OMG does locking work in CUDA??!!??!
What to do for variables inside the critical section?
volatile PTX

- `ld.volatile`, `st.volatile` do not permit cache operations
- all operations go straight to global memory
- critical section will operate correctly
non-volatile PTX

- ld caches at all levels (L1, L2) by default
- st invalidates L1 copy and updates L2 by default
  - so d_counter is cached in the L1 only briefly
  - not safe in general!
BB2_2:

mov.u64    %rd1, d_mutex;
atom.global.cas.b32  %r5, [%rd1], 0, 1;
setp.ne.s32 %p1, %r5, 0;
@%p1 bra   BB2_2;

membar.gl;

ld.global.u32  %r6, [d_counter];
add.s32       %r7, %r6, 1;
st.global.u32  [d_counter], %r7;

ld.volatile.global.u32  %r8, [dv_counter];
add.s32       %r9, %r8, 1;
st.volatile.global.u32  [dv_counter], %r9;

membar.gl;
atom.global.exch.b32  %r10, [%rd1], 0;
add.s32       %r11, %r11, 1;
setp.lt.s32  %p2, %r11, 100;
@%p2 bra   BB2_1;
what about __threadfence?

- “You can force the L1 cache to flush back up the memory hierarchy using the appropriate __threadfence_*() function. __threadfence_block() requires that all previous writes have been flushed to shared memory and/or the L1. __threadfence() additionally forces global memory writes to be visible to all blocks, and so must flush writes up to the L2. Finally, __threadfence_system() flushes up to the host level for mapped memory.”
init: \( \begin{align*} \text{(global x=0)} \\ \text{(global y=0)} \end{align*} \) | final: \( r_1=1 \land r_2=0 \) | threads: inter-CTA
---
0.1 | \text{st.cg} [x],1 | 1.1 | \text{ld.ca r1,}[y] \\
0.2 | \text{fence} | 1.2 | \text{fence} \\
0.3 | \text{st.cg} [y],1 | 1.3 | \text{ld.ca r2,}[x]

<table>
<thead>
<tr>
<th>obs/100k</th>
<th>fence</th>
<th>GTX5</th>
<th>TesC</th>
<th>GTX6</th>
<th>Titan</th>
<th>GTX7</th>
</tr>
</thead>
<tbody>
<tr>
<td>no-op</td>
<td>fence</td>
<td>4979</td>
<td>10581</td>
<td>3635</td>
<td>6011</td>
<td>3</td>
</tr>
<tr>
<td>membar.cta</td>
<td>no-op</td>
<td>0</td>
<td>308</td>
<td>14</td>
<td>1696</td>
<td>0</td>
</tr>
<tr>
<td>membar.gl</td>
<td>membar.cta</td>
<td>0</td>
<td>187</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>membar.sys</td>
<td>membar.gl</td>
<td>0</td>
<td>162</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 3:** PTX \( \text{mp} \) w/ L1 cache operators (\( \text{mp-L1} \))

**what is \( \_\_\_\_\text{threadfence} \) (aka membar.gl) doing on Tesla C2075??!!??

*from “GPU Concurrency: Weak behaviours” paper*
**Table 5.** We ignore the additional fences (lines 0.2 and 1.3) for simplicity. Communication with the authors confirms that the weak behaviour is unintentional. Fig. 2 shows the weak behaviour of an OpenCL version of DLB-MP 

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>st.cg [x],1</td>
<td>* 1.1</td>
<td>atom.cas r1,[m],0,1</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2</td>
<td>membar.gl</td>
<td>5 1.2</td>
<td>setp.eq r2,r1,0</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.3</td>
<td>atom.exch r0,[m],0</td>
<td>6 1.3</td>
<td>@r1 membar.gl</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*original line in Fig. 2</td>
</tr>
</tbody>
</table>

*Threadfence still necessary even if L1 isn’t used* from “GPU Concurrency: Weak behaviours” paper

### Figure 9: PTX compare-and-swap spin lock (cas-sl)
what gets cached where?

<table>
<thead>
<tr>
<th>CUDA compute capability</th>
<th>default caching policy</th>
<th>opt-in to L1 caching?</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.x</td>
<td>ca (L1 &amp; L2)</td>
<td>n/a</td>
</tr>
<tr>
<td>3.x</td>
<td>cg (L2 only)</td>
<td>no</td>
</tr>
<tr>
<td>3.5, 3.7</td>
<td>cg (L2 only)</td>
<td>yes</td>
</tr>
<tr>
<td>5.x</td>
<td>cg (L2 only)</td>
<td>yes</td>
</tr>
</tbody>
</table>
Conclusions

• volatile seems safe but unnecessarily expensive as it avoids L1 and L2 caching

• NVCC by default caches only at L2 these days (CC ≥ 3.x)

  • HW2 locks therefore seem ok (for CC ≥ 3.x)

  • “-Xptxas -dlcm=ca” opts-in to L1 caching

  • no difference in code on AWS instance :-/

• __threadfence() still necessary to prevent other reorderings
What is the L1 good for?

- the L1 by default is used only for local memory and read-only globals
  - probably due to lack of coherence
  - need to opt-in to get more utility out of L1
- shared memory is the easiest, fastest writable memory level
GMRace
Mai Zheng, Vignesh T. Ravi, Feng Qin and Gagan Agrawal
How is GMRace different from GRace?
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 ← 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 1. Interwarp Race Detection by GMRace-stmt.

1:  for stmtIdx1 = tid 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 17
5:      end if
6:      if BlkStmtTbl[stmtIdx1].accessType is read and
        BlkStmtTbl[stmtIdx2].accessType is read
7:        then
8:          Jump to line 17
9:      end if
10:     for targetIdx = 0 to warpSize – 1 do
11:       for sourceIdx = 0 to warpSize – 1 do
12:         if BlkStmtTbl[stmtIdx1][sourceIdx] =
           BlkStmtTbl[stmtIdx2][targetIdx] then
13:           Report a Data Race
14:         end if
15:       end for
16:     end for
17:     stmtIdx1+ = threadNum
18:   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
Algorithm 2. Interwarp Race Detection by GMRace-flag.
1: for idx = 0 to shmSize - 1 do
2: writeSum ← 0
3: readSum ← 0
4: for warpID = 0 to warpID = warpNum - 1 do
5: writeSum+ = wWarpShmMaps[warpID][idx]
6: readSum+ = rWarpShmMaps[warpID][idx]
7: end for
8: if writeSum = 0 then
9: Jump to line 25
10: else if writeSum >= 2 then
11: Report Data Races
12: else if writeSum = 1 then
13: if readSum = 0 then
14: Jump to line 25
15: else if readSum >= 2 then
16: Report Data Races
17: else if readSum = 1 then
18: wWarpID = getWarpIDofNonZeroFlag (wWarpShmMaps, idx)
19: rWarpID = getWarpIDofNonZeroFlag (rWarpShmMaps, idx)
20: if wWarpID! = rWarpID then
21: Report a Data Race
22: end if
23: end if
24: end if
25: end for
launch 8 blocks with 256 threads per block. For blocks with 320 threads per block. For parameters or inputs to trigger the data races. For race conditions. For the kernel functions, we use bug-triggering pa-
does not report pairs of statements, threads, or warps involved in address where other threads have conflicting accesses before. It ports a data race whenever the current thread accesses a memory reported by the tool. Unlike GRace-stmt or GRace-addr, B-tool re-
of threads or warps. For B-tool, we present the number of data races addition, and the number of false positives within the reported pairs racing statements, the number of memory addresses involved in addr, we measure four metrics, including the number of pairs of 2.1 Source Code. We evaluate three schemes, including GRace-stmt, GRace-addr, and the previous work B-tool [7]. For GRace-stmt and GRace-addr, there are no false positives for both GRace-stmt and effectively detect data races. For example, among the reported data races, there are no false positives for both GRace-stmt and justified by the fact that they are generated by the tool. As shown in Table 1, B-tool generates 45,870 false positives, while GRace-stmt and GRace-addr do not generate any false positives.

Table 1. Experimental Results

<table>
<thead>
<tr>
<th>Scheme</th>
<th>R-Stmt#</th>
<th>R-Mem#</th>
<th>R-Thd#</th>
<th>FP#</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRace-stmt</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GRace-addr</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B-tool</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>45,870</td>
</tr>
</tbody>
</table>

We measure the execution time for two applications: co-cluster and em. The y-axis is on a logarithmic scale. Figure 5 shows that GRace-addr and GRace-stmt incur lower runtime overhead than B-tool. For co-cluster, GRace-stmt and GRace-addr have a runtime overhead of 1.1 KB and 514 MB, respectively, while B-tool has a runtime overhead of 9 MB. For em, the runtime overhead is 1.1 KB, 257 MB, and 257 MB for GRace-stmt, GRace-addr, and B-tool, respectively.

**Figure 5.** Runtime overhead of GRace. Note that the y-axis is on a logarithmic scale.
Fig. 5. Runtime overhead of different schemes of GMRace and GRace. Note that the $y$-axis is on a logarithmic scale.
kinds of races

• Is it possible for GMRace to have false positives?

• Can indirect memory accesses cause data races on GPUs?

• The LDetector paper says that intra-warp races are "trivial" and mostly decidable at compile-time, so they leave it out. However, GMrace treats intra-warp detection explicitly and reuses the warp tables for inter-warp detection. What is the real importance of intra-warp detection?
We have read a few papers on data race detection but none of them talk about correcting these data races. In general, at the point where a data race is detected, does the system take a checkpoint and rollback/replay to modify the thread scheduling to avoid occurrence of the data-race in the re-execution or does it just abort?
performance

• How is it that inserted code by the dynamic checker affects register assignment and, although it doesn’t affect the detection capabilities, does it affect performance?

• What is the overhead of the static analysis?
benchmarks

• Co-clustering and EM-clustering keep showing up as benchmarks in these race detector papers. Why are these algorithms particularly important benchmarks?

• I noticed that most of the race detectors seem to have very few benchmarks compared to the other papers we looked at, is there a reason for this?