threadIdx.x == 0):
10                 size = 0
11         __syncthreads()
12         if (v.level == cur_level):
13             /* Enqueue */
14             atomic_add(size, 1);
15             DFQ[size] = v;
16         __syncthreads()
17     foreach ftr in DFQ:
18         VISIT(ftr, 32); /* Drain */
19     grid.sync();
```

Fig. 4: The code snippet of PBFS using our decentralized frontier queue.

the number of threads in a warp. In Figure 4, before a warp starts to scan the status array (line:3), we first assign a delegated thread within the warp to atomically reserve the space of *stride* (line:6). If the current size exceeds the capacity, the sub-queue has a chance to overflow once we start the scanning this time. In this case, we will first stop the status array scanning of this warp. Then, we synchronize all the warps which attempt to append to this sub-queue and *drain* out the queue. We use a warp whose size is 32 to parallelly visit a frontier node *ftr*'s neighbors (line: 8), and assign delegated thread to reset *size* (line: 10-11). We will discuss the details of the drain in the next section. Otherwise, the warp can scan the status array successfully and append all its frontiers to the sub-queue without overflow (line: 14-18). After scanning this level, we will visit the rest frontiers in the DFQ by warps (line: 20). Because we always prepare for the worst case that every thread in the warp will have a frontier to append to the sub-queue, it is intrusive to derive that our queue will never overflow. Furthermore, the overflow-free enqueue method enables the constant size of sub-queues to work correctly, hence we can place sub-queues to the size-limited shared memory and enjoy its fast access latency.

C. Asynchronous Sub-queue Drain

The overflow-free enqueue method requires additional synchronization when the potential overflow is detected. Because the update of the status array is independent of the BFS level, namely, a warp can update the status array at this level even though other warps are still scanning the status array, asynchronously popping frontiers from a sub-queue to visit is possible. In Figure 4, We allow each sub-queue to drain when its size is almost approaching its capacity. During the draining process, we synchronize all the warps that append frontiers to this sub-queue. We use the *cooperative groups* libary [14], CUDA's latest and flexible synchronization libary, to group multiple warps to a same *tile* and use __syncthreads() to synchronize between warps. (Actually, we previously used tile_partition() to partition a CTA to different tiles(warps) instead of grouping them.) After that, we use all threads in the tile for neighbor visiting until the sub-queue is empty. Different tiles can drain asynchronously; thus, the frontier number imbalance can be alleviated.

TABLE I: Benchmark data set and the description

Data set	Type	Description
Amazon0302	Product network	Amazon product co-purchasing network from March 2 2003
appu	Random generated	Random sparse matrix used in app benchmark, NASA AMES Research Cente
com-lj	Ground-truth communities	LiveJournal online social network
com-orkut		Orkut online social network
graph500	Synthesis	The synthesis graph used in the benchmark of evaluating supercomputers
soc-castor	Social Network	This Network contains friendships between users of the Catster web site
web-Berkstan	Web graph	Web graph of Berkeley and Stanford
web-Google		Web graph from Google

V. Evaluation

This section will first introduce the experiment setup. Then, we will evaluate the performance between our decentralized frontier queue and the state-of-the-art design. We first discuss the overall speedup of execution time, then we point out that the improvement of our design comes from better scalability. Last, we calculate the memory space complexity of the state-of-the-art design and our DFQ.

A. Experiment Setup

We selected the following graph from SNAP [15] shown in Table I for benchmark, since they represent different kinds of the graph in various domains. We used CSR format to represent all these data sets, and all the node numbers were first shuffled to simulate the real-world condition. Before recording the performance, we prefetched the neighbor list and status array to the global memory. We ran the experiments on the AMD 3950x 16-core processor @ 2.2GHz with 128GB DDR4 and an Nvidia RTX 3090 GPU, which has 10496 Cuda cores (maximum 82 CTAs), 24GB global memory, and 128KB L1 cache/shared memory for each stream processor. We ran 100 times to get the average execution time in each experiment. We faithfully implemented three types of frontier queues and one state-of-the-art together with the virtual warp-centric neighbor visiting [6] method to analyze the frontier generation performance:

- **BASE**: The single centralized frontier queue locates in the global memory.
- **HQ**: The state-of-the-art hierarchical queue [1]. We faithfully implement the scan-free and virtual warp-centric version on it.
- **DFQ-G-2048**: This is the decentralized frontier queue located in the global memory. We created 82 sub-queues with 2048 frontier capacity and assigned each CTA per sub-queue.

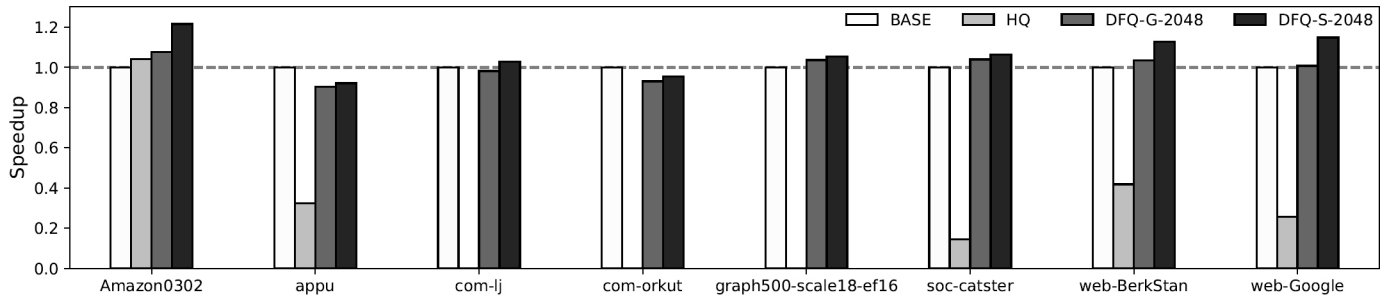

Fig. 5: The execution time of four frontier queue implementations on eight data set. HQ fails to execute *com-lj*, *com-orkut*, and *graph500*, since the maximum fan-out degree of these data set exceeds the capacity of the shared memory in the RTX 3090 GPU, which causes overflow on the low level sub-queue.

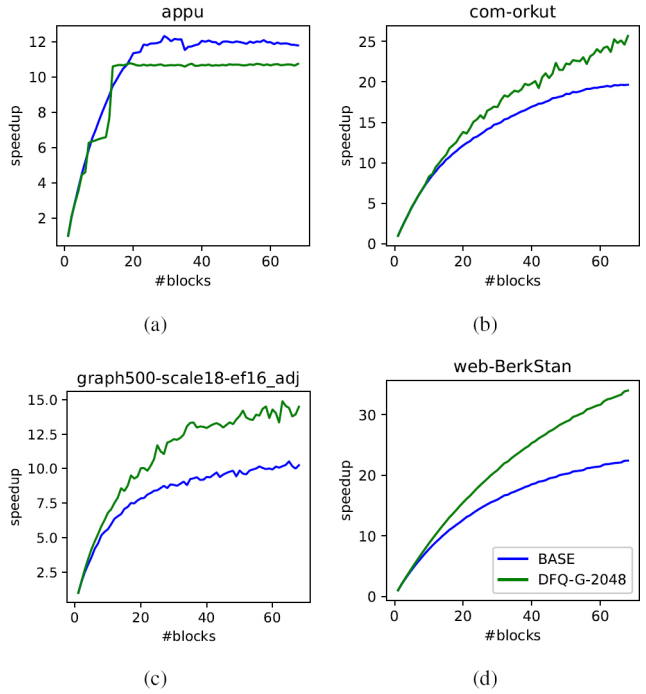

Fig. 6: Execution time speedup across different CTA number

- **DFQ-S-2048**: The configuration is the same as DFQ-G-2048, but we placed the 82 sub-queues to each CTA's shared memory.

B. Scalability

To prove that the decentralized queue can indeed reduce the contention on the frontier queue and then improve the scalability, we compared BASE and DFQ-G-2048 shown in Figure 6. We measured the scalability by comparing the speedup of different frontier queue implementations with increasing thread (CTA) numbers. The speedup is computed relative to each work with only one CTA (each CTA comprises 1024 threads). We selected four different types of the graph to discuss. Generally, the DFQ-G-2048 has better scalability than BASE except for the *appu* data set. This graph is a small data set with few frontiers on each level; hence the number of frontiers in each sub-queue might be pretty different. This imbalanced condition hurts the scalability of DFQ. On the other hand, the reduction of contention on the frontier queue can reflect the scalability enhancement on the graph, which has more vertexes and larger fan-out degree nodes, such as web networks or social networks like *web-BerkStan* and *com-orkut*, respectively.

C. Speedup

Figure 5 shows the speedup compared to the above four implementations. We utilized the full RTX 3090 computing power for every case. Surprisingly, HQ performed the worst among all implementations. It gains from -2.94x to 1.06x execution time speedup relative to the BASE. HQ encounter overflow error in *com-lj*, *com-orkut*, and *graph500* because the ready-to-append frontier number exceed the low-level sub-queue during the execution. The terrible performance of HQ might come from the atomic compare-and-swap (CAS). To avoid repeatedly visiting the same frontier by different warps, HQ needs to use atomic CAS on status array updating, which is proven to be slower than atomic fetch-and-add operation in modern GPUs [12]. Moreover, we observed that the frontier propagation method only benefits those graphs with few fan-out degree nodes. Using single-scan on status array updating is more solid than the frontier propagation. DFQ-G-2048 and DFQ-S-2048 have equal or even better execution times than BASE across most of the data sets, except for *appu* and *com-orkut*. *Com-orkut* has numerous communities; while *appu* is a tiny data set whose frontier number in each BFS level is far less than the frontier queue size. Since we used single scan on status array and enqueue frontiers to each sub-queue based on the thread's block ID, both of these properties exacerbate the imbalance of frontier number among all sub-queus This frontier imbalance directly leads to workload imbalances and hides the improvement of contention reduction. Generally, DFQ-S-2048 can gain a slight 1.04x speedup over DFQ-G-2048 because of the fast accessing time of the shared memory. We argue that despite the speedup of DFQ designs is not obvious, the enhancement of scalability will finally transform into speedup when the CUDA core in a GPU keeps increasing.

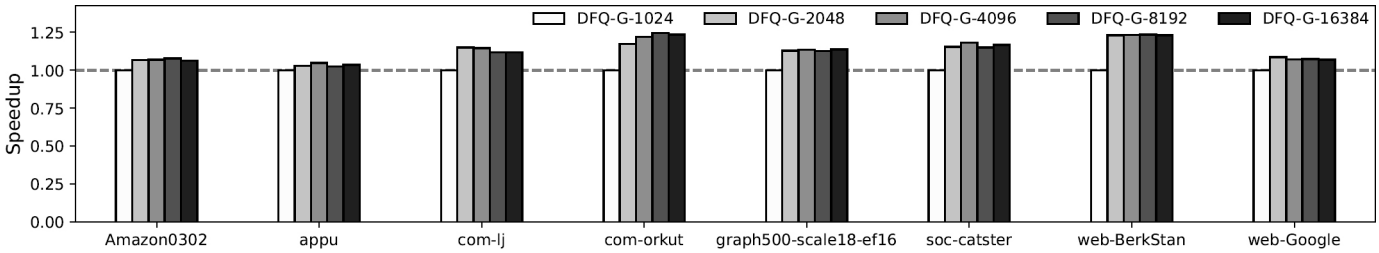

Fig. 7: The execution time speedup of different sub-queue capacities across eight data set.

TABLE II: The memory usage of the frontier queue across different scales of the graph500 data set. The unit is megabytes (MB), and the number in the bracket shows the usage growth rate compared to scale-18 work.

	scale-18	scale-20	scale-22	scale-24	scale-26
BASE	0.69	2.58 (207%)	9.58 (1288%)	18.44 (2572%)	131.21 (18915%)
DFQ-G-2048	0.67	0.67 (0%)	0.67 (0%)	0.67 (0%)	0.67 (0%)

D. Space Complexity

The overflow-free enqueue method (Section IV) significantly reduce the minimum requirement of the DFQ's capacity to a constant level. We evaluated the memory usage of static frontier allocation across the above data set shown in Table II. We can observe that the centralized frontier queue requires $O(|V|)$ space, which increases usage through the growth of graph size. ($|V|$ denotes the number of graph vertex) On the contrary, the space complexity of our DFQ is $O(|SQ|)$, where the $|SQ|$ denotes the number of sub-queue. The usage of our DFQ depends on how much the number of CTA a GPU has.

E. Sub-queue Capacity

In this section, we evaluated the impact of different sub-queue capacities on the execution time speedup in Figure 7. The DFQ-G-1024 is the baseline, which remains the least capacity of the frontier sub-queue. A CTA contains 1024 threads, so if any thread adds a frontier to the sub-queue, this queue must drain before adding another frontier. We used the whole RTX 3080 and ran all the cases. We found that increasing capacity has limited improvement in execution time (averagely 1.09x speedup). The DFQ-G-2048 and 4096 are two competitive configurations in this experiment.

VI. Conclusion

In this paper, we argued that the increasing contention on the centralized frontier queue would become the scalability bottleneck of the parallel breadth-first-search algorithm on GPUs. Therefore, we proposed a novel decentralized frontier queue (DFQ), which alleviates the contention on the single queue by distributing atomic operations to multiple sub-queues. Besides, we designed two optimizations that solve the excessive memory space problem and performance issue in naive DFQ. We demonstrated that our DFQ had better scalability compared to state-of-the-art designs. Besides, it also gained an average 1.04x execution speedup and significant memory efficiency in our experiments.

Acknowledgment

We thank to National Center for High-performance Computing (NCHC) for providing computational and storage resources. This research was supported by the National Science and Technology Council under grant NSTC 111-2221-E-002 -133 -MY3

References

[1] L. Luo, M. Wong, and W.-m. Hwu, "An effective gpu implementation of breadth-first search," in *Design Automation Conference*. IEEE, 2010, pp. 52–55.

[2] D. S. Bassett and O. Sporns, "Network neuroscience," *Nature neuroscience*, vol. 20, no. 3, pp. 353–364, 2017.

[3] S. A. Myers, A. Sharma, P. Gupta, and J. Lin, "Information network or social network? the structure of the twitter follow graph," in *Proceedings of the 23rd International Conference on World Wide Web*, 2014, pp. 493–498.

[4] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *International conference on high-performance computing*. Springer, 2007, pp. 197–208.

[5] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," *ACM Sigplan Notices*, vol. 47, no. 8, pp. 117–128, 2012.

[6] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," *Acm Sigplan Notices*, vol. 46, no. 8, pp. 267–276, 2011.

[7] H. Liu and H. H. Huang, "Enterprise: breadth-first graph traversal on gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[8] A. Gaihre, Z. Wu, F. Yao, and H. Liu, "Xbfs: exploring runtime optimizations for breadth-first search on gpus," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 121–131.

[9] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel *et al.*, "Gunrock: Gpu graph analytics," *ACM Transactions on Parallel Computing (TOPC)*, vol. 4, no. 1, pp. 1–49, 2017.

[10] X. Mei and X. Chu, "Dissecting gpu memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2016.

[11] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–10.

[12] D. Troendle, T. Ta, and B. Jang, "A specialized concurrent queue for scheduling irregular workloads on gpus," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–11.

[13] L. Nyland and S. Jones, "Understanding and using atomic memory operations," in *4th GPU Technology Conf.(GTC'13), March*, 2013, pp. 1–61.

[14] M. Harris and K. Perelygin, "Cooperative groups: Flexible cuda thread programming," *NVIDIA Developer Blog*, 2017.

[15] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.