The Broker Queue: A Fast, Linearizable FIFO Queue for Fine-Granular Work Distribution on the GPU

Bernhard Kerbl, Michael Kenzel, Joerg H. Mueller, Dieter Schmalstieg, Markus Steinberger
{bernhard.kerbl | michael.kenzel | joerg.mueller | dieter.schmalstieg | markus.steinberger}@icg.tugraz.at
Graz University of Technology

ABSTRACT

Harnessing the power of massively parallel devices like the graphics processing unit (GPU) is difficult for algorithms that show dynamic or inhomogeneous workloads. To achieve high performance, such advanced algorithms require scalable, concurrent queues to collect and distribute work. We show that previous queuing approaches are unfit for this task, as they either (1) do not work well in a massively parallel environment, or (2) obstruct the use of individual threads on top of single-instruction-multiple-data (SIMD) cores, or (3) block during access, thus prohibiting multi-queue setups. With these issues in mind, we present the *Broker Queue*, a highly efficient, fully linearizable FIFO queue for fine-granular parallel work distribution on the GPU. We evaluate its performance and usability on modern GPU models against a wide range of existing algorithms. The Broker Queue is up to three orders of magnitude faster than non-blocking queues and can even outperform significantly simpler techniques that lack desired properties for fine-granular work distribution.

CCS CONCEPTS

• **Theory of computation → Massively parallel algorithms**; • **Software and its engineering →** *Scheduling*;

KEYWORDS

GPU, queuing, concurrent, parallel, scheduling

ACM Reference Format:
Bernhard Kerbl, Michael Kenzel, Joerg H. Mueller, Dieter Schmalstieg, Markus Steinberger. 2018. The Broker Queue: A Fast, Linearizable FIFO Queue for Fine-Granular Work Distribution on the GPU. In *ICS '18: 2018 International Conference on Supercomputing, June 12–15, 2018, Beijing, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3205289.3205291

1 INTRODUCTION

While the high processing power and programmability of the modern *graphics processing unit* (GPU) make it an ideal co-processor for compute-intensive tasks, its massively parallel nature creates difficulties not present on the CPU. To harness the power of the throughput-oriented GPU architecture, an application has to fit into a rigid execution model, which lacks task management and load balancing features. Without versatile task management, it is inefficient or impossible to run many common algorithms on GPUs.

This lack of features has led many researchers to implement task management for the GPU in software [4–6, 27, 28, 32]. At the core of all task management strategies are concurrent queues, which collect and distribute work, usually in a first-in-first-out (FIFO) manner. The available literature on concurrent queues has a strong focus on lock-freedom, which is often held as key to performance in concurrent systems. However, these algorithms are commonly geared towards CPU architectures and do not cater to the peculiarities of powerful and ubiquitous GPU hardware. The lock-free property is often achieved by methods in the spirit of *optimistic concurrency control* [16], *e.g.*, through algorithms that assume a low incidence of failed *atomic compare-and-swap* (CAS) operations from competing threads. However, as others [11, 13] have already noted, the overhead that is actually caused by repeatedly failing code sections can outweigh the benefits of true lock-freedom. As an alternative, blocking queues have been proposed specifically for the GPU [25]. Unfortunately, conventional blocking queues substantially limit options for load balancing, as they do not return the control to the calling thread in underflow or overflow situations. Thus, they cannot be used in multi-queue setups, which are common in advanced work distribution strategies, *e.g.*, work stealing [5].

In this paper, we present a new queue design, fit for work distribution and general queuing on the GPU. First, we identify desired properties for efficient work distribution on the GPU and assess the fitness of previous algorithms in this respect (Section 3). Based on these properties, we describe a scalable, linearizable queue, the *broker queue* (BQ), which shows the performance of a blocking queue, but can return control to the scheduler in case of underflow or overflow (Section 4). Additionally, we present two variants of the BQ, which can further improve performance at the expense of linearizable FIFO behavior. All presented algorithms

- support all execution paradigms common on the GPU: individual threads, wave fronts of single-instruction-multiple-data (SIMD) width, and cooperative thread groups,
- store arbitrarily sized data in a ring-buffer and thus avoid costly dynamic memory management,
- ensure that enqueue/dequeue are not fully-blocking if the queue is full or empty, thus enabling multi-queue setups, and
- avoid optimistic concurrency control, assigning threads to a unique spot in the queue after a fixed number of operations if the queue is not close to underflow or overflow.

For the broker queue, we prove linearizability (Section 5) and describe specifics for implementing variants of the BQ in Section 6. We compare our designs to the state-of-the-art in both synthetic tests, as well as a realistic use case (Section 7).

ACM ISBN 978-1-4503-5783-8/18/06…$15.00
https://doi.org/10.1145/3205289.3205291

2 RELATED WORK

While a basic understanding of GPU execution is necessary to address GPU queuing strategies, we avoid a lengthy summary here, but refer the interested reader to the CUDA programming guide [23] and the OpenCL specification [30]. In the following, we use the terms common in the CUDA model: *kernel* for a GPU launch; *block* for a group of threads executing on a *multiprocessor*; and *warp* for a group of threads that execute in lock-step on a GPU SIMD unit.

2.1 Massively parallel work distribution

The *persistent threads* model [1] copes with the lack of fine-grained task scheduling mechanisms in the kernel-based execution model. Persistent threads fill up the entire GPU and execute in an infinite loop. In every iteration, each thread consumes a work item from a queue, until the queue is empty. Cederman and Tsigas [4] were the first to build a load balancing system based on this approach. Although a single work queue is sufficient for simple work distribution, multiple queues are used for advanced scheduling mechanisms, such as inserting tasks from the CPU [6], task donation [32], stealing [5], or managing different task types [27, 28]. Obviously, the use of multiple queues requires the interface to be *non-blocking* and return control when a thread tries to dequeue from an already empty queue (underflow). While alternative structures other than queues have been effectively employed in work stealing schemes, doing so usually implies the abandonment of FIFO ordering [2, 12].

2.2 FIFO queues and linearizability

Arguably, the FIFO queue design, which defines a *head* from which items are drawn and a *tail* for appending items, is the most common choice in previous work. FIFO naturally lends itself to work distribution, because it implicitly maintains tasks in the order in which they were submitted. Since our focus lies with this type of queue, we henceforth use the term *queue* interchangeably with FIFO queue. In case of a single producer and single consumer scenario, a strict FIFO ordering, as in Lamport's original algorithm [17], *FastForward* [8], or *MCRingBuffer* [18], is easy to achieve.

In a massively parallel scenario, however, queues must support multiple producers and consumers. Under these circumstances, FIFO in its original sense is not applicable, since many threads can concurrently interact with the queue. This fact gives rise to the concept of linearizability, which can be used to prove observable FIFO behavior when operations overlap temporally. In short, linearizability can be understood as the constraint that an external observer, observing only the abstract data structure operations, gets the illusion that each of these operations takes effect instantaneously at some point between its invocation and its response [14].

2.3 Concurrent queue designs

One way of constructing a concurrent queue design is by using a *linked list*. Valois [33] provided one of the first lock-free, link-based queues using CAS instructions. Problems with the original design were later corrected by Michael and Scott [20], although the corrections greatly diminish its practical value. An alternative is provided by the Michael-Scott queue [21] (MSQ), which is still among the most popular lock-free concurrent queues. The authors further present a blocking queue, that supports concurrent insertion and removal using two locks, which we call *dual mutex queue* (2MQ). The baskets queue (BAQ) by Hoffman et al. [15] presents a variation on the Michael-Scott queue, exploiting the fact that no binding order can be defined for elements that are concurrently inserted into the queue, and thus any ordering is equally valid.

Array-based queues employ a continuous array of elements, commonly operated as a ring buffer. The early work by Gottlieb et al. [9] introduced a first array-based queue (GQ), that scales linearly to a large number of cores due to its fine-grained locking approach. In addition to head and tail pointers, it uses two counters to track concurrent enqueues and dequeues. However, due to its simple design, GQ is not linearizable [3]. Orozco et al. [24] addressed this issue by presenting the circular buffer queue (CBQ), which avoids the additional counters, but acts fully blocking during enqueue and dequeue. As an alternative, they propose the high throughput queue (HTQ), which returns to the problematic two-counter approach. Valois [33] proposed a lock-free, array-based queue. However, it relies on CAS on non-aligned memory addresses, which is not supported by common processor architecture and thus renders it impractical. The ring buffer by Shann et al. [26] appears to be the first practical lock-free, array-based queue (SHCQ). Also lock-free, Tsigas and Zhang [31] present another queue (TZQ), which reduces contention by updating parts of the queue only periodically. Recently, Morrison and Afek [22] proposed the lock-free linked concurrent ring queue (LCRQ), which consists of multiple linked arrays (CRQs) and avoids contention on CAS operations. Similarly, Yang and Mellor-Crummey [34] presented a segmented, wait-free queue (WFQ) that uses a fast-path/slow-path dynamic to avoid stalling threads. However, both techniques rely heavily on dynamic memory allocation, cleanup routines and hazard pointers, which are slow and tedious mechanisms to include in GPU implementations.

Recently, Scogland and Feng [25] proposed a blocking array-queue, built on top of a ring buffer ticket system (SFQ), that can also be used on the GPU. Their queue essentially extends CBQ with a closing mechanism to jump out of the blocking waits. While their approach offers high performance, it is blocking when the queue is full or empty, and only a single thread in a warp is allowed to interface with the queue. A non-blocking interface to the queue (NSFQ) circumvents this, but also drastically reduces performance.

3 REQUIREMENTS OF GPU QUEUES

Queuing algorithms on the GPU not only have to handle thousands of concurrent enqueue and dequeue operations correctly, but also need to consider the specifics of the underlying hardware. This includes confinement to a limited amount of memory, constraining register usage, and operating on a SIMD device, where individual lanes can diverge. Thus, the desired characteristics for a work queue on the GPU differ significantly from those on the CPU. A listing for availability of these properties in the queuing methods mentioned above, as well as our technique, is given in Table 1.

Many recent non-blocking queuing algorithms rely on optimistic concurrency control [16]. However, the high resource contention on the GPU—when thousands of threads try to access the same data element—can lead to a significant number of retries, *e.g.*, hundreds to thousands of repeated CAS operations for a single enqueue.

Table 1: While many parallel queues have been proposed, most lack desired properties for efficient work distribution on the GPU. General lock-free queues are commonly dependent on dynamic memory and therefore difficult to realize on the GPU. Faster queuing approaches either lack linearizability, or their rigorous blocking behavior precludes multi-queue setups. Our queue (BQ) fulfills all identified desired properties for massively parallel work distribution.

	MSQ	BAQ	SHCQ	TZQ	GQ	CBQ	HTQ	LCRQ	SFQ	NSFQ	2MQ	WFQ	*BQ*
highly scalable					●	●	●	●	●			●	●
fair ordering						●		○	●			○	●
linearizability	●	●		●		●		●	●	●	●	●	●
low resource footprint	●	●	●	●	●	●	●		●	●	●		●
static memory only					●				●	●	●		●
all execution paradigms	●	●	●	●	●		●	●		●	●	●	●
multi-queue support	●	●	●	●	●		●	●		●	●	●	●
multi-element dequeue											●		●

● . . . fulfills the requirement, ○ . . . partially fulfills the requirement

Obviously, such behavior impacts performance negatively. In accordance with other authors [11, 13], we argue that, in order to design a work queue that is highly scalable, a potentially blocking algorithm is preferable over using contended CAS operations. To *avoid retries* on failed CAS operations, every thread has to be assigned a unique spot in the queue. This requirement intuitively leads to an array-based queue design, using atomic fetch-and-add (FAA) instead of CAS. Although such a design also brings forth a single point of contention, performance on the GPU can still be high, as FAA operations are very efficient on recent GPU generations [10].

Using FAA on head and tail pointers, in combination with a per-element ticketing system, can be further extended to enable ***fair ordering***: the algorithm does not constrain head and tail pointers to the size of the ring buffer, but rather allows them to wrap around to simulate an array of infinite length, so they can yield both a ring buffer location and a ticket. As soon as a thread reaches the FAA on a required pointer (head to dequeue, tail to enqueue), its position is assigned. From that point on, there is no risk of that thread taking significantly longer to complete an enqueue or dequeue due to interference of queuing-related operations, *e.g.*, due to failing CAS. Our queue, CBQ and SFQ offer these guarantees; LCRQ and WFQ use tickets that may be invalidated by contending threads.

An important property of a queue for work distribution is guarantee of predictable behavior. For example, the queue must not sporadically report overflow or underflow, or appear to reorder elements already in the queue. The accepted standard for proving predictable behavior is ***linearizability***, which applies to most related work, with the exception of GQ (shown by Blelloch et al. [3]), HTQ (by design [24]), and TZQ (shown by Colvin and Groves [7]).

On the GPU, the degree of parallelism (or *occupancy*) that can be achieved at runtime is dictated by the resource requirements of a kernel. For example, exceeding a certain number of registers may reduce the number of concurrently launched warps and thus the ability of the GPU to effectively hide latency. Since the queuing algorithm must be embedded in the kernel in order to use it for work distribution, a ***low resource footprint*** is desirable to allow for high occupancy of the routines built on top of it. Due to their elaborate design, even bare-bone implementations of LCRQ and WFQ reduce occupancy on current GPUs according to our tests.

Large numbers of dynamic memory management operations are known to be a potential bottleneck for GPU execution [29]. Using ***static memory only*** implicitly avoids these potential overheads. Hence, a work queue on the GPU should avoid dynamic memory allocation, which, in theory, puts array-based queues at an advantage over link-based queues. However, array-based queues often need to allocate memory for queued elements individually, as the ring buffer is operated using CAS, and thus can only store pointers to the actual elements. If elements are instead stored in the ring buffer directly, access to the buffer needs to be secured, to avoid read-before-write and write-before-read hazards.

The peculiarities of the GPU yield multiple ***programming and execution paradigms***. General queue designs must be able to work within all of them, including independent thread execution, warp-synchronous execution, sub-block execution, and cooperative block execution. This requires a queue design that does not transfer blocking states between threads in the same warp, *i.e.*, halt ready-to-execute threads because other threads are being stalled. Similarly, ***multi-queue*** setups require threads to eventually return from dequeue operations if a queue is already empty, so they can probe other queues for available data. Essentially, both requirements boil down to queues being *non-blocking when a queue is full or empty.*

Our proposed design, the broker queue—although forgoing the non-blocking property of most recent queue designs—exhibits all desired properties listed above. Furthermore, it enables a thread to ***dequeue multiple elements*** at once, raising efficiency in cooperative block execution scenarios. It shows all advantages of simpler, conventional blocking queues, while also ensuring linearizability and detecting overflow and underflow without blocking execution.

4 THE BROKER QUEUE

The core functionality of the broker queue is defined by its four integral components: (1) a ring buffer for directly storing elements, (2) a head and a tail pointer for ticketing, (3) a ticket buffer that locks individual queue elements, and (4) an explicit counter to weigh enqueue against dequeue operations. The configuration of these buffers, as well as the interface to enqueue/dequeue, is given in Algorithm 1. Note that $\Lleftarrow$ indicates an atomic transaction, whereas $\Leftarrow$ is a non-atomic transaction, and $\longleftarrow$ a local variable assignment.

ALGORITHM 1: Broker Queue of size **N**

```
1  $QueueElements\ RingBuffer[\mathbf{N}]$          with $\mathbf{N} = 2^n$
2  $unsigned\ int\ Tickets[\mathbf{N}] \leftarrow \{0, 0, \cdots, 0\}$
3  $unsigned\ int\ Head \leftarrow 0, Tail \leftarrow 0$
4  $int\ Count \leftarrow 0$
5  enqueue $(Element)$
6      while not ensureEnqueue () do
7          $(head, tail) \Lleftarrow (Head, Tail)$
8          if $\mathbf{N} \leq tail - head < \mathbf{N} + MaxThreads/2$ then
9              return Full
10     putData $(Element)$
11     return Success
12 ensureEnqueue ( )
13     $Num \Lleftarrow Count$
14     while true do
15         if $Num \geq \mathbf{N}$ then
16             return false
17         if atomicFetch&Add $(Count,1) < \mathbf{N}$ then
18             return true
19         $Num \leftarrow$ atomicSub $(Count,1) - 1$
20 putData $(Element)$
21     $Pos \leftarrow$ atomicFetch&Add $(Tail,1)$
22     $P \leftarrow Pos \,\%\, \mathbf{N}$
23     waitForTicket $(P, 2 \cdot (Pos/\mathbf{N}))$
24     $RingBuffer[P] \Leftarrow Element$
25     $Tickets[P] \Lleftarrow 2 \cdot (Pos/\mathbf{N}) + 1$
26 dequeue ( )
27     while not ensureDequeue () do
28         $(head, tail) \Lleftarrow (Head, Tail)$
29         if $\mathbf{N} + MaxThreads/2 \leq tail - head - 1$ then
30             return Empty
31     return readData ()
32 ensureDequeue ( )
33     $Num \Lleftarrow Count$
34     while true do
35         if $Num \leq 0$ then
36             return false
37         if atomicSub $(Count,1) > 0$ then
38             return true
39         $Num \leftarrow$ atomicFetch&Add $(Count,1) + 1$
40 readData ( )
41     $Pos \leftarrow$ atomicFetch&Add $(Head,1)$
42     $P \leftarrow Pos \,\%\, \mathbf{N}$
43     waitForTicket $(P, 2 \cdot (Pos/\mathbf{N}) + 1)$
44     $Element \Leftarrow RingBuffer[P]$
45     $Tickets[P] \Lleftarrow 2 \cdot ((Pos + \mathbf{N})/\mathbf{N})$
46     return $Element$
47 waitForTicket $(Pos, ExpectedTicket)$
48     $Ticket \Lleftarrow Tickets[Pos]$
49     while $Ticket \neq ExpectedTicket$ do
50         backoff ()
51         $Ticket \Lleftarrow Tickets[Pos]$
```

4.1 Brokering

Usually, atomically operated head and tail pointers for ticketing prohibit a non-blocking reaction to over- and underflow. For example, if the queue holds a single element and multiple threads increase the head pointer atomically, the head is moved past the tail. Although threads could detect that the pointer was moved too far, reverting the move is difficult, as it would require a coordinated effort of all involved threads. Additionally, other threads could, in the meantime, enqueue new elements, thereby validating some of the dequeue operations that were already rolled back.

To avoid these issues, we introduce an additional counter variable (*Count*). It ensures that only threads which are guaranteed to eventually complete their enqueue or dequeue operation (and thus validly move head or tail) are allowed to interact with those pointers. For enqueue, this assurance is provided by the ensureEnqueue method, which returns **true**, iff there is either sufficient space in the ring buffer to store an element, or a sufficient number of other threads have already committed to dequeuing elements from the queue. Similarly, ensureDequeue returns **true**, iff there is an element in the ring buffer for the thread to dequeue, or at least one other thread committed to enqueue an unclaimed element. Thus, *Count* essentially models the relation between head and tail after all operations of concurrently active threads have completed.

If *Count* is decreased below zero or increased above the ring buffer size, a thread can perform a rollback by simply calling the opposite atomic operation (lines 19 and 39), without the need of explicit coordination with other threads. Note that, due to the chance of other threads modifying *Count* in the meantime, the result of the rollback may suggest that the operation is now, in fact, possible. As other threads may have picked up on this (previously invalid) assurance, the thread must try to verify this by atomically modifying *Count* one more time. This retry behavior is realized by a loop over the corresponding instructions (lines 13-19 and 33-39).

This part of the queue interface can be viewed as a *broker*, giving rise to the name of our queue. The broker not only considers items stored in the ring buffer of the queue, but also accepts assurances to provide or consume items, before the actual transactions occur.

4.2 Data storage and exchange

The internal methods of the broker queue match the assurances of the broker to actual ring buffer slots and create a connection between enqueue and dequeue operations. A slot identifies the location for writing/reading the centralized ring buffer of the broker queue. The atomic operations on *Head* and *Tail* (line 21 and 41) return a tally for computing the ticket number and, implicitly, a ring buffer slot for reading or storing elements (line 22 and 42). The ticketing itself assigns even-numbered tickets to enqueue operations and odd numbered tickets to dequeue operations. Since the broker already confirmed at this point, that performing the assured operations will yield a valid queue state and thus will eventually succeed, putData and readData simply implement a blocking behavior. This is achieved by waiting on a spinlock in waitForTicket, until the thread's turn has come to interact with the ring buffer location. To achieve favorable scheduling, threads back off after an unsuccessful spin. Each successful operation increases the ticket position by one. Consistency on integer wrap-around can be easily guaranteed by choosing a power of two as queue size and using unsigned integers for pointers and tickets.

The methods `ensureEnqueue` and `ensureDequeue` are themselves called from a loop by the queue interface. The motivation behind this design is linearizability. A broker state indicating that there are no available slots does not necessarily guarantee that **Full** or **Empty** must actually be observable in the linearized operation of the queue. For example, a thread might reduce the *Count* variable to zero through `dequeue`, but get suspended before changing the *Head*. Another thread—just examining the *Count* variable—would assume the queue to be empty, although the previously assured `dequeue` might happen much later, and thus the queue never (observably) reached the **Empty** state. As *Count* might have changed during the execution of `putData` or `readData`, threads are required to continuously try to register their operation. Therefore, a thread that detects a potential **Full** or **Empty** state waits until that state can be definitely observed (loop from line 6 to 9 and 27 to 30).

4.3 Further remarks

Next to enabling simultaneous access by an arbitrary number of threads, the biggest advantage of the BQ is that threads are only stalled if the queue is close to running empty or full. If there is sufficient time between adding an element and it being read, no thread has to wait. Another advantage of the BQ is that the ticketing system can grant threads access to queue elements for an extended period. As read and write operations on the actual elements do not need to be atomic, the queue can return a pointer to the acquired slot, *i.e.*, returning *P* instead of reading or writing (line 24 / 44).

To determine whether the queue is full or empty, we rely on comparing *Head* and *Tail*. Thus, both variables need to be read in a single atomic instruction, which is enabled on current GPU designs by defining them as 32-bit wide offsets from the buffer address and placing them together in a 64-bit word. Alternatively, actual 64-bit pointers could be used on architectures that support the atomic CAS2 operation. Note that, due to our assurance-based interaction with the pointers, *Head* can overtake *Tail*, and the distance between the pointers can grow beyond the queue size. Thus, special care needs to be taken when comparing the pointers (line 8 and 29).

At first glance, it would appear that the *Count* variable presents a central choke point for the queuing algorithm. Recent approaches, such as LCRQ and WFQ, take special care to avoid singular, global variables for communicating queue states across threads. This is motivated by the fact that, on many conventional architectures (*e.g.*, x86), contended atomic operations incur a severe performance penalty. However, due to their importance for massively parallel applications, atomic operations are extremely efficient in GPU hardware and handle contention well. Figure 1 shows the average time required for FAA operations on a single global variable, relative to uncontended memory access. While this ratio rises sharply for CPU architectures with an increasing number of contending threads, the GPU architecture is much more forgiving. Furthermore, the contention on *Count* becomes significant only when the queue is facing either underflow or overflow; *i.e.*, when *Count* is changed multiple times by a single thread. Hence, the usage of *Count* in the algorithm comes at the consideration of the underlying hardware and its low demand in balanced scenarios.

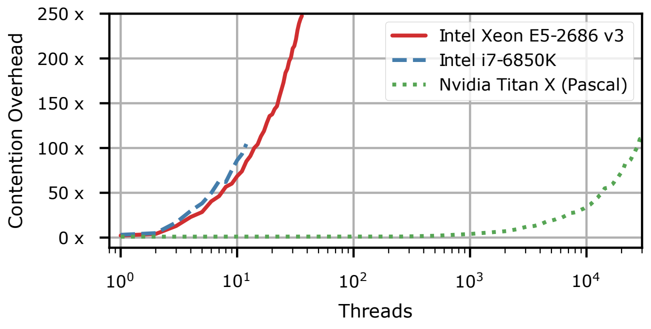


Figure 1: Average time required for a contended FAA instruction, relative to a single, non-atomic memory transaction on the respective architecture. On the GPU, a $10\,000\times$ contended FAA shows roughly the same overhead as $10\times$ on the CPU.

5 LINEARIZABILITY

To prove linearizability [14], one can model access to the queue as a history *H*. Every function call is represented by an invocation-response pair of events in the history. Two events are said to be ordered in *H*, if the response of one precedes the invocation of the other. If such an ordering is not possible for any two events, they are considered overlapping, and a linearizable data structure is allowed to order them arbitrarily. Linearizability is given if the partial ordering of event pairs can consolidate a total ordering such that the specifications of the data structure are fulfilled.

The semantics of the broker queue are those of a concurrent FIFO queue, which takes on three states: **Success** in case `enqueue` or `dequeue` succeeded, **Full** if `enqueue` is not possible, as the queue is full, and **Empty** in case there is no element in the queue. There are two relevant parts for showing linearizability of the broker queue: the exchange of data through `putData` and `readData`, as well as the brokering through `ensureEnqueue` and `ensureDequeue`.

5.1 Data storage and exchange

To show the linearizability of `putData` and `readData`, we consider threads that never see **Full** or **Empty** (ignoring `ensureEnqueue` and `ensureDequeue` for now). To this end, we use an auxiliary array *H* of infinite length, storing event pairs observed for every enqueue call $E_i = (e_i, \bar{e}_i)$ and dequeue call $D_i = (d_i, \bar{d}_i)$. Each event shall be associated with its position in *H*, *i.e.*, $e_i < e_j$ iff e_i is recorded before e_j. An event shall be recorded during the atomic operations on *Tail* (e_i) and *Head* (d_i) (line 21 and 41) and after receiving a ticket ($\bar{e}_i, \bar{d}_i$) (line 23 and 43). For example, $H = \{e_1, e_2, \bar{e}_2, \bar{e}_1, d_1, \bar{d}_1, \ldots\}$. Every event pair E_i and D_i shall be associated with $Pos = i$ obtained by the calling thread, and *Pos* shall reflect the FIFO ordering of elements in the queue (ignoring wrap-around of *Pos* for now). Thus, for linearizability, the following ordering must hold:

$$E_i < E_j \ \land \ D_i < D_j \ \land \ E_i < D_i \ \land E_i < D_j \qquad \forall i < j$$

Obviously, $E_i < E_j$ and $D_i < D_j$ is trivial to observe, as the atomic counter makes sure that $e_i < e_j$ and $d_i < d_j$. Thus, either the respective calls are non-overlapping ($\bar{e}_i < e_j$, $\bar{d}_i < d_j$) and no reordering is necessary, or they do overlap and can be reordered to fulfill the requirements. For a single pair of calls E_i and D_i, it can be shown that $E_i < D_i$: given that tickets are unique, `waitForTicket` during dequeue must wait for enqueue to issue the dequeue ticket, and $\bar{e}_i < \bar{d}_i$. Thus, an ordering $E_i < D_i$ is certainly possible, as they are either ordered correctly or overlapping. What remains to be shown, is that all three requirements hold at the same time, *i.e.*, one reordering does not contradict another and $E_i < D_j \ \forall i < j$. The only possibility for an overall reordering to fail is if $\bar{d}_j < e_i$, *i.e.*, a dequeue finishes before an earlier enqueue starts, as this would make the calls non-overlapping and prohibit a reordering. This is not possible, due to the atomic on *Tail*, which yields $e_i < e_j$. In combination with $\bar{e}_i < \bar{d}_i$ and $e_i < \bar{e}_i$, we find $e_i < e_j < \bar{e}_j < \bar{d}_j$, and $\bar{d}_j \nless e_i$. Thus, all calls can be reordered according to *Pos*.

Since *RingBuffer* is of limited size, threads may potentially be competing to access the same elements. If multiple enqueue and dequeue operations are assigned to the same element, the ticket system makes sure that the order is kept as intended. The ticket for E_i is given by $T_{E_i} = 2 \cdot \lfloor i/\mathbf{N} \rfloor$, for D_i, $T_{D_i} = 2 \cdot \lfloor i/\mathbf{N} \rfloor + 1$. After a wrap-around, $T_{E_{i+\mathbf{N}}} = 2 \cdot \lfloor (i+\mathbf{N})/\mathbf{N} \rfloor = 2 \cdot \lfloor i/\mathbf{N} \rfloor + 2$ and $T_{D_{i+\mathbf{N}}} = 2 \cdot \lfloor i/\mathbf{N} \rfloor + 3$, *i.e.*, every operation receives a unique ticket which is monotonically increasing. In this way, the ordering at each spot of the ring buffer is ensured, as long as the the tickets do not wrap around. If *Pos* wraps around at 2^{32}, the tickets wrap around at $2^{32}/\mathbf{N} = 2^{32-n}$. As long as the number of threads concurrently interacting with the queue stays below this value, the same ticket cannot be issued more than once at the same time. Hence, the order of operations on individual elements follows *Pos*, and the queue in general maintains the indented linearizable FIFO behavior.

5.2 Brokering

Brokering revolves around the ensure functions, which may return **Full** or **Empty**. If ensureEnqueue/ensureDequeue returns **true**, a thread is forwarded to `putData`/`readData`, which results in linearizable behavior as outlined above. Thus, only the **Full** and **Empty** cases require a more detailed analysis. Ignoring the wrap-around of *Head* and *Tail* for now, we define two additional events $\infty_{h,t}$ and $\varnothing_{h,t}$. If the call returns **Full** or **Empty**, these events shall be recorded during the combined head and tail reads (line 7 and 28), with $h = head$ and $t = tail$. Linearizability at underflow is given, if calls can be reordered such that an **Empty** state is reached at $\varnothing_{h,t}$:

$$D_t < \varnothing_{h,t} < E_{t+1}.$$

Ignoring wrap-around, **Empty** is returned for $t - h \leq 0$, *i.e.*, when both pointers are the same or *Head* has overtaken *Tail*. Observing such a pointer pair means that e_t and $d_t, d_{t+1}, \ldots, d_h$ have been recorded, and e_{t+1} has not happened yet:

$$e_t, d_t < \varnothing_{h,t} < e_{t+1}.$$

All D_i with $i > t$ are irrelevant for the **Empty** in question, all e_i and d_i for $i \leq t$ have already taken place (and thus E_i and D_i are either completed or overlapping), and e_{t+1} has not occurred yet. Thus, there is no E or D that prevents a reordering to achieve $D_t < \varnothing_{h,t} < E_{t+1}$. Furthermore, there are no other **Empty** or **Full** events that can interfere with creating such an **Empty** state: $\infty_{h,t}$ cannot take place at the same time (as the conditions for h and t are different). Another event $\varnothing_{h_2,t_2}$ with $t_2 = t$ and $h_2 = h$ may take place at the same time—and can simply be inserted right before or after $\varnothing_{h,t}$. An **Empty** event with $t_2 = t$ and $h_2 \neq h$ is also possible, which would be treated identically. If $t_2 < t$, the event has already been inserted into H earlier. Thus, the **Empty** state is linearizable.

Linearizability considering the **Full** state is exactly symmetrical to the **Empty** state, with $E_{h+\mathbf{N}} < \infty_{h,t} < D_{h+1}$. For the sake of brevity, we omit repeating the derivation here.

Finally, the wrap-around of the pointer after 2^{32} must be considered. It is possible for *Head* to overtake *Tail*, with a factor equal to half the maximum number of concurrently active threads—if all threads are concurrently enqueuing and dequeuing, and all operations on the *Head* occur before the ones on *Tail*. Similarly, *Tail* can advance by half the maximum number of concurrently active threads further than **N** away from *Head*. These conditions can simply be included into the comparison as an additional margin (line 8 and 29). This condition obviously fails if $\mathbf{N} + MaxThreads/2 \geq 2^{32}$.

6 BROKER QUEUE VARIANTS

To ensure linearizability, our broker queue potentially waits until suspected **Full** and **Empty** states are observable from *Head* and *Tail*. Obviously, waiting comes at a cost. Hence, we also derive a simplified version of BQ which avoids waiting by shedding linearizability, yielding the *broker work distributor* (BWD). Dropping linearizable FIFO behavior opens the door for potentially even more efficient work distribution methods, *e.g.*, work stealing [2, 12]. As BQ is also applicable in these use cases, we additionally describe the *broker stealing queue* (BSQ) for effective stealing of queued tasks.

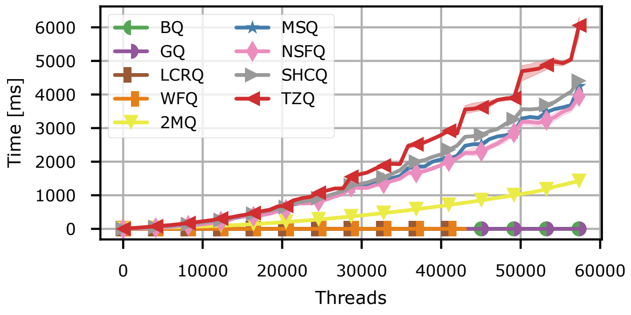
6.1 The Broker Work Distributor

The conversion from broker queue to the broker work distributor is straightforward. Instead of waiting for ensureEnqueue and ensureDequeue in a loop to ensure **Full**/**Empty** are actually observable, these function are called only once by enqueue and dequeue. The result of this call is taken at face value, returning **Full**/**Empty** if the broker cannot find a slot/match immediately.

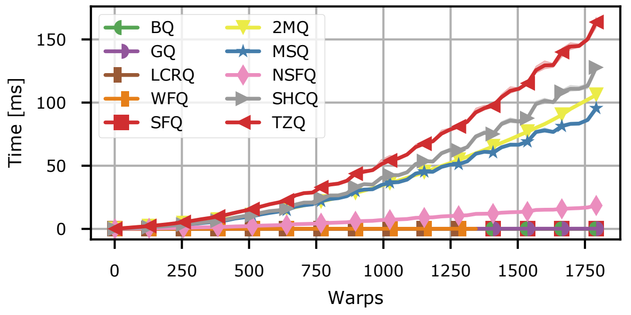
The downside of the BWD is its non-linearizability. Since *Count* is only used as an assurance swap, it does not faithfully represent the real queue state observable when the actual data is put into the queue or taken out of the queue. While this behavior is undesirable when a queue needs to behave strictly like a concurrent FIFO queue, it is generally not detrimental during work distribution. If an ensureEnqueue yields **false**, it indicates that, according to all threads that started interacting with the queue thus far, all elements will be drained from the queue; *i.e.*, unless another thread starts enqueue, the queue will reach **Empty**. This behavior is arguably sufficient for work distribution and, with regard to multi-queue setups, provides a reasonable indicator for efficiently switching to another queue that might already contain work.

6.2 The Broker Stealing Queue

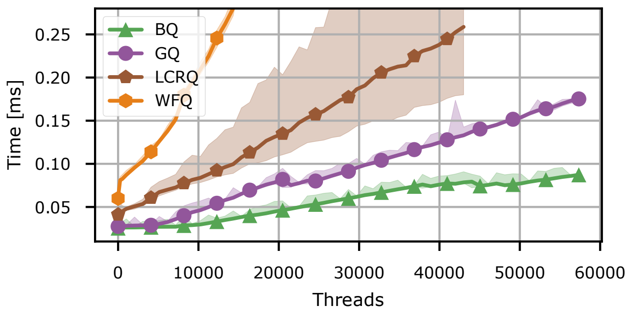
The broker stealing queue (BSQ) provides a simple work stealing implementation by abstracting multiple underlying queues through one interface. Each executing block on the GPU is assigned its own, default BQ for storing and reading queued elements. If a thread in a block cannot find an item in its assigned default queue, it tries to steal work from a different block. This is achieved by iterating over all available queues and performing a standard dequeue on each, until an element is found or all queues have been checked.



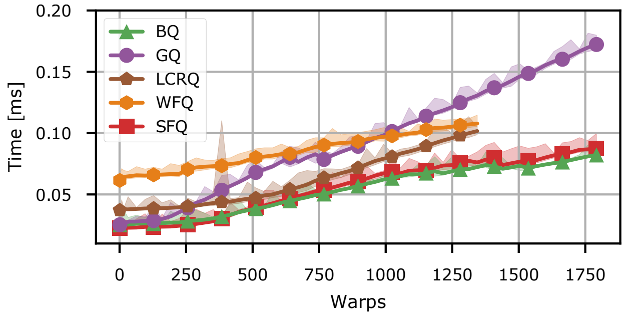
(a) Runtime of all queues, with thread granularity



(b) Runtime of all queues, with warp granularity



(c) Details for fastest queues, with thread granularity



(d) Details for fastest queues, with warp granularity

Figure 2: Runtime performance results of all queues for 10 enqueue/dequeue operations, with detailed focus on fastest queues.

7 EVALUATION

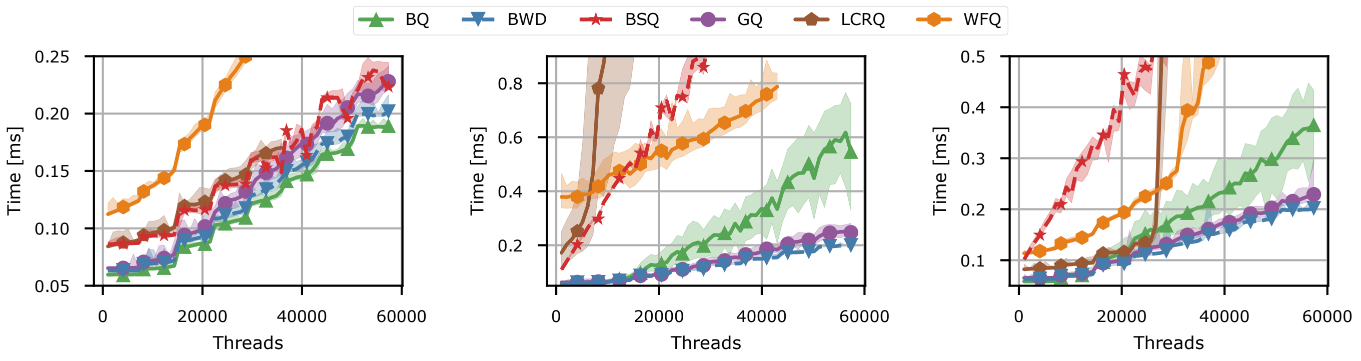
To evaluate our techniques, we compare their aptitude for work distribution with previous techniques. We implemented the queues listed in Table 1 in CUDA—with the exception of BAQ (as it is within 2× of MSQ), and CBQ and HTQ, which are similar to SFQ and GQ, respectively. In order to offer an exhaustive, yet reasonably concise evaluation of our algorithm against numerous previous approaches, we first identify the most competitive techniques in a microbenchmark. For the strongest contenders, we provide a more detailed analysis under both lenient and strenuous conditions. All tests were performed on an NVIDIA GTX Titan X (Pascal). Supplemental material documenting test results on NVIDIA Maxwell and Kepler architectures (and confirming the trends presented here) is available under https://bitbucket.org/brokering/broker-queue.

7.1 Initial runtime comparison

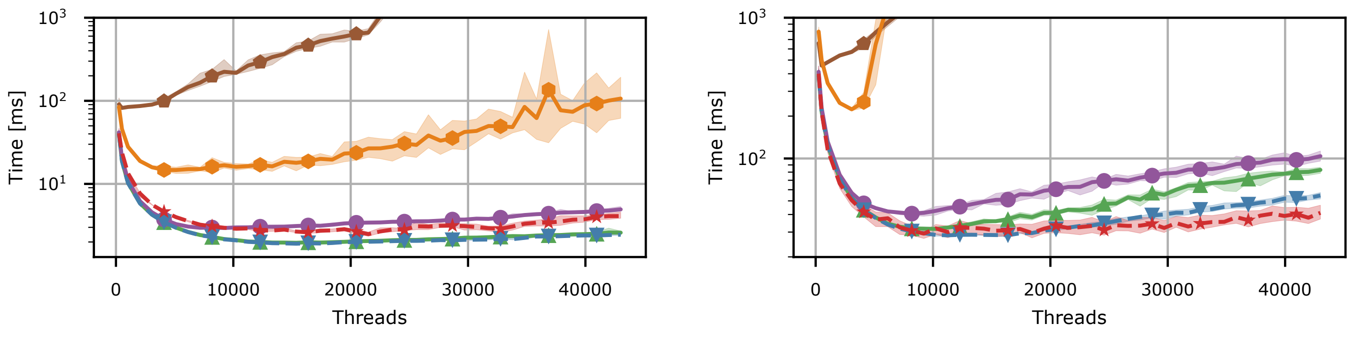
Our initial microbenchmark performs 10 alternating enqueue-dequeue pairs over a varying number of concurrently running threads. Due to this ideally balanced setup, we can include blocking queues into the test, as neither **Empty** or **Full** states are reached. Figure 2a shows the average runtimes. Due to their high register usage, LCRQ and WFQ reach the maximum number of concurrently running threads at 43 008 threads on the Titan X (Pascal), *i.e.*, they achieve 37% less occupancy than the other approaches. Since SFQ can only execute at a per-warp granularity, we repeat the above experiment with only one thread in each warp accessing the queue (Figure 2b).

These initial experiments confirm that non-blocking strategies, based around the concept of optimistic concurrency control, do not work well with thousands of threads. All four non-blocking queues built around optimistic CAS (TZQ, SHCQ, NSFQ, and MSQ) are trailing significantly behind the others. Even a queue that allows only two threads concurrent access (2MQ) can be significantly faster. However, as the number of concurrent threads approaches maximum occupancy, all of the above techniques are more than 1000× slower than the remaining algorithms with per-thread granularity, and more than 100× slower with per-warp queuing interaction. Fastest runtimes for these initial tests are obtained by our queues, as well as GQ and the recently proposed SFQ, LCRQ and WFQ. Note that we have omitted BWD and BSQ from the plots, since they exhibit virtually identical behavior to BQ in this balanced scenario.

A closer look at the runtime performance of the faster contenders is given in Figure 2c and 2d, which include lowest and highest measured runtimes as overlay. Although GQ is non-linearizable, it trails behind BQ with a slowdown of more than 2× for launch configurations exceeding 45 056 threads (or 1504 warps), caused by continuous modification of the two counters in addition to front and back pointers. LCRQ and WFQ have a higher base cost than all other techniques (3–6× compared to BQ) and quickly deteriorate at per-thread granularity, but catch up with the non-linearizable GQ per-warp. With an increasing number of threads accessing the queue, LCRQ also shows the highest variance in runtime. Per-warp, LCRQ slowly loses its advantage over WFQ's higher base cost. Although SFQ is conceptually much simpler and less versatile than our queue, it is still narrowly outperformed by the BQ. For launch configurations with >512 warps, we found a relative slowdown between 1.4% and 9%. We ascribe this fact to BQ not having to poll a closed state. Overall, BQ poses the fastest queue in this scenario.



(a) $\mathbb{P}(enq) = 50\%$, $\mathbb{P}(deq) = 25\%$, **no pre-fill** **(b)** $\mathbb{P}(enq) = 25\%$, $\mathbb{P}(deq) = 50\%$, **no pre-fill** **(c)** $\mathbb{P}(enq) = 25\%$, $\mathbb{P}(deq) = 50\%$, **pre-filled**



(d) Page rank for *p2p-Gnutella31* **(e) Page rank for *web-NotreDame***

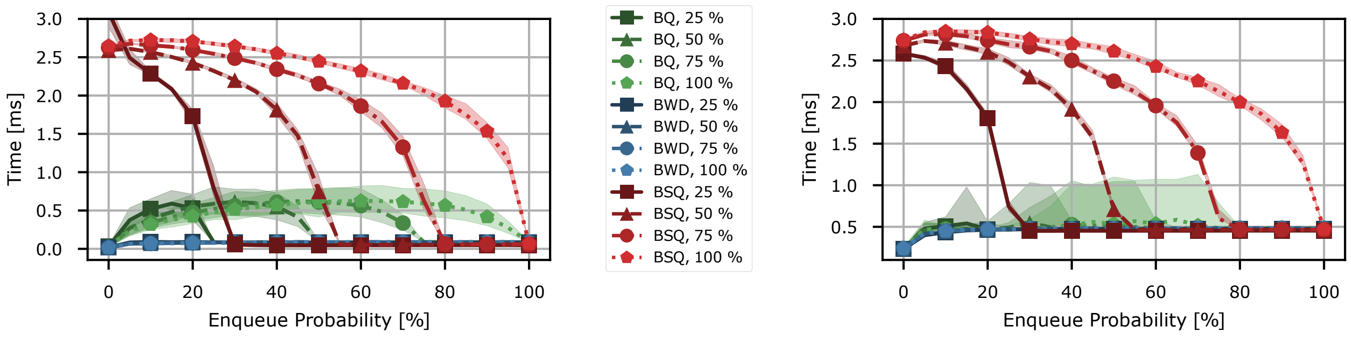
Figure 3: We consider imbalanced test cases, both synthetic and realistic. We test enqueue with probability $\mathbb{P}(enq)$ and dequeue with $\mathbb{P}(deq)$ on initially empty (a, b) and pre-filled queues (c). In (a), $\mathbb{P}(enq) > \mathbb{P}(deq)$ and the queues never run empty. Using a constant workload reduces the performance gap compared to the initial benchmark. In (b), queues quickly hit underflow, which has a devastating effect on the performance of LCRQ and WFQ, and, to a much lesser extent, on BQ. With a pre-filled queue (c), the performance drop is delayed by the time it takes to empty it. Testing queues in a real-world example to compute 8 iterations of page rank, we find that our algorithms (BQ, BWD and BSQ) are the fastest available techniques. For both medium-sized networks (d, 60k nodes) and large ones (e, 300k nodes), our queues achieve lower runtimes than simpler alternatives (GQ).

7.2 Imbalanced and real-world scenarios

In order to be useful in actual work scheduling systems, queuing algorithms must be able to efficiently handle cases where each task produces a certain amount of work, the number of enqueues and dequeues is not balanced, and queues can actually run empty.

Synthetic benchmark. To produce imbalanced scenarios, we first extend our initial test case such that every thread randomly performs between 1 and 10 enqueue/dequeue pairs. We call enqueue and dequeue themselves with reduced probability. Consequently, the number of enqueue and dequeue operations is no longer balanced, which introduces the possibility of underflow. Furthermore, we simulate a workload for each task by executing 128 fused multiply-add (FMA) instructions after each successful dequeue. Since these scenarios require threads to recover from underflow in order to finish the test, they cannot be evaluated for the fully-blocking SFQ. Also note that we avoid overflow in this test by allocating sufficient memory for all queues. LCRQ cannot handle underflow well, but rather reacts to it by allocating and initializing new ringbuffers. In order to mask this dependency on dynamic memory management, we pre-allocate and initialize a memory chunk 32× the size of the other queues to provide LCRQ with sufficient resources.

We show our results for imbalanced test cases in Figure 3. With simulated workload added, the differences across techniques diminish for the default behavior. This can be observed in Figure 3a, where a 2× higher probability of enqueue than dequeue ensures that every thread can perform its task without delay, given that overflow does not occur. In the opposite case—probability of dequeue exceeds that of enqueue 2×—underflow occurs, and performance figures change considerably (Figure 3b). Since there is never a substantial amount of work to steal, BSQ keeps unsuccessfully checking other queues, and its overhead is never amortized. The non-blocking techniques LCRQ and WFQ are at least 2.5× slower than GQ, BQ and BWD. Note that both approaches behave destructively, as queue slots can become unusable if a dequeue arrives there before an enqueue. In contrast to LCRQ, WFQ counteracts slot thrashing with its slow-path/fast-path dynamic, by turning unsuccessful dequeue threads into enqueue helpers. This is reflected by its runtime rising ~6× slower than LCRQ at underflow. However, BQ significantly outperforms both approaches and



(a) Runtimes with no workload **(b) Runtimes with 320 FMAs workload**

Figure 4: Detailed performance comparison of BQ, BWD and BSQ at different enqueue/dequeue ratios shows that BSQ in general reduces contention in ideal cases, but suffers from massive overhead otherwise. (a) As BQ is hitting a potentially empty queue, it waits until the the state is observable, which reduces its performance, especially if that state is likely to change again. (b) This effect becomes less pronounced as the task workload (simulated by 320 FMAs after each dequeue operation) increases.

hence poses the fastest linearizable queue among those tested. The simpler, non-linearizable GQ achieves up to 64% faster runtimes than BQ, but is also prone to erroneously detecting empty states. In a real-world scheduling scenario—where the workload is not known beforehand—this may cause threads to quit prematurely, compromising performance and correctness. The fastest runtimes are reported for our non-linearizable version of BQ, the BWD (10% faster than GQ at maximum occupancy). Compared to GQ, underflow detection by the BWD is less problematic, since it makes all interactions immediately visible via the *Count* variable. If the queues are initially pre-filled (Figure 3c), these trends still hold, but underflow and its effects on the queues are delayed accordingly: LCRQ, WFQ and BQ are affected by underflow only >25 600 threads, with similar performance for LCRQ and BQ at lower thread counts.

Page Rank. In order to provide a real-world example, we evaluate all competitive queues that are capable of handling underflow on the computation of page rank for two directed networks. Specifically, we compute the first 8 iterations for the data sets *p2p-Gnutella31* and *web-NotreDame*, provided by the Stanford Large Network Dataset Collection [19]. We pre-fill queues with one work item per node and launch a megakernel that tries to dequeue elements, until all threads agree that no more work is being generated. Active nodes pass on their latest available page rank value to their neighbors. If a node N finds that it is the last to contribute to the page rank of another node M in iteration i, M is enqueued for iteration $i + 1$. In order for LCRQ and WFQ to run fairly stable without immediately consuming all available memory, we had to shrink segments far below the recommended size (<128 slots per segment). Furthermore, we added traversal of previous *Head* pointers to LCRQ to reclaim abandoned segments. Performance measurements for tested queues are shown in Figures 3d and 3e. LCRQ/WFQ quickly fall behind, with slowdown of at least 60/6× over GQ, BQ and its variants in *p2p-gnutella31*, and 95/400× in *web-NotreDame* for more than 10 240 threads. In both networks, BQ outperforms GQ. This confirms our assumption that GQ's erroneous underflow detection is detrimental for tasks that only terminate when no more data is produced, which holds for the page rank test (in contrast to our synthetic tests). Consequently, BQ and BWD are consistently 10–15% faster than GQ for configurations >10 240 threads. Furthermore, we find that BSQ performs best for the large *web-NotreDame* network at maximum occupancy (9% over BWD). This is due to new work being generated in bursts when a neighborhood of nodes finish an iteration simultaneously, allowing for work stealing to take effect.

7.3 Broker Queue variants comparison

To investigate differences in behavior between BQ, BWD and BSQ in detail, we test various enqueue and dequeue probabilities under maximum occupancy. Figure 4a shows that, if enqueue probability is higher than dequeue, there is negligible difference in queue performance among the three approaches (<10%, thus within usual variance), with BSQ being marginally faster due to reduced contentions. However, at lower enqueue rates, the performance of BSQ suffers considerably (up to 30× slowdown). This is explained by its modus operandi: at maximum occupancy, a high number of thread blocks (and thus distributed queues) is employed. Hence, with few work items being generated at all times, work stealing constantly checks many queues, just to determine that they are all empty.

The largest difference between the BQ and BWD queues can be observed when dequeue happens about twice as often as enqueue. At this point, every other dequeue attempt observes a potential **Empty** state (BQ up to 5× slower). It is unlikely to observe an actual underflow of the queue, as there are still many enqueue operations happening, leading to multiple check-and-retry attempts. For lower enqueue probabilities, it is easier to observe **Empty** and thus the performance increases. On the other hand, for higher enqueue probabilities, it is more likely for a dequeue to immediately succeed, also increasing performance. Hence, ensuring linearizability of BQ can increase runtime by up to 20× if the queue is nearly empty/full all the time. However, already a small simulated workload (320 FMA operations) mitigates this effect (Figure 4b): under load, BSQ shows lower relative slowdown (~6×), and **Empty**/**Full** are likely matched by the pointers, as fewer threads access the queue concurrently. Hence, average performance of BQ and BWD is nearly identical.

8 CONCLUSION

In this paper, we presented new queuing strategies geared towards effective work distribution on the GPU: the broker queue, as well as two simple, general-purpose variants. Previous work in this field usually follows either of two strategies: relying on optimistic concurrency control and thus being non-blocking, or showing strict blocking behavior, even when the queue is full or empty. While the former shows poor scalability in massively parallel environments with thousands of threads, the latter prohibits effective scheduling mechanisms for work distribution on the GPU. Instead of following either strategy, we have combined the most desirable features of both, keeping the scalability of blocking queues, while ensuring versatility through non-blocking detection of under- or overflow.

Comparing to an extensive body of previous work, we found that our techniques consistently rank among the most competitive approaches. Since the broker queue was conceived with GPU hardware in mind, it does not rely on exotic or impractical hardware features, rendering its implementation straightforward. Our evaluation showed the broker queue to be the fastest linearizable queue for distributing work on the GPU in various scenarios. In balanced and realistic setups, the broker queue outperformed all previous algorithms. We also presented an even faster, non-linearizable variant of the broker queue, for the purpose of general work distribution: the broker work distributor. In terms of performance, the broker work distributor surpassed all previous approaches, even in synthetic imbalanced scenarios. Adding work stealing on top of our queue can ideally increase efficiency even further under realistic load. Although our proposed algorithms do not fulfill the non-blocking property, they are resilient to under- and overflow scenarios, making them prime candidates for work distribution on the GPU. In fact, our queues can be effectively applied in any scenario where a fast, concurrent queue is needed. Source code for our queues and evaluation is available at https://bitbucket.org/brokering/broker-queue. Finally, extending the idea used in the broker queue (*i.e.*, making the most relevant parts of an efficient blocking queue non-blocking), the design of a complete non-blocking queue with a highly efficient core appears to be a possible goal of future work.

ACKNOWLEDGMENTS

This research was supported by DFG grant STE 2565/1-1 and the Austrian Science Fund (FWF) I3007. The GPU for this research was donated by NVIDIA Corporation.

REFERENCES

[1] Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proc. High Performance Graphics 2009 (HPG '09)*. New York, NY, USA, 145–149.

[2] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 1998. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proc. ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*. New York, NY, USA, 119–129.

[3] Guy E. Blelloch, Perry Cheng, Phillip B. Gibbons, and P. B. Gibbons. 2003. Theory of Computing Systems Scalable Room Synchronizations.

[4] Daniel Cederman and Philippas Tsigas. 2008. On dynamic load balancing on graphics processors. In *Proc. ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware (GH '08)*. Aire-la-Ville, Switzerland, Switzerland, 57–64.

[5] Sanjay Chatterjee, Max Grossman, Alina Sbirlea, and Vivek Sarkar. 2011. Dynamic Task Parallelism with a GPU Work-Stealing Runtime System. In *Proc. Workshop on Languages and Compilers for Parallel Computing (LCPC '11)*.

[6] Long Chen, O. Villa, S. Krishnamoorthy, and G.R. Gao. 2010. Dynamic load balancing on single- and multi-GPU systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. 1–12.

[7] R. Colvin and L. Groves. 2005. Formal verification of an array-based nonblocking queue. In *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*. 507–516.

[8] John Giacomoni, Tipp Moseley, and Manish Vachharajani. 2008. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proc. ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP '08)*. New York, NY, USA, 43–52.

[9] Allan Gottlieb, Boris D. Lubachevsky, and Larry Rudolph. 1983. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *ACM Trans. Program. Lang. Syst.* 5, 2 (April 1983), 164–189.

[10] Mark Harris. 2014. Maxwell: The most advanced CUDA GPU ever made. (2014).

[11] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proc. ACM symposium on Parallelism in algorithms and architectures (SPAA '10)*. New York, NY, USA, 355–364.

[12] Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. 2006. A Dynamic-sized Nonblocking Work Stealing Deque. *Distrib. Comput.* 18, 3 (Feb. 2006), 189–207.

[13] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2003. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proc. International Conference on Distributed Computing Systems (ICDCS '03)*. Washington, DC, USA.

[14] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492.

[15] Moshe Hoffman, Ori Shalev, and Nir Shavit. 2007. The baskets queue. In *Proc. international conference on Principles of distributed systems (OPODIS'07)*. Berlin, Heidelberg, 401–414.

[16] H. T. Kung and John T. Robinson. 1981. On optimistic methods for concurrency control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213–226.

[17] Leslie Lamport. 1983. Specifying Concurrent Program Modules. *ACM Trans. Program. Lang. Syst.* 5, 2 (April 1983), 190–222.

[18] Patrick P. C. Lee, Tian Bu, and Girish Chandranmenon. 2009. A lock-free, cache-efficient shared ring buffer for multi-core architectures. In *Proc. ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '09)*. New York, NY, USA, 78–79.

[19] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).

[20] Maged M. Michael and Michael L. Scott. 1995. *Correction of a Memory Management Method for Lock-Free Data Structures.* Technical Report. Rochester, NY, USA.

[21] Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. ACM symposium on Principles of distributed computing (PODC '96)*. New York, NY, USA, 267–275.

[22] Adam Morrison and Yehuda Afek. 2013. Fast Concurrent Queues for x86 Processors. *SIGPLAN Not.* 48, 8 (Feb. 2013), 103–112.

[23] Nvidia. 2017. CUDA Programming guide. (2017).

[24] Daniel Orozco, Elkin Garcia, Rishi Khan, Kelly Livingston, and Guang R. Gao. 2012. Toward High-throughput Algorithms on Many-core Architectures. *ACM Trans. Archit. Code Optim.* 8, 4, Article 49 (Jan. 2012), 21 pages.

[25] Thomas R.W. Scogland and Wu-chun Feng. 2015. Design and Evaluation of Scalable Concurrent Queues for Many-Core Architectures. In *Proc. ACM/SPEC International Conference on Performance Engineering (ICPE '15)*. New York, NY, USA, 63–74.

[26] Chien-Hua Shann, T.-L. Huang, and Cheng Chen. 2000. A practical nonblocking queue algorithm using compare-and-swap. In *Parallel and Distributed Systems, 2000. Proceedings. Seventh International Conference on*. 470–475.

[27] Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. 2012. Softshell: dynamic scheduling on GPUs. *ACM Trans. Graph.* 31, 6, Article 161 (Nov. 2012), 11 pages.

[28] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. 2014. Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU. *ACM Trans. Graph.* 33, 6, Article 228 (Nov. 2014), 11 pages.

[29] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. 2012. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. In *Innovative Parallel Computing (InPar), 2012*. 1–10.

[30] John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12, 3 (2010), 66–73.

[31] Philippas Tsigas and Yi Zhang. 2001. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *Proc. ACM symposium on Parallel algorithms and architectures (SPAA '01)*. New York, NY, USA, 134–143.

[32] Stanley Tzeng, Anjul Patney, and John D. Owens. 2010. Task management for irregular-parallel workloads on the GPU. In *Proc. High Performance Graphics (HPG '10)*. Aire-la-Ville, Switzerland, Switzerland, 29–37.

[33] John D. Valois. 1994. Implementing Lock-Free Queues. In *Proc. International Conference on Parallel and Distributed Computing Systems*. 64–69.

[34] Chaoran Yang and John Mellor-Crummey. 2016. A Wait-free Queue As Fast As Fetch-and-add. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. New York, NY, USA, Article 16, 13 pages.