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

ICPE 2015

February 2nd, 2015

Thomas R. W. Scogland, Wu-chun Feng

Lawrence Livermore National Laboratory

LLNL-PRES-666776

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Why *another* concurrent queue?

Heterogeneity and many-core are a *fact* of life in modern computing

Everythi l phones

By Zach Vega (Own work)
[CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons

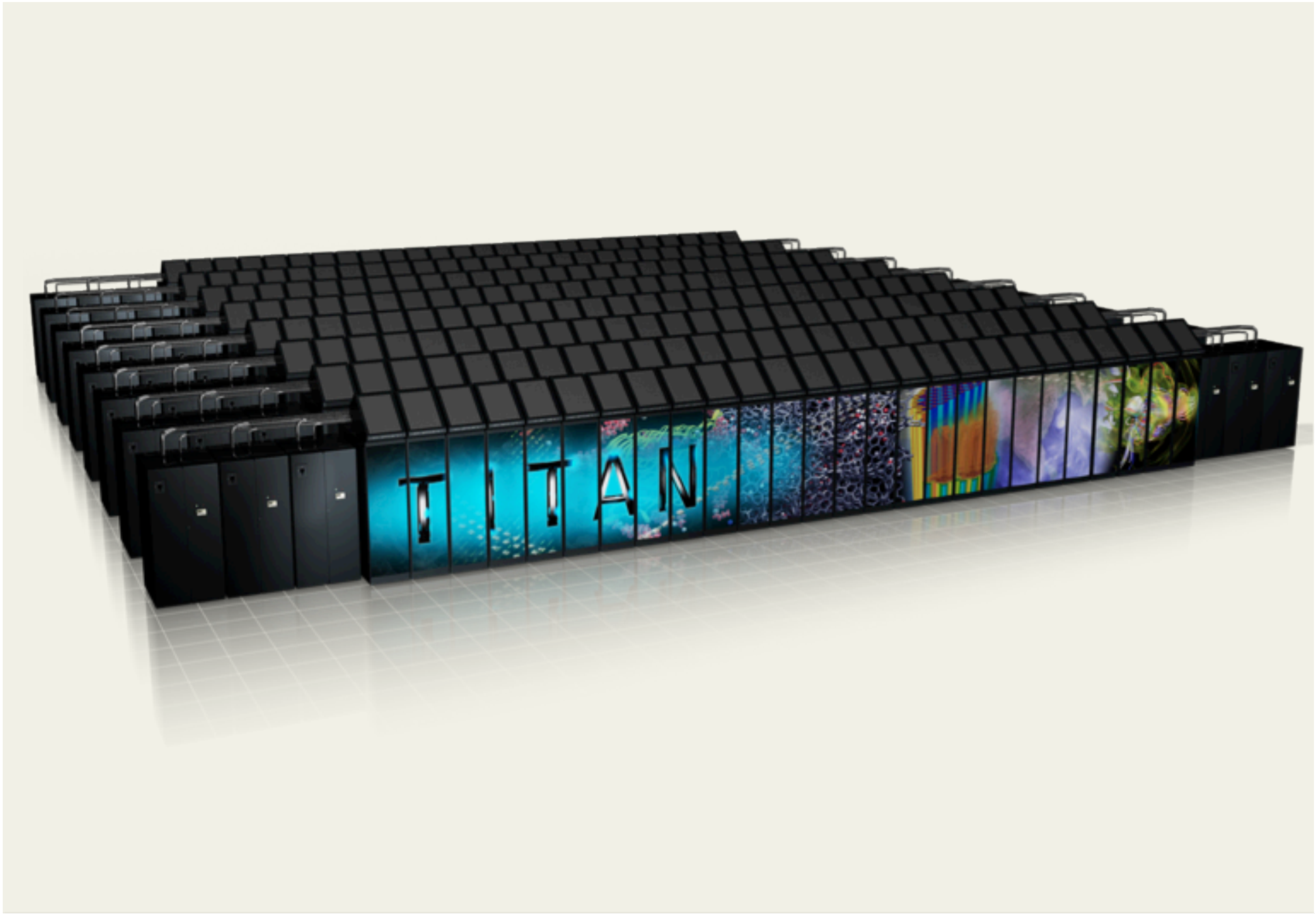
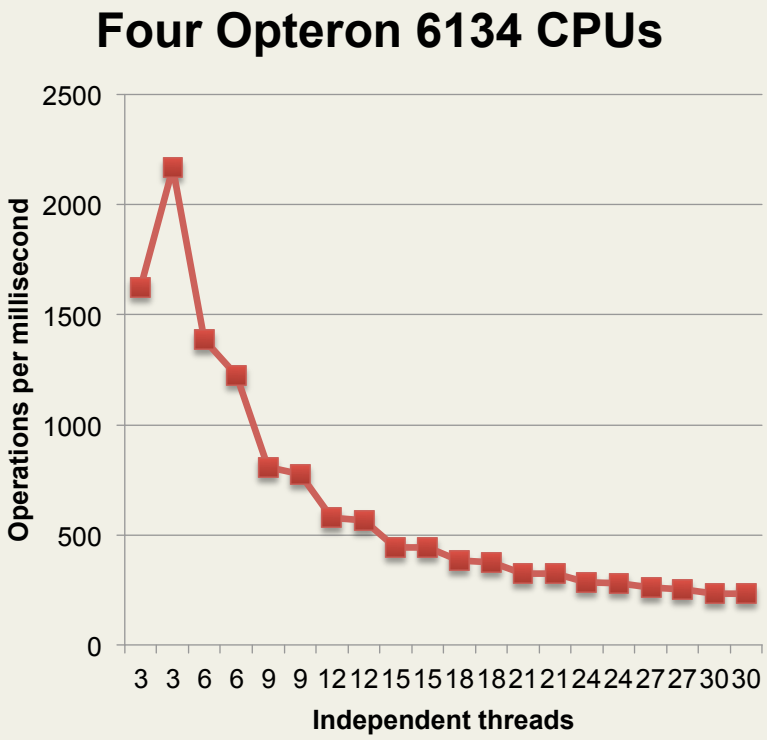
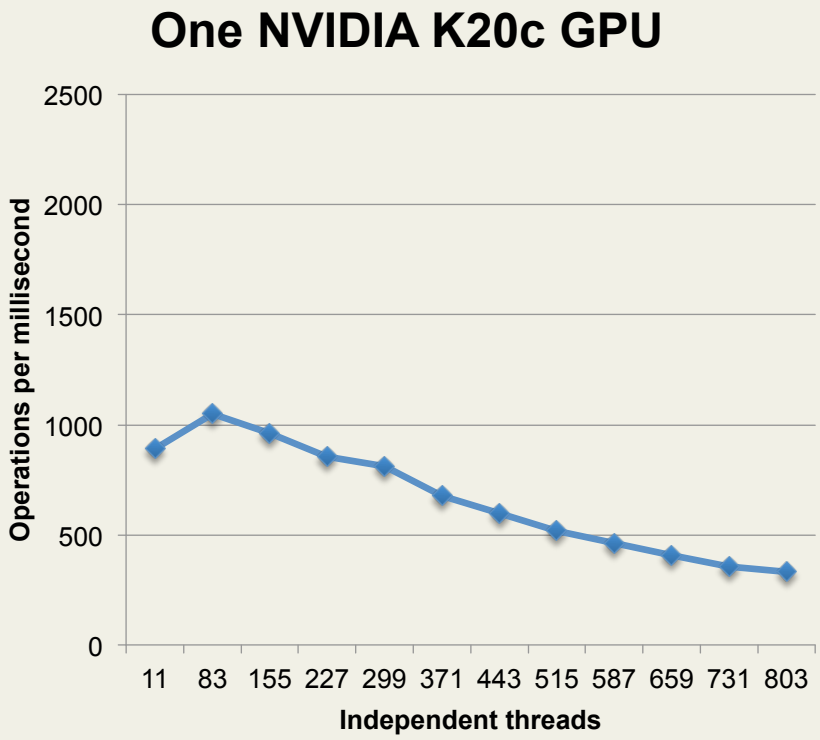


 Image Courtesy of Oak Ridge National Laboratory, U.S. Dept. of Energy

Why not existing lock-free queues?

- Traditional lock-free queues focus on progress over throughput
- Perfect for over-subscribed systems, but they **do not scale**





Outline

- Definitions and abstractions
- Building blocks: Evaluating atomic operations
- Queue types and modeling
- Our queue design
- Performance evaluation
- Conclusions

Definitions: What is a “thread”?

- Work-item: The basic unit of work in OpenCL
  - Groups of work-items execute in lock-step
  - Work-items are **not** threads

- Thread: An independently schedulable entity
  - An OS thread on CPUs
  - In OpenCL, defined as a group of work-items of size “PREFERRED_WORK_GROUP_SIZE_MULTIPLE”

Abstractions

- All operations defined in terms of **atomics**
- On CPU:
  - Add: Atomic Fetch-and-add (FAA)
  - Read: Normal load
  - Write: Normal store
  - CAS: Atomic Compare and Swap
- On OpenCL:
  - Add: Atomic Fetch-and-add (FAA)
  - Read: Atomic Fetch-and-add 0, or atomic_read, or regular read after flush if available
  - Write: Atomic exchange
  - CAS: Atomic Compare and Swap

Experimental Setup: Hardware: CPUs

Device	Num. devices	Cores/ device	Threads/ core	Max. threads	Max. achieved
AMD Opteron 6134	4	8	1	32	32
AMD Opteron 6272	2	16	1	32	32
Intel Xeon E5405	2	4	1	8	8
Intel Xeon X5680	1	12	2	24	24
Intel Core i5-3400	1	4	1	4	4

Experimental Setup: Hardware: GPUs/Co-processors

Device	Num. devices	Cores/ device	Threads/ core	Max. threads	Max. achieved
AMD HD5870	1	20	24	496	140
AMD HD7970	1	32	40	1280	386
AMD HD7990	1 (of 2 dies)	32	40	1280	1020
Intel Xeon Phi P1750	1	61	4	244	244
NVIDIA GTX 280	1	30	32	960	960
NVIDIA Tesla C2070	1	14	32	448	448
NVIDIA Tesla K20c	1	13	64	832	832

Experimental setup: Software

- Debian Wheezy Linux 64-bit kernel version 3.2
- NVIDIA driver v. 313.3 with CUDA SDK 5.0
- AMD fglrx driver v. 9.1.11 and APP SDK v. 2.8
- Intel Xeon Phi driver MPSS gold 3
- CPU and Phi OpenMP use Intel ICC v. 13.0.1

Experimental setup: Detecting the real number of threads

```
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));
}
```

Check if kernel is complete

Increment number of threads, returns TID

Set kernel complete

Outline

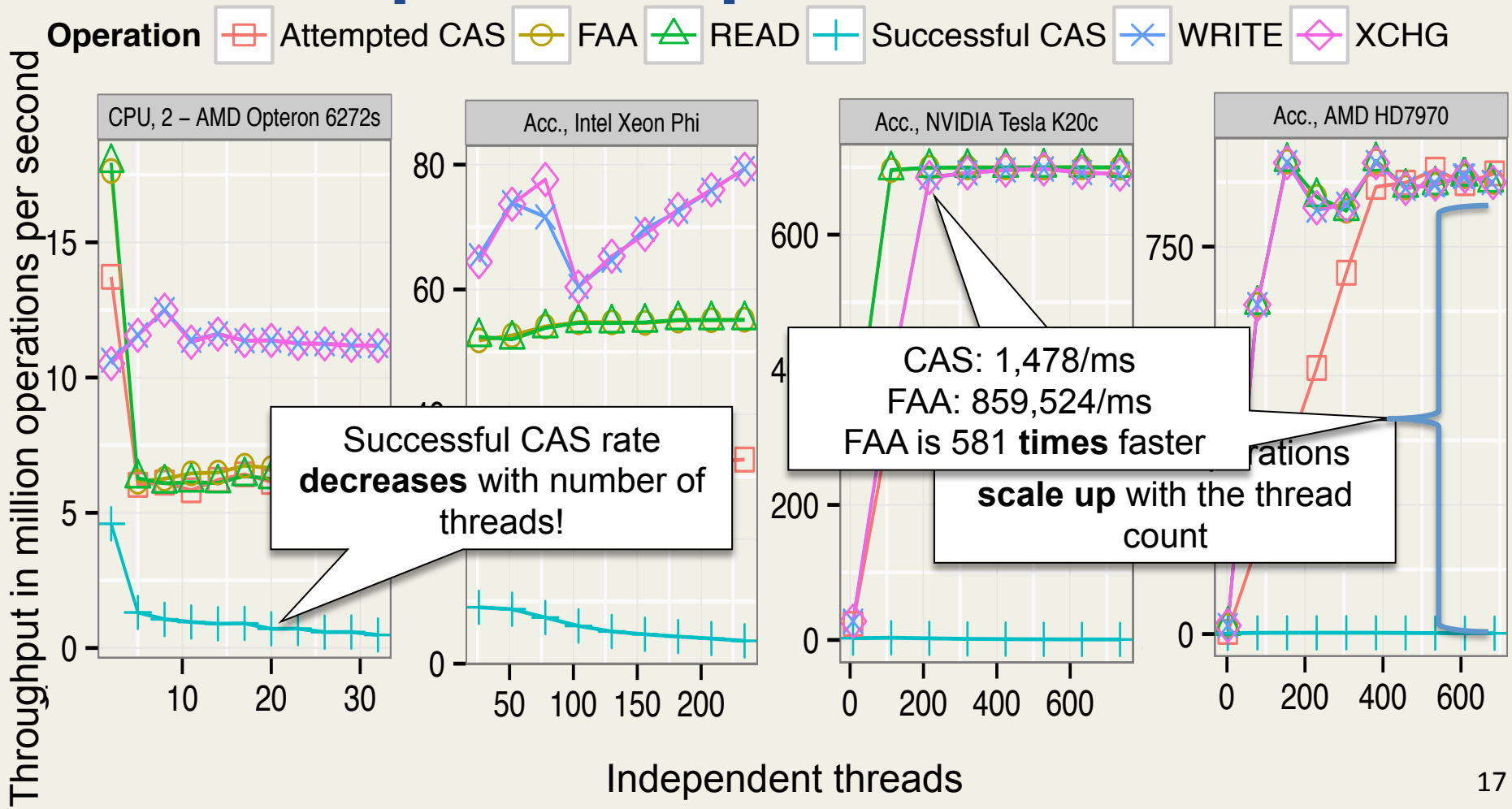
- Definitions, abstractions and experimental setup
- Building blocks: Evaluating atomic operations
- Queue types and modeling
- Our queue design
- Performance evaluation
- Conclusions

Atomic performance test

```
kernel void cas_test(__global unsigned * in, __global unsigned * out, unsigned iterations){
  const unsigned tid = (get_local_id(1)*get_local_size(0)) + get_local_id(0);
  const unsigned gid = (get_group_id(1)*get_local_size(0)) + get_group_id(0);
  __local unsigned success;
  unsigned my_success = 0;

  if(tid == 0){
    unsigned prev = 0;
    for(size_t i=0; i < iterations; ++i){
      prev = atomic_add(in,0);
      my_success += atomic_cmpxchg(in,prev,prev+1) == prev ? 1 : 0;
    }
    out[gid] = my_success;
  }
}
```

Atomic operation performance



For more architectures, see the paper

Outline

- Definitions and abstractions
- Building blocks: Evaluating atomic operations
- Queue types and modeling
- Our queue design
- Performance evaluation
- Conclusions

General modeling of queues

- All concurrent queues require either:
  - Locks, or
  - Atomic operations
- Model result: Throughput (T) for a given number of threads (t)
- Terms, average latency of constituent atomics:
  - Read: r
  - Write: w
  - Successful contended CAS: c
  - Attempted CAS: C

Queue types

- Contended CAS
  - MS queue and TZ queue

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

- Un-contended CAS
  - LCRQ

$$T_t = \frac{1}{(a_t + r_t + C_t)}$$

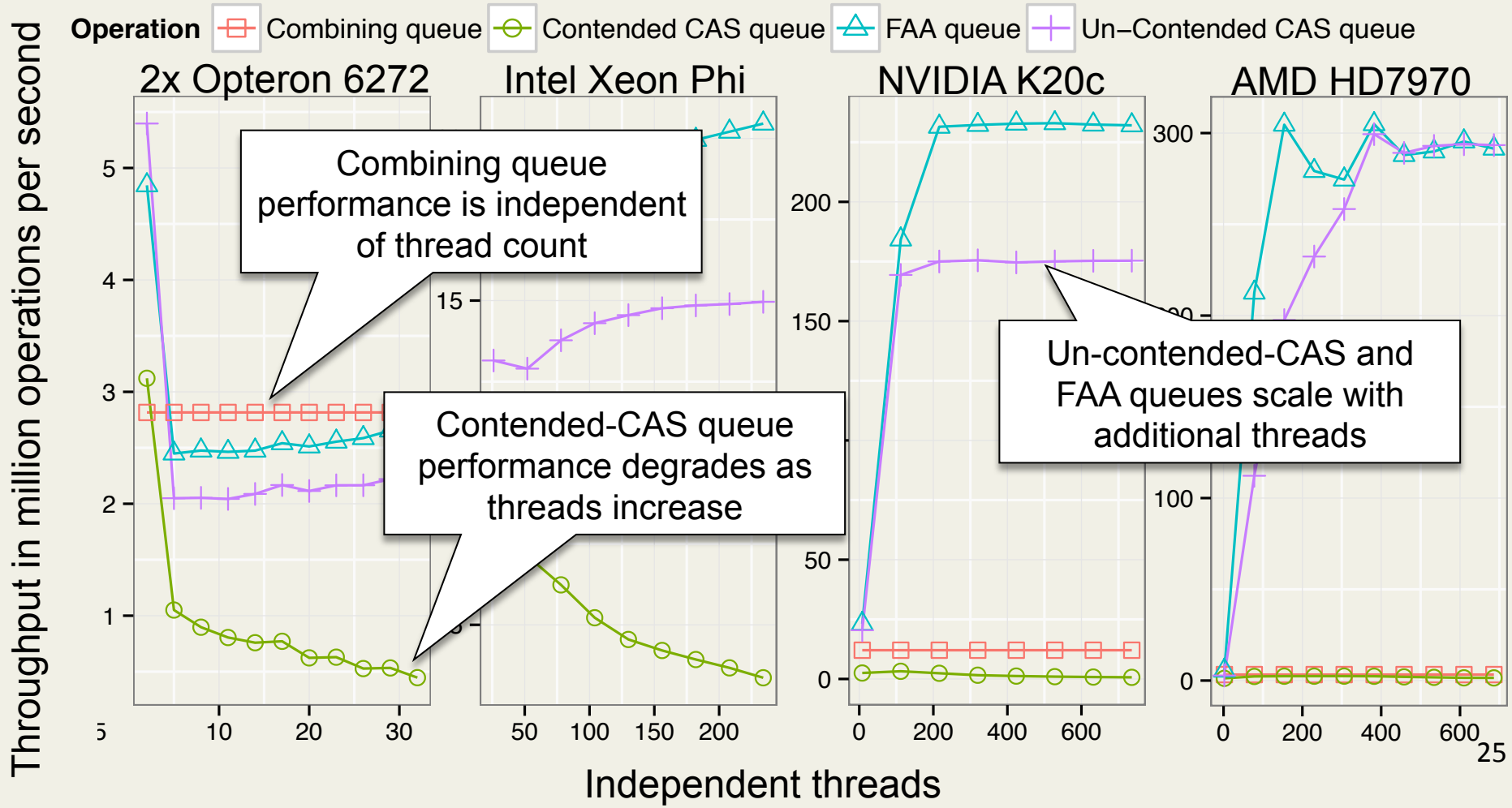
- Combining
  - FC queue

$$T_t = \frac{2}{(r_1 + w_1 \times 2) + (r_1 \times 2 + w_1)}$$

- FAA or blocking array
  - CB queue and our queue

$$T_t = \frac{2}{(a_t + r_t + w_t) + (a_t + w_t \times 2)}$$

Modeled queue throughput



Outline

- Definitions and abstractions
- Building blocks: Evaluating atomic operations
- Queue types and modeling
- Our queue design
- Performance evaluation
- Conclusions

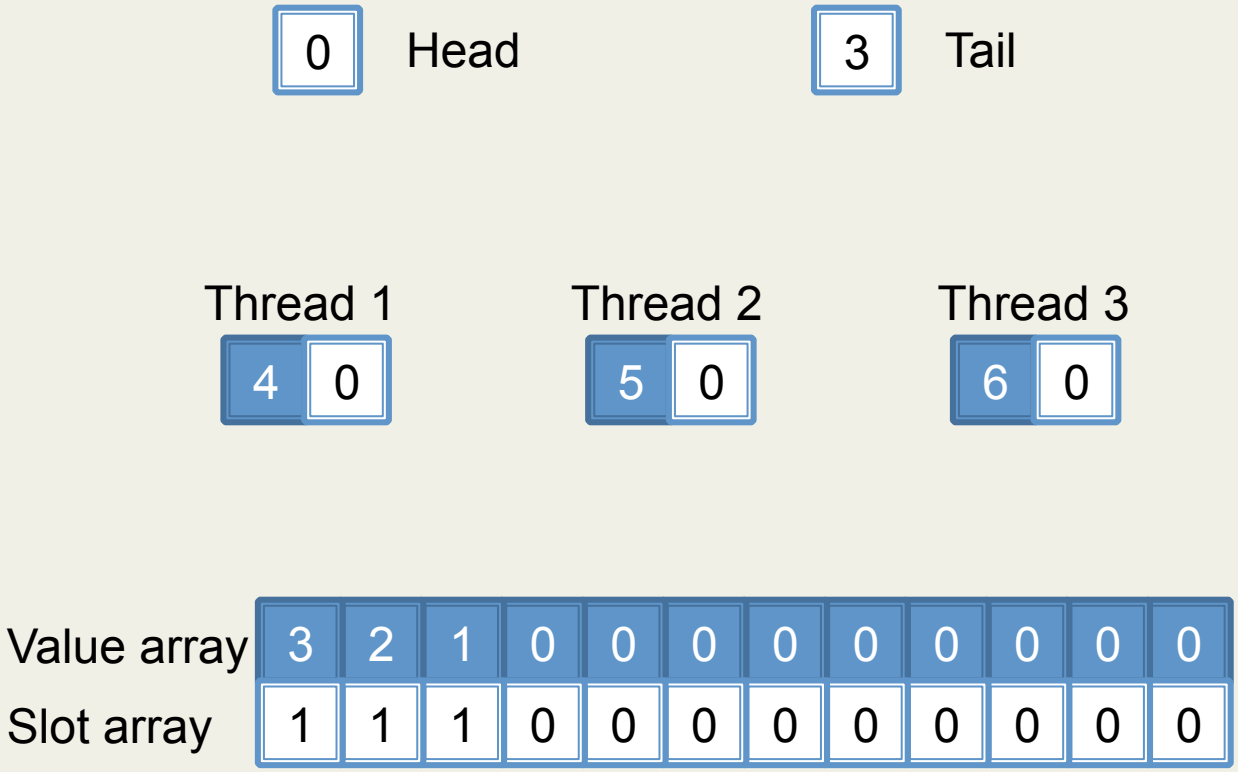
Our queue design: Goals

- Scale well on many-core architectures
  - Avoid contended CAS!
- Maintain Linearizability and FIFO ordering
- Allow the status of the queue to be inspected

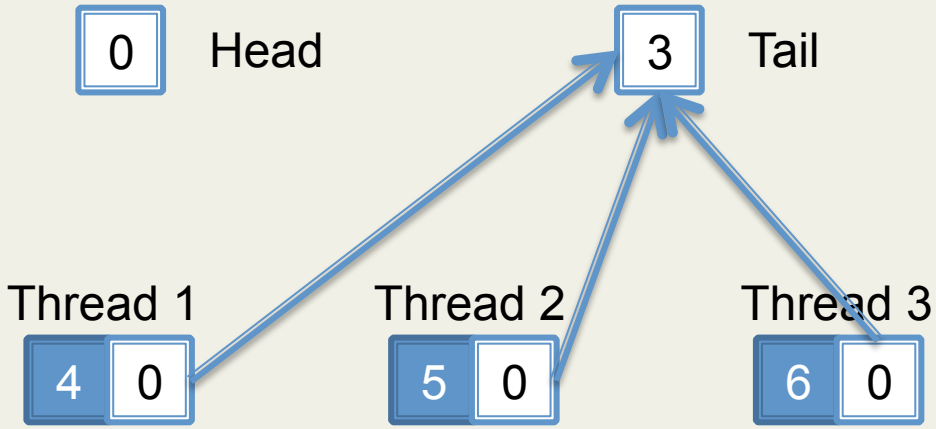
Our queue design: Solution, divide the interfaces

- Blocking interface: The fast, concurrent interface
  - enqueue(q, data) -> success or closed
  - dequeue(q, &data) -> success or closed
- Non-waiting interface:
  - enqueue_nw(q, data) -> success, not_ready or closed
  - dequeue_nw(q, &data) -> success, not_ready or closed
- Status inspection interface
  - distance(q) -> the distance between head and tail, corrected for rollover
  - waiting_enqueuers(q) -> number of enqueuers blocking
  - waiting_dequeuers(q) -> number of dequeuers blocking
  - is_full(q) -> true if full, else false
  - is_empty(q) -> true if empty, else false

Our queue's blocking behavior: Enqueue example: Get targets with FAA

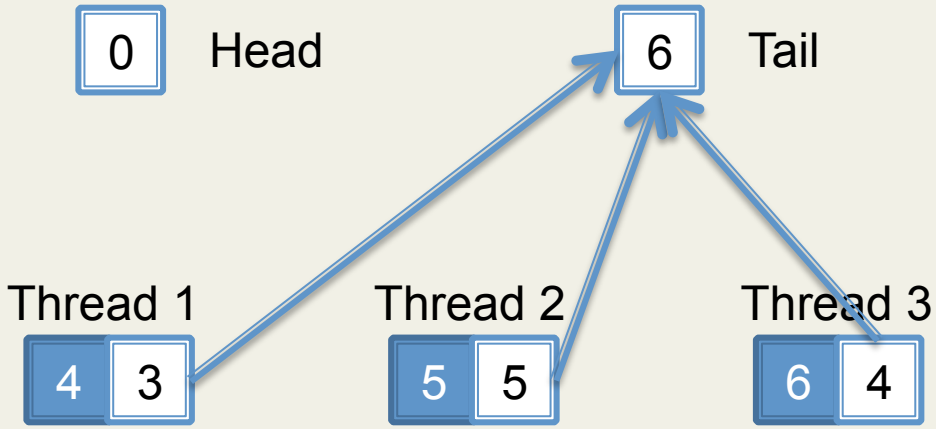


Our queue's blocking behavior: Enqueue example: Get targets with FAA



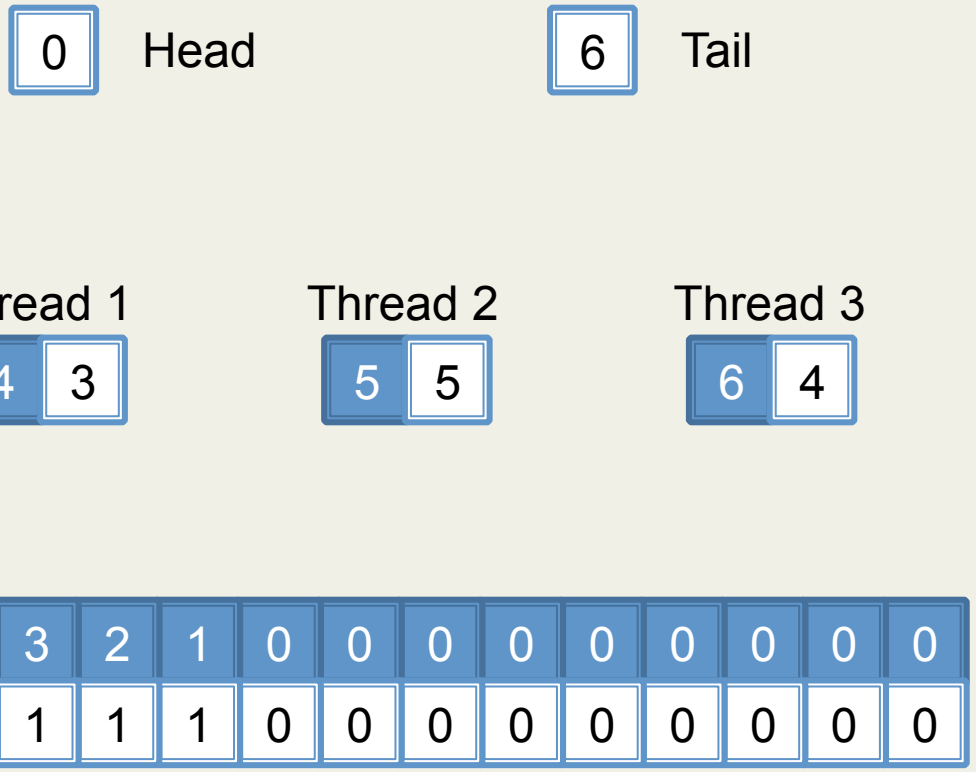
Value array	3	2	1	0	0	0	0	0	0	0	0	0
Slot array	1	1	1	0	0	0	0	0	0	0	0	0

Our queue's blocking behavior: Enqueue example: Get targets with FAA

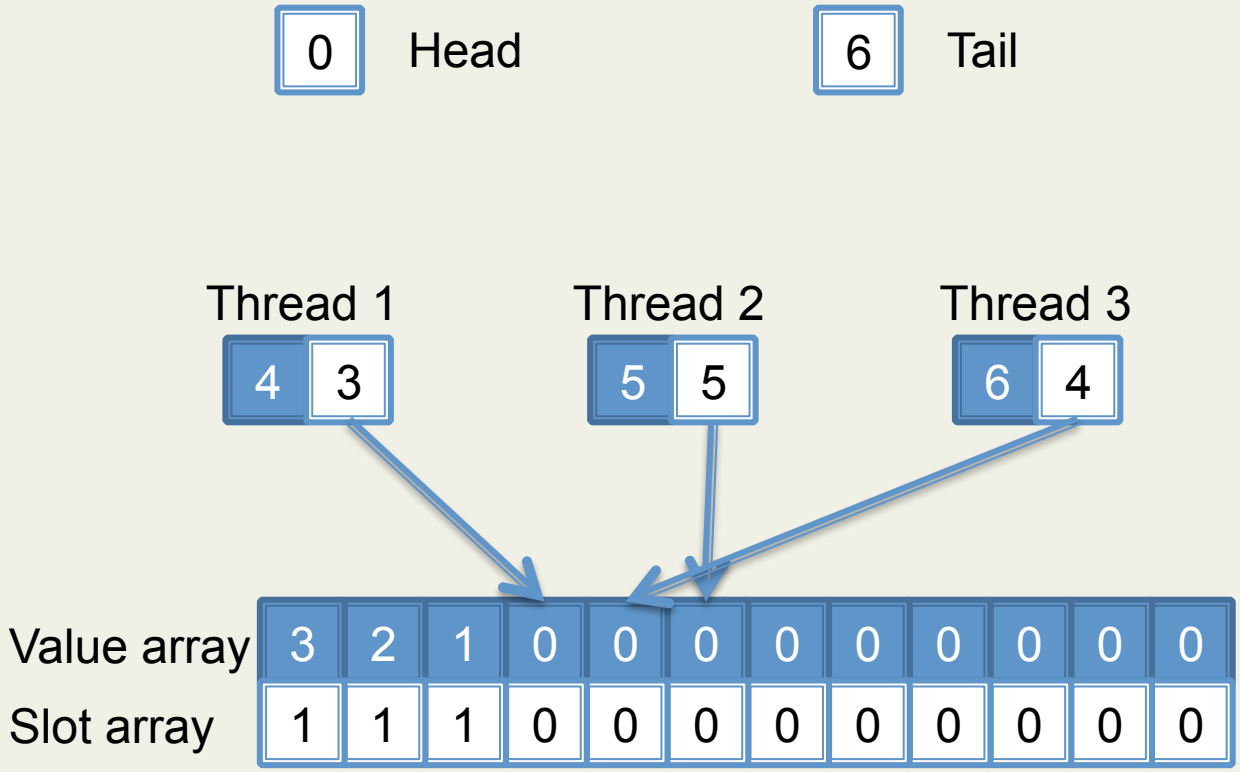


Value array	3	2	1	0	0	0	0	0	0	0	0	0
Slot array	1	1	1	0	0	0	0	0	0	0	0	0

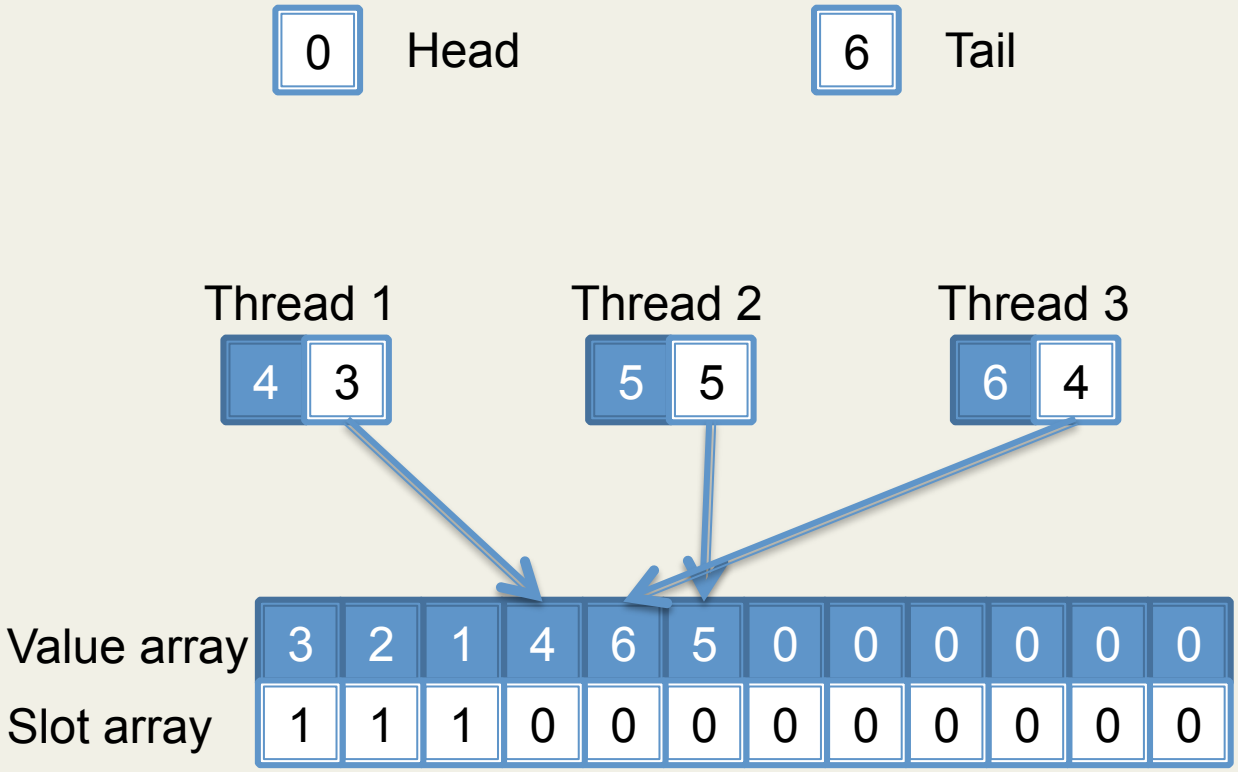
Our queue's blocking behavior: Enqueue example: Write values



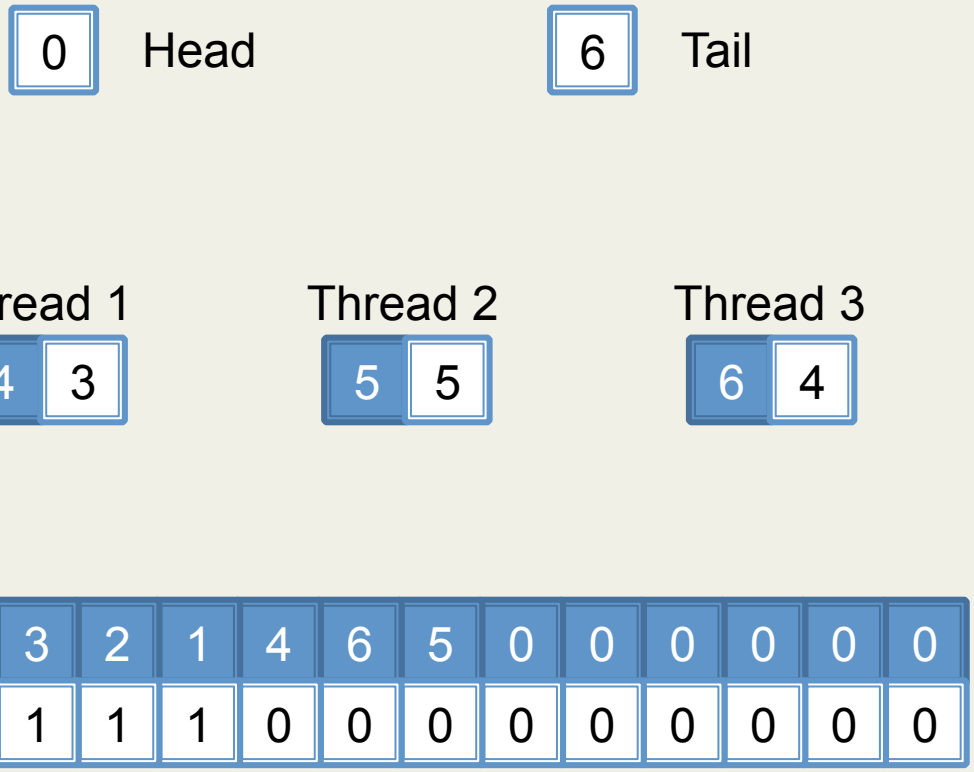
Our queue's blocking behavior: Enqueue example: Write values



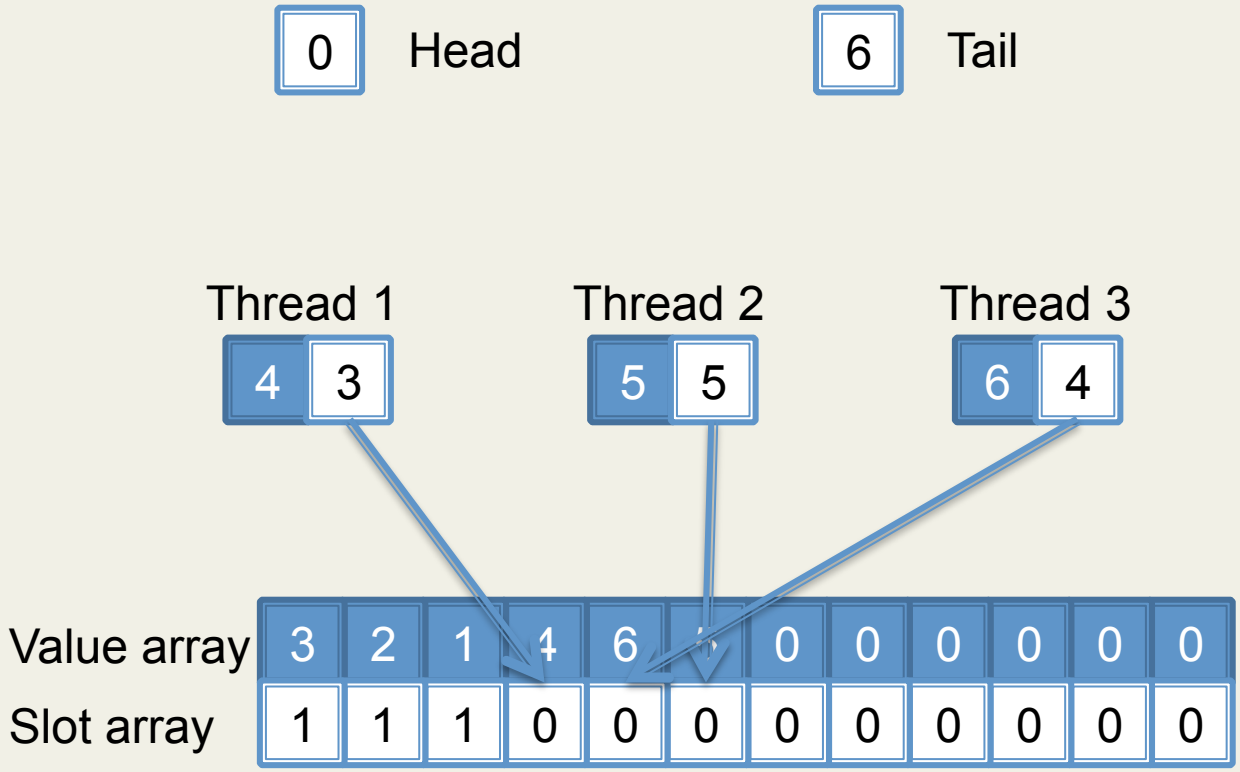
Our queue's blocking behavior: Enqueue example: Write values



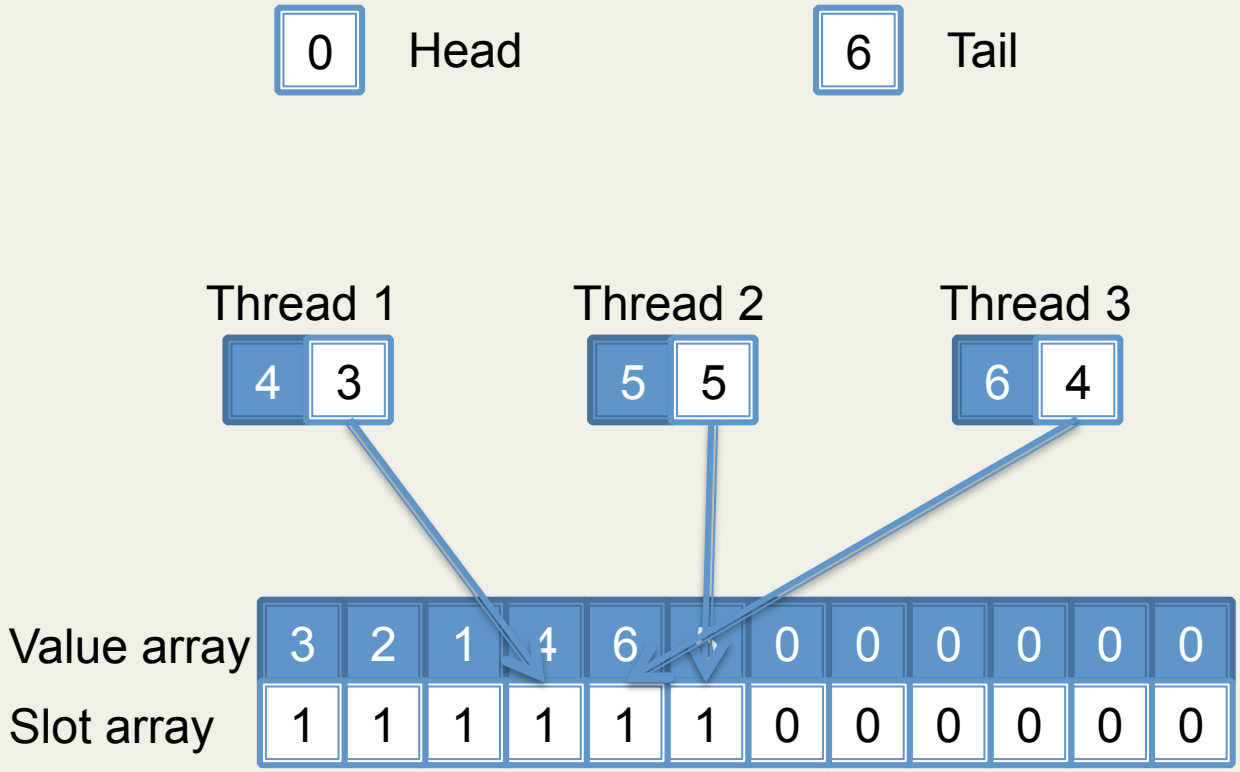
Our queue's blocking behavior: Enqueue example: Update slots



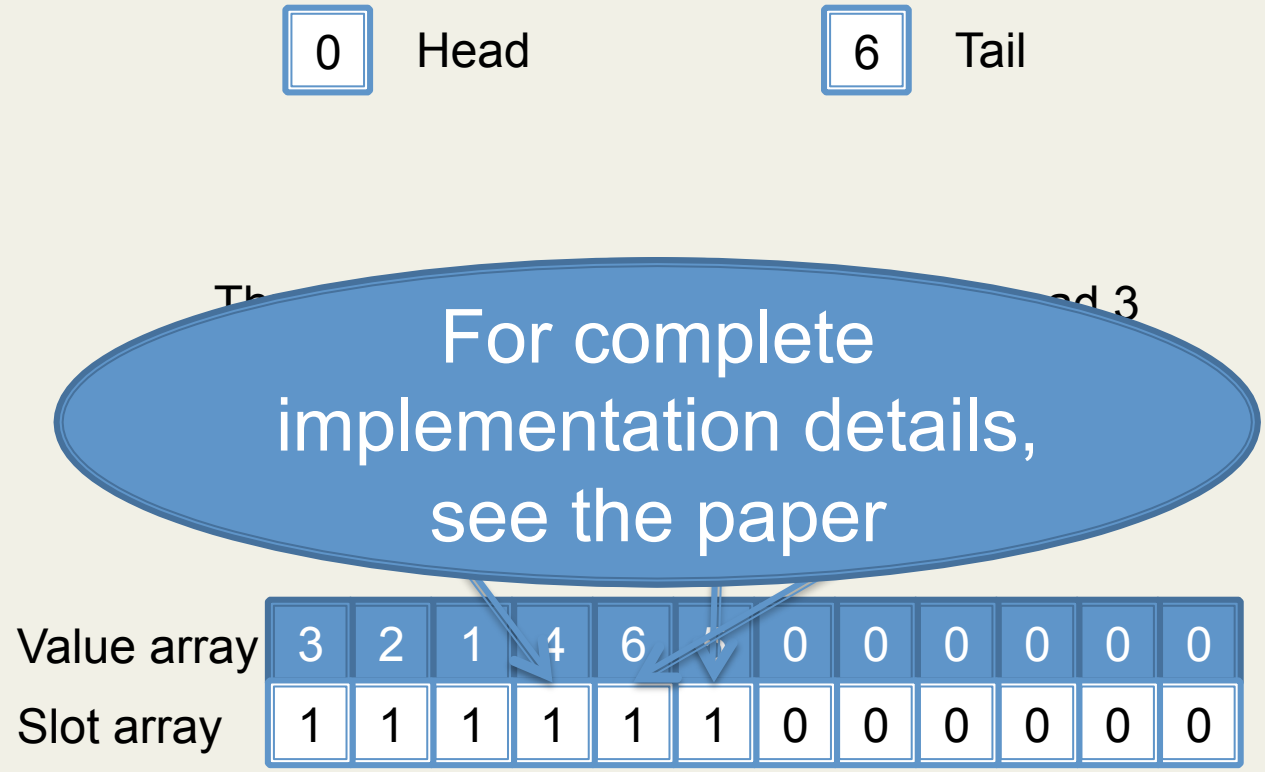
Our queue's blocking behavior: Enqueue example: Update slots



Our queue's blocking behavior: Enqueue example: Update slots



Our queue's blocking behavior: Enqueue example: Update slots



Outline

- Definitions and abstractions
- Building blocks: Evaluating atomic operations
- Queue types and modeling
- Our queue design
- Performance evaluation
- Conclusions

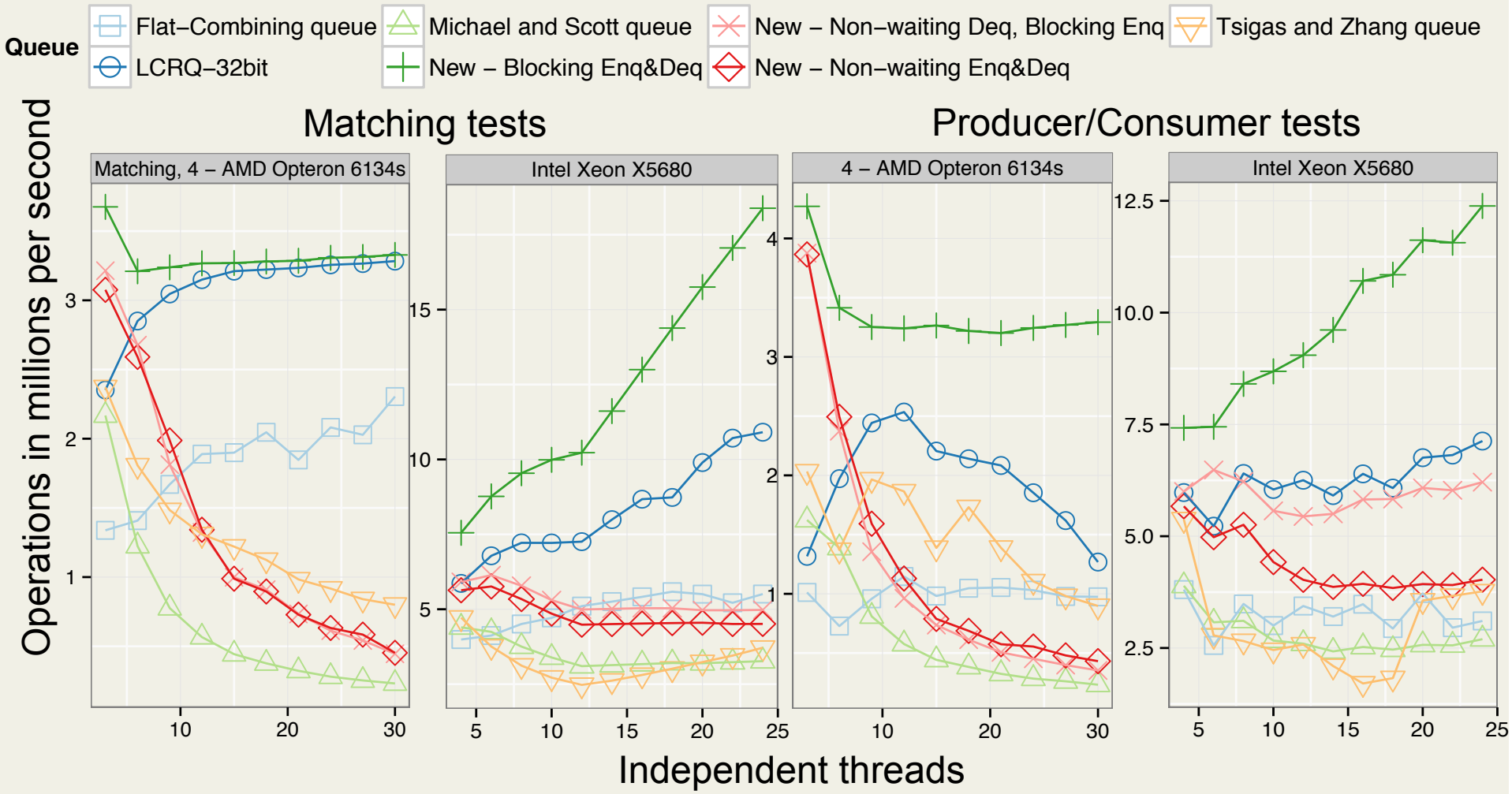
Evaluation: Queues Under Consideration

- Michael & Scott (MS) queue: Contended CAS
  - Storage: Unbounded linked list
  - Progress guarantee: lock-free
  - Coherence mechanism: CAS on head and tail
- Tsigas & Zhang (TZ) queue: Contended CAS
  - Storage: Bounded array
  - Progress guarantee: lock-free
  - Coherence mechanism: CAS on head and tail
- Flat-combining (FC) queue: Combining
  - Storage: Unbounded linked list
  - Progress guarantee: lock-free, *blocking*
  - Coherence mechanism: Serialization, single worker thread at a time
- Linked Concurrent Ring Queue (LCRQ): Un-contended CAS
  - Storage: Unbounded linked-list of blocking array-based queues
  - Progress guarantee: lock-free
  - Coherence mechanism: Double-wide CAS (precludes implementation on AMD GPUs)

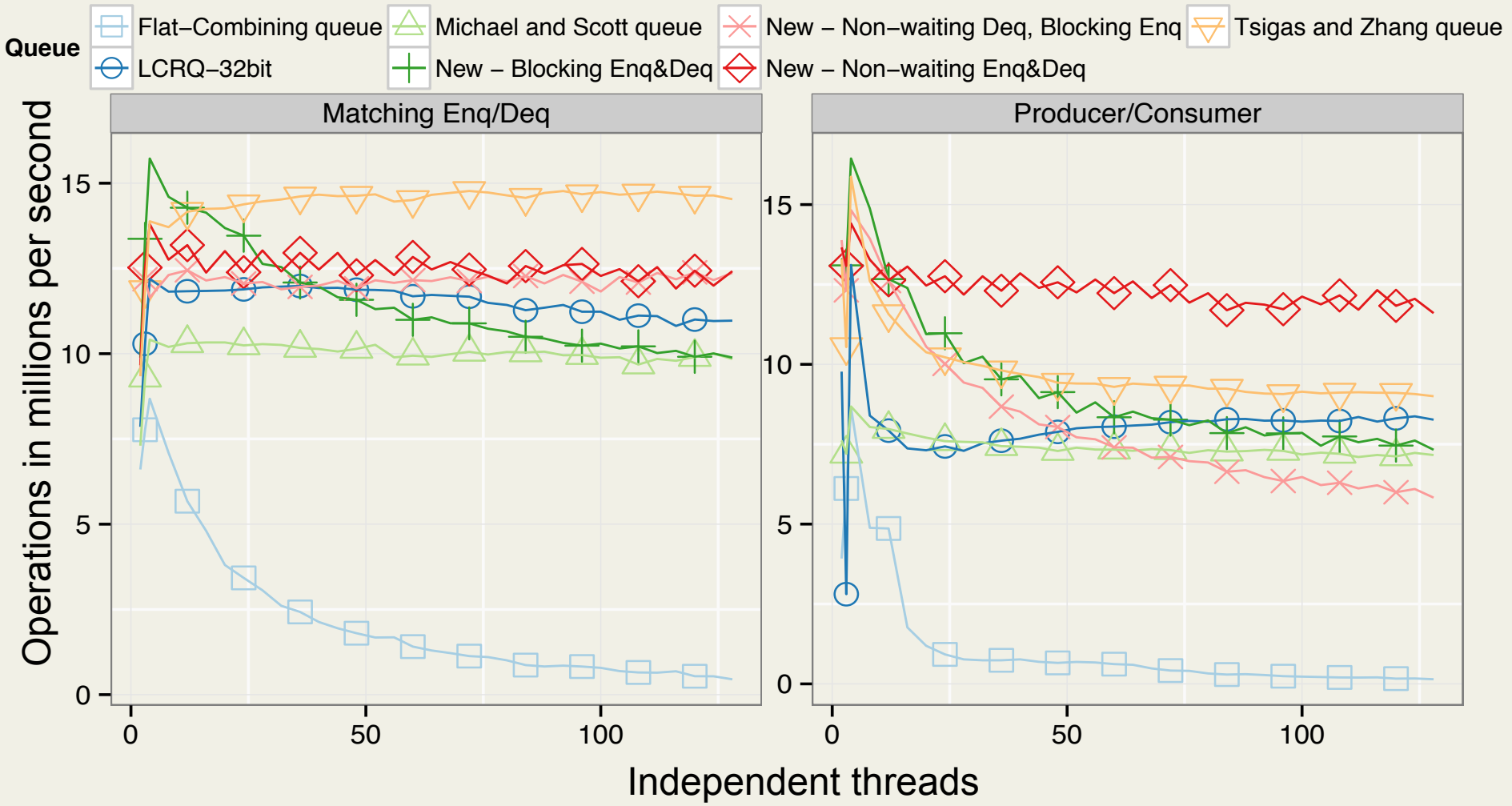
Evaluation: Test loops

- Matching enqueue/dequeue:
  - All threads:
    - Dequeue a value
    - Work on the value for 100 iterations
    - Enqueue the new value
    - Work out-of-band for 100 iterations
- Producer/consumer:
  - 25% of all threads:
    - Enqueue a value
    - Work for 100 iterations
  - The remaining 75%:
    - Dequeue a value
    - Work for 100 iterations
- All tests run for 5 seconds and are self-stopped on the device

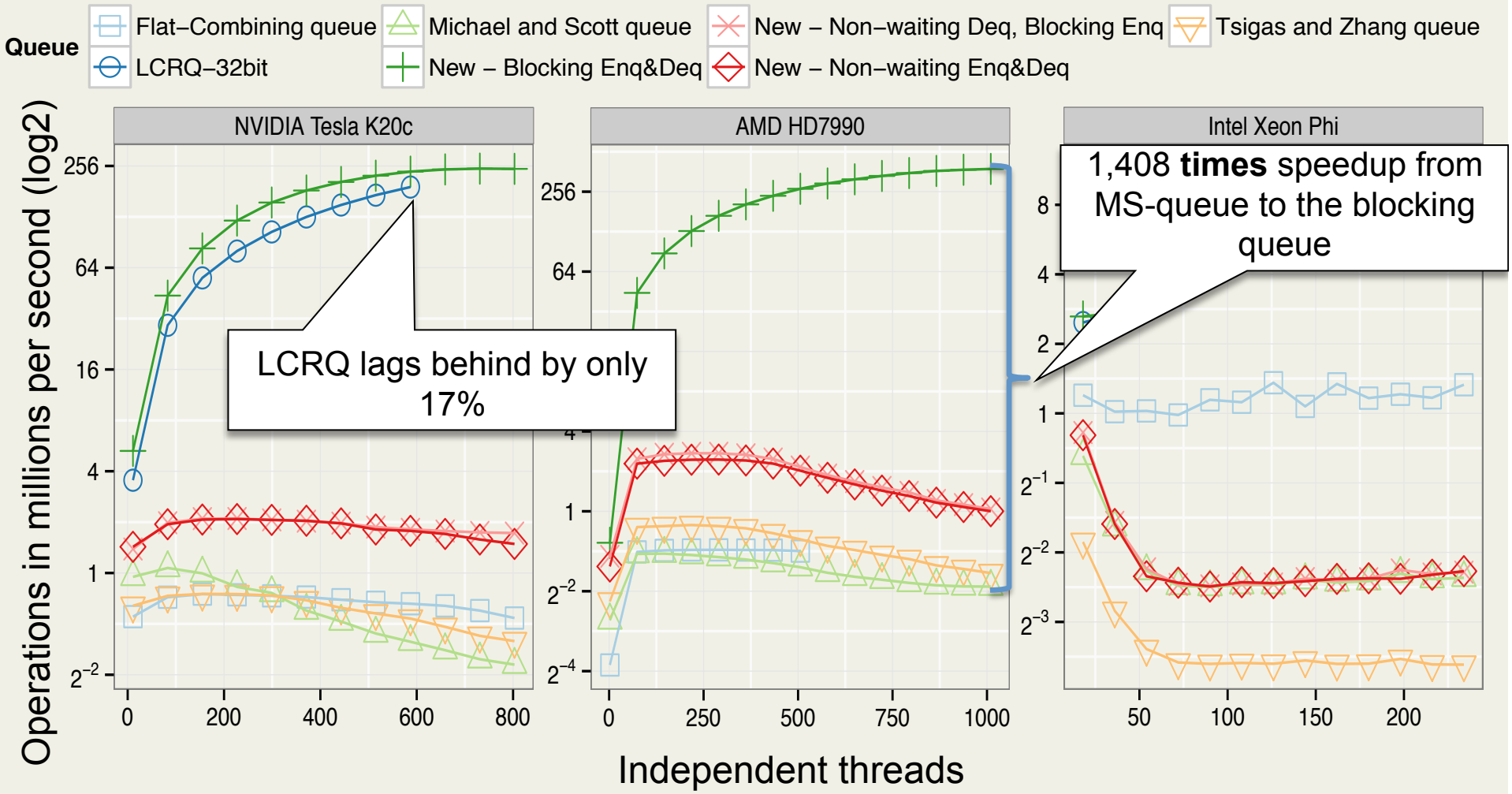
Evaluation: CPU performance



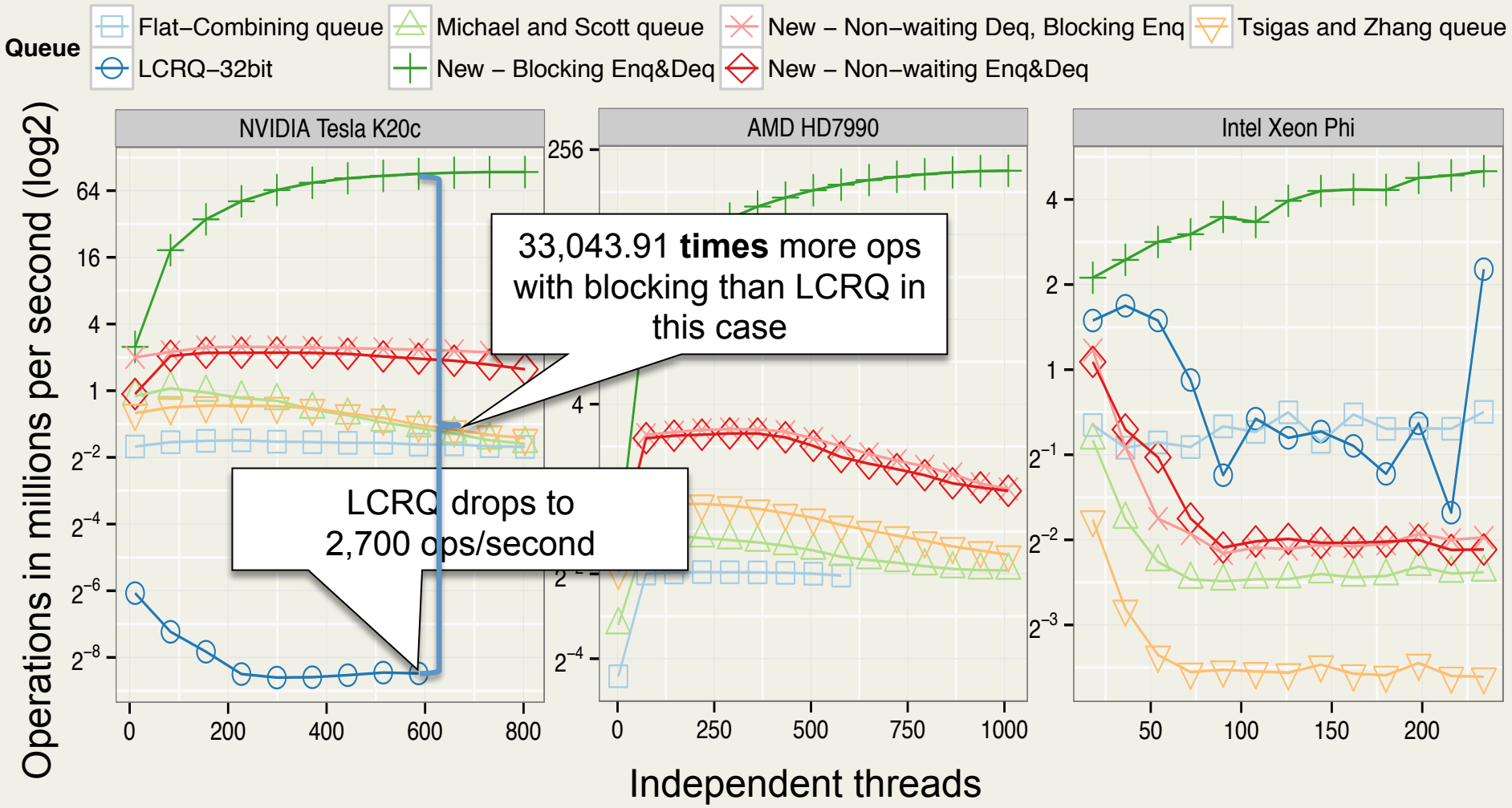
Evaluation:
CPU performance: Oversubscribing



Evaluation: Acc. performance: Current-Gen: Matching benchmark



Evaluation: Acc. performance: Current-Gen: Prod./Cons.



Conclusions

- Designing concurrent data-structures for **throughput** is important in modern architectures
- CAS can be **dangerous** with enough threads
- Our queue shows between a 1.5x and 1000x speedup over state of the practice for many-core architectures
- Allowing blocking can be beneficial!