GPU Concurrency:
Weak Behaviours and Programming Assumptions

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

\textsuperscript{1} University College London \quad \textsuperscript{2} Microsoft Research \quad \textsuperscript{3} University of Cambridge
\textsuperscript{4} Imperial College London \quad \textsuperscript{5} University of Utah \quad \textsuperscript{6} University of Oxford

Jyh–Jing Hwang, Yiren(Max) Lu
03/02/2017
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
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).

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
<th>figures</th>
</tr>
</thead>
<tbody>
<tr>
<td>coRR</td>
<td>coherence of read-read pairs</td>
<td>1, 4</td>
</tr>
<tr>
<td>mp</td>
<td>message passing (viz. handshake)</td>
<td>3, 5, 7, 9</td>
</tr>
<tr>
<td>lb</td>
<td>load buffering</td>
<td>8, 11</td>
</tr>
<tr>
<td>sb</td>
<td>store buffering</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 3: Glossary of idioms
Weak Behaviors
coherence of read-read pairs (coRR)

<table>
<thead>
<tr>
<th>init: global x=0</th>
<th>final: r1=1 ∧ r2=0</th>
<th>threads: intra-CTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1 st.cg [x],1</td>
<td>1.1 ld.cg r1,[x]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.2 ld.cg r2,[x]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>obs/100k</th>
<th>GTX5</th>
<th>TesC</th>
<th>GTX6</th>
<th>Titan</th>
<th>GTX7</th>
<th>HD6570</th>
<th>HD7970</th>
</tr>
</thead>
<tbody>
<tr>
<td>11642</td>
<td>8879</td>
<td>9599</td>
<td>9787</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 1: PTX test for coherent reads (coRR)

SC is not guaranteed!
Weak Behaviors

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

<table>
<thead>
<tr>
<th>init: global x=0</th>
<th>final: r1=1 ∧ r2=0</th>
<th>threads: intra-CTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1 st.cg [x], 1</td>
<td>1.1 ld.cg r1, [x]</td>
<td></td>
</tr>
<tr>
<td>1.2 fence</td>
<td>1.3 ld.ca r2, [x]</td>
<td></td>
</tr>
</tbody>
</table>

<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>2556</td>
<td>2982</td>
<td></td>
<td>2</td>
<td>141</td>
<td>0</td>
</tr>
<tr>
<td>membar.cta</td>
<td>1934</td>
<td>2180</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>membar.gl</td>
<td>0</td>
<td>1496</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>membar.sys</td>
<td>0</td>
<td>1428</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

threads: inter-CTA

<table>
<thead>
<tr>
<th>data</th>
<th>flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>st.cg [x], 1</td>
</tr>
<tr>
<td>0.2</td>
<td>fence</td>
</tr>
<tr>
<td>0.3</td>
<td>st.cg [y], 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>flag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ld.ca r1, [y]</td>
</tr>
<tr>
<td></td>
<td>fence</td>
</tr>
<tr>
<td></td>
<td>ld.ca r2, [x]</td>
</tr>
</tbody>
</table>

<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></td>
<td>4979</td>
<td>10581</td>
<td>3635</td>
<td>6011</td>
<td>3</td>
</tr>
<tr>
<td>membar.cta</td>
<td></td>
<td>0</td>
<td>308</td>
<td>14</td>
<td>1696</td>
<td>0</td>
</tr>
<tr>
<td>membar.gl</td>
<td></td>
<td>0</td>
<td>187</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>membar.sys</td>
<td></td>
<td>0</td>
<td>162</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

No matter how strong the fences are!
Weak Behaviors

message passing (viz. handshake) (mp)

\[
\text{init: } \begin{cases} \text{global } x=0 \\ \text{global } y=0 \end{cases} \quad \text{final: } r1=1 \land r2=0 \quad \text{threads: inter-CTA}
\]

<table>
<thead>
<tr>
<th>data</th>
<th>0.1</th>
<th>\text{st.cg } [x], 1</th>
<th>1.1</th>
<th>\text{ld.ca } r1, [y]</th>
<th>flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>fence</td>
<td></td>
<td>1.2</td>
<td>fence</td>
<td></td>
</tr>
<tr>
<td>flag</td>
<td>0.3</td>
<td>\text{st.cg } [y], 1</td>
<td>1.3</td>
<td>\text{ld.ca } r2, [x]</td>
<td>data</td>
</tr>
</tbody>
</table>

\begin{align*}
\text{Interleaving 1} & : & \text{Interleaving 2} & : & \text{Interleaving 3} \\
\text{a: } x & \leftarrow 1; & \text{a: } x & \leftarrow 1; & \text{a: } x & \leftarrow 1; \\
\text{b: } y & \leftarrow 1; & \text{c: } r1 & \leftarrow y; & \text{c: } r1 & \leftarrow y; \\
\text{c: } r1 & \leftarrow y; & \text{b: } y & \leftarrow 1; & \text{d: } r2 & \leftarrow x; \\
\text{d: } r2 & \leftarrow x; & \text{d: } r2 & \leftarrow x; & \text{b: } y & \leftarrow 1; \\
\text{Final: } r1 = 1 & \land r2 = 1 & \text{Final: } r1 = 0 & \land r2 = 1 & \text{Final: } r1 = 0 & \land r2 = 0 \\
\text{Interleaving 4} & : & \text{Interleaving 5} & : & \text{Interleaving 6} \\
\text{c: } r1 & \leftarrow y; & \text{c: } r1 & \leftarrow y; & \text{c: } r1 & \leftarrow y; \\
\text{a: } x & \leftarrow 1; & \text{a: } x & \leftarrow 1; & \text{d: } r2 & \leftarrow x; \\
\text{b: } y & \leftarrow 1; & \text{d: } r2 & \leftarrow x; & \text{a: } x & \leftarrow 1; \\
\text{d: } r2 & \leftarrow x; & \text{b: } y & \leftarrow 1; & \text{b: } y & \leftarrow 1; \\
\text{Final: } r1 = 0 & \land r2 = 1 & \text{Final: } r1 = 0 & \land r2 = 1 & \text{Final: } r1 = 0 & \land r2 = 0
\end{align*}
Weak Behaviors

PTX mp with volatiles (mp-volatile)

Enforce sequential consistency? **No! sorry PTX manual!**

<table>
<thead>
<tr>
<th>init:</th>
<th>final:</th>
<th>threads:</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared x=0 [\text{shared y=0}]</td>
<td>r1=1 &amp; r2=0</td>
<td>intra-CTA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>obs/100k</th>
<th>GTX5</th>
<th>TesC</th>
<th>GTX6</th>
<th>Titan</th>
<th>GTX7</th>
</tr>
</thead>
<tbody>
<tr>
<td>6301</td>
<td>4977</td>
<td>2753</td>
<td>2188</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

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

Same block, but **different warps!**
• 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)

```ptx
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

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>extra thread 1</th>
<th>····</th>
<th>extra thread n</th>
</tr>
</thead>
<tbody>
<tr>
<td>run T0 test program</td>
<td>run T1 test program</td>
<td>loop: read or write to scratchpad</td>
<td></td>
<td>loop: read or write to scratchpad</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th></th>
<th>Nvidia GTX Titan</th>
<th>AMD Radeon HD 7970</th>
</tr>
</thead>
<tbody>
<tr>
<td>coRR (intra-CTA)</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>lb (inter-CTA)</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td>10959 8979 31895 29092 13510 12729 29779 26737 5094 9360 37624 38664 5321 10054 32796 34196</td>
</tr>
<tr>
<td>mp (inter-CTA)</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td>212 31 243 158 277 46 318 247 473 217 1289 563 611 339 2542 1628</td>
</tr>
<tr>
<td>sb (inter-CTA)</td>
<td>0 0 0 0 0 2 0 0 2 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

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

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)
Proposed Model: Candidate Executions

init: \( \left( \begin{array}{l} \text{global } x=0 \\ \text{global } y=0 \end{array} \right) \)

final: \( r0=1 \land r2=0 \)

threads: intra-CTA

<table>
<thead>
<tr>
<th></th>
<th>Memory events</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>st.cg ([x],1)</td>
<td>1.1</td>
</tr>
<tr>
<td>0.2</td>
<td>membar.cta</td>
<td>1.2</td>
</tr>
<tr>
<td>0.3</td>
<td>st.cg ([y],1)</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Memory events: write (W), read (R)  
with Cache operator: L1 (.ca), L2 (.cg)
**Proposed Model: Candidate Executions**

<table>
<thead>
<tr>
<th>init:</th>
<th>final:</th>
<th>threads:</th>
</tr>
</thead>
<tbody>
<tr>
<td>((global \ x=0))</td>
<td>(r0=1 \land r2=0)</td>
<td>intra-CTA</td>
</tr>
<tr>
<td>((global \ y=0))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 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 (\texttt{cta}), grid (\texttt{gl}), system (\texttt{sys})
Proposed Model: Candidate Executions

- 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
Proposed Model: Candidate Executions

init: \[(\text{global } x=0) \quad \text{(global } y=0)\]

final: \(r0=1 \land r2=0\)

threads: intra-CTA

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>st.cg [x],1</td>
<td>1.1</td>
</tr>
<tr>
<td>0.2</td>
<td>membar.cta</td>
<td>1.2</td>
</tr>
<tr>
<td>0.3</td>
<td>st.cg [y],1</td>
<td>1.3</td>
</tr>
</tbody>
</table>

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

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

T₀
a: Wx=1
b: Wx=2

cWW

T₀
a: Rx=1
b: Wx=1

cRW1

T₀
a: Rx=2
b: Wx=1
c: Wx=2

cRW2

T₀
a: Rx=1
b: Rx=2
c: Wx=2

cWR

T₀
a: Rx=1
b: Rx=0
c: Wx=1

cRR
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**

\[
\begin{align*}
T_0 & \quad T_1 \\
ld. cg r0 [x] & \quad ld. cg r1 [y] \\
st. cg [y] r0 & \quad st. cg [x] r1
\end{align*}
\]

\[
\begin{align*}
a: R_x=1 & \quad c: R_y=1 \\
b: W_y=1 & \quad d: W_x=1 \\
\text{dp} & \quad \text{rf rf} \quad \text{dp} \\
\text{lb+ppos} & \quad \text{X}
\end{align*}
\]
Proposed Model: Constraints

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

```plaintext
let rmo(fence) = dp | fence | rfe | co | fr
let sys-fence = membar.sys
let gl-fence = membar.gl | sys-fence
let cta-fence = membar.cta | gl-fence
let rmo-cta = rmo(cta-fence) & cta
let rmo-gl = rmo(gl-fence) & gl
let rmo-sys = rmo(sys-fence) & sys

acyclic rmo-cta as cta-constraint
acyclic rmo-gl as gl-constraint
acyclic rmo-sys as sys-constraint
```
Folklore 1: “GPUs exhibit no weak memory behaviours”

```c
volatile int head, tail;
void push(task){
    tasks[tail] = task;
    _threadfence();
tail++;}
Task steal(){
    int oldHead = head;
    if (tail <= oldHead.index) return EMPTY;
    _threadfence();
task = tasks[oldHead.index];
    _threadfence();
    newHead = oldHead; newHead.index++;
    if (CAS(&head, oldHead, newHead)) return task;
    return FAILED; }
Task pop(){

    tail--;
    ... 
    if( oldTail == oldHead.index )
        if( CAS(&head, oldHead, newHead) ) {
            _threadfence();
            return task; }
        atomicExch(head, newHead);
    head = newHead;
    return FAILED; }
```

<table>
<thead>
<tr>
<th>init:</th>
<th>final:</th>
<th>threads:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(global t=0)</td>
<td>r0=1 ∧ r1=0</td>
<td>inter-CTA</td>
</tr>
<tr>
<td>(global d=0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0.1 st.cg [d],1 3 1.1 ld.volatile r0,[t] *
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. 5

<table>
<thead>
<tr>
<th>obs/100k</th>
<th>GTX5</th>
<th>TesC</th>
<th>GTX6</th>
<th>Titan</th>
<th>GTX7</th>
<th>HD6570</th>
<th>HD7970</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>36</td>
<td>65</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

0.1 Wd=1 po rf rf 1.1 Lr0=1 dp
0.5 Wt=1
1.4 Lr1=0
Folklore 2: “Atomic operations provide synchronization”

init: \((\text{global } x=0)\) \(\text{global } m=1\)  
final: \(r_1=0 \land r_3=0\) 
 threads: inter-CTA

<table>
<thead>
<tr>
<th>obs/100k</th>
<th>GTX5</th>
<th>TesC</th>
<th>GTX6</th>
<th>Titan</th>
<th>GTX7</th>
<th>HD6570</th>
<th>HD7970</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>47</td>
<td>43</td>
<td>512</td>
<td>0</td>
<td>0</td>
<td>508</td>
<td>748</td>
</tr>
</tbody>
</table>

*original line in Fig. 2

0.1 \(\text{st.cg } [x], 1\)  
0.2(+) \(\text{membar.gl}\)  
0.3 \(\text{atom.exch } r_0, [m], 0\)  
1.1 \(\text{atom.cas } r_1, [m], 0, 1\)  
1.2 \(\text{setp.eq } r_2, r_1, 0\)  
1.3(+) \(\text{@r1 membar.gl}\)  
1.4 \(\text{@r1 ld.cg } r_3, [x]\)
Folklore 3: “Only unlocks need fences”

```c
bool leaveLoop = false;
while(!leaveLoop) {
    int lockValue = atomicCAS(lockAddr, 0, 1);
    if(lockValue == 0) {
        leaveLoop = true;
        __threadfence();
    }
    // critical section
    __threadfence();
    atomicExch(lockAddr, 0);
    *lockAddr = 0;
    __threadfence();
}
```

**init:** (global \(x=0\))

**final:** \(r0=1 \land r2=0\)

**threads:** inter-CTA

<table>
<thead>
<tr>
<th>0.1</th>
<th>ld.cg r0,[x]</th>
<th>7</th>
<th>1.1</th>
<th>atom.cas r2,[m],0,1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2(+): membar.gl</td>
<td>8</td>
<td>1.2</td>
<td>setp.eq p,r2,0</td>
<td></td>
</tr>
<tr>
<td>0.3(+): atom.exch r1,[m],0</td>
<td>9</td>
<td>1.3</td>
<td>@p mov r3,1</td>
<td></td>
</tr>
<tr>
<td>0.4(-): st.cg [m],0</td>
<td>10</td>
<td>1.4(+): @p membar.gl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.5(-): membar.gl</td>
<td>11</td>
<td>1.5</td>
<td>@p st.cg [x],1</td>
<td></td>
</tr>
</tbody>
</table>

**obs/100k**

<table>
<thead>
<tr>
<th>GPU</th>
<th>GTX5</th>
<th>TesC</th>
<th>GTX6</th>
<th>Titan</th>
<th>GTX7</th>
<th>HD6570</th>
<th>HD7970</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>99</td>
<td>41</td>
<td>58</td>
<td>0</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

*original line in Fig. 10*
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.
• 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.
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! :)

Penn
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. ;)

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! :)

Penn
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! :)
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