AnyQ: An Evaluation Framework for Massively-Parallel Queue Algorithms

Michael Kenzel[*] [§], Stefan Lemme[*], Richard Membarth[*] [†],
Matthias Kurtenacker[*], Hugo Devillers[‡], Markus Steinberger[§], Philipp Slusallek[*] [‡]
[*] Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Saarland Informatics Campus,
{michael.kenzel,stefan.lemme,richard.membarth,matthias.kurtenacker,philipp.slusallek}@dfki.de
[†] Technische Hochschule Ingolstadt (THI), Research Institute AImotion Bavaria, richard.membarth@thi.de
[‡] Saarland University, Saarland Informatics Campus, {hugo.devillers,slusallek}@cg.uni-saarland.de
[§] Graz University of Technology, {kenzel,steinberger}@icg.tugraz.at

***Abstract*—Concurrent queue algorithms have been subject to extensive research. However, the target hardware and evaluation methodology on which the published results for any two given concurrent queue algorithms are based often share only minimal overlap. A meaningful comparison is, thus, exceedingly difficult. With the continuing trend towards more and more heterogeneous systems, it is becoming more and more important to not only evaluate and compare novel and existing queue algorithms across a wider range of target architectures, but to also be able to continuously re-evaluate queue algorithms in light of novel architectures and capabilities.**

To address this need, we present AnyQ, an evaluation framework for concurrent queue algorithms. We design a set of programming abstractions that enable the mapping of concurrent queue algorithms and benchmarks to a wide variety of target architectures. We demonstrate the effectiveness of these abstractions by showing that a queue algorithm expressed in a portable, high-level manner can achieve performance comparable to hand-crafted implementations. We design a system for testing and benchmarking queue algorithms. Using the developed framework, we investigate concurrent queue algorithm performance across a range of both CPU as well as GPU architectures. In hopes that it may serve the community as a starting point for building a common repository of concurrent queue algorithms as well as a base for future research, all code and data is made available as open source software at https://anydsl.github.io/anyq.

***Index Terms*—Massively Parallel, Concurrent Queue, GPU**

I. Introduction

Queues are an essential component of many concurrent systems. Whether it is for distributing tasks among processing nodes, to link producers and consumers in a processing pipeline, or as a general data structure in parallel algorithms; where there is concurrency, there are queues.

Due to their fundamental importance, concurrent queue algorithms have been subject of extensive research. However, the target hardware, evaluation methodology, and competing algorithms considered in the publications of any two given concurrent queue algorithms often share only minimal overlap, resulting in a lack of comparability of the published results. It is currently unclear, for example, whether the queue algorithm designed by Kerbl et al. [1] for the GPU should be expected to perform similarly well on current CPU architectures, or whether the algorithms by Morrison and Afek [2] or Yang and Mellor-Crummey [3] are still to be preferred. New hardware features such as independent thread scheduling (ITS) [4] might affect how these algorithms compare even on more recent GPU architectures. It remains also unclear how well any one of these algorithms should be expected to perform on, e.g., many ARM or RISC-V processors, where the performance characteristics of relevant atomic operations are different from the x86 architecture considered by most previous work.

In general, based on the available published data, we cannot conclude what the current state-of-the-art in concurrent queue algorithms looks like across an increasingly heterogeneous computing landscape. It is now more important than ever to be able to evaluate and compare new and existing queue algorithms across a wide range of target platforms. Given the rapid evolution of parallel processing hardware, it is of particular importance to also be able to continuously re-evaluate the body of existing algorithms in light of novel platforms, applications, and hardware features.

To address this need, we present an evaluation framework for concurrent queue algorithms, making the following contributions:

- We design a set of abstractions that allow for the efficient mapping of concurrent queue algorithms to a wide variety of target architectures.
- We demonstrate the effectiveness of these abstractions by showing that they can achieve performance comparable to hand-crafted implementations.
- Based on these abstractions, we design a system for testing and benchmarking concurrent queue algorithms.
- Using this system, we perform a set of experiments and present selected insights concerning the current state of the art in concurrent queue algorithms across multiple CPU as well as GPU architectures.

By making all our code and results available as open source software, we provide the community with a starting point for what we hope would become a common repository of concurrent queue algorithms and benchmarks. Such a repository could not only ensure continued evaluability of concurrent queue algorithms across current and future hardware architectures, but also significantly reduce the effort of conducting such comparisons for future research.

II. Related Work

At its core, a queue is a first in, first out (FIFO) data structure with two operations: *push* and *pop*. A push operation inserts a new element at the *tail* of the queue. A pop operation removes the next available element from the *head* of the queue. A *concurrent queue* allows push and pop operations to take place concurrently.

However, the very idea of a FIFO data structure inherently depends on operations being performed in some order, a concept that no longer simply applies in the face of concurrency. Different approaches exist to restore meaning and be able to reason about the behavior and correctness of a concurrent queue algorithm. The most common approach is to design the algorithm to be *linearizable* [5], i.e., such that all threads agree on one total order in which all operations logically take effect, even if their execution overlapped. Linearizability, however, can be costly to achieve, and many applications do not require such a strong guarantee in order to function. Therefore, formalisms with weaker properties have been developed and employed to increase performance [6], [7], [8], [1].

The purpose of a queue is to store elements for the duration from when they are pushed until they are popped. Concurrent queue algorithms broadly fall into one of two categories for how storage of queue elements is organized: A *bounded* queue operates within a buffer of a fixed, predefined size while an *unbounded* queue can dynamically grow and shrink within available memory. Bounded queues tend to be *array-based*, i.e., storing the queue contents in one contiguous array, while unbounded queues tend to be *link-based*, i.e., using some data structure that consists of nodes and links between nodes.

Queue operations can be either blocking, or non-blocking with obstruction-free, lock-free, or wait-free progress guarantees [9]. Basis of any concurrent queue algorithm is a set of atomic operations which are assumed as given and used to orchestrate inter-thread communication and synchronization. The choice of this set of atomic operations has profound implications for the performance and portability of a queue algorithm. Furthermore, applicability of a queue algorithm also depends on what type of element data it is able to store. Some queue algorithms rely on the ability to load and store queue element data atomically while other algorithms can store arbitrary data.

One of the first practical lock-free concurrent queue algorithms that found widespread adoption was described by Michael and Scott [10]. Like many other lock-free algorithms, their algorithm relies on an *optimistic concurrency control* [11] approach based on an atomic compare-and-swap (CAS) primitive. Queue operations are performed in two steps: the algorithm first prepares the follow-up state of the queue based on the last observed state and then attempts to transition the queue into this prepared state in a way that atomically either succeeds or fails. Downside of such an approach is that only one such operation can succeed at any point in time while any other concurrent attempts will fail and be forced to retry, which limits scalability.

More recently, algorithms like the lock-free LCRQ [2] and SCQ [12], and the wait-free algorithm by Yang and Mellor-Crummey [3] have started to take advantage of the fact that many processor architectures can natively perform atomic fetch-and-add (FAA) operations, which can be used to avoid retries of failed operations, resulting in significantly increased scalability and performance.

However, some widely-used architectures such as certain versions of ARM do not directly support atomic read-modify-write (RMW) operations like FAA. Instead, these architectures rely on a load-linked conditional store (LLCS) mechanism [13] where RMW atomics must be emulated by continuously retrying the operation until it succeeds, which potentially voids all of the benefits of relying on FAA over of CAS.

The design and evaluation of all the algorithms mentioned so far exclusively considered the x86 CPU architecture as target. On the GPU side, the Broker Queue [1] also relies on atomic FAA and has been shown to outperform other queue algorithms. However, their evaluation considered only NVIDIA GPUs.

Comprehensive evaluations that consider a variety of both CPU and GPU architectures are rare in the literature. One notable exception is the work by Scogland and Feng [14]. However, while their results were very valuable, they represent the state a few hardware generations ago and did not consider CPU architectures other than x86.

To the best of our knowledge, no previous work has considered the effect of vectorization on queue performance on the CPU, whether algorithms designed for the GPU might outperform classic CPU queue designs in such a setting, or how LLCS-based atomicity might shift the balance in favor of algorithms and design aspects that have otherwise received less attention. In contrast to previous work, the goal of this present work is not to design and evaluate a novel queue algorithm but to provide an update on the state of performance of concurrent queue algorithms on current CPU and GPU hardware as well as to design tools that will allow us to ensure continued evaluability for the body of work on concurrent queue algorithms going forward.

III. Framework Design

Prerequisite for being able to evaluate and compare concurrent queue algorithms across a wide range of architectures is the ability to map queue and benchmark implementations to the desired target hardware. Given our goal of continued evaluability, it is imperative that the compiler technology at the core of this mapping can be expected to be reasonably future-proof. Due to its widespread use and large industry backing, the de-facto standard choice for such technology is LLVM [15].

On top of LLVM, we rely on AnyDSL [16], a high-level programming framework with Rust-like syntax which can target CPUs as well as GPU compute platforms like CUDA [17], OpenCL [18], and HSA [19]. AnyDSL also provides mechanisms for explicit automatic vectorization based on the Region Vectorizer [20] to utilize single instruction, multiple data (SIMD) processing on CPUs as well as a runtime

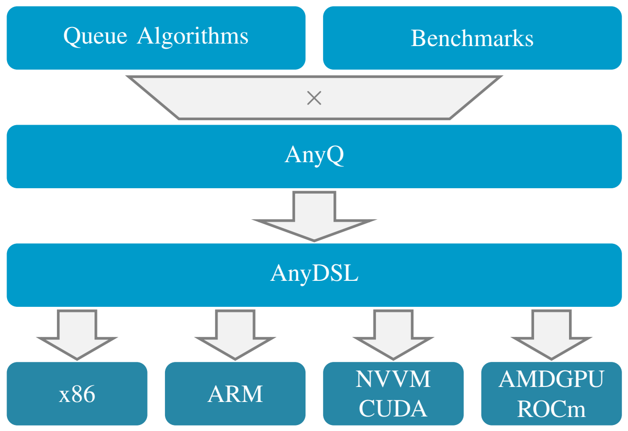


Figure 1: The AnyQ stack. Queue and benchmark implementations can be written independently using the abstractions provided by AnyQ in the form of an AnyDSL library. Any combination of queue and benchmark implementation can be compiled together with the AnyQ library through AnyDSL in order to obtain a fully-specialized benchmark executable for any of the supported target platforms.

layer for running the generated code on all supported platforms. Key feature of AnyDSL is its partial evaluation mechanism which allows us to generate specialized implementations for every combination of benchmark, queue, and target platform. Figure 1 illustrates the structure of this software stack.

The AnyQ evaluation framework consists of three major components: A set of concurrency abstractions that allow us to write parallel programs in a way that maps to any of the desired target platforms, a queue interface to decouple queue implementations from benchmarks, and a benchmark layer that provides facilities for writing benchmarks, instantiating benchmarks for given sets of queues and target platforms, and collecting performance measurements.

A. Concurrency Abstractions

To write queue benchmarks that can target both scalar and SIMD execution on CPUs as well as single instruction, multiple thread (SIMT) execution on GPUs, we need a set of abstractions that maps to all three execution models. We use a single program, multiple data (SPMD) approach similar to CUDA, OpenCL, or `ispc` [21]. Computation takes place in a *grid* of concurrent function invocations, each invocation representing a logical *thread*. These threads are grouped into *waves* that correspond to the natural unit of execution on the target device. On GPUs, these waves directly map to the warps or wavefronts of the native execution model. For vectorized execution on CPUs, waves are mapped to function invocations that are evaluated in parallel in different SIMD lanes. Scalar execution is modeled as a special case of vectorized execution with a wave size of one thread.

To represent these abstractions in code, we follow an approach similar to CUDA cooperative thread groups [17]. A `device` object represents a particular processor and offers a method to launch a grid of threads on that processor. The groupings of grid, waves, and threads are represented as *context* objects. Threads, that are part of the same group, have access to various operations in the form of methods on their context object. Table I gives an overview of context types as well as the operations they support. The memory model is based on the C++ memory model [22].

Table I: Types of thread groups as well as the operations each type of group context offers.

grid	split into waves/threads
wave	split into threads barrier (with reduction) shuffle
thread	atomic load/store, xchg, cas/cas_weak, atomic add/sub, inc, min/max, and/or/xor memory_barrier wait

A simple example program using these abstractions could look as follows:

```
for grid in device.launch(1024) {
  for wave in grid.waves() {
    for thread in wave.threads() {
      let i = wave.idx()/2 + thread.idx(0);

      if wave.idx() % 2 == 0 {
        data(i) = generate();
        thread.atomic_store(rdy(i), 1, release);
      } else {
        thread.wait(@||
          thread.atomic_load(rdy(i), acquire) == 1);
        process(data(i));
}}}}
```

We launch 1024 threads. All threads in even numbered waves generate data and then set a ready flag `rdy`. All threads in odd numbered waves wait until they see the ready flag and then read the corresponding data. We note that, due to the underlying SIMD execution, threads within the same wave generally cannot synchronize with one another except when running on architectures that support ITS. Support for ITS can be queried via a property on the given device object.

For every target platform, we provide a mapping layer that implements the primitives described above. For CPUs, we support both a scalar mapping as well as a vectorized mapping utilizing SSE/AVX on x86 and NEON on ARM. NVIDIA GPUs are supported via a CUDA and AMD GPUs via a HSA mapping using the NVPTX and AMDGPU LLVM backends. While scheduling of waves is handled automatically by the hardware on GPUs, execution on CPUs has to take place in threads managed by the operating system. We rely on Intel's TBB [23] library to schedule our waves on top of a thread pool utilizing all available CPU cores.

B. Queue Interface

A concurrent queue offers two fundamental operations: `push` and `pop`. Depending on the queue algorithm, each of these operations may be blocking or non-blocking, and may fail sporadically, or because the queue is full or empty. To be able to independently vary queue and benchmark implementations, an interface is needed that can accommodate any such concurrent queue design. To allow for a meaningful comparison of very different queue algorithms in various different benchmarks, it is important that this interface does not impose artificial restrictions that would prevent a given benchmark from making optimal use of a given queue.

The interface we have arrived at to satisfy these requirements is modeled parameterized on the queue element type `T` as

```
type source[T] = fn() -> T;
type sink[T] = fn(T) -> ();

struct ProducerConsumerQueue[T] {
    push: fn(source[T]) -> fn(ctx) -> i32,
    pop:  fn(sink[T]) -> fn(ctx) -> i32
}
```

where `ctx` is the context object associated with the thread performing the operation. Once more, due to the underlying SIMD execution, invoking queue operations from divergent control flow within the same wave can generally only be supported on devices that support ITS.

The `push` and `pop` operations are higher-order functions that take as parameters a source and sink function respectively. The source function is invoked by the queue implementation to generate the element value to be pushed into the queue. The sink function is invoked by the queue implementation to receive the value of a popped element. This inversion of control allows for code associated with the generation or processing of queue elements to be evaluated lazily. A queue implementation can decide to only invoke these source and sink functions once the algorithm has determined that it can indeed accept a new element or does have an element to dequeue, which allows for queue algorithms with such ability to be utilized appropriately. At the same time, client code is given the flexibility to decide which computations to potentially perform lazily inside or always non-lazily outside of the source and sink functions it passes to the queue.

Calling the `push` and `pop` functions yields itself a function that will perform the push or pop operation with the given source or sink when invoked. The result of calling this generated function is either `0` or `1` depending on whether the operation failed or succeeded. Client code uses a given queue assuming every operation is non-blocking and might fail. Queues that block or only fail in certain circumstances simply return constant `0` or `1` along the respective control flow paths. The partial evaluator in AnyDSL then ensures that the resulting code collapses to an optimal implementation for any given queue and benchmark combination.

To verify the correctness of queue implementations, we provide a set of tests that check invariants that must be preserved even by the most relaxed concurrent queue designs: pushed elements must eventually be returned by a pop, popped elements must have previously been pushed, and the number of successful push and pop operations must be consistent with the number of attempted and failed operations.

Listing 1: A simple queue benchmark program that exercises a given queue by attempting to push and pop elements `num_attempts` times. Note the use of the queue instrumentation wrapper provided by the framework to collect timings and queue operation statistics. AnyDSL's `for` construct provides an easy way to pass source and sink functions to queue operations.

```
let stats = create_queue_instrumentation(device);

for recorder in stats.collect() {
  for grid in device.launch(num_threads) {
    for wave in grid.waves() {
      for thread in wave.threads() {
        for n in range(0, num_attempts) {
          for q in recorder.record(thread, queue) {
            // queue operations in here are recorded

            for q.push(thread) {
              42  // return the value to push
            }

            wave.barrier();

            for value in q.pop(thread) {
              // discard popped value
            }

            wave.barrier();
}}}}}}

device.synchronize();  // wait for launch to finish

let results = stats.results();  // obtain results
```

C. Benchmark Layer

To facilitate the collection of benchmark data, we provide a queue wrapper that can be used to transparently add instrumentation on top of any given queue implementation that conforms to the queue interface. This queue instrumentation wrapper abstracts away the complexities of accurately measuring time on all the various target platforms as well as collecting and aggregating timings and statistics at the granularity of individual queue operations over the course of a benchmark run. A passthrough implementation of the queue instrumentation wrapper is also provided to allow for simple switching between using and not using instrumentation. Listing 1 shows an example of a benchmark implemented using our queue instrumentation wrapper.

The framework furthermore includes a build system that allows for the easy addition of queue and benchmark implementations, the generation of executables for combinations of queues, benchmarks, and target platforms, as well as scripts to automate the running of benchmark sets, and the processing and visualization of results derived from benchmark data.

D. Framework Evaluation

We use the simple benchmark shown in Listing 1 to compare the code generated by our framework for the Broker Work Distributor (BWD) [1] and Yang and Mellor-Crummey (YMCQ) [3] queue algorithms with the reference implementations published with each algorithm. The same benchmark implementation written in AnyQ is used with both

Table II: Run times (mean and standard deviation across 8 runs after 2 warmup runs) in ms for a simple benchmark comparing a BWD and YMCQ implementation using our AnyQ framework with the reference implementation provided by the authors of each respective algorithm. As can be seen, both versions show comparable, if not virtually identical performance.

algorithm	processor	implementation	run time	
			mean	stddev
BWD	GTX 1080	AnyQ	0.4120	0.0016
		reference	0.4156	0.0035
	RTX 2080 SUPER	AnyQ	0.0684	0.0137
		reference	0.0653	0.0127
YMCQ	Ryzen 9 5950X	AnyQ	1.7151	0.0554
		reference	1.6496	0.0593
	Apple M1	AnyQ	1.9936	0.2287
		reference	1.8929	0.2027

the AnyQ as well as the reference queue implementations. In order for the AnyQ benchmark to use a reference queue implementation, an adapter is used that implements the AnyQ queue interface and forwards all calls to the reference implementation. During partial evaluation, this adapter code is folded away and, thus, does not introduce any overhead.

When targeting CUDA, AnyDSL generates CUDA C++ code for the parts to be run on the GPU. The BWD reference implementation is itself originally written in CUDA C++ and can, thus, simply be included into the generated benchmark code. Similarly, on x86 and ARM, the YMCQ reference implementation written in C can simply be compiled and statically linked to the benchmark code.

The BWD benchmark was run with $100\,\mathrm{k}$ threads on both an NVIDIA GeForce GTX 1080 as well as GeForce RTX 2080 SUPER GPU with each thread performing 20 enqueue and dequeue attempts. The YMCQ benchmark was run via our scalar mapping on an AMD Ryzen 9 5950X x86 CPU with 16 threads as well as on an Apple M1 ARM CPU with 8 threads, each thread performing 1600 enqueue and dequeue attempts. The differences in thread counts and number of enqueue and dequeue attempts are to account for the differences in degree of parallelism as well as base overhead between the different systems and queue algorithms.

As can be seen in Table II, the AnyQ implementations perform almost exactly the same as the reference implementations on all devices, demonstrating that our programming abstractions enable a highly-performant implementation of queue algorithms on each of these platforms.

IV. Benchmark Setup and Methodology

To evaluate various queues across a range of different scenarios, we use a synthetic benchmark inspired by Kerbl et al. [1]. We launch a given number of threads that will act as concurrent producers and consumers. Each thread runs through a predefined number of iterations in each of which it randomly decides to produce an element, consume an element, or both.

Due to the SIMD nature of wave execution, we must avoid to perform enqueue and dequeue operations in parallel in threads within the same wave as this could result in a deadlock. To avoid this problem, we issue a wave barrier between enqueue and dequeue operations. To avoid introducing bias as a result of threads always attempting to enqueue before attempting to dequeue, we also randomize whether the wave as a whole will perform its enqueue or dequeue attempts first.

By varying the enqueue and dequeue probabilities, different classes of workloads can be simulated. An enqueue probability greater than the dequeue probability will result in queues operating in a closer to full state while the reverse configuration will operate queues in a closer to empty state. Equal probabilities model a balanced workload.

A per-thread 32-bit xorshift [24] generator (RNG) is used as a low-overhead source of randomness. Each benchmark is run a number of times with each thread going through an individual but consistent sequence of RNG seeds across runs. This ensures that each queue algorithm will be taken through the exact same reproducible regime of enqueue and dequeue operations while also avoiding artifacts that might arise as a result of correlations if each thread were to perform the exact same sequence of operations in every benchmark run.

A typical application will spend a certain amount of time performing computations and not interacting with the queue. To model this computational effort associated with the production and processing of queue elements, we introduce a simulated workload. Before each enqueue operation, the value to be enqueued is computed by advancing the RNG a predefined number of steps. Similarly, after each dequeue operation, the RNG is also advanced a predefined number of steps. Thus, we introduce a fixed amount of purely arithmetic work that can be precisely controlled, cannot be optimized away by the compiler, and does not put additional pressure onto the memory system.

By using synthetic benchmarks, we are able to analyze and compare queue performance under a certain usage pattern in isolation in an easily reproducible manner. The context of a complete application would bring with it many complicating factors like memory and compute loads unrelated to queue operation, resulting in complex and hard to predict interactions and noise, making it difficult to attribute observations to specific causes. Nevertheless, our framework enables users to write benchmarks that model the workload presented by a specific application to any desired level of detail.

To quantify queue performance, we measure the total runtime t of each benchmark pass. Using our queue instrumentation wrapper, we count the total number N_{enq} and N_{deq} of successful, and $N_{\neg enq}$ and $N_{\neg deq}$ of failed enqueue and dequeue attempts within each benchmark pass. We can then calculate the achieved queue throughput

$$T = \frac{N_{deq}}{t} \tag{1}$$

as the number of elements per second that could successfully be passed through the queue during the run.

We furthermore use our queue instrumentation wrapper to measure the latency of each successful and failed enqueue and dequeue attempt. The measured latencies are accumulated to the total time Λ_{enq}, $\Lambda_{\neg enq}$, Λ_{deq}, and $\Lambda_{\neg deq}$ spent on each set of attempts. Given the total time Λ_{op} spent on a given set of attempts as well as count N_{op} of attempts, we can calculate the average latency $\bar{\lambda}_{op}$ as well as the success or failure rate η_{op} for these attempts as

$$\bar{\lambda}_{op} = \frac{\Lambda_{op}}{N_{op}} \qquad \text{and} \qquad \eta_{op} = \frac{N_{op}}{N_{op} + N_{\neg op}}. \tag{2}$$

V. Discussion

Using our framework, we ran an extensive suite of benchmarks of select queue algorithms in variations of the synthetic producer-consumer scenario described above on an AMD Ryzen 9 5950X x86 CPU and an Apple M1 ARM CPU as well as on NVIDIA GeForce GTX 1080 and GeForce RTX 2080 SUPER, and AMD Radeon RX 6800 GPUs.

As one of the most-widely used queue algorithms, the Michael & Scott queue (MSQ) [10] serves as a baseline and representative of a linked, unbounded queue. Since this algorithm requires dynamic allocation of nodes, we provide a pre-allocated node pool large enough for the given benchmark scenario to keep allocation overhead to a minimum and study this algorithm in a best-case scenario.

Second, we benchmark the Broker Queue, a bounded queue designed for the GPU that has been shown to outperform other queue algorithms on the GPU [1]. We compare two versions of the Broker Queue: the non-linearizable Broker Work Distributor (BWD) and an optimized index queue variant (BIQ) of the BWD designed to store integer numbers.

Third, we benchmark the `moodycamel` queue (MOQ), which may be among the fastest concurrent queue implementations on the CPU [25] and is also widely used in practice. Here, we utilize the original author's implementation of the MOQ integrated into our framework via its C interface in the manner described in section III-D.

Benchmark scenarios include all combinations of enqueue and dequeue probabilities p_{enq} and p_{deq} of 1, 0.5, and 0.25 with workload sizes W of 1 and 512 for powers of two in thread count N between 1 and 2^{20}. For BWD, BIQ, and MOQ, queue sizes S of $16\,\text{k}$ and $1\,\text{M}$ are used. In all cases, queues store 32-bit integer elements.

Measurements were averaged over 8 out of 10 benchmark runs where the first 2 runs are discarded as warm up runs. The version of the framework we used was compiled with LLVM 14. When targeting the Apple M1, the target CPU was explicitly set to ensure appropriate code generation as we found that the target would otherwise default to an older version.

In the following section, we present some insights gained using the AnyQ evaluation framework. We will only highlight certain subsets of the data relevant to the discussion of the respective points. The complete dataset and interactive visualizations generated by the AnyQ framework are available online at https://anydsl.github.io/anyq.

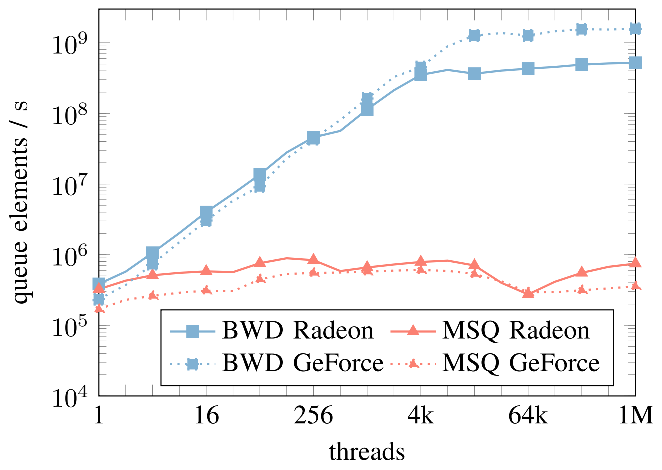


Figure 2: Throughput of BWD and MSQ on an NVIDIA GeForce RTX 2080 SUPER and AMD Radeon RX 6800 in a balanced scenario ($p_{enq} = p_{deq} = 0.5$, $W = 1$ with $S = 10\,\text{k}$).

A. Fastest Queue on the GPU

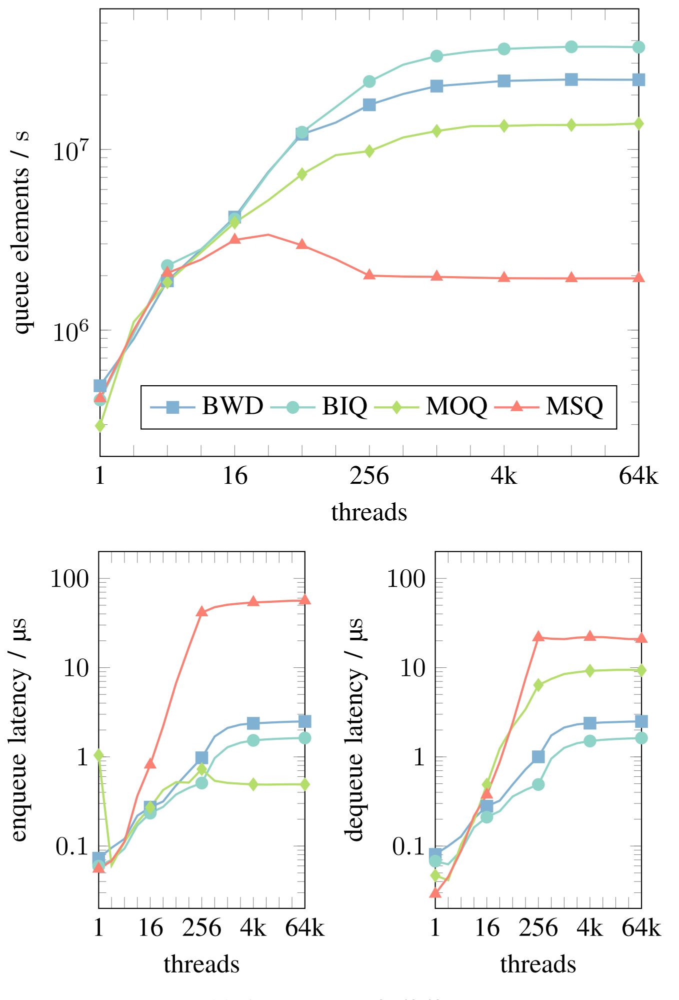
As another validation of our framework, we wanted to reproduce previously published results. While it is difficult to match exact numbers as we did not have access to the same GPUs used by Kerbl et al. [1], we can confirm the superior performance of the BWD on NVIDIA GeForce GTX 1080 and GeForce RTX 2080 SUPER GPUs. As can be seen in Figure 2, we can further amend these results with numbers that show BWD performance on an AMD Radeon RX 6800 to be very similar to its performance on NVIDIA hardware.

B. Fastest Queue on the CPU

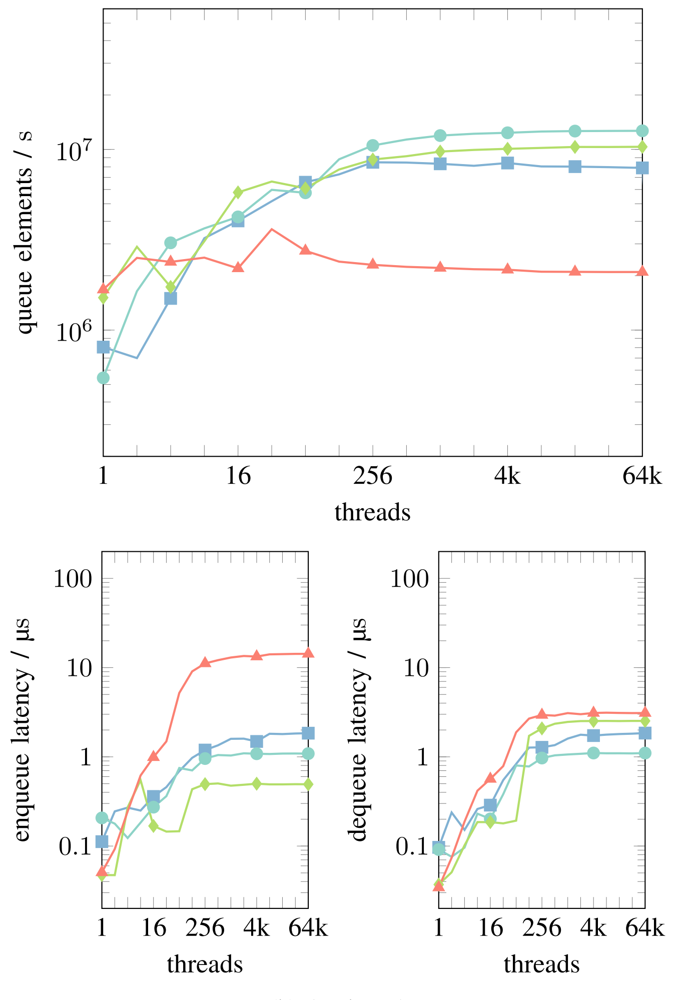
The MOQ has been shown by its author to outperform several other commonly-used concurrent queue implementations such as the queues provided by the Intel TBB and Boost.Lockfree libraries [25]. We expand upon on these results by comparing MOQ against algorithms that have been studied in the scientific literature. At the same time, we provide a first evaluation of the Broker Queue algorithm on the CPU.

Figure 3 shows the average throughput as well as enqueue and dequeue latencies on an AMD Ryzen 9 5950X and Apple M1 CPU for a balanced scenario using our vectorized mapping. On both architectures, BIQ achieves the highest throughput, demonstrating that queue algorithms designed for the GPU translate well to SIMD execution on modern CPUs.

The relative performance of queue algorithms behaves similarly on both the x86 and ARM CPUs. For low thread counts, all queue algorithms are closely matched on the x86 CPU. On the ARM CPU, we find that MSQ outperforms the other algorithms at low levels of contention but is quickly overtaken by the other queues as contention increases. BWD, BIQ, and MOQ simply approach and then plateau at their maximum throughput. This further demonstrates the effectiveness of our mapping as we see scaling well beyond hardware concurrency on both CPUs, limited only by atomic operation throughput. MSQ peaks at around 32 threads. As a result of its optimistic concurrency



(a) AMD Ryzen 9 5950X



(b) Apple M1

Figure 3: Average throughput as well as enqueue and dequeue latency for BWD, BIQ, MOQ, and MSQ on (a) an AMD Ryzen 9 5950X x86 CPU and (b) an Apple M1 ARM CPU in a vectorized balanced scenario ($p_{enq} = p_{deq} = 0.5$, $W = 1$, $S = 16\,\mathrm{k}$).

control, any further increase in concurrent queue operations simply leads to more and more contending operations causing each other to fail, tying up hardware resources in the process, and resulting in the algorithm settling at a lower throughput.

Due to its integration as an external library, MOQ does not lend itself to vectorization. Nevertheless, we found that both Broker Queue variants consistently outperform MOQ on the x86 CPU even under the scalar mapping. On the ARM CPU, however, the relative disadvantage of MOQ was significantly less already in the vectorized case, likely due to the smaller SIMD width. As can be seen in Figure 4, MOQ outperforms all other queues on the ARM CPU in the scalar case.

Thus, while MOQ still seems to have an edge in traditional scalar threaded applications on ARM, the Broker Queue needs to be considered on x86 and for SIMD processing in general. We also point out that, on both CPUs, MOQ showed faster enqueue times while BWD and BIQ showed faster dequeue times, which may be a relevant factor for latency-sensitive applications to take into account.

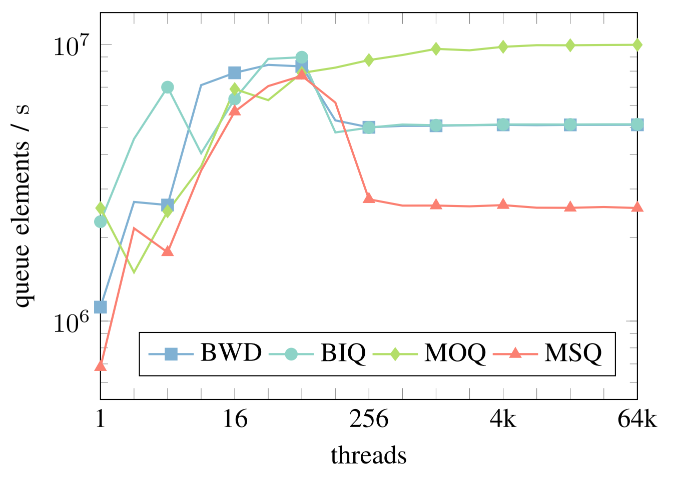


Figure 4: Average throughput for BWD, BIQ, MOQ, and MSQ on an Apple M1 ARM CPU in a scalar balanced scenario ($p_{enq} = p_{deq} = 0.5$, $W = 1$, $S = 16\,\mathrm{k}$).

C. *Effect of Vectorization*

Comparing queue performance under our vectorized and scalar mappings, we find that the vectorized mapping consistently outperforms the scalar mapping across the entire range of benchmark parameters for all queue algorithms on both the AMD Ryzen 9 5950X x86 CPU and the Apple M1 ARM CPU. However, as can be seen in Figure 5, some queue algorithms benefit from vectorization more than others.

The BWD and BIQ algorithms use atomic FAA to assign queue slots to threads. The vectorizer can aggregate the FAAs performed by all threads within a wave via a parallel reduction. Thus, a single atomic FAA can service all threads in the wave that are performing the same queue operation in parallel. As a result, successful queue operations within the same wave will also be operating on consecutive queue elements, which further benefits performance and may give rise to additional vectorization opportunities.

While the BWD and BIQ represent a close to ideal case for vectorization, the MSQ algorithm represents a closer to worst-case scenario. At its core, the MSQ consists of CAS loops where parallel queue operations inherently cannot succeed in parallel. However, even given these algorithmic limitations, MSQ still benefits from vectorization as at least the atomic loads for fetching the current queue state can be performed as one shared operation for all threads in a wave.

Since queue throughput is ultimately limited by atomic operation throughput, the vectorized mapping achieves higher performance by being able to perform more queue operations for the same amount of atomic operations. However, latency of queue operations also increases due to additional work for distributing results of atomics back to the individual threads as well as due to the wave being dominated by its slowest thread.

D. *Independent Thread Scheduling*

Starting with the Volta architecture, NVIDIA GPUs support ITS, which provides a forward progress guarantee for all threads irrespective of branching behavior. As a result, the constraints concerning queue operations in divergent branches within the same wave can be dropped. To investigate the impact of ITS on queue performance, we implemented a variant of our benchmark without wave barriers separating enqueue and dequeue attempts, and compare the performance of both variants for BWD and MSQ on the GeForce RTX 2080 SUPER.

As can be seen in Figure 6, overall queue throughput with ITS is decreased for BWD and almost unchanged for MSQ. We attribute the lower BWD performance to the fact that enqueue and dequeue operations for the same wave can now both run concurrently, resulting in more atomic operations in flight and, thus, increased pressure on the memory system. Despite the loss in throughput, individual operation latency is slightly reduced, however, most-likely as a result of the lack of wave-level synchronization allowing operations to complete without having to wait on others.

Since MSQ is based on contending threads helping one another complete each other's operations, the impact of increasing the number of concurrent queue operations is mitigated and we observe no significant change in overall throughput with ITS. Due to less synchronization and potentially there being more helpers, the latency at which individual queue operations complete is noticeably reduced.

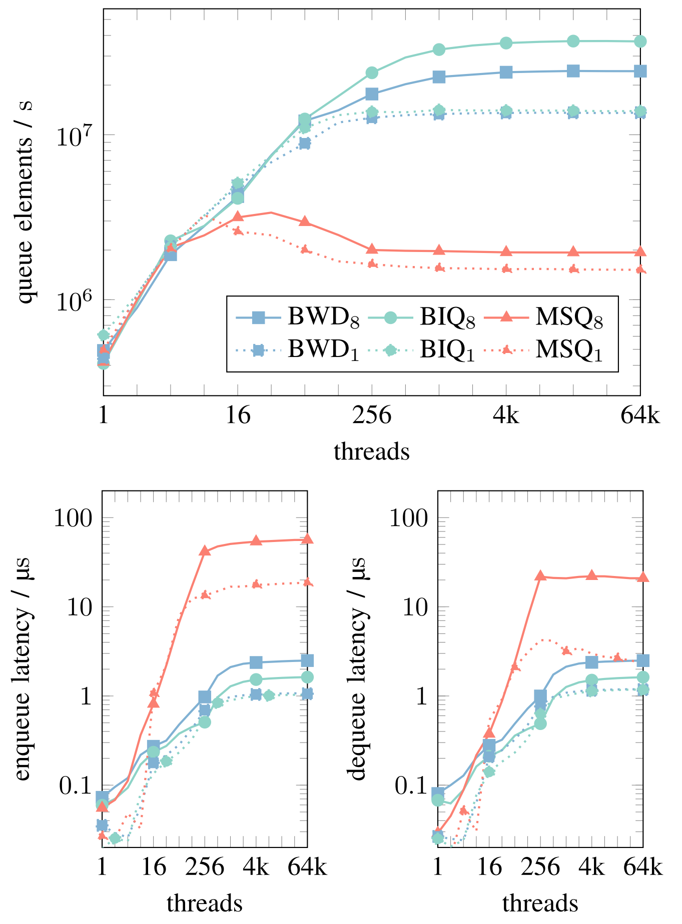


Figure 5: Comparison of scalar and 8-wide vectorized execution of the same balanced scenario on an AMD Ryzen 9 5950X. The vectorized BIQ_8, MOQ_8, MSQ_8 consistently outperform the corresponding scalar BIQ_1, MOQ_1, MSQ_1. As a result of the reduction in atomic operations, the vectorized versions achieve a significantly higher peak throughput at exactly $8\times$ the number of threads at the cost of increased latency.

E. *Atomics on ARM Architecture*

As pointed out earlier, certain architectures rely on LLCS instructions to implement the atomic operations queue algorithms depend on. Due to the different performance characteristics of such atomics compared to native atomic RMW instructions, we would expect there to be potentially significant differences in how various queue algorithms compare on such architectures. To investigate, we benchmark the BIQ and MSQ algorithms on an Apple M1 ARM CPU. As BIQ heavily relies on FAA while MSQ exclusively uses CAS, we would expect these two candidates to be representative of the range of behaviors one might encounter.

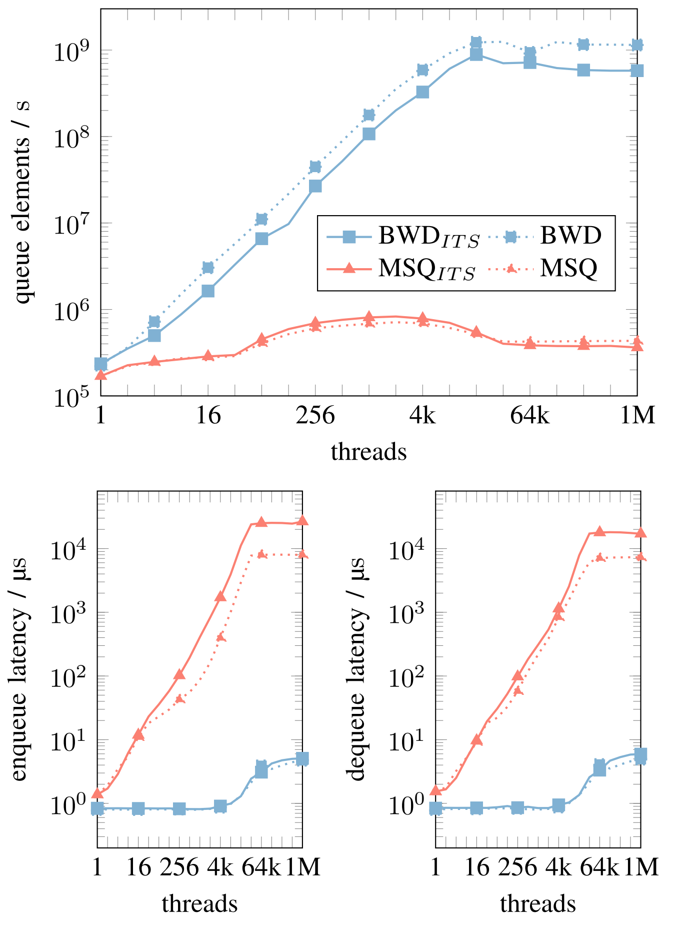


Figure 6: Effect of ITS on a NVIDIA GeForce RTX 2080 SUPER. The BWD experiences reduced performance with ITS as a result of increased contention from more queue operations being executed concurrently. Performance for MSQ meanwhile is mostly unaffected. Due to the lack of wave-level synchronization, ITS generally results in reduced latency.

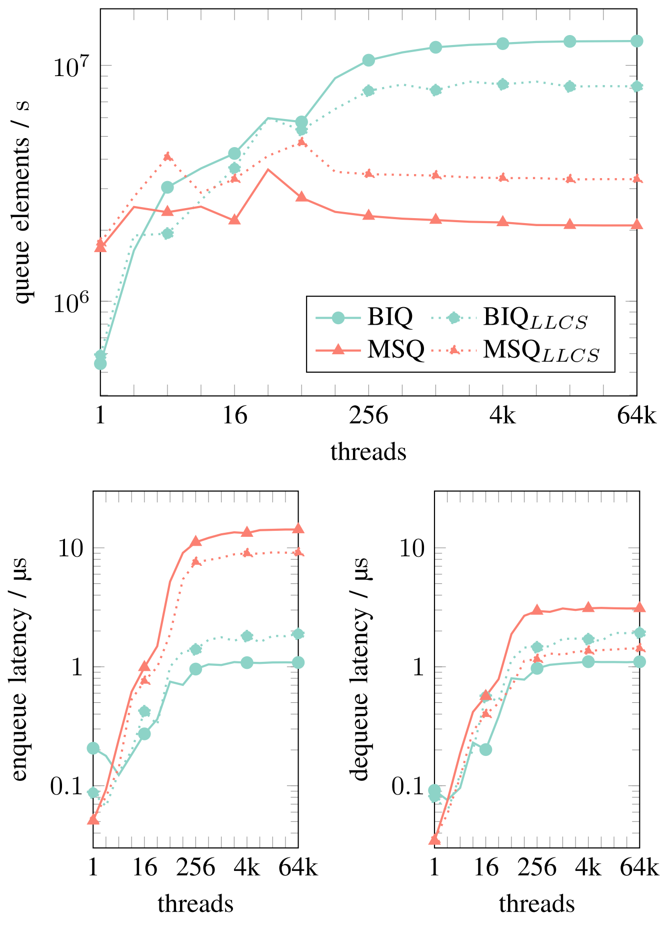


Figure 7: Effect of using native atomic RMW operations on an Apple M1 ARM CPU over LLCS instructions. While BIQ benefits from native RMW atomics and finds its performance severely hampered by the need to implement RMW operations using LLCS instructions, LLCS affords MSQ a unique ability to fail early, resulting in performance much more competitive than would otherwise be expected.

The Apple M1 supports the Armv8.1 Large System Extensions (LSE) [26], which add native atomic RMW instructions. We compiled the same benchmark once with LSE enabled and once with LSE disabled. With LSE, the compiler will use native RMW atomics, turning LSE off will cause the compiler to fall back to LLCS instructions. We verified that the generated machine code only differs in the instructions used for the respective atomic operations. The only other differences we found were minor shifts in register and address allocation as a consequence of the changes around atomic operations.

Thus, we can now compare the exact same benchmark on the exact same machine with the only difference being whether LLCS or native RMW atomics are being used. Figure 7 shows the results for a balanced workload using our vectorized mapping. Similar trends, albeit less pronounced, were observed with the scalar mapping. As expected, both throughput as well as latency significantly improve for BIQ when using native RMW atomics. The algorithm is designed to take advantage of FAA. With LLCS, only one contending FAA can succeed at a time and will cause all others to fail and require a retry. As contention rises, this quickly turns into a bottleneck.

MSQ performance, however, increases dramatically when using LLCS instructions. The MSQ is built around CAS retry loops and our implementation takes advantage of weak CAS where possible. If another thread interferes at any point during an attempted queue operation, the algorithm needs to retry the entire operation. When using LLCS, interference by another thread will immediately cause the weak CAS to fail spuriously. The native CAS, on the other hand, must wait for the hardware to resolve contending operations before being able to even detect that a retry is required, which explains the lower throughput and higher latencies.

Thus, while BIQ behaved as expected, somewhat counter-intuitively, it may be advantageous to avoid LSE extensions and take advantage of LLCS for highly-contended weak CAS operations in MSQ and similar style algorithms on ARM and other architectures that offer both kinds of instructions.

VI. Conclusion

We have presented the design and implementation of AnyQ, an evaluation framework for concurrent queue algorithms. AnyQ enables us to express concurrent queue algorithms and benchmarks in a way that allows for automatic mapping to a wide range of target architectures. As a result, it is now possible to easily study and compare the performance of different queue algorithms on a variety of machines.

Using the developed framework, we set out to re-evaluate various existing queue algorithms to both provide updated results on their performance as well as address unanswered questions. We showed that concurrent queues can benefit significantly from SIMD execution on CPUs, and that queue algorithms designed for the GPU, such as the Broker Queue, perform similarly well on modern CPUs. We found similarities and differences in how queue algorithms perform on architectures not considered in their original design and evaluation.

Our results suggest that LLCS may offer unique abilities and should be considered more than simply a way to implement the usual atomic FAA and CAS primitives. LLCS also avoids the ABA problem that has been plaguing concurrent algorithms for a very long time [27]. We believe that the design of algorithms that specifically take advantage of these abilities presents an interesting avenue for future research. While architectures like ARM with LSE today offer both kinds of instructions as an accidental consequence of their evolution, it may turn out desirable for instruction set architectures in the future to offer both native atomic RMW as well as LLCS instructions.

All source code and data is available as open source software at https://anydsl.github.io/anyq. We hope that it may serve as a starting point for the community to build a common repository of concurrent queue algorithms and benchmarks. We openly invite contributions of additional queue algorithms, benchmarks, and results. Such a repository would move us closer to the goal of ensuring continued evaluability of queue algorithms as the landscape of computing hardware keeps evolving. It could provide a wider audience of developers with a quick and easy way to find out which queue algorithm to use by writing an AnyQ benchmark modeling their application's workload. And it would offer a base for future research to develop and easily evaluate concurrent queue algorithms in a way that is representative, comparable, and reproducible.

Acknowledgments

This work is supported by the Federal Ministry of Education and Research (BMBF) as part of the ENGAGE, HorME, and MetaDL projects.

References

[1] B. Kerbl, M. Kenzel, J. H. Mueller, D. Schmalstieg, and M. Steinberger, "The broker queue: A fast, linearizable FIFO queue for fine-granular work distribution on the GPU," in *Proc. 2018 International Conference on Supercomputing*, ser. ICS. ACM, 2018, pp. 76–85.

[2] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," *SIGPLAN Not.*, vol. 48, no. 8, pp. 103–112, 2013.

[3] C. Yang and J. Mellor-Crummey, "A wait-free queue as fast as fetch-and-add," in *Proc. 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP. ACM, 2016.

[4] NVIDIA, "NVIDIA tesla V100 GPU architecture," NVIDIA Corporation, Tech. Rep., 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[5] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[6] Y. Afek, G. Korland, and E. Yanovsky, "Quasi-linearizability: Relaxed consistency for improved concurrency," in *Principles of Distributed Systems*, C. Lu, T. Masuzawa, and M. Mosbah, Eds. Springer, 2010, pp. 395–410.

[7] C. M. Kirsch, M. Lippautz, and H. Payer, "Fast and scalable, lock-free k-FIFO queues," in *Parallel Computing Technologies*, V. Malyshkin, Ed. Springer, 2013, pp. 208–223.

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

[9] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *Proc. 23rd International Conference on Distributed Computing Systems*, 2003, pp. 522–529.

[10] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proc. 15th Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC. ACM, 1996, pp. 267–275.

[11] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, 1981.

[12] R. Nikolaev, "A scalable, portable, and memory-efficient lock-free FIFO queue," *CoRR*, vol. abs/1908.04511, 2019.

[13] E. H. Jensen, J. M. Broughton, and G. W. Hagensen, "A new approach to exclusive data access in shared memory multiprocessors," Lawrence Livermore National Laboratory, Tech. Rep. UCRL-97663, 1987.

[14] T. R. Scogland and W.-c. Feng, "Design and evaluation of scalable concurrent queues for many-core architectures," in *Proc. 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE. ACM, 2015, pp. 63–74.

[15] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization (CGO)*, 2004, pp. 75–86.

[16] R. Leißa, K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. Müller, and B. Schmidt, "AnyDSL: A partial evaluation framework for programming high-performance libraries," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 2, no. OOPSLA, pp. 119:1–119:30, 2018.

[17] NVIDIA, *CUDA C Programming Guide v11.6.0*, NVIDIA Corporation, 2022.

[18] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science & Engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

[19] HSA, *HSA Programmer's Reference Manual Specification 1.2*, HSA Foundation, 2018.

[20] S. Moll and S. Hack, "Partial control-flow linearization," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2018, pp. 543–556.

[21] M. Pharr and W. R. Mark, "ispc: A SPMD compiler for high-performance cpu programming," in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–13.

[22] H.-J. Boehm and S. V. Adve, "Foundations of the C++ concurrency memory model," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI. ACM, 2008, pp. 68–78.

[23] Intel, *oneAPI Threading Building Blocks*, Intel Corporation, 2021. [Online]. Available: https://threadingbuildingblocks.org

[24] G. Marsaglia, "Xorshift RNGs," *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 2003. [Online]. Available: https://www.jstatsoft.org/index.php/jss/article/view/v008i14

[25] C. Desrochers, "A fast general purpose lock-free queue for C++," moodycamel, Tech. Rep., 2014. [Online]. Available: https://moodycamel.com/blog/2014/a-fast-general-purpose-lock-free-queue-for-c++.htm

[26] ARM, *Arm Architecture Reference Manual for A-profile architecture*, ARM Limited, 2022.

[27] D. Dechev, P. Pirkelbauer, and B. Stroustrup, "Understanding and effectively preventing the aba problem in descriptor-based lock-free designs," in *Proc. 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 2010, pp. 185–192.