GPU Concurrency: Weak Behaviours and Programming Assumptions

Jade Alglave[1,2] Mark Batty[3] Alastair F. Donaldson[4] Ganesh Gopalakrishnan[5]
Jeroen Ketema[4] Daniel Poetzl[6] Tyler Sorensen[1,5] John Wickerson[4]

[1] University College London [2] Microsoft Research [3] University of Cambridge
[4] Imperial College London [5] University of Utah [6] University of Oxford

Jyh-Jing Hwang, Yiren(Max) Lu

03/02/2017

Penn

Outline

1. Introduction
2. Weak behaviors examples
3. Test methodology
4. Proposed memory model
5. Folklores

Introduction

- Current specifications of languages and hardware for GPU are inconclusive
- Programmers often rely on folklore assumptions

Contributions

- A framework for generating and running litmus tests to question memory consistency on GPU chips
- A set of heuristics for provoking weak behaviors
- An extensive empirical evaluations across seven GPUs
- Revealed ten correctness issues
- A formal model of Nvidia GPUs to build more reliable chips, compilers and applications

Penn

Weak Behaviors

- Architectures implement weak memory models where the hardware is allowed to re-order certain memory instructions
- Weak memory models can allow weak behaviors (executions that do not correspond to a specified interleaving)

Weak Behaviors

- Architectures implement weak memory models where the hardware is allowed to re-order certain memory instructions
- Weak memory models can allow weak behaviors (executions that do not correspond to an interleaving)

name	description	figures
coRR	coherence of read-read pairs	1, 4
mp	message passing (viz. handshake)	3, 5, 7, 9
lb	load buffering	8, 11
sb	store buffering	12

Table 3: Glossary of idioms

Weak Behaviors

coherence of read-read pairs (coRR)

init: `global x=0` **final:** `r1=1 ∧ r2=0` **threads:** intra-CTA

0.1	`st.cg [x],1`	
1.1		`ld.cg r1,[x]`
1.2		`ld.cg r2,[x]`

obs/100k	GTX5	TesC	GTX6	Titan	GTX7	HD6570	HD7970
	11642	8879	9599	9787	0	0	0

Figure 1: PTX test for coherent reads (**coRR**)

SC is not guaranteed!

Penn

Weak Behaviors

PTX coRR mixing cache operators (coRR-L2-L1)

load from l2 cache

load from l1 cache

init: `global x=0` **final:** `r1=1 ∧ r2=0` **threads:** intra-CTA

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

obs/100k	*fence*	GTX5	TesC	GTX6	Titan	GTX7
`no-op`		2556	2982	2	141	0
`membar.cta`		1934	2180	0	0	0
`membar.gl`		0	1496	0	0	0
`membar.sys`		0	1428	0	0	0

Figure 4: PTX **coRR** mixing cache operators (**coRR-L2-L1**)

Weak Behaviors

message passing (viz. handshake) (mp)

init: (global x=0, global y=0) **final:** r1=1 ∧ r2=0 (stale data) **threads:** inter-CTA

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

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**)

No matter how strong the fences are!

Penn

Weak Behaviors

message passing (viz. handshake) (mp)

init: (global x=0, global y=0) **final:** r1=1 ∧ r2=0 (stale data) **threads:** inter-CTA

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

Interleaving 1	Interleaving 2	Interleaving 3
a: x ← 1;	a: x ← 1;	a: x ← 1;
b: y ← 1;	c: r1 ← y;	c: r1 ← y;
c: r1 ← y;	b: y ← 1;	d: r2 ← x;
d: r2 ← x;	d: r2 ← x;	b: y ← 1;
Final: r1 = 1 ∧ r2 = 1	Final: r1 = 0 ∧ r2 = 1	Final: r1 = 0 ∧ r2 = 0
Interleaving 4	**Interleaving 5**	**Interleaving 6**
c: r1 ← y;	c: r1 ← y;	c: r1 ← y;
a: x ← 1;	a: x ← 1;	d: r2 ← x;
b: y ← 1;	d: r2 ← x;	a: x ← 1;
d: r2 ← x;	b: y ← 1;	b: y ← 1;
Final: r1 = 0 ∧ r2 = 1	Final: r1 = 0 ∧ r2 = 1	Final: r1 = 0 ∧ r2 = 0

Weak Behaviors

PTX mp with volatiles (mp-volatile)

Enforce sequential consistency? No! sorry PTX manual!

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

0.1	`st.volatile [x],1`	1.1	`ld.volatile r1,[y]`	
0.2	`st.volatile [y],1`	1.2	`ld.volatile r2,[x]`	

obs/100k	GTX5	TesC	GTX6	Titan	GTX7
	6301	4977	2753	2188	0

Figure 5: PTX **mp** with volatiles (**mp-volatile**)

Same block, but different warps!

Test Methodology

- Extended the `litmus` CPU testing tool of Alglave and Maranget to run GPU tests
- Given a GPU litmus test, generates an executable CUDA or OpenCL code for the test

Generate Test Code

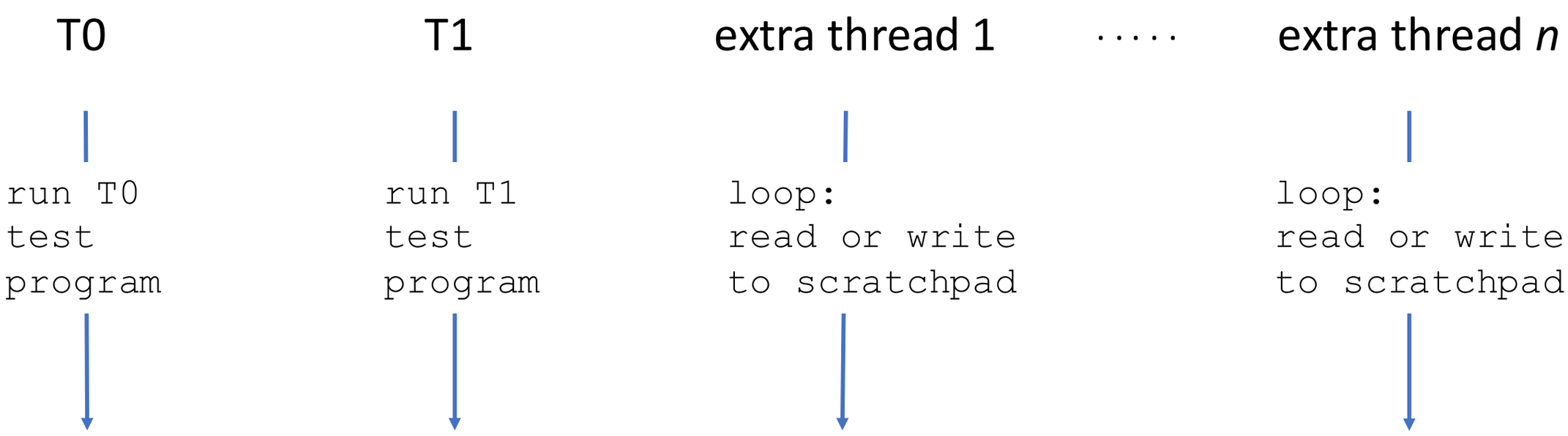
store buffering (sb)

```
1   GPU_PTX SB
2   {0:.reg .s32 r0;       0:.reg .s32 r2;
3    0:.reg .b64 r1 = x;   0:.reg .b64 r3 = y;
4    1:.reg .s32 r0;       1:.reg .s32 r2;
5    1:.reg .b64 r1 = y;   1:.reg .b64 r3 = x;}
6   T0                     | T1                     ;
7   mov.s32 r0,1           | mov.s32 r0,1           ;
8   st.cg.s32 [r1],r0      | st.cg.s32 [r1],r0      ;
9   ld.cg.s32 r2,[r3]      | ld.cg.s32 r2,[r3]      ;
10  ScopeTree(grid(cta(warp T0) (warp T1)))
11  x: shared, y: global
12  exists (0:r2=0 /\ 1:r2=0)
```

Figure 12: GPU PTX litmus test **sb**

Test Methodology: Memory Stress

Heuristics: Stressing caching protocols might trigger weak behaviors

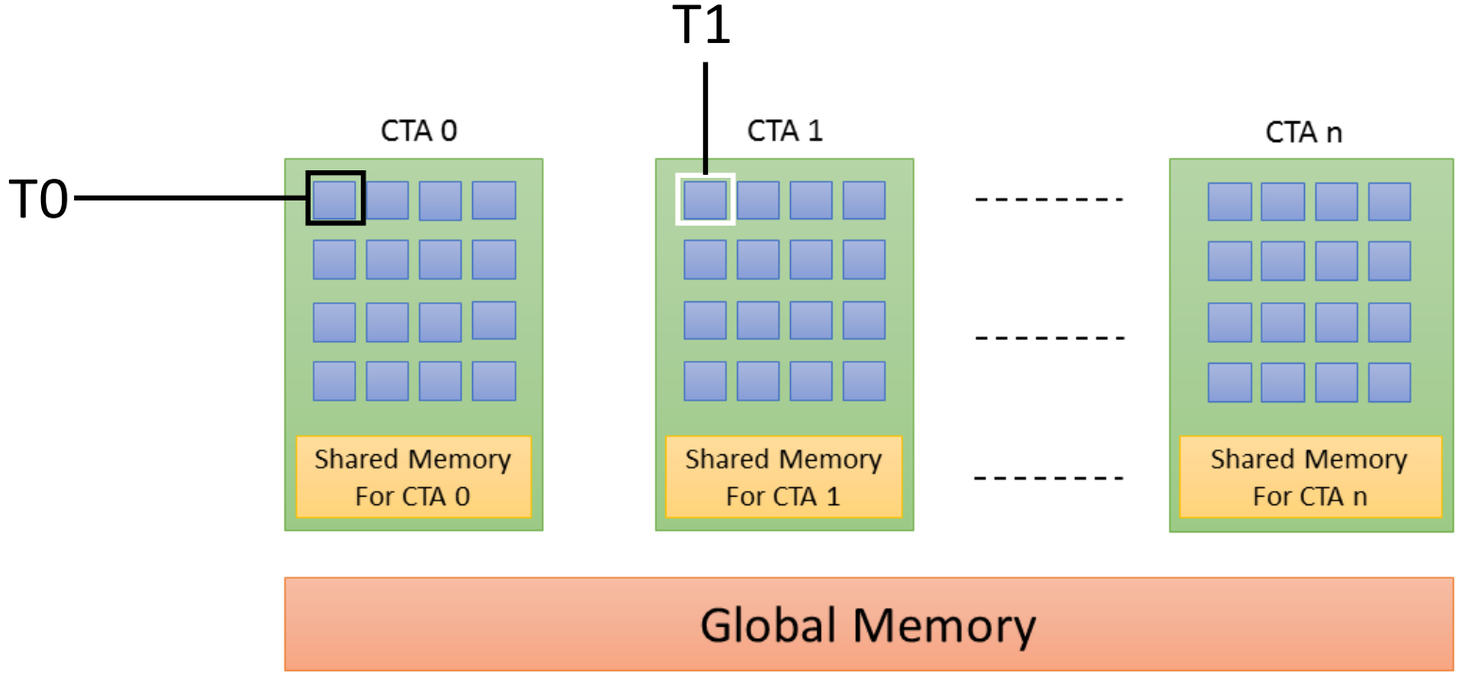


Test Methodology: General Bank Conflicts

Heuristics: GPUs access shared memory through banks, which can handle only one access at a time. Bank conflicts occur when multiple threads in a warp seek simultaneous access to locations in the same block. Hardware might handle accesses out of order to hide the latency.

Test Methodology: Thread Randomization

Heuristics: Varying the thread ids of testing threads and the number of threads per kernel might exercise different components and paths through the hardware and hence, increase the likelihood of weak behaviors.



Test Methodology: Thread Synchronization

Heuristics: Synchronizing testing threads immediately before running the test promotes interactions while values are actively moving through the memory system

Test Methodology

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	memory stress									●	●	●	●	●	●	●	●
	general bank conflicts					●	●	●	●					●	●	●	●
	thread synchronisation			●	●			●	●			●	●			●	●
	thread randomisation		●		●		●		●		●		●		●		●
Nvidia GTX Titan	**coRR** (intra-CTA)	0	0	0	0	0	1235	0	9774	161	118	847	362	632	3384	3993	**9985**
	lb (inter-CTA)	0	0	0	0	0	0	0	0	181	1067	1555	**2247**	4	37	83	486
	mp (inter-CTA)	0	0	0	0	0	621	0	2921	315	1128	2372	**4347**	7	94	442	2888
	sb (inter-CTA)	0	0	0	0	0	0	0	0	462	1403	3308	**6673**	3	50	88	749
AMD Radeon HD 7970	**coRR** (intra-CTA)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	lb (inter-CTA)	10959	8979	31895	29092	13510	12729	29779	26737	5094	9360	37624	**38664**	5321	10054	32796	34196
	mp (inter-CTA)	212	31	243	158	277	46	318	247	473	217	1289	563	611	339	**2542**	1628
	sb (inter-CTA)	0	0	0	0	**2**	0	**2**	0	0	0	0	0	0	0	0	0

Table 6: Observations out of 100k executions for combinations of incantations (all tests target global memory)

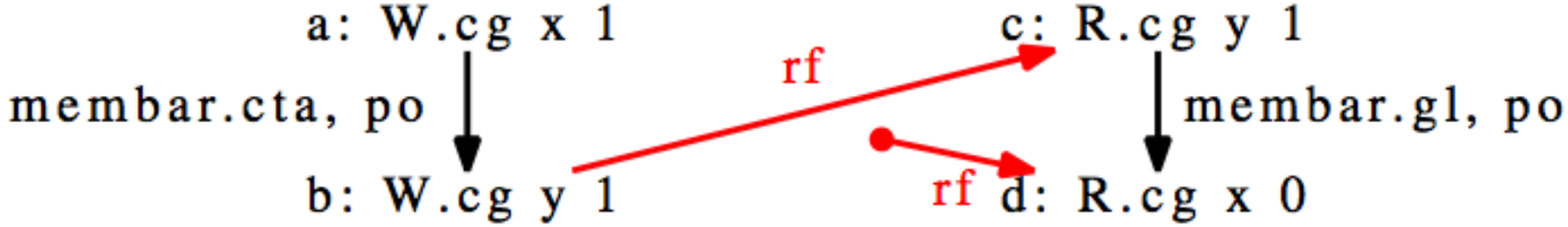
Test Methodology: Checking Optimization

SASS code may vary (extra lines, reordering) from the PTX code due to compile optimization. The authors developed a tool to ensure the consistency.

Proposed Model: Candidate Executions

init: $\begin{pmatrix}\texttt{global x=0}\\ \texttt{global y=0}\end{pmatrix}$ **final:** r0=1 ∧ r2=0 **threads:** intra-CTA

0.1	st.cg [x],1	1.1	ld.cg r0,[y]
0.2	membar.cta	1.2	membar.gl
0.3	st.cg [y],1	1.3	ld.cg r2,[x]



Memory events: write (W), read (R)

Scope relations: block (cta), grid (gl), system (sys)

Program order (po) > Dependency (dp): address (addr), data (data), control (ctrl)

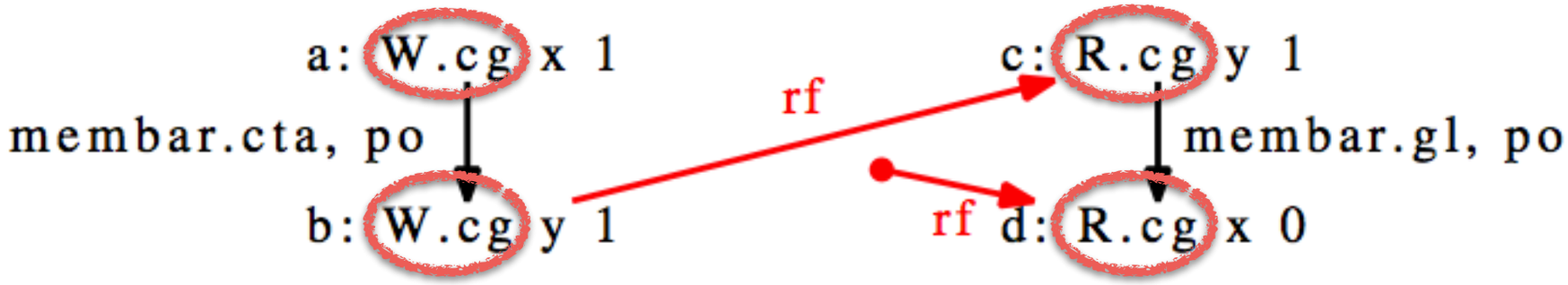
Communication relations [inter-threads]: read-from relation (rf)

Penn

Proposed Model: Candidate Executions

init: $\begin{pmatrix} \texttt{global x=0} \\ \texttt{global y=0} \end{pmatrix}$ **final:** $\texttt{r0=1} \wedge \texttt{r2=0}$ **threads:** intra-CTA

0.1	`st.cg [x],1`	1.1	`ld.cg r0,[y]`
0.2	`membar.cta`	1.2	`membar.gl`
0.3	`st.cg [y],1`	1.3	`ld.cg r2,[x]`

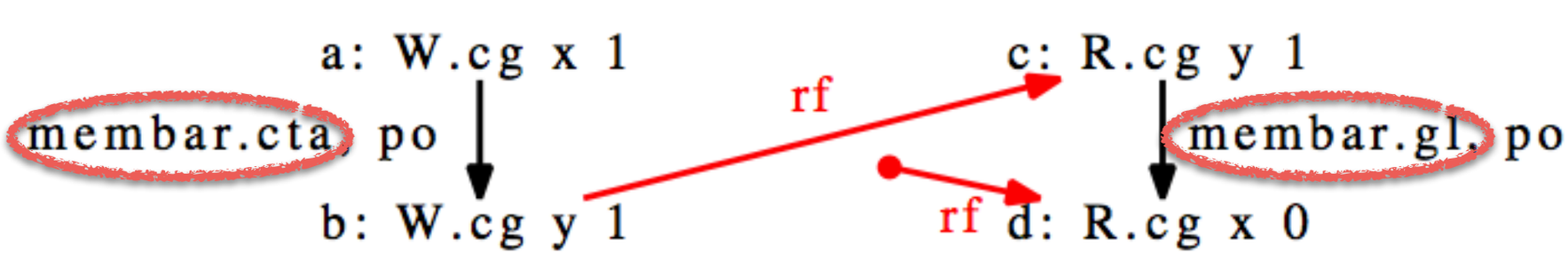


Memory events: write (W), read (R)
with Cache operator: L1 (.ca), L2 (.cg)

Proposed Model: Candidate Executions

init: $\begin{pmatrix}\texttt{global x=0}\\ \texttt{global y=0}\end{pmatrix}$ **final:** r0=1 ∧ r2=0 **threads:** intra-CTA

0.1	st.cg [x],1	1.1	ld.cg r0,[y]
0.2	membar.cta	1.2	membar.gl
0.3	st.cg [y],1	1.3	ld.cg r2,[x]

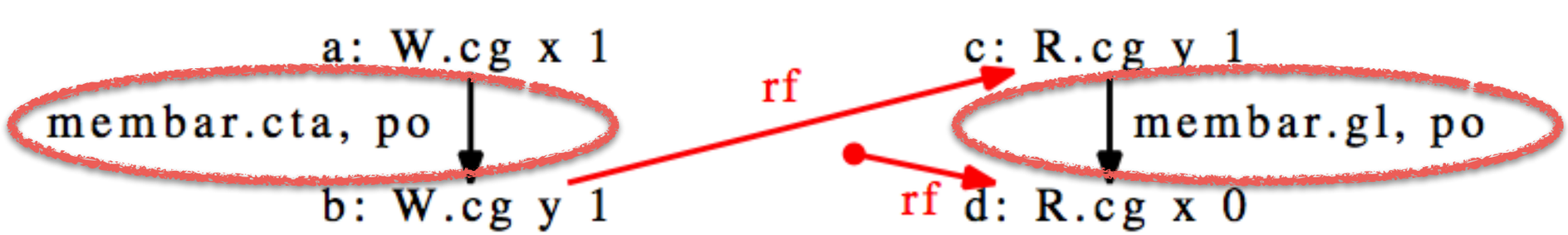


Fences with Scope relations: block (cta), grid (gl), system (sys)

Proposed Model: Candidate Executions

init: $\begin{pmatrix} \texttt{global x=0} \\ \texttt{global y=0} \end{pmatrix}$ **final:** $\texttt{r0=1} \wedge \texttt{r2=0}$ **threads:** intra-CTA

0.1	`st.cg [x],1`	1.1	`ld.cg r0,[y]`
0.2	`membar.cta`	1.2	`membar.gl`
0.3	`st.cg [y],1`	1.3	`ld.cg r2,[x]`



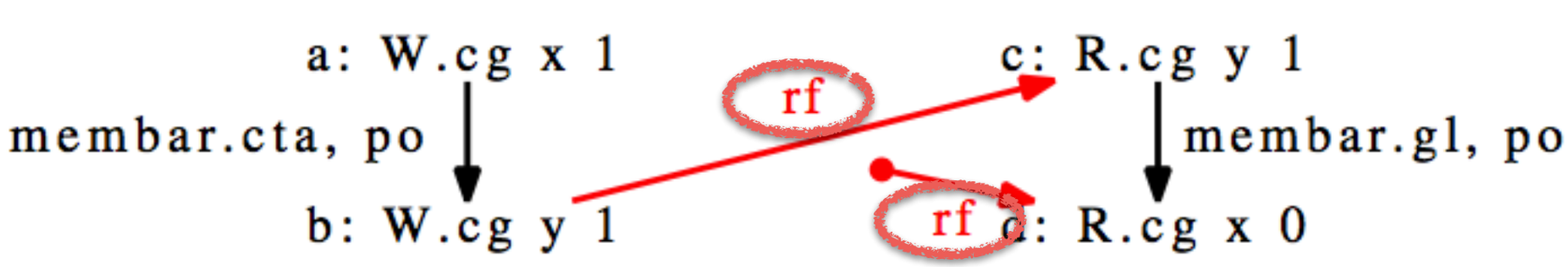
- Program order (**po**): total order within a thread
  - Dependency (**dp**): included in **po**, instructions separated by address (**addr**), data (**data**), or control (**ctrl**)
  - Fence relations: included in **po**, **member** w.r.t. a scope

Penn

Proposed Model: Candidate Executions

init: $\begin{pmatrix} \texttt{global x=0} \\ \texttt{global y=0} \end{pmatrix}$ **final:** $\texttt{r0=1} \wedge \texttt{r2=0}$ **threads:** intra-CTA

0.1	`st.cg [x],1`	1.1	`ld.cg r0,[y]`
0.2	`membar.cta`	1.2	`membar.gl`
0.3	`st.cg [y],1`	1.3	`ld.cg r2,[x]`



Communication relations [inter-threads]: read-from relation (**rf**)

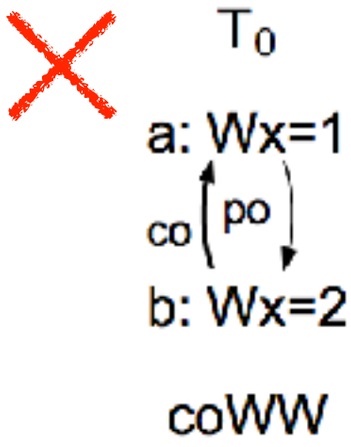
Proposed Model: Constraints

```
1   let com = rf | co | fr
2   let po-loc-llh =
3     WW(po-loc) | WR(po-loc) | RW(po-loc)
4   acyclic (po-loc-llh | com) as sc-per-loc-llh
5   let dp = addr | data | ctrl
6   acyclic (dp | rf) as no-thin-air
7   let rmo(fence) = dp | fence | rfe | co | fr
```

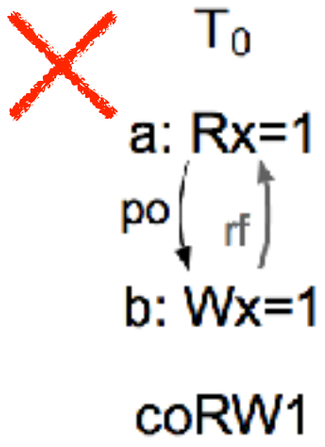
- **Sparc's Relaxed Memory Order (RMO)**: allows any pair of memory accesses to be **reordered**, unless a dependency or a fence.
- Three Principles of RMO:
  1. **SC per Location with Load-Load Hazard**

Proposed Model: Constraints

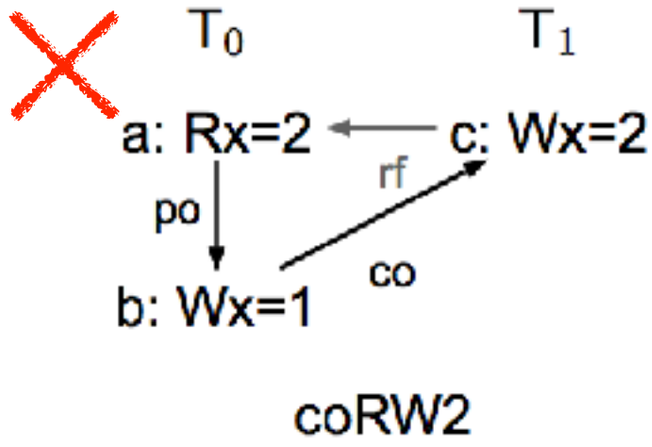
```
1   let com = rf | co | fr
2   let po-loc-llh =
3     WW(po-loc) | WR(po-loc) | RW(po-loc)
4   acyclic (po-loc-llh | com) as sc-per-loc-llh
```



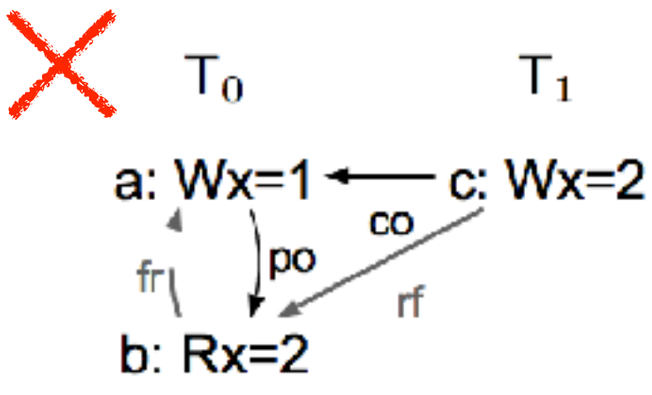
coWW



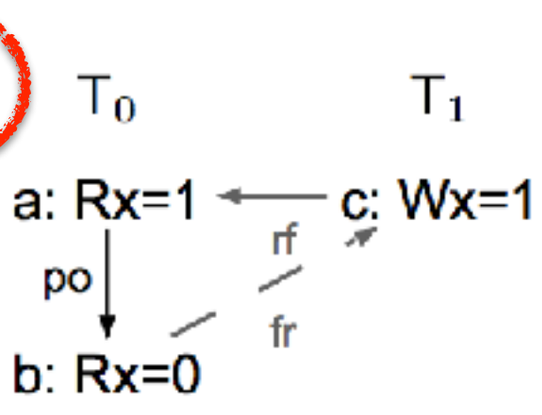
coRW1



coRW2



coWR



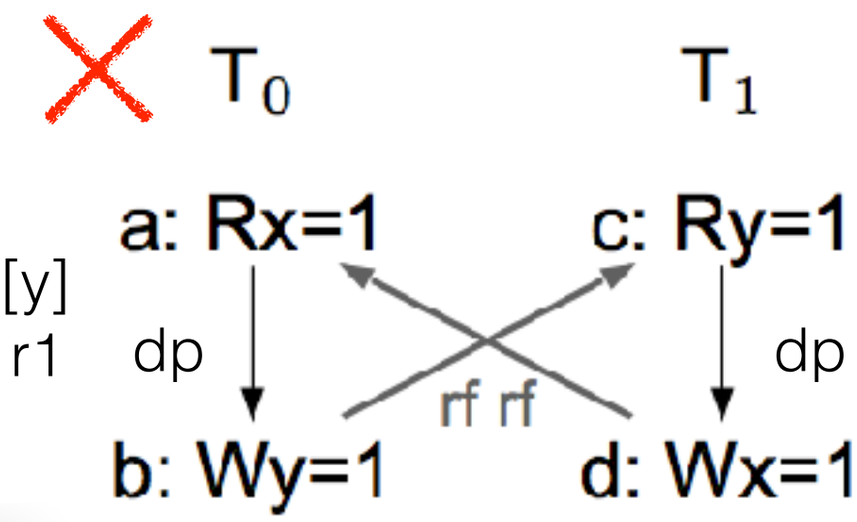
coRR

Proposed Model: Constraints

```
1   let com = rf | co | fr
2   let po-loc-llh =
3     WW(po-loc) | WR(po-loc) | RW(po-loc)
4   acyclic (po-loc-llh | com) as sc-per-loc-llh
5   let dp = addr | data | ctrl
6   acyclic (dp | rf) as no-thin-air
```

- **Sparc's Relaxed Memory Order (RMO)**: allows any pair of memory accesses to be reordered, unless a dependency or a fence.
- Three Principles of RMO:
  2. **No Thin Air**

T0 | T1
ld.cg r0 [x] | ld.cg r1 [y]
st.cg [y] r0 | st.cg [x] r1

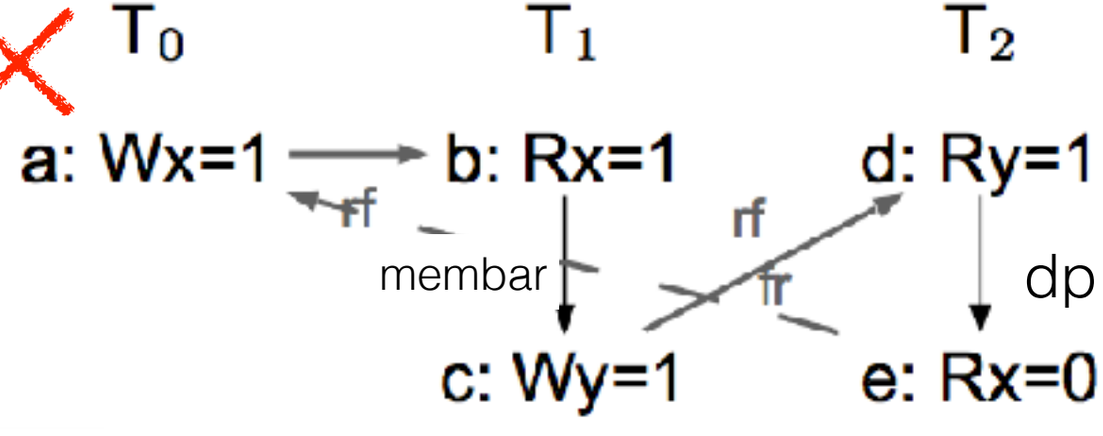


lb+ppos

Proposed Model: Constraints

```
7   let rmo(fence) = dp | fence | rfe | co | fr
8   let sys-fence = membar.sys
9   let gl-fence = membar.gl | sys-fence
10  let cta-fence = membar.cta | gl-fence
11  let rmo-cta = rmo(cta-fence) & cta
12  let rmo-gl = rmo(gl-fence) & gl
13  let rmo-sys = rmo(sys-fence) & sys
14  acyclic rmo-cta as cta-constraint
15  acyclic rmo-gl as gl-constraint
16  acyclic rmo-sys as sys-constraint
```

- **Sparc's Relaxed Memory Order (RMO)**: allows any pair of memory accesses to be reordered, unless a dependency or a fence.
- Three Principles of RMO:
  3. The **rmo** Relation



Folklore 1: "GPUs exhibit no weak memory behaviours"

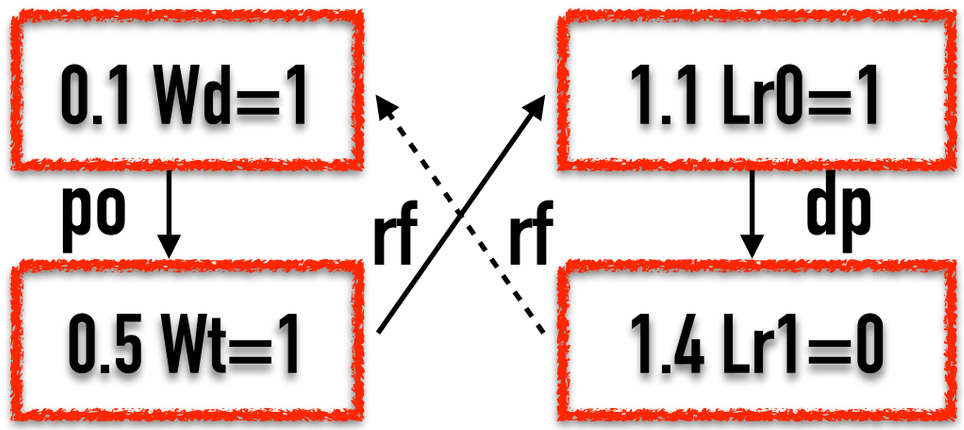
```
1    volatile int head, tail;
2    void push(task){
3      tasks[tail] = task;
4(+)   __threadfence();
5      tail++; }
6    Task steal(){
7      int oldHead = head;
8      if (tail <= oldHead.index) return EMPTY;
9(+)   __threadfence();
10     task = tasks[oldHead.index];
11(+)  __threadfence();
12     newHead = oldHead; newHead.index++;
13     if (CAS(&head,oldHead,newHead)) return task;
14     return FAILED; }
15   Task pop(){
16     ...
17     tail--;
18     ...
19     if( oldTail == oldHead.index )
20       if( CAS(&head, oldHead, newHead) ) {
21(+)      __threadfence();
22        return task; }
23(+)  atomicExch(head, newHead);
24(-)  head = newHead;
25     return FAILED; }
```

init: (global t=0, global d=0) **final:** r0=1 ∧ r1=0 **threads:** inter-CTA

		*			*
0.1	st.cg [d],1	3	1.1	ld.volatile r0,[t]	8
0.2(+)	membar.gl	4	1.2	setp.eq p4,r0,0	8
0.3	ld.volatile r2,[t]	5	1.3(+)	@!p4 membar.gl	9
0.4	add r2,r2,1	5	1.4	@!p4 ld.cg r1,[d]	10
0.5	st.volatile [t],r2	5			

**original line in Fig. 6*

obs/100k	GTX5	TesC	GTX6	Titan	GTX7	HD6570	HD7970
	0	4	36	65	0	0	0



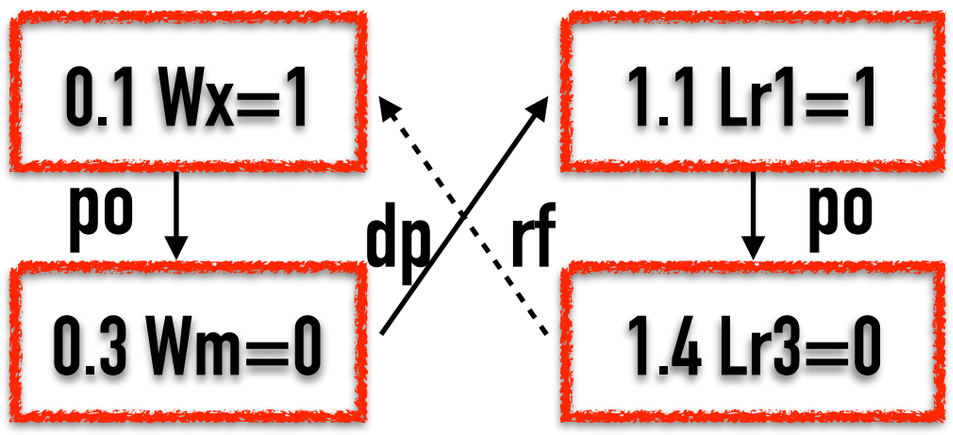
Folklore 2: "Atomic operations provide synchronization"

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*

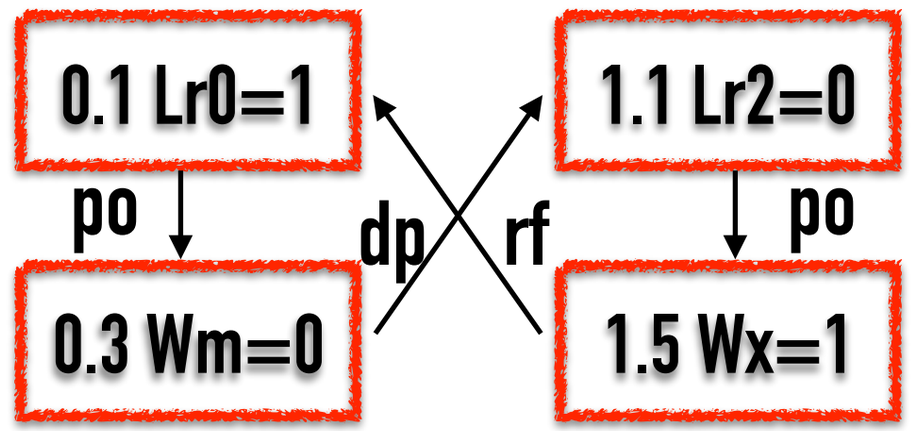
obs/100k	GTX5	TesC	GTX6	Titan	GTX7	HD6570	HD7970
	0	47	43	512	0	508	748



Penn

Folklore 3: "Only unlocks need fences"

```
1     bool leaveLoop = false;
2     while(!leaveLoop) {
3       int lockValue = atomicCAS(lockAddr,0,1);
4       if(lockValue == 0) {
5         leaveLoop = true;
6(+)      __threadfence();
7         // critical section
8(+)      __threadfence();
9(+)      atomicExch(lockAddr, 0);
10(-)     *lockAddr = 0;}
11(-)   __threadfence();}
```



init: (global x=0, global m=1) **final:** r0=1 ∧ r2=0 **threads:** inter-CTA

		*			*
0.1	ld.cg r0,[x]	7	1.1	atom.cas r2,[m],0,1	3
0.2(+)	membar.gl	8	1.2	setp.eq p,r2,0	4
0.3(+)	atom.exch r1,[m],0	9	1.3	@p mov r3,1	5
0.4(-)	st.cg [m],0	10	1.4(+)	@p membar.gl	6
0.5(-)	membar.gl	11	1.5	@p st.cg [x],1	7

**original line in Fig. 10*

obs/100k	GTX5	TesC	GTX6	Titan	GTX7	HD6570	HD7970
	0	99	41	58	0	n/a	n/a

Thank you!

Any Questions?
(We actually have got A LOT!)

Example Questions

- Q: Page 2, Weak Behaviours: Thread0 stores 1 to address x. Thread1 performs 2 loads from address x to r1 and r2. Does the Read-read coherence violation occur because thread1 happens before thread0 and the loads in Thread1 get reordered ? That is, first ld r2, [x] happens, which loads 0 into r2. Then thread0 happens which stores 1 at address x. Then thread1 loads 1 into r1 ?

- A: We've talked about this.

Example Questions

- Q: Page 2, Programming Assumptions: Could you explain how the absence of threadfence() function makes both stale and future values available to the critical section?

- A: We've talked about this.

Example Questions

- Q: "Fig 4 shows that on the Tesla C2075, no fence guarantees that updated values can be read reliably from the L1 cache even when first reading an updated value from the L2 cache. This issue does not apply to AMD chips". Could this be possible because the 2 loads are not re-orderer in AMD, or could there be any other possibility?

- A: Yes, or a read from L2 cache will also update L1 cache.

Example Questions

- Q: Is it possible to provide an example to explain the "No Thin Air" principle in section 5.2.2?

- A: Yes, we've provided one in our presentation.

Testing Methodology Questions

- Q: Given some tests have very small obs/100k values (e.g. 3 in Fig 3, and 2 in Fig 4), how/why did the authors decide 100k runs was "good enough"?

- A: It's the "possibility" that matters, not the "probability".

Testing Methodology Questions

- Q: Please give a high level explanation(maybe in context of CPU) of the different litmus tests that the authors aim to test (coRR, mp, lb, sb). Are there other litmus tests the authors could have tested? Why did they choose these tests to expose inconsistencies?

- A: We believe they've tested a lot more experiments but only showed those "surprising" weak behaviors. Also, we believe those 4 kinds of operations are commonly used in practice.

Testing Methodology Questions

- Q: In the testing methodology mentioned in Section 4, could you please elaborate on the function of the "scope tree”?
- A

Execution hierarchy A test specifies the location of its threads in the concurrency hierarchy (see Sec. 2.1) through a *scope tree* (borrowing the term *scope* from [24, 25]). In Fig. 12, we declare the scope tree on line 10: T_0 and T_1 are in the same CTA but different warps.

Testing Methodology Questions

- Q: If I understand correctly, the numerous "bugs" they found were caused by either the compiler reordering instructions or the hardware not doing what the documentation claimed it would for all possible executions. My question is: what methods exists to verify this type of correctness for hardware?

- A: We have no idea of methodology beyond this paper, so let's ask Joe! :)

Cache Questions

- Q: I'm a bit confused about what the caches in CUDA do exactly. Caches are typically transparent. The paper, however, mentions loads and stores that target particular caches. Can you explain this?

- A: Normally we write programs and the rest (like cache accesses) will be handled by compilers and hardwares. But here, the authors find a way to look at assembly directly, so they can target at a specific cache level.
  BTW the authors don't know what caches do exactly either. This is the reason why this paper ever exists. ;)

Penn

Cache Questions

- Q: Why did it seem like adding fences was the solution to all of the consistency problems that were revealed from the litmus tests? Was it because fences had direct access to the caches and it gave the programmer control over when they could flush the cache (synchronize)? Was this ability not possible through some of the other semantics (atomics, CAS, volatile)?

- A: It's because this paper focuses on memory access operations.

etc. Questions

- Q: What is the purpose of the volatile keyword if it does not ensure sequential consistency?

- A: We don't know. Let's ask Joe! :)

etc. Questions

- Q: Do compiler optimizations affect the incidence of weak behaviors?
- A: That is possible. But in their experiments, compilers did not help mitigate the weak behaviors.

etc. Questions

- Q: Why does the GTX 280 not exhibit any weak behaviors, while the others do? Is that because it had no store buffers? Does it even have fences? If not, how was it even tested because the litmus tests need fences as far as I can see?

- A: We don't know. Leet's ask Joe! :)

etc. Questions

- Q: In section 4.3.2 how do bank conflicts reduce the number of inter-CTA weak behaviors?

- A: The paper does not explain it. Let's ask Joe or Nvidia! :)

etc. Questions

- Q: I'm pretty confused about the outcome of section 5… have the authors just created a model one can use to understand possible reorderings? Could a set of candidate executions and the author's model be extended to ensure a given PTX file doesn't violate the model?

- A: The model is proposed to predict the memory behaviors and it succeeded to do so in many experiments.

etc. Questions

- Q: In the section about "Checking for Optimizations" I did not understand how their method worked specifically the part about: "....we first add instructions to the PTX code of a litmus test that specify certain properties of the test, such as the order of instructions within a thread. The compiled code thus contains both the litmus test code and the specification... A specification (in PTX) consists of a sequence of xor instructions, placed at the end of each thread..."

- A: We will cover it in our presentation