A Concurrent Relaxed Queue for Unordered Parallel Accesses on GPUs

Mengshen Zhao
Computer and Information Science
University of Mississippi
University, USA
mzhao@go.olemiss.edu

David Troendle
Computer and Information Science
University of Mississippi
University, USA
david@cs.olemiss.edu

Byunghyun Jang
Computer and Information Science
University of Mississippi
University, USA
bjang@cs.olemiss.edu

***Abstract*—We propose a relaxed concurrent queue for Graphic Processing Units (GPUs) which supports groups of unordered enqueue and/or dequeue operations. When the group size is one, it becomes a conventional strict First-In-First-Out (FIFO) queue. We call these groups Parallel Operations Groups (POGs) and leverage a persistent thread model to support the processing of an arbitrary number of POGs. To minimize thread contention and synchronization overhead, queue operations in each POG are processed as a group then committed to the queue in a single update process. The experiment compares our proposed queue with a non-blocking concurrent queue (baseline) on synthetic inputs that simulate the diverse configurations of POGs present in real-world applications. The experiments show that our relaxed queue achieves a significant speed up over the baseline when processing a large number of POGs while retaining good scalability.**

***Index Terms*—Concurrent Queue, Relaxed FIFO Semantics, GPU, Parallel Operations Group, Data Structure**

I. Introduction

In multithreaded shared memory systems, threads synchronize and communicate with one another through data structures in shared memory. Concurrent data structures play a crucial role in achieving good performance on such systems. Queues are one of the most important data structures and are pervasively used across many domains. A number of concurrent queue design techniques have been proposed on modern multicore CPUs [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], but most of them are not directly applicable to GPUs where threads are scheduled and executed by hardware in a Single Instruction Multiple Threads (SIMT) fashion. This gives rise to increased demand for high performance and scalable queue implementation on GPUs.

In this paper, we propose a concurrent queue for GPUs with relaxed semantics. This relaxed queue supports group of unordered parallel accesses while keeping the FIFO property among groups. We call these groups *Parallel Operations Groups (POGs)*, and find them used in some parallel applications. For example, Breadth-First-Search (BFS) uses a queue to schedule nodes to search level by level in a FIFO fashion where the processing order of nodes at the same level does not matter[1]. The relaxed property of our queue can handle such cases efficiently by allowing unordered parallel enqueues and dequeues of nodes at the same level while maintaining BFS semantics. We also propose several optimization techniques utilizing the unordered property of POGs to further improve the throughput of the queue. These optimizations aim at reducing thread contention and synchronization overhead. Our experimental results show that our proposed optimizations improve the scalability of the queue when compared to the baseline where the processing time of arbitrary POG sizes are almost identical. Our proposed queue is implemented using persistent threads to minimize the kernel launch overhead while maximizing the utilization of GPU hardware resources. The contributions of this paper can be summarized as follows.

- We introduce the concept of POGs which is frequently found in parallel computing.
- We present a scalable concurrent queue with relaxed semantics that is specifically designed for highly parallel processing applications on GPUs.
- We propose a set of optimizations that leverage the combination of the relaxed semantics of the queue and the unordered property of POGs.

II. Related Work

A. Concurrent Queue on CPUs

Michael and Scott [2] presented a non-blocking concurrent queue algorithm (MS-Queue) where the ABA problem is handled by double-word Compare-And-Swap (CAS) combining the address of the new node and its version number. They also proposed two lock-based algorithms on systems that do not provide CAS. Both designs adapt the dummy node [1] thus eliminate the deadlock caused by the head and tail accessing each other. However, like many other queue implementations, the MS-queues do not scale well with a large number of concurrent threads due to its single head

[1]Note that each POG can be regarded as a concurrent pool. Thus, our relaxed semantic queue can be regarded as a FIFO of pools with domain specific optimizations.

and tail design. Repeatedly failing retries of CAS cause high contention on shared object that greatly increase the memory traffic, congesting the memory bandwidth.

One way to reduce thread contention is elimination [10]. By pairing two operations with reversed effects, these two operations can cancel each other's effect without accessing the data structure. The main challenge of implementing the elimination on queue is the FIFO property of queue. The elimination can be performed only when: 1) the queue is empty and 2) an enqueue immediately followed by dequeue. Shavit [10] proposed a technique to avoid such requirements by introducing an elimination array. When concurrent enqueues occur, they are viewed as one group. The enqueue operation successfully acquired the queue adds its element to the queue. Failed enqueue operations back off and put their elements on the elimination array before retrying. When concurrent dequeue operations begin, they are redirected to the array and search for "ripe" elements. Elements belonging to the same group are removed by dequeues prior to the one in the queue. In such case, elements on the array can be eliminated safely as if they were enqueued to the queue. According to the experiments, the queue with elimination technique is most efficient under high load since concurrent accesses to the queue are significantly reduced.

The Basket Queue [6] proposed a new type of queue that contains a list of ordered "baskets." When a group of enqueue operations executes, the thread successfully acquired the global tail places itself into the queue. Instead of retrying, the enqueue operations failed on the same CAS (issued at the same time with the successful enqueue operation) insert their nodes in front of the newly added node, forming a basket. The dequeue operation can remove nodes in the same basket in any order but it can only move to the next basket when the current basket is empty. The basket design improves the bandwidth since enqueues can be performed on multiple baskets concurrently. The major drawback is that the dequeue operations can be blocked by a basket with in-progress enqueue operations.

Kirsch [8] proposed a non-atomic k-FIFO queue implementation with relaxed queue semantics. The implementation contains k instances of regular FIFO queues and a scheduler which distributes the queue operations among the k instances. When a queue operation is ready to execute, the scheduler selects one instance randomly and calls the operations on the queue. Since the selection and queuing are performed non-atomically, the queue can potentially perform at a better performance and better scalability under high contention. The trade-off is that the queue semantic is not strictly FIFO due to the same reason. According to their test, the selection of k and type of the scheduler determines how elements are distributed, as well as the operations potentially be performed concurrently and in parallel.

B. Concurrent Queues on GPUs

Zhang et al. [14] presented a concurrent queue design that performs enqueues and dequeues on local shared data before committing the result to global queue. Enqueue and dequeue operations are mapped to each warp and copied to local storage. The number of operations are computed in the local storage and assigned with a unique id in each warp. A proxy thread in the warp then competes for the global tail and increments the value of tail by the number of enqueue operations in winning warp. Then elements are copied from local storage to global queue in one step using the unique ids of each elements. The dequeue works in a similar fashion. Although this is a simple optimization, it offers three design techniques when implementing concurrent queue on GPUs: 1) use low-latency local storage when possible, 2) reduce the number of operations performed by grouping the same type of operations and performing updates in a batch 3) use a proxy thread for a group of operations to reduce the overall contention.

Scogland and Feng [9] and the Broker Queue [11] proposed a similar queue implementation on GPU based on circular array and ticketing. The blocking ticketing technique is implemented with atomic Fetch-And-Add to avoid frequent retries caused by Compare-And-Swap operations. This technique assigns a unique id for each enqueue and dequeue operations. The id serves as two purposes: 1) selecting the target local in the global queue and 2) the id for the transaction. The first paper implements both blocking (with FAA and FAS) and non-blocking (CAS) ticketing. The experiment results prove that blocking is more scalable when compared to non-blocking under GPU's massive threaded environment. The second paper uses blocking ticketing along with work-stealing to further improve the performance of the queue. Two major merits of these papers are: 1) Strong progress conditions do not guarantee the scalability/performance of Concurrent Data Structures (CDS) on GPU. 2) When the concurrency is high, blocking may perform better than non-blocking since the CAS retries can cause high memory contention on shared object.

While existing GPU queues implement strict FIFO semantics, we propose a flexible (supporting both relaxed and strict FIFO) queue that is useful in a parallel execution environment such as GPU. Being the first in our knowledge, our proposed queue supports the concept of parallel operation groups (POGs) where operations in each group are unordered. When the POG size is one, it becomes a conventional FIFO queue.

III. Concurrent Relaxed FIFO Queue

We propose a GPU-friendly relaxed concurrent queue that supports First-In First-Out (FIFO) operations on POGs. The POG is defined as a set of unordered operations that access the queue structure simultaneously. The POG operations can be reordered arbitrarily without affecting correctness. This relaxation allows sets of unordered enqueues and dequeues to modify the dedicated sections of the queue. Each section is characterized by the queue size ($QSize$), number of enqueues ($Enqs$), and number of dequeues ($Deqs$) at run time. Depending on the queue size, there are five cases.

1) When $Qsize > 0$,
   a) $Deqs \leq Qsize$
   b) $Qsize < Deqs < Qsize + Enqs$

c) $Deqs >= Qsize + Enqs$

2) When $Qsize = 0$,

a) $Deqs \leq Enqs$

b) $Deqs > Enqs$

Figure 2a visualizes case $1a$. When dequeues and enqueues are regrouped, they are safe to execute in parallel. Since enqueues and dequeues do not overlap, the costly $atomicCAS()$ in each section can be replaced by simple load/write operations, assuming that each operation is mapped to a unique queue node. The thread mapping is shown in Figure 1. The mapping is determined at run time with low cost $atomicAdd()$. The execution order of warps does not matter as long as all operations in each section are executed. The $HEAD$ and $TAIL$ indices are simply incremented the number of dequeues and the number of enqueues at the end of the process by a proxy thread.

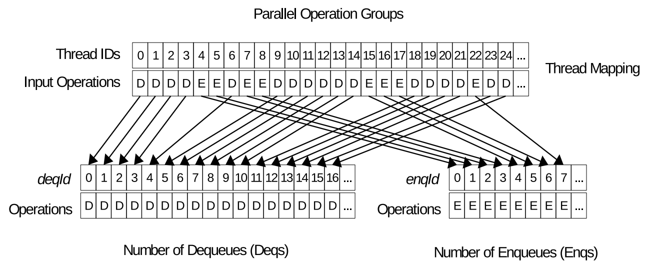
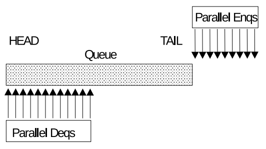
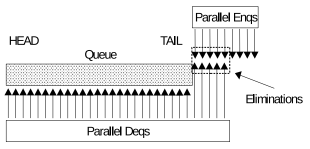

Parallel Operation Groups
Thread IDs
Input Operations
Thread Mapping
deqId
Operations
enqId
Operations
Number of Dequeues (Deqs)
Number of Enqueues (Enqs)

Fig. 1: Thread mapping on GPU.



(a) Case 1a.



(b) Case 1b.

Fig. 2: Example cases of relaxed FIFO queue with POGs.

Case $1b$ is more involved. As shown in Figure 2b, part of dequeues and enqueues are safe to execute in parallel on their assigned sections. The overlapped section in the middle indicates how many enqueues and dequeues operations perform an $enq - deq$ pair eliminations in parallel on their dedicated node. This process can be simplified by allowing dequeue operations to return items directly from the enqueues. The $HEAD$ and $TAIL$ are incremented with actual number of dequeues and enqueues by subtracting the operations in the overlapped section.

The case $1c$ works similarly to case $1a$ except the extra dequeues return $\emptyset$ symbols when all items in the queue are removed and all enqueues are consumed in the direct return process. When the queue is empty, cases $2a$ and $2b$ follow the pairing process. Dequeues simply return items from the enqueues directly. The details of queue operations are explained in section III-B.

Our approach offers two optimization opportunities. First, the queue semantics within each POG is relaxed with unordered bulk operations. Additionally, these operations significantly reduce the synchronization overhead within POGs by replacing the costly $atomicCAS()$ operations with simple read/write operations. Secondly, pairing process in the eliminations can be performed without accessing the queue structure, which alleviates the contention on the queue.

A. Operations Supported

The symbols used in this section are described below:

Q:	The Queue
N:	Count of successful request(s)
I:	Items of the queue
R:	Count of removed request(s)
$\emptyset$:	Empty symbol
$\{\}$:	A collection of objects
$\vert\ \vert$:	Size of objects
$\leftarrow$:	Object(s) returned from function
$oldX$:	Old status of the queue
$HEAD$:	The head pointer
$TAIL$:	The tail pointer

Our proposed queue supports following operations:

- $Q \leftarrow ENQ(oldQ, TAIL, \{I\})$: *ENQ* function adds a set of new items ($\{I\}$) to the old queue at position *TAIL*, the function returns a new queue containing the newly added items. Notice that the position of the items is added as a bulk without any internal order. The position of each item is determined at run-time.
- $Q, \{\emptyset\} \leftarrow DEQ(oldQ, HEAD, R)$, if $|Q| = \emptyset$, $Q, \{I\} \leftarrow DEQ(oldQ, HEAD, R)$, if $|Q| \geq R$, $Q, \{I, \emptyset\} \leftarrow DEQ(oldQ, HEAD, R)$, if $|Q| < R$: The dequeue (*DEQ*) function has three different behaviors depending on the size of the queue ($|Q|$) and the number of requests (R). If the queue is empty ($|Q| = \emptyset$), removing items from the queue returns a set of $\emptyset$s and an empty queue. If the size of the queue is greater than or equal to the number of requests ($|Q| \geq R$), the *DEQ* functions remove *R* items in bulks from the queue. The items are returned as a set with *R* items $\{I\}$. When the size of the queue is less than the number of remove requests ($|Q| < R$), all items are removed from the queue. Then the $|Q| - R$ requests are returned as empty symbols. Similar to the *ENQ* function, items are removed from the queue as a bulk. Thus, the returned items do not have an order.
- $TAIL \leftarrow TAIL\_ICN(oldTAIL, N)$ and $HEAD \leftarrow HEAD\_ICN(oldHEAD, N)$: *TAIL_ICN* and *HEAD_ICN* functions simply add an integer value *N* to the *TAIL* or *HEAD* pointer atomically, indicating the changes of *TAIL* and *HEAD*. The value of *N* is equal to the size of $\{I\}$ in ENQ and DEQ functions, excluding the $\emptyset$s.

B. Implementation

The queue structure is implemented with a bounded 1-D array that is pre-allocated on GPU's global memory. The *TAIL* and *HEAD* are atomic pointers that point to the TAIL and HEAD of the queue. Elements are enqueued to the TAIL and dequeued from the other end. The queue returns EMPTY and FULL status by verifying the size of queue.

The FIFO ordering of POGs is achieved by creating a synchronization point after each POG is processed. Our implementation takes advantage of the grid synchronization capability provided by CUDA Cooperative-Groups (CGs). This built-in function provides an efficient and safe synchronization method to the entire thread grid when needed.

To process the entire workload without relaunching the kernel, our kernel adapts the persistent thread execution model. As shown in Algorithm 1, the main body of our kernel is enclosed by a while loop. The loop repeats until the value of the *pid* (POG index) reaches the value *totalPid* (total number of POGs). In this setup, the coexisting POGs in each iteration are filtered and processed by matching their *pog_idx* and the global *pid*.

Algorithm 1 POG kernel

```
1: while $pid < totalPid$ do
2:   $pos = g\_rank + offset$
3:   if $(pogIdx[pos] == pid)$ then
4:     $enqId \leftarrow atomicAdd(\&numEnq, 1)$
5:   else
6:     $deqId \leftarrow atomicAdd(\&numDeq, 1)$
7:   end if
8:   $sync(grid)$
9:   if $(pogIdx[pos] == pid)$ then
10:    if $deqId > 0$ then
11:      $bulk\_deque()$
12:    end if
13:    if $enqId > 0$ then
14:      $elimination()$
15:      $bulk\_enque()$
16:    end if
17:  end if
18:  $sync(grid)$
19:  if $gRank == 0$ then
20:    if all POG in current grid is processed then
21:      $offset \leftarrow offset + grid\_size$
22:    end if
23:    $increment\_pid()$
24:    $reset\_vars()$
25:    $update\_queue()$
26:  end if
27:  $sync(grid)$
28: end while
```

The kernel (code executes on GPUs) is divided into three major steps. The first step is data pre-processing that simply counts the total number of enqueues and dequeues in the current POG. Each operation is assigned with a category-specific unique id using an atomic counter. The order of acquiring the unique ids does not preserve the order of operations in the input. Due to the warp scheduling mechanism of GPUs, the threads of a large POG which spans multiple thread blocks can be scheduled in any order. Therefore, a global synchronization is required to ensure all threads in the POG reach the same stage before proceeding. Without the synchronization, threads acquired the atomic operations could execute next block of code while other threads are still waiting at the atomic operation, which results in incorrect branch selection and results.

Algorithm 2 bulk_enque

```
1: $enqPos \leftarrow atomicAdd(realEnq, 1);$
2: if $QSIZE \neq 0$ then
3:   $minVal \leftarrow min((numDeq - QSIZE), numEnq)$
4:   if $enqId > minVal$ then
5:     $queue[TAIL + enqPos] \leftarrow ops[pos]$
6:   end if
7: else
8:   if $enqId > numDeq$ then
9:     $queue[TAIL + enqPos] \leftarrow ops[pos]$
10:  end if
11: end if
```

Algorithm 3 bulk_deque

```
1: if $QSIZE \neq 0$ then
2:   $minVal \leftarrow min(QSIZE, numDeq)$
3:   if $deqId \leq minVal$ then
4:     $deqPos \leftarrow atomicAdd(realDeq, 1);$
5:     $retVal[pos] \leftarrow queue[HEAD + deqPos]$
6:     $queue[HEAD + deqPos] \leftarrow 0$
7:   end if
8:   if $deqId > (numEnq + QSIZE)$ then
9:     $retVal[pos] \leftarrow -99$
10:  end if
11: else
12:   if $deqId > numEnq$ then
13:     $retVal[pos] \leftarrow -99$
14:   end if
15: end if
```

The second step contains three major operations: *bulk_enque()*, *bulk_deque()* and *elimination()*. The bulk update operations do not access the queue structure. This includes dequeuing items directly from the enqueue operations, and returning $\emptyset$ symbols by comparing values of certain variables without actually dequeuing items until the queue is empty. The details of these optimizations are given in the following subsections. Thread divergences [15] can occur with multiple control-flow paths (enqueues and dequeues are mapped to the same warp) but it can be minimized by regrouping the operations based on their types (Figure 1).

The last step is simply resetting/updating the variables for the next execution cycle of a new POG (line 20∼26). This also handles the scenarios where a POG spans multiple thread blocks. If the last operations of current thread block has the same POG index as the first operations in the next thread block, these two operations belong to the same POG. As a result, the *pid* does not increase until the POG processing is complete.

C. Enqueue, Dequeue, and Elimination Operations

The *bulk_enque()* function takes an arbitrary number of enqueue operations and commits them to the queue in a single step. As shown in the Algorithm 2, the enqueue has two paths depending on the queue size. The atomic function at line 1 indicates that the enqueue operation does not preserve the order in the *ops* array. The order of acquiring the *enqPos* determines the position to the queue in a bulk.

When the queue is non-empty (line 2 ∼ 5), the enqueue operations are compared to the min value of *(numDeq-QSIZE)* and *numEnq* to determine how many enqueue requests are actually need to be executed. The rest of the enqueue requests are being used in other functions. When the queue is empty (line 8 ∼ 9), only the operations with the *enqId* that is greater than the *numDeq* require to execute the enqueue operations.

The $bulk\_deque()$ function has similar code path as the $bulk\_enque()$ function. The pseudo code is listed in Algorithm 3. When the queue is not empty, it first determines the *min* value between the queue size and $numDeq$ (line 2). The operations whose *deqId* is less or equal to the min value performs $bulk\_deque()$ and store the returned values in the $retVal$ array (line 4 ∼5). Used slots in the queue are marked with zero values. The code block in line 8 ∼ line 9 performs the dequeues by comparing the value of $deqId$ and the summation of $numEnqs$ and $QSZIE$. The operations that meet the condition on line 8 simply return the empty symbol (-99) since the number of dequeues is greater than the number of enqueues plus items already in the queue. We simply return empty symbols if the $deqId$ is greater than the number of enqueues (line 12 ∼ 13).

To improve performance, we perform eliminations when possible. The elimination allows dequeue operations to return items directly from the enqueue operations without accessing the main queue structure. It also takes two paths depending on the size of queue. The complicated path occurs when the size of queue is not empty. By comparing the difference between $(numDeq - QSIZE)$ with $numEnq$, the threads whose $enqId$ are greater than the min value perform eliminations. The return values are stored in the corresponding slots in the $retVal$ (line 4) by referencing the value of *pos* (the index of enqueue operations in the *ops* array) since the dequeues obtain data directly from the enqueue operations. Line 3 and 4 are executed only when the number of dequeues is greater than the queue size, which avoids invalid eliminations. Selecting threads to perform elimination is simple when the queue is already empty. The enque operations with $enqId$ less than or equals to the number of dequeues return their items directly as the result of the eliminations (line 7 ∼ 8).

Algorithm 4 Elimination

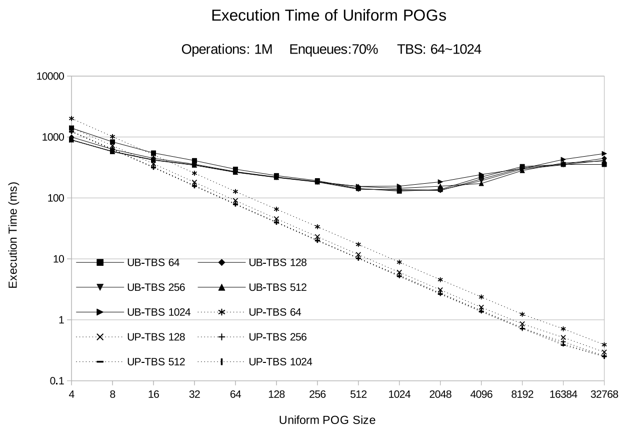
```
1: if $QSIZE \neq 0$ then
2:     $minVal \leftarrow min((numDeq - QSIZE), numEnq)$
3:     if $enqId \leq minVal$ then
4:         $retVal[pos] \leftarrow ops[pos]$
5:     end if
6: else
7:     if $enqId \leq numDeq$ then
8:         $retVal[pos] \leftarrow ops[pos]$
9:     end if
10: end if
```

IV. Experiment and Analysis

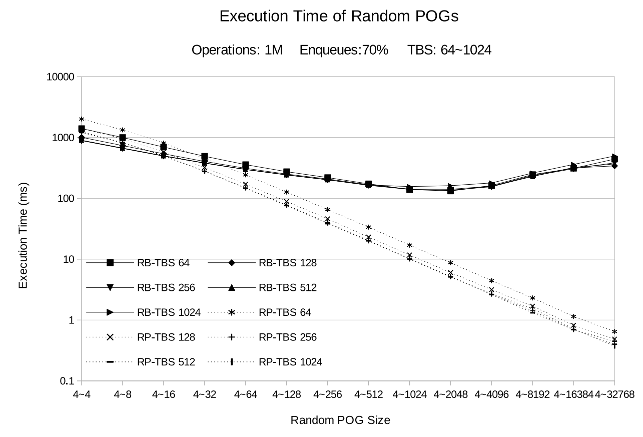
We evaluate our proposed queue under various configurations to simulate the real world use scenarios. All experiments are conducted on Nvidia RTX 3090 GPU with CUDA toolkit 11.4 under Ubuntu 20.04. The total number of persistent threads launched is set to 51,200 because it is the balance point of hardware utilization, contention, and the cost of synchronization when using CG. All tests are conducted with pre-generated synthetic input using both baseline and proposed queues. The baseline queue is a simple GPU port of a non-blocking array-based queue that supports basic enqueue and dequeue operations.

Experiments are categorized into two depending on the types of POGs - uniform and random. The uniform test case uses POGs of all the same size while the random test case contains POGs of different sizes which randomly vary between the lower and upper bounds.

First, we evaluate the performance across different thread block sizes (TBS) changing from 64 to 1024 when the POGs are uniform in their sizes (Figure 3). The number of active threads in each iteration is determined by the actual size of POGs during execution. The number of enqueue operations is set to 70% in order to avoid empty queue error when dequeue is performed. This enqueue-dequeue ratio ensures that most dequeue operations take full path instead of simple return. When the size of POG is small (e.g., 4), both the baseline and our proposed queue suffer from hardware under-utilization (Figure 3a). As the size of POG grows, both versions start to benefit from the increased concurrency with more active threads. The baseline version reaches the peak performance when the size of POG is 1024 then degrades. The increased number of threads compete for the same memory access, extending the waiting time of failed CAS operations before retrying. In contrast, the proposed queue achieves an almost linear speed up continuously as the size of POG grows. The bulk update and elimination minimize the thread contention on shared data by reducing the amount of concurrent accesses to the structure and the number of atomic operations required by each POG. Our queue implementation, however, introduces slightly more control flow in comparison to the baseline

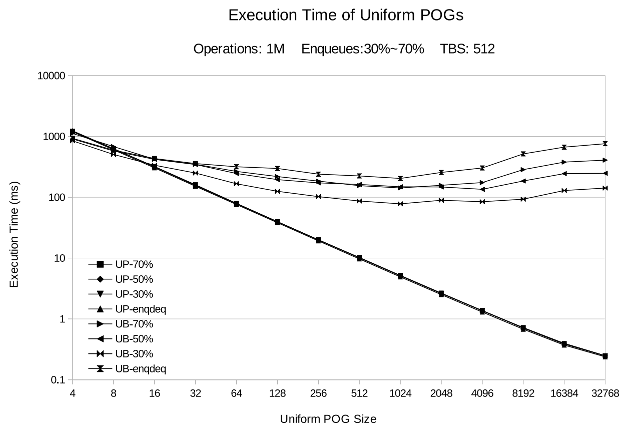


(a) Uniform POGs

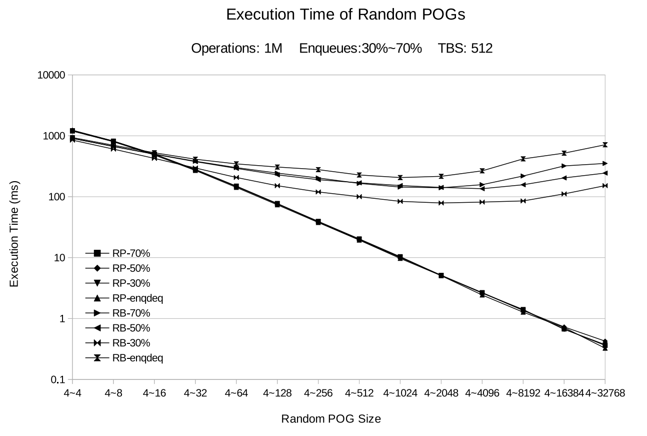


(b) Random POGs

Fig. 3: Performance across different Thread Block Sizes (TBS).



(a) Uniform POGs



(b) Random POGs

Fig. 4: Performance across different operations ratios. The **enqdeq** suffix indicates that all threads in the POG execute an enq-deq pair.

version, whose performance impact becomes noticeable when dealing with small POGs. This explains the performance gap observed at POG size 4 where the baseline version outperforms. This gap quickly decreases after the size 32 where the performance gain of higher concurrency outweighs the overhead of thread divergence. The proposed queue achieves up to 1811.34x speedup over the baseline version on the largest POG tested. This result demonstrates that our queue scales well with respect to POG size. We also notice that the overall optimal block sizes of the baseline version and our implementation are both 512.

Next, we evaluate random POG sizes under the identical configuration as previous experiment. The x-axis of Figure 3b indicates the lower and upper bounds of POG size. With a higher bound of 8192, for example, the size of POGs can be any arbitrary number between 4 and 8192. This random case contains more small POGs compared to the uniform case. Performance degradation is observed in both baseline and proposed versions with random POGs. This is because processing small POGs causes hardware underutilization and increases the global synchronization frequency in persistent thread model. The overall optimal block size of the baseline version and our implementation are both 512.

Our second experiment simulates more realistic scenarios by changing the configuration of POGs and the queue operation types (i.e., ratio of enqueues and dequeues). The thread block size (TBS) is set to 512 for optimal performance. Interestingly, we observe a similar performance trend as in the previous experiment. The baseline version, however, is sensitive to the ratio of enqueue and dequeue operations in both uniform and random POGs settings. We observed a big performance gap between UB-70% and UB-50% even though the total number of operations (one million) in both cases are the same. In UB-70%, the execution time is contributed by the waiting time of failed enqueue operations on the contended tail and data writing time. The UB-50% case performs fewer enqueues, thus the tail contention is alleviated. Although more frequent dequeue operations can exacerbate the head contention, it also increases the chances of dequeuing from an empty queue, which simply returns an empty symbol without any memory operations. The UB-30% performs even more dequeue operations compared to other cases thus the performance is further improved for the same reason. Our queue eliminates the performance gap with the proposed bulk update operations and eliminations. First, the bulk enqueue and dequeue combine the same type of updates in a single step. The contention on both ends of the queue are the same regardless of the number of enqueues and dequeues. Second, the elimination cancels out a significant number of paired operations without accessing the queue. It further reduces the memory traffic by allowing dequeue operations return values directly from paired enqueue operations. The performance difference becomes noticeable in the random POG test due to the synchronization of small POGs.

With the same distribution ratio of operations, the gap between UB-50% and UB-enqdeq is caused by the execution pattern of operations on GPUs. The queue operations are distributed randomly in UB-50% where a thread block may contain a series of the same type of operations. When executed on GPU, thread divergence occur less in such thread block. On the contrary, the enqueue and dequeue pairs in the UB-enqdeq case are forced to execute both cases due to the lock-step execution model. Depends on the size of the POGs, such execution pattern can lead to a severe performance degradation due to the thread divergence on GPUs.

In contrast, our proposed queue achieves a stable processing time in all cases regardless of the operations distribution ratio in the POG. The processing speed only slightly increases at the largest POGs where contention start to take effect. The test result shows that our queue scales well and is insensitive to the effect of ratio/distribution of operations even when the concurrency is high.

V. Conclusion

In this paper, we proposed a relaxed concurrent queue that efficiently processes unordered parallel operation groups in a FIFO fashion on GPUs. Our experiments demonstrate that our proposed queue outperforms the baseline version and shows a good scalability under high concurrency environment thanks to the persistent thread model and proposed optimizations. However, the queue suffers from hardware underutilization when processing small POGs. Processing multiple POGs in a persistent thread cycle would resolve this issues, which remains as a future work.

Acknowledgment

This research was supported by the NSF CCF 1907838.

References

[1] J. D. Valois, "Implementing Lock-Free Queues," in *In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV*, 1994, pp. 64–69. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.8674

[2] M. M. Michael and M. L. Scott, "https://doi.org/10.1145/248052.248106Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms," in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 267–275. [Online]. Available: https://doi.org/10.1145/248052.248106

[3] P. Tsigas and Y. Zhang, "A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems," in *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 134–143. [Online]. Available: https://doi.org/10.1145/378580.378611

[4] E. Ladan-Mozes and N. Shavit, "An optimistic approach to lock-free fifo queues," in *Distributed Computing*, R. Guerraoui, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 117–131.

[5] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit, "https://doi.org/10.1145/1073970.1074013Using Elimination to Implement Scalable and Lock-Free FIFO Queues," in *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 253–262. [Online]. Available: https://doi.org/10.1145/1073970.1074013

[6] M. Hoffman, O. Shalev, and N. Shavit, "The baskets queue," in *Principles of Distributed Systems*, E. Tovar, P. Tsigas, and H. Fouchal, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 401–414.

[7] D. Orozco, E. Garcia, R. Khan, K. Livingston, and G. Gao, "High throughput queue algorithms," *CAPSL Technical Memo*, vol. 103, 2011.

[8] C. M. Kirsch, H. Payer, H. Röck, and A. Sokolova, "Performance, scalability, and semantics of concurrent fifo queues," in *Algorithms and Architectures for Parallel Processing*, Y. Xiang, I. Stojmenovic, B. O. Apduhan, G. Wang, K. Nakano, and A. Zomaya, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 273–287.

[9] T. R. Scogland and W.-c. Feng, "https://doi.org/10.1145/2668930.2688048Design and Evaluation of Scalable Concurrent Queues for Many-Core Architectures," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 63–74. [Online]. Available: https://doi.org/10.1145/2668930.2688048

[10] N. Shavit and D. Touitou, "Elimination trees and the construction of pools and stacks: Preliminary version," in *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 54–63. [Online]. Available: https://doi.org/10.1145/215399.215419

[11] B. Kerbl, M. Kenzel, J. H. Mueller, D. Schmalstieg, and M. Steinberger, "https://doi.org/10.1145/3205289.3205291The Broker Queue: A Fast, Linearizable FIFO Queue for Fine-Granular Work Distribution on the GPU," in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 76–85. [Online]. Available: https://doi.org/10.1145/3205289.3205291

[12] J. Iacono, B. Karsin, and N. Sitchinava, "A parallel priority queue with fast updates for GPU architectures," *CoRR*, vol. abs/1908.09378, 2019. [Online]. Available: http://arxiv.org/abs/1908.09378

[13] 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*, ser. ICPP 2019. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3337821.3337837

[14] X. Zhang, Y. Deng, and S. Mu, "Toward concurrent lock-free queues on gpus," *IEICE Transactions on Information and Systems*, vol. E97.D, pp. 1901–1904, 07 2014.

[15] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 10.2.89," 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit