Design and Evaluation of Scalable Concurrent Queues for Many-Core Architectures [*]

Thomas R. W. Scogland
Department of Computer Science
Virginia Tech
tom.scogland@vt.edu

Wu-chun Feng
Department of Computer Science
Virginia Tech
feng@cs.vt.edu

ABSTRACT

As core counts increase and as heterogeneity becomes more common in parallel computing, we face the prospect of programming hundreds or even thousands of concurrent threads in a single shared-memory system. At these scales, even highly-efficient concurrent algorithms and data structures can become bottlenecks, unless they are designed from the ground up with throughput as their primary goal.

In this paper, we present three contributions: (1) a characterization of queue designs in terms of modern multi- and many-core architectures, (2) the design of a *high-throughput, linearizable, blocking, concurrent FIFO queue* for many-core architectures that avoids the bottlenecks and pitfalls common in modern queue designs, and (3) a thorough evaluation of concurrent queue throughput across CPU, GPU, and co-processor devices. Our evaluation shows that focusing on throughput, rather than progress guarantees, allows our queue to scale to as much as *three orders of magnitude* (1000×) *faster* than lock-free and combining queues on GPU platforms and *two times* (2×) *faster* on CPU devices. These results deliver critical insights into the design of data structures for highly concurrent systems: (1) progress guarantees do *not* guarantee scalability, and (2) allowing an algorithm to block can *increase* throughput.

1. INTRODUCTION

Multicore architectures have taken over the CPU market, and many-core accelerators and co-processors, such as GPUs and Intel Xeon Phi, are becoming available to all segments of computing. Each new generation contains more cores, further compounding the demands on the scalability of software. That scalability, more often than not, is governed by the cost of synchronization and communication.

Concurrent data structures have become basic building blocks for the new wave of highly parallel applications, providing intuitive abstractions atop the complexities of low-level synchronization and memory-coherence primitives. The result can be both increased productivity and, when designed well, performance. One of the most ubiquitous of these is the concurrent first-in, first-out (FIFO) queue.[1]

The concurrent queue has been studied extensively over the last four decades. It has gone through a variety of forms – from infinite-array queues [7, 4] to lock-free queues [11, 14] to advanced distributed lock-free [8, 5] or even wait-free [9, 10] variants. As concurrency has increased, so has the contention on concurrent queues and the cost of synchronization, and frequently serialization, in these designs.

Our goal with this paper is to characterize the performance requirements and considerations of concurrent queues in the multi- and many-core era and to create a concurrent queue that is tailored for high throughput, even under extreme contention. Our design and evaluation span CPU, GPU and co-processor architectures using the C, OpenCL, and OpenMP programming models. Specifically, this paper makes the following contributions:

1. A characterization of queue designs in terms of modern multi- and many-core architectures, demonstrating that allowing blocking in high-contention and low-oversubscription environments can improve throughput significantly over non-blocking designs.

2. The design of a linearizable and inspection-compatible, blocking FIFO queue based on the above characterization which is, to our knowledge, the first to combine these properties.

3. A thorough evaluation of our queue in OpenCL and OpenMP, including a comparison with several classic and state-of-the-art concurrent queues and demonstrating up to a 2-fold speedup on CPUs and as much as a 1000-fold speedup on GPUs running more than 1000 concurrent threads.

The rest of the paper is laid out as follows. Section 2 presents the background and setup for our work, including the machine abstraction that we employ to discuss synchronization and threading in OpenCL and OpenMP environments interchangeably. Related work follows in Section 3. Section 4 characterizes the bottleneck points in concurrent queue designs and models their performance in terms

[*]This work was supported in part by the AFOSR Basic Research Initiative (BRI) via Grant No. FA9550-12-1-044; the NSF Major Research Instrumentation (MRI) program via Grant No. CNS-0960081; and Lawrence Livermore National Laboratory (LLNL-CONF-664304).

ICPE'15, Jan. 31–Feb. 4, 2015, Austin, Texas, USA.
Copyright © 2015 ACM 978-1-4503-3248-4/15/01 ...$15.00.
http://dx.doi.org/10.1145/2668930.2688048 .

[1]As we only discuss FIFO queues in this paper, the term *queue* shall be used in place of *FIFO queue* henceforth.

of the atomic-operation throughput of many-core architectures. Section 5 presents the design of our queue and its three interfaces while Section 6 discusses linearizability [7]. Section 7 presents our experimental setup and benchmarks while Section 8 presents the results of our experiments. Section 9 presents concluding remarks and future work.

2. BACKGROUND

In order to discuss the properties of our target architectures in a uniform manner, we first present our abstraction of the concurrency and memory model that we use across devices. This section discusses the abstraction that we employ in this paper in order to discuss OpenMP on CPUs and OpenCL on CPUs, GPUs and co-processors all interchangeably, along with our microbenchmark evaluation of atomic operations that make this possible across each architecture.

2.1 Threading Abstraction

While the threading models of OpenMP and OpenCL are significantly different, they can be reconciled. An OpenCL kernel runs a set of work-groups, each consisting of work-items or, as they are sometimes unfortunately misnamed "threads." We exclusively use the term "work-item" to refer to these throughout this paper. Work-items are usually a single lane of a vector computation, rather than an independent thread of control. In OpenMP, there is no observable equivalent to the work-item, though a single iteration of a loop parallelized by an `omp simd` directive would be closest.

OpenCL does have an equivalent to the OpenMP thread however, but its interpretation changes from device to device. In NVIDIA GPUs, one thread is a "warp," composed of 32 work-items. In AMD GPUs, a thread is a "wavefront" of 32 or 64 work-items. When run on CPUs, work-items may be either operating system threads or individual lanes of vector calculations as on GPUs. For common CPUs, this means each thread may be composed of one to eight work-items. The width of the thread-equivalent used in a compiled kernel in OpenCL can be reliably determined based on the OpenCL 1.1 kernel work group info property "preferred work group size multiple," which is what we use in our implementations. To establish consistent terminology, we use the term *thread* to refer to *OpenMP threads* on the CPU and Xeon Phi or *independent groups of work-items* in OpenCL. Work-items within a thread must execute in lockstep. It is unsafe for more than one work-item in a thread to interact with a concurrent data structure simultaneously. When a thread accesses any queue in this paper, only one work-item is active.

The additional wrinkle is that OpenCL has no mechanism to get the number of threads that actually run concurrently. While a user can request any number of threads, the number that run concurrently can be anywhere from one to the requested number. We add counters as depicted in Figure 1 to all benchmarks to count the number of threads that exist before the first thread finishes execution, which is a reliable upper bound on the number of concurrent working threads regardless of the behavior of the OpenCL runtime.

2.2 Memory Model

CPU models like OpenMP depend on cache-coherent shared memory for correctness. The OpenCL standard does *not* provide a sufficiently strong coherence model or a memory flush that can be used to implement one. The standard states that "there are no guarantees of memory consistency between different work-groups executing a kernel [1]." Writes in different work-groups are only guaranteed to be synchronized at the end of a kernel, and are thus available in subsequent kernels. The standard specifically allows writes to global memory to *never* become visible to other work-groups within a single kernel.

```
void test(unsigned *num_threads, unsigned *present){
    if(atomic_read(num_threads) != 0)
        return;
    atomic_fetch_and_add(present,1);
    run_benchmark();
    atomic_compare_and_swap(num_threads, 0, atomic_read(present));
}
```

Figure 1: Design of concurrency detection in OpenCL benchmarks

The exception is atomic operations, available since OpenCL 1.1, which are guaranteed to be visible and coherent across work-groups within a kernel as long as all work-groups are executing on the same device. Thus, *every* write and *every* read to global memory that is shared between work-groups *must be atomic* to ensure correctness in OpenCL. In practice, some OpenCL devices support a more coherent memory model than this, but it is not required and several architectures do not. For example, NVIDIA GPUs present a weak coherence model, but offer a fence/flush through the PTX instruction *membar.gl*, but this is not standard OpenCL and must be used carefully. AMD GPUs have similar instructions at the ISA level but inline assembly only accepts the intermediate CAL language, which has no equivalent.

For consistency, we express all algorithms as a set of abstract atomically coherent instructions. In OpenCL, all operations are implemented with explicit atomic intrinsics, including load and store, to maintain coherence. In the CPU implementation, atomic reads and writes are standard load and store instructions, while fetch-and-add (FAA) and compare-and-swap (CAS) use the sequentially consistent memory ordering. The algorithm does not intrinsically require that the ordering be that strong, using the relaxed model on increments with matching acquire and release on reads and exchanges would be sufficient. We use the stronger consistency model because it is the default in OpenCL 2.0 and the only model exposed in OpenCL 1.2, which we used to implement our non-CPU device tests.

3. RELATED WORK

Concurrent queues have been studied for decades, nearly as long as computers with multiple computational units have existed to run them. We will elide some of the early history and refer the reader to the surveys provided by the papers referenced below, especially the Michael and Scott [11] survey, which provides significant discussion of early designs.

Array queues. The array queue proposed by Gottlieb et al. [4] in 1983 is notable for scaling near-linearly to 100 cores in simulation at the time. The Gottlieb queue can scale to as many threads as the hardware can run concurrently due to the use of a a pair of counters to select a location and fine-grained locking on each location in the queue. Unfortunately, however, the Gottlieb queue has been proven to be non-linearizable [2] due to the counters, which can cause the queue to appear empty or full spuriously. Orozco et al. [13] present two related array queues called the Circular Buffer Queue (CB-Queue) and the High-Throughput Queue (HT-

Queue). The CB-queue merges the Gottlieb queue's two counters per side into one and in so doing offers linearizability. In so doing however, the CB-queue loses the ability to detect and return full or empty states to the user. Further, the paper asserts that full and empty status cannot be determined for the CB-queue and provide only blocking enqueue and dequeue calls. Lacking inspection, a closed state, and a non-waiting interface, the CB-queue becomes impractical for any case where the queue may over- or under-flow, as either case may cause an irrecoverable deadlock. The proposed solution to the weaknesses of the CB-Queue is to use the HT-Queue, which regains the ability to detect full and empty by using the same flawed double-counter mechanism employed by the Gottlieb queue.

Contended-CAS queues. Michael and Scott [11] present a pair of unbounded linked-list queues, one lock-free (MS-queue hereafter) and one lock-based. The MS-queue offers a linearizable, lock-free queue using a portable single-word CAS operation and has become the standard unbounded lock-free queue. While this queue has no full state, due to its unbounded nature, it naturally detects empty as a function of the CAS operation, all queues of this and later types support these states without further work, unlike the array queues above. An alternative bounded variant has also been proposed by Tsigas and Zhang (TZ-queue) [14], which uses a slightly different mechanism but performs similarly due to its use of contended CAS for committing operations. Queues like these are also common components of relaxed queues [8, 5]. Relaxed queues reduce contention by relaxing the semantics of linearizability from a strict FIFO queue and spreading the operations across multiple underlying queues.

List of array queues. Morrison et al. [12] combine array and list queues to create the Linked Concurrent Ring Queue (LCRQ). The LCRQ retains lock-freedom while avoiding contended CAS operations in the common case, by using a FAA to select a target element like a blocking array queue might. Since the item selection method is inherently blocking, a dequeuer could get a location and then be forced to wait indefinitely on a slow enqueuer, the LCRQ maintains lock-freedom by allowing threads to skip operations that block for too long, introducing the need for retries. After a certain number of operations, or retries, the underlying concurrent ring queue (CRQ) is closed, requiring enqueuers to allocate and initialize new CRQs and then enqueue them into the LCRQ. The downsides to this approach are the reliance on a double-wide CAS (which while common in x86 is not widely available in mobile or many-core architectures) and the reliance on the potentially frequent and expensive allocation and initialization of new CRQs.

Combining. Hendler et al. [6] embrace the serial nature of lock-free designs and propose a queue that uses coarse-grain locking along with a request-and-assist model called the flat-combining (FC) queue. Since only one thread is actually accessing the queue at any given time, fulfilling requests from other threads, the synchronization overhead and cache coherence traffic are comparatively low. The downside is that the maximum throughput of the FC queue is the maximum throughput of a single thread, regardless of the number of accessors. Even so, the throughput limit is higher than with CAS queues like MS-queue, but it is still bounded to serial performance.

Finally, several of these queues have been evaluated on CUDA GPUs by Cederman et al. [3]. Out of a number of lock-based and two lock-free designs (i.e., MS-queue and TZ-queue), they conclude that for higher concurrency, the two lock-free queue designs are nearly always highest performing. The performance they observe for the MS and TZ queues is similar to that found in our results for the same number of workers on comparable GPUs.

4. QUEUE CHARACTERIZATION

Each queue type's throughput can be modeled in terms of the time each successful operation blocks other operations from succeeding. In essence, the throughput is the average number of times the critical section of each design can be executed per unit of time. In most queues however, there is no explicit critical section where a lock is acquired and released. Rather the critical section is the time spent in a successful atomic operation, or set of operations, required to complete an action on the queue. This section benchmarks the performance of basic atomic operations across CPUs, GPUs and Xeon Phi and then models the scalability and throughput of different queue types in terms of atomic operation throughput.

4.1 Atomic Performance

To understand the scaling behavior of current queues, we must first understand the scalability of atomic operations on modern architectures. We measure the throughput of each atomic operation on a contended memory location for each number of threads. This is accomplished with a set of microbenchmarks in OpenCL that execute each operation 1,000,000 times per thread, not work-item, and for each successful operation increment a 32-bit counter in a register. At the end of the test, each thread's count is written to a separate memory location in global memory to be summed on the host. The throughput is computed as the number of operations completed divided by the time taken to execute the test, not including data movement or host-side setup.

Figure 2 shows the results of our atomic benchmarks for the five atomic primitives that queue designs commonly rely on, especially the compare-and-swap (CAS) and fetch-and-add (FAA) instructions. The CAS test is further broken down into two components because it is the only atomic operation we tested that can fail. For CAS we present the number of operations that were *attempted* per unit time, and the number that actually *succeeded* as separate values. We do not include results for any atomic arithmetic or bitwise operations other than add, but they all perform similarly.

The scalability of the operations on each of the three CPU systems generally follows common knowledge, FAA is faster than successful CAS at high contention by as much as 10×. Neither operation scales well however, losing throughput with additional threads on both Intel systems and gaining only marginally on the AMD system. An unexpected result here however was the write/XCHG performance is higher than read and FAA for most thread counts on the AMD Opteron CPU. This is probably due to a difference in the way that the AMD CPUs handle memory invalidation in their coherence protocol, but a full analysis of the cause is beyond the scope of this paper. In the past, the higher cost of CAS has been considered acceptable in order to offer strong progress guarantees in concurrent algorithms, ensuring that at least one thread makes progress at any given time. The full cost of it was also limited by the compar-

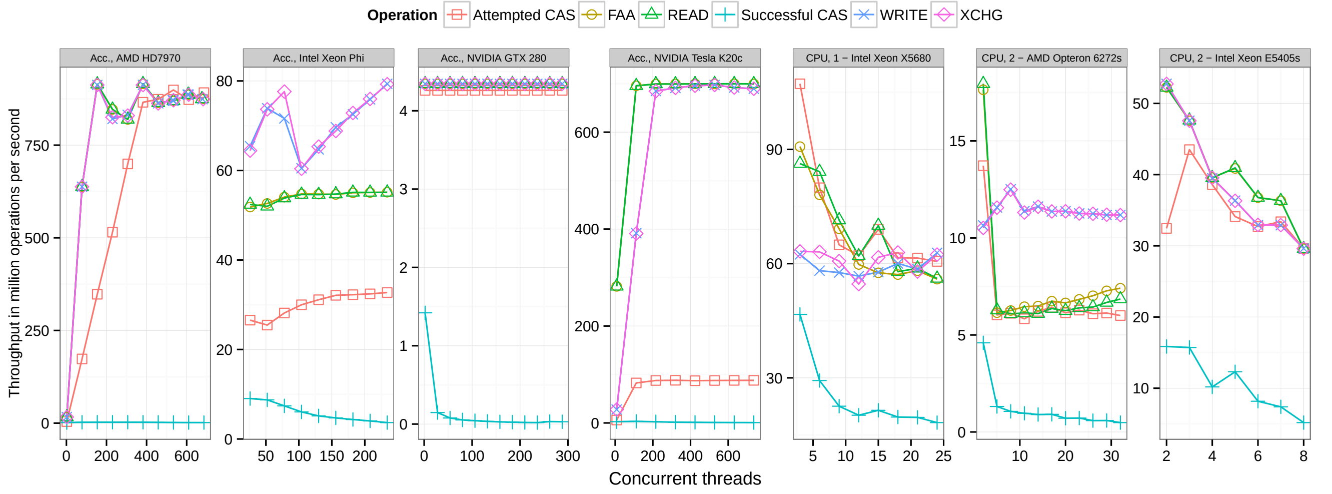


Figure 2: Contended atomic operation throughput, each thread executes its instruction 1,000,000 times, successful CAS represents only the CAS operations that succeed in updating the value

atively small number of threads that could be run concurrently on CPU systems.

On the GPUs and Intel's Xeon Phi FAA, read, write and exchange throughput scale up dramatically as more threads are added, and attempted CAS operations increase as well. Successful CAS throughput does not increase however, in fact it universally falls as the number of threads contending to update the single target value increases. At worst, the difference between successful CAS and FAA expands to 600× with 1000 threads on the AMD 7970. The stark difference between these results is due to the fact that operations such as FAA, read, write, and exchange can be executed completely in the memory controller without any chance of failure or retry. Contending operations are then simply queued there for later execution. On modern hardware, a single memory location can be incremented by FAA *once per cycle* in this fashion. CAS operations can also be handled at the memory controller, but if they fail they cannot retry without returning to the program first. As a result, every failure requires a full round-trip back to the processing core for *every failing thread*, and often an extra read or other logic, before another attempt.

4.2 Queue Throughput Modeling

To evaluate the expected performance of each type of queue across our target architectures, we construct an idealized model of maximum throughput for each type based on its critical operations. We will model the throughput, T, of each queue class in terms of the average latency of the atomic operations used to implement it. By modeling the queues in this way, we can extrapolate their expected scaling behavior as the number of available threads increase in terms of the scaling of their constituent atomics. To simplify the discussion, we treat the arrival distribution and service latency as deterministic and uniform and an arrival rate approaching the throughput. Fixing these allows us to deal directly with the service rate, or the per-operation latency.

Specifically we are going to discuss contended-CAS queues, like the Michael and Scott or Tsigas and Zhang queues; uncontended CAS queues, like the LCRQ or k-fifo queue; combining queues, like the FC-queue; and finally blocking FAA queues, like the classic array queue of Gottlieb et al. or the CB-queue. Each model is based on an idealized, minimum critical section for each type, and serves as an upper bound on the expected throughput. They do not account for effects of intermixing other operations on the performance of the critical section, which can have significant effects especially on cache-coherent devices. We leave these model extensions to future work.

Contended-CAS queues rely on a CAS operation on a head or tail value to update the queue. We base our model on the canonical MS-queue. The critical section for the MS-queue is the amount of time between a read of the head or pointer value and the completion of a CAS to update it, since the value must not have changed in the intervening period, or the CAS will fail. In addition to the read and CAS, an extra write is required to update the next pointer on an enqueue whereas an extra read to dereference the next pointer is required on dequeue. The resulting max throughput for a given number of threads t, which we will represent as T_t, is modeled in terms of the average latency of read, r_t, write, w_t, and successful contended-CAS, c_t, by Equation 1. In words, two operations, one enqueue and one dequeue, can complete after a period of three reads, one write and two contended-CAS operations.

$$T_t = \frac{2}{(r_t \times 2 + c_t) + (r_t + w_t + c_t)} \quad (1)$$

An un-contended-CAS queue behaves quite differently from a contended version. They tend to use an FAA instruction to either round-robin between queues, in the case of k-FIFO and similar, or to select a slot in the manner of an array queue, which they then update with CAS for safety but without the failure cost of contended-CAS queues. As a result, the maximum throughput is not dependent on successful CAS latency but rather on attempted CAS latency, since in the best case there are no CAS failures in this type of queue. The enqueue and dequeue are also more symmetrical in this case, since the CAS is used as both a reading and writing operation, and no pointer chasing is required in array-based variants. The resulting model, using C_t and a_t

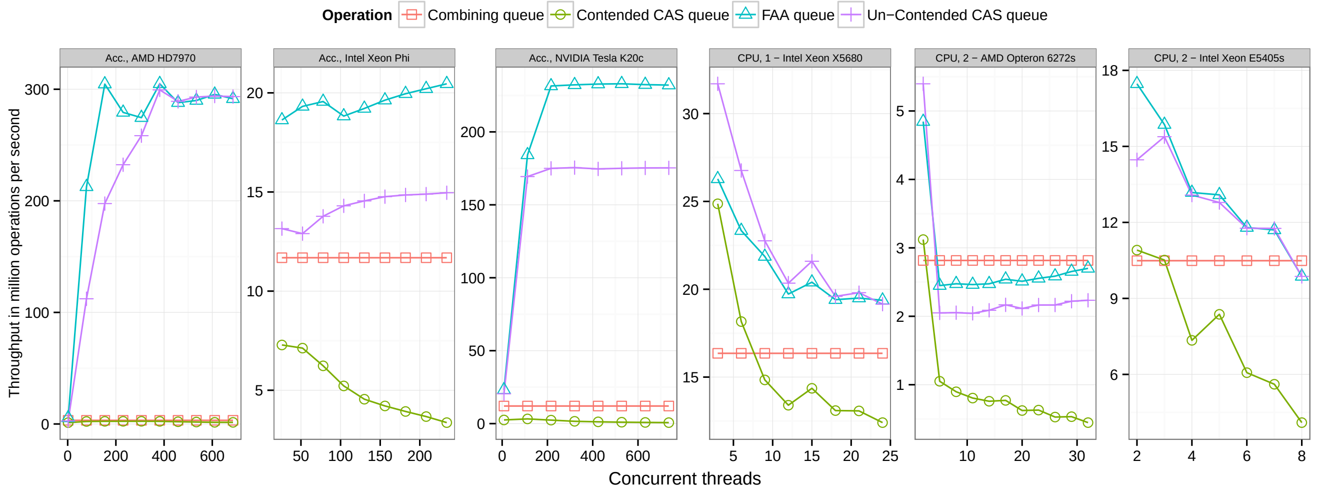


Figure 3: Predicted maximum throughput of each queue for each device and thread count

for average attempted CAS and FAA latency respectively, is presented in Equation 2.

$$T = \frac{1}{a + r + C} \tag{2}$$

The combining queue type is unique in that its throughput is dependent on the latency of reads and writes for *exactly one thread* rather than the throughput a given number of threads is capable of. By using a single thread to perform all operations on the queue at any given time, the maximum throughput at any level of contention is the same. At all numbers of threads we model the combining queue using the atomic latency for one thread in terms of Equation 3. This form is effectively the same as one would use for a serial queue, except that an additional read is performed to determine the operation to perform. This is followed by the operation itself, a read or a write into the queue, and a write to inform the requesting thread of completion.

$$T = \frac{2}{(r_1 + w_1 \times 2) + (r_1 \times 2 + w_1)} \tag{3}$$

Lastly, FAA queues are relatively similar to the un-contended-CAS type in behavior, except that instead of relying on a CAS for the final update they depend on an algorithmic guarantee that the target they receive from the initial FAA will eventually be valid for their use. As a result, the best-case performance is just an add, a read or write depending on the operation, and a write to update the target for the next thread.

$$T = \frac{2}{(a + r + w) + (a + w + w)} \tag{4}$$

The predicted maximum throughput for each design based on the models above and the atomic latency measured in the last section is presented in Figure 3. On the CPUs, it is clear why combining queues have come into favor, since the throughput of a single thread is universally better than that with threads on all cores contending on memory. The contended-CAS type starts as best on all three CPUs with one thread, but universally falls below as the number of threads increases. The un-contended CAS and FAA queue designs are the most promising for modern high-contention devices such as GPUs however. The prediction for the AMD 7970 and Tesla K20c GPUs show the FAA queue throughput reaching as much as 196 and 377 *times* faster respectively than the contended-CAS queue on the maximum number of threads. Our modeled results show that the bottleneck of contended-CAS-based synchronization becomes progressively more onerous as the number of threads increases, and incurs an unacceptable level of overhead on many-core devices. Despite the fact that the FAA queue shows the best scaling on high thread counts however, as mentioned in Section 3, the existing array or FAA queue variants all lack at least one desirable property of safety or usability.

5. DESIGN

The main goal of our design is to create a queue with as little overhead and as much concurrency as possible while maintaining linearizability and usability. Given the significant throughput advantages demonstrated in Section 4, our goal becomes to produce a linearizable and practical concurrent queue without the need for contended CAS operations in the common case. The CAS operations are normally used to provide a strong progress guarantee, at least non-blocking if not complete lock-freedom, that is highly valuable in environments where oversubscription is common and threads being scheduled out while in critical sections is a significant performance concern. Combining queues have come into being based on the fact that CPUs are less oversubscribed now than they were in the past, and many-core devices such as GPUs often cannot be oversubscribed at all, lacking a context-switching mechanism.

Because oversubscription has become relatively rare in many-core, we propose an array-based blocking queue that, unlike those existing in the literature, offers both inspection of state, in terms of full, empty and number of operations in-flight, and linearizability together. We further support safe interaction with the queue from threads that cannot block, or should deal with full/empty states, by providing two distinct interfaces to the same queue, similar to those offered by communication libraries (e.g., TCP sockets). Each interface has different waiting characteristics: (1) a high-throughput blocking interface and (2) a low-latency non-waiting interface.

5.1 The Queue Structure

Our queue's structure, represented in Figure 4a, is relatively simple, containing head and tail counters as unsigned integers along with arrays of items and IDs. For simplicity we use unsigned integers as the values, but extensions for arbitrary data are trivial.

In order to correctly handle integer rollover, a real concern as 2^{32} queue operations can complete in a matter of seconds on GPUs, we include a `#define` for the maximum id value based on the queue size. The maximum is selected such that when the head or tail roll over to zero, the id value will do the same. The `MAX_ID` value must be at least double the value of `MAX_THREADS+1` and the sum of `MAX_THREADS` and `QUEUE_SIZE`, the `MAX_DISTANCE`, must be less than half of the maximum value representable by the unsigned integer type storing head and tail. All values in the data structure should be initialized to zero.

5.2 The Blocking Interface

Our blocking functions are represented in Figure 4b.[2] Our blocking functions begin by acquiring a ticket for their current transaction by atomically incrementing their respective counters on line 4. A ticket serves to select both the target location, by being modded by `QUEUE_SIZE` on line 5, and the id for the transaction. The ID is the number of times the algorithm has passed through the entire queue's length, calculated by the `GET_ID()` macro, plus one in the dequeue case. Once the target id is equal to the transaction id the current thread effectively holds a lock on that element of the queue and leaves the loop. A value is then either added to or copied from the queue, as appropriate on line 12, and the id safely incremented to preserve consistency across rollover on line 13, which also frees the next transaction on that slot.

Subsequent, or concurrent, calls to the blocking interface receive unique target addresses and ID combinations without retries or waiting, thanks to the FAA. As long as the queue is not full, each enqueue can complete with a constant total of four atomic operations. When the target item is busy, it waits in the loop at line 7 checking for closed status and backing off as appropriate. This can be a truly blocking operation, yielding to the system to wait on a notification, but is implemented as a busy wait in our tests. The CPU version does use a scheduling yield in the backoff routine to allow other threads to proceed however. The ordering imposed by incrementing the target address at each request has the effect of also enforcing fair ordering and preventing individual starvation in this interface short of a thread dying inside a critical section or complete OS starvation of a thread.

Unlike other blocking array queues in the literature, we contend that a closed state is required for usability, and does not interfere with linearizability. Once all items that will exist have been completely enqueued and dequeued, there must be a way to inform the threads blocked in dequeue to exit. The `closed` value in the queue is used for this purpose. Setting `closed` to true invalidates the queue for all future operations. Closing causes any enqueue or dequeue that is waiting to reverse its ticket acquisition, allowing the status inspection interface to detect when all waiting threads have left, and return immediately with the status `CLOSED`. All subsequent calls also return `CLOSED` without attempting to update the queue. This is equivalent to closing a communication channel like a socket or file descriptor.

[2]Note that the CPU and OpenCL implementations of all of our interfaces are identical save for the addition of memory qualifiers in the OpenCL version. In fact, our evaluation uses a single source version of all queues for both CPU and OpenCL tests, simply compiled with different compilers.

5.3 The Non-Waiting Interface

Figure 4c presents our non-waiting interface. Fundamentally, the non-waiting functions are inverted versions of the blocking functions. Rather than immediately reserving a ticket, which could require them to block, the non-waiting functions simply read the value. Having read a ticket, they check whether the current target item is ready on line 7. If the id value indicates the item is busy then acquiring an item at this time would require blocking, so `BUSY` is returned immediately. If the id matches, the thread attempts to acquire the associated target by incrementing the counter with the CAS on line 9. If the CAS fails, the non-waiting function returns `BUSY`, otherwise it has successfully acquired a ready target so it completes its operation and returns `SUCCESS`.

While we implemented the blocking interface completely without CAS, to implement the non-waiting functions in that way is infeasible. Specifically, there is no way to atomically acquire a specific ticket without a conditional atomic or transaction, such as CAS, Load-Linked Store-Conditional, or optimally a compare-and-add. Using an unconditional FAA in place of the CAS would get a ticket, but with no guarantee that it would be the ticket which had been checked ahead of time. While this has the same performance consequences as the contended-CAS queues face, it has the advantage of interleaving with the blocking queue, allowing threads to attempt an operation on the queue safely, or a single thread using the interface persistently to watch for persistent full or empty states and act to remedy them. This cannot be done with other blocking queue designs that preserve linearizability, and is a key benefit to the overall design.

Note that we avoid the use of the terms "wait-free," "lock-free" and "non-blocking" in this section. While these functions do not wait, calls to either `enqueue_nb()` or `dequeue_nb()` will fail if another operation is in progress on the slot they request, or if the queue is full. In a non-full queue, the non-waiting interface does guarantee that at least one thread makes progress at a time, equivalent to the guarantees made by other array-based queues such as the tz-queue.

5.4 The Status Inspection Interface

Much like the CB-queue, our blocking interface does not support returning "full" or "empty" states directly from the enqueue or dequeue functions. While they are not required for a correct concurrent queue, these states are often used to simplify the detection of completion in a concurrent algorithm, and as such are missed when they are unavailable. Rather than re-designing the algorithm to address this weakness however, we design a separate interface that provides checks for these states as well as the number of waiting threads on either end of the queue. The main difficulty in implementing this functionality is that head and tail can *not* be directly compared. There is a distinct chance that one, but not the other, has rolled over causing the less-than or greater-than relationships to be reversed.

```c
/* Defines */
#define MAX_ID (UINT32_MAX/(QUEUE_SIZE*2))
/* Maximum possible distance between head and
 * tail for the status inspection interface */
#define MAX_DISTANCE (QUEUE_SIZE \
                      + MAX_THREADS)

/* Macros */
#define GET_ID(X) ((X / QUEUE_SIZE) * 2)
#define INC_SAFE(Q, T, ID) atomic_write( \
                            &(Q)->ids[T],\
                            ((ID)+1) % MAX_ID)

/* Structures */
typedef struct {
  union {//anonymous pair, allows inspection
    uint64_t combined;
    struct {
      uint32_t head, tail;
    };
  };
  bool closed;
  uint32_t items[QUEUE_SIZE];
  uint32_t ids[QUEUE_SIZE];
} queue_t;
```

(a) Definitions and Structure

```c
int enqueue(queue_t *q, uint32_t item) {
  if (atomic_read(&q->closed) != 0)
    return CLOSED;
  uint32_t ticket = atomic_add(&q->tail,1);
  uint32_t target = ticket % QUEUE_SIZE;
  uint32_t id = GET_ID(ticket);
  while(atomic_read(&q->ids[target])!=id){
    if (atomic_read(&q->closed) != 0){
      atomic_sub(&q->tail,1);
      return CLOSED; }
    backoff(); }
  atomic_write(&q->items[target], item);
  INC_SAFE(q, target, id);
  return SUCCESS;
}
```

```c
int dequeue(queue_t *q, uint32_t * p) {
  if (atomic_read(&q->closed) != 0)
    return CLOSED;
  uint32_t ticket = atomic_add(&q->head,1);
  uint32_t target = ticket % QUEUE_SIZE;
  uint32_t id = GET_ID(ticket) + 1;
  while(atomic_read(&q->ids[target])!=id){
    if (atomic_read(&q->closed) != 0){
      atomic_sub(&q->head,1);
      return CLOSED; }
    backoff(); }
  *p = atomic_read(&q->items[target]);
  INC_SAFE(q, target, id);
  return SUCCESS;
}
```

(b) Blocking Interface

```c
int enqueue_nb(queue_t *q, uint32_t item) {
  if(atomic_read(q->closed) != 0)
    return CLOSED;
  uint32_t ticket = atomic_read(&q->tail);
  uint32_t target = ticket % QUEUE_SIZE;
  uint32_t id = GET_ID(ticket);
  if(atomic_read(&q->ids[target]) != id)
    return BUSY;//next slot not ready
  if(atomic_cas(q->tail,
                ticket, ticket+1) != ticket)
    return BUSY;//CAS failed, return
  atomic_write(q->items[target], item);
  INC_SAFE(q, target, id);
  return SUCCESS;//element enqueued
}
```

```c
int dequeue_nb(queue_t *q, uint32_t * p) {
  if(atomic_read(q->closed) != 0)
    return CLOSED;
  uint32_t ticket = atomic_read(q->head);
  uint32_t target = ticket % QUEUE_SIZE;
  uint32_t id = GET_ID(ticket);
  if(atomic_read(&q->ids[target]) != id)
    return BUSY;//oldest not ready
  if(atomic_cas(&q->head,
                ticket, ticket+1) != ticket)
    return BUSY;//CAS failed, return
  *p = atomic_read(q->items[target]);
  INC_SAFE(q, target, id);
  return SUCCESS;//element dequeued
}
```

(c) Non-waiting Interface

Figure 4: Structure and interfaces to the queue with the volatile keyword removed for space; All "atomic_*" calls map to the corresponding atomic intrinsic

We address this case by establishing the maximum absolute distance possible between head and tail and checking to see if the current distance is greater than that maximum. If this happens, it must be because one counter has rolled over and the distance should be calculated across the rollover point. The maximum distance between head and tail in our queue is the sum of the queue length and the maximum number of threads that are allowed to interact with the queue concurrently. If the maximum distance is less than half the maximum value representable by the counter, single-counter rollover can be reliably detected in this fashion. For a 64-bit unsigned integer, the sum of the queue's length and the maximum concurrent accessors must be less than or equal to $2^{63} - 1$, which we believe is a reasonable limit. If more is required, the size of the counter should be increased.

6. LINEARIZABILITY

To provide a proof of linearizability, we must first define the semantics of our target data structure. Based on instruction ordering our algorithm models a concurrent FIFO queue. When return states, such as empty, full, closed and busy, are included in the requirements for linearizability however, our states do not match. We give our queue the semantics of a *channel queue*: a queue which models a double-sided communication channel, such as is presented by file descriptors and sockets, that can return success, closed, busy, empty or full. If a channel queue is in the closed state then all functions will return closed. If a non-waiting function cannot complete without blocking, busy is returned. All other cases model a concurrent FIFO queue, allowed only to return success, full or empty. In truth, this semantic is more common of concurrent queues in production than the traditional model's restriction to empty, full and success, and is modeled by the interface of the standard BlockingQueue class in Java as well as the interface to the concurrent WaitingQueue class proposed for inclusion in the C++1y standard.

Using the techniques and definitions presented by Herlihy et. al. [7], we model access to our concurrent queue as a history h. That history is a potentially infinite series of invocation and response events, representing the beginning and end of calls to functions defining our interface. Any response in h is necessarily preceded by a matching invocation in h but it is valid for an invocation in h to remain pending, lacking a response, at the end of h. Events are said to be ordered in h only if the response of an event e_1 precedes the invocation of an event e_2 and this relation is denoted by $e_1 <_h e_2$. Any pair of events that cannot be compared in this way is said to overlap, and thus may be ordered arbitrarily with respect to one another. The history, h, is linearizable if the strict partial order can be fixed into a total order $\rightarrow_h$ such that the specifications of the object are preserved.

Any history that can be produced by our implementation can be associated with a history mapped onto an auxiliary array of infinite size. Using this auxiliary array, our algorithm guarantees that every enqueue, blocking or non-waiting, will monotonically increase the values of the tail counter and thus insert elements consecutively into the infinite array. In the same way, our dequeues monotonically increase the value of head and consume elements consecutively. Thus all items are dequeued in the same order they were enqueued, or are overlapped. Any element that is added is accounted for, and cannot be removed until it is acquired. Acquisition can only happen in order, preventing any items from being skipped or dequeued before being enqueued. All interleaving between the blocking and non-waiting interface are also in-order, as they acquire and interact with the queue using the same ticket and turn mechanism.

Given these, the only source of non-linearizable behavior possible is from multiple operations waiting on the same target item with the same id. Given our invariant that the

`MAX_ID` is greater than double `MAX_THREADS`, this case cannot occur, as ids are not recycled until the queue has been cycled through completely at least `MAX_ID` times. Given that invariant, even if a queue of length one were to have the maximum number of threads waiting on it, both ordering and fairness are preserved between those accesses.

The above sketches a proof of the key invariant for concurrent queues, that if enqueue(x) < enqueue(y), where x and y are the values enqueued, then dequeue(x) < dequeue(y) or dequeue(x) and dequeue(y) overlap. To simplify reasoning about the ordering, our functions "take effect" at specific points between their invocation and response, but as with any algorithm employing critical sections there is no single instruction that serves as the universal linearization point. Operations are considered to take effect on the status of the queue, observable only through the `get_distance()` function and its siblings, after committing to the addition or removal of an element by acquiring a ticket with either FAA or incrementing CAS. All enqueue and dequeue operations are ordered in the sequential history by their increment of the id associated with their target item. Any temporary discrepancy in queue structure between invocation and response is protected by the critical region formed between ticket acquisition and id increment.

7. EXPERIMENTAL SETUP

In order to perform our evaluation across a wide range of modern hardware, we have created a version of each queue using both OpenMP and OpenCL. This section will discuss the evaluated queues, our benchmark designs and the hardware evaluated.

7.1 The Queues

In addition to our own, we include implementations of two traditional lock-free queues, the TZ and MS queues, the Flat Combining (FC) queue, and the LCRQ. All queues are implemented to store 32-bit unsigned integers and where memory allocation would normally be necessary use a non-blocking concurrent free-list of appropriately-sized objects that is pre-allocated before each test. The same free-list mechanism is used on both CPUs and GPUs for consistency, and nearly eliminates de-allocation cost but still incurs initialization cost.

Our MS-queue implementation is directly derived from the source code used in the original MS-queue publication [11]. It has been modified minimally to support thread-based rather than process-based parallelism and the memory model presented by OpenCL. The TZ-queue has been faithfully re-implemented using the algorithm and optimizations described in the paper proposing it [14]. The flat combining queue is based on the authors source but reimplemented in C/OpenCL from the original C++. Lastly, the LCRQ is based on the pseudocode in the publication proposing it [12][3]. Our LCRQ implementation deviates in two key ways from the original pseudocode, it includes the spin waiting optimization proposed in the paper, and uses 32 rather than 64 bit values. The value size is changed to allow the algorithm to function on devices that support 64-bit but not 128-bit CAS operations. We evaluated the 32-bit version against a 64-bit version of the algorithm and found that the throughput remains within the range of measurement error for all cases.

The OpenCL and OpenMP implementations of each queue share the same source, with only memory location qualifiers, atomics and memory synchronization primitives differentiated through C macros. For all fixed-length queues, the queue length was set at 65,536 elements for the purpose of our evaluation, separate tests with varied sizes did not reveal significant correlation with performance except when using very small sizes, so these results are elided.

7.2 Benchmarks and Methodology

These queue implementations are evaluated across a pair of microbenchmarks designed to measure queue throughput. We define throughput for our evaluation as the number of enqueue and dequeue operations successfully completed per second, or queue operations per second. The first benchmark is a traditional matching enqueue and dequeue benchmark, essentially a balanced producer consumer pattern. All threads execute a loop containing an enqueue, a call to some work, a dequeue, and another call to work. The work between each queue operation is comprised of 100 iterations of addition and multiplication on a value read from and stored back to positions in global memory determined by the value last received from the queue. This work is sufficient to avoid a single thread running through multiple operations without interference, and decreases the performance of the highest throughput implementations by approximately 10% compared to a version without work[4]. Our second benchmark is based on an imbalanced producer/consumer pattern. One in every four threads only enqueues, and the other three only dequeue, these operations are also separated by the same work as in the first benchmark. Both benchmarks are configured to perform as many operations as possible in five seconds and report the number of successful operations. We selected five seconds after running a round of tests ranging from two seconds to a minute and a half per data point and finding that anything over three seconds is sufficient to overcome variance effects across our target platforms.

The OpenMP implementation ends the test by creating an extra thread that sleeps and sets a *done* value, stopping the test after the specified time. OpenCL offers no such mechanism, neither the extra thread nor the sleep. To get around this, we assign one thread to execute a loop performing mathematical operations on its registers for approximately five seconds. Since the number of operations required changes based on the device, the test and sometimes the queue under test, as a result of register usage changes, our run-scripts automatically tune the number of iterations such that each test runs for between 4.95 and 5.5 seconds on all OpenCL platforms. The downside to this approach is that we lose one potential thread, but with throughputs that range up to three orders of magnitude, evaluation using a fixed time rather than a fixed number of operations is essential.

7.3 Devices

Table 1 lists the devices used to conduct our experiments,

[3]We did correct one error in the pseudocode, line 45 should compare (safe,idx,val) rather than (safe,h,val) as the original states, the text description in the original paper agrees with this modification.

[4]Some implementations, including LCRQ, perform *better* with the work than without it, as a result of reduced contention on the queue producing fewer CAS retries.

Device	Cores/ device	Threads/ core	Max. threads	Max. achieved
GPUs/Co-processors				
AMD HD5870	20	24	496	140
AMD HD7970	32	40	1280	386
AMD HD7990(one die)	32	40	1280	1020
Intel Xeon Phi P1750	61	4	244	244
NVIDIA GTX 280	30	32	960	960
NVIDIA Tesla C2070	14	32	448	448
NVIDIA Tesla K20c	13	64	832	832
CPUs				
2xAMD Opteron 6272s	16	1	32	32
4xAMD Opteron 6134s	8	1	32	32
2xIntel Xeon E5405s	4	1	8	8
Intel Xeon X5680	12	2	24	24
Intel Core i5-3400	4	1	4	4

Table 1: Target hardware platforms

along with their core counts, the number of thread contexts that can be loaded concurrently on each core, and the maximum hardware threads on the device. Note that the maximum threads listed in the table is the theoretical maximum, and in the case of GPUs is not always achievable due to limitations on available register space. The maximum achieved column lists the largest number of concurrent threads available to our tests, not all queues make it to those values but none make it above. All test systems run Debian Wheezy Linux on a 64-bit 3.2.0 stock kernel. NVIDIA devices use driver version 313.30 and the CUDA 5.0 SDK for OpenCL. AMD GPUs use the AMD APP SDK version 2.8 for OpenCL and the FGLRX version 9.1.11 driver. The Intel Xeon Phi card uses the MPSS gold update 3 driver and firmware. OpenMP tests were compiled with the Intel ICC compiler version 13.0.1 with optimization level 3 and inter-procedural-optimization turned on.

8. RESULTS AND DISCUSSION

We evaluate all queues across all hardware discussed above, with the exception of LCRQ on AMD GPUs because the AMD GPUs do not support bitfields or 64-bit atomic CAS. Since our queue presents two interfaces, we present three different configurations for it. Each is labeled in the figures as “New -” followed by which enqueue and dequeue functions it uses for all enqueues and dequeues in the test. The three configurations are the two homogeneous configurations, paired sets of blocking or non-waiting interface calls, plus a version using the non-waiting dequeue with the blocking enqueue. We expect that the most common use-case would be using the blocking interface for all but one, or perhaps a small number, of threads using the non-waiting interface to detect algorithm completion, which is best represented by the blocking results.

8.1 CPU Performance

The CPU results based on these tests can be found in Figure 5. Each CPU is tested from two threads up to the maximum number of hardware threads supported by the system. In multi-socket systems threads are spread in round-robin fashion across dies using the Intel OpenMP “scatter” affinity policy. While the multi-socket systems tend to maintain or lose throughput as threads are added, the single socket Intel Xeon X5680 gains throughput with each additional thread. This due to the fact that the single CPU only has one memory controller, allowing atomics to be completed without out-of-die coherence overhead. The AMD Opteron 6272 results are also notable for having better performance in practice than the predicted maximums from Section 4. Since our predictions are based entirely on 100% contentious atomic throughput, our models evidently under-predict for platforms that gain higher throughput of atomic operations when contention is lower.

In terms of the individual queues, in almost all cases the highest throughput comes from our blocking interface, followed by the LCRQ. The TZ and MS queues fare poorly in general across each of the CPUs, their performance degrading with each additional thread due to the increasing CAS retry overhead. On the AMD devices and the Xeon X5680, the FC-queue performs materially better than the classic lock-free variants for the matching enqueue/dequeue benchmark. The FC-queue even gains performance with additional threads on the AMD devices thanks to its comparatively low coherence overhead.

LCRQ’s performance on the AMD systems reveals an important characteristic of its design. In the matching enqueue/dequeue test it scales well, performing nearly as well as our blocking interface up to 32-cores. The producer/-consumer benchmark, on the other hand, shows LCRQ’s performance degrading sharply as more threads are added. This is due to retries and memory initialization overhead caused, not by CAS, but by LCRQ operations skipping slots by marking them unsafe. Whenever an operation times out, as is common in our imbalanced producer/consumer benchmark, the item reserved by that operation is marked unsafe, and it retries potentially marking many more unsafe along the way. This also means that the matching operation on that item must retry. Eventually, the retries cascade into the closing of the CRQ as a whole, forcing initialization of a new CRQ by all threads attempting to enqueue at that time. We employ the optimizations proposed to minimize this behavior, specifically spin waiting before marking a slot unsafe and employing a high starvation cutoff for enqueues, but still observe the problem. The Intel X5680 does not observe this behavior because of those optimizations, but they are insufficient for the multi-socket systems. This condition could be avoided in LCRQ if it were allowed to wait indefinitely for a matching enqueue, but that would make it blocking, and can actually produce deadlocks in the algorithm, since dequeuers might not be aware of the need to move to a new CRQ.

8.2 Effects of Oversubscription

While we designed our queue with no oversubscription in mind, and for architectures where it is often not possible, it is still at least a potential reality in CPU systems. In order to evaluate the effect of oversubscription on throughput we tested all queues with thread counts from two to 128 on a four-core CPU in Figure 6. All queues include a thread yield as part of their back-off routine, immediately allowing another thread to be scheduled in place of the thread which is waiting.

As has been shown in other recent work [12], the FC-queue suffers greatly from oversubscription as a result of the combiner being scheduled out frequently. The lock-free queues, MS, TZ and LCRQ on the other hand perform quite well in this test, as expected since this is the environment they are designed for. Both the MS and LCRQ designs maintain their performance across the full range. On the other hand the TZ-queue and our non-waiting interface tend to perform better than either by between 10 and 75%.

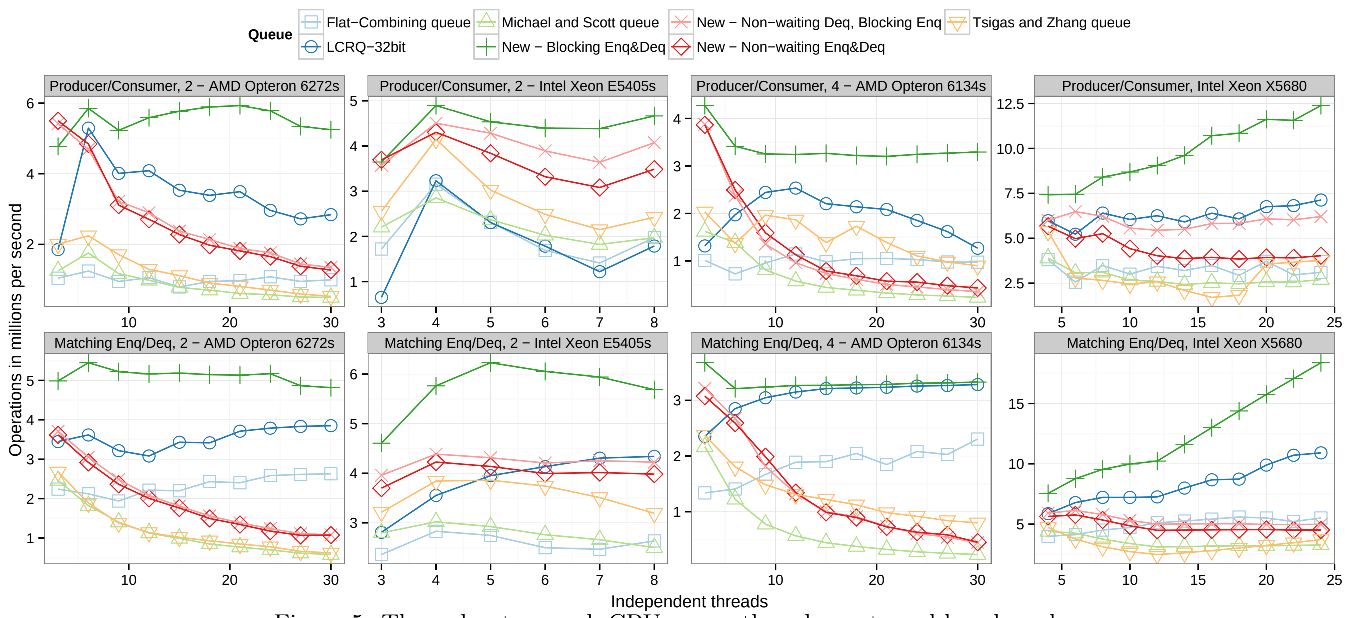


Figure 5: Throughput on each CPU across thread counts and benchmarks

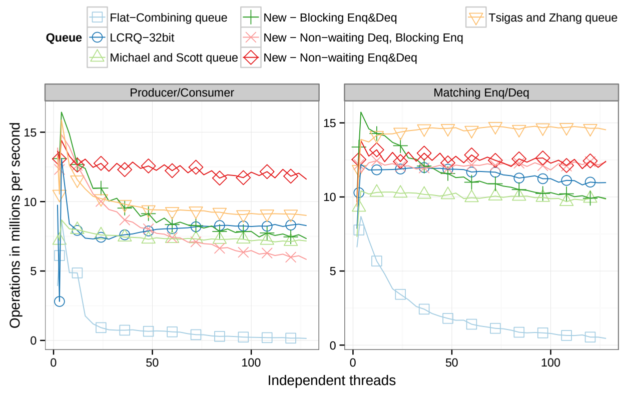


Figure 6: CPU performance when heavily oversubscribed, results of tests running from two to 128 threads on a four core Intel CPU

Finally the blocking interface does lose performance as more threads are added, but not so much as might be expected from a blocking design. Since the blocking is extremely fine-grained, and the potential concurrency extremely high, the blocking interface actually outperforms the MS-queue and maintains 50% of its maximum throughput with 32× more threads than hardware thread contexts.

8.3 Accelerator Performance

This section presents throughput results with the same benchmarks across seven many-core accelerator architectures in Figures 7a and 7b. Please note that unlike the CPU results, the range in performance on the accelerators requires us to use a log-scale for the bandwidth axis on our plots.

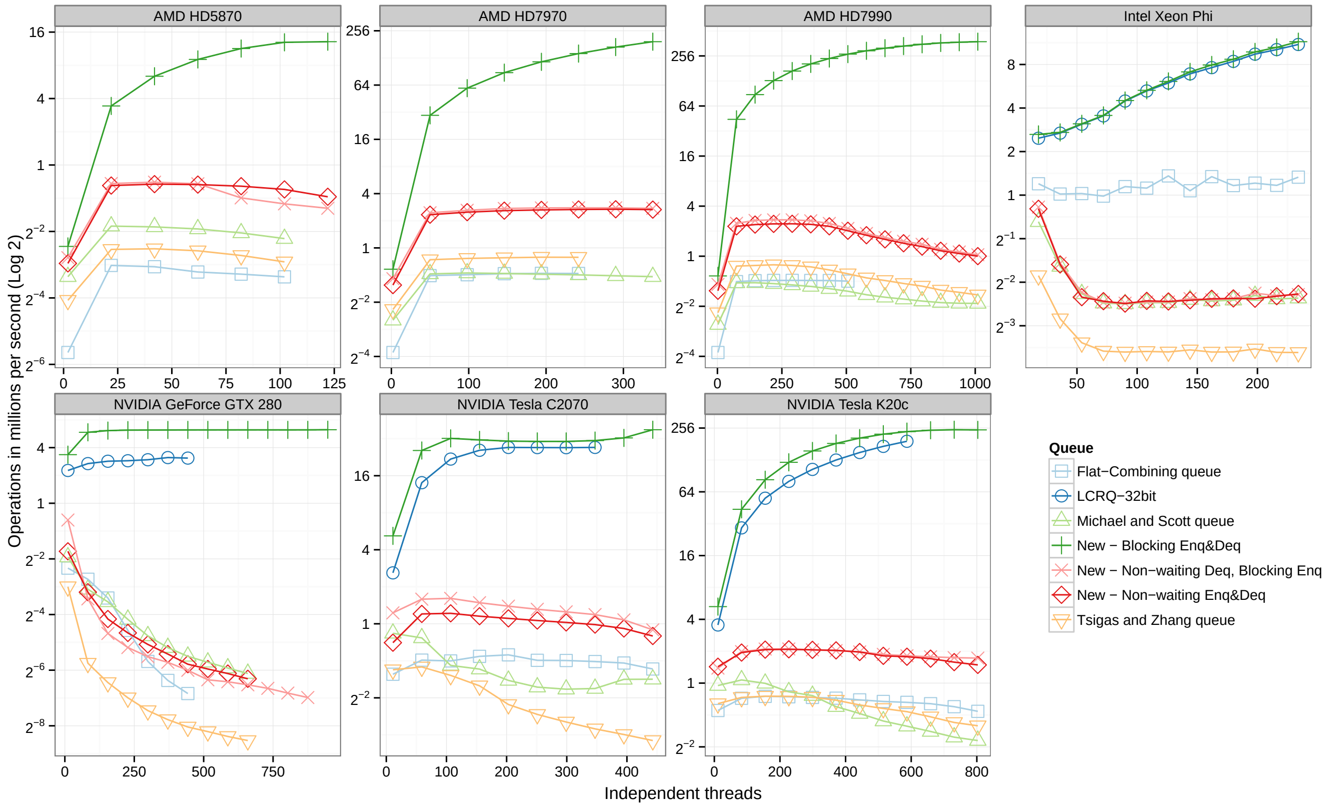
The first important difference between the accelerators and CPUs is the sheer number of thread contexts the accelerators support. Even the smallest, the AMD 5870, hosts 140 concurrent thread contexts for most benchmarks, more than four times as many as the CPUs. Recall that this is in threads, not OpenCL work-items, for the number of work-items multiply the threads on AMD GPUs by 64, and NVIDIA GPUs by 32 to get the full number. The two largest go far higher, with the 7990 reaching 1020 concurrently loaded threads, and the K20c hosting 832 for a total of 65,280 and 26,624 work-items respectively. The Phi device runs the OpenMP benchmark source from the CPU tests, so its 244 threads are standard OpenMP threads.

8.3.1 *Matching Enqueue/Dequeue Results*

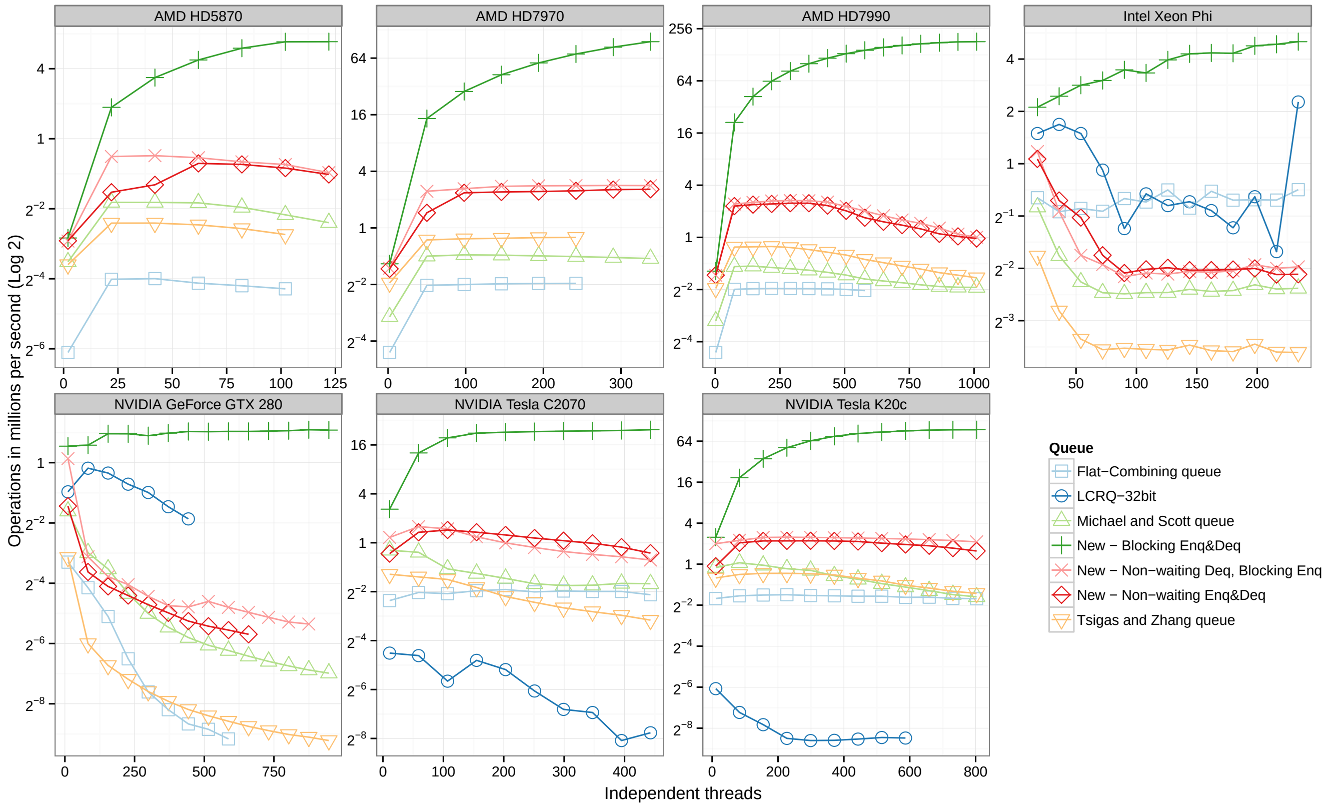
The enqueue/dequeue results on accelerators (Figure 7a) scale more like a single-socket CPU than a multi-socket system. Since the accelerator cores all share a single memory controller, this is expected. The material difference from the CPUs is that each additional thread increases performance noticeably for our blocking interface. On the 7990 the performance scales from 0.585 million operations per second on two threads to 380 million operations per second on 1019 threads. This makes for a 650× increase in throughput for a roughly 509× increase in the number of threads[5]. Similarly, the K20c attains 256× higher throughput with 415× more threads. The cache-coherent Intel Xeon Phi coprocessor scales somewhat less than the GPUs, going from 0.963 Mops/s to 11.462, for a more modest but still significant increase in throughput of 12× for roughly 120× more threads. The exceptions to the rule in terms of scalability are the GTX280 and C2070 NVIDIA GPUs, whose atomic implementations are less mature, and as a result only scale to a fraction of the throughput of the others.

By far the best performing lock-free design across the accelerators is the LCRQ. On the Xeon Phi its performance is nearly indistinguishable from that of our blocking interface. The NVIDIA implementations do not scale to the full number of threads due to LCRQ's high register usage, but for the thread-counts supported the throughput is quite high.

[5]The super-linear increase in throughput is not due to any super-linear property of the algorithm, but rather to the fact that the GPU tends to run in a lower performance state when under-utilized.



(a) Throughput on each accelerator for the weak-scaling matched enqueue/dequeue benchmark



(b) Throughput on each accelerator for the producer/consumer benchmark, of every four threads, one is producer the other three are consumers

Figure 7: Accelerator benchmark results

LCRQ's highest performance on the K20c, at 623 threads, is 201.848 Mops/s, only 16% below the throughput of our blocking interface with the same number of threads.

The contentious-CAS-based queues, FC-queue and our non-waiting interface, tend to lose performance as the number of threads increases and the rate of successful CAS operations drops. The fastest of these, the TZ and our non-waiting design, only achieve 0.342 and 0.994 Mops/s respectively on 1019 threads on the AMD 7990, or $1,112\times$ and $383\times$ lower respectively than the blocking interface in the same test. The K20c results are similar, with TZ performing 0.386 Mops/s and our non-waiting performing 1.357 for differences of $635\times$ and $181\times$ respectively. FC performs similarly to these traditional designs on the GPUs, but achieves $5\times$ better throughput on the cache-coherent Phi, where its cache friendly design offers material benefits. It is worth noting that, while it is not lock free, the non-waiting interface of our queue tends to outperform its counterparts in this space on these architectures, seemingly due to the lower number of instructions per operation.

8.3.2 Producer/Consumer Benchmark Results

Results for the producer/consumer test are presented in Figure 7b. As expected, the producer consumer test shows roughly 50% lower throughput across the board due to using 50% less enqueuers than dequeuers. All queues are affected by the imbalance roughly equally, except LCRQ.

The change in LCRQ results is most visible on the Xeon Phi, where rather than being nearly a match for the blocking interface, it drops to the performance of FC-queue after only 75 threads. Though LCRQ's throughput is variable, it never reaches half of the throughput of the blocking interface in this test on Xeon Phi. On the NVIDIA GPUs, LCRQ is now below the traditional lock-free designs and our non-waiting interface by a factor of 8. LCRQ's performance degrades with each added thread on the k20c, reaching a low of 0.003 Mops/s with 623 threads, where the next lowest, the FC-queue, is 0.322 Mops/s, and the blocking interface performs 91.803 Mops/s, *four orders of magnitude* higher throughput than LCRQ. It is quite apparent that applications of this nature, where consumers and producers are imbalanced, are pathologically bad for LCRQ. None of the other queues are materially affected by the imbalance.

9. CONCLUSIONS

In this paper, we present a characterization of concurrent queue designs across multi- and many-core architectures, and our design of a linearizable, inspection capable, high-throughput FIFO queue engineered for many-core architectures. Our characterization found that, largely due to the serialization caused by CAS operations, either an uncontended CAS design or a FAA-based array queue should scale best. Despite this, there are algorithms that are difficult or impractical to implement on a queue with only a blocking interface that does not allow detection of full or empty states. To address this limitation, our queue design includes both high-throughput blocking and low-latency non-waiting interfaces to customize interactions with the queue on a per-thread or per-interaction basis, both of which are linearizable to the semantics of a "channel queue," as well as a status inspection interface which can reveal full and empty states as well as how many blocking enqueues or dequeues exist. While queues with hard progress guarantees and unbounded size have their benefits, we have shown that focusing on throughput and avoiding retry-based algorithms can produce exceptionally high throughput across a wide range of real-world multi- and many-core hardware. Counter-intuitively, designing an algorithm that allows blocking to occur but increases the maximum concurrency of the structure results in greater throughput. In fact, our evaluation finds that performance can be improved by as much as *1000-fold* for some problems in an environment with more than 1000 concurrent threads.

In the future, we intend to investigate ways to create data structures of this type that are capable of offering some of the progress and safety guarantees of lock-free structures. Our queue might for example serve the purpose that the CRQ serves for the LCRQ data structure. An extension to support blocking, rather than spinning, thread waiting semantics could also be added by exchanging the id-based scheme for another. Further, we believe that this queue could be used to enhance a number of design patterns such as dynamic load-balancing and persistent threading on GPU and fused CPU/GPU architectures.

10. REFERENCES

[1] The OpenCL Specification. `https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`, Nov. 2012.

[2] G. E. Blelloch, P. Cheng, and P. B. Gibbons. Scalable Room Synchronizations. *Theory of Computing Systems*, 36(5):397–430, Aug. 2003.

[3] D. Cederman, B. Chatterjee, and P. Tsigas. Understanding the Performance of Concurrent Data Structures on Graphics Processors. *Euro-Par 2012 Parallel Processing*, 2012.

[4] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *Transactions on Programming Languages and Systems*, 5(2), Apr. 1983.

[5] A. Haas, M. Lippautz, T. A. Henzinger, H. Payer, A. Sokolova, C. M. Kirsch, and A. Sezgin. Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation. In *ACM International Conference on Computing Frontiers*, New York, New York, USA, 2013. ACM Press.

[6] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *ACM Symposium on Parallelism in Algorithms and Architectures*, June 2010.

[7] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[8] C. M. Kirsch, M. Lippautz, and H. Payer. Fast and scalable, lock-free k-FIFO queues. In *International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[9] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Symposium on Principles and Practice of Parallel Programming*, pages 223–234. ACM, 2011.

[10] A. Kogan and E. Petrank. A methodology for creating fast wait-free data structures. In *Symposium on Principles and Practice of Parallel Programming*. ACM, Feb. 2012.

[11] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *ACM Symposium on Principles of Distributed Computing*. ACM, May 1996.

[12] A. Morrison and Y. Afek. Fast concurrent queues for x86 processors. In *Symposium on Principles and Practice of Parallel Programming*. ACM, Feb. 2013.

[13] D. Orozco, E. Garcia, R. Khan, K. Livingston, and G. R. Gao. Toward high-throughput algorithms on many-core architectures. *ACM Transactions on Architecture and Code Optimization*, 8(4):1–21, Jan. 2012.

[14] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, July 2001.