Correctness Discussion of a SIMT-induced Deadlock Elimination Algorithm

Draft v1.0 Updated 13 August 2016

Ahmed ElTantawy, Tor M. Aamodt
University of British Columbia

1. Introduction

Current GPUs adapt the Single Instruction Multiple Threads (SIMT) execution model. In SIMT model, logically independent threads are divided into groups (or warps) that operate in lockstep in a Single Instruction Multiple Data (SIMD) fashion. To enable divergence within a warp, current SIMT implementations serialize execution of divergent threads while forcing reconvergence at postdominator points. This imposes restrictions to thread scheduling that may conflict with the forward progress requirements of divergent, yet communicating threads. This makes a Multiple Input Multiple Data (MIMD) synchronization between divergent threads not always possible on current SIMT implementations. Therefore, programmers currently have to consider these thread scheduling constraints when synchronizing divergent threads on GPUs. In particular, with no explicit control on thread scheduling, they need to restructure their control flow graphs to force the hardware to schedule divergent synchronizing threads in a manner that guarantees their forward progress. In our paper "MIMD Synchronization on SIMT Architectures", we propose compiler algorithms that enables MIMD synchronization on SIMT hardware without the need of programmers' intervention [4]. The purpose of this technical report is to provide correctness proofs for the elimination algorithm proposed in this paper. In this technical report, we start by providing some definition for the terms used in this report. Also, we state the baseline assumptions that are assumed to be true in both the parallel kernels and the SIMT machine under study. Next, we present our proof outlines.

2. Definitions

This section defines basic terms we use in both the paper and this report.

Parallel kernel: a kernel that are executed in parallel by multiple threads.

MIMD Machine: a machine that guarantees loose fairness in thread scheduling (i.e., there is no thread (or group of threads) that may not be scheduled indefinitely.).

SIMT Machine: a machine that divides threads into one or more groups (or warps) that operate in lockstep when there is no divergence. We assume the warp scheduling on the SIMT machine guarantees loose fairness (i.e., there is no warp that may not be scheduled indefinitely), however, there are constraints on scheduling divergent threads within each warp. In particular, upon divergence, the machine serializes divergent threads such that: if a warp or (a warp *split* [11]) **W** encounters a divergent branch $\mathbf{BR_{T,NT\mapsto R}}$, and diverges into two splits; $\mathbf{W_{P_{T\mapsto R}}}$ and $\mathbf{W_{P_{NT\mapsto R}}}$, where $\mathbf{W_{P_{T\mapsto R}}}$ is the split that contains threads that diverges to the taken path, and $\mathbf{W_{P_{NT\mapsto R}}}$ is the split that contains threads that diverges to the not-taken path. Both splits are required to reconverge (i.e., restore the lockstep synchronous execution) at the $\mathbf{BR_{T,NT\mapsto R}}$ reconvergence point **R**. The following scheduling constraints are applied:

Constraint-1 Warp split $\mathbf{W_{P_{NT\mapsto R}}}$ starts executing the not-taken path **iff** $\mathbf{W_{P_{T\mapsto R}}}$ reaches **R** (or vice versa according to the order in which divergent paths are pushed into the stack[1]). This constraint is a side effect of the stack forcing divergent paths to be serialized up to their reconvergence point.

Constraint-2 When warp split $\mathbf{W_{P_{T\mapsto R}}}$ reaches **R**, it remains blocked until warp split $\mathbf{W_{P_{NT\mapsto R}}}$ reaches **R** as well (or vice versa). This constraint is a side effect of forcing reconvergence for diverged splits.

We briefly refer to these two constraints as the *reconvergence scheduling constraints*.

SIMT-induced deadlock: is a situation in which at least one thread in a diverged warp is indefinitely blocked (i.e., not scheduled) due to cyclic dependencies between the requirements of the forward progress of diverged threads within this warp to their reconvergence point, and at least one of these dependencies involves a reconvergence scheduling constraint.

3. Baseline Assumptions

This section states our baseline assumptions for the parallel kernel under study. We assume a single kernel function **K** with no function calls and with a single exit. We assume that **K** is guaranteed to terminate (i.e., is a deadlock and livelock free) if executed on *any* MIMD machine. We also assume that **K** is barrier divergence-free [1] (i.e., for all barriers within the kernel, if a barrier is encountered by a warp, the execution predicate evaluates true across all threads within this warp [2].). Finally, for purposes of clarity and simplicity, we assume that the kernel has structured CFG. Finally, we assume that there are only two instructions in the Instruction Set Architecture (ISA) that may conflict with the forward progress of threads after being executed; barriers and backward branches. For the sake of brevity, we refer to all memory spaces capable

[1]Nvidia's GPUs executes the taken path first [3].

[2]In current GPU programming models, barriers are used to synchronize all threads within a number of warps. However, within the scope of this paper, we are only concerned about barriers synchronizing all threads within a warp.

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

-**BB(I):** basicblock which contains instruction (i.e., $\mathbf{I} \in \mathbf{BB(I)}$).
-$\mathbf{P_{BB1 \mapsto BB2}}$: union set of basicblocks in execution paths that connect **BB1** to **BB2**.
-**IPDom(I)** : immediate postdominator of an 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 basicblocks at the branch targets $\mathbf{BB_T}$ and $\mathbf{BB_{NT}}$ (i.e., the common postdominator that strictly postdominate both $\mathbf{BB_T}$ and $\mathbf{BB_{NT}}$ and does not postdominate any of their other common postdominators.).
-**IPDom(arg1,arg2)** is the immediate common postdominator for **arg1** and **arg2**; **arg1** and **arg2** could be either basicblocks or instructions.

-**LSet:** the set of loops in the kernel, where $\forall$ $\mathbf{L} \in \mathbf{LSet}$:
-**BBs(L):** the set of basicblocks within loop **L** body.
-**ExitConds(L):** the set of branch Instructions at loop **L** exits.
-**Exits(L):** the set of basicblocks outside the loop that are immediate successors to a basicblock in the loop.
-**Latch(L):** loop **L** backward edge, **Latch(L).src** and **Latch(L).dst** are the edge source and destination basicblocks respectively.
-Basicblock **BB** is *reachable* from loop **L**, iff there is a non-null path(s) connecting the reconvergence point of **Exits(L)** with basicblock **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

```
for each loop $L \in LSet$ do
    $Shrd_{Reads}(L) = \emptyset$, $Shrd_{Writes}(L) = \emptyset$, $Redef_{Writes}(L) = \emptyset$
    for each instruction I, where $BB(I) \in BBs(L)$ do
        if I is a shared memory read $\wedge$ ExitConds(L) depends on I then
            $Shrd_{Reads} = Shrd_{Reads} \cup I$
        end if
    end for
    for each instruction I do
        if BB(I) is parallel to or reachable from L then
            if I is a shared memory write then
                $Shrd_{Writes}(L) = Shrd_{Writes}(L) \cup I$
            end if
        end if
    end for
    for each pair $(I_R, I_W)$, where $I_R \in Shrd_{Reads}(L)$ and $I_W \in Shrd_{Writes}(L)$ do
        if $I_W$ does/may alias with $I_R$ then
            $Redef_{Writes}(L) = Redef_{Writes}(L) \cup I_W$
        end if
    end for
    if $Redef_{Writes}(L) \neq \emptyset$ then Label L as a potential SIMT-induced deadlock.
    end if
end for
```

of holding synchronization variables as *shared* memory (i.e., including both global and shared memory using CUDA terminology). Next, we list the main algorithms and definitions used in the paper and referenced to in this technical report. Please refer to the paper for full explanation to the Algorithms.

4. Formal Proof

This section presents the outlines of a proof that the transformation **T** provided by Algorithm 3 is *correct*. Next theorem defines correctness.

Theorem 1 Let P_i be a program that is an input to transformation **T** and $P_o = T(P_i)$ be a program that results from applying **T** on P_i, then any observable behaviour of P_o on a SIMT machine is also an observable behaviour of P_i on a MIMD machine (i.e., a machine that guarantees loose fairness

Algorithm 2 Safe Reconvergence Points

```
$SafePDom(L) = IPDom(Exits(L))\ \forall\ L \in LSet$
for each loop $L \in LSet$ do
    for each $I_W \in Redef_{Writes}(L)$ do
        $SafePDom(L) = IPDom(SafePDom(L), I_W)$
        if $BB(I_W)$ is reachable from L then
            for each branch instruction $I_{BR} \in ReachBrSet(L, BB(I_W))$ do
                $SafePDom(L) = IPDom(SafePDom(L), I_{BR})$
            end for
        end if
        if $BB(I_W)$ is parallel to L then
            for each branch instruction $I_{BR} \in ParaBrSet(L, BB(I_W))$ do
                $SafePDom(L) = IPDom(SafePDom(L), I_{BR})$
            end for
        end if
    end for
end for
resolve_SafePDom_conflicts()
```

Algorithm 3 SIMT-Induced Deadlock Elimination

```
$SwitchBBs = \emptyset$
for each loop $L \in LSet$ do
    if L causes potential SIMT-induced deadlock then
        if $SafePDom(L) \notin SwitchBBs$ then
            $SwitchBBs = SwitchBBs \cup SafePDom(L)$
            if SafePDom(L) is the first instruction of a basicblock BB then
                Add a new basicblock $BB_S$ before the BB.
                Incoming edges to $BB_S$ are from BB predecessors.
                Outgoing edge from $BB_S$ is to BB.
            else
                Split BB into two blocks $BB_A$ and $BB_B$.
                $BB_A$ contains instructions up to but not including SafePDom(L).
                $BB_B$ contains remaining instructions including SafePDom(L).
                $BB_S$ inserted in the middle, $BB_A$ as predecessor, $BB_B$ as successor.
            end if
            Insert a PHI node to compute a value cond in $BB_S$, where:
                for each predecessor Pred to $BB_S$
                    cond.addIncomingEdge(0,Pred)
                end for
            Insert Switch branch swInst on the value cond at the end of $BB_S$, where:
                swInst.addDefaultTarget($BB_S$ successor)
        end if
        $BB_S$ is the basicblock immediately preceding SafePDom(L)
        Update PHI node cond in $BB_S$ as follows:
        cond.addIncomingEdge(UniqueVal,Latch(L).src) -unique to this edge
        Update Switch branh swInst at the end of $BB_S$ as follows:
        swInst.addCase(UniqueVal,Latch(L).dst)
        Set $Latch(L).dst = BB_S$.
    end if
end for
```

in thread scheduling)[3].

4.1. Proof Sketch

This definition of correctness has been used as a correctness criteria for both program transformations [6, 8–10] and execution models [2, 5]. We define observable behavior as the shared memory state, the return value of the program -if any- and the termination properties. We divide the proof of Theorem 1 into two steps that need to be proven:
1. Any observable behaviour of P_o is also an observable behaviour of P_i when both are executed on a MIMD machine.
2. Any observable behaviour of P_o on a SIMT machine is also an observable behaviour of P_o on a MIMD machine.

A. Semantics across transformations: To prove this

[3]We assume that in both SIMT and MIMD machines, the number of launched threads do not exceed the number of threads that can reside simultaneously on the hardware.

part we use the same methodology used in TRANS [7, 10]. A program P is modeled as a labeled directed flow graph G_P formed from the program nodes. Each node represents an instruction. The execution trace of the program is modeled by a sequence of *state transitions*, where each state is represented by the current instruction(s), shared memory state and individual thread(s) local state. In a single thread context, the transformation correctness is proven by finding a *simulation* relation that can express how the output program simulates the same behaviour of the input program in a number of finite transitions assuming we start from the same initial state. This simulation relation is found by observation and then proven to always hold true using induction over the number of the length of the execution trace of P_i. This single-thread simulation relation is then lifted to parallel execution using the principles provided in [9, 10].

B. Semantics across execution models: In this part we rely on prior work [2, 5] that proves that the execution of an arbitrary program P on a SIMT machine can be simulated by some schedule of the traditional interleaved thread execution (i.e., MIMD execution) of the same program. Thus, terminating kernels on a SIMT machine produce a valid observable behaviour compared with MIMD execution. However, it is still possible that a program that always (i.e., under any loosely fair scheduling) terminate on MIMD not to terminate on SIMT. Therefore, it is sufficient for us to prove that P_o, an output of $\mathbf{T}$, always terminate on a SIMT machine if P_o (or equivalently P_i) always terminate on a MIMD machine (i.e., with any arbitrary loosely fair schedule). Termination is trivially proven if we can prove that all threads executing any arbitrary branch in P_o eventually reach to the branch reconvergence point. The essence of the proof is that we find that the hypothesis that threads divergent at a certain branch in P_o may not eventually reach their reconvergence point contradicts with either one of the following premises: (1) that P_o terminates under any arbitrary fair scheduling, (2) the valuation of the exit conditions in any loop in P_o is independent of the valuation of paths parallel or reachable from the loop (see the definitions in Listing 1). Accordingly, we conclude that the P_o should terminate on SIMT.

4.2. Detailed Proof

A. Semantics across transformations: To prove this part we use the same methodology used in TRANS [7, 10].

Let a program P has form:

$$\begin{array}{rcl}
P & : & \text{Entry};\ I_1;\ I_2;\ \cdots;\ I_{m-1};\ \mathit{Exit} \\
\text{Inst} & \ni & I ::= \text{nop} \mid X := E \mid M \mid \text{if } E \text{ goto } n \\
\text{M. Inst} & \ni & M ::= X := m(E) \mid m(E) := X \\
\text{Mem.} & \ni & M ::= \mathit{shared\ memory} \\
\text{Expr} & \ni & E ::= X \mid O(E) \mid C \\
\text{Op} & \ni & O ::= \mathit{various\ unspecified\ operators} \\
\text{Var} & \ni & X ::= x \mid y \mid z \mid \ldots \\
\text{Const} & \ni & C ::= \mathit{bool} \mid \mathit{integer} \mid \mathit{float} \mid \ldots\ \mathit{value} \\
\text{Label} & \ni & n ::= 0 \mid 1 \mid \ldots \mid m
\end{array}$$

Program P can be represented as a labeled directed flow graph G_P formed from the program nodes ($N = \{0, 1, ..., m\}$. Each node n has at least one sequential edge to another node $seq(n)$; and possibly another branch edge to a node $brn(n)$. Thus, an edge in the flow graph is defined by two nodes and an edge type. I_n refers to the instruction labeled by node n. Hence, G_P can be represented by the tuple $\langle N, E \subseteq N \times N \times EdgeType, I : N \mapsto Instr\rangle$. Finally, a valuation function σ is used to map a pattern of meta variables to a (sub)object in G (i.e., node, edge, instr, expr, ..).

In concurrent MIMD execution, a set of thread traverse through G_P such that at any time the execution state of P can be represented by a tuple $(nodes, states, m)$; where $nodes$ is a vector of the labels representing the current location of each thread in G_P, $states$ is a vector of the local state of each thread, and m is the current state of the externally observable shared memory. Thus, the execution trace of a program can be modeled by *state transitions:* $st_i \rightarrow st_{i+1}$, where $st_i = (nodes, states, m)_i$. The state of an individual thread is represented as $st(t) = (n, s, m) = (nodes_t, states_t, m)$. A state transition happens when a thread executes the instruction labeled by its current node. Next, we present the state transition relations for an instructions I_n at node n:

$$\begin{array}{rcl}
(s,n,m) & \rightarrow & (s, seq(n), m);\ \ I_n = nop \\
(s,n,m) & \rightarrow & (s \oplus x \mapsto eval(e,s)), seq(n), m);\ \ I_n = x := e \\
(s,n,m) & \rightarrow & (s \oplus x \mapsto m(eval(e,s)), seq(n), m);\ \ I_n = x := m(e) \\
(s,n,m) & \rightarrow & (s, seq(n), m \oplus m(eval(e,s));\ \ I_n = m(e) := x \\
(s,n,m) & \rightarrow & (s, seq(n), m);\ \ I_n = if\ e\ goto\ brn(n),\ eval(e,s) = F \\
(s,n,m) & \rightarrow & (s, brn(n), m);\ \ I_n = if\ e\ goto\ brn(n),\ eval(e,s) = T
\end{array}$$

Next, we describe the transformation in Algorithm 3 using the TRANS transformation language. For clarity, the transformation is simplified to handle transformation of a single structured loop (L_0). However, both the algorithm description

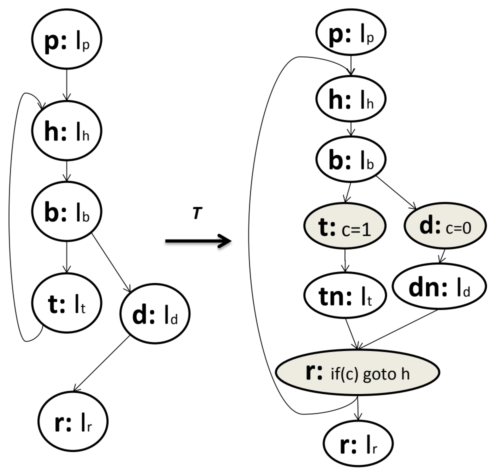

Figure 1: Visualization of **T**

and the proof can be extended in a straightforward manner to handle the transformation of multiple (un)structured loops. In TRANS, a transformation is specified as a set of *actions* performed on a flow graph G under certain *conditions*. We describe our transformation T as follows:

$$\lceil replace\ r \text{ with } r \mapsto \sigma(nop);\ rn \mapsto I_r \rceil(\sigma, G) \tag{a1}$$

$$\lceil move\_edge(t,h,r) \rceil(\sigma, G) \tag{a2}$$

$$\lceil add\_cond\_edge(r,h,t) \rceil(\sigma, G) = \tag{a3}$$

$$\lceil replace\ d \text{ with } d \mapsto \sigma(c := 0);\ dn \mapsto I_d \rceil(\sigma, G) \tag{a3.1}$$

$$if\ d \models EX(node(r)) \wedge d \neq t \wedge A\neg E(true\ U\ use(c))$$

$$\lceil replace\ t \text{ with } t \mapsto \sigma(c := 1);\ tn \mapsto I_t \rceil(\sigma, G) \tag{a3.2}$$

$$if\ A\neg E(true\ U\ use(c))$$

$$\lceil replace\ s \text{ with } s \mapsto \sigma(if(c)\ goto\ h) \rceil(\sigma, G) \tag{a3.3}$$

if

$$loop(p,h,b,t) \equiv L_0 \tag{c1}$$

$$r \models SafePDOM(L_0) \tag{c2}$$

A program P is modeled as a labeled directed flow graph G_P formed from the program nodes. Each node n has at least one sequential edge to another node $seq(n)$; and possibly another branch edge to a node $brn(n)$. The execution trace of a program can be modeled by a sequence of *state transitions:* $st_i \rightarrow st_{i+1}$, where $st_i = (nodes, states, m)_i$. The state of an individual thread is represented as $st(t) = (n,s,m)$ where n is a label to current location of t, s represents the local state of t and m represents shared memory state. State transitions are triggered by threads executing instructions. Figure 1 visualize the application of **T** on a loop $L_0 = loop(p,h,b,t)$ using TRANS notation of nodes. We omit formal description of **T** with TRANS for space limitations. Next, we prove the correctness of transformation **T** when applies to L_0 assuming a MIMD execution model.

Single-Thread Equivalence: We prove that there exists a relation R that relates the execution states of the original program with that of the transformed program for a thread t with the assumption that other threads $u \neq t$ remain at the same initial execution state. We then prove that if this relation R holds then then it implies that the two programs have the same observable behavior as defined earlier.

The set of actions are performed on a loop L_0 with preheader p, header h, break (exit) b and tail t. Node r is the $SafePDOM(L_0)$. Actiona1 adds a *nop* instruction at node r. The backward edge from the loop tail t to the loop header h is moved from h to r by Action a2. Finally, Action a3 adds an edge from r to h that would be only taken if node r is entered from t. This replaces the *nop* instruction in r with a conditional branch instruction that diverges to h if a variable c had a value of '0' (i.e., if the thread is reaching r from any predecessor d other than t) and to h if the c had a value of '1' (i.e., if the thread is reaching r from t). Thus, node r acts as a switch that redirects the flow of the execution to an output node according to the input node. Conditions on actions a3.1 and a3.2 assures that c is not used by any node that follows t or d other than the new added branch instruction at r. According to these actions the following properties are satisfied:

$$I_n)_{P_i} = I_n)_{P_o},\ seq(n)_{P_i} = seq(n)_{P_o}\ for\ n \notin \{d,r,t\} \tag{p1}$$

$$I_d)_{P_o} = \sigma(c := 0),\ seq(d)_{P_o} = dn \tag{p2}$$

$$I_{dn})_{P_o} = I_d)_{P_i},\ seq(dn)_{P_o} = seq(d)_{P_i} \tag{p3}$$

$$I_r)_{P_o} = \sigma(if(c)\ goto\ h),\ seq(r)_{P_i} = rn \tag{p4}$$

$$I_{rn})_{P_o} = I_r)_{P_i},\ seq(rn)_{P_o} = seq(r)_{P_i} \tag{p5}$$

$$I_t)_{P_o} = \sigma(c := 1),\ seq(t)_{P_o} = tn,\ I_{tn})_{P_o} = I_t)_{P_i} \tag{p6}$$

$$seq(t)_{P_i} = h,\ seq(tn)_{P_o} = r \tag{p7}$$

Assume the following state transitions for a thread t:

$$st_0(t) \rightarrow st_1(t) \rightarrow \ldots\ st_i(t) = (n,s,m)\ \ldots \rightarrow st_k(t) \text{ from } P_i\ and$$
$$st'_0(t) \rightarrow s'_1(t) \rightarrow \ldots\ st'_i(t) = (n',s',m')\ \ldots \rightarrow st'_l(t) \text{ from } P_o$$

also assume that for all other threads $u \neq t$ $st_0(u) = st'_0(u)$ Then, the following relation R holds

$$R1.\ s'_{l_{except\ \sigma(c)}} = s_k \quad R2.\ m'_l = m_k \quad R3.\ I'_l = I_k$$

$$R4.\ node(n'_l) = \begin{cases} node(n_k) & node(n_k) \notin \{d,r,h\} \\ dn & node(n_k) = d \\ rn & node(n_k) = r \\ tn & node(n_k) = t \end{cases}$$

; for $l = k+u+v+w+y$ where:
$u =$ no. of nodes n_i where $i \leq k$ such that $node(n_i) = d$,
$v =$ no. of nodes n_i where $i \leq k$ such that $node(n_i) = r$,
$w =$ no. of nodes n_i where $i \leq k$ such that $node(n_i) = t$,
$y =$ no. of transitions $node(n_i) \rightarrow node(n_{i+1}) = t \rightarrow h$ where $i < k$;
R simply states that P_o simulates the behavior of P_i but in potentially more execution state transitions according to the specific execution path that was taken [4]. These extra transitions account for the execution of the added nodes during the

[4] R in this case is called a *simulation* relation.

transformation **T**. The local state of a thread executing P_o at l may only be different by the new added variable c that control the branch at r. We prove R by induction over k.

Proof Logic: assuming that the relation holds for k, then we consider one transition from k to $k+1$. We find that according to R, P_o should simulate the behaviour of P_i in either one or two transitions from l. This is determined according to $node(n_{k_n})$ and $node(n_{k_n-1})$. Then, we proceed by proving that this indeed holds for all possible transitions from k to $k+1$. Given that relation R holds, it is trivial to prove program equivalence. First, if an execution state transition terminates for P_i at k then it will terminate for P_o at l=k+u+v+w+y. Further, $I_k = I_l = ret(e)$, $s'_{l_{except\ \sigma(c)}} = s$ and e does not use $\sigma(c)$, then return values will be the same.

Base Case is trivially . For the same input and same initial state R1,R2 and R3.2 holds.

Step Case Assume that R is true for k. Prove R is true for $k_n = k+1$.

Case-1: $\sigma(n_{k_n}) \notin \{d,r,t\}$ and $\sigma(n_k) \to \sigma(n_{k+1}) \neq t \to h$ Given the condition on Case-1, $u_n = u$, $v_n = v$, $w_n = y$, and $l_n = l+1$ (i.e., execution of P_o should simulate the execution of P_i at k_n in a single step from l). From R1 and R3 at k and the state transition relations, it is trivial to prove that that $s'_{l_n}/\sigma(c)$ = $s'_l/\sigma(c) \oplus \sigma(I'_l)$ = $s_k \oplus \sigma(I_k)$=s_{k_n}; i.e., R1 holds for k_n. Similarly, $m'_{l_n} = m'_l \oplus \sigma(I'_l) = m_{k_n}$; i.e., R2 holds for k_n. From conditions of Case-1, $n_k \notin \{d,t\}$. In case $\sigma(n_k) \neq r$, then $\sigma(n'_l)$=$\sigma(n_k)$. Thus, $\sigma(n'_l \oplus I'_l)$=$\sigma(n_k \oplus I_k)$=$\sigma(n'_{l_n})$=$\sigma(n_{k_n})$. Also for $\sigma(n_k) = r$, $\sigma(n'_l) = rn$ and $seq(r)_{P_i} = seq(rn)_{P_o}$. Thus, $\sigma(n'_{l_n})$=$\sigma(n_{k_n})$ for $\sigma(n_k) = r$. Therefore, R4 applies for k_n in all possible cases. Finally, since R4 holds and that both executions are at the same label then from property p1 we can conclude that R3 hols at k_n.

Case-2: $\sigma(n_{k_n}) = d$. Then, $u_n = u+1$ and $l_n = k_n+1 = k_2$. This means that execution of P_o should simulate the execution of P_i at k_n in two steps from l. Since R holds for k, we can prove that $\sigma(n'_{l_n}) = \sigma(n_{k_n}) = d$. However, according to property p2, $I_d)_{P_o} \neq I_d)_{P_i}$ (i.e., R3 does not hold from a single step from l). Instead, using properties p2 and p3, we find that R3 is satisfied after two steps. Similar to Case-1, we can prove all other properties at $l_n = l+2$. We can similarly prove **Case-3:** $\sigma(n_{k_n}) = t$.

Case-4: $\sigma(n_k) \to \sigma(n_{k+1}) = t \to h$. Then, $y = 1$ and $l_n = l+2$. Since R4 holds at k, we know that $\sigma(n'_l) = tn$. According to property p7, the next node in P_o is h and in P_i is r (i.e., R3 and R4 does not hold from a single step from l). We also know that by coming from tn that $\sigma(c) = 1$ and that $I_r)_{P_o}$ evaluates to taken branching to h. Thus, we can simply find that R holds for k_n at $l_n = l+2$. We can similarly prove **Case-5:** $\sigma(n_{k_n}) = r$.

Lifting simulation relation to parallel execution: We need to prove that steps by threads other than t preserve the simulation relation [9, 10]. R implies that ($m' = m$) then it is guaranteed that both the original and the transformed program maintains exactly the same view of shared memory for a thread $u \neq t$. Thus, a memory read from an arbitrary location loc by a thread u from m' and m will yield the same values. This conclusion is intuitive as the transformation does not reorder memory operations (read, writes or memory barriers). Further, we need to prove that memory updates by a thread u can not change the memory such that the simulation relation R no longer holds. This is evident by the proof of R that is independent from the shared memory state. Relation R is built on subtle changes in the CFG that is only dependent on the new added local variable c. Note that we did not depend on a specific memory model to prove that the simulation relation holds for the case of parallel execution.

B. Semantics across execution models: In this part we rely on prior work [2, 5] that proves that the execution of an arbitrary program P on a SIMT machine can be simulated by some schedule of the traditional interleaved thread execution (i.e., MIMD execution) of the same program. Thus, terminating kernels on a SIMT machine produce a valid observable behaviour compared with MIMD execution. However, it is still possible that a program that always (i.e., under any loosely fair scheduling) terminate on MIMD to not terminate on SIMT. Therefore, it is sufficient for us to prove that P_o, an output of **T**, always terminate on a SIMT machine if P_o (or equivalently P_i) always terminate on a MIMD machine (i.e., with any arbitrary loosely fair schedule).

Proof Logic Termination is trivially proven if we can prove that all threads executing any arbitrary branch in P_o eventually reach to the branch reconvergence point (i.e., its immediate postdominator). To construct such a proof, we rely on two main claims:

Claim-1: P_o terminates on any arbitrary loosely fair scheduling.

Proof: We assume that P_i terminates on any MIMD machine (i.e., under any arbitrary loosely fair scheduling). However, according to the proof presented earlier the transformation T preserves the program semantics on a MIMD machine and that P_o simulates the behaviour of P_i on a MIMD machine including the termination properties. Thus, we conclude that P_o terminates on a MIMD machine under any arbitrary fair scheduling.

Claim-2: The valuation of the exit condition in any loop in P_o is independent of the valuation of paths parallel to or reachable from the loop. The definitions of parallel and reachable paths is listed in Listing 1.

Proof: This is a forced property by transformation **T** presented in Algorithm 3. As shown by Algorithm 3, any loop that has its exit dependent on the valuation of paths parallel to or reachable from the loop is transformed such that the backward edge of the loop is converted into a forward edge to SafePDom and a backward edge to the original loop header. SafePDOM postdominates 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 (Please refer to Algorithm 2 and its explanation in the paper).

Now we proceed to prove that all threads executing any arbitrary branch in P_o eventually reach to the branch reconvergence point. We prove this by induction over the nesting depth of the control flow graph. For this purpose let's consider an arbitrary branch I_{BR}^k which is a branch with a nesting depth of k. We define the nesting depth as the maximum number of static branch instructions encountered in a control flow path connecting the branch instruction with its reconvergence point R. Note that all branches in the path between I_{BR}^k and its reconvergence point (R) either have the same reconvergence point or a reconvergence point that is postdominated by R. Thus, branches between I_{BR}^k and its reconvergence point has a nesting depth that is equal to or less than k.

Base Case-1: I_{BR}^0. No static branches between the branch instruction and its reconvergence point. Since there is no barriers placed in divergent code, threads diverged to either side of the branch are guaranteed to reach the branch reconvergence point. This follows from two facts: 1) in the absence of barriers and loops (i.e., branches), nothing prevents the forward progress of threads, 2) according to constraint-2, once threads executing one side of the branch reach its reconvergence point, execution switches to threads diverged to the other side. Note that this also applies to single sided branches

Base Case-2a: I_{BR}^1 where $I_{BR}^1 \in P_{T \mapsto R} \vee I_{BR}^1 \in P_{NT \mapsto R}$. This means that the branch itself is encountered again before reaching R. Thus, the branch is part of a loop whose exit reconvergence point is R (in this case, I_{BR}^1 is actually the only exit of this loop). Threads may never reach R if the valuation this loop exits never leads to exit this loop. This could happen under only two hypothesis.

(1) The valuation of the loop exit is independent of thread scheduling (i.e., it is independent of the execution of other paths parallel to or reachable paths to the loop. However, it never evaluates to decision that leads to exit the loop. This contradicts with Claim 1 as it implies that P_o does not terminate under fair scheduling. Thus, we exclude this hypothesis.

(2) The valuation of the loop exit is dependent on scheduling threads at the bottom of the stack (i.e., it is dependent on the execution of other paths parallel to or reachable paths to the loop that contains I_{BR}^1). However, for these threads to get scheduled, the looping threads need to exit and reach their reconvergence point to be popped out of the stack allowing for threads at bottom stack entries to get schedule. However, this hypothesis contradicts with Claim 2 since the operation of **T** forces the valuation of the loop exit condition to be independent of parallel to or reachable from the loop. Thus, we reject this hypothesis. From (1) and (2), we conclude that threads divergent at a branch that follows base case 2a pattern reach their reconvergence point.

Base Case-2b: I_{BR}^1 and $I_{BR}^1 \notin P_{T \mapsto R} \wedge I_{BR}^1 \notin P_{NT \mapsto R}$. This means that the branch is not encountered again before reaching R, however another branch is encountered. This other branch could follow the pattern of base case 1 or base case 2a. However, we have just proven that threads encountering branches that follow the pattern of base Case 1 or 2a are guaranteed to reach their reconvergence points. Further, nothing else could prevent the forward progress of divergent threads divergent at a branch the follows bas case 2b pattern (i.e., no divergent barriers and the branch itself is not encountered by the divergent threads before they reach R). Thus, we conclude that threads divergent at a branch that follows base case 2b pattern reach their reconvergence point. From (Base case-2a) and (Base case-2b), we conclude that in general threads divergent at I_{BR}^1 reach their reconvergence point.

Step Case: Assume that divergent threads at a any branch of nested depth of k or less that belongs to P_o will reach the branch reconvergence point if executed on a SIMT machine. Prove that this will be true for a branch of nested depth of $k_n = k+1$. The step case can be proven is a similar way to the proof of Base Case 2.

Thus we conclude that P_o which is an output of transformation **T** is guaranteed to terminate on a SIMT machine if it terminates on a MIMD machine with any arbitrary loosely fair thread scheduling.

We finally conclude that according to the proof outlines above, **Theorem 1** holds true.

References

[1] 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*, 2012, pp. 113–132.

[2] P. Collingbourne, A. F. Donaldson, J. Ketema, and S. Qadeer, "Interleaving and lock-step semantics for analysis and verification of gpu kernels," in *Programming Languages and Systems*. Springer, 2013, pp. 270–289.

[3] B. Coon and J. Lindholm, "System and method for managing divergent threads in a simd architecture," Apr. 1 2008, uS Patent 7,353,369. [Online]. Available: https://www.google.com/patents/US7353369

[4] A. ElTantawy and T. M. Aamodt, "Mimd synchronization on simt architecture," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2016.

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

[6] C. A. R. Hoare, *Communicating sequential processes*. Springer, 1978.

[7] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen, "Proving correctness of compiler optimizations by temporal logic," *ACM SIGPLAN Notices*, vol. 37, no. 1, pp. 283–294, 2002.

[8] X. Leroy, "A formally verified compiler back-end," *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 363–446, 2009.

[9] W. Mansky, "Specifying and verifying program transformations with PTRANS," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2014.

[10] W. Mansky and E. L. Gunter, "Verifying optimizations for concurrent programs," in *OASIcs-OpenAccess Series in Informatics*, vol. 40. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[11] 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)*, 2010, pp. 235–246.