Efficient Inspected Critical Sections in Data-Parallel GPU Codes

Thorsten Blaß(✉), Michael Philippsen, and Ronald Veldema

Programming Systems Group, Friedrich-Alexander University, Erlangen, Germany
{Thorsten.Blass,Michael.Philippsen,Ronald.Veldema}@fau.de

Abstract. Optimistic concurrency control and STMs rely on the assumption of sparse conflicts. For data-parallel GPU codes with many or with dynamic data dependences, a pessimistic and lock-based approach may be faster, if only GPUs would offer hardware support for GPU-wide fine-grained synchronization. Instead, current GPUs inflict dead- and livelocks on attempts to implement such synchronization in software.

The paper demonstrates how to build GPU-wide non-hanging critical sections that are as easy to use as STMs but also get close to the performance of traditional fine-grained locks. Instead of sequentializing all threads that enter a critical section, the novel programmer-guided Inspected Critical Sections (ICS) keep the degree of parallelism up. As in optimistic approaches threads that are known not to interfere, may execute the body of the inspected critical section concurrently.

Keywords: GPGPU · CUDA · SIMT · Critical section · Mutual exclusion

1 Introduction

Optimistic concurrency control – as it is implemented in Software Transactional Memory (STM) – comes with some overhead for logging and rollback [5]. This overhead grows with the number of threads that collide in their memory accesses. On asynchronous multicores often only a few of the running threads are in an atomic region at any time, whereas on a GPU with its data-parallel/lock-step execution model, all threads must enter this critical section at exactly the same time. Hence, optimistic approaches may cause significant overhead on GPUs.

```
while(atomicCAS(&lock, -1, TID) != -1); //spin
// critical section code here
atomicExch(lock, -1);
```

```
bool leaveLoop = false; //thread local
while(!leaveLoop){
    if(atomicCAS(&lock, -1, TID) == TID){
        // critical section code here
        leaveLoop = true;
        atomicExch(lock, -1);
    }
    // point of convergence
}
```

Fig. 1. Spin lock implementations, with and w/o a SIMT-deadlock. TID is the global thread Id.

Assume you want to study this hypothesis. You pick benchmarks from the GPU-STM community, you take (or re-implement) an STM prototype for GPUs [6, 12, 16, 20], and to gauge the overhead, you re-work the atomic regions of the benchmark codes into pessimistic concurrency control,

L. Rauchwerger (Ed.): LCPC 2017, LNCS 11403, pp. 223–239, 2019.
https://doi.org/10.1007/978-3-030-35225-7_15

Table 1. Fraction of the runs affected by dead- or livelocks; see Appendix.

	Hash table		Bank	Graph		Labyrinth	Genome	Kmeans	Vacation
Problem size	1,572,864	786,432	25,165,824	25%	75%	(512,512,7)	Configuration see Appendix		
# threads	1,572,864		25,165,824	10,280		512	811,008	3,014,656	4,194,304
FGL	43%	42%	45%	29%	42%	33%	49%	37%	53%
STM	39%	45%	49%	35%	43%	37%	50%	41%	55%

i.e., your threads simply acquire a fine-grained lock for each of the data items that they may access concurrently at runtime. If threads need to acquire multiple locks, you use a global order to avoid deadlocks. Since you know that the Single Instruction Multiple Thread (SIMT) execution model is prone to deadlocks[1] [9,11] you re-work your code as shown in Fig. 1, i.e., you pull the loop out of the `if`-statement that holds the CAS. With the resulting convergence point after the `if`-statement, regardless of the SIMT-scheduling, both sets of threads make progress; the lock is eventually released.

At this point you will understand the first motivation of our work. In our experiments and with a particular STM framework [20], our otherwise correct benchmark code (see Appendix) often hangs in dead- or livelocks that are beyond our control, see Table 1. Your mileage will vary. It depends on the size, configuration, version and vendor of your GPU, the number of threads that your code spawns, the unknown scheduling strategies that run on your GPU, . . . , and the compiler version that you are using.[2] Hence, either it works by coincidence or you need to carefully fine-tune your setup to avoid similar dead- or livelocks – both for the STM codes and for the codes with the fine-grained locks (FGL). While the FGL-codes are straightforward to construct, unfortunately in general they are incorrect. The STM codes hang on the GPU because the STM framework internally uses such error prone synchronization.

In Sect. 2 we discuss that there are fundamental architectural reasons for those dead- and livelocks on current GPUs. We also show how to construct a non-hanging GPU-wide critical section.

This brings us to our second motivation: In all the successful, non-hanging runs (and only those), the FGL-code has less overhead and clearly outperforms

[1] Recall that the upper code in Fig. 1 can deadlock on a GPU. Generally speaking, the `while`-condition splits the threads (of a warp/wavefront) into two sets. One set has the thread that has acquired the lock, the other set holds all the other threads. The SIMT instruction scheduler then runs the sets in turn, one after the other, and up to a convergence point where the sets are combined again. The problem is that there is no yield and that the instruction scheduler does not switch between sets. If the scheduler chooses to issue instructions to the set of the spinning threads first, the set with the winning thread never receives a single instruction, it does not make any progress, and thus it never releases the lock.

[2] For Table 1 we compiled with `-O0`. If we use `-O3` *all* the runs hang. In our benchmark environment the compiler seems to undo the manual anti-SIMT-deadlock transformation shown in Fig. 1.

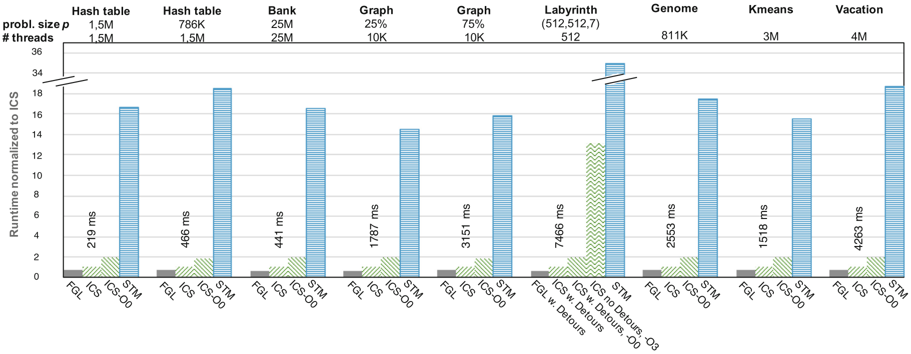


Fig. 2. Runtimes, normalized to ICS codes (`-O3`).

the optimistic concurrency control on the GPU, see the first and last bar of each bundle in Fig. 2. With more collisions (smaller hash table and $p = 75\%$ for the Graph benchmark) the STM versions get even slower.

Sections 3 and 4 then extend our GPU-wide critical section to so-called **I**nspected **C**ritical **S**ections (ICS) that get close to the FGL-performance (without the hanging) and that also make the reasoning for the programmer as straightforward as for the FGL-codes. See the ICS-columns in Fig. 2 that we have used to normalize the other runtimes.[3]

2 Non-hanging GPU-wide Critical Sections

To explain where the dead- and livelocks come from, we sketch a GPU's execution model and its vendor-provided schedulers. When a GPU programmer creates a grid of threads s/he also organizes them into b **groups** of t threads each (NVIDIA: block, AMD: workgroup). We call t the **group-size**. The GPU breaks up a group into **warps** of typically 32 threads (AMD: wavefront). To keep the discussion simple, assume that the grid size is exactly $b{\cdot}t$ and that

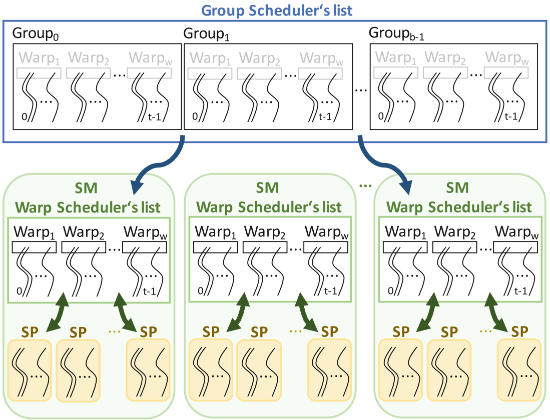


Fig. 3. GPU architecture and schedulers.

[3] Since we can only show `-O0` numbers for STM and FGL due to the compiler issue, we show those numbers for ICS as well, even though the compiler issue did not prevent `-O3` for our approach.

t is a multiple of 32. Figure 3 shows a grid of threads in the work list of the GPU's *group scheduler*. It picks a group and assigns it to one of the GPU's **Streaming Multiprocessors (SM)**. Groups are never preempted before they have completed execution. Since in general, there are unassigned groups (even if some SMs can process more than a group), there cannot be a GPU-wide barrier for all threads of the grid. Assume that all threads of all the SM-assigned groups had reached such a barrier. As they all still have work to do beyond that barrier, these groups are unfinished. This prevents pending groups from being assigned to an SM; the SM-assigned threads wait forever.

When a group is assigned to an SM, its *warp scheduler* dispatches the threads of the group's warps to the SM's many **Stream Processors (SP)**. Multiple SPs then execute all the threads of the warp in SIMT-mode. To achieve this, the *instruction scheduler* spawns the same instruction to all threads at the same time in a lock-step fashion.

GPU vendors do not disclose the strategies that their schedulers use and could change them at will between hardware releases and compiler versions. As it is unspecified for which of the branches of a condition the instruction dispatcher spawns the instructions first, the upper spin lock of Fig. 1 was incorrect and prone to SIMT-deadlocks. As there are no fairness guarantees for the warp scheduler on when to replace unfinished warps with pending ones, the code in the lower part of Fig. 1 is also incorrect. It can cause a livelock because in contrast to the group scheduler, the warp scheduler *can* take back unfinished warps from the SPs at any time. Assume that more warps are dispatched to an SM than it has SPs. Some of these warps can be part of the same group, or they can belong to other groups assigned to the same SM. Immediately after a winning thread has entered the critical section (line 4), the warp scheduler can choose to replace the winner's warp by another warp. Afterwards all the SPs run warps whose threads all wait for the critical section to become available again. From the viewpoint of the warp scheduler all the scheduled warps perform useful work because they all process the `while` loop. So there is no need to (ever) schedule back in the winner; the other warps may hence spin forever. The code hangs in a livelock.[4]

Figure 4 shows our new non-hanging GPU-wide critical section. It hoists the lock-acquisition and the lock-release out of the individual threads of a group. Instead of having *each* of the t threads of a group compete for the global lock, the lock is acquired only once per group.

```
if (GID == 0) // 1 thread has Group-level ID 0
    global.lock(); // exclude other groups
<thread-group-barrier> //CUDA: __syncthreads()

// Run threads in the group one after the other
for (i=0 .. GROUP_SIZE-1) {
    if (i == GID)
        // critical section code here
    <thread-group-barrier>
}

if (GID == 0) // 1 thread of the set (= group)
    global.unlock(); // allow other groups
```

Fig. 4. Non-hanging GPU-wide critical section.

[4] To circumvent this problem, many codes use the incorrect spin lock with a grid size that stays below (warp size · # of SPs). With such an underutilization of the GPU the warp scheduler does not have to re-schedule because there is no more than one warp per SP.

All the threads of an SP-assigned warp execute the code in Fig. 4 concurrently in a lock-step fashion. In each of the SM-assigned groups the one thread with the group-level ID 0 within that group acquires the GPU-wide lock `global` (of the critical section) in line 2. (Release is in line 11.)

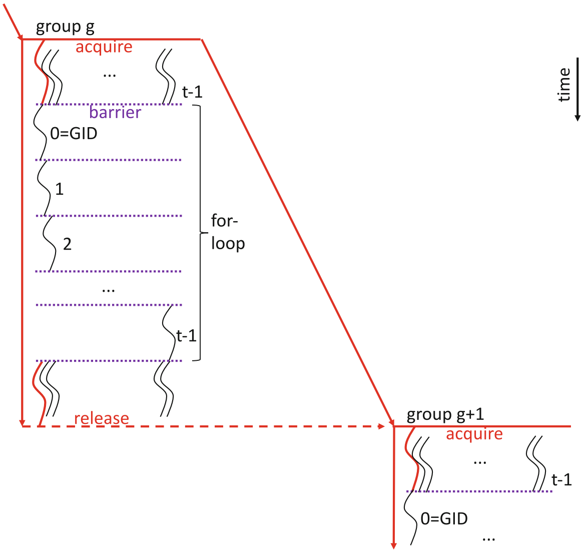


Fig. 5. Schematic execution of Fig. 4. (Color figure online)

In Fig. 5 time flows from top to bottom. The thread with GID = 0 (red) of a group acquires and releases the lock. Let us ignore the global locking (and unlocking) for now – it only matters that the locking is followed by a `<thread-group-barrier>` in line 3 of the code (or the dotted line in Fig. 5, resp.). This is the SM-wide barrier that interacts with the warp scheduler. The `GID`-condition in line 1 splits a warp into two sets. The set with the single thread acquires the lock and runs into the barrier. The threads in the other set also run into the barrier. Which set of threads receive their SIMT-instructions first is irrelevant as the convergence point for all threads is before the barrier. Note that the warp scheduler replaces such warps (whose threads all are inactive) with other (pending) warps that still have active threads. As more and more warps find all their threads inactive in the barrier, eventually all warps of a group will be assigned to SPs and will reach the barrier. Thus, regardless of the scheduler's internal strategy, the hanging-problem is gone.

Note that it is irrelevant whether and when other groups that are scheduled to other SMs reach their barriers. It only matters that all the threads of the warp with the winning thread eventually finish their work. Then other warps finish because of the same reasoning. Then eventually an SM finishes and another (pending) group can be assigned to this SM.

There are two issues left to explain. First, the global lock (lines 2 and 11) can be implemented with any CPU-style lock, e.g., with the upper spin lock from Fig. 1. As the group scheduler never takes back the group that acquired the lock before completion, this group is still active when it eventually releases the lock.

The second question is, how the group's threads actually perform the body of the critical section sequentially. In line 3 there is an SM-wide barrier. After that barrier *all* threads of the group are active again (not just the thread that acquired the lock). However, the critical work needs to be performed sequentially by one thread at a time. This is what the code in lines 4–9 achieves. In SIMT-mode *all* threads execute the `for` loop. But in each iteration only one of them finds the loop counter to be equal to its own GID. This thread executes the body of the critical section, the others pause, before all threads meet again in the SM-wide barrier in line 8 which is the point of convergence for all threads. This barrier is needed to prevent other threads from prematurely starting with their critical section work. After the barrier the next thread executes the body of the critical section, as shown in the `for`-loop area of Fig. 5. When all threads are done, the thread with GID = 0 releases the lock to let the next group proceed.

We will later reuse this idea for each of the architectural levels of a GPU (and for more levels) to efficiently implement inspected critical sections.

3 Inspected Critical Sections

Now that we have non-hanging GPU-wide critical sections we can let the threads execute the code sequentially. However, this is obviously overly restrictive as threads that work on disjoint data should be able to do their work in parallel.

Thus, we introduce `ics(list of items)` { `block` }, an **I**nspected **C**ritical **S**ection wherein the programmer declares all the items (data elements, memory addresses, fields, ...) up front that each of the data-parallel threads needs isolated access to (in the `block`). As for the FGL-codes, the programmer must know what is thread-local data and what is shared between threads and thus needs to be mentioned in the `list of items`. For the hash table with chaining, the list holds the number of the bucket that a `put` uses. This number represents the shared data that potentially causes a conflicting access. It is more straightforward than the memory address of the bucket.

The semantics of the inspected critical sections is inspired by the classic inspector-executor paradigm [3]. Upon entry to the critical section, it is checked whether the lists of items intersect between threads. If so, the code will run sequentially, otherwise the threads run the `body` in parallel, as they would do for an atomic region. For the inspection, the threads atomically register their items in a bitmap. If a bit is already set when they try to set it, a conflict is detected.

There are three differences to the traditional inspector-executor. First, we start from parallel code and the inspector downgrades to a sequential execution, whereas originally the inspector is used to parallelize a loop if possible. Second, we rely on the programmer to identify where conflicts may be. As there is no automatic detection of all the accessed memory addresses, the programmer can leave out irrelevant addresses. There is also no longer the problem that computed memory addresses may fool the automatic detection (think of `a[foo()]` where `foo` is impure). In general, there is not even the need to consider all memory addresses. Instead, smaller data structures suffice. For the hash table a bitmap with one bit per bucket is large enough. A traditional inspector works with the full memory addresses of the buckets.

Therefore, in addition to the `ics`-statement, the developer *has to* specify an `__upperBound()` of the size of the shared data structure and hence the size of the bitmap. The developer *can* also overwrite the mapping function __idx() with an application-specific one that maps a potentially conflicting item (data element, memorylocation, . . .) to an index of the bitmap.[5] For the hash table, the number of available buckets is the upper bound; as the bucket number used by the put is a good index, there is no need to use the memory address.

The third difference to the traditional inspector-executor is that we do better than all-or-nothing. Instead of sequentializing as soon as there is a conflict, Sect. 4 introduces a gradual retrenchment of parallelism that keeps up the degree of parallelism for those threads that do not interfere.

Some algorithms (like the Labyrinth benchmark) can find a detour if an initially available resource can no longer be used because another thread has taken it. Instead of sequentializing to deal with this conflict, there may be an application-specific way around. For such situations the developer *can* provide a method __alternative(item) that exploits knowledge about which item causes the conflict, finds a detour, and retries the conflict check with a new list of items.

The pseudo code in Fig. 6 shows how the initial inspection checks whether there are conflicts between the concurrent threads. (For a better understanding, ignore the level-indices for now and assume single values instead – Sect. 4 will fill in the details.) All threads run the code in parallel. For each potentially critical item the threads use a CAS operation to set the corresponding bit (line 9) in the bitmap. The index of the bit is determined by means of __idx(). If there is a conflict and if there is a valid function pointer to an application-specific __alternative() callback function in line 11, then the application gets the chance to modify the local results and to retry with a different set of potentially conflicting items (lines 11–17). There is an upper bound on the number of retries. If the application cannot find a patch/an alternative that avoids the conflicting item, the global conflict flag is set. We optimize this if there is only a single item to inspect.

```
 1  void check4confl(level, items[]) {
 2      // precondition: hasConflict = false;
 3      resetBits(bitmap);
 4      barrier[level]();
 5      iter = 0;
 6      retries = 0;
 7      do {
 8          item = items[iter];
 9          if (atomicCAS(&bitmap[__idx(item)],
10                              0, 1) == 1) {
11              if(__alternative != NULL) { // Detour?
12                  resetMyBits(bitmap, items);
13                  items = __alternative(item);
14                  iter = 0;
15                  retries++;
16                  continue;
17              }
18              hasConflict = true;
19          }
20          iter++;
21      } while (items.size()>iter && retries!=3);
22  }
```

Fig. 6. Check for conflicts, with "Detour" option.

[5] Note that if needed, the developer can trade time for space: Ideally __idx() is an injective projection of an item to [0..__upperBound()-1]. With a smaller co-domain of __idx(), the bitmap can be smaller, but the conflict detection may announce false positives that then cause sequential execution and hence longer runtimes.

4 Gradual Retrenchment of Parallelism

Sequentializing all threads once a conflict is found is too slow to be practical. To make inspected critical sections efficient, we use a divide-and-conquer approach that instead of instantly switching to a fully sequential execution, splits the threads into smaller sets [7]. We process these sets in order, set after set. Within such a set, the threads *could* still modify the data without a conflict. Hence, before the threads of a set perform their work sequentially, they again check for conflicts, but this time only among themselves. If there is no conflict, this set of threads can run in parallel. Otherwise we apply this idea recursively. Since all the sets of the same level are always processed one after the other, the threads that caused the initial conflict can never run at the same time.

The GPU architecture from Fig. 3 guides the hierarchical splitting into sets of threads. If the conflict is on the first level of the recursion, between the threads that run on all the SMs, then we split them into their groups. The SMs those groups are assigned to process them sequentially. One level down it may be possible that all the threads in a group can run without a conflict.

If there is still a conflict among all the threads on that SM, then the SM needs to process its warps (i.e., the next level of sets) sequentially. One level down, potentially all the threads in a warp can run without a conflict.

The next level down are pairs of threads. To keep it simple, we have left this out in Fig. 4. Pairs are executed in order, but within a pair the two threads can run concurrently unless they interfere.

The base level is a full sequentialization of the threads as shown in the figure.

```
 1  static bool hasConflict;
 2  void retrench(level, items[]) {
 3      lock[level].acquire(); //includes ?ID==0 test
 4      hasConflict = false;
 5      barrier[level]();
 6      if (level < 4) {
 7          check4confl(level, items[]);
 8          barrier[level]();
 9      }
10      if (!hasConflict) {
11          //critical section code here; body of ics
12      } else {
13          retrench(level+1, items[]);
14      }
15      barrier[level]();
16      lock[level].release(); //include ?ID==0 test
17  }
```

Fig. 7. Hierarchical retrenchment of parallelism.

The recursive pseudo code is shown in Fig. 7. This code is a generalization of the code shown in Fig. 4. As discussed above, there cannot be GPU-wide barriers. Thus the recursion does not start from the full grid but from the SM level. To implement the user's `ics`-statement, all active threads execute `retrench` (which calls the `body` of the `ics`). As in Sect. 2 only one of them acquires the lock of the current level (line 3). The `acquire` method comprises the ?ID==0 test known from before (? stands for the level, e.g., G for group level). The other sets of that hierarchy level wait; the lock acquisition serializes them. All threads of the winning set leave the barrier in line 5. There are different barrier implementations for each level: a (home-grown spin-based) barrier across the SMs, the hardware-supported SM-wide `<thread-group-barrier>`, a (home-grown) warp-wide barrier, and conceptually even a barrier for a pair of threads in a warp. The recursion `level` is used to pick the appropriate type of barrier. In line 7 all those threads inspect the items that they intend to work with for conflicts. If there is none, they can execute the critical section code concurrently (line 11). Otherwise, we recursively split the set of threads into smaller sets. Notice that on the lowest level 4, there is no checking for conflicts as there is only one active thread. Thus the recursion always ends in line 11 as soon as it reaches the level of a single thread. On the way out of the recursion, one thread releases the level's lock (line 16). This releases another set of threads that is waiting for the lock.

Figure 8 shows what happens at the group level. Initially the thread with GID = 0 (red, on the left) acquires the lock. Then all the group's threads check whether there are conflicts (c4c for `check4confl`). If there is none, the threads concurrently execute the critical section code (bottom left of the figure) and the thread with GID = 0 releases the lock so that another SM can proceed with its group (not shown). Otherwise the recursive invocation of `retrench` splits the t threads into warps. Each of the warps has a thread with WID = 0 that tries to acquire the (orange) lock (line 3 of the code; conflict side of Fig. 8). The warps are processed in sequence, one after the other. For each of the warps there is again the concurrent checking for conflicts. A warp can either run in parallel or – if there is a conflict among the warp's threads – the recursion proceeds to pair level. This decision can vary from warp to warp.

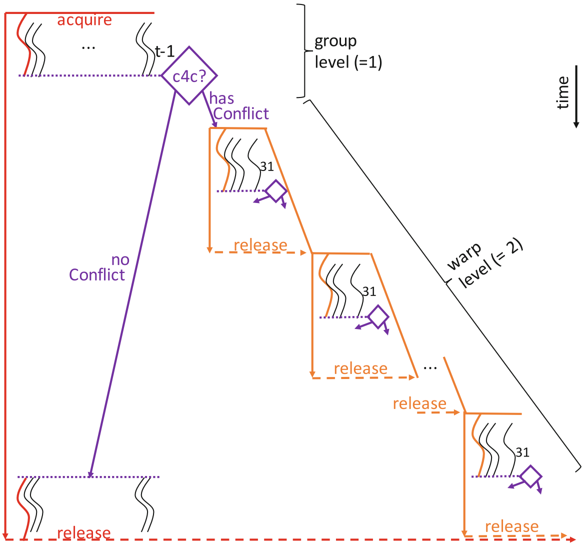


Fig. 8. From group to warp level. (Color figure online)

The above recursive pseudo code is simplified to get the idea across. The actual implementation not only unrolls the recursion, but it also cuts off the recursive descent as soon as it reaches warp level (=2). Here the `for`-loop known from the non-hanging GPU lock, see Fig. 4, suffices due to the SIMT-execution. Moreover, for pairs of threads that execute in a lock-step fashion anyway, `check4confl` can be optimized as no longer bitmaps with atomic operations are needed. Due to space restrictions, we cannot get into details, but eventually the lower two levels of the retrenchment are fused into a single efficient `for`-loop that also saves on the number of synchronization barriers.

5 Evaluation

Recall that the quantitative results are the motivation of this work, see Sect. 1: The STM versions (with mostly given atomic regions) and the FGL versions (written by us) of the benchmark codes (see Appendix) frequently hang in dead- or livelocks, see Table 1. The ICS versions never hang, they use straightforward `__idx` functions to indicate where the threads may be in conflict at runtime,[6] and they are much faster than the STM codes and often get close to the FGL-versions (provided the latter do not hang), see Fig. 2.

Table 2. Level on which ICS executes the critical code.

	Hash table		Bank	Graph		Labyrinth	Genome	Kmeans	Vacation
Problem size	1,572,864	786,432	25,165,824	25%	75%	(512,512,7)	Configuration see Appendix		
# threads	1,572,864		25,165,824	10,280		512	811,008	3,014,656	4,194,304
8·SM	215,040	23,040	5,360,640	0	0	0	122,880	872,448	906,240
group	1,282,560	416,128	11,438,336	3456	128	0	318,080	1,482,240	2,129,536
warp	72,608	799,648	8,233,856	6176	3392	160	351,904	639,936	1,085,856
pair	2642	333,980	132,844	648	6698	312	18,102	19,988	72,634
single	14	68	148	0	62	40	42	44	38

Let us now look into three more aspects of these general results. First, as the key idea of our approach is to retrench parallelism gradually so that threads that work on non-conflicting parts of the shared data can run concurrently instead of being sequentialized, Table 2 shows on which level of the retrenchment cascade the threads actually execute the critical code (average over 100 runs).

For the large hash table 215,040 (14%) of the threads execute the critical section code in parallel on the first level of the retrenchment cascade. Since the recursive decent stops on the first level only a few barriers cause overhead. The majority of the threads (82%) can retain group-level parallelism, where 128 threads run in parallel. Only 14 threads need to run in isolation. For the smaller hash table with more collisions the numbers shift towards the lower end of the scale; still retaining a high degree of parallelism. Bank is similar. The other four benchmarks have more collisions, but they also achieve a bell-shaped distribution of levels.[7]

[6] For Hash we use the number of the bucket, for Bank it is the account numbers. Labyrinth uses the coordinates of the points in the mesh as `__idx`. Genome uses a common subsequence (string) to identify a hash bucket that holds common DNA-segments. Kmeans uses the Id of a cluster. Vacation uses Ids of hotel rooms, flighs, and cars. We never use memory addresses as items.

[7] If we force ICS to always assume a conflict and to go down to the *single* thread level, runtimes are much slower than the STM version (Hash table: 14x and 11x, Bank: 13x, Graph: 6x and 2x, Labyrinth: 2x, Genome: 12x, Kmeans: 10x, Vacation: 15x).

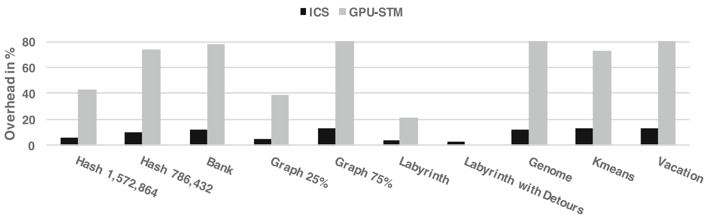
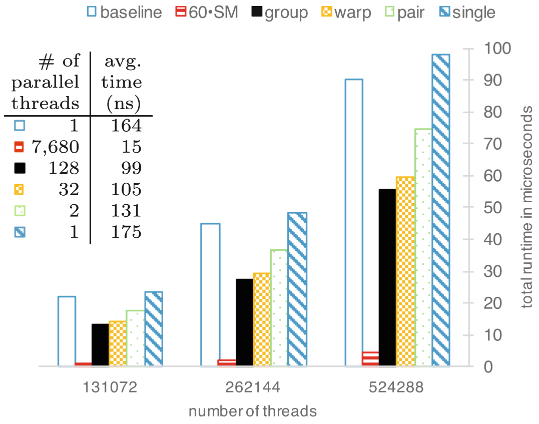


Fig. 9. Fraction of overhead of ICS and STM.

Second, let us study the overhead. STM research separates the runtime spent in the atomic region from the time spent for logging, commit processing, and rollback. We mimic this and also isolate the time spent in the retrenchment cascade and the barriers from the critical section bodies. Figure 9 shows that the GPU-STMs overhead is 24%–84% (similar results can be found in [20]), while the ICS codes only have an overhead of 2.7%–13%.

To understand where this small overhead comes from, Fig. 10 compares an ICS with both an empty `block` and with no items to check for conflicts, to the non-hanging GPU-wide critical section from Fig. 4 with its full sequentialization. The first bar of each group depicts this base line (total runtime of all threads; the overhead per thread is given in the table on the side of Fig. 10).[8]



	# of parallel threads	avg. time (ns)
□	1	164
▤	7,680	15
■	128	99
▦	32	105
⬚	2	131
▧	1	175

Fig. 10. Overhead of the `retrench` cascade.

The bars that follow show the runtimes of the recursive retrenchment cascade. For a certain retrenchment level ($60 \cdot SM, group, \ldots, single$), we let `check4confl` on the surrounding levels (if any) always signal a conflict. On the measured level there is no conflict so that the parallel threads perform their (empty) critical section in parallel. Hence, on the $60 \cdot SM$ level, on each of the 60 SMs of our GPU all the 128 threads of the assigned group run in parallel (i.e., there is a total number of $60 \cdot 128 = 7,680$ parallel threads). On the *group* level, the SMs process their assigned groups sequentially, while within a group all 128 threads run in parallel. The last bar shows that the worst case overhead of the level-wise retrenchment adds about 7% to the base line.

There are additional aspects to note. (a) The overhead per thread is better with fewer retrenchment, see side table, since higher levels have fewer barriers along the cascade. Hence, the fewer dynamic dependences an application has, the smaller is the runtime fee that it pays. (b) The warp-level optimizations and the pairing of threads pay off. (c) Doubling the number of threads approximately

[8] All measurements with `-O3`; the compiler did not remove the empty `block`.

doubles the total runtime and leaves the overhead per thread fixed. (d) If the `body` had not been empty, the inner four bars of each bundle would shrink in relation to the fully sequential execution of both the first and the last bar.

Third, if there is a conflict in the Labyrinth benchmark, the STM has to completely undo the transaction. The results in Fig. 2 show that due to the enhanced expressiveness, the FGL-code and the ICS-code with the optional `__alternative()` can do much better. It also lowers the overhead in Fig. 9.

6 Limits

Inspected critical sections trade STM-comfort for runtime performance. When using the inspected critical section, a programmer may miss items that can be in conflict between threads. Failure to declare such items is likely to cause races. Because of the lock-step execution it may be a bit easier to avoid such bugs than in general MIMD codes. For collision detection, ICS expects that all accessed memory addresses are known a-priori. If there are unforeseeable addresses, i.e., conflicting accesses to computed memory addresses it is much more difficult to keep the degree of parallelism up.

The programmer can trick the inspected critical sections into a buggy behavior with wrong auxiliary functions. Examples are a too narrow `__upperBound()` that does not match the co-domain of the mapping function `__idx()`, or an `__idx()` that is stateful and yields different answers when invoked for a single item (data element, memoryaddress, ...) twice and/or by different threads.

The pseudo code that this paper uses to explain how an inspected critical section is implemented, assumes that (like in all the benchmark codes) all data-parallel threads of the GPU kernel do enter the critical section, i.e., there cannot be a surrounding condition that lets some threads avoid the critical section. (The reasons are: (a) current GPUs require that *all* threads must reach a `thread-group-barrier`, and (b) for correctness our pseudo code requires that the thread with ?ID=0 has entered the critical section.) So far, we circumvent this problem by (manually) hoisting the critical section out of the condition. In general, there is a performance penalty for this as the critical sections get larger.

7 Related Work

Several authors study how to correctly and efficiently implement synchronization, locking, and barriers on the GPU and on its architectural levels. A general difference to our work is that most of the related work comes from MIMD-parallelism and deals with threads that perform individual tasks. Our base line is different because we assume that all threads follow the same instructions in data-parallel code, but there are some code fragments that need synchronization. So whenever a locking is needed, conceptually all threads are involved.

ElTantawy and Aamodt [9] work on the SIMT-induced deadlocks and build their solution into a compiler transformation. Another published workaround moves the convergence point to a statement that all threads can reach – no matter if they have the lock [15]. At the lowest level of our recursive approach, we use similar ideas, but we also guarantee progress.

Xiao et al. [17,18] also work on inter-group barriers for GPUs. Whereas our threads also proceed after the ICS in parallel, there is also the `block` of code that conceptually they execute in isolation – running in parallel if there are no data dependences. Another difference is that on every level, our recursive retrenchment of parallelism uses smaller barriers that wait for fewer threads. The fewer and the more local the threads are that wait in a barrier, the more efficient the barrier code gets. On some levels of the GPU architecture there is even hardware support for barriers. Their group-level locking cannot use a similar optimization. Xu et al. [19] build livelock-free lock stealing and lock virtualization for GPUs. Their techniques only work on warp level, whereas our mechanisms not only work across all levels of the GPU architecture, but we also present optimizations on sub-warp level, e.g., for pairs of threads. Another difference is that their lock stealing makes it necessary that the developer provides undo-methods that reinstate functional correctness in case of a stolen lock. We do not need to supply such code.

With respect to low-level locks and barriers there is orthogonal work that we may be able to incorporate and benefit from. Whenever our system-level implementation needed a barrier or lock, we used a basic CPU-style spin lock (except where discussed in detail in the paper). Operating systems research has targeted the efficiency of locking techniques. Some authors improve the time delay, the memory traffic, and storages costs compared to locks based on atomics [21]. Others optimize for situations in which many locks are acquired and released often [13]. SmartLocks [8] is a library for spin lock implementations. Its goal is that the scheduler always picks from the spinning threads the one that probably contributes most to a certain goal, e.g., the overall runtime, the energy consumption, etc. It is orthogonal research to port such ideas to the GPU and to use the best types in our system-level implementation, especially as some ideas require hardware support that is not (or not yet?) available on GPUs.

There are several Transactional Memory implementations in software (STM) for GPUs [6,12,16,20]. Their common principle is to log all read and write operations that happen in a critical section. Multiple threads execute the critical section concurrently. If they detect a conflicting access in the logs at commit time, then they undo the work. All of this causes storage costs and memory traffic. In our approach we also execute critical sections concurrently, but only if we can check beforehand that there will be no conflicts. We assume that the developer knows the application well enough to be able to indicate those data elements/memory locations that at runtime threads may access in a conflicting way. This is less costly because there is no need to rollback. Moreover, instead of logging all memory accesses, we use application-specific knowledge and only check those memory accesses that the developer knows to be potentially critical. This lowers the checking overhead even further. STMs are general-purpose. They hence need to be conservative and check and log every single memory access to achieve correctness.

Systems that rely on a static code analysis to find spots where concurrent threads can have conflicting access to data usually face a similar type of drawback. Due to their conservative approach, these systems, like race detection tools, in general produce many false positives. If these tools cannot prove the absence of a dependence, then they must assume that there is one. They do not benefit from application-specific knowledge. In contrast, we let the programmer specify the potentially critical data elements – and in a converse reasoning – it is known that other data does not cause any correctness problems. Synchronize via Scheduling (SvS) by Best et al. [4] is such a static analysis that checks whether certain tasks can run in parallel because they access disjoint variables. Because of the many false positives, SvS instruments the tasks with runtime checks that compare the working sets of the tasks. In a way, this is similar to our `check4confl`. However, we only have to consider a few programmer-indicated data elements while SvS – due to its general-purpose approach – has to process the whole state of a thread, if not the reachable graph of objects on the heap. SvS also does not optimize for GPUs whereas we carefully map the checking to the GPU architecture so that we retain as much parallelism as possible, even for the checking itself.

CUDA 9 is announced to offer so-called Cooperative Groups [1] that can bundle threads for collective operations. On current NVIDIA GPUs these bundles stick to the GPU hardware hierarchy and are unlikely to impede the results of this paper. On the announced Volta architecture [2] there will be a thread scheduling that is independent of the GPU hardware hierarchy. Although that will potentially make some dead- and livelocks go away, the programmer still has to make sure by hand that all threads are active that need to synchronize. We expect this to be as complicated as the mechanisms presented here. These issues and performance comparisons are future work.

8 Conclusion

On current GPUs, thread synchronization often suffers from dead- and livelocks (because of the SIMT execution and the schedulers). This makes porting of parallel applications to GPUs error-prone, especially when efficient fine-grain synchronization is needed. *Inspected Critical Sections* that make use of application-specific knowledge on which data items may cause dynamic data-dependences among data-parallel threads, outperform optimistic STM approaches on GPUs and get close to (unreliable) implementations with fine-grain locking. The key to the efficiency of ICS is a divide-and-conquer approach that exploits the architectural levels of GPUs and that employs a dead- and livelock free GPU-wide barrier with guaranteed progress.

Appendix

Benchmark Infrastructure

For all measurements we use a 1,5 GHz Desktop NVIDIA TITAN Xp GPU with 12 GBytes of global memory and 3.840 cores in 60 SMs (with 64 SPs each) that runs CUDA (Version 8.0) code.

The group-size in all measurements is 128 threads. The reason is that on our GPU the kernels can use up to 32.000 registers per group, i.e., 250 registers per thread. Both the retrenchment cascade and the STM framework need 70 of those registers. This leaves 180 registers for the local variables of the applications. Since the benchmarks need that many, we could not use larger group-sizes. While smaller group-sizes are possible, we only present measurements for a group-size of 128 threads because our experiments did not show qualitatively different results for smaller group-sizes.

We repeated all measurements 100 times; all given numbers are averages. For the code versions with fine-grained locks and the STM-based implementations we only measured those runs that did not face a dead- or livelock.

Benchmark Set

We use seven benchmarks, some of which are taken from the STAMP benchmark suite [14] with given atomic regions. We always use the largest possible shared data structure and/or the maximal number of threads that fit onto our GPU.

Hash Table. We use $1.5M$ threads and a hash table with the same number of buckets, each of which holds the linked lists of colliding entries. The threads randomly put a single entry into the shared hash table. ICS uses the bucket number as item to check for conflicts. The bucket operation is the atomic region in the STM code. The fine-grained lock code (FGL) uses one lock per bucket. To study the effect of the number of collisions, we also use half the buckets.

Bank. There are $24M$ accounts. $24M$ parallel threads withdraw an amount of money from one randomly picked account and deposit it to another. The two accounts are the items for conflict checking. There is a conflict if two threads use an account in common. STM: the transfer happens in the atomic region. FGL: there is one lock per account.

Graph. The $G(n, p)$-instance of the Erdős-Rényi Graph Model (ERGM) [10] starts from an edgeless graph with $n = 10K$ nodes. A thread per node adds an undirected edge to any other node (= ICS item for conflict checking) with probability p. To illustrate the effect of the number of collisions we study the two probabilities $p = 25\%$ and $p = 75\%$. STM: the atomic region is the insertion of an edge. FGL: the code locks the adjacency lists of both the nodes that the new edge connects.

Labyrinth. The largest 3D-mesh from the STAMP input files that fits into our memory has size $(512, 512, 7)$. Thus 512 threads plan non-intersecting routes in parallel. All nodes of the route are the items for conflict checking. STM: a full routing step is the atomic region. FGL: there is a lock per mesh point. FGL and ICS: if a route hits a spot that is already part of another route, the thread tries (three times) to find a detour around it. This avoids recalculating the full route.

Genome. $8M$ threads try to reconstruct a genome from DNA segments that reside in a shared pool, that is a hash table. There may not be duplicates and only one thread may check whether and where a segment from the pool matches the given genome. ICS checks conflicts on the bucket number. We consider a genome size of $65,536$, DNA segments have a size of 192, and there are $1,677,726$ such segments. STM and FGL: see Hash table.

Kmeans. $3M$ threads partition the same number of data items from a 32-dimensional space into $1,536$ subsets (clusters). Until a fix point is reached, all threads check the distance to the centers of all of the clusters and migrate a data item to the closest cluster (= item for conflict checking). STM: the migration is the atomic region. FGL: there is one lock per cluster; the code locks the two clusters that are affected by a migration.

Vacation. The travel reservation system uses hash tables to store customers and their reservations for a hotel, a flight, and a rental car, i.e., on three potentially conflicting items. $4M$ parallel threads perform $4M$ (random) reservations, cancellations, and updates for full trips. There may be conflicts. There are configuration parameters for the likelihood of such conflicts and the mix of operations (for the STAMP expert: we use $r = 629148$, $u = 93$, $q = 90$). STM: one operation on all three components of a trip is in the atomic region. FGL: there is a lock per hotel, flight, and car.

References

1. CUDA 9 Features Revealed: Volta, Cooperative Groups and More (2017). https://devblogs.nvidia.com/parallelforall/cuda-9-features-revealed/. Accessed 03 July 2017
2. Inside Volta: The World's Most Advanced Data Center GPU (2017). https://devblogs.nvidia.com/parallelforall/cuda-9-features-revealed/. Accessed 03 July 2017
3. Baxter, D., Mirchandaney, R., Saltz, J.H.: Run-time parallelization and scheduling of loops. In: (SPAA 1989): Symposium on Parallel Algorithms and Architecture, Santa Fe, NM, pp. 603–612, June 1989
4. Best, M.J., Mottishaw, S., Mustard, C., Roth, M., Fedorova, A., Brownsword, A.: Synchronization via scheduling: techniques for efficiently managing shared state. In: (PLDI 2011): International Conference on Programming Language Design and Implementation, San Jose, CA, pp. 640–652, June 2011
5. Cascaval, C., et al.: Software transactional memory: why is it only a research toy? Queue **6**(5), 40:46–40:58 (2008)
6. Cederman, D., Tsigas, P., Chaudhry, M.T.: Towards a software transactional memory for graphics processors. In: (EG PGV 2010): Eurographics Conference on Parallel Graphics and Visualization, Norrköping, Sweden, pp. 121–129, May 2010

7. Dang, F.H., Rauchwerger, L.: Speculative parallelization of partially parallel loops. In: (LCR 2000): International Workshop Languages, Compilers, and Run-Time Systems for Scalable Computers, Rochester, NY, pp. 285–299, May 2000
8. Eastep, J., Wingate, D., Santambrogio, M.D., Agarwal, A.: Smartlocks: lock acquisition scheduling for self-aware synchronization. In: (ICAC 2010): International Conference on Autonomic Computing, Washington, DC, pp. 215–224, June 2010
9. ElTantawy, A., Aamodt, T.M.: MIMD synchronization on SIMT architectures. In: (MICRO 2016): International Symposium on Microarchitecture, Taipei, Taiwan, pp. 1–14, October 2016
10. Erdős, P., Rényi, A.: On random graphs I. Publ. Math. (Debrecen) **6**, 290–297 (1959)
11. Habermaier, A., Knapp, A.: On the correctness of the SIMT execution model of GPUs. In: (ESOP 2012): European Symposium on Programming, Tallinn, Estonia, pp. 316–335, March 2012
12. Holey, A., Zhai, A.: Lightweight software transactions on GPUs. In: (ICPP 2014): International Conference on Parallel Processing, Minneapolis, MN, pp. 461–470, September 2014
13. Li, A., van den Braak, G.J., Corporaal, H., Kumar, A.: Fine-grained synchronizations and dataflow programming on GPUs. In: (ICS 2015): International Conference on Supercomputing, Newport Beach, CA, pp. 109–118, June 2015
14. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: (IISWC 2008): International Symposium on Workload Characterization, Seattle, WA, pp. 35–46, September 2008
15. Ramamurthy, A.: Towards scalar synchronization in SIMT architectures. Master's thesis, University of British Columbia, September 2011
16. Shen, Q., Sharp, C., Blewitt, W., Ushaw, G., Morgan, G.: PR-STM: priority rule based software transactions for the GPU. In: (Euro-Par 2015): International Conference on Parallel and Distributed Systems, Vienna, Austria, pp. 361–372, August 2015
17. Xiao, S., Aji, A.M., Feng, W.C.: On the robust mapping of dynamic programming onto a graphics processing unit. In: (ICPADS 2009): International Conference on Parallel and Distributed Systems, Shenzhen, China, pp. 26–33, December 2009
18. Xiao, S., Feng, W.: Inter-Block GPU communication via fast barrier synchronization. In: (IPDPS 2010): International Symposium on Parallel and Distributed Processing, Atlanta, GA, pp. 1–12, April 2010
19. Xu, Y., Gao, L., Wang, R., Luan, Z., Wu, W., Qian, D.: Lock-based Synchronization for GPU architectures. In: (CF 2016): International Conference on Computing Frontiers, Como, Italy, pp. 205–213, May 2016
20. Xu, Y., Wang, R., Goswami, N., Li, T., Gao, L., Qian, D.: Software transactional memory for GPU architectures. In: (CGO 2014): International Symposium on Code Generation and Optimization, Orlando, FL, pp. 1:1–1:10, February 2014
21. Yilmazer, A., Kaeli, D.R.: HQL: a scalable synchronization mechanism for GPUs. In: (IPDPS 2013): International Symposium on Parallel and Distributed Processing, Cambridge, MA, pp. 475–486, May 2013