Formal Analysis of GPU Programs with Atomics via Conflict-Directed Delay-Bounding*

Wei-Fan Chiang[1], Ganesh Gopalakrishnan[1], Guodong Li[2], and Zvonimir Rakamarić[1]

[1] School of Computing, University of Utah, USA
{wfchiang,ganesh,zvonimir}@cs.utah.edu
[2] Fujitsu Labs of America, USA
gli@us.fujitsu.com

Abstract. GPU based computing has made significant strides in recent years. Unfortunately, GPU program optimizations can introduce subtle concurrency errors, and so incisive formal bug-hunting methods are essential. This paper presents a new formal bug-hunting method for GPU programs that combine *barrier*s and *atomic*s. We present an algorithm called *conflict-directed delay-bounded scheduling algorithm* (CD) that exploits the occurrence of conflicts among atomic synchronization commands to trigger the generation of alternate schedules; these alternate schedules are executed in a *delay-bounded* manner. We formally describe CD, and present two correctness checking methods, one based on final state comparison, and the other on user assertions. We evaluate our implementation on realistic GPU benchmarks, with encouraging results.

1 Introduction

General purpose Graphics Processing Units ("GPU") are being widely deployed in both low-end (mobile) and high-end (supercomputing) systems in order to accelerate computation [12]. Unfortunately, GPU program optimizations can introduce subtle concurrency errors such as data races and deadlocks. While many tools for formally debugging GPU programs have been proposed [3,14,15, 17,29], none of these tools cater to programs that combine *barrier*s and *atomic*s—features found in popular GPU programming languages such as CUDA [23] and OpenCL [24]. In this paper, we present an extension of our tool GKLEE [17] to address this program class. The extension is based on a new scheduling algorithm called *conflict-directed delay-bounded scheduling algorithm* (CD). The subject of this paper is a formal description as well as a thorough evaluation of the CD algorithm on programs that employ CUDA atomics in subtle ways.

While programs employing *barriers* (*e.g.*, `__syncthreads()` in CUDA) and *atomic operations* (*e.g.*, *atomic add*, *atomic min*, *compare-and-swap* [11] in CUDA) are not numerous, there are many important programs, including the GPU Gem [22] called N-body simulation [19, 21] that use them. In this paper, we

* Supported by NSF CCF 1255776, OCI 1148127, and the Microsoft SEIF Award.

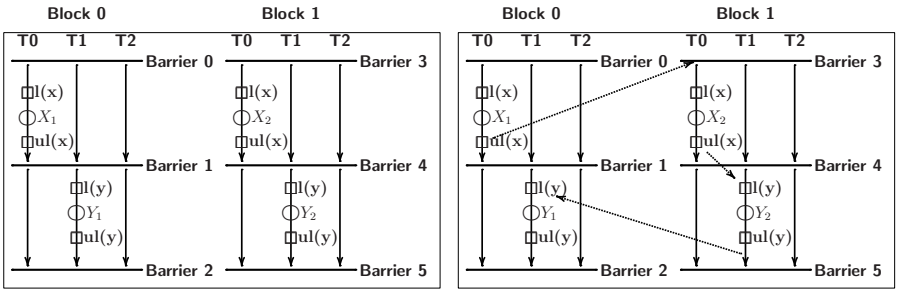

Fig. 1: Basics of CUDA, Thread Blocks, Races, and Conflicts

formally describe CD, and present two correctness checking methods, one based on final state comparison across two schedules, and the other on user assertions. We evaluate our implementation on realistic GPU benchmarks, with encouraging results; we also publicly release our benchmark suite [2].

1.1 Background

Consider a contrived GPU "kernel" program `ArraySum` that employs threads to update each location `a[i]` of an array to the value `a[(N+i-1)%N]+b`:

```
void __global__ ArraySum (int *a, int b) {
  __shared__ int temp[N];
  __syncthreads(); // Barrier 0; also Barrier 3 for threads in [512-1023]
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < N) temp[idx] = a[(N+idx-1)%N] + b;
  __syncthreads(); // Barrier 1; also Barrier 4 for threads in [512-1023]
  if (idx < N) a[idx] = temp[idx];
  __syncthreads(); // Barrier 2; also Barrier 5 for threads in [512-1023]
}
```

CUDA [1] presents three memory spaces: the *Global* space visible to all threads, the *Shared* space visible to threads within a *thread block* (typically 512 contiguous threads, abstractly referred to as `BLOCK_SIZE`), and the *Local* space visible to specific threads.[2] Assume that arrays `a` and `temp` are allocated in the *Shared* memory space. An invocation of kernel `ArraySum` with `N = BLOCK_SIZE` creates `BLOCK_SIZE` threads in block Block 0. Fig. 1 (left) provides a high level illustration of this situation.[3] The use of `__syncthreads()` in this example enforces the fact that threads before the barrier are executed before those after the barrier.

[1] Other GPU languages also have similar notions.

[2] CUDA threads are scheduled in batches called *warps*. While the typical warp-size is 32, it is not guaranteed to be so in all situations. In this paper, we take the conservative approach of taking the warp-size to be 1.

[3] Please momentarily ignore `l(x)`, `l(y)`, `ul(x)`, and `ul(y)` of this figure. Also, for simplicity, we highlight only three threads, namely T0, T1, and T2.

For uniformity, we also assume the presence of `__syncthreads()` statements at entry/exit (if not already present). For example, Barrier 0 and Barrier 2 in Fig. 1 (left) illustrate this convention.

Continuing our explanation of kernel `ArraySum`, as per CUDA conventions, all its threads execute the same code; however, each thread computes it own specific location `idx` to act upon. Each thread reads location `(N+idx-1)%N`, adds `b` to the value read, and assigns it to location `idx` of array `temp`. Now, if one were to remove Barrier 1, data races would be introduced; for example, `temp[2] = a[1]` and `a[1] = temp[1]` would be executed in parallel.

Now imagine the *same* kernel being executed concurrently by *twice* as many (*i.e.*, 1,024) threads. The threads are now split between Block 0 and Block 1 (see Fig. 1(left)), and observe that inter-block synchronization through barriers no longer works. For example, even with Barrier 1 and Barrier 4 present, accesses X_1 and X_2 can *conflict* (*i.e.*, involve the same memory location with one of them being a write). Similarly, potential conflicts are also (X_1,Y_2), (Y_1,X_2), and (Y_1,Y_2). Next, imagine that the user has realized "lock" (`l`) and "unlock" (`ul`) instructions using CUDA atomics. (Also, you may now stop ignoring the `l()` and `ul()` instructions in the figure.) Now, if we protect the pair (X_1,X_2) using the same lock variable `x`, we will avoid one data race. Likewise, assuming that (X_1,Y_2) and (Y_1,X_2) involve different addresses, we can protect (Y_1,Y_2) using another lock variable `y`. This will prevent data races among all pairs of accesses.

Following standard terminology (*e.g.*, [10, 27]), we distinguish between *ordinary* and *synchronizing* memory accesses. Two conflicting *synchronization* instructions are *not* involved in a race; however, two conflicting ordinary instructions are involved in a race. For example, lock instructions in Fig. 1 are conflicting, but are not involved in a data race. In general, property checking requires that non-commuting actions [5], such as atomic regions protected by locks, be explored under all interleavings. For illustration, suppose Fig. 1(left) executes the program in the order the barriers are numbered 0 through 5 (thus performing the Y_1 action before the Y_2 action). Then, Fig. 1(right) illustrates a *conflict-directed* alternative schedule in which the order is $Y_2; Y_1$.

Let us now turn to Fig. 2 to understand the basics of scheduling. As it turns out, data races can be detected by running a *single* schedule. In particular, as described in our previous work [17], we can execute a specific sequential schedule [1] shown by the zig-zag lines, running T0, T1,.... We call this schedule the *canonical* schedule. During a canonical schedule, suppose we record every access (read/write), and the (symbolic) path conditions under which the access occurs. Then, at the end of the canonical schedule, we can check for a race as follows. For each access pair containing at least one write, such as (P,Q), we check whether the conjunction of the path conditions can be satisfied, and also whether the address expressions become equal; if so, we report a race. If all access pairs visited along the single (canonical) schedule avoid a conflict, then no other schedule needs to be considered for race checking. Essentially, all schedules are equivalent for finding a "*first race*" [16]. Clearly, the canonical schedule can detect races across thread blocks (such as between (A,B) in the figure) as well.

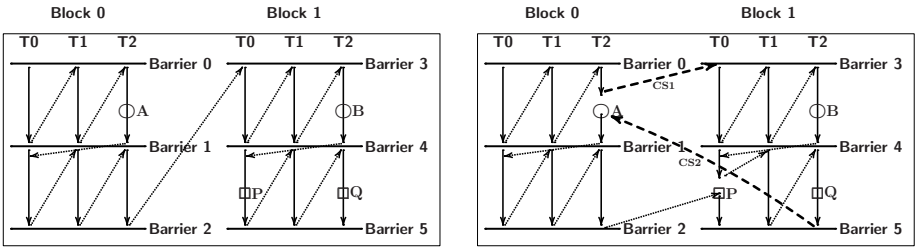

Fig. 2: Illustration of Canonical (left) and CD (right) Scheduling

Historically, it was the canonical scheduling approach that allowed us to extend the KLEE [4] *sequential* program concolic analyzer to handle GPU *concurrency*.

As illustrated previously with respect to Y_1 and Y_2, property checking in the presence of synchronization instructions requires generating alternative schedules of non-commuting actions. Unfortunately, the conflict-directed approach can require us to explore *all possible* orderings of the atomic regions (*e.g.*, similar to dynamic partial order reduction or DPOR [9]). In this context, the CD algorithm can be understood to be a suitable search bounding method inspired by previous work [8], but tailored to handle atomics in the context of canonical scheduling. Assume that we have executed a canonical schedule, and in this schedule we observe a conflict between A and P, with A encountered before P (see Fig. 2 (right)). As per the DPOR algorithm, we will be required to execute another schedule where P is executed before A. However, before we can execute P, we must execute *all* the threads in the barrier interval [3-4] (including instruction B), cross Barrier 4, and only then be able to execute P. The CD algorithm has been designed to smoothly handle such details, and also apply bounding.

In summary, the CD algorithm can be seen as a light-weight design that (1) is conflict-directed in its approach to delay-bounding (original paper [8] was not so), (2) is a light-weight approximation to DPOR while also incorporating the barrier semantics and happens-before predecessors, and (3) is built on the backbone of canonical scheduling, allowing us to incorporate other interesting heuristics related to canonical scheduling to improve performance and scalability. In particular, one such technique that enables scalability to a large number of threads is described in our recent work [18].

2 CD Algorithm for Schedule Generation

We now provide a rough sketch of the CD algorithm. Consider Fig. 2, where instructions {A,B,P,Q} are synchronization instructions (e.g., `atomicCAS`, `lock`, `atomicMin`) guarding specific regions of the user code. We first execute the CUDA program along a canonical schedule while checking for data races (among ordinary accesses) as well as recording conflicts among synchronization instructions. These conflicts are inserted into a delay list D,[4]and used to delay threads

when the program is re-executed to generate alternate schedules. We now walk through an illustrative CD execution with the help of Fig. 2(left and right).[5] Initially, let list $D = []$ and delay-bound $K = 2$. Anytime $|D| > K$, skip over this D list (details in §4). We present step-by-step several executions of the algorithm.

• First, while executing the initial canonical schedule shown in Fig. 2(left), collect the conflicting pairs, which are (A,B) and (P,Q) in our example. Based on the collected conflict pairs, obtain a list L of *delay points*. The delay points are the first instructions of each conflict pair, where the notion of first is defined by the delaying canonical scheduling order. In our example, we will obtain $L = [A, P]$.

• For each member l of L, append l at the end of D, thus obtaining an augmented D list. In our example, given that we started with $D = []$, the initial augmented D lists are $[A]$ and $[P]$. Now re-execute with D set to $[A]$ and $[P]$ in turn.

• Consider the execution with $D = [A]$. This delays A, switching (via the CS1 arrow) to T0 of Block 1. After we remove A from D, we have $D = []$. Continue the canonical execution of Block 1 entirely (Barrier 3 through Barrier 5). By this time, we would have observed instruction B in Block 1's barrier interval [3-4], and the conflict (P,Q) again. At the end of the execution of T2 at Barrier 5, the execution resumes with A, and then finishes the code between Barrier 1 and Barrier 2. The conflicts observed are (B,A) and (P,Q), and we augment the initial $D = [A]$ with B and P to obtain $D = [A, B]$ and $D = [A, P]$, respectively.

• Re-execute with $D = [A, B]$, which generates the following execution: (1) delay A; (2) go to Block 1 and there delay B; (3) resume by executing A, and then resume with B. This traverses the conflicting instructions in the order A, B, P, Q. *Notice that because of the barrier semantics, we must necessarily cycle over A and B again before we reach into P and Q.*

• Re-execute with $D = [A, P]$. (It is helpful to point out that Fig. 2(right) depicts how CD executes with $D = [A, P]$.) This generates the following execution: (1) execute till A, then delay A; (2) execute through Block 1's Barrier 4; (3) descent into the barrier interval [4-5] of Block 1; (4) delay P since it is now at the head of D; (5) switch via transition CS2 to resume A and finish the barrier interval [1-2] of Block 0; (6) finally, resume at P and finish up Block 1 entirely.

• Since we assumed delay-bound $K = 2$, after exploring the delay list $[A, P]$, we do not augment it any further. (Such augmentations generate 3-instruction delay lists which are skipped.) Instead, we backtrack and re-execute with $D = [P]$. *Implementation note: we process all smaller D sets before going to larger sets.*

The above CD execution achieves several interesting schedules, and in the end we get the following orders of dependent actions in global traces:

- `...; A ;...; B ;...; P ;...; Q ;...` (in the run with $D = []$),
- `...; B ;...; P ;...; Q ;...; A ;...` (in the run with $D = [A]$),
- `...; A ;...; B ;...; Q ;...; P ;...` (in the run with $D = [P]$),
- `...; B ;...; Q ;...; A ;...; P ;...` (in the run with $D = [A, P]$).

[4] Later, we will show that D is in fact a list of lists, but for now a simple list suffices.

[5] For simplicity, assume here that the same instructions A, B, P, Q will be encountered each time we replay. In general, the control flow will change due to global state differences caused by delaying, and different instructions are likely to be encountered.

```
1 leafid := tree[target] ;
2 if leafid ≠ LOCK then
3 |   leafid := tree[target] ;
4 |   if leafid = atomicCAS(&tree[target], leafid, LOCK) then
5 |   |   assert(leafid ≠ LOCK) ;
6 |   |   tree[target] := func() ;
```

Fig. 3: Code Excerpt from a GPU Implementation of the N-Body Algorithm

A precise formal specification of CD is in §4. Next, we present a case study and assess how well CD performs on it.

3 Motivating Example

In this section, we motivate the need for CD with a realistic example: an aggressively optimized GPU-based implementation of the well-known Barnes-Hut algorithm for performing an N-body simulation [21]. The pseudocode shown in Fig. 3 is a variant of the original code excerpt containing a bug planted by us. In the example, each thread tries to insert a node into a tree structure (encoded by *tree* array) where *target* is the index of the intended insertion. It is possible for multiple threads to have the same *target*. Variable *leafid* contains the value pointed by *target* in *tree*; if this value is *LOCK*, then the target location is currently unavailable for modification. Note that line 3 is the extraneous line—a bug—not present in the original code. It presents a redundant read from *tree[target]* that had already been done on line 1. The value read is used by the consequent *atomicCAS*.

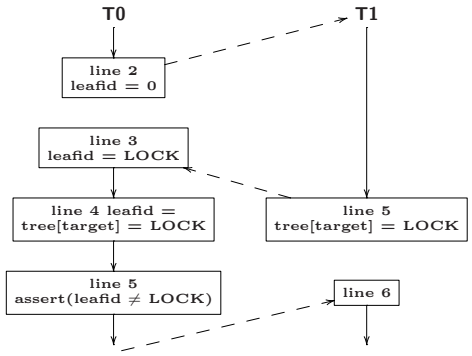

Fig. 4: Schedule Revealing N-Body Bug

The central operation of our interest is $atomicCAS(addr, expected, new)$ which atomically: (1) checks whether *addr* holds the *expected* value; (2) if so, it replaces it with *new*, else it leaves the value in *addr* unaffected; (3) it always returns the original value in *addr*. In the example, *atomicCAS* instruction on line 4 tries to put *LOCK* in *tree[target]* and returns the original value in *tree[target]*. Once a thread succeeds on condition in line 4 and proceeds to line 5, it exclusively owns *tree[target]* until it releases the ownership by assigning to it on line 6. Therefore, the old value of *tree[target]* should not be *LOCK* on line 5; if it were, it would mean that the location is already owned by another thread. We consider this to be our safety property of interest, and we encoded it as an assertion on line 5.

Fig. 4 shows why the code in Fig. 3 is erroneous. Suppose that threads T0 and T1 are accessing the same target, whose value was initialized to 0. Thread

T0 first executes lines 1-2, then is delayed, and preempted by T1. Thread T1 executes lines 1-5, then is delayed, and preempted by T0. Thread T0 on line 3 reads *leafid* from *tree[target]*; since its value is $LOCK$, after T0 executes line 3, *leafid* is $LOCK$. (If buggy line 3 was not introduced, *leafid* would still be a non-$LOCK$ value thanks to the conditional on line 2.) Then, T0 proceeds to line 4 and *atomicCAS* succeeds even though it has also succeeded for T1. On line 5, T0 triggers a failure of our safety property of interest. Note that this error takes at least two delays to be discovered. As our experimental results will show, our implementation of CD was successful in detecting such bugs within a reasonable number of overall delays and with acceptable runtimes.

4 Formal Description of CD

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ where numbers can also be viewed as sets, *e.g.*, $3 = \{0, 1, 2\}$. Consider a CUDA program *pgm* meant for execution within $BID \in \mathbb{N}$ thread blocks. Let $barid \in \mathbb{N}$ number the barriers within each block, with $lastbar(bid) \in \mathbb{N}$ being the number assigned to the last barrier within block $bid \in BID$.[6] A barrier interval (BI) is the interval (block of code) enclosed by two successive barriers. There are $TID \in \mathbb{N}$ threads per block with identifiers $tid \in TID$. For each thread, let $pc \in \mathbb{N}$ be its program counter.

We employ the tuple $cs = \langle bid, barid, tid, pc \rangle$ to specify the control state of execution. The way in which cs advances is depicted in Fig. 2. A canonical schedule begins at $InitCS = \langle 0, 0, 0, 0 \rangle$. (Here, $barid = 0$ models being in barrier interval [0-1].) Predicate $DoneBar(bid, barid, tid)$ tells whether thread tid has executed barrier $barid$ within block bid. This predicate is initialized to $InitDoneBar$ where $InitDoneBar(0, 0, 0)$ is true.

The entire state of the execution of *pgm* is captured by cs and S, where S is the data state (CUDA shared, global, and local variables). We do not elaborate on S, nor the CUDA instructions that update S. We maintain $S0$, a shadow copy of the initial data state used during program re-execution. Given the current instruction $ins = cur(S, cs, pgm)$, the state update of S caused by ins is modeled using $nxt(S, pgm, cs, ins)$.

Consider a CUDA program *pgm*, and let *ins* be Bar (barrier) for all S and cs. Informally speaking, a canonical schedule that begins in this state moves each thread until its *pc* is at the next barrier; then another thread is picked, and so on until all threads are at their next barrier. Whenever a thread is at the next barrier, the *DoneBar* predicate associated with that thread and barrier is set to true. When all the threads are at their next barrier, execution must switch over to the "next" barrier interval, determined as $nxtBI(S, pgm, cs)$. Any reasonable implementation of these abstract functions is permissible. For instance, *nxtBI* could mean either (1) *stay within the same thread block and execute the next*

[6] CUDA programs are assumed to be terminating. We also assume the usual *textually aligned* barriers. For example, CUDA programs are expected not to branch on thread IDs, with only half the threads encountering a barrier.

sequential barrier interval; or (2) *switch over to the next thread block and execute the earliest unexecuted barrier interval.*

Delay List: Let $[a, b, c]$ be a list. Then $hd([a, b, c])$ is a and $tl([a, b, c])$ is $[b, c]$. We maintain the delay list D as a list of lists.[7] For example, $D = [[], [a, b, c], [p, q, r]]$, where a, b, c and p, q, r are instructions, is a delay list.

We describe the operational semantics of CD in terms of the rules in Fig. 5. The rules are of the form $\frac{pre}{\Sigma \rightarrow \Sigma'}$, where if $pre(\Sigma)$ holds then Σ can evolve to Σ'; Σ is maintained as $\langle S0, S, cs, DoneBar, D0, D, RW, Confl \rangle$, where $cs = \langle bid, barid, tid, pc \rangle$. The CD algorithm starts with $D = [[]]$. If no conflicts are encountered, D remains $[[]]$ until the entire program execution is finished, at which point D becomes $[]$. The only inference rule provided for the case $D = []$ is to stop the entire execution of CD (Rule TERMINATION). We keep a shadow-copy of the starting delay list in $D0$. Suppose $D0 = D = [[]]$ at the beginning, and say it grows to $D = [[a, b], [p, q]]$ at the end of execution as per the CD schedule. We then re-initialize $D0$ and D according to $D0 = D = [[a, b], [p, q]]$ and re-execute the whole program. Here are the details of such a re-execution:

- When instruction a is encountered, it will be delayed, and D will be updated to $[[b], [p, q]]$, meaning that a (already delayed) need not be delayed any more. (Rules NXTTIDCSDEL and NXTBIDCS cover these cases.)
- Thereafter when b is encountered, it is delayed, and D is updated to $[[], [p, q]]$. Then (and only then) we start recording conflicts (Rule $\text{NXTPCBI}_{\text{rec}}$; notice that it checks for $hd(D) = []$). This is because the conflicts recorded must be as a consequence of delaying a, b (and further conflicts discovered in the process will later augment D). At the end of the entire *pgm* execution, let us say we have encountered conflicts in the order (i, j) and (k, l). Then we update D to $[[p, q], [a, b, i], [a, b, k]]$ (Rule RETRIG).
- Now if the delay bound K is 2, we will execute again with $[p, q]$, but skip over $[a, b, i]$ and $[a, b, k]$ (Rule BOUND).

Read-Write Set: We maintain a read-write set RW that records all reads and writes encountered; function $rwOf(ins)$ obtains the reads and writes of instruction ins. These will be used for race checking and also for forming conflicts. (We do not detail race checking here.) Entries will be added to RW as/when memory accesses (reads/writes) are encountered (Rule NXTPCBI).

Conflict List: We maintain a conflict list $Confl$ as a list of pairs of the kind shown above, *e.g.*, $Confl = [(i, j), (k, l)]$. $Confl$ is updated via function rec only when $hd(D) = []$ (Rule $\text{NXTPCBI}_{\text{rec}}$). At the end of *pgm*, we will change $D0$ from $[[a, b], [p, q]]$ to $[[p, q], [a, b, i], [a, b, k]]$.

Delay Bounding, Termination, Retriggering: Whenever $length(hd(D0)) > K$, we update D to $tl(D)$, which achieves delay bounding (Rule BOUND). The algorithm terminates when $D = D0 = []$ (Rule TERMINATION). On the other hand, if $D \neq []$, $length(hd(D0)) \leq K$, and $DoneBar(bid, lastbar(bid), tid)$ for all bid, tid, we retrigger the execution with the augmented (function aug) D and

[7] In our implementation, we maintain D sorted ascending in size. This allows all shorter delay sequences to be executed before executing any longer delay sequence. This is purely a heuristic, and has no bearing on the overall correctness of CD.

• TERMINATION:

$$\frac{D = []}{\langle S0, S, cs, DoneBar, D0, D, RW, Confl\rangle \rightarrow STOP}$$

• BOUND:

$$\frac{D \neq [] \ \wedge \ length(hd(D0)) > K}{\langle S0, S, cs, DoneBar, D0, D, RW, Confl\rangle \rightarrow \langle S0, S, cs, DoneBar, tl(D0), tl(D), RW, Confl\rangle}$$

Assume $D \neq [] \wedge length(hd(D0)) \leq K$ is a part of the precondition of the rules below.

• RETRIG:

$$\frac{\forall b \in BID : \ \forall t \in TID : \ DoneBar(b, lastbar(b), t) \ \wedge \ D1 = aug(D0, Confl)}{\langle S0, S, cs, DoneBar, D0, D, RW, Confl\rangle \rightarrow \langle S0, S0, InitCS, InitDoneBar, D1, D1, \emptyset, []\rangle}$$

• NXTBI:

$$\frac{\begin{array}{c} cur(S, cs, pgm) = \mathsf{Bar} \ \wedge \ \forall t \in TID \setminus \{tid\} : \ DoneBar(bid, barid + 1, t) \\ \wedge \ Db = DoneBar[\langle bid, barid + 1, tid\rangle \leftarrow true] \end{array}}{\langle S0, S, cs, DoneBar, D0, D, RW, Confl\rangle \rightarrow \langle S0, S, nxtBI(S, pgm, cs), Db, D0, D, RW, Confl\rangle}$$

• NXTTIDCS:

$$\frac{\begin{array}{c} cur(S, cs, pgm) = \mathsf{Bar} \ \wedge \ \exists t \neq tid \in TID : \ \neg DoneBar(bid, barid + 1, t) \\ \wedge \ Db = DoneBar[\langle bid, barid + 1, tid\rangle \leftarrow true] \ \wedge \ cs1 = nxtTidCS(S, pgm, cs) \end{array}}{\langle S0, S, cs, DoneBar, D0, D, RW, Confl\rangle \rightarrow \langle S0, S, cs1, Db, D0, D, RW, Confl\rangle}$$

• NXTPCBI:

$$\frac{\begin{array}{c} hd(D) \neq [] \ \wedge \ ins = cur(S, cs, pgm) \ \wedge \ ins \neq \mathsf{Bar} \\ \wedge \ ins \neq hd(hd(D)) \ \wedge \ cs1 = nxtPCBI(S, pgm, cs) \ \wedge \ RW1 = add(RW, rwOf(ins)) \end{array}}{\langle S0, S, cs, DoneBar, D0, D, RW, Confl\rangle \rightarrow \langle S0, S, cs1, DoneBar, D0, D, RW1, Confl\rangle}$$

• NXTTIDCSDEL:

$$\frac{\begin{array}{c} hd(D) \neq [] \ \wedge \ cur(S, cs, pgm) = hd(hd(D)) \ \wedge \ cs1 = nxtTidCS(S, pgm, cs) \\ \wedge \ \exists t \neq tid \in TID : \ \neg DoneBar(bid, barid, t) \ \wedge \ D1 = D[hd(D) \leftarrow tl(hd(D))] \end{array}}{\langle S0, S, cs, DoneBar, D0, D, RW, Confl\rangle \rightarrow \langle S0, S, cs1, DoneBar, D0, D1, RW, Confl\rangle}$$

• NXTBIDCS:

$$\frac{\begin{array}{c} hd(D) \neq [] \ \wedge \ cur(S, cs, pgm) = hd(hd(D)) \ \wedge \ cs1 = nxtBidCS(S, pgm, cs) \\ \wedge \ \forall t \in TID \setminus \{tid\} : \ DoneBar(bid, barid, t) \ \wedge \ D1 = D[hd(D) \leftarrow tl(hd(D))] \end{array}}{\langle S0, S, cs, DoneBar, D0, D, RW, Confl\rangle \rightarrow \langle S0, S, cs1, DoneBar, D0, D1, RW, Confl\rangle}$$

• NXTPCBI$_{\text{rec}}$:

$$\frac{\begin{array}{c} hd(D) = [] \ \wedge \ ins = cur(S, cs, pgm) \ \wedge \ ins \neq \mathsf{Bar} \\ \wedge \ cs1 = nxtPCBI(S, pgm, cs) \ \wedge \ Confl1 = rec(Confl, RW, rwOf(ins)) \end{array}}{\langle S0, S, cs, DoneBar, D0, D, RW, Confl\rangle \rightarrow \langle S0, S, cs1, DoneBar, D0, D1, RW, Confl1\rangle}$$

Fig. 5: Operational Semantics of the CD Algorithm

$D0$, with cs reset to $InitCS$, S to $S0$, $DoneBar$ to $InitDoneBar$, RW to $\emptyset$, and $Confl$ to [] (Rule RETRIG).
Setting $DoneBar$**:** When $cur(S, cs, pgm)$ equals Bar, we update $DoneBar$ to true for $\langle bid, barid + 1, tid\rangle$, and cs to $nxtBI(S, pgm, cs)$ (Rules NXTBI, NXTTIDCS).
Staying within a BI upon Delay: If $cur(S, cs, pgm) = hd(hd(D))$, we set D to $D[hd(D) \leftarrow tl(hd(D))]$, *i.e.*, $hd(D)$ is replaced with $tl(hd(D))$ in D. Now, if there is a tid for which $DoneBar(bid, barid, tid)$ is false, we stay in the same BI and use function $nxtTidCS(S, pgm, cs)$ to update cs (Rule NXTIDCSDEL).
Moving over to another BI upon Delay: When $DoneBar$ is true of all the threads except a thread tid, and this thread is delayed, we move over to the next block in the scheduling order. Such a block must exist because (1) we are replaying a schedule already traversed before, but with an instruction to delay, (2) we recorded the *first* part of a conflict pair, (3) which means there is another instruction (conflict partner in the pair) that is "yet to be seen", and (4) we will hit that instruction. Our selection policy in this case is to context-switch to the next bid (in a modulo fashion) and the lowest $barid$ such that for that $bid, barid$, there is a lowest ranked tid for which $DoneBar$ is false. We will use function $nxtBidCS(S, pgm, cs)$ to return this control state (Rule NXTBIDCS).

5 Experimental Results

We have implemented CD in an extension of our tool GKLEE called GKLEE$_{atm}$. GKLEE$_{atm}$ was evaluated using the following CUDA benchmarks [2]: **nbody:** the classical Barnes-Hut N-body algorithm [21], [260 LOC]; **tsp:** traveling salesman algorithm [28], [130 LOC]; **aMin:** implements atomicMin for double-precision floating point, [20 LOC]; **aMinUpdate:** use of atomicMin to set a shared location to min, [35 LOC]; **bintree:** tree insertion designed similar to wait-free ray tracing cache [7], [75 LOC]. The N-body (nbody) and traveling salesman (tsp) benchmarks are real-life CUDA programs; others are synthetic benchmarks we modeled after real programs. We created both bug-free and buggy versions for each benchmark. Each buggy benchmark contains a non-trivial realistic bug related to a potential algorithm implementation error. We also inserted assertions for checking correctness, which is a commonly used approach by programmers. The first four benchmarks (nbody, tsp, aMin, aMinUpdate) contain *lost-atomicity* bugs similar to the bug shown in §3. Such bugs are commonly created by programmers when they are trying to prevent side-effects of preemptions using atomics, but fail at their attempt. Our last benchmark, bintree, contains a *missing-atomicity* bug caused by unprotected shared memory accesses. In the experiments, we test two versions of GKLEE$_{atm}$ with different conflict selection policies: (1) unoptimized picks any detected conflict to trigger the generation of a new schedule; (2) optimized picks conflicts containing at least one read/write belonging to a "conditional atomic operation" such as atomicCAS.

For all of our benchmarks a delay bound of $K = 2$ was sufficient. We performed two sets of experiments with two different CUDA configurations: (1) 2 thread-blocks with each of them containing one thread, and (2) 3 thread-blocks

benchmark	buggy						bug-free					
	unoptimized			optimized			unoptimized			optimized		
	#sch.	t.	rlt.	#sch.	t.	rlt.	#sch.	t.	rlt.	#sch.	t.	rlt.
nbody(pa)	221	910	TP	64	182	TP	356	1134	TN	106	271	TN
nbody(fc)	44	278	TP	17	52	TP	356	1231	TN	106	295	TN
tsp(pa)	4	40	TP	4	40	TP	39	432	TN	27	293	TN
tps(fc)	4	41	TP	4	40	TP	39	426	TN	27	297	TN
aMin(pa)	25	28	TP	25	28	TP	53	57	TN	53	57	TN
aMin(fc)	4	5	TP	4	7	TP	53	56	TN	53	57	TN
aMinUpdate(pa)	18	32	TP	10	11	TP	51	81	TN	27	33	TN
aMinUpdate(fc)	18	32	TP	10	11	TP	51	88	TN	27	34	TN
bintree(pa)	83	90	FN	53	56	FN	83	90	TN	61	66	TN
bintree(fc)	2	3	FP	2	3	FP	2	3	FP	2	3	FP

(a) Experimental Results for 2 Thread-Blocks with 1 Thread Each

benchmark	buggy						bug-free					
	unoptimized			optimized			unoptimized			optimized		
	#sch.	t.	rlt.	#sch.	t.	rlt.	#sch.	t.	rlt.	#sch.	t.	rlt.
nbody(pa)	448	2414	TP	126	429	TP	1195	4900	TN	336	1019	TN
nbody(fc)	83	606	TP	28	126	TP	1195	5366	TN	336	1137	TN
tsp(pa)	4	53	TP	4	56	TP	114	1738	TN	60	1019	TN
tps(fc)	4	55	TP	4	55	TP	114	2331	TN	60	1091	TN
aMin(pa)	107	117	TP	107	117	TP	431	463	TN	431	463	TN
aMin(fc)	6	7	TP	6	7	TP	431	464	TN	431	465	TN
aMinUpdate(pa)	6	8	TP	4	5	TP	653	912	TN	294	361	TN
aMinUpdate(fc)	6	7	TP	4	5	TP	653	882	TN	294	350	TN
bintree(pa)	14	17	TP	191	202	FN	835	902	TN	405	431	TN
bintree(fc)	2	3	FP	2	3	FP	2	3	FP	2	3	FP

(b) Experimental Results for 3 Thread-Blocks with 1 Thread Each

Fig. 6: Experimental Results. *pa* tags benchmarks with manually inserted assertions for correctness checking; *fc* tags benchmarks where final state comparison is performed for correctness checking; *#sch.* gives number of schedules explored; *t.* gives runtimes in seconds; *rlt.* gives analysis results.

with each of them containing one thread. Other configurations were not necessary due to the high symmetry of CUDA programs.

5.1 Results and Discussion

Our experimental results are shown in Fig. 6. The result (*rlt.*) column shows the outcome of the analysis:

- true-positive (TP): a true bug was reported (successful detection);
- true-negative (TN): no bug reported, none exists (no false alarm or omission);
- false-positive (FP): a bug was reported, but no error exists (false alarm);
- false-negative (FN): no bug was reported, but a real bug exists (omission).

Note that none of these bugs can be detected using previous tools due to the lack of support for atomic operations.

Comparison Between Different Thread Configurations. Benchmarks in Fig. 6a use fewer blocks/threads than those of Fig. 6b. Using fewer blocks generates fewer conflicts and therefore also fewer schedules to explore. However, using the two-block configuration for the bintree benchmark results in GKLEE$_{atm}$ missing a bug, since bintree requires at least three threads to trigger the bug scenario.

```
1  while true do
2      if T[curr] ≠ null then
3          if T[curr] ≤ val then child := curr+2 ;
4          else child := curr+1 ;
5          if T[child] = null then
6              new := atomicAdd(&TSize, 3) ;
7              T[new] := val ;
8              T[child] := new; break ;
9          else curr := T[child] ;
10     else
           if null = atomicCAS(&T[curr], null, val) then break ;
11 assert(the tree is valid and connected) ;
```

Fig. 7: Pseudocode Showing a Bug in our bintree Benchmark

Next we further elaborate why our bintree benchmark bug requires at least three threads to be discovered.

Fig. 7 gives an abstraction of our bintree kernel. The tree is encoded as usual into an array T, where three consecutive elements in T denote a node. The first of these elements is the value of the node (hence the index of this element is also the index of the node). The value of the second element is the index of the left subtree (node), which contains all values less than the value of the current node. The value of the third element is the index of the right subtree (node), which contains all values greater than or equal to the value of the current node. In the pseudocode, *curr* denotes the index of the current traversed node, *child* denotes the index of the subtree to visit, and *new* denotes the index of a newly created node. Lines 2-9 perform tree traversal and the insertion of new nodes, lines 3-4 decide which subtree to traverse, and lines 5-8 insert a new node into an empty subtree. Line 9 sets the current traversed index to continue the tree traversal. The introduced bug is on line 8 where the user should have used an *atomicCAS* operation to update $T[child]$ and continue the tree traversal upon failure. The buggy code instead directly updates the tree, resulting in dangling nodes.

Fig. 8 illustrates a buggy scenario where T0 builds the root node (line 10), and T1 and T2 are concurrently attempting to insert a value (*val*) into the tree. Suppose that T0 has value 1 to insert, T1 has value 2, and T2 has value 3. First, T0 runs through the code and generates the root node. Then, T1 traverses the tree and decides to insert value 2 in the right subtree of the root node. However, it is preempted just before line 8. Thread T2 then traverses the tree and inserts value 3 in the right subtree of the root node. When T1 resumes, it overwrites

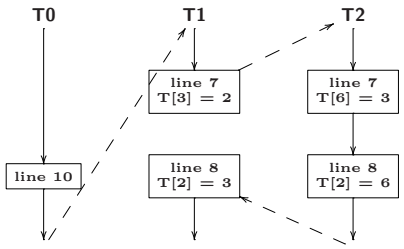

Fig. 8: Schedule Revealing bintree Bug

the right subtree with value 2. The final tree is (1, 3, null, 2, null, null, 3, null, null). The node (3, null, null) is a dangling node and the tree is not connected. Note that while this scenario has 3 context-switches, it requires just one delay (of T1) to be discovered.

Comparison Between Different Conflict Selection Policies. We observe that using the optimized strategy explores fewer schedules and produces the same results as the unoptimized strategy, except for bintree under the three-block configuration. The most remarkable savings are obtained for the nbody kernel: our conflict selection optimization reduces the number of schedules to explore by nearly a third (221 versus 64 in Fig. 6a), while producing the same test outcomes. However, in general, this optimization might prune a certain "critical conflict" necessary to trigger a schedule leading to a bug. For example, in Fig. 6b, the bug in bintree can only be detected without conflict selection optimization. The "lost atomicity" bug in this example pertains to two instances of line 8 in Fig. 7. In a sense, since the programmer "forgot" to guard $T[child] = new$ on line 8 with an atomic construct, these lines are in a real data race.

Comparison Between Different Correctness Checking Strategies. Our two correctness checking strategies, *fc* and *pa*, produce different outcomes on 3 of our benchmarks: bintree, aMin, and nbody. In particular, *fc* produces a false positive outcome for bintree in all experiments. This is because the bintree benchmark inherently produces two distinct states (*i.e.*, bit-level state layouts) even for two logically equivalent trees. Therefore, checking correctness by simply comparing final states will end up generating a false alarm for GPU programs that generate nondeterministic bit-state outcomes (which are nevertheless logically equivalent). In contrast, for aMin and nbody, the *fc* strategy shows distinct advantages, often exploring only a fifth of the number of schedules (*e.g.*, 448 versus 83 for nbody) before finding a bug. Our results suggest that the final state comparison strategy is better overall in terms of the number of schedules that get explored before a bug is reached.

6 Discussion, Related Work, and Conclusions

GKLEE$_{atm}$ is the first tool we know that employs the combination of conflict-directed and delay-bounded testing. Its CD algorithm extends the canonical scheduling method which has already proven successful [17]. Since the intended semantics of most CUDA programs is sequential, race/conflict freedom is the norm. In these cases, we avoid generating interleavings. GKLEE$_{atm}$ also inherits additional features of GKLEE (*e.g.*, test-case generation, test-case reduction, bank conflict and memory coalescing estimation) that are now available for examples that employ CUDA atomics.

CD is not formally complete. For example, the initial canonical schedule in §2 gave us the delay points $[A, P]$. Suppose we had executed the sequential schedule that began with thread T2 of Block 1, we would initially encounter $[B, Q]$. At that point, the control flows may change, and instead of seeing $[P, A]$ later, we may see some other conflicting operations (or perhaps no conflicts at all). Our future

work may combine static analysis with CD to determine which other sequential schedules to consider. The manner in which CD is realized using the abstract functions *nxtTidCS*, *nxtBidCS*, *nxtBI*, and *nxtPCBI* gives us the intriguing possibility of choosing these functions based on static analysis or randomizing them for separate runs. In the paper, we have explored prioritizing atomics involving conditional comparisons, such as atomicMin (see §5).

Race-directed Testing. Race-directed testing for traditional multithreaded programs was proposed previously [25, 26]. It detects races in a schedule and takes them as "hints" for introducing context-switches, which in turn generate more schedules for detecting property violations. GKLEE$_{atm}$ extends race-directed testing with delay-bounding, and selects a subset of races suitable for testing CUDA programs with atomic operations.

Bounded Testing. Bounded testing is a well-known technique for analysis of traditional multithreaded programs (*e.g.*, [8, 13, 20]). It was empirically shown that most of concurrency bugs can be detected by introducing only a limited amount of nondeterminism (*e.g.*, context-switches, delays). GKLEE$_{atm}$ takes this approach to efficiently detect bugs in CUDA programs, and mixes it with our conflict-directed feedback for obtaining new delay locations. Traditional bounded testing typically does not employ such feedback and blindly introduces delays at all potential conflict locations.

GPU Program Testing Tools. Recently, several GPU program testing tools were proposed [3, 6, 14]. Test amplification [14] starts with dynamic testing of GPU programs, but employs a test amplification technique to generalize the results of the dynamic analysis over a large space of inputs. The amplification relies on a static information flow analysis to prune inputs not affecting the property to be verified. GPUVerify [3] performs symbolic analysis of GPU programs similar to PUG [15], but provides a precise CUDA operational semantics for predicated executions. KLEE-CL [6] employ symbolic analysis to perform equivalence checking for C programs and their accelerated OpenCL implementations. KLEE-CL can also check data race for OpenCL programs. However, none of these tools support atomic instructions.

Conclusions. We propose the first conflict-directed delay-bounding approach to schedule multithreaded programs. We formally describe our CD algorithm that implements this approach as a new tool GKLEE$_{atm}$, and apply it for testing aggressively optimized GPU programs that employ atomic instructions and barriers. Furthermore, we evaluate several scheduling policies and property checking approaches. In addition to detecting subtle concurrency bugs, CD proves to be a light-weight and tailorable approximation to more complete (but also more expensive) algorithms such as DPOR. Our future work will include exploiting thread symmetry [18] and informing concolic verification through static analysis.

References

1. H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be elim-

inated. In *POPL*, pages 487–498, 2011.
2. http://www.cs.utah.edu/fv/CdDb.
3. A. Betts, N. Chong, A. F. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: a verifier for GPU kernels. In *OOPSLA*, pages 113–132, 2012.
4. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.
5. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Cheking*. MIT Press, 1999.
6. P. Collingbourne, C. Cadar, and P. Kelly. Symbolic testing of OpenCL code. In *Haifa Verification Conference*, 2011.
7. K. Debattista, P. Dubla, L. P. P. dos Santos, and A. Chalmers. Wait-free shared-memory irradiance caching. *Comp. Graphics and Applications*, 31(5):66–78, 2011.
8. M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded scheduling. In *POPL*, pages 411–422, 2011.
9. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.
10. B. Goetz, J. Bloch, J. Bowbeer, D. Lea, D. Holmes, and T. Peierls. *Java Concurrency in Practice*. Addison-Wesley Longman, Amsterdam, 2006.
11. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. 2008.
12. W.-M. W. Hwu. *GPU Computing Gems Emerald Edition*. 2011.
13. A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Form. Methods Syst. Des.*, 35(1):73–97, Aug. 2009.
14. A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner. Verifying GPU kernels by test amplification. In *PLDI*, pages 383–394, 2012.
15. G. Li and G. Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *FSE*, pages 187–196, 2010.
16. G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. Rajan. GKLEE technical report. http://www.cs.utah.edu/fv/GKLEE/gklee_tr.pdf.
17. G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. GKLEE: concolic verification and test generation for GPUs. In *PPOPP*, 2012.
18. P. Li, G. Li, and G. Gopalakrishnan. Parametric flows: Automated behavior equivalencing for symbolic analysis of races in CUDA programs. In *SC*, 2012.
19. M. Méndez-Lojo, M. Burtscher, and K. Pingali. A GPU implementation of inclusion-based points-to analysis. In *PPOPP*, pages 107–116, 2012.
20. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455, 2007.
21. CUDA implementation of the tree-based Barnes-Hut N-body algorithm. http://www.gpucomputing.net/?q=node/1314.
22. H. Nguyen. *GPU Gems 3*. Addison-Wesley Professional, first edition, 2007.
23. Nvidia. CUDA parallel computing platform. http://www.nvidia.com/object/cuda_home_new.html.
24. OpenCL. OpenCL - the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl.
25. K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.
26. K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Haifa Verification Conference*, pages 166–182, 2007.
27. D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. 2011.
28. http://www.cs.txstate.edu/~burtscher/research/TSP_GPU/.
29. M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal. GRace: a low-overhead mechanism for detecting data races in GPU programs. In *PPOPP*, pages 135–146, 2011.