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
Everything from cell phones

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

**Four Opteron 6134 CPUs**

<table>
<thead>
<tr>
<th>Independent threads</th>
<th>Operations per millisecond</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2500</td>
</tr>
<tr>
<td>3</td>
<td>2000</td>
</tr>
<tr>
<td>6</td>
<td>1500</td>
</tr>
<tr>
<td>9</td>
<td>1000</td>
</tr>
<tr>
<td>12</td>
<td>500</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
</tr>
</tbody>
</table>

**One NVIDIA K20c GPU**

<table>
<thead>
<tr>
<th>Independent threads</th>
<th>Operations per millisecond</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>2500</td>
</tr>
<tr>
<td>83</td>
<td>2000</td>
</tr>
<tr>
<td>155</td>
<td>1500</td>
</tr>
<tr>
<td>227</td>
<td>1000</td>
</tr>
<tr>
<td>299</td>
<td>500</td>
</tr>
<tr>
<td>371</td>
<td>1</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Device</th>
<th>Num. devices</th>
<th>Cores/device</th>
<th>Threads/core</th>
<th>Max. threads</th>
<th>Max. achieved</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Opteron 6134</td>
<td>4</td>
<td>8</td>
<td>1</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>AMD Opteron 6272</td>
<td>2</td>
<td>16</td>
<td>1</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Intel Xeon E5405</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Intel Xeon X5680</td>
<td>1</td>
<td>12</td>
<td>2</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>Intel Core i5-3400</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
## Experimental Setup: Hardware: GPUs/Co-processors

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

Operation: Attempted CAS, FAA, READ, Successful CAS, WRITE, XCHG

CAS: 1,478/ms
FAA: 859,524/ms
FAA is 581 times faster

Successful CAS rate decreases with number of threads!

scale up with the thread count

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
- Un-contended CAS
  - LCRQ
- Combining
  - FC queue
- FAA or blocking array
  - CB queue and our queue

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

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

\[
T_t = \frac{2}{(a_t + w_t \times 2) + (r_1 \times 2 + w_1)}
\]

\[
T_t = \frac{2}{(a_t + r_t + w_t) + (a_t + w_t \times 2)}
\]
Modeled queue throughput

- Combining queue performance is independent of thread count.
- Contended-CAS queue performance degrades as threads increase.
- Un-contended-CAS and FAA queues scale with additional threads.

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
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

Thread 1
4 0

Thread 2
5 0

Thread 3
6 0

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

Slot array
1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
Our queue’s blocking behavior:
Enqueue example: Get targets with FAA

0  Head

3  Tail

Thread 1
4  0

Thread 2
5  0

Thread 3
6  0

Value array

3  2  1  0  0  0  0  0  0  0  0  0  0  0

Slot array

1  1  1  0  0  0  0  0  0  0  0  0  0  0
Our queue’s blocking behavior: Enqueue example: Get targets with FAA

- Head: 0
- Tail: 6
- Slot array:
  - Thread 1: 4 3
  - Thread 2: 5 5
  - Thread 3: 6 4
- Value array:
  - 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0

- Slot array:
  - 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
Our queue’s blocking behavior: Enqueue example: Write values

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 3</td>
<td>5 5</td>
<td>6 4</td>
</tr>
</tbody>
</table>

Value array: 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0
Slot array: 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
Our queue’s blocking behavior: Enqueue example: Write values

Thread 1: 4 3
Thread 2: 5 5
Thread 3: 6 4

Slot array:

Value array:

0 1 2 3 4 5 6

Head 0 6 Tail
Our queue’s blocking behavior:
Enqueue example: Write values

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Value array: 3 2 1 4 6 5 0 0 0 0 0 0 0 0 0
Slot array: 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
Our queue’s blocking behavior: Enqueue example: Update slots

Thread 1
4 3

Thread 2
5 5

Thread 3
6 4

Value array: 3 2 1 4 6 5 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: Update slots

0  Head

6  Tail

Thread 1

4  3

Thread 2

5  5

Thread 3

6  4

Value array

3  2  1  4  6  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: Update slots

```
<table>
<thead>
<tr>
<th>Slot array</th>
<th>Value array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 0 0 0 0 0 0 0 0</td>
<td>3 2 1 4 6 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 3</td>
<td>5 5</td>
<td>6 4</td>
</tr>
</tbody>
</table>

Head: 0  Tail: 6
Our queue’s blocking behavior: Enqueue example: Update slots

For complete implementation details, see the paper.
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

Matching tests

Operations in millions per second

Independent threads

Matching, 4 – AMD Opteron 6134s
Intel Xeon X5680
4 – AMD Opteron 6134s
Intel Xeon X5680

Producer/Consumer tests

Operations in millions per second

Independent threads

Matching, 4 – AMD Opteron 6134s
Intel Xeon X5680
4 – AMD Opteron 6134s
Intel Xeon X5680
Evaluation:

CPU performance: Oversubscribing

Queues:
- Flat–Combining queue
- Michael and Scott queue
- New – Non–waiting Deq, Blocking Enq
- New – Non–waiting Enq&Deq
- Tsigas and Zhang queue

Matching Enq/Deq

Independent threads
Operations in millions per second

Producer/Consumer

Operations in millions per second

Lawrence Livermore National Laboratory
Evaluation: Acc. performance:
Current-Gen: Matching benchmark

- LCRQ lags behind by only 17%
- 1,408 times speedup from MS-queue to the blocking queue

Queue
- Flat–Combining queue
- Michael and Scott queue
- New – Non–waiting Deq, Blocking Enq
- New – Blocking Enq&Deq
- New – Non–waiting Enq&Deq
- LCRQ–32bit

- NVIDIA Tesla K20c
- AMD HD7990
- Intel Xeon Phi

Operations in millions per second (log2)

Independent threads

- NVIDIA GeForce GTX 280
- AMD HD5870
- NVIDIA Tesla C2070
- Intel Xeon Phi

33,043.91 times more ops with blocking than LCRQ in this case

LCRQ drops to 2,700 ops/second
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!