Toward Concurrent Lock-Free Queues on GPUs

Xiangyu ZHANG, Yangdong DENG, Nonmembers, and Shuai MU, Member

SUMMARY General purpose computing on GPU (GPGPU) has become a popular computing model for high-performance, data-intensive applications. Accordingly, there is a strong need to develop highly efficient data structures to ease the development of GPGPU applications. In this work, we proposed an efficient concurrent queue data structure for GPU computing. The GPU based provably correct, lock-free FIFO queue allows a massive number of concurrent producers and consumers. Warp-centric en-queue and de-queue procedures are introduced to better match the underlying Single-Instruction, Multiple-Thread execution model of modern GPUs. It outperforms the best previous GPU queues by up to 40 fold. The correctness of the proposed queue operations is formally validated by linearizability criteria.

key words: FIFO queue, concurrent, lock-free, GPU

1. Introduction

Recently, general purpose computing on GPU (GPGPU) has become a popular computing model for high-performance, data-intensive applications [1]. The massively parallel programming model of modern GPUs, however, pose unique challenges to efficiently programing GPUs [2]. Traditionally, such a problem has been addressed by libraries that provide highly efficient low-level data structures and relevant routines. Many GPU based libraries supporting various programming primitives have been proposed (e.g., [3], [4]). On the other hand, the FIFO queue, which is one of the most fundamental data structures and has wide applications, has only attracted limited research efforts (e.g., [5], [6]). In this paper, we propose an efficient concurrent lock-free queue for GPGPU.

A GPU based queue data structure must support massively parallel operations. Concurrent data structures can be either lock-based or lock-free. Since current GPUs do not provide an explicit lock mechanism, we study lock-free data structures that depend on atomic primitives to ensure concurrency. When multiple threads try to access a queue, the lock-free property ensures that at least one of the threads makes progress and also offers immunity to deadlocks and priority inversion. To build an efficient lock-free queue, it is essential to solve two fundament problems, avoidance of contention among threads and maintenance of correctness during parallel operations.

A large body of research has been dedicated to concurrent queues on multi-core and/or multithreaded CPU platforms. Michael and Scott [7] introduced a linked list based queue with atomic Compare and Swap (CAS) loops. Many researchers then proposed improvements for better scalability to a larger number of parallel threads. Edya and Nir [8] used a double-linked list as the basis of the concurrent queue. Hoffman et al. [9] developed a basket linked list based approach. Gidenstam et al. [10] presented a linked-list of arrays as the container. Morrison and Afek [11] designed a new queue structure with fetch-and-add instead of CAS. Researchers also proposed another line of approach, request-combining methods [12]–[14], to avoid contention among threads. The idea is to have multiple threads sending operation requests to a single shared structure and a single thread scanning the pending requests and applying them to the underlying data structure. We also adopt such a combining strategy and enhance it by exploiting the SIMD execution of GPU threads to parallelize the combining process.

The research on GPU based concurrent queues is still rather limited. Misra and Chaudhuri [6] demonstrated the usage of GPUs CAS operator to implement various concurrent data structures. Cederman et al. [5] implemented the concurrent lock-free queue proposed by Michael and Scott on GPUs. Both of these works only enable a lower level of performance than what can be achieved on multi-core platforms. A salient problem is that they treat the thread as a single unit for concurrent operations while modern GPUs always schedule jobs on a group of threads (i.e., a warp in NVIDIA’s terminology). The single thread based operations tend to incur more contentions in CAS operations. As reported by Cederman et al. [5], the GPU based queue operations are slower than their multi-core equivalents.

In this work, we developed an efficient concurrent lock-free queue, designated as combined and CAS loop queue (CCLQ), for GPUs. By well matching the queue operations to the Single-Instruction, Multiple-Thread execution model of modern GPUs, our implementation outperforms the best previous result on GPGPUs up to 40 fold. We present the detailed implementation in Sect. 2. In Sect. 3, we analyze the correctness of CCLQ using linearizability criteria [15]. The evaluation results are presented in Sect. 4.

2. The Algorithm

In this section, we present CCLQ data structure and its de-
tailed GPU implementation. First, the overall data structure is elaborated. Then we explain how to implement enqueue and dequeue operations with request combining and CAS loops. The pseudo code is listed in Fig. 1.

The underlying data structure of the queue is implemented as a linked list of nodes. The corresponding CUDA code is listed in lines 1–4 in Fig. 1 (a). All nodes in the linked list are allocated from a pre-allocated memory pool in GPUs global memory. To be more efficient for GPU execution, we use three separate arrays (lines 6 and 7) to store the three fields of a node in the linked list. In other words, the queue structure in line 5 only stores the indexing information of the nodes linked to the queue. The indexes are actually pointing to the three separate arrays. On modern GPUs, a given number of threads are organized into a SIMD group (i.e., warp in NVIDIA terminology and a warp has 32 threads). Each warp has an array, data list, which serves as a local shared store for threads in this warp. 32 data items for queuing operations are first combined as a single unit in this local store with start_pos and end_pos as pointers to the head and tail. Global_head and Global_tail (line 8) are the global head and tail of the queue. The range of data is between Global_head+1 and Global_tail. ScanWarp (line 9–17) is a prefix-sum function called by enqueue and dequeue operations.

Figure 1 (b) lists the pseudo code of the enqueue operation of CCLQ. The operation is organized as two steps as follows. The parameter “request” indicates that whether a thread needs to enqueue indeed. Threads with “request[threadIdx.x] = 0” just assist to execute prefix-sum.

- Step one: All threads in a warp submit its request to the shared local store. A prefix-sum operation is first performed by all threads in the warp (line 22) to determine the insertion position of each item in the local store. Then threads will actually insert the items into the local store according to the position computed by the prefix-sum operation.

- Step two: One thread (the last one in the pseudo code) gets the actual number of requests in the warp. It executes CAS loops to dequeue the corresponding number of data items.

Figure 1 (c) lists the pseudo code of dequeue procedure of CCLQ. It consists of three steps.

- Step one: All threads in a warp submit their requests to the shared local store. Threads in a warp will all execute prefix-sum (line 39) to compute the insertion positions with a prefix-sum operation.

- Step two: The last thread of the warp gets the actual number of requests in the warp. It executes CAS loops to dequeue the corresponding number of data items.

![Fig. 1](image-url) Pseudo code of CCLQ data structure of manipulation routines.
from the queue and stores in the shared data structure.
- Step three: Each thread of the warp gets its corresponding dequeued data from the shared data structure.

3. Correctness of Our Approach

A data structure is correct under concurrent manipulations if the manipulation operations are linearizable. An operation to a shared data object is linearizable if it appears to happen instantaneously to producers and consumers of data. Linearizability is a correctness condition for concurrent manipulation to a shared data structure. It permits programmers to compose and reason about concurrent objects using techniques from the sequential domain. In this section we prove the linearizability of our approach.

Note that all CAS operations are atomic. A linearizability point is the time point that an operation succeeds or takes effect. In other words, it is the point that others threads will see the operation. Accordingly, we have the following observations.

**Observation 1.** The linearizability point of the enqueue operation is the successful CAS operation at line 31 of the enqueue kernel.

**Observation 2.** The linearizability points of the dequeue operation are the CAS operations at lines 55, 60 and 46 of the dequeue kernel.

**Lemma 1.** The enqueue operation is linearizable with respect to other enqueue and dequeue operations according to Observations 1 and 2.

**Proof.** Consider two enqueue operations A and B. Suppose they both get the same nextpos. According to Observation 1, if they execute line 31 at the same time, one will fail because of the nature of atomic CAS operation and only one will succeed. If A executes line 31 first, then B will fail and continue to loop. The same happens if B executes line 31 first. It is obvious that they successfully execute line 31 in a sequential manner. So the enqueue operation obeys the linearizability criteria.

**Lemma 2.** The dequeue operation is linearizable with respect to other enqueue and dequeue operations according to Observations 1 and 2.

**Proof.** Consider two dequeue operations C and D. If they both get the same old_start and execute line 52 of pseudo code listed in Fig. 1, only one will succeed and the other one will continue to loop. It is the same if they both execute line 55 or line 60. If C executes line 55 and D executes line 60, as they compare and swap the same variable, still only one will succeed. It is clear that the two operations can only succeed in a sequential fashion. So the dequeue operation obeys the linearizability criteria.

According Lemma 1 and 2, we know that Enqueue and Dequeue operations are both linearizability and correct during execution.

4. Evaluation and Analysis

We evaluate the performance of our GPU based queue on an NVIDIA GeForce GTX 780Ti (Kepler architecture) GPU. We re-implemented the Michael and Scotts queue (MSQ) on GPU by following the method proposed by Cederman et al. Since our implementation achieves a better throughput, we use it as the comparison reference of GPU. We also use linked list concurrent ring queue (LCRQ) [11] and cache-aware lock-free queue (CAQ) [10] as representatives of CPU implementations. We evaluate the performance of CPU algorithms on an Intel(R) Xeon(R) E5-2620 microprocessor. The processor has 6 2.0-GHz CPU cores, each supporting 2 threads and thus we have 12 threads in total. The original performance results of LCRQ and CAQ are translated to our CPU based on the clock speed. To make a consistent comparison, we only used the results for 12 threads.

We use the following benchmarks with different distribution of enqueue and dequeue operations. Each thread will execute one thousand enqueue and dequeue operations in total. The queue initially contains $2^{25}$ items.

- **Benchmark 1 (100/0):** Each thread executes only Enqueue operations.
- **Benchmark 2 (70/30):** A thread executes queuing operations among which 70% are enqueue operations and 30% are dequeue operations.
- **Benchmark 3 (50/50):** A thread executes queuing operations among which 50% are enqueue operations and 50% are dequeue operations.
- **Benchmark 4 (30/70):** A thread executes queuing operations among which 30% are enqueue operations and 70% are dequeue operations.
- **Benchmark 5:** Each thread executes only dequeue operations.

We test the four algorithms with the five benchmarks. Each benchmark is executed for ten times and we show the average number of million operations per second (MOPS) in Fig. 2.

As shown in Fig. 2, the proposed CCLQ data structure and its routines outperform all other works in the first four benchmarks. On the fifth benchmark, CCLQ has a minor disadvantage against LCRQ. It is because the dequeue operation of CCLQ needs more CAS operations. On all bench-

![Fig. 2](image-url) Enqueue/Dequeue throughput of the four algorithms on NVIDIA GeForce GTX 780Ti and Intel(R) Xeon(R) CPU E5-2620 with 12 threads used.
marks, our algorithm outperforms MSQ by up to 40 fold. We also investigate the scalability of CCLQ and MSQ on both NVIDIA GeForce GTX 570 (Fermi architecture) and NVIDIA GeForce GTX 780Ti (Kepler architecture). We only use benchmarks 2, 3, and 4 because these are close to real scenarios. Benchmarks are tested with different number of blocks on GPGPUs. The results are shown in Figs. 5, 6, and 7, respectively. The performance of Benchmark 2 scales from 23 MOPS to 40 MOPS and then stays steady because GPU hardware resource is saturated. For Benchmarks 3 and 4, our work scales from 23 MOPS to 40 MOPS and 17 Mops/second to 30 MOPS, respectively. With an increasing number of blocks, CCLQ shows a better scalability on Kepler GPU, while its performance drops on Fermi GPU. This is due to the better support of atomic operations by Kepler GPU [16].

5. Conclusion

In this paper, we have presented CCLQ, a concurrent lock-free FIFO queue on GPUs. CCLQ exploits both CAS loops and request-combining techniques to deliver superior queuing operations. In particular, CCLQ explicitly utilizes the SIMD execution feature of GPUs to accelerate the request-combining process. Experiments prove that CCLQ outperforms prior concurrent queues on both CPUs and GPUs. It is faster than a previous GPU based concurrent queue by up to 40X.

References

[1] D. Blythe, “Rise of the graphics processor,” Proc. IEEE, vol.52, no.3, pp.761–778, May 2008.
[2] Y.D. Deng and S. Mu, “A survey on GPU based electronic design automation computing,” Foundation and Trends in Electronics Design Automation, vol.7, no.1-2, pp.1–176, June 2013.
[3] NVIDIA, Thrust: Code at the speed of light, https://code.google.com/p/thrust/, accessed Feb. 5, 2014.
[4] NVIDIA, CUDA Basic Linear Algebra Subroutines (cuBLAS), https://developer.nvidia.com/cuBLAS, accessed Feb. 5, 2014.
[5] D. Cederman, B. Chatterjee, and P. Tsigas, “Understanding the performance of concurrent data structures on graphics processors,” Euro-Par12, pp.883–894, Rhodes Island, Greece, Aug. 2012.
[6] P. Misra and M. Chaudhuri, “Performance evaluation of concurrent lock-free data structures on GPUs,” ICPADS12, pp.53–60, Singapore, Dec. 2012.
[7] M.M. Michael and M.L. Scott, “Simple, fast, and practical non-blocking and blocking concurrent queue algorithms,” PODC96, pp.267–275, Philadelphia, PA, USA, May 1996.
[8] E. Ladan-Mozes and N. Shavit, “An optimistic approach to lock-free FIFO queues,” DISCO4, pp.117–131, Amsterdam, Netherlands, Oct. 2004.
[9] M. Hoffman, O. Shalev, and N. Shavit, “The baskets queue,” OPODIS’07, pp.401–414, Guadeloupe, French West Indies, Dec. 2007.
[10] A. Gidenstam, H. Sundell, and P. Tsigas, “Cache-aware lock-free queues for multiple producers-consumers and weak memory consistency,” OPODIS10, pp.302–317, Tozeur, Tunisia, Dec. 2010.
[11] A. Morrison and Y. Afek, “Fast concurrent queues for x86 processors,” PPOPP13, pp.103–112, Shenzhen China, Feb. 2013.
[12] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, “Flat combining and the synchronization-parallelism tradeoff,” SPAA10, pp.355–364, Santorini, Greece, June 2010.
[13] P. Fatourou and N.D. Kallimanis, “A highly-efficient wait-free universal construction,” SPAA’11, pp.325–334, California, USA, June 2011.
[14] P. Fatourou and N.D. Kallimanis, “Revisiting the combining synchronization technique,” PPOPP’12, pp.257–266, New Orleans, LA, USA, Feb. 2012.
[15] M.P. Herlihy and J.M. Wing, “Linearizability: A correctness condition for concurrent objects,” TOPLAS, vol.12, no.3, pp.463–492, July 1990.
[16] NVIDIA, NVIDIA Next Generation CUDA Compute Architecture: Kepler TM GK110, http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, accessed Feb. 5, 2013.