Scalable SMT-based Verification for GPU Kernel Functions

Guodong Li and Ganesh Gopalakrishnan

Salt Lake City, UT 84112, USA

www.cs.utah.edu/fv/PUG
GPUs are fueling HPC advances!

• Tianhe-1a: “World’s fastest supercomputer” - uses Nvidia GPUs

• Personal supercomputers used for scientific research (biology, physics, ...) increasingly based on GPUs
Growing Computational Power of GPUs

(courtesy of Nvidia)
Performance v.s. Correctness

• When properly programmed, GPUs are 20x to 100x faster than multicore CPUs; but, we must
  – keep all the fine-grained GPU threads busy
  – ensure coalesced data movements from the global memory to the shared memory
  – minimize bank conflicts when the GPU threads step through the shared memory

• Performance optimizations are error-prone!
  – e.g. removing a barrier can improve performance, but may induce data races
State of the Art of GPU Code Verification

• Many ad-hoc debuggers

  – No coverage guarantees
    • Can’t analyze over all feasible inputs, schedules and configurations

  – Can’t detect mismatched barriers (and races)
    • machines known to hang!

• No prior formal verification research / tools supporting GPU program analysis
Our Contributions

• We offer the first formal symbolic analyzer for CUDA kernels

• Currently Nvidia's CUDA is the dominant GPU architecture/framework

• Our tool is called PUG

• PUG finds bugs and issues in realistic CUDA kernels
Rest of the Talk

• Brief Introduction of CUDA
• Formal Verification of CUDA Kernels
  – Concurrency encoding
  – Identifying concurrency bugs
• Results
What is the CUDA language?

• A simple dialect of C lacking in heap allocation, recursive functions, etc.
• However it has a notion of thread-blocks and thread teams that run in SIMD mode
• Threads can synchronize through barriers
Kernel = Many Concurrent Threads

- Many threads execute the same kernel
  - Each thread executes the same code...
  - ...on different data based on its threadID
- Threads are grouped into thread blocks
  - threads in the same block may be synchronous with barriers

```c
float x = input[threadID];
float y = func(x);
output[threadID] = y;
...```

```
threadID 0 1 2 3 4 5 6 7
```
Example: Increment Array Elements

**CPU program**

```c
void inc_cpu(float* a, float b, int N) {
    for (int idx = 0; idx < N; idx++)
        a[idx] = a[idx] + b;
}

void main() {
    .....  
    increment_cpu(a, b, N);
}
```

**CUDA program**

```c
__global__ void inc_gpu(float* A, float b, int N) {
    int tid = blockIdx.x*blockDim.x+ threadIdx.x;
    if (tid < N)
}

void main() {
    .....  
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize ) );
    increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Example: Increment Array Elements
Example: Increment Array Elements

Increment N-element vector A by scalar b

```c
__global__ void inc_gpu(float*A, float b, intN) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < N)
}
```
The case with Race or Deadlock

Increment N-element vector A by scalar b

<table>
<thead>
<tr>
<th>tid</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
A[0]+b \quad \cdots \quad A[15]+b
\]

\[
\text{__global__ void inc_gpu(}
    \text{float}\* A, \text{float} b, \text{int} N) \{\}
    \text{int tid = blockIdx.x * blockDim.x + threadIdx.x;}\]
    \text{if (idx < N)}\]
\} \]

RACE!

\[
\text{__global__ void inc_gpu(}
    \text{float}\* A, \text{float} b, \text{int} N) \{\}
    \text{int tid = blockIdx.x * blockDim.x + threadIdx.x;}\]
    \text{if (idx < N)}\]
    \text{__syncthreads();}\]
\} \]

DEADLOCK!
Why can’t traditional shared memory thread verifiers work well for CUDA?

• Need to model SIMD execution semantics

• Model control-flow and barriers on flow paths

• Model memory hierarchy, banks, local/shared..
Internal Architecture of PUG

Kernel Program In C

Analyzer supported by LLNL Rose

Verification Conditions in SMT format

SMT checker for unsatisfiability

configuration

UNSAT: The instance is “OK” (no bug is found)

SAT: The instance has bugs (the SAT instance gives a counter example)
PUG in Action : Example with a Race

```c
void __global__ kernel
(int *a, int b, int N) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < N)
        a[idx] = a[idx-1] + b;
}
```

Race occurs for t1 = 0 and t2 = 1.

**Witness**

**Tells us**

**Exactly how**

**Race happens!**

(Found conflict:
There exists a race on variable a.
solving time = 0.01s)

```
(and (and (/= t1.x t2.x)
        (bv-lt t1.x bdim.x)
        (bv-lt t2.x bdim.x))
    (and (= idx1@t1 (bv-add (bv-mul bid.x bdim.x) t1.x))
        (ite (bv-lt idx1@t1 N0) true (= a1 a0)))
    (and (= idx1@t2 (bv-add (bv-mul bid.x bdim.x) t2.x))
        (ite (bv-lt idx1@t2 N0) true (= a1 a0)))
    true
    true
    (or (and (bv-lt idx1@t1 N0)
                  (bv-lt idx1@t2 N0)
                  (= (bv-sub idx1@t1 0b0000000001) idx1@t2))
        (and (bv-lt idx1@t1 N0)
                  (bv-lt idx1@t2 N0)
                  (= idx1@t1 idx1@t2)))))
```

(satisfiable
```
(= bid.x 0b00000000000)
(= bdim.x 0b00000010000)
(= t1.x 0b00000000001)
(= t2.x 0b00000000000)
(= idx1@t1 0b00000000001)
(= N0 0b11111111110)
(= idx1@t2 0b00000000000)
(= (a1 0b00000000000) 0b0000000001)
(= (a0 0b00000000000) 0b00000000000)
```

!!!!!!! Found conflict:
Review: Encoding sequential execution

Program:

\[\begin{align*}
\text{if } i > 0 \{ \\
  & j = i \times 10; \\
  & k = j - i; \\
\}
\text{else} \\
& k = i + k;
\end{align*}\]

Symbolic Constraint  \( P: \)

\[\text{ite } (i0 > 0, \\
  \quad j1 = i0 \times 10 \land k1 = j1 - i0, \\
  \quad k1 = i0 + k0 \land j1 = j0 \\
)\]

Claim:  
Input:  \( i = x, \ k = y \)  
Output:  \( k = x + y? \)

\[\begin{align*}
P \land \\
i0 = x \land k0 = y \land \\
k1 \neq x + y
\end{align*}\]

Satisfiable:  
the claim is false

The symbolic constraint models the behavior of the program, 
e.g. the relation between inputs and outputs
How to Model CUDA Programs?

Number the thread IDs:

<table>
<thead>
<tr>
<th>Block 1</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>t_0</td>
<td>t_1</td>
<td>t_2</td>
<td>t_3</td>
<td>t_4</td>
<td></td>
</tr>
<tr>
<td>Thread (0, 0)</td>
<td>Thread (1, 0)</td>
<td>Thread (2, 0)</td>
<td>Thread (3, 0)</td>
<td>Thread (4, 0)</td>
<td></td>
</tr>
<tr>
<td>Thread (1, 1)</td>
<td>Thread (1, 1)</td>
<td>Thread (2, 1)</td>
<td>Thread (3, 1)</td>
<td>Thread (4, 1)</td>
<td></td>
</tr>
<tr>
<td>t_5</td>
<td>t_6</td>
<td>t_7</td>
<td>t_8</td>
<td>t_9</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block 2</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>t_{10}</td>
<td>t_{11}</td>
<td>t_{12}</td>
<td>t_{13}</td>
<td>t_{14}</td>
<td></td>
</tr>
<tr>
<td>Thread (0, 0)</td>
<td>Thread (1, 0)</td>
<td>Thread (2, 0)</td>
<td>Thread (3, 0)</td>
<td>Thread (4, 0)</td>
<td></td>
</tr>
<tr>
<td>Thread (1, 1)</td>
<td>Thread (1, 1)</td>
<td>Thread (2, 1)</td>
<td>Thread (3, 1)</td>
<td>Thread (4, 1)</td>
<td></td>
</tr>
<tr>
<td>t_{15}</td>
<td>t_{16}</td>
<td>t_{17}</td>
<td>t_{18}</td>
<td>t_{19}</td>
<td></td>
</tr>
</tbody>
</table>

A thread local variable "v" has one copy in each thread:

\[ v^{t_0}, v^{t_1}, v^{t_2}, \ldots \]
Modeling Shared Variable Access

Different **interleavings (schedules)** lead to different results:

- **v** is shared; initially **v** = 0
  - thread **t1**
    - **v** := **v** + 1;
  - thread **t2**
    - **v** := **a**;
    - **v** := **b**;

**v**’s possible value after the execution may be 1, **b**, **a+1** or **b + 1**

- **schedule 1:**
  - **v** := 0
  - **v** := **a**
  - **v** := **b**
  - **v** := 0 + 1

- **schedule 2:**
  - ...  
  - **v** := **b**

- **schedule 3:**
  - ...  
  - **v** := **a**
  - **v** := **a + 1**

- **schedule 4:**
  - ...  
  - **v** := **b**
  - **v** := **b + 1**
Approach to encode interleavings *symbolically*

1. **Assign Schedule IDs to shared variable accesses**

   **Original program**
   
<table>
<thead>
<tr>
<th>thread t1</th>
<th>thread t2</th>
</tr>
</thead>
<tbody>
<tr>
<td>v := v + 1;</td>
<td>v := a;</td>
</tr>
<tr>
<td>v := b;</td>
<td></td>
</tr>
</tbody>
</table>

   **Program with Schedule IDs**
   
<table>
<thead>
<tr>
<th>thread t1</th>
<th>thread t2</th>
</tr>
</thead>
<tbody>
<tr>
<td>v[t_{10}] := v[t_{11}] + 1;</td>
<td>v[t_{21}] := a;</td>
</tr>
<tr>
<td>v[t_{22}] := b;</td>
<td></td>
</tr>
</tbody>
</table>

2. **Build Constraints modeling control-flow and possible interleavings**

   \[ v_{t0}, v_{t1}, v_{t2} \in \{1,2,3,4\} \]
   \[ v_{t0} < v_{t1} \land v_{t2} < v_{t2} \]

   for \( i \in \{2,3,4\} \), \( v[i]'s \) previous value is \( v[i-1] \)

   \{ the solutions of these constraints give all the possible schedules \}

   Each solution of the constraints represents a schedule. Example:

   \[
   \begin{align*}
   v[t_{21}] &= 1 & v[v[t_{11}]] &= a & v &= a \\
   v[t_{01}] &= 2 & \text{read } v[v[t_{01}]] & v &= a \\
   v[t_{22}] &= 3 & v[v[t_{22}]] &= b & v &= b \\
   v[t_{11}] &= 4 & v[v[t_{11}]] &= v[v[t_{01}]] + 1 & v &= b + 1
   \end{align*}
   \]
Schedule Reduction Exploiting Serializability

**Theorem:** For race-free programs, any schedule will do!
- Encode race-freedom checking per barrier-interval (*scalable!*)
- If barrier interval race-free, pick arbitrary schedule; then move to next barrier interval!

![Diagram showing schedule reduction](image)

A valid schedule:

![Valid schedule example](image)

A valid schedule:

![Valid schedule example](image)
An Example Kernel and its Encoding

```c
__global__
  kernel
  (unsigned*
  k)
  {
    unsigned
    s[2][3]
    =
    {{0,1,2},{3,4,5}};
    unsigned i = threadIdx.x;
    unsigned j = k[i] - i;
    if (j < 3)
      { k[i] = s[j][0]; j = i + j; }
    else
      s[1][j && 0x11] = k[i] * j;
    __syncthreads();
    k[j] = s[1][2] + j;
  }
```

Serialized version (n threads):

TRANS(t) ≡

\[
\begin{align*}
s_t^0[0] &= \lambda i \in \{0,1,2\}.i \land s_t^1[1] = \lambda i \in \{0,1,2\}.i + 3 \land
i_t^1 = t \land j_t^1 = k[k_0^t][i_t^1] - i_t^1 \land
\text{ite}(j_t^1 < 3, k[k_1^t] = k[k_1^t] - 1 \lor ([i_t^1] \mapsto s_t^1[j_t^1][0]) \land j_t^2 = i_t^1 + j_t^1
\land s_t^2 = s_t^1,
& s_t^2 = s_t^1 \lor ([1][j_t^1 \# 0x11] \mapsto k[k_2^t][i_t^1] \times j_t^1) \land
j_t^2 = j_t^1 \land k[k_1^t] = k[k_1^t] - 1) \\
k[\text{bar}_0] &= k[\text{bar}_0 - 1] \land
k[k_3^t] = k[k_3^t] \lor ([j_t^2] \mapsto s_t^2[1][2] + j_t^2)
\end{align*}
\]

TRANS(t_1, \ldots, t_n) ≡ \bigwedge_{i \in [1,n]} TRANS(t_i)

ORDER(t_1, \ldots, t_n) ≡

\[
\begin{align*}
(1) \quad & \bigwedge_{i \in [1,n]} (k_0^{t_i} < \{k_0^{t_i}, k_1^{t_i}\} < \text{bar}_0 < k_0^{t_i}) \\
(2) \quad & \text{bar}_0 < l \land \bigwedge_{i \in [1,n], j \in [0,3]} (k_1^{t_i} < l) \text{ where } l = 4n + 1. \\
(3) \quad & \text{rank(\text{bar}_0)} = 0 \land \bigwedge_{i \in [1,n], j \in [0,3]} \left(\text{rank}(k_1^{t_i}) = 4i + j\right)
\end{align*}
\]

TRANS(t_x, n) ≡

\[
\begin{align*}
s_x^0[0] &= \lambda i \in \{0,1,2\}.i \land s_x^1[1] = \lambda i \in \{0,1,2\}.i + 3 \land
i_x^1 = t \land j_x^1 = k[x - 1][i_x^1] - i_x^1 \land
\text{ite}(j_x^1 < 3, k[x] = k[x - 1] \lor ([i_x^1] \mapsto s_x^1[j_x^1][0]) \land
j_x^2 = i_x^1 + j_x^1 \land s_x^2 = s_x^1,
& s_x^2 = s_x^1 \lor ([1][j_x^1 \# 0x11] \mapsto k[x - 1][i_x^1] \times j_x^1) \land
j_x^2 = j_x^1 \land k[x] = k[x - 1]) \\
k[\text{bar}_0] &= k[\text{bar}_0 - 1] \land
k[n + x] = k[n + x - 1] \lor ([j_x^2] \mapsto s_x^2[1][2] + j_x^2)
\end{align*}
\]

TRANS(t_1, \ldots, t_n) = \bigwedge_{x \in [1,n]} TRANS(t_x, n)
Results, Part 1: Function Correctness Checking through Post-condition Assertions

<table>
<thead>
<tr>
<th>Kernels</th>
<th>#threads = 2</th>
<th>#threads = 4</th>
<th>#threads = 8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Correct</td>
<td>Bug</td>
<td>Correct</td>
</tr>
<tr>
<td>simple reduction</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>2.8</td>
</tr>
<tr>
<td>matrix transpose</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>1.9</td>
</tr>
<tr>
<td>bitonic sort</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>3.7</td>
</tr>
<tr>
<td>scalar product</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>6.9</td>
</tr>
</tbody>
</table>

Solving times in seconds; T.O indicates >5 minutes

The checker identified important poorly documented issues:

• “bitonic sort” is incorrect when #threads is not the power of 2

• “scalar product” is incorrect when ACCN is not the power of 2

• “matrix transpose” is incorrect for matrix size < block size
Focus of the rest of the talk

Other Correctness Issues
• Mismatched barriers
• Data races

Making the Analysis Scalable
• Loop abstraction and refinement
• Other SE techniques
Identifying thread divergence
(can cause deadlocks, performance loss)

Unconditional Barrier

Conditional Barrier
Checking Barrier Mismatch

BIx contains conditional barriers

These barriers match iff

$$\forall t_1, t_2 : p(t_1) = p(t_2)$$

A barrier mismatch will lead to a deadlock
Checking Shared Variable Conflicts

BI1 is conflict free iff:

\[ \forall t_1, t_2 : a_1^{t_1}, a_2^{t_1}, a_1^{t_2} \text{ and } a_2^{t_2} \text{ do not conflict with each other} \]

BI 3 is conflict free iff the following accesses do not conflict for all \( t_1, t_2 \):

\[
\begin{align*}
a_3^{t_1}, \sim p & : a_4^{t_1}, \sim p : a_5^{t_1}, \\
a_3^{t_2}, \sim p & : a_4^{t_2}, \sim p : a_5^{t_2}
\end{align*}
\]
Handling loops efficiently

• Motivation : Loops must be handled efficiently to prevent SMT explosion

• Techniques (NO loop unrolling) :
  – Loop Normalization : Convert loops to unit stride loops (not always possible)
  – Then do loop refinement by finding simple loop invariants
    • Many loop invariants automatically discovered by PUG
    • Sometimes user has to help
Loop Normalization Optimization

int k = x;
for (int i = lb; i ≤ ub; i += stride) {
    a[i]++;
    int j = i * i;
    ...
    k *= x;
}
int k = x;
for (int i = lb; i ≤ ub; i += stride) {
    a[i]++;
    int j = i * i;
    ...
    k *= x;
}

for (int i' = 0; i' ≤ (ub - lb) / stride; i'++) {
    i = i' * stride + lb;
    a[i]++;
    int j = i * i;
    ...
    k *= x;
}
Example of Loop Refinement

```
int j = 1;
for (int i = n; i > 0; i >>= 1) {
  s[tid * n + j] = ...;
  j = j * 2;
}
```

Loop Invariant:
```
i * j = n
```

the stride is non linear

```
int j = v;
for (int i = 0; i < ub; i++) {
  ...;
  j += k;
}
```

Loop Invariant:
```
j = v + i * k
```

the variable is loop carrying
(can not expressed as a linear expression of i)
Real Example with Race

```c
__global__ void computeKernel(int *d_in, int *d_out, int *d_sum) {

    d_out[threadIdx.x] = 0;
    for (int i = 0; i < SIZE/BLOCKSIZE; i++) {
        d_out[threadIdx.x] += compare(d_in[i*BLOCKSIZE+threadIdx.x], 6);
    }
    __syncthreads();

    if (threadIdx.x%2 == 0) {
        for (int i = 0; i < SIZE/BLOCKSIZE; i++) {
            d_out[threadIdx.x+SIZE/BLOCKSIZE*i] += d_out[threadIdx.x+SIZE/BLOCKSIZE*i+1];
        ...
    }

    /* The counter example given by PUG is:
        t1.x = 2, t2.x = 10, i@t1 = 1, i@t2 = 0,
        that is,
        d_out[threadIdx.x+8*i] += d_out[threadIdx.x+8*i+1];
        d_out[2+8*1] += d_out[10+8*0+1];
    */
```
Real Example with Race

```c
__global__ void computeKernel(int *d_in, int *d_out, int *d_sum) {

    d_out[threadIdx.x] = 0;
    for (int i = 0; i < SIZE/BLOCKSIZE; i++) {
        d_out[threadIdx.x] += compare(d_in[i*BLOCkSIZE+threadIdx.x], 6);
    }
    __syncthreads();

    if (threadIdx.x%2 == 0) {
        for (int i = 0; i < SIZE/BLOCKSIZE; i++) {
            d_out[threadIdx.x+SIZE/BLOCKSIZE*i] += d_out[threadIdx.x+SIZE/BLOCKSIZE*i+1];
            ...
        }
    }

    /* The counter example given by PUG is:
       t1.x = 2, t2.x = 10, i@t1 = 1, i@t2 = 0,
       that is,
       d_out[threadIdx.x+8*i] += d_out[threadIdx.x+8*i+1];
       d_out[2+8*1] += d_out[10+8*0+1];
    */
```
Real Example with Race

__global__ void computeKernel(int *d_in, int *d_out, int *d_sum) {

    d_out[threadIdx.x] = 0;
    for (int i=0; i<SIZE/BLOCKSIZE; i++) {
        d_out[threadIdx.x] += compare(d_in[i*BLOCKSIZE+threadIdx.x],6);
    }

    __syncthreads();

    if(threadIdx.x%2==0) {
        for(int i=0; i<SIZE/BLOCKSIZE; i++) {
            d_out[threadIdx.x+SIZE/BLOCKSIZE*i]+=d_out[threadIdx.x+SIZE/BLOCKSIZE*i+1];
        }
    }

    /* The counter example given by PUG is:
    t1.x = 2, t2.x = 10, i@t1 = 1, i@t2 = 0,
    that is,
    d_out[threadIdx.x+8*i]+=d_out[threadIdx.x+8*i+1];
    d_out[2+8*1]+=d_out[10+8*0+1];
    */
Other Details

• **Incremental Solving**
  – Reuse previous solving results to improve performance

• **Slicing**
  – Discard assignments irrelevant to the accesses to shared variables

• **Handling Aliasing**
  – Handle variable references and pointers

• **Checking** performance bugs (e.g. bank conflicts)

• Implement the system in a type checking manner
### Experimental Results, Part 2
(Conflict Checking)

<table>
<thead>
<tr>
<th>Kernels (in CUDA SDK)</th>
<th>loc</th>
<th>+O</th>
<th>+C</th>
<th>+R</th>
<th>B.C.</th>
<th>Time (pass)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitonic Sort</td>
<td>65</td>
<td></td>
<td></td>
<td></td>
<td>HIGH</td>
<td>2.2s</td>
</tr>
<tr>
<td>MatrixMult</td>
<td>102</td>
<td>*</td>
<td></td>
<td></td>
<td>HIGH</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>Histogram64</td>
<td>136</td>
<td></td>
<td>*</td>
<td></td>
<td>LOW</td>
<td>2.9s</td>
</tr>
<tr>
<td>Sobel</td>
<td>130</td>
<td></td>
<td></td>
<td>*</td>
<td>HIGH</td>
<td>5.6s</td>
</tr>
<tr>
<td>Reduction</td>
<td>315</td>
<td></td>
<td></td>
<td></td>
<td>HIGH</td>
<td>3.4s</td>
</tr>
<tr>
<td>Scan</td>
<td>255</td>
<td>*</td>
<td></td>
<td></td>
<td>LOW</td>
<td>3.5s</td>
</tr>
<tr>
<td>Scan Large</td>
<td>237</td>
<td>*</td>
<td></td>
<td></td>
<td>LOW</td>
<td>5.7s</td>
</tr>
<tr>
<td>Nbody</td>
<td>206</td>
<td></td>
<td></td>
<td>*</td>
<td>HIGH</td>
<td>7.4s</td>
</tr>
<tr>
<td>Bisect Large</td>
<td>1400</td>
<td>*</td>
<td></td>
<td></td>
<td>HIGH</td>
<td>44s</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>1150</td>
<td>*</td>
<td></td>
<td></td>
<td>LOW</td>
<td>39s</td>
</tr>
<tr>
<td>Eigenvalues</td>
<td>2300</td>
<td>*</td>
<td></td>
<td></td>
<td>HIGH</td>
<td>68s</td>
</tr>
</tbody>
</table>

+ **O**: require no bit-vector overflowing

+ **C**: require constraints on the input values

+ **R**: require manual refinement

**B.C.**: measure how serious the bank conflicts are

**Time**: SMT solving time
Bugs caught in GPU class examples

We tested 57 assignment submissions from a recently completed graduate GPU class taught in our department.

<table>
<thead>
<tr>
<th>Defects</th>
<th>Barrier Error or Race</th>
<th>Refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>benign</td>
<td>fatal</td>
</tr>
<tr>
<td>13 (23%)</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

**Defects**: Indicates how many kernels are not well parameterized, i.e. work only in certain configurations

**Refinement**: Measures how many loops need automatic refinement (by PUG).
Ongoing Work (PUG-parametric, PUG-dynamic)

- **Parametric verification** (the analysis, e.g. functional correctness checking, is independent of the number of threads)
- **Equivalence checking** (tells whether the optimized version of a kernel is equivalent to the original one)

- Combination of the dynamic and the static method
- **Automatic test generation and replaying:** use the symbolic analysis to generate inputs with high coverage guarantee

- Intuitive user-interfaces (based on Eclipse)
- Parallelizing the symbolic analysis in GPU (100x speedup?)
Thank You!

Questions?
Exploring schedules by Constraining the Shc Order

Loosing the constraints
\[ v^{t_0}, v^{t_1}, v^{t_1}_1, v^{t_2}_1, v^{t_2}_2 \in \{1,2,3,4\} \]
for \( i \in \{2,3,4\} \), \( v[i] \)'s previous value is \( v[i-1] \)

Can model relaxed memory models (e.g. CUDA's accesses to GPU memory uses the Weak Consistency model).

Strengthening the constraints
\[ v^{t_0}, v^{t_1}, v^{t_1}_1, v^{t_2}_1, v^{t_2}_2 \in \{1,2,3,4\} \]
\[ v^{t_0} < v^{t_1}_1 \land v^{t_1}_1 < v^{t_2}_2 \]
for \( i \in \{2,3,4\} \), \( v[i] \)'s previous value is \( v[i-1] \)
\[ \land \]
\[ v^{t_0} < v^{t_2}_1 \]