Scalable Lock Implementations: Evaluating GPU Concurrency Performance

Arnav Gattani and Akash Anickode

Project Motivation

- Due to thread parallelism we get **race conditions** when threads between different blocks may simultaneously modify a data structure.
- Program: thread A in block 1 and thread B in block 2 are asked to increment a counter, we expect the counter to equal 2.
    - However, threads A and B could read different values, update them locally, and then write an inconsistent value to global, resulting in a deviation.
- Use locks to choose when to block threads to prevent any deviation from expectation.
- We investigate the correctness and performance of different locking mechanisms when looking at concurrent updates, using the complexity of our algorithms as the benchmark (low vs high contentions).
- We look at tradeoffs such as thread fairness, global contention, and code complexity between different locking implementations

Locks Implemented

1. No-lock (control)
2. Basic spin lock
3. Ticket lock
4. Anderson’s array based queue-lock
5. MCS list-based queue-lock

Spin Lock

```
1    __device__ void lock( void ) {
2      while( atomicCAS( mutex, 0, 1 ) != 0 );
3(+)   __threadfence();}
4    __device__ void unlock( void ) {
5(+)   __threadfence();
6      atomicExch( mutex, 0 );}
```

Figure 2: CUDA spin lock of [38, p. 253] with added fences

Source: GPU concurrency Weak behaviours and programming assumptions paper

Ticket Lock

```
type ticketlock = record
    ticket : integer := 0
    users  : integer := 0
end record

// parameter my_ticket, below, points to a private variable
// in an enclosing scope

procedure acquire_lock(L : ^ticketlock, my_ticket : ^integer)
    my_ticket^ := atomic_add(&L->users, 1)    // get unique ticket number
    repeat while L->ticket != my_ticket^      // spin while the ticket number matches to acquire lock
        __threadfence()                       // fence till all memory operations complete
end procedure

procedure release_lock(L : ^ticketlock, my_ticket : ^integer)
    __threadfence()                           // fence till all memory operations complete before release lock
    atomic_add(&L->ticket, 1)                 // increment ticket to allow the next thread to acquire lock
end procedure
```

Anderson's array based queue-lock

```
type lock = record
    slots : array [0..numprocs -1] of (has_lock, must_wait)
        := (has_lock, must_wait, must_wait, ..., must_wait)
        // each element of slots should lie in a different memory module
        // or cache line
    next_slot : integer := 0

// parameter my_place, below, points to a private variable
// in an enclosing scope

procedure acquire_lock (L : ^lock, my_place : ^integer)
    my_place^ := fetch_and_increment (&L->next_slot)
        // returns old value
    if my_place^ mod numprocs = 0
        atomic_add (&L->next_slot, -numprocs)
        // avoid problems with overflow; return value ignored
    my_place^ := my_place^ mod numprocs
    repeat while L->slots[my_place^] = must_wait      // spin
    L->slots[my_place^] := must_wait                  // init for next time

procedure release_lock (L : ^lock, my_place : ^integer)
    L->slots[(my_place^ + 1) mod numprocs] := has_lock
```

Source: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors paper

MCS list-based queue-lock

```
type qnode = record
    next : ^qnode
    locked : Boolean
type lock = ^qnode      // initialized to nil

// parameter I, below, points to a qnode record allocated
// (in an enclosing scope) in shared memory locally-accessible
// to the invoking processor

procedure acquire_lock (L : ^lock, I : ^qnode)
    I->next := nil
    predecessor : ^qnode := fetch_and_store (L, I)
    if predecessor != nil      // queue was non-empty
        I->locked := true
        predecessor->next := I
        repeat while I->locked              // spin

procedure release_lock (L : ^lock, I: ^qnode)
    if I->next = nil        // no known successor
        if compare_and_store (L, I, nil)
            return
            // compare_and_store returns true iff it stored
        repeat while I->next = nil          // spin
    I->next->locked := false
```

Source: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors paper

Specs

- GPU: NVIDIA GTX Titan X (Maxwell)
- Number of Blocks: 512
- Number of Threads Per Block: 1024

Basic atomic addition benchmark (No RC)

Testing Methodology:

```
__global__ void noLockImpl(int *nblocks) {
    if (threadIdx.x == 0) {
        atomicAdd(nblocks, 1);
    }
}
```

```
__global__ void lockImpl(Lock lock, int *nblocks) {
    if (threadIdx.x == 0) {
        lock.lock();
        *nblocks = *nblocks + 1;
        lock.unlock();
    }
}
```

Results:

```
No lock counted 512 blocks in 0.027520 ms.
Spin lock counted 512 blocks in 0.300352 ms.
Ticket lock counted 512 blocks in 0.566112 ms.
```

Benchmark => No lock
Spin lock: 991.40% slower
Ticket lock: 1957.09% slower

Low Complexity Critical Section

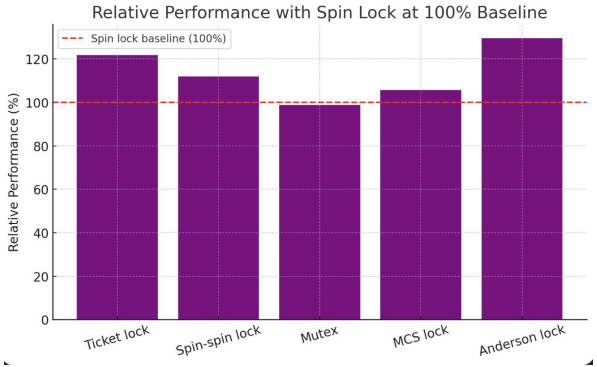
Testing Methodology:

```
__global__ void noLockImpl(int *nblocks, int* x) {
    if (threadIdx.x == 0) {
        atomicAdd(x, 1);
        for (int i = 0; i < 100; ++i) {
            atomicAdd(x, -1);
            atomicAdd(x, 1);
        }
        atomicAdd(nblocks, *x);
    }
}
```

```
__global__ void lockImpl(Lock lock, int *nblocks, int* x) {
    if (threadIdx.x == 0) {
        lock.lock();
        *x = *x + 1;
        for (int i = 0; i < 100; ++i) {
            atomicAdd(x, -1);
            atomicAdd(x, 1);
        }
        *nblocks = *nblocks + *x;
        lock.unlock();
    }
}
```

Results:

```
No lock counted 138999 blocks in 0.193536 ms.
Spin lock counted 131328 blocks in 1.221792 ms.
Ticket lock counted 131328 blocks in 1.489152 ms.
Spin spin lock counted 131328 blocks in 1.368384 ms.
Mutex counted 131328 blocks in 1.206784 ms.
MCS lock counted 131328 blocks in 1.291488 ms.
Anderson lock counted 131328 blocks in 1.583104 ms.
```



Benchmark => Spin lock
Ticket lock: +21.88% (slower)
Spin-spin lock: +12.00% (slower)
Mutex: -1.23% (slightly faster)
MCS lock: +5.70% (slower)
Anderson lock: +29.57% (slower)

High Complexity Critical Section

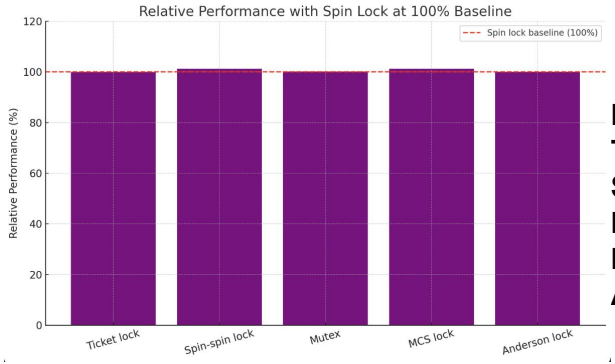
Testing Methodology:

```
__global__ void noLockImpl(int *nblocks, int* x) {
    if (threadIdx.x == 0) {
        atomicAdd(x, 1);
        for (int i = 0; i < 100000; ++i) {
            atomicAdd(x, -1);
            atomicAdd(x, 1);
        }
        atomicAdd(nblocks, *x);
    }
}
```

```
__global__ void lockImpl(Lock lock, int *nblocks, int* x) {
    if (threadIdx.x == 0) {
        lock.lock();
        *x = *x + 1;
        for (int i = 0; i < 100000; ++i) {
            atomicAdd(x, -1);
            atomicAdd(x, 1);
        }
        *nblocks = *nblocks + *x;
        lock.unlock();
    }
}
```

Results:

```
No lock counted 139700 blocks in 117.942879 ms.
Spin lock counted 131328 blocks in 696.508911 ms.
Ticket lock counted 131328 blocks in 695.908875 ms.
Spin spin lock counted 131328 blocks in 705.346313 ms.
Mutex counted 131328 blocks in 697.673828 ms.
MCS lock counted 131328 blocks in 705.135559 ms.
Anderson lock counted 131328 blocks in 696.266724 ms.
```



Benchmark => Spin lock
Ticket lock: -0.09% (slightly faster)
Spin-spin lock: +1.27% (slightly slower)
Mutex: +0.17% (slightly slower)
MCS lock: +1.24% (slightly slower)
Anderson lock: -0.03% (slightly faster)

What is the best lock?

- Spin lock: Low fairness, high global contention, low scalability
- Ticket lock: High fairness, high global contention, low scalability
- Anderson lock: Medium fairness, medium global contention, medium scalability
- MCS lock: High fairness, low global contention, high scalability

Other Ideas

- Testing lock performance on newer GPU architectures
- Adding stressors such as bank conflicts or memory stress
- Testing more advanced locking mechanisms
  - IBM patented K42 MCS Lock
  - Scalable tree based barriers

References

1. Mellor-Crummey, John and Scott, Michael, “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors.” *Scalable Synchronization*, https://www.cs.rochester.edu/u/scott/papers/1991_TOCS_synch.pdf. Accessed 9 Dec. 2024.
2. “Spinlocks and Read-Write Locks.” *The University of Texas at Austin*, www.cs.utexas.edu/~pingali/CS378/2015sp/lectures/Spinlocks%20and%20Read-Write%20Locks.htm. Accessed 9 Dec. 2024.
3. Grote, Phillip. “Lock-Based Data Structures on GPUs with Independent Thread Scheduling.” *Technische Universitat Berlin*, www.clemenslutz.com/pdfs/bsc_thesis_phillip_grote.pdf. Accessed 10 Dec. 2024.
4. Landau, Will. *Cuda C: Race Conditions, Atomics, Locks, Mutex, and Warps*, Iowa State University, 21 Oct. 2013, wlandau.github.io/gpu/lectures/cudac-atomics/cudac-atomics.pdf.
5. Alglave, Jade, et al. “GPU Concurrency Weak Behaviours and Programming Assumptions.” *ACM Digital Library*, doi.org/10.1145/2775054.2694391. Accessed 10 Dec. 2024.