LDetector: A low overhead data race detector for GPU programs

PENGCHENG LI
CHEN DING
XIAOYU HU
TOLGA SOYATA

UNIVERSITY OF ROCHESTER
Introduction & Contribution

- **Data races in GPU**
  - Impact correctness as bugs
  - Impact determinism
  - **Inter-warp data races** and intra-warp data races

- **Contribution**
  - A two-pass approach to detect data races in GPU
    - As a debugging tool
    - Simple, but effective and low overhead
    - Zero per memory access monitoring
  - Seek other optimizations to reduce two-pass cost
A data race example

```c
__global__ void Jacobi(int * data)
{
    extern __shared__ int a[]; // size 64
    int tid = threadIdx.x;
    ...
    if (tid < BLOCK_SIZE-1 && tid >0 )
                  A[tid+1]) / 3;
    __syncthreads();
}
```

Array size: 64
BLOCK_SIZE: 64
WARP_SIZE: 32
# of Warps: 2
# of threads: 64

read set

write set

value flow
A data race example

```c
__global__ void Jacobi(int * data) {
    extern __shared__ int a[]; // size 64
    int tid = threadIdx.x;
    ...
    if (tid < BLOCK_SIZE-1 && tid >0 )
          A[tid+1]) / 3;
    __syncthreads();
}
```

Array size: 64
BLOCK_SIZE: 64
WARP_SIZE: 32
# of Warps: 2
# of threads: 64
```c
__global__ void Jacobi(int * data)
{
    extern __shared__ int a[]; // size 64
    int tid = threadIdx.x;
    ... if (tid < BLOCK_SIZE-1 && tid > 0)
    __syncthreads();
}
```

**Application scenario:**
- Inter-warp data races
- Shared array (GPU internal) oriented
- No atomics in code regions

- Array size: 64
- BLOCK_SIZE: 64
- WARP_SIZE: 32
- # of Warps: 2
- # of threads: 64

**read set**

**write set**
Shared array oriented data race detection

- **Synchronized code block**
  
  *A code region between two adjacent synchronization statements is called a synchronized block. Inter-warp data races on a shared array happen only in the same synchronized block.*

- **API: __syncthreads()**

- **A two-pass approach**
  
  - 1st pass: write-write data race detection
  - 2nd pass: read-write data race detection
First pass: write-write race detection

- Original copy
- First written
- Second written
- Conflict part

Shared

- Copy
- Warp 0
- Warp 1

Private

- Case 1
- Case 2

Time line

- Initialization
- First pass
- Finalization

Temporary map

Report w-w race, abort

Union copy
Second pass: read-write race detection

- original copy
- first written
- second written
- conflict part

Shared

| warp 0 | warp 1 |

Private

- compare

r-w race if any difference otherwise, no race

case 2 continues ...

time line

initialization

second pass

finalization
Correctness

- Correctness of write-write race detection
  - Value-based check, region overlaps mean write-write conflicts

- Correctness of read-write race detection
  - On condition that no write-write races exist
  - If outputs differ, must be from read-write races

- Convergence
A efficient lightweight runtime and compiler supports

- **Four operations**: **Diff, union, combine, compare**
  - Massive parallelism means high efficiency for bulk operations
  - Take diff as example

- **Original copy**

- **Private copy**

- **Two-level parallelism**
  - Warp-level parallelism (32 warps)
  - Intra-warp thread-level parallelism (32 threads)

- **Well scalability**

- **Byte-wise operations**

- **Data duplication**
  - Data structure promotion in compiler
Using speculation to reduce two-pass overhead

- Main kernel speculatively continues execution
- Executions overlap

Diagram:
- First pass
- Main kernel
- Child kernel
- Kernel launch
- W-w race
- Second pass
- Diff writes
- Output comparison
- R-w race
Evaluation

- **Platform settings**
  - Host machine: Xeon E5-2643 with 3.3GHz, 16GB memory
  - Device: Nvidia Tesla K20c GPU
    - 1 Kepler GK110 architecture, 2496 cores, 208GB/sec memory bandwidth, 5GB memory
    - Dynamic parallelism

- **Two benchmarks on data mining**
  - EM
  - Co-clustering

- **Compared with GRace[Zheng et al. PPoPP’11]**
## Performance overhead

<table>
<thead>
<tr>
<th>Comparisons</th>
<th>EM</th>
<th>Co-clustering</th>
</tr>
</thead>
<tbody>
<tr>
<td>native</td>
<td>60.27 (1x)</td>
<td>28.91 (1x)</td>
</tr>
<tr>
<td>LDetector</td>
<td>250.15 (4.15x)</td>
<td>99.46 (3.44x)</td>
</tr>
<tr>
<td>LDetector-spec</td>
<td>177.42 (2.94x)</td>
<td>107.04 (3.70x)</td>
</tr>
<tr>
<td>GRace-stmt</td>
<td>104445.9 (1733x)</td>
<td>1643781.88 (56858x)</td>
</tr>
<tr>
<td>GRace-addr</td>
<td>18866.83 (313x)</td>
<td>7236.8 (250x)</td>
</tr>
</tbody>
</table>

- measured time in milliseconds (slowdown)
## Memory overhead

<table>
<thead>
<tr>
<th>Comparisons</th>
<th>EM</th>
<th>Co-clustering</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDetector</td>
<td>16K, 2.6M</td>
<td>16K, 2M</td>
</tr>
<tr>
<td>LDetector-spec</td>
<td>0K, 2.7M</td>
<td>0K, 2.2M</td>
</tr>
<tr>
<td>GRace-stmt</td>
<td>1.1K, 1000M*</td>
<td>1.1K, 1000M*</td>
</tr>
<tr>
<td>GRace-addr</td>
<td>1.1K, 18M</td>
<td>1.1K, 9M</td>
</tr>
</tbody>
</table>

- shared memory usage, global memory usage
- * Instrument as many memory accesses when memory overhead gets up to 1GB
Future work

• Conclusion
  ○ A two-pass approach to detect data races in shared memory
  ○ Zero per memory access monitoring
  ○ A high efficient lightweight runtime
  ○ Speculative parallelization optimization

• Future work
  ○ Support atomics, locks
  ○ Solutions on data races in global memory
    ▪ Memory consumption explosion
• Thanks!

• Q & A
Data structure expansion

- Data privatization techniques
First pass: write-write race detection

- original copy
- first written
- second written
- conflict part

- shared
- private
  - warp 0
  - warp 1

- copy
- diff
case 1
diff
case 2

- union copy
- temporary map
  - report w-w race, abort

- time line
  - initialization
  - first pass
  - finalization
Second pass: read-write race detection

- original copy
- first written
- second written
- conflict part

- shared
- private

- warp 0
- warp 1

- time line
- initialization
- second pass
- finalization

- compare

- r-w race if any difference
  otherwise, no race
Our contribution

- **Target:**
  - Shared array oriented
  - With no atomics in code regions
  - Inter-warp data races
Using speculation to reduce two-pass overhead

- kernel
- compiled kernel
- detected candidate
- setup environment for child kernel
- child kernel
- kernel launch
- Diff writes
- w-w race
- kill main kernel
- r-w race
- kill main kernel
- synchronization block
- will-be-detected sync block
- operations introduced by main kernel
- child kernel launch
- race detection by child kernel
- race happens, kill main kernel

operations introduced by main kernel
child kernel launch
race happens, kill main kernel
A two-pass approach

Initialization

first-execution

Finalization

original copy

first written

second written

conflict part

shared

copy

warped

warp 0

warp 1

case 1

diff

case 2

diff

temporary map

report: write write race

union copy

union copy

read write race

if any difference,
otherwise, no race

write

write

write

write

write

write

write

write

write

write

write

write

write

write

write

write

write