HW2: CUDA Synchronization

Take 2

OMG does locking work in CUDA??!!??!

```
 9  __device__ volatile unsigned d_mutex = 0;
10
11  __device__ void jldlock() {
12      unsigned u = d_mutex;
13      while (atomicCAS((unsigned*)&d_mutex, 0, 1) != 0) {}
14      __threadfence();
15  }
16
17  __device__ void jldunlock() {
18      __threadfence();
19      atomicExch((unsigned*)&d_mutex, 0);
20  }
21
22  __device__ unsigned d_counter = 0;
23  __device__ volatile unsigned dv_counter = 0;
24
25  __global__ void useLocks() {
26      for (int i = 0; i < 100; i++) {
27          jldlock();
28          d_counter++;
29          dv_counter++;
30          jldunlock();
31      }
32  }
```

What to do for variables inside the critical section?

volatile PTX

```
322   //C:/Users/Administrator/Source/Repos/cis601/PtxSandbox/PtxSandbox/lock.cu:32      dv_counter++;
323       .loc    1 32 1
324       mov.u64     %rd3, dv_counter;
325       cvta.global.u64     %rd4, %rd3;
326       ld.volatile.u32     %r7, [%rd4];
327       add.s32     %r8, %r7, 1;
328       st.volatile.u32     [%rd4], %r8;
329
```

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

non-volatile PTX

```
314   //C:/Users/Administrator/Source/Repos/cis601/PtxSandbox/PtxSandbox/lock.cu:31       d_counter++;
315       .loc    1 31 1
316       mov.u64     %rd1, d_counter;
317       cvta.global.u64     %rd2, %rd1;
318       ld.u32  %r5, [%rd2];
319       add.s32     %r6, %r5, 1;
320       st.u32  [%rd2], %r6;
```

- 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!

Release PTX

```
73  BB2_2:
74      mov.u64     %rd1, d_mutex;
75      atom.global.cas.b32     %r5, [%rd1], 0, 1;
76      setp.ne.s32 %p1, %r5, 0;
77      @%p1 bra    BB2_2;
78
79      membar.gl;
80      ld.global.u32   %r6, [d_counter];
81      add.s32     %r7, %r6, 1;
82      st.global.u32   [d_counter], %r7;
83      ld.volatile.global.u32  %r8, [dv_counter];
84      add.s32     %r9, %r8, 1;
85      st.volatile.global.u32  [dv_counter], %r9;
86      membar.gl;
87      atom.global.exch.b32    %r10, [%rd1], 0;
88      add.s32     %r11, %r11, 1;
89      setp.lt.s32 %p2, %r11, 100;
90      @%p2 bra    BB2_1;
```

acquire (lines 73–79)

non-volatile (lines 80–82)

volatile (lines 83–85)

unlock (lines 86–87)

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.”
  — siebert, https://devtalk.nvidia.com/default/topic/489987/l1-cache-l2-cache-and-shared-memory-in-fermi/

init: $\begin{pmatrix} \texttt{global x=0} \\ \texttt{global y=0} \end{pmatrix}$ **final:** `r1=1` $\wedge$ `r2=0` **threads:** inter-CTA

0.1	`st.cg [x],1`	1.1	`ld.ca r1,[y]`
0.2	*fence*	1.2	*fence*
0.3	`st.cg [y],1`	1.3	`ld.ca r2,[x]`

obs/100k	*fence*	GTX5	TesC	GTX6	Titan	GTX7
	`no-op`	4979	10581	3635	6011	3
	`membar.cta`	0	308	14	1696	0
	`membar.gl`	0	187	0	0	0
	`membar.sys`	0	162	0	0	0

Figure 3: PTX **mp** w/ L1 cache operators (**mp-L1**)

what is __threadfence (aka membar.gl) doing on Tesla C2075??!!??

from "GPU Concurrency: Weak behaviours" paper

init: $\begin{pmatrix}\texttt{global x=0}\\ \texttt{global m=1}\end{pmatrix}$ **final:** `r1=0` ∧ `r3=0` **threads:** inter-CTA

0.1	`st.cg [x],1`	*	1.1	`atom.cas r1,[m],0,1`	*2
0.2(+)	`membar.gl`	5	1.2	`setp.eq r2,r1,0`	2
0.3	`atom.exch r0,[m],0`	6	1.3(+)	`@r1 membar.gl`	3
			1.4	`@r1 ld.cg r3,[x]`	

**original line in Fig. 2*

obs/100k	GTX5	TesC	GTX6	Titan	GTX7	HD6570	HD7970
	0	47	43	512	0	508	748

Figure 9: PTX compare-and-swap spin lock (**cas-sl**)

__threadfence still necessary even if L1 isn't used

from "GPU Concurrency: Weak behaviours" paper

what gets cached where?

CUDA compute capability	default caching policy	opt-in to L1 caching?
2.x	ca (**L1** & L2)	n/a
3.x	cg (L2 only)	no
3.5, 3.7	cg (L2 only)	yes
5.x	cg (L2 only)	yes

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
            then
7:              Jump to line 17
8:          end if
9:          for targetIdx = 0 to warpSize − 1 do
10:             for sourceIdx = 0 to warpSize − 1 do
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
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
```

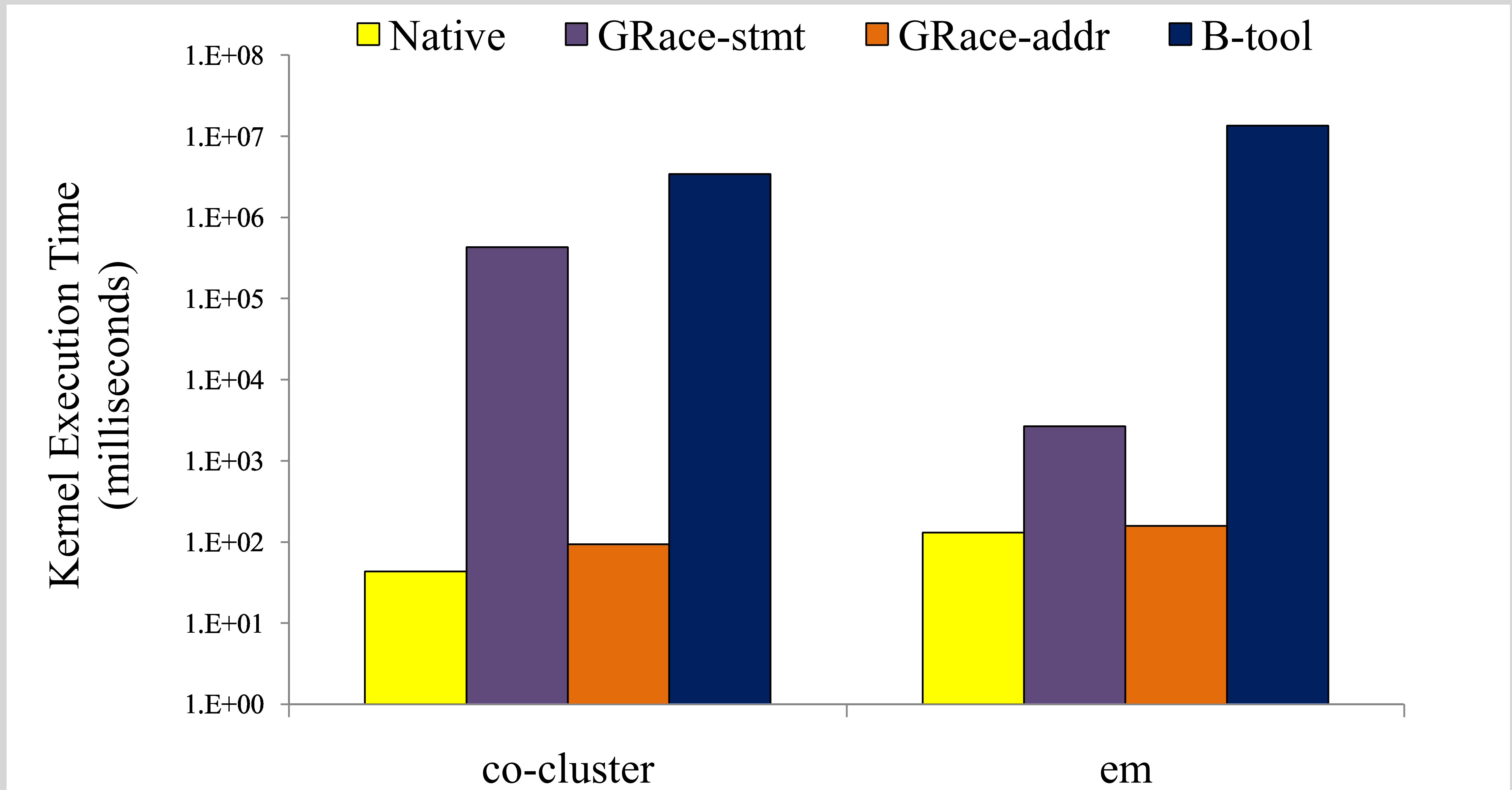

Figure 5. Runtime overhead of GRace. Note that the y-axis is on a logarithmic scale.

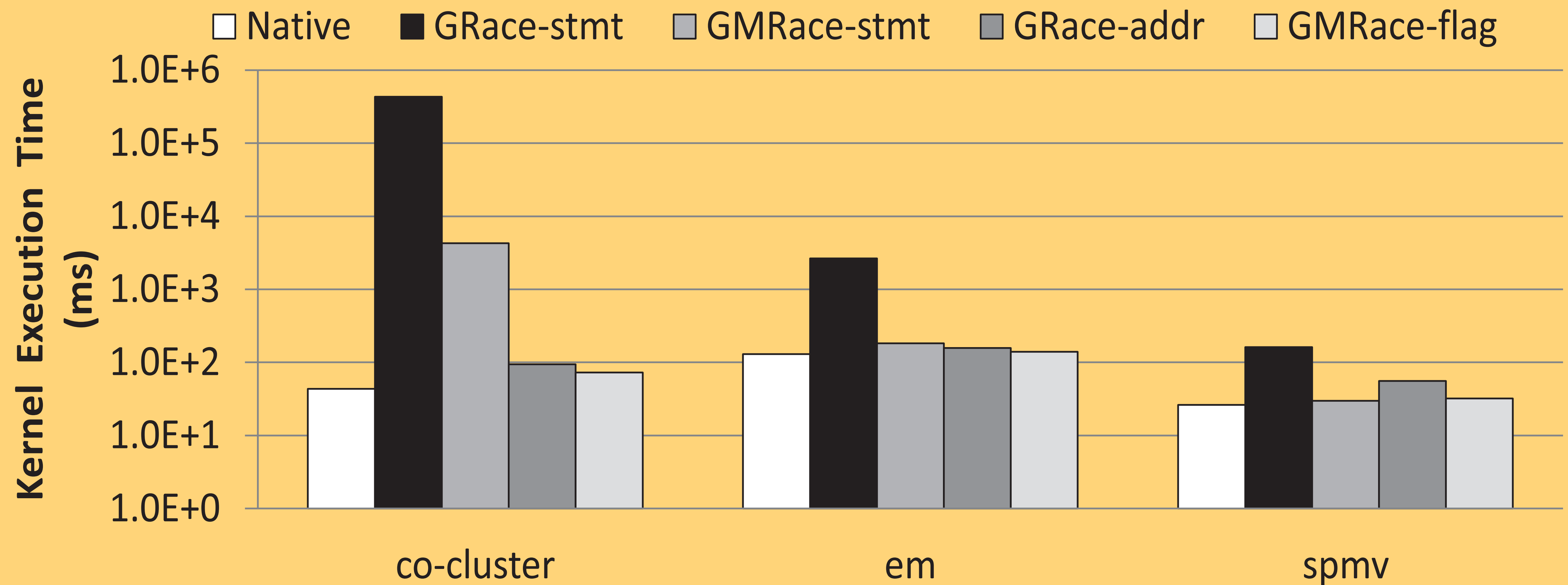

Fig. 5. Runtime overhead of different schemes of GMRace and GRace. Note that the y-axis is on a logarithmic scale.

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?

fixing races

- 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?