A Scalable Queue for Work Distribution on GPUs

Bernhard Kerbl Jörg Müller Michael Kenzel Dieter Schmalstieg Markus Steinberger
Graz University of Technology
{kerbl | joerg.mueller | kenzel | schmalstieg | steinberger}@icg.tugraz.at

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 present a new concurrent work queue, the *Broker Queue*, a highly efficient, linearizable queue for fine-granular work distribution on the GPU. We evaluate its usability and benefits in contrast to existing queuing algorithms. Our queue is up to one order of magnitude faster than non-blocking queues, and outperforms simpler queue designs that are unfit 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 Jörg Müller Michael Kenzel Dieter Schmalstieg Markus Steinberger. 2018. A Scalable Queue for Work Distribution on GPUs. In *PPoPP '18: PPoPP '18: 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3178487.3178526

1 Introduction

While the high processing power and programmability of a *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. As a means towards efficient execution of complex, task-based routines, previous work on this domain proposes and describes ways to implement task management for the GPU in software [4]. At the core of virtually all task management strategies are concurrent queues to distribute work in a first in, first out (FIFO) manner. The available literature on concurrent queues has a strong focus on lock-freedom, which is held as key to performance in concurrent systems. However, these algorithms are usually geared towards CPU architectures, ignoring the peculiarities of powerful and ubiquitous GPU hardware.

2 Massively Parallel Queuing on GPUs

As Hendler et al. already noted, the additional cost of redundant operations can potentially outweigh the benefits of true lock-freedom in a massively parallel environment [2]. To provide an adequate design in such a domain, we first analyze the requirements for an efficient work queue design on the GPU, before presenting our proposed algorithm. The underlying design in the majority of GPUs yields multiple ***programming and execution paradigms*** that an algorithm should support, including independent execution per-thread, per-warp, sub-warp execution, and cooperative block execution. As a general design choice, sticking to ***static memory only*** helps resolve potential points of contention caused by dynamic memory management of the GPU, which itself is very costly. To guarantee predictable behavior, a queue should further exhibit ***linearizability***, which ensures that the final state of the queue after executing temporally overlapping operations is the same as when executing said operations sequentially in a particular order. ***Multi-queue*** setups, which can be used to enable fundamental prioritization strategies, require the ability to probe queues for available workload. Based on these desired properties and features, we present a concurrent, linearizable queue, the *broker queue* (BQ), which shows the performance of a blocking queue, but can return the control to the scheduler if the queue is empty or full.

The Broker Queue The broker queue employs a ring buffer to directly store elements, a head and a tail pointer for ticketing, a ticket buffer that locks individual queue elements, and an explicit counter to weigh enqueue against dequeue operations. The ticketing itself assigns even-numbered tickets to enqueue operations and odd numbers to dequeue operations. The setup of these buffers, as well as the data structure interface, is given in Algorithm 1. Note that $\Lleftarrow$ indicates an atomic transaction, whereas $\Leftarrow$ is a non-atomic transaction; $\leftarrow$ stands for a simple local variable assignment. Usually, atomically operated head and tail pointers for ticketing prohibit a non-blocking reaction to full and empty conditions. For example, if there is one element in the queue and multiple threads increase the head pointer atomically, it is moved beyond the tail pointer. 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 the threads involved. Additionally, other threads could in the meanwhile enqueue new elements, validating some of the dequeues that were already rolled back. To avoid these issue, we introduce an additional counter variable (*Count*). It ensures that only

ALGORITHM 1: The Broker Queue

```
QueueElements RingBuffer[N]
unsigned int H ← 0, T ← 0, Tickets[N] ← {0, ··· , 0}
int Count ← 0
enqueue (Element)
    while not ensureEnqueue () do
        (head, tail) ⇚ (Head, Tail)
        if N < tail − head < N + MaxThreads/2 then
            return Full
    putData (Element)
    return Success
ensureEnqueue ()
    Num ⇚ Count
    while true do
        if Num ≥ N then
            return false
        if atomicAdd (Count,1) < N then
            return true
        Num ← atomicSub (Count,1) −1
putData (Element)
    LinearPos ← atomicAdd (T,1)
    Pos ← LinearPos % N
    waitForTicketNumber (Pos, 2 · (LinearPos/N))
    RingBuffer[Pos] ⇚ Element
    Tickets[Pos] ⇚ ← 2 · (LinearPos/N) + 1
dequeue ( )
    while not ensureDequeue () do
        (head, tail) ⇚ (Head, Tail)
        if N + MaxThreads/2 < tail − head − 1 then
            return Empty
    return readData ()
ensureDequeue ()
    Num ⇚ Count
    while true do
        if Num ≤ 0 then
            return false
        if atomicSub (Count,1) > 0 then
            return true
        Num ← atomicAdd (Count,1) +1
readData ( )
    LinearPos ← atomicAdd (H,1)
    Pos ← LinearPos % N
    waitForTicketNumber (Pos, 2 · (LinearPos/N) + 1))
    Element ⇚ RingBuffer[Pos]
    Tickets[Pos] ⇚ 2 · ((LinearPos + N)/N)
    return Element
waitForTicketNumber (Pos,ExpectedTicket)
    Ticket ⇚ Tickets[Pos]
    while Ticket ≠ ExpectedTicket do
        Ticket ⇚ Tickets[Pos]
```

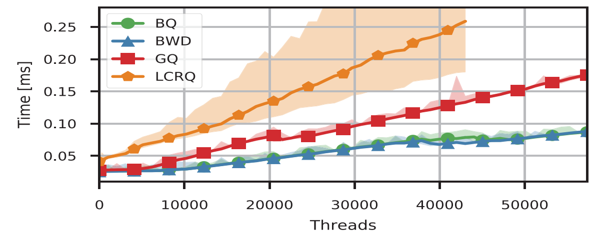

Figure 1. Microbenchmark comparing against relevant competitors. Both of our queues are faster than the alternatives.

threads which certainly will be able to enqueue or dequeue (and thus validly move head and tail) are allowed to interact with the pointers. For enqueue, this assurance is provided by `ensureEnqueue`, 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 committed to dequeue an element. Similarly, `ensureDequeue` returns **true** iff there is an element in the ring buffer for the thread to dequeue, or other threads already committed to enqueue an element. *Count* essentially models the relation between head and tail after all operations of concurrently active threads are completed. This mechanic is what we refer to as **brokering**.

3 Results

We compare our algorithm against the fastest lock-free algorithm to date, the Linked Concurrent Ring Queue (LCRQ) and the non-linearizable Gottlieb Queue (GQ) [1, 3]. Furthermore, we provide a simpler, non-linearizable version of our queue, the Broker Work Distribution (BWD) to evaluate the overhead of ensuring linearizability in our design. We run a microbenchmark with 10 iterations of enqueue followed by dequeue in each thread, for a varying number of concurrent threads. Recorded results are shown in Figure 1. Both algorithms outperform the alternative approaches. As proven by the obtained results, the overhead incurred by ensuring linearizability in BQ is negligible in this balanced scenario.

References

[1] 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. https://doi.org/10.1145/69624.357206

[2] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures (SPAA '10)*. ACM, New York, NY, USA, 355–364. https://doi.org/10.1145/1810479.1810540

[3] Adam Morrison and Yehuda Afek. 2013. Fast Concurrent Queues for x86 Processors. *SIGPLAN Not.* 48, 8 (Feb. 2013), 103–112. https://doi.org/10.1145/2517327.2442527

[4] 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. https://doi.org/10.1145/2661229.2661250