HW2: CUDA Synchronization Primitives

CIS 601

31 March 2016

WarpLevelLock

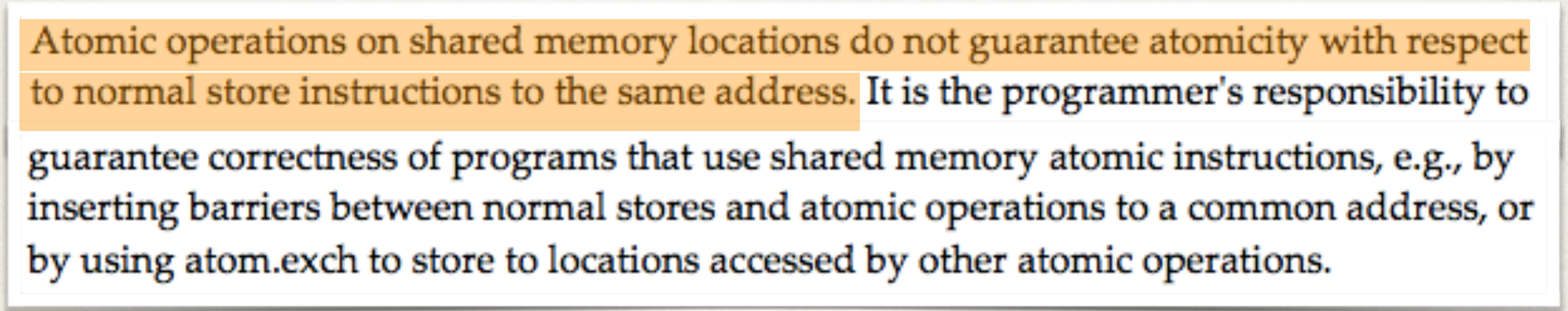
- fences go right next to the critical section

```
__device__ void lock() {
  while (atomicCAS((int*)&theLock, LOCK_FREE,LOCK_HELD)
  ;
  __threadfence();
}

__device__ void unlock() {
  __threadfence();
  atomicExch((int*)&theLock, LOCK_FREE);
}
```

WarpLevelLock

- always use atomic operations on the lock location



Atomic operations on shared memory locations do not guarantee atomicity with respect to normal store instructions to the same address. It is the programmer's responsibility to guarantee correctness of programs that use shared memory atomic instructions, e.g., by inserting barriers between normal stores and atomic operations to a common address, or by using atom.exch to store to locations accessed by other atomic operations.

ThreadLevelLock

- beware assumptions about which threads are active

```
for (int w = 0; w < warpSize; w++) {
  if (w == threadIdx.x % warpSize) {
    warpLock.lock();
    fun();
    warpLock.unlock();
  }
}
```

SpinBarrier pseudocode

```
shared count : integer := P
shared sense : Boolean := true
processor private local_sense : Boolean := true

procedure central_barrier
  // each proc toggles its own sense
  local_sense := not local_sense
  if fetch_and_decrement(&count) = 1:
    count := P
    // last proc toggles global sense
    sense := local_sense
  else:
    repeat until sense = local_sense
```

How should `local_sense` be implemented?

Common SpinBarrier issues

- extraneous atomics, __threadfences
- insufficient locking

```
__device__ virtual void wait() {
    warpLock.lock();
    bool localSense = !sense;
    arrived++;
    if (arrived >= m_expected) {
        arrived = 0;
        sense = localSense;
    }
    warpLock.unlock();

    while (sense != localSense);
}
```

SpinBarrier

```
__device__ virtual void wait() {
  lock();
  bool localSense = !sense;
  arrived += 1;

  if (arrived == m_expected) {
    arrived = 0;
    sense = localSense;
    unlock();
    return;
  } else {
    unlock();
    while (1) {
      lock();
      if (sense == localSense) {
        unlock();
        return;
      }
      unlock();
    }
  }
}
```

2-Level Barrier (1 __syncthreads)

- no data races here, but doesn't synchronize client code as expected

```
__device__ virtual void wait() {
  if (threadIdx.x == 0) {
    SpinBarrier::wait();
  }
  __syncthreads();
}
```

2-Level Barrier (2 __syncthreads)

- works as expected

```
__device__ virtual void wait() {
  __syncthreads();
  if (threadIdx.x == 0) {
    SpinBarrier::wait();
  }
  __syncthreads();
}
```

GRace

A Low-Overhead Mechanism for Detecting Data Races in GPU Programs

Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal

GRace overview

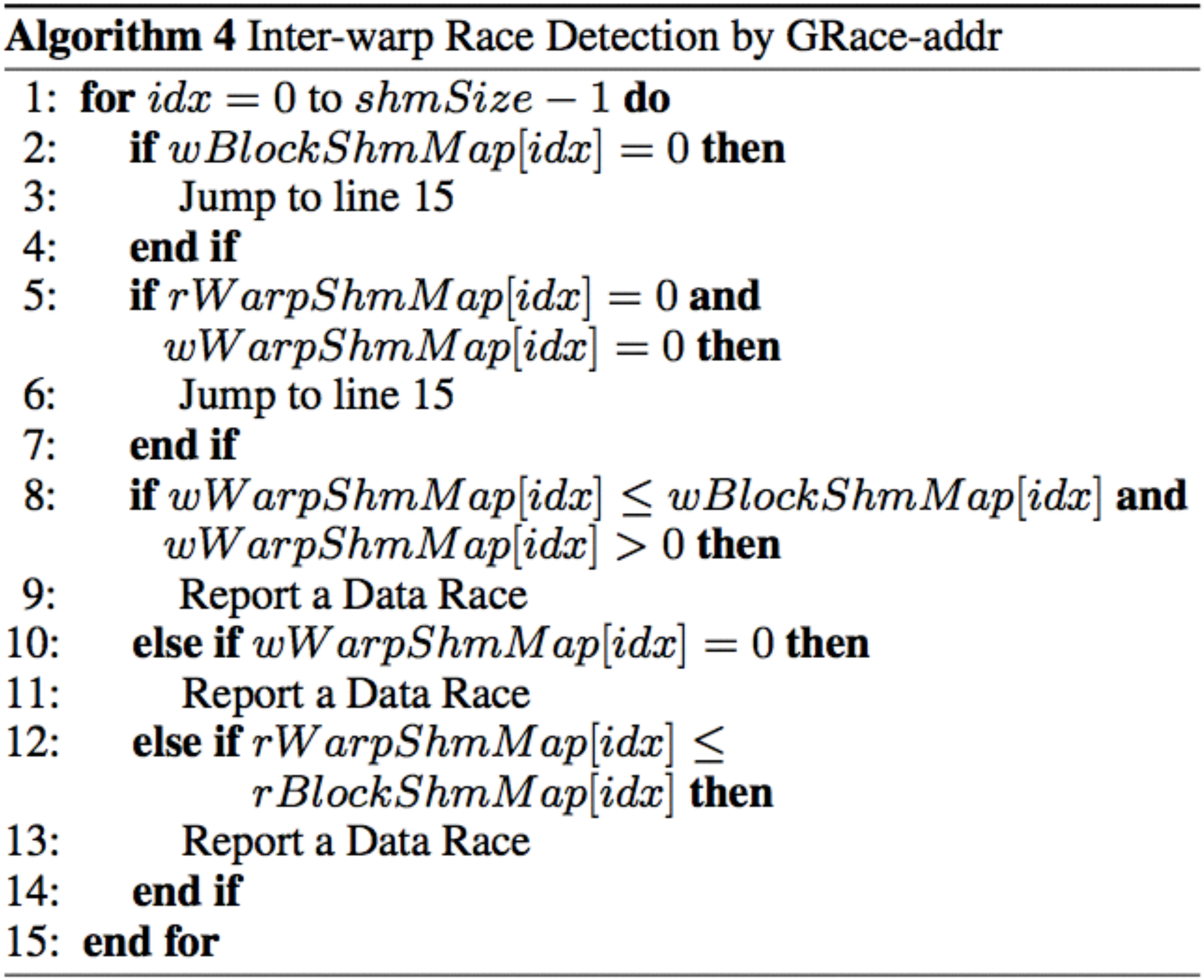
- detects **shared memory** races **between __syncthreads calls**
- ignores other synchronization (atomics, fences) and device memory
- assumes consistent access granularity?

Algorithm 2 Intra-warp Race Detection

```
1: if $accessType$ is $read$ then
2:     Stop execution of this algorithm
3: end if
4: for $targetIdx = 0$ to $warpSize - 1$ do
5:     $sourceIdx \leftarrow tid \% warpSize$
6:     if $sourceIdx \neq targetIdx$ then
7:         if $warpTbl[sourceIdx] =$
             $warpTbl[targetIdx]$ then
8:             Report a Data Race
9:         end if
10:    end if
11: end for
```

Algorithm 3 Inter-warp Race Detection by GRace-stmt

```
1: for $stmtIdx1 = 0$ to $maxStmtNum - 1$ do
2:     for $stmtIdx2 = stmtIdx1 + 1$ to $maxStmtNum$ do
3:         if $BlkStmtTbl[stmtIdx1].warpID =$
             $BlkStmtTbl[stmtIdx2].warpID$ then
4:             Jump to line 15
5:         end if
6:         if $BlkStmtTbl[stmtIdx1].accessType$ is $read$ and
             $BlkStmtTbl[stmtIdx2].accessType$ is $read$ then
7:             Jump to line 15
8:         end if
9:         for $targetIdx = 0$ to $warpSize - 1$ do
10:            $sourceIdx \leftarrow tid \% warpSize$
11:            if $BlkStmtTbl[stmtIdx1][sourceIdx] =$
                 $BlkStmtTbl[stmtIdx2][targetIdx]$ then
12:                Report a Data Race
13:            end if
14:        end for
15:    end for
16: end for
```



Algorithm 4 Inter-warp Race Detection by GRace-addr

```
1: for $idx = 0$ to $shmSize - 1$ do
2:     if $wBlockShmMap[idx] = 0$ then
3:         Jump to line 15
4:     end if
5:     if $rWarpShmMap[idx] = 0$ and
         $wWarpShmMap[idx] = 0$ then
6:         Jump to line 15
7:     end if
8:     if $wWarpShmMap[idx] \leq wBlockShmMap[idx]$ and
         $wWarpShmMap[idx] > 0$ then
9:         Report a Data Race
10:    else if $wWarpShmMap[idx] = 0$ then
11:        Report a Data Race
12:    else if $rWarpShmMap[idx] \leq$
                $rBlockShmMap[idx]$ then
13:        Report a Data Race
14:    end if
15: end for
```

Background

- Where do the race detector programs that implement mechanisms like GRace run? Do they run on CPU in parallel with the GPU kernel being executed?
- Can you go over the layout of warpTable? Is it indexed/keyed by the address of the instruction performing a memory operation, or by (threadId, memoryAddress)?

False Positives

- Shouldn't they also use comparisons that show the possible false positives in GRace?

Intra-warp races

- In the intra-warp race detection algorithm, they completely disregard read memory accesses and are only concerned with write-write data race. Why are they not dealing with read-write data race? How will intra-warp detection algorithm work if we take into account control flow divergence?

Usage Model

- How do you create "bug-triggering inputs and parameters" if you don't really know about data races on GPUs and want to use GRace to help you finding them?
- Why do both GRace and cuda-memcheck only consider data races in shared memory accesses? Is it that much more work to detect them in global accesses? These tools seem to be targeted at already "tuned" code where the assumption is that you're heavily using shared memory (and that the global sync portion is relatively simple/race-free).

Static Analysis

- How does the static analyzer handle aliasing? Will aliasing cause the static analyzer to miss races?

- The static analysis component of GRace appears to be limited to cases where memory addresses are defined in terms of tid (the thread ID) and i (some loop index). There exist more sophisticated data-flow analysis techniques that would enable more sophisticated reasoning about possible values of memory addresses in the code. (One heavy-duty example would be a framework like KLEE, which is specific to LLVM.) Do you know why the authors didn't use these techniques? Do you think it would be possible to implement them without substantial refactoring of their static analysis algorithm?

Scalability

- Both the schemes GRace-stmt (device memory scope) and GRace-addr (shared memory scope) perform well only with a small number of statements being instrumented and executed. Are there any other tools that are more scalable and work well larger number of statements?
- Can this be sped up by adding any hardware features?
- Are there any thread count, memory size constraints?

Evaluation

- What is EM clustering?
- The paper only shows that GRace-stmt and GRace-addr for only 3 applications out of which B-tool (the previous work) could not be evaluated for 'scan' because of new hardware and software. Isn't this a weak proof that GRace is better in performance than B-tool?
- Are three kernels really sufficient to test GRace? Why did they pick these kernels in particular and shouldn't a more comprehensive battery of smaller tests be used?