MIMD Synchronization on SIMT Architectures

Ahmed ElTantawy and Tor M. Aamodt
University of British Columbia
{ahmede,aamodt}@ece.ubc.ca

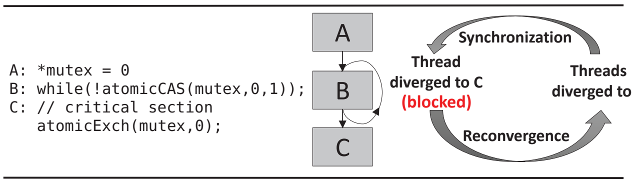
***Abstract*—In the single-instruction multiple-threads (SIMT) execution model, small groups of scalar threads operate in lockstep. Within each group, current SIMT hardware implementations serialize the execution of threads that follow different paths, and to ensure efficiency, revert to lockstep execution as soon as possible. These constraints must be considered when adapting algorithms that employ synchronization. A deadlock-free program on a multiple-instruction multiple-data (MIMD) architecture may deadlock on a SIMT machine. To avoid this, programmers need to restructure control flow with SIMT scheduling constraints in mind. This requires programmers to be familiar with the underlying SIMT hardware.**

In this paper, we propose a static analysis technique that detects SIMT deadlocks by inspecting the application control flow graph (CFG). We further propose a CFG transformation that avoids SIMT deadlocks when synchronization is local to a function. Both the analysis and the transformation algorithms are implemented as LLVM compiler passes. Finally, we propose an adaptive hardware reconvergence mechanism that supports MIMD synchronization without changing the application CFG, but which can leverage our compiler analysis to gain efficiency. The static detection has a false detection rate of only 4%–5%. The automated transformation has an average performance overhead of 8.2%–10.9% compared to manual transformation. Our hardware approach performs on par with the compiler transformation, however, it avoids synchronization scope limitations, static instruction and register overheads, and debuggability challenges that are present in the compiler only solution.

I. Introduction

The single-instruction multiple-thread (SIMT) programming model was introduced and popularized for graphics processor units (GPUs) along with the introduction of CUDA [1]. It has seen widespread interest and similar models have been adopted in CPU architectures with wide-vector support [2]. Arguably, a key reason for the success of this model is that it largely abstracts away the underlying SIMD hardware. In SIMT-like execution models, scalar threads are combined into groups that execute in lockstep on single-instruction multiple-data (SIMD) units. These groups are called warps by NVIDIA [1], wavefronts by AMD [3], and gangs by Intel [2]. The programming model divides the burden of identifying parallelism differently than traditional approaches of vector parallelism. The programmer, who is armed with application knowledge, identifies far-flung outer-loop parallelism and specifies the required behaviour of a single thread in the parallel region.

On current hardware the SIMT model is implemented via predication, or in the general case using stack-based masking of execution units [2], [4]–[7]. This mechanism enables threads within the same warp to diverge (i.e., follow different control flow paths). However, they achieve this by serializing the execution of different control-flow paths while restoring SIMD utilization by forcing divergent threads to reconverge as soon as possible (typically at an immediate postdominator point) [2], [5], [8]. This in turn creates implicit scheduling constraints for divergent threads within a warp. Therefore, when GPU kernel code is written in such a way that the programmer intends divergent threads to communicate, these scheduling constraints can lead to surprising (from a programmer perspective) deadlock and/or livelock conditions. Thus, a multi-threaded program that is guaranteed to terminate on a MIMD architecture may not terminate on machines with current SIMT implementations [9][1].



```
A: *mutex = 0
B: while(!atomicCAS(mutex,0,1));
C: // critical section
   atomicExch(mutex,0);
```

Fig. 1: SIMT-Induced Deadlock

Figure 1 shows a typical MIMD implementation of a spin lock guarding a critical section. On a SIMT machine, this code deadlocks. In particular, a thread that acquires the lock is indefinitely blocked at the loop exit waiting to reconverge with lagging threads from the same warp. However, lagging threads never exit the loop because they wait for the lock to be released by the leading thread. Similar scenarios occur with fine-grained synchronization. We refer to a case where the forward progress of a diverged thread is prevented due to the implicit SIMT scheduling constraints as *SIMT-induced deadlock* or briefly *SIMT deadlock*.

The possibility of SIMT-induced deadlocks is a challenge, given the increasing interest in using SIMT architectures for irregular applications [10]–[12]. Moreover, parallel algorithms developed for MIMD execution can serve as starting points for GPU kernel development provided SIMT deadlock can be avoided. For complex applications writing functionally correct code can be challenging as programmers need to reason about how synchronization interacts with the SIMT implementation. Further, the code is vulnerable to compiler optimizations that may modify the control flow graph (CFG) assumed

[1] We use the term "MIMD machine" to mean any architecture that guarantees loose fairness in thread scheduling so that threads not waiting on a programmer synchronization condition make forward progress.

978-1-5090-3508-3/16/$31.00 © 2016 IEEE

by programmers. SIMT deadlocks also present challenges to emerging OpenMP support for SIMT architectures [13]–[17] and to the transparent vectorization of multi-threaded code on SIMD CPU architectures [2], [18].

In this paper, we propose a static analysis technique that (1) detects potential SIMT-induced deadlocks and (2) identifies safe locations for delayed reconvergence that allow for inter-thread communication. The static analysis information is leveraged by two different proposed solutions for SIMT deadlocks (1) a compiler-based solution that alters the CFG of the application to adhere to the recommended safe reconvergence points and (2) a hardware-based solution that implements a SIMT reconvergence mechanism with the flexibility to delay reconvergence without changing the application's CFG. We implemented both the compiler analysis and the CFG transformation algorithms as compiler passes in LLVM 3.6 and evaluated them using a large set of GPU kernels. We also implemented and evaluated our proposed hardware SIMT reconvergence mechanism using a cycle level simulator.

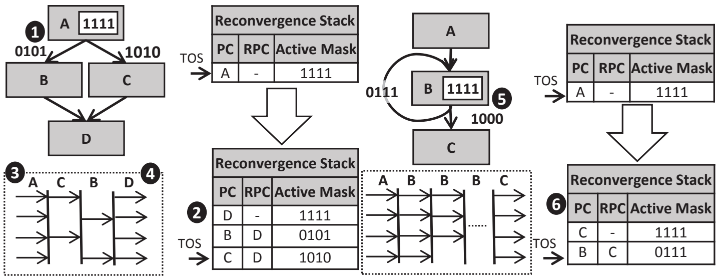
To the best of our knowledge, this is the first paper to propose techniques to execute arbitrary MIMD code with inter-thread synchronization on vector hardware. The contributions of this paper are:

- A static analysis technique for detecting potential SIMT-induced deadlocks in parallel kernels.
- A code transformation algorithm that modifies the CFG to eliminate SIMT-induced deadlocks.
- A SIMT-reconvergence mechanism that avoids some limitations of the compiler-only approach.
- A detailed evaluation of both approaches.

Codes to implement both approaches are available online [19]

II. SIMT Model Challenges

Conceptually, in a stack-based SIMT implementation, a per-warp stack is used to manage divergent control flow. Each entry contains three fields that represent a group of scalar threads executing in lock-step: (1) a program counter (PC) which stores the address of the next instruction to execute, (2) a reconvergence program counter (RPC) which stores the instruction address at which these threads should reconverge with other threads from the same warp and (3) an active mask that indicates which threads have diverged to this path. Initially, the stack has a single entry. Once a divergent branch is encountered, the PC field of the divergent entry is replaced with the RPC of the encountered branch and the branch outcomes' entries are pushed onto the stack. Only threads at the top of the stack entry are eligible for scheduling. Once executing threads reach reconvergence, their corresponding entry is popped out of the stack. In some implementations the stack is implemented and/or manipulated in software [4]–[7].



Reconvergence Stack		
PC	RPC	Active Mask
A	-	1111

Reconvergence Stack		
PC	RPC	Active Mask
D	-	1111
B	D	0101
C	D	1010

Reconvergence Stack		
PC	RPC	Active Mask
A	-	1111

Reconvergence Stack		
PC	RPC	Active Mask
C	-	1111
B	C	0111

(a) If-else Example **(b)** Loop Example

Fig. 2: Stack-Based Reconvergence

Figure 2 illustrates two examples of how the reconvergence stack changes after executing a divergent branch. Next, we briefly describe these changes, however, more in depth explanation for the stack operation on similar examples can be found in [20]. In Figure 2, bit masks (e.g., "1111") inside the CFG basic blocks indicate active threads executing a basicblock. Bit masks at the CFG edges indicate which threads diverged to which path. In Figure 2a, an if-else branch is executed at the end of basic block **A** ($\mathbf{BB_A}$) ❶. The PC of the first stack entry changes to the RPC of the branch (i.e., $\mathbf{BB_D}$), and two new entries are pushed onto the stack representing the branch taken and not-taken paths ❷. Only the top of the stack (TOS) entry is eligible for scheduling. Therefore, this warp starts executing threads that diverged to $\mathbf{BB_C}$ ❸. After these threads reach the reconvergence point (i.e., $\mathbf{BB_D}$), their entry is popped from the stack and the execution of $\mathbf{BB_B}$ starts. Eventually, all threads reconverge at $\mathbf{BB_D}$ ❹. This sequence allows the stack to track multiple reconvergence points in case of more complex CFGs that contain nested branches. In Figure 2b, a loop branch is executed at the end of $\mathbf{BB_B}$ ❶. One thread exits the loop while others continue iterating. Similar to the first example, the PC of the first stack entry changes to the RPC of the branch (i.e., $\mathbf{BB_C}$). Only a single entry is pushed onto the stack representing threads that diverged to the loop header ❷. The warp keeps iterating through the loop until all threads eventually exit the loop and start executing $\mathbf{BB_C}$. This mechanism enables the GPU to reconverge diverged threads improving SIMD units utilization. However, it imposes thread scheduling constraints which we discuss next.

A. SIMT-Induced Deadlocks

Conventionally, in MIMD environment, programmers do not worry about specifics of thread schedulers to write functionally correct code. It is assumed that the hardware guarantees "loose" fairness in thread scheduling [9]. SIMT programming models attempt to provide similar guarantees [1]. However, current SIMT implementations fail to do so. In current SIMT implementations, thread scheduling is constrained by the CFG of the executed application such that: if a warp **W** encounters a branch $\mathbf{BR_{T,NT\mapsto R}}$ with two possible successor basic blocks **T** and **NT** and reconvergence point **R**, it may diverge into two *splits* [21]; $\mathbf{W_{P_{T\mapsto R}}}$ and $\mathbf{W_{P_{NT\mapsto R}}}$. $\mathbf{W_{P_{T\mapsto R}}}$ contains threads that diverge to the taken path and $\mathbf{W_{P_{NT\mapsto R}}}$ contains threads that diverge to the not-taken path. On current SIMT implementations execution respects the following:

Constraint-1: Serialization. If $\mathbf{W_{P_{T\mapsto R}}}$ executes first then $\mathbf{W_{P_{NT\mapsto R}}}$ blocks until $\mathbf{W_{P_{T\mapsto R}}}$ reaches **R** (or vice versa)

Constraint-2: Forced Reconvergence. When $\mathbf{W_{P_{T\mapsto R}}}$ reaches **R**, it blocks until $\mathbf{W_{P_{NT\mapsto R}}}$ reaches **R** (or vice versa).

```
done = false;
*mutex = 0;
while(!done){
   if(atomicCAS(mutex,0,1)==0){
      //Critical Section
      atomicExch(mutex,0);
      done = true;
   }
}
```

Fig. 3: Modified SIMT compliant Spin Lock

Collectively we refer to these two constraints as the *reconvergence scheduling constraints*.

A *SIMT-induced deadlock* occurs when a thread is indefinitely blocked due to cyclic dependency between either of these constraints and a synchronization operation in the program. We categorize SIMT-induced deadlocks into two types according to their cause:

Conditional Loops: In Figure 1, all threads attempt to acquire the same lock (*mutex*). To acquire the lock, each thread repeatedly executes an atomic compare and swap. The loop condition evaluates to false for the single thread that successfully acquires the lock. We call this the leading thread. The remaining threads fail to acquire the lock. We call these lagging threads. The lagging threads continue to iterate through the loop waiting for the leading thread to release the lock by executing *atomicExch(...)*. However, in current SIMT implementations the leading thread never reaches *atomicExch(...)* as it is blocked by Constraint-2 (i.e., the leading thread is forced to wait for lagging threads at the loop exit reconvergence point). The lagging threads, on the other hand, cannot make forward progress because the leading thread owns the lock. This issue is known among GPU application developers [9], [22]–[26], and can be avoided by restructuring the CFG of the spin lock code as shown in Figure 3 (assuming that the compiler maintains the intended CFG unchanged).

Barriers: Following MIMD execution semantics, a programmer may place two barrier instructions in diverged code with the intention that every thread reaches either the first or second barrier before any thread continues. However, if divergence happens within a single warp, such code will either lead to a deadlock due to the serialization constraint (if barrier arrival is counted per scalar thread) or lead to hard to predict and/or implementation dependent behaviour (if barrier arrival is counted per warp [27]). Prior work studied detection of SIMT deadlocks due to barrier divergence [28]–[30].

B. Programmability Challenges

Figure 3 shows a well-known workaround to implement a spin lock on current SIMT implementations [24]. The while loop body includes both the lock acquisition and release. Hence, the reconvergence point of the while loop does not prevent the required communication between threads. However, besides not having a general solution, there are key downsides to solely relying on programmer intervention to work around current SIMT implementations' constraints:

Programming Practice: The *CUDA Programming Guide* [1] states that "For the purposes of correctness, the programmer can essentially ignore the SIMT behaviour, ..." and suggests that SIMT behaviour is primarily relevant for performance tuning purposes. In current SIMT implementations, dealing with SIMT deadlocks is the exception. SIMT deadlocks are also fundamental obstacles to the adoption of higher level programming models such as OpenMP 4.0 [31]. For example, the implementation of *omp_set_lock*(...) in the OpenMP 4.0 GPU library [17] generates code like that shown in Figure 1 and a general, correct and efficient implementation that considers current SIMT constraints is not clear as it is dependent on the *omp_set_unlock*(...) locations.

Unstandardized SIMT Behavior: Different GPU vendors have their own implementation of the SIMT execution model. As a consequence, the order in which divergent paths are executed and the locations of reconvergence points is not standardized and often undocumented. One reason a pure compiler or hardware solution is preferable is that programs dependent on undocumented behaviour may lack forward compatibility and portability.

Vulnerability to Compiler Optimizations: The manual workaround provided in Figure 3 assumes the compiler maintains the same control flow described by the high level language. However, optimizations that either modify the CFG or move statements across basic blocks can conflict with the intended manual transformation. For example, optimizations such as *jump-threading* and *simplifycfg* [32] transform the code in Figure 3 back to the code in Figure 1 as they eliminate redundant branches. Both are enabled with *-O1* in LLVM 3.6. LLVM is used for the standard CUDA and OpenCL compilation flow on several platforms [33], [34]. Thus, without a compiler that is aware of SIMT deadlocks, manual transformations are not reliable [2].

In this paper, we argue that the compiler and the hardware should be capable of abstracting away SIMT implementation nuances when necessary. Thus, the choice of conventional high level MIMD semantics versus emerging low level SIMD semantics (e.g., vote and shuffle operations in CUDA and subgroups in OpenCL 2.0) can be made by the programmer, rather than dictated by the underlying implementation.

III. SIMT Deadlock Static Analysis

This section proposes a static analysis that (A) detects potential SIMT deadlocks and (B) identifies safe locations for delayed reconvergence that allows for otherwise blocked inter-thread communication. The static analysis results are used by SIMT deadlock elimination techniques in Section IV.

A. SIMT-Induced Deadlock Detection

This section presents a static analysis technique that conservatively detects SIMT deadlocks due to conditional loops.

[2] We observed this problem in NVCC but less frequently. Test codes [19] that use variations of Figure 3 code terminate when compiled with device optimization disabled (i.e., with -Xcicc -O0 -Xptxas -O0) but deadlocks with default optimizations. This was tested with NVCC 7.5 on GTX1080 (Pascal). Online discussions also report this problem on other compilers (e.g., OpenCL on Xeon CPU [35] and GLSL on Nvida GTX 580M [36]).

Listing 1 Definitions and Prerequisites for Algorithms 1, 2, 3

-**BB(I):** basic block which contains instruction I (i.e., $\mathbf{I} \in \mathbf{BB(I)}$).
-$\mathbf{P_{BB1 \mapsto BB2}}$: union set of basic blocks in execution paths that connect **BB1** to **BB2**.
-**IPDom(I)** : immediate postdominator of instruction **I**. For non-branch instructions, IPDom(**I**) is the instruction immediately follows **I**. For branch instructions, IPDom(**I**) is defined as the immediate common postdominator for the basic blocks at the branch targets $\mathbf{BB_T}$ and $\mathbf{BB_{NT}}$.
-**IPDom(arg1,arg2)** is the immediate common postdominator for **arg1** and **arg2**; **arg1** and **arg2** could be either basic blocks or instructions.

-**LSet:** the set of loops in the kernel, where $\forall$ **L** $\in$ **LSet**:
-**BBs(L):** the set of basic blocks within loop **L** body.
-**ExitConds(L):** the set of branch instructions at loop **L** exits.
-**Exits(L):** the set of basic blocks outside the loop that are immediate successors of a basic block in the loop.
-**Latch(L):** loop **L** backward edge, **Latch(L).src** and **Latch(L).dst** are the edge source and destination basic blocks respectively.
-Basicblock **BB** is *reachable* from loop **L**, iff there is a non-null path(s) connecting the reconvergence point of **Exits(L)** with basic block **BB** without going through a barrier.
ReachBrSet(L,BB) is a union set of conditional branch instructions in all execution paths that connects the reconvergence point of **Exits(L)** with **BB**.
-Basicblock **BB** is *parallel* to loop **L**, iff there is one or more conditional branch instructions where $\mathbf{BBs(L)} \subset \mathbf{P_{T \mapsto R}}$ and $\mathbf{BB} \in \mathbf{P_{NT \mapsto R}}$ or vice versa, where **R** is the reconvergence point of the branch instruction. **ParaBrSet(L,BB)** is a union set that includes all branch instructions that satisfy this condition.

Algorithm 1 SIMT-Induced Deadlock Detection

```
 1: for each loop $L \in LSet$ do
 2:   $Shrd_{Reads}(L) = \emptyset$, $Shrd_{Writes}(L) = \emptyset$, $Redef_{Writes}(L) = \emptyset$
 3:   for each instruction I, where $BB(I) \in BBs(L)$ do
 4:     if I is a shared memory read $\wedge$ ExitConds(L) depends on I then
 5:       $Shrd_{Reads} = Shrd_{Reads} \cup I$
 6:     end if
 7:   end for
 8:   for each instruction I do
 9:     if BB(I) is parallel to or reachable from L then
10:       if I is a shared memory write then
11:         $Shrd_{Writes}(L) = Shrd_{Writes}(L) \cup I$
12:       end if
13:     end if
14:   end for
15:   for each pair ($I_R$,$I_W$), where $I_R \in Shrd_{Reads}(L)$ and $I_W \in Shrd_{Writes}(L)$ do
16:     if $I_W$ does/may alias with $I_R$ then
17:       $Redef_{Writes}(L) = Redef_{Writes}(L) \cup I_W$
18:     end if
19:   end for
20:   if $Redef_{Writes}(L) \neq \emptyset$ then Label L as a potential SIMT-induced deadlock.
21:   end if
22: end for
```

In the remainder of this section, we consider the case of a single kernel function **K** with no function calls (either natively or through inlining) and with a single exit (e.g., by merging return statements). We assume that **K** is guaranteed to terminate (i.e., is deadlock and livelock free) if executed on any MIMD machine. We also assume that **K** is barrier divergence-free [28] (i.e., for all barriers within the kernel, if a barrier is encountered by a warp, the execution predicate evaluates to true across all threads within this warp) [3]. This assumption excludes the possibility of SIMT deadlocks due to barriers in divergent code (whose detection techniques have been extensively studied in prior work [28]–[30]). We refer to memory spaces capable of holding synchronization variables as *shared* memory (i.e., including both global and shared memory using CUDA terminology). Listing 1 summarizes definitions used in Algorithms 1, 2, 3 which we discuss as we explain each algorithm.

Under barrier divergence freedom, the only way to prevent forward progress of a thread is a loop whose exit condition depends upon a synchronization variable. Forward progress is prevented (i.e., SIMT deadlock occurs) if a thread enters a loop for which the exit condition *depends* on the value of a shared memory location and that location will only be set by another thread that is blocked due to Constraint 1 or 2. For each loop, Algorithm 1 finds all shared memory reads that the loop exit could depend on ($\mathbf{Shrd_{Reads}}$). Then, it finds all shared memory writes that could be blocked due to SIMT constraints ($\mathbf{Shrd_{Writes}}$). It then checks if any of the $\mathbf{Shrd_{Writes}}$ could affect any of the $\mathbf{Shrd_{Reads}}$ values. If so, blocking these writes could result in a SIMT deadlock. This technique conservatively reports potential SIMT deadlocks. In practice it has a small false detection rate (more details in Section V-B1).

To determine $\mathbf{Shrd_{Reads}}$, we consider the static backward slice of the loop exit condition. If the loop exit conditions do not depend on a shared memory read operation that occurs inside the loop body then the loop cannot have a SIMT-induced deadlock. If a loop exit condition *does* depend on a shared memory read instruction $\mathbf{I_R}$, we add $\mathbf{I_R}$ in the set of shared reads $\mathbf{Shrd_{Reads}}$ on lines *4-7*. A potential SIMT-induced deadlock exists if any of these shared memory reads can be redefined by divergent threads. The next steps of the algorithm detect these shared memory redefinitions.

Lines *8-14* record, in set $\mathbf{Shrd_{Writes}}$, all shared memory write instructions $\mathbf{I_W}$ located in basic blocks that cannot be executed, due to the reconvergence scheduling constraints, by a thread in a given warp as long as some of the threads within that warp are executing in the loop. These basic blocks fall into two categories (Listing 1): The first category we call *reachable*. In a structured CFG, the reachable blocks are those blocks that a thread can arrive at following a control flow path starting at the loop exit. The second category we call *parallel*. The parallel blocks are those that can be reached by a path that starts from a block that dominates the loop header but avoids entering the loop. The detection algorithm requires that reconvergence points can be precisely determined at compile time. In our implementation, reconvergence is at immediate postdominators. We limit reachable basic blocks to those before a barrier due to our assumption of barrier-divergence freedom. Lines *15-20* check each pair of a shared memory read from $\mathbf{Shrd_{Reads}}$ and write from $\mathbf{Shrd_{Writes}}$ for aliasing. If they do or "may" alias, then the write instruction might affect the read value and hence affect the exit condition. If such a case occurs, the loop is labeled as a potential SIMT-induced deadlock and we add the write to the redefining writes set ($\mathbf{Redef_{Writes}}$).

For example, consider the application of Algorithm 1 to the code in Figure 1 and 3. In Figure 1, the loop exit is data dependent on the *atomicCAS* instruction. There is one shared memory write that is reachable from the loop exit, the *atomicExch* instruction. The two instructions alias. Hence, SIMT-induced deadlock is detected. In Figure 3, although the

[3] In section IV-B, we propose a hardware mechanism that is capable of supporting MIMD code with divergent barriers.

Algorithm 2 Safe Reconvergence Points

```
1:  SafePDom(L) = IPDom(Exits(L)) ∀ L ∈ LSet
2:  for each loop L ∈ LSet do
3:      for each $I_W$ ∈ $Redef_{Writes}(L)$ do
4:          SafePDom(L) = IPDom(SafePDom(L), $I_W$)
5:          if BB($I_W$) is reachable from L then
6:              for each branch instruction $I_{BR}$ ∈ ReachBrSet(L, BB($I_W$)) do
7:                  SafePDom(L) = IPDom(SafePDom(L), $I_{BR}$)
8:              end for
9:          end if
10:         if BB($I_W$) is parallel to L then
11:             for each branch instruction $I_{BR}$ ∈ ParaBrSet(L, BB($I_W$)) do
12:                 SafePDom(L) = IPDom(SafePDom(L), $I_{BR}$)
13:             end for
14:         end if
15:     end for
16: end for
17: resolve_SafePDom_conflicts()
```

loop exit is control dependent on the *atomicCAS* instruction, there are no shared memory write instructions that are parallel to, or reachable from, the loop exit. Therefore, *no* SIMT deadlock is detected.

B. Safe Reconvergence Points Identification

This stage identifies code locations where reconvergence of loop exits should be moved to allow for inter-thread communication that otherwise blocked by SIMT Constraints[4]. The key idea is that we can choose any postdominator point as the reconvergence point including the kernel exit. However, from a SIMD utilization perspective, it is preferable to reconverge at the earliest postdominator point that would not block the required inter-thread communication. We call these postdominator points *safe postdominators* (SafePDoms).

The notion of delaying reconvergence to overcome SIMT-induced deadlocks is intuitive when we consider the second scheduling constraint (i.e., the forced reconvergence). A SIMT deadlock, due to this constraint, happens when threads that exit a conditional loop are blocked at the loop reconvergence point indefinitely waiting for looping threads to exit, while looping threads are waiting for blocked threads to proceed beyond the current reconvergence point to set their exit conditions and allow them to exit the loop. This cyclic dependency breaks if the loop reconvergence point is delayed so that even if threads were to pass the new delayed reconvergence point they could not affect the loop exit conditions (i.e., they could not reach a redefining write). Therefore, SafePDoms should be computed so that they postdominate the loop exit branches and all control flow paths that lead to redefining writes from the loop exits (lines *4-9* in Algorithm 2).

It is less intuitive to think about how delaying reconvergence helps with the first scheduling constraint (i.e., serialization). To understand this, it is necessary to understand how SafePDoms are used by the SIMT deadlock elimination technique. Due to the serialization constraint, threads iterating in a loop could be always prioritized over the loop exit path and/or other paths that are parallel to the loop. A SIMT-induced deadlock occurs if these indefinitely blocked paths must execute to enable the exit conditions of the looping threads. To avoid this, our compiler based SIMT deadlock elimination algorithm (explained in more details in Section IV-A) replaces the backward edge of a loop identified by Algorithm 1 with two edges: a forward edge towards the loop's SafePDom, and a backward edge from SafePDom to the loop header. This modification combined with the forced reconvergence constraint, guarantees that threads iterating in the loop wait at the SafePDom for threads executing other paths postdominated by SafePDom before attempting another iteration. Accordingly, SafePDom should postdominate the original loop exits, the redefining writes, and all control flow paths that could lead to redefining writes that are either reachable from the loop (lines *4-9* in Algorithm 2) or parallel to it (lines *10-14* in Algorithm 2).

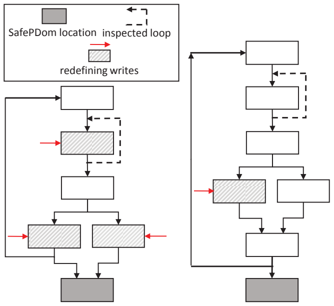


Fig. 4: SIMT-induced deadlock scenarios

However, conflicts in identifying SafePDoms for different loops may exist. In particular, SafePDom(L) should postdominate SafePDom of any loop in the path between the exits of loop **L** and SafePDom(L). Otherwise, it is not a valid reconvergence point. We resolve these conflicts by recalculating SafePDom(L) for each loop to postdominate all SafePDom of loops in the path between loop **L** exits and SafePDom(L). The process is then iterated until it converges. Convergence is guaranteed because in the worst case, the exit node of the kernel is a common postdominator. We force a single exit using a merge return pass that merges multiple return points (if they exist) into one.

The application of Algorithm 2 to the code in Figure 1 is straightforward. The initial reconvergence point of the loop is the instruction that immediately follows its exit edge. However, there is a redefining write (the *atomicExch* instruction) in a basic block that is reachable from the *while* loop exit. Thus, line *4* of Algorithm 2 updates SafePDom of the loop to be the instruction that immediately follows the *atomicExch*. No further updates to SafePDom by the rest of the algorithm. Figure 4 shows the appropriate choice of SafePDom for more complex scenarios. For example, the CFG to the right resembles a scenario found in the sort kernel of BarnesHut application [37] when compiler optimizations are enabled. Threads iterating in the self loop are supposed to wait for a ready flag to be set by other threads executing the outer loop. Threads executing the outer loop may need more than one iteration to set the ready flag of waiting threads. The dark basic block is an appropriate choice of SafePDom as

[4] Due to the serialization constraint, delaying reconvergence is necessary but not sufficient to eliminate SIMT deadlocks on current SIMT implementations (more in Section IV-A).

Algorithm 3 SIMT-Induced Deadlock Elimination

```
 1: SwitchBBs = $\emptyset$
 2: for each loop L $\in$ LSet do
 3:   if L causes potential SIMT-induced deadlock then
 4:     if SafePDom(L) $\notin$ SwitchBBs then
 5:       SwitchBBs = SwitchBBs $\cup$ SafePDom(L)
 6:       if SafePDom(L) is the first instruction of a basic block BB then
 7:         Add a new basic block $BB_S$ before the BB.
 8:         Incoming edges to $BB_S$ are from BB predecessors.
 9:         Outgoing edge from $BB_S$ is to BB.
10:       else
11:         Split BB into two blocks $BB_A$ and $BB_B$.
12:         $BB_A$ contains instructions up to but not including SafePDom(L).
13:         $BB_B$ contains remaining instructions including SafePDom(L).
14:         $BB_S$ inserted in the middle, $BB_A$ as predecessor, $BB_B$ as successor.
15:       end if
16:       Insert a PHI node to compute a value cond in $BB_S$, where:
17:         for each predecessor Pred to $BB_S$
18:           cond.addIncomingEdge(0,Pred)
19:         end for
20:       Insert Switch branch swInst on the value cond at the end of $BB_S$, where:
21:         swInst.addDefaultTarget($BB_S$ successor)
22:     end if
23:     $BB_S$ is the basic block immediately preceding SafePDom(L)
24:     Update PHI node cond in $BB_S$ as follows:
25:     cond.addIncomingEdge(UniqueVal,Latch(L).src) -unique to this edge
26:     Update Switch branh swInst at the end of $BB_S$ as follows:
27:     swInst.addCase(UniqueVal,Latch(L).dst)
28:     Set Latch(L).dst = $BB_S$.
29:   end if
30: end for
```

it postdominates all reachable paths to the redefining writes (i.e., leading threads may only wait for lagging ones after they finish all iterations of the outer loop).

IV. MIMD Execution on SIMT Machines

This section proposes two techniques to enable the execution of MIMD code with inter-thread synchronization on SIMT machines. The first technique is SSDE; a static SIMT deadlock elimination algorithm. The second technique is AWARE; an adaptive hardware reconvergence mechanism that overcomes key limitations in SSDE. Both techniques leverage the static analysis techniques proposed in Section III.

A. SSDE: Static SIMT Deadlock Elimination

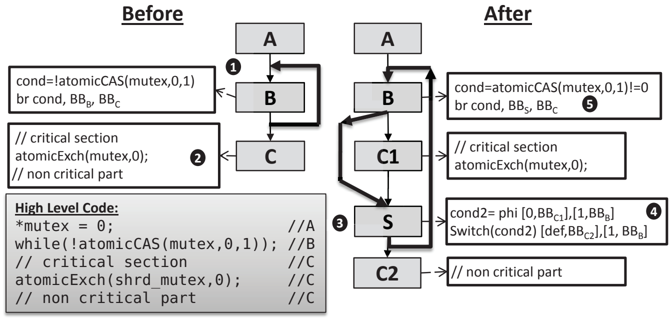
To avoid SIMT-induced deadlocks, loose fairness in the scheduling of the diverged threads is required. Constrained by existing SIMT implementations, we achieve this by manipulating the CFG. Algorithm 2 picks the earliest point in the program that postdominates the loop exit and all control flow paths that could affect the loop exit condition if executed. Algorithm 3 modifies the CFG by replacing the backward edge of a loop identified by Algorithm 1 by two edges: a forward edge towards SafePDom, and a backward edge from SafePDom to the loop header. This guarantees that threads iterating in the loop wait at the SafePDom for threads executing other paths postdominated by SafePDom before attempting another iteration allowing for inter-thread communication. Algorithm 3 is a generalization and automation for the manual workaround shown in Figure 3. We refer to this code transformation as Static SIMT Deadlock Elimination.

Figure 5 shows the steps of our SSDE algorithm to eliminate the SIMT deadlock in the code from Figure 1. The CFG on the left is the original and the one to the right is after the transformation. According to Algorithm 2, the SafePDom of the self loop at $\mathbf{BB_B}$ ❶ is the first instruction after the *atomicExch* ❷. Accordingly, a new basic block $\mathbf{BB_S}$ is added ❸ as described in lines *11-14*. A PHI node and switch branch instructions are added to $\mathbf{BB_S}$ ❹ according to lines *16-26*. Finally, destination of the loop edge is modified to $\mathbf{BB_S}$ ❺ on line *28*. Thus, the end result is that the backward branch of the loop is replaced by two edges: a forward edge to $\mathbf{BB_S}$ and a backward edge to the loop header. The new added basic block acts as a *switch* that redirects the flow of the execution to an output basic block according to the input basic block. The PHI node is translated by the back-end to *move* instructions at predecessors of $\mathbf{BB_S}$. The switch instruction is lowered into branches such that it diverges first to the default not-taken path. This guarantees that $\mathbf{BB_S}$ remains the IPDOM of subsequent backward branches forcing (potentially) multiple loops to interleave their execution across iterations.

Algorithm 3 preserves the MIMD semantics of the original code because the combination of the PHI node and branch in $\mathbf{BB_S}$ guarantees the following: (1) each control flow path in the original CFG has a single equivalent path in the modified CFG that maintains the same observable behaviour of the original path by maintaining the same sequence of static instructions that affect the state of the machine (e.g., update data registers or memory), for example, the execution path ($\mathbf{BB_{C1}}$-$\mathbf{BB_S}$-$\mathbf{BB_{C2}}$) in the transformed CFG is equivalent to the path ($\mathbf{BB_C}$) in the original CFG. The loop path ($\mathbf{BB_B}$-$\mathbf{BB_B}$) is equivalent to ($\mathbf{BB_B}$-$\mathbf{BB_S}$-$\mathbf{BB_B}$) and (2) control flow paths in the modified CFG that have no equivalent path in the original CFG are fake paths and would not execute, for example, the path ($\mathbf{BB_{C1}}$-$\mathbf{BB_S}$-$\mathbf{BB_B}$) in the modified CFG would not execute because the PHI node in $\mathbf{BB_S}$ controls the switch condition such that if a thread is coming from $\mathbf{BB_{C1}}$ it branches to $\mathbf{BB_{C2}}$. The elimination algorithm does not reorder memory instructions or memory synchronization operations and therefore it does not impact the consistency model assumed by the input MIMD program. Formal proof outlines can be found in [38].

Algorithm 3 ensures a SIMT-induced deadlock free execution. For example, threads that execute the branch at the end of $\mathbf{BB_B}$ wait at the beginning of $\mathbf{BB_S}$ for other threads within the same warp to reach this reconvergence point. This allows the leading thread that acquired the lock to execute the *atomicExch* instruction and release the lock before attempting another iteration for the loop by the lagging threads.

1) Compatibility with Nvidia GPUs: There is no official documentation for Nvidia's reconvergence mechanism; however, a close reading of Nvidia's published patents [5], [6] and disassembly [39] suggests that for a branch instruction to have a reconvergence point at its immediate postdominator, the branch must dominate its immediate postdominator, and for loops, the loop header must dominate the loop body (i.e., the loop must be a single-entry/reducible loop). We account for the additional constraints as follows: first, in Algorithm 1, reconvergence points of divergent branches are not necessarily their immediate postdominators. If the branch's immediate postdominator is not dominated by the branch basic block



```
*mutex = 0;                    //A
while(!atomicCAS(mutex,0,1)); //B
// critical section           //C
atomicExch(shrd_mutex,0);     //C
// non critical part          //C
```

Fig. 5: SIMT-Induced Deadlock Elimination Steps

then the reconvergence point of the branch is considered the same as the immediate dominating branch that dominates its immediate postdominator. Second, in Algorithm 3, we have to guarantee that the new added basic block is a valid reconvergence point. We guarantee this by forcing the new created loop to be a single entry loop.

2) SSDE Limitations: SSDE has some limitations that motivated the exploration of a combined hardware and compiler approach. These limitations include the following:

Inter-procedural Dependencies: We assumed a single kernel function with no function calls to other procedures. This only detects SIMT deadlocks when synchronization is local to a function. It is possible to extend our detection algorithm to handle function calls with an inter-procedural analysis that tracks dependencies across function calls [40], [41]. However, extending the elimination algorithm is less straightforward.

CFG Modifications: To avoid SIMT-induced deadlocks, the elimination algorithm modifies the original application's CFG. This could make debugging the original code a harder task [42], [43]. The modified CFG adds more instructions that do not do useful work except to work-around the constraints of the current reconvergence mechanisms. These added instructions increase the static code size and dynamic instructions count. Finally, the modified CFG could result in increased liveness scope for some variables.

False Detections: Since we rely upon alias analysis it is possible to flag a loop causing SIMT deadlock where a programmer may reason such a deadlock cannot occur.

Indirect Branches: The static analysis can be extended to deal with indirect branches with known potential targets (as supported in Nvidia's PTX 2.1). Without clues about potential targets, the analysis would be conservative in labeling potential SIMT-induced deadlocks including indirect branches as they might form loops. This leads to significant overheads due to excessive potential false detections.

Warp-Synchronous Behaviour: Some GPU applications rely on implicit warp synchronous behaviour. For such applications, it is necessary to disable our transformation possibly using pragmas.

B. AWARE: Adaptive Warp Reconvergence

This section presents our proposal for an adaptive hardware warp reconvergence mechanism which we refer to as AWARE. AWARE is a MIMD-Compatible reconvergence mechanism that avoids most limitations inherent in a compiler-only approach. It extends the Multi-Path (MP) execution model [44]. The goal of MP is to impose fewer scheduling constraints than stack-based reconvergence to improve performance. To achieve this, MP decouples the tracking of diverged splits from their reconvergence points using two tables as shown in Figure 6. The warp Split Table (ST) records the state of warp splits executing in parallel basic blocks, which can be scheduled concurrently. The Reconvergence Table (RT) records reconvergence points for the splits. ST and RT tables hold the same fields as the SIMT stack with RT holding an extra field called the Pending Mask. The Pending Mask represents threads that have not yet reached the reconvergence point. This decoupling enables MP to not serialize divergent paths up to the reconvergence point. However, as described by ElTantawy and Aamodt [44], it still forces threads to wait at immediate postdominator reconvergence points. AWARE modifies MP as follows:

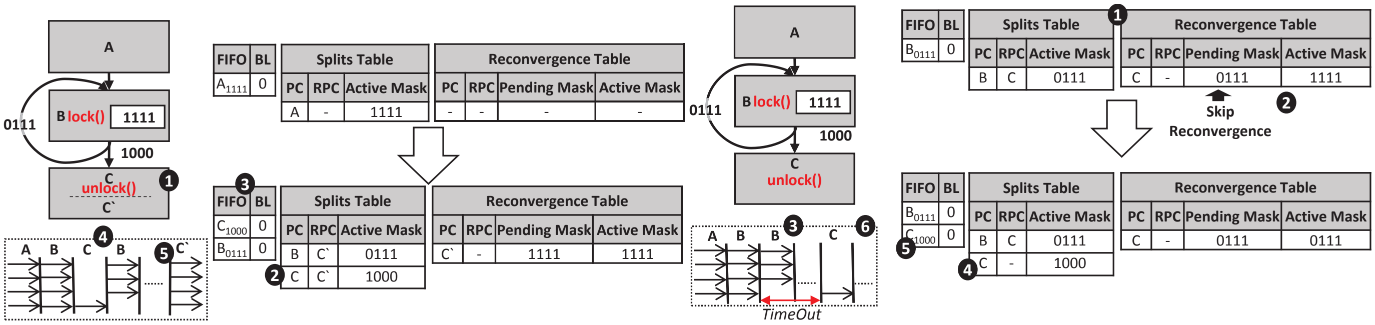
Warp Splits Scheduling: MP applies changes to the scoreboard and instruction buffer designs to maximize thread level parallelism. On the contrary, AWARE starts from a simplified version that limits the architectural changes to the divergence unit. In contrast to MP, AWARE allows only one warp split to be eligible for scheduling at a time; thus limiting interleaving of warp splits. AWARE also conservatively respects register dependencies on a warp granularity. Thus, it does not require changes to the current scoreboard design[5].

AWARE switches from one warp split to another only after encountering divergent branches, reconvergence points, or barriers. A warp split is selected for greedy scheduling in FIFO order with respect to the ST. A warp split is pushed into the FIFO when it is first inserted into the ST as an outcome of a branch instruction or as a reconverged entry. It is scheduled when it reaches the output of the FIFO. It is popped out of the FIFO when its entry in ST is invalidated after encountering a branch instruction or reaching a reconvergence point. This scheduling replaces the depth-first traversal of divergent control-flow paths imposed by the reconvergence stack with a breadth-first traversal. This guarantees fairness in scheduling different control paths. Greedy scheduling of the output entry ensures good memory performance [46].

Handling Barriers: In AWARE, we support barriers in divergent code by adding a field in the FIFO queue to mark warp splits that reach the barrier as blocked. The PC of the blocked split is updated in both the ST and the FIFO to point to the instruction after the barrier. When a blocked entry is at the FIFO output, it is pushed to the back of the FIFO without being scheduled for fetch. Splits are released when all threads reach the barrier.

Delayed Reconvergence: MP restricts reconvergence to IPDOM points, which can lead to SIMT deadlocks. For any loop that Algorithm 1 reports as being the potential cause of a SIMT deadlock, AWARE uses the reconvergence points

[5] Current GPUs use a per-warp scoreboard mechanism to track pending writes to registers on warp granularity [45].



(a) Delayed Reconvergence

(b) Timed-out Reconvergence

Fig. 6: MIMD-Compatible Reconvergence Mechanism Operation

computed using Algorithm 2. Since AWARE does not have the serialization constraint, Algorithm 2 can be simplified to consider only redefining writes that are reachable from the loop reconvergence points (removing lines *10–14*). It is necessary to recalculate reconvergence points of other branches to guarantee that the reconvergence point of any branch post-dominates the reconvergence points of all branches on the path from the branch to its reconvergence point. Figure 6a illustrates the operation of AWARE with delayed reconvergence. The reconvergence point of the loop is modified from the IPDOM point (i.e., **C**) to SafePDom (i.e., **C'**) ❶. Once the loop branch instruction is encountered, the RPC field of RT and ST are updated to **C'** ❷. The FIFO queue has two valid entries with priority given to the not-taken path ❸. Hence, the thread that diverged to $\mathbf{BB_C}$ (i.e., exited the loop) executes first ❹. It releases the lock and waits at the reconvergence point (**C'**). Eventually all threads exit the loop and reconverge at **C'** ❺.

Timed-out Reconvergence: To avoid the limitations of the compiler approach we extend AWARE with a timeout mechanism. Figure 6b shows how the timed-out reconvergence mechanism operates on the code snippet in Figure 1. In this case, we assume that reconvergence points are set to the IPDOM points (i.e., delayed reconvergence is off). Initially, there is a single entry in the ST ❶ representing threads that are iterating through the loop attempting to acquire the lock whereas the thread that exits the loop keeps waiting at the reconvergence point ❷. This state continues ❸ until the reconvergence timeout logic is triggered. Once the waiting time of threads at the reconvergence point exceeds the *TimeOut* value, a new entry is added to the ST with the same PC and RPC of the entry in the RT and an active mask that is the subtraction of the Active Mask and the Pending Mask from the RT entry ❹. The new entry is added to the FIFO queue ❺. The new entry C_{1000} is guaranteed to be executed as entry B_{0111} gets to the tail of the FIFO queue once the loop branch is executed ❻. In a nested control flow graph, threads that skip reconvergence at a certain point are still able to reconverge at the next reconvergence point because the RT tracks nested reconvergence points. The *TimeOut* value could be empirically determined by profiling a large number of GPU kernels. It should be large enough not to impact reconvergence behaviour of regular GPU applications.

AWARE Basic Implementation: With 32 threads per warp, ST and RT tables have maximum theoretical size of 32 entries per warp (max splits is 32 and RT entries added only when a split diverges which implies a maximum of 32 RT entries). Thus, AWARE can be realized using RAMs by adding a 5-bit field to the FIFO, ST and RT tables. Upon insertion of an entry into the ST, the 5-bit index of this ST entry is stored in the corresponding FIFO queue entry. Upon insertion of an entry in the RT table, the RT entry index is stored in a field in its ST entries. Look ups into ST and RT tables use these indices with no need to search the contents of the tables. We keep track of invalid entries in a free queue implemented with a small (32 x 5-bits) stack. To insert an ST or RT entry the next index from the associated free queue is used. We model insertion and lookup latency to be 1 cycle. A warp split inserted into ST needs to wait for the next cycle to be eligible for fetch.

AWARE Virtualized Implementation: The basic implementation has a large area overhead (about 1 KB storage requirement per warp). This is because the implementation has to consider the worst case scenario (i.e., 32 ST and RT entries). However, typical occupancy of ST and RT tables is much lower. Therefore, we study the impact of virtualizing ST and RT tables by spilling entries that exceed their physical capacity to the memory system and filling them when they are to be scheduled (for ST entries) or updated (for RT entries).

Figure 7 illustrates our virtualized AWARE implementation. A branch instruction of a warp is scheduled only if both the ST and RT Spill Request Buffers of this warp are empty. Also, instructions from a warp are eligible for scheduling only if the warp Reconverged Entry and Pending Mask Updates Buffers are empty. When a new entry is required to be inserted into a full ST or RT table, an existing entry is spilled to their respective Spill Request Buffers. We use a FIFO replacement policy for the ST and an LRU replacement polciy for the RT [6]. When an entry is spilled its corresponding entry in the FIFO is labeled as virtual. When a virtual entry is at the FIFO output, a fill request for this entry is sent and the entry is labeled as transient. This is to avoid sending multiple fill requests for

[6] This is essentially to leverage the existing FIFO for the ST and the age bits used for Timeout calculation in the RT.

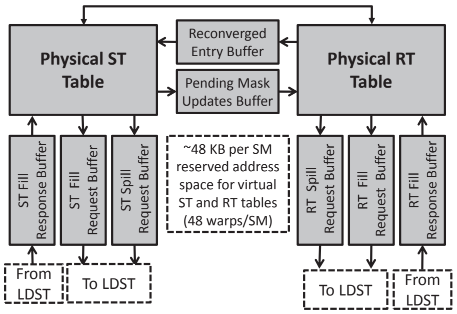


Fig. 7: AWARE Virtualized Implementation

TABLE I: Evaluated Kernels

Kernel	Language	Description
HT [25]	CUDA	Chained Hash Table of 80K entries and 40K threads
ATM [25]	CUDA	ATM 122K transactions, 1M accounts, 24K threads
CP-DS [25], [47]	OpenCL	Distance Solver in Cloth Physics simulation
BH-TB [37], [48]	CUDA & OpenMP	Tree Building in BarnesHut (30,000 bodies)
BH-SM [37], [48]	CUDA & OpenMP	Summarization kernel in BarnesHut
BH-ST [37], [48]	CUDA & OpenMP	Sort kernel in BarnesHut
BH-FC [37], [48]	CUDA & OpenMP	Force Calculation in BarnesHut
TL [49]	OpenMP	Test Lock microbenchmark
AM [50]	OpenMP	Find Array Max from 133K entries in an array

the same entry. Also, the entry is pushed to the back of the FIFO. When a pending mask update is required for a virtual RT entry, a fill request is sent for this entry and the pending mask update buffer remains occupied until a response to the fill request is received and the entry is inserted in the RT table. Further, a fill request is sent when an RT entry reconvergence is timed-out. ST spill requests is 12 bytes and RT spill requests is 16 bytes. Conveniently, a global memory address space of $32 \times 32 = 1KB$ bytes per warp is reserved for virtual ST and RT Tables. The FIFO and free queues use virtual entry IDs (between 0 and warp-size-1) that are used along with the warp id to decide the address of spill and fill requests. These virtual IDs are stored as new fields to the physical ST and RT entries. Age bits for RT entries are not virtualized. Each buffer in Figure 7 is sized to queue only one entry. As we discuss in Section V-B2, the physical sizes of ST and RT can be set to 4 and 2 entries respectively with limited impact on performance (see Figure 12). This effectively reduce the storage required per warp by a factor of 5× compared to the basic implementation which makes the storage requirement comparable to the reconvergence stack.

V. Implementation and Evaluation

A. Implementation and Methodology

We implemented both the detection and the elimination algorithms as two passes in LLVM 3.6 [32]. For alias analysis, we used the basic alias analysis pass in LLVM [51] [7]. We ran passes that inline function calls and lower generic memory spaces into non-generic memory spaces before our detection and elimination passes. We used the approach described in [52] to identify irreducible loops. We disabled optimizations in the back-end compilation stages from LLVM-IR to PTX and later to SASS to guarantee that there are no further alternation in our generated CFG. This could be avoided if the elimination algorithm is applied at the SASS code generation stage. We also implemented AWARE in GPGPU-Sim 3.2.2 [53], [54]. We use the TeslaC2050 configuration released with GPGPU-Sim. However, we replaced the Greedy Then Oldest (GTO) scheduler with a Greedy then Loose Round Robin (GLRR) scheduler that forces loose fairness in warp scheduling as we observed that unfairness in GTO leads to livelocks due to inter-warp dependencies on locks [8]. Modified GPGPU-Sim and LLVM codes can be found online [19].

TABLE II: Code Configuration Encoding

Code Configuration XYZ					
X: Original code format		Y: Compiler optimizations		Z: Our Analysis/Transf.	
M	MIMD	0	-O0	S	Elimination
S	SIMT	2	-O2	D	Delayed Rec.
		2*	-O2 without jumpthreading and simplifycfg		

We use CUDA, OpenCL and OpenMP applications for evaluation. OpenCL applications are compiled to LLVM Intermediate Representation (LLVM-IR) using Clang compiler's OpenCL frontend [55] with the help of *libclc* library that provides LLVM-IR compatible implementation for OpenCL intrinsics [56]. For CUDA applications, we use *nvcc-llvm-ir* tool [57]. This tool allows us to retrieve LLVM-IR from CUDA kernels by instrumenting some of the interfaces of *libNVVM* library [58]. OpenMP compilation relies on recent support for OpenMP 4.0 in LLVM [15]. For hardware results, we use a Tesla K20C (Kepler). Our benchmarks include OpenCL SDK 4.2 [59], Rodinia 1.0 [60], KILO TM [25], both CUDA and OpenMP versions of BarnesHut [37], [48], and two OpenMP microbenchmarks [49], [50]. OpenCL SDK and Rodinia applications do not have synchronization between divergent threads, however, kernels from [25], [37], [48]–[50] require synchronization between divergent threads. Table I describes briefly all the kernels that are affected by our elimination algorithm.

Table II shows the encoding we use in the results discussion. The first character encodes whether the code accounts for the reconvergence constraints (S for SIMT) or not (M for MIMD). The second encodes the level of the optimization performed on the code before we run our passes. The last character encodes the type of analysis or transformation performed, it could be either applying the elimination algorithm (S), calculating the delayed reconvergence points without CFG transformations (D) or neither (-).

B. Evaluation

1) Static SIMT Deadlock Elimination Evaluation:

Detection Algorithm Evaluation: First, we ran our detection on the CUDA and OpenCL applications mentioned in Section V-A. These applications were written to run on current GPUs, so they are supposed to be free of SIMT-induced

[7] We did not observe lower false detection rates using other LLVM alias analysis passes as they focus on optimizing inter-procedural alias analysis.

[8] For kernels under study that do not suffer livelocks with GTO, the impact of GLRR on L1 miss rate is minimal compared to GTO with the exception of BH-ST (in Table I) which suffers 7% increase in L1 cache misses [46]. The study of "fair" and efficient warp schedulers is left to future work.

TABLE III: Static Overheads for the Elimination Algorithm

Kernel	Tf. Loops				SASS-Static Inst.						SASS-Used Reg. + Stack Size (bytes)					
	-O0		-O2		M0-	S0-	M0S	M2-	S2*-	M2S	M0-	S0-	M0S	M2-	S2*-	M2S
	T	F	T	F												
HT	1	-	1	-	408	436	422	184	177	198	12+128	12+128	12+128	16+0	16+0	18+0
ATM	1	1	1	-	506	506	527	233	247	394	11+168	11+168	11+168	20+0	21+0	30+0
CP-DS	2	-	2	-	624	631	617	631	631	632	31+0	37+0	37+0	31+0	39+0	46+0
BH-TB	1	3	1	3	1871	1836	1899	534	891	1164	17+344	17+336	17+344	40+0	38+24	40+296
BH-SM	-	2	-	1	1983	1983	2011	933	996	1192	36+176	36+176	36+176	55+0	55+0	72+0
BH-ST	1	-	1	1	520	485	541	219	226	261	16+96	16+88	16+96	22+0	22+0	24+0
BH-FC	-	3	-	1	1549	1549	1591	765	765	891	36+272	36+272	36+272	48+40	48+32	48+56

TABLE IV: Detection Pass Results

Opt. Level	Krnls	N. Brs	N. Lps	Detections	T. Det.	F. Det.	F. Rate
O0	159	2751	277	14	0	14	5.05%
O2		1832	242	14	4	10	4.13%

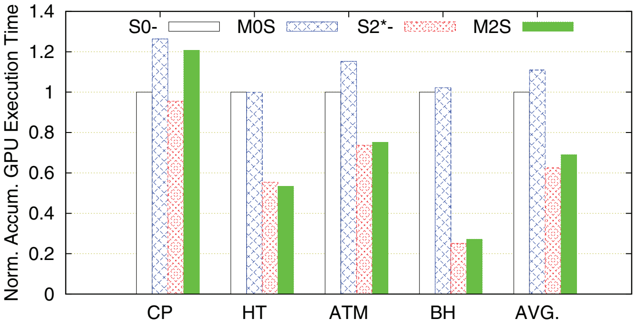


Fig. 8: Normalized Accumulative GPU Execution Time

deadlocks. Table IV summarizes the detection results from 159 kernels. With default compiler optimizations in LLVM 3.6 enabled, four true detections in four different kernels (ATM, HT, CP-DS and BH-ST) were successfully detected (See Vulnerability to Compilers' Optimizations in Section 2.2) [9]. True detections were confirmed by both manual inspection of the machine code and by verifying that the code does not terminate when executed. False detections include four detections that occur in the histogram256 kernel from the OpenCL SDK and one false detection in the bucketsort kernel in MGST from Rodinia. In both cases, kernels are dependent on warp synchronous behaviour to implement an atomic increment operation (because atomics were not supported in earlier CUDA versions). Both kernels will not produce their expected results with either our compiler or hardware approaches (see IV-A2). This issue is left to future work. Other false detections exist in applications that involve inter-thread synchronization [25], [37]. No false negatives were observed (i.e., after using SSDE, no deadlocks were observed at runtime). As an additional check, we ran the detection pass on the transformed kernels after applying the elimination pass which yielded *no* detections.

Besides alias analysis, main reasons for false detection include conservative reachability and dependence analysis that relies on the static CFG layout and instruction dependencies without considering the dynamic behaviour. For example, one of BH-FC loops is control dependent on a shared array. However, the conditional check on this shared array uses the __all() CUDA intrinsic that evaluates the same for all threads within a warp forcing all threads within the same warp to exit at the same iteration. This motivates leveraging runtime information and elaborate static analysis as a future work.

Elimination Algorithm Evaluation: We rewrote CUDA and OpenCL kernels that require inter-thread synchronization assuming the simpler MIMD semantics. Table III compares six different code versions in terms of static instructions, register and stack storage per thread. The *M0-* and *M2-* versions deadlock on current GPUs. Enabling default compiler optimizations in *S2-* led all our four applications to deadlock. In configuration *S2*-*, we selectively enabled passes that were empirically found not to conflict with the manual code transformation. Specifically, we excluded *-simplifycfg* and *-jumpthreading* passes.

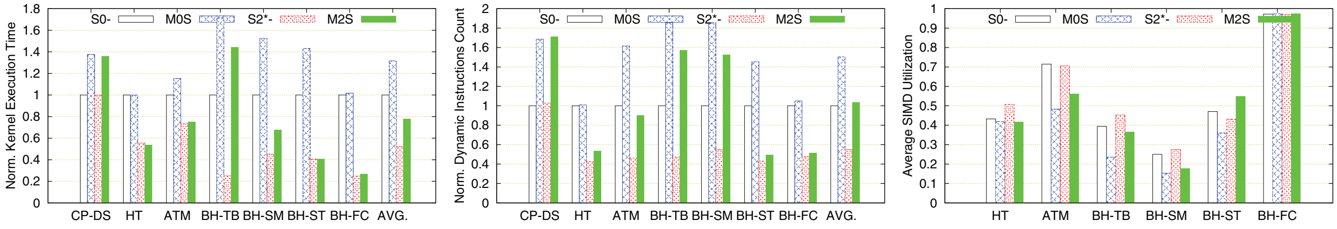
For the non-optimized versions (i.e., *S0-* and *M0S*), Table III shows that static instruction overhead is small in both manual and compiler based transformations (*S0-* and *M0S* are comparable to *M0-*). Also, they have little to no overhead in terms of registers (e.g., CP) compared to the MIMD version when we consider the non-optimized versions. Turning on compiler optimizations generally reduces stack usage and increases the number of registers. This has a negative impact on kernels with false positives. For example, in BH-TB kernel for *M2S*, there is an increase in both registers and stack usage. This is due to increased register spills resulting from increased register liveness scopes after applying SSDE.

We evaluated run-time overheads using performance counters. Figure 8 shows accumulated GPU time for all kernel launches averaged over 100 runs. HT and ATM have a single kernel, CP has four kernels and BH has 6 major kernels. *M0S* has a 10.9% overhead on average compared to *S0-*. *M2S* leads to a speedup of 44.7% compared to *S0-*. *M2S* is 8.2% slower compared to *S2*-*. For HT, the benefit of enabling all compiler optimizations overcomes automated transformation overheads leading to improvement compared to *S2*-*. Figure 9 breaks down the results of kernels that are affected by our transformation. As shown in Figure 9a, some of the kernels that exhibit no false detections still suffer from considerable overheads due to conservative estimation of safe postdominator points (e.g., CP-DS and ST), which reduces both SIMD utilization (e.g., ST in Figure 9c)[10] and increases dynamic instruction (e.g., CP-DS and ST in Figure Figure 9b).

For kernels with false detections, the overhead is dependent on the run-time behaviour. For example, although BH-FC has 3 false detections in its *M0S* version, this hardly impacts its performance. This is mainly because the kernel has very high utilization which indicates low divergence rate. Thus, delaying reconvergence does not impact its SIMD utilization. In other cases (e.g. BH-TB and BH-SM), false detections lead to significant performance overheads. We attribute these overheads to both the reduced SIMD utilization (Figure 9c) and increased dynamic instruction count (Figure 9b). In BH-

[9] Using Nvidia's NVCC compiler, only Cloth Physics deadlocks after turning on compiler optimizations.

[10] Nvidia profiler does not measure SIMD efficiency for OpenCL applications. Thus, SIMD utilization for CP is not reported.



(a) Norm. Kernel Execution Time **(b)** Norm. Dynamic Instruction Count **(c)** Average SIMD Utilization

Fig. 9: Evaluation of the Static SIMT-Induced Deadlock Elimination on Tesla K20C GPU

TABLE V: SSDE Evaluation on OpenMP Kernels

Kernel	C-4T -O2	G-OMP M2-	G-OMP M2S	G-CUDA S2*-	Kernel	C-4T -O2	G-OMP M2-	G-OMP M2S	G-CUDA S2*
TL-FG	1	D	**3.41**	N/A	BH-TB	1	D	0.28	9.66
TL-CG	1	D	0.01	N/A	BH-SM	1	3.78	**1.72**	3.54
AM-FG	1	D	**32.58**	N/A	BH-FC	1	0.70	0.73	1.25
AM-CG	1	D	**17.87**	N/A	BH-BB	1	0.44	0.44	9.00
BH-ST	1	D	**2.97**	3.17	BH-IN	1	1.21	**1.21**	7.53

TB, with compiler optimizations enabled, there is also a significant increase in memory traffic due to excessive register spills. The Nvidia profiler reports an increase of 3.4× in the DRAM requests per cycle for *M2S* versus *S2**- on BH-TB. Finally, although *M2S* performs poorly for both BH-TB and BH-SM, the overall impact on performance is not severe because BH-FC dominates execution time for BH.

OpenMP support: The OpenMP 4.0 standard supports the offloading of a parallel region to an accelerator (e.g., a GPU). The OpenMP programming model is appealing because of both its abstraction and portability across architectures. Thus, it helps accelerators reach to a broader set of developers [61]. Currently, there is a co-ordinated effort by many technology companies to add OpenMP 4.0 support for accelerators in LLVM [15], [17], [62]. Programs written in OpenMP 4.0 can suffer from SIMT deadlock. Specifically, current support generates code for a spin lock with back-off delay for *omp_set_lock*(...). Hence, a valid OpenMP program that executes properly on CPUs may not terminate on GPUs due to SIMT deadlocks

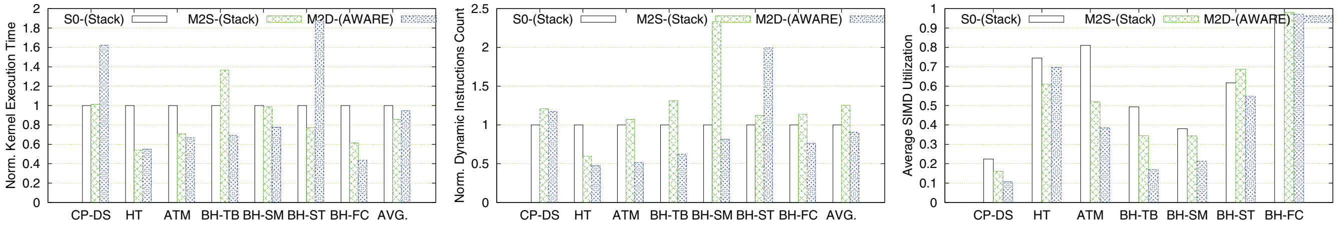
We addressed this by modifying the OpenMP compilation chain to include our detection and elimination passes. Due to the recency of OpenMP support for accelerators, few applications make use of it. Therefore, we modified eight existing OpenMP kernels to enable offloading parallel regions to Nvidia GPUs [48]–[50]. Six of these kernels are for an OpenMP BarnesHut implementation that uses an algorithm similar to the CUDA version except for some GPU specific optimizations. We also modified TL and AM kernels to emulate fine-grained and coarse-grained synchronization.

Table V shows the speed up of four different configurations that run on Tesla K20C GPU compared with using 4 threads on Intel Core i7-4770K CPU. Compilation with current LLVM support for GPUs in OpenMP 4.0 deadlocks in many cases (labeled 'D'). However, with our detection and elimination passes all kernels run to termination with correct outputs maintaining portability between the CPU and the GPU. Numbers in bold show instances where the GPU code achieved a speed up compared to the CPU without performance tuning. For other cases, the developer may choose either not to offload the kernel to a GPU or to performance tune the code for GPUs starting from functionally correct (MIMD) code. Due to false detections in BH-SM and BH-FC, *OMP M2S* performs worse than *OMP M2-* with a 2.2× slowdown for BH-SM. As expected, the GPU performs poorly compared to the CPU with high contention on locks (e.g., TL-CG where 3K threads are competing for one lock) while performing better with fine-grained synchronization (e.g., TL-FG). In AM, a non-blocking check (whether the current element is larger than the current maximum) happens before entering the critical section. This significantly reduces the contention over the critical section that updates the maximum value. Thus, the execution is highly parallel and achieves a large speed up versus the CPU. AM-FG finds multiple maximum values within smaller arrays, thus reducing contention even further. Table V also compares the performance of the OpenMP version of BH with the CUDA version. In most cases, the CUDA version significantly outperforms the OpenMP version due to various GPU-specific optimizations. Efforts to reduce this performance gap are in progress [63], [64] and they are outside the scope of this paper.

2) Adaptive Warp Reconvergence Evaluation: We limit our evaluation of AWARE to CUDA and OpenCL applications because current OpenMP support relies on linking device code at runtime to a compiled OpenMP runtime library [63][11] which is not straightforward to emulate in GPGPU-Sim. We compare executing the *M2D* version on AWARE against executing *S0-* and *M2S* on the stack-based reconvergence baseline. We configure AWARE to enable delayed reconvergence and with the TimeOut mechanism disabled. Figure 10a shows the normalized average execution time for individual kernels. On average, executing the MIMD version on AWARE is on par with executing the *M2S* version on the stack-based reconvergence baseline [12]. However, for some kernels such as BH-TB, *M2D* on AWARE achieves better performance than M2S run on stack-based reconvergence. This is mainly because AWARE does not introduce static instruction overheads. Executing the MIMD version on AWARE often leads to a reduced number of dynamic instructions with the exception of BH-ST (Figure 10).

[11] We only linked OpenMP library calls for synchronization at compile time. Enabling general link time optimizations for OpenMP runtime library requires an engineering effort that is beyond the scope of this paper.

[12] We simulate the PTX ISA on GPGPU-Sim because SASS is not fully supported. Note that PTX uses virtual registers rather than spilling to local memory diminishing some of the advantage of AWARE over SSDE.



(a) Norm. Kernel Execution Time **(b)** Norm. Dynamic Instruction Count **(c)** Average SIMD Utilization

Fig. 10: Evaluation of the Adaptive Warp Reconvergence Mechanism using GPGPU-Sim

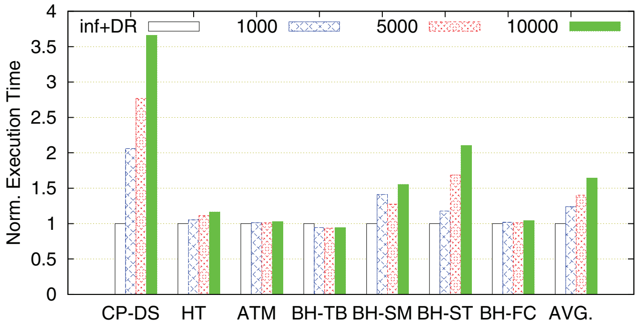


Fig. 11: Sensitivity to the TimeOut value (in cycles)

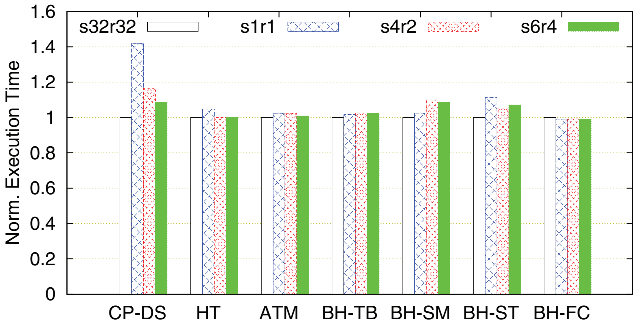


Fig. 12: Effect of AWARE Virtualization on Performance

BH-ST behaves similar to a spin lock; the AWARE FIFO-based scheduling mechanism allows a warp split that did not acquire a lock to attempt acquiring it again even before it is released by the other split. This repeats according to the number of dynamic branch instructions executed along the not-taken path before the lock is released. Manual transformation eliminates this behaviour, and our compiler elimination algorithm also reduces this behaviour.

Figure 11 illustrates AWARE sensitivity to TimeOut values. Kernels that suffer SIMT deadlocks (e.g., BH-ST) favor smaller TimeOut values as they allow blocked threads to make forward progress and release other threads attempting to enter a critical section. For kernels that do not suffer SIMT deadlocks (e.g., BH-SM), smaller TimeOut values reduce SIMD utilization and lower performance. On average, delayed reconvergence with TimeOut disabled achieves the best results. This suggests applying delayed reconvergence whenever a SIMT deadlock is detected and setting TimeOut to a large value such that it is only triggered when there is a high likelihood of an undetected SIMT deadlock.

Figure 12 illustrates the impact of AWARE virtualization on the overall performance. We can see that in the worst case execution time increases by only 16% when we use 4 and 2 physical entries for ST and RT respectively instead of 32 entries; the average is only 5%. Our analysis suggests that the performance overhead is mainly due to the extra traffic caused by the fill and spill requests. For example, using a single entry for both ST and RT with CP-DS kernel increases memory requests by 21% and 15% of this extra traffic miss in L1 cache. Congestion on the miss queues increases MSHR (Miss Status Holding Register) reservation failure rate by a factor of 2.5×. This leads to an increase in the stalls due to structural hazards by 51%. A potential solution that can further reduce the performance overhead is a dedicated small victim cache that is shared among warps to cache recently used spilled entries. A central storage across warps is motivated by the observation of a disparity in the ST and RT occupancy requirement across different warps at a given execution window. In depth study for this solution is left for future work.

VI. Related Work

Branch Divergence on GPUs: There are many research papers that recognize performance implications of branch divergence in GPUs [20], [21], [44], [65]–[69]. However, less attention has been paid to the functional implications. Temporal-SIMT is a hardware proposal that enables more flexible placement of barriers [70] but the use of explicit reconvergence points in Temporal-SIMT can still cause SIMT deadlocks. Dynamic Warp Formation (DWF) [20] enables flexible thread scheduling that could be exploited to avoid SIMT-deadlocks, however, it has higher hardware complexity.

GPU kernel verification: In [9], the authors provide formal semantics for NVIDIA's stack-based reconvergence mechanism and a formal definition for the scheduling *unfairness* problem in the stack-based execution model that may lead to valid programs not being terminated. However, they do not attempt to provide ways to detect or to prevent this problem. There is also some recent work on verification of GPU kernels that focuses on detecting data-races and/or barrier divergence freedom in GPU kernels [28]–[30], [71]. However, none of the verification tools considered the problem of SIMT deadlocks due to conditional loops. Our SIMT deadlock detection is complementary to these efforts.

Code Portability between CPUs and GPUs: There are efforts to make GPU programming accessible through different well-established non-GPU programming languages. These efforts include source-to-source translation [13], [14] as well as developing non-GPU front-ends [15] for language and machine independent optimizing compilers such as LLVM [72]. However, these proposals do not handle SIMT deadlocks. MCUDA [73] is an opposite approach that translates CUDA kernels into conventional multi cores CPU code.

Synchronization on GPUs: In [24], the author attempts to provide a hardware that modifies the behaviour of the reconvergence stack when executing lock or unlock instructions to avoid possible deadlocks. However, the proposed solution fails if the locking happens in diverged code [24]. It is limited to mutexes and it applies only in very restricted cases. Both hardware and software transactional memory support has been proposed [25], [26] to enable easier synchronization on GPUs. In [74], the authors propose hardware support for a blocking synchronization mechanism on GPGPU. Using their proposed synchronization APIs, SIMT deadlocks could mitigated in restricted cases (similar to [24]). In [75], Li et al. propose a fine-grained inter-thread synchronization scheme that uses GPUs shared memory. However, it is left to programmers to use their locking scheme carefully to avoid SIMT-induced deadlocks [75]. Recently, software lock stealing and virtualization techniques were proposed to avoid circular locking among GPU threads and to reduce the memory cost of fine-grain locks [76]. As acknowledged in [76], one of their limitations is that “locks are not allowed to be acquired in a loop way” due to deadlocks (i.e., SIMT deadlocks using our terminology). Our work adopts a distinct approach that aims to change the SIMT execution model to be functionally compatible with the more intuitive MIMD execution model. Fully achieving this goal saves programmers the trouble of adapting MIMD code to a target specific SIMT implementation.

VII. Conclusion

In this paper, we argue that the compiler and the hardware should be capable of abstracting away SIMT implementation nuances. To this end, we propose techniques that enable the execution of arbitrary MIMD code with inter-thread synchronization on SIMT/SIMD-like hardware. We propose a compile-time algorithm that detects potential SIMT deadlocks with a false detection rate of 4%-5%. Next, we proposed a CFG transformation that works around the deadlock problem on the current hardware. The automated transformation has an average overhead of 8.2%-10.9% compared to the manual transformation with the same level of compiler optimization. Finally, we propose microarchitectural modifications to the SIMT branch handling unit to avoid key limitations in the compiler-only approach. Both techniques can be disabled if a programmer is interested in low level optimizations that are dependent on predefined reconvergence locations. Future directions of this work include designing tools that leverage runtime information to detect SIMT deadlocks, compilers that restrict valid transformations according to the SIMT behaviour, and efficient hardware-based MIMD-compatible reconvergence mechanisms.

Acknowledgments

We would like to thank Mieszko Lis, Matei Ripeanu, Hassan Halawa, Tayler Hetherington, Andrew Boktor, Shadi Assadikhomami, Timothy G. Rogers, and the reviewers for their insightful feedback. This research was funded in part by a Four Year Doctoral Fellowship from University of British Columbia and the Natural Sciences and Engineering Research Council of Canada.

References

[1] NVIDIA, CUDA, “NVIDIA CUDA Programming Guide,” 2011.

[2] Intel Corporation, “The ISPC Parallel Execution Model,” 2016.

[3] AMD, “Accelerated Parallel Processing: OpenCL Programming Guide,” 2013.

[4] A. Levinthal and T. Porter, “Chap — A SIMD Graphics Processor,” *Proc. ACM Conf. on Comp. Grap. and Interactive Tech. (SIGGRAPH)*, 1984.

[5] B. Coon and J. Lindholm, “System and method for managing divergent threads in a simd architecture,” 2008. US Patent 7,353,369.

[6] B. Beylin and R. S. Glanville, “Insertion of multithreaded execution synchronization points in a software program,” 2013. US Patent 8,381,203.

[7] AMD Corporation, “Southern Islands Series Instruction Set Architecture,” 2012.

[8] M. HOUSTON, B. Gaster, L. HOWES, M. Mantor, and D. Behr, “Method and System for Synchronization of Workitems with Divergent Control Flow,” 2013. WO Patent App. PCT/US2013/043,394.

[9] A. Habermaier and A. Knapp, “On the Correctness of the SIMT Execution Model of GPUs,” in *Programming Languages and Systems*, pp. 316–335, Springer, 2012.

[10] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, “Accelerating CUDA Graph Algorithms at Maximum Warp,” in *Proc. ACM Symp. on Prin. and Prac. of Par. Prog. (PPoPP)*, pp. 267–276, 2011.

[11] M. Burtscher, R. Nasre, and K. Pingali, “A Quantitative Study of Irregular Programs on GPUs,” in *Proc. IEEE Symp. on Workload Characterization (IISWC)*, 2012.

[12] D. Merrill, M. Garland, and A. Grimshaw, “Scalable GPU Graph Traversal,” in *Proc. ACM Symp. on Prin. and Prac. of Par. Prog. (PPoPP)*, pp. 117–128, 2012.

[13] S. Lee, S.-J. Min, and R. Eigenmann, “OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization,” in *Proc. ACM Symp. on Prin. and Prac. of Par. Prog. (PPoPP)*, pp. 101–110, 2009.

[14] G. Noaje, C. Jaillet, and M. Krajecki, “Source-to-source Code Translator: OpenMP C to CUDA,” in *IEEE Int’l Conf. on High Performance Computing and Communications (HPCC)*, 2011.

[15] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O’Brien, Z. Sura, A. C. Jacob, T. Chen, and O. Sallenave, “Coordinating GPU Threads for OpenMP 4.0 in LLVM,” in *Proc. LLVM Compiler Infrastructure in HPC*, 2014.

[16] S. Antao, C. Bertolli, A. Bokhanko, A. Eichenberger, H. Finkel, S. Ostanevich, E. Stotzer, and G. Zhang, “OpenMP Offload Infrastructure in LLVM,” tech. rep.

[17] OpenMP Clang Frontend, “OpenMP Clang Frontend Documentation.” https://github.com/clang-omp, 2015.

[18] X. Tian and B. R. de Supins, “Explicit Vector Programming with OpenMP 4.0 SIMD Extension,” *Primeur Magazine 2014*, 2014.

[19] A. ElTantawy, “SSDE and AWARE codes.” https://github.com/ElTantawy/mimd_to_simt/, 2016.

[20] W. Fung, I. Sham, G. Yuan, and T. Aamodt, “Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow,” in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, pp. 407–420, 2007.

[21] J. Meng, D. Tarjan, and K. Skadron, “Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance,” in *Proc. IEEE/ACM Symp. on Computer Architecture (ISCA)*, pp. 235–246, 2010.

[22] NVIDIA Forums, “atomicCAS does NOT seem to work.” http://forums.nvidia.com/index.php?showtopic=98444, 2009.

[23] NVIDIA Forums, “atomic locks.” https://devtalk.nvidia.com/default/topic/512038/atomic-locks/, 2012.

[24] A. Ramamurthy, “Towards Scalar Synchronization in SIMT Architectures,” Master’s thesis, The University of British Columbia, 2011.

[25] W. W. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware Transactional Memory for GPU Architectures," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, pp. 296–307, 2011.

[26] Y. Xu, R. Wang, N. Goswami, T. Li, L. Gao, and D. Qian, "Software Transactional Memory for GPU Architectures," in *Proc. IEEE/ACM Symp. on Code Generation and Optimization (CGO)*, p. 1, 2014.

[27] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU Microarchitecture Through Microbenchmarking," in *Proc. IEEE Symp. on Perf. Analysis of Systems and Software (ISPASS)*, pp. 235–246, 2010.

[28] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "GPUVerify: a Verifier for GPU Kernels," in *Proc. ACM Int'l Conf. on Object oriented programming systems languages and applications*, pp. 113–132, 2012.

[29] 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.

[30] R. Sharma, M. Bauer, and A. Aiken, "Verification of Producer-Consumer Synchronization in GPU Programs," in *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI)*, pp. 88–98, 2015.

[31] "How does the OpenACC API relate to the OpenMP API?." OpenACC.org.

[32] LLVM Compiler, "LLVN 3.6 Release Information." http://llvm.org/releases/3.6.0/, 2015.

[33] M. Villmow, "AMD OpenCL Compiler." LLVM Developers Conference, 2010. Presentation.

[34] NVIDIA, "CUDA LLVM Compiler." https://developer.nvidia.com/cuda-llvm-compiler, 2015.

[35] Intel Developer Zone, "Weird behaviour of atomic functions." https://software.intel.com/en-us/forums/opencl/topic/278350, 2012.

[36] NVIDIA Forums, "GLSL Spinlock." https://devtalk.nvidia.com/default/topic/768115/opengl/glsl-spinlock/, 2014.

[37] M. Burtscher and K. Pingali, "An Efficient CUDA Implementation of the Tree-based Barnes Hut n-Body Algorithm," *GPU computing Gems Emerald edition*, 2011.

[38] A. ElTantawy and T. M. Aamodt, "Correctness Discussion of a SIMT-induced Deadlock Elimination Algorithm," tech. rep., University of British Columbia, 2016.

[39] NVIDIA, "CUDA Binary Utilities," *http://docs.nvidia.com/cuda/cuda-binary-utilities/*, 2015.

[40] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," in *PLDI*, 1988.

[41] M. Burke and R. Cytron, "Interprocedural Dependence Analysis and Parallelization," in *Proc. ACM SIGPLAN Symp. on Compiler Construction*, 1986.

[42] U. Hölzle, C. Chambers, and D. Ungar, "Debugging Optimized Code with Dynamic Deoptimization," in *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI)*, 1992.

[43] J. Hennessy, "Symbolic Debugging of Optimized Code," *ACM Trans. on Prog. Lang. and Sys.(TOPLAS)*, vol. 4, no. 3, pp. 323–344, 1982.

[44] A. ElTantawy, J. W. Ma, M. O'Connor, and T. M. Aamodt, "A Scalable Multi-Path Microarchitecture for Efficient GPU Control Flow," in *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, 2014.

[45] B. W. Coon, P. C. Mills, S. F. Oberman, and M. Y. Siu., "Tracking Register Usage during Multithreaded Processing Using a Scoreboard having Separate Memory Regions and Storing Sequential Register Size Indicators. US Patent 7,434,032," 2008.

[46] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, pp. 72–83, 2012.

[47] A. Brownsword, "Cloth in OpenCL," tech. rep., Khronos Group, 2009.

[48] J. Coplin and M. Burtscher, "Effects of Source-Code Optimizations on GPU Performance and Energy Consumption," in *Proc. ACM Workshop on General Purpose Processing on Graphics Processing Units*, 2015.

[49] J. M. Bull, "Measuring synchronisation and scheduling overheads in openmp," in *Proc. European Workshop on OpenMP*, vol. 8, p. 49, 1999.

[50] Microsoft. https://msdn.microsoft.com/en-us/library/b38674ky.aspx, 2016.

[51] LLVM Compiler, "LLVM Alias Analysis Infrastructure." http://llvm.org/docs/AliasAnalysis.html, 2015.

[52] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee, "Identifying loops using dj graphs," *ACM Trans. on Prog. Lang. and Sys. (TOPLAS)*, 1996.

[53] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Proc. IEEE Symp. on Perf. Analysis of Systems and Software (ISPASS)*, pp. 163–174, 2009.

[54] T. M. Aamodt et al., *GPGPU-Sim 3.x Manual*. University of British Columbia, 2013.

[55] LLVM Compiler, "Clang Front End." http://clang.llvm.org/, 2015.

[56] LLVM Compiler, "LIBCLC Library." http://libclc.llvm.org/, 2015.

[57] Dmitry Mikushin, "CUDA to LLVM-IR." https://github.com/apc-llc/nvcc-llvm-ir, 2015.

[58] NVIDIA, "LibNVVM Library." http://docs.nvidia.com/cuda/libnvvm-api/, 2015.

[59] NVIDIA, "CUDA SDK 3.2," September 2013.

[60] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proc. IEEE Symp. on Workload Characterization (IISWC)*, pp. 44–54, 2009.

[61] Jeff Larkin, NVIDIA, "OpenMP and NVIDIA." http://openmp.org/sc13/SC13_OpenMP_and_NVIDIA.pdf, 2013.

[62] Michael Wong, Alexey Bataev, "OpenMP GPU/Accelerators Coming of Age in Clang." http://llvm.org/devmtg/2015-10/slides/WongBataev-OpenMPGPUAcceleratorsComingOfAgeInClang.pdf, 2015.

[63] C. Bertolli, S. F. Antao, G.-T. Bercea, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, *et al.*, "Integrating GPU support for OpenMP offloading directives into Clang," in *Proc. ACM Int'l Workshop on the LLVM Compiler Infrastructure in HPC*, 2015.

[64] G.-T. Bercea, C. Bertolli, S. F. Antao, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, *et al.*, "Performance Analysis of OpenMp on a GPU Using a Coral Proxy Application," in *Proc. ACM Int'l Workshop on Perf. Modeling, Benchmarking, and Simulation of High Perf. Computing Sys.*, 2015.

[65] M. Rhu and M. Erez, "The Dual-Path Execution Model for Efficient GPU Control Flow," in *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, pp. 235–246, 2013.

[66] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili, "SIMD Re-convergence at Thread Frontiers," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, pp. 477–488, 2011.

[67] W. W. L. Fung and T. M. Aamodt, "Thread Block Compaction for Efficient SIMT Control Flow," in *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, pp. 25–36, 2011.

[68] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, pp. 308–317, 2011.

[69] M. Rhu and M. Erez, "CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures," in *Proc. IEEE/ACM Symp. on Computer Architecture (ISCA)*, pp. 61–71, 2012.

[70] Y. Lee, R. Krashinsky, V. Grover, S. Keckler, and K. Asanovic, "Convergence and Scalarization for Data-Parallel Architectures," in *Proc. IEEE/ACM Symp. on Code Generation and Optimization (CGO)*, pp. 1–11, 2013.

[71] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal, "GRace: a Low-overhead Mechanism for Detecting Data Races in GPU Programs," in *ACM SIGPLAN Notices*, vol. 46, pp. 135–146, ACM, 2011.

[72] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *Proc. IEEE/ACM Symp. on Code Generation and Optimization (CGO)*, 2004.

[73] J. A. Stratton, S. S. Stone, and W. H. Wen-mei, "MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs," in *Languages and Compilers for Parallel Computing*, Springer, 2008.

[74] A. Yilmazer and D. Kaeli, "HQL: A Scalable Synchronization Mechanism for GPUs," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, 2013.

[75] A. Li, G.-J. van den Braak, H. Corporaal, and A. Kumar, "Fine-grained Synchronizations and Dataflow Programming on GPUs," 2015.

[76] Y. Xu, L. Gao, R. Wang, Z. Luan, W. Wu, and D. Qian, "Lock-based Synchronization for GPU Architectures," in *Proc. Int'l Conf. on Computing Frontiers*, 2016.