Extending and Verifying the Weft Race Detection Algorithm

ROHAN YADAV, Stanford University, USA
RUPANSHU SOI, Stanford University, USA
BENJAMIN DRISCOLL, Stanford University, USA
FREDRIK KJOLSTAD, Stanford University, USA
MICHAEL BAUER, NVIDIA, USA
ALEX AIKEN, Stanford, USA

1 Introduction

This document describes our recent efforts to extend the Weft [1] algorithm for race detection in GPU programs that leverage barrier synchronization primitives. This document will assume familiarity with Weft and GPU programming. Our extensions to Weft are focused in two directions:

(1) Modeling a new form of barrier synchronization, *shared-memory* barriers, which were introduced in the Hopper generation of GPUs.
(2) Performing race detection on programs with loops, rather than just straight-line programs.
(3) A mechanization of Weft and our extended results in Lean.

As part of developing these extensions, we had to revisit the results and proof strategies presented in the original Weft paper. This revisiting was to allow for the new results to be proved, and for the recovery of proof strategies to drive the mechanization effort in Lean. During this effort, we found some errors in the original presentation of Weft, and have developed the necessary corrections. All of the main results of the Weft paper still hold, though different proof strategies are required than what was presented in the paper.

This document will first revisit Weft and describe the modeled language and race detection algorithm. We will describe concrete proof strategies for each theorem and will sketch important cases of the proofs. Next, we will discuss the extensions of Weft to shared-memory barriers, and describe how the existing proofs can be adapted to the new language. After that, we will describe an approach to verifying programs with loops with a Weft-style approach. Finally, we will detail several limitations of our approach as well as describe several avenues for future work. We've developed a prototype implementation of these extensions to Weft here and have mechanized all of the presented theorems here. The mechanized proofs themselves are LLM-generated (the theorem statements and definitions were closely human-scrutinized) and unwieldy to read; an LLM can be used to interact with the proofs in a digestable way.

2 Weft

2.1 Weft Language

We will leave a thorough and slower introduction to the Weft language to the original publication [1], which we assume the reader is familiar with. The syntax of Weft's language is shown in Figure 1. The language consists of a CTA T, which is N thread programs consisting of instructions that read and write to shared-memory locations, and instructions that arrive and wait on named barriers.

Authors' Contact Information: Rohan Yadav, Computer Science, Stanford University, Stanford, CA, USA, rohany@stanford.edu; Rupanshu Soi, Computer Science, Stanford University, Stanford, CA, USA; Benjamin Driscoll, Computer Science, Stanford University, Stanford, CA, USA; Fredrik Kjolstad, Computer Science, Stanford University, Stanford, CA, USA; Michael Bauer, NVIDIA, Santa Clara, CA, USA; Alex Aiken, Computer Science, Stanford, Stanford, CA, USA.

2027. ACM 2475-1421/2027/1-ART1
https://doi.org/

$$
\begin{array}{lll}
b \in \{0, \ldots, B-1\} & & \text{barrier names} \\
i \in \{1, \ldots, N\} & & \text{thread identifiers} \\
l & & \text{shared memory locations} \\
n \in \mathbb{N} & & \text{arrival counts} \\
T ::= P_1 \parallel P_2 \parallel \cdots \parallel P_N & & \text{CTAs} \\
P ::= \textbf{return} \mid c \,;\, P & & \text{thread programs} \\
c ::= \textbf{read}\ l \mid \textbf{write}\ l & & \text{commands} \\
\quad \mid \textbf{arrive\_nb}\ b\ n \mid \textbf{wait\_nb}\ b\ n & & \\
S ::= (\mathcal{E}, \mathcal{B}) & & \text{states} \\
\mathcal{E} : \{1, \ldots, N\} \to \{\textbf{true}, \textbf{false}\} & & \text{enabled map} \\
\mathcal{B} : \{0, \ldots, B-1\} \to (\mathcal{I} \times \mathcal{A} \times (\mathbb{N} \cup \{\bot\})) & & \text{barrier map} \\
\mathcal{I}, \mathcal{A} ::= [\,] \mid i :: \mathcal{I} & & \text{synced / arrived lists}
\end{array}
$$

Fig. 1. Syntax of Weft's named barrier language.

The operational semantics of the Weft language are shown in Figure 2. The operational semantics relation $\mathcal{E}, \mathcal{B}, T \rightsquigarrow \mathcal{E}', \mathcal{B}', T'$ relates a CTA T, an enabled-thread map $\mathcal{E}$ and the barrier state $\mathcal{B}$ with the result of a small-step execution. The threads progress independently in execution steps and interact through barrier operations, blocking on synchronizations and notifying other threads to continue. The barrier state contains a list of threads that are parked waiting at a barrier, the threads who have arrived on a barrier, and a natural number representing the expected arrival count (which may be $\bot$ while the barrier is not configured). The rules in Figure 2 describe how a CTA as a whole takes a step, which is either by stepping an arbitrary enabled thread, or by recycling a barrier that has reached the total number of expected participants. It is an important detail that the premises of these rules are mutually exclusive, which forces atomic recycling of barriers. The semantics also contain error productions, which describe how improper use of barriers can result in a runtime error.

2.2 Well Synchronization

2.2.1 Definitions. To introduce the key property of well synchronization, we need to introduce several definitions that allow us to speak concretely about Weft programs. These definitions and some of the surrounding prose have been lifted directly from Weft [1].

A *configuration* C is a pair of a state $s = (\mathcal{E}, \mathcal{B})$ and a CTA T; we use (s, T) to abbreviate $\mathcal{E}, \mathcal{B}, T$. The *initial state* I has $\forall i.\, \mathcal{E}(i) = \textbf{true}$ and $\forall b.\, \mathcal{B}(b) = ([\,], [\,], \bot)$.

Definition 1 (Partial Trace). *A* partial trace *or a* subtrace *is a sequence of configurations* $(s_0, T_0), \ldots, (s_n, T_n)$ *such that for any two successive configurations we have* $(s_j, T_j) \rightsquigarrow (s_{j+1}, T_{j+1})$.

A *complete* trace ends in either *done*, **err**, or a deadlock, and a *successful* trace ends in *done*.

Definition 2 (Trace). *A (complete)* trace τ *starting from a configuration* (s, T) *is a subtrace* $(s, T), \ldots, (s', \mathit{done})$, *or* $(s, T), \ldots, (\textbf{err}, T')$, *or* $(s, T), \ldots, (s', T')$ *where* $T' \neq \mathit{done}$ *and no rule is applicable (deadlock).*

We also need a notion of time t which says at what step a command is executed in a trace. Program points are defined in the standard manner: before and after each command c. Each program point is uniquely identified by the command just before it: a program point η and a command c_η *correspond*

The operational semantics is given by two small-step relations: one that steps a whole CTA, and one that steps an individual thread program (with identifier i):

$$\mathcal{E}, \mathcal{B}, T \rightsquigarrow \mathcal{E}', \mathcal{B}', T' \qquad\qquad \mathcal{E}, \mathcal{B}, i, P \rightsquigarrow \mathcal{E}', \mathcal{B}', i, P'$$

$$\frac{\forall b.\, \big(\mathcal{B}(b) = ([\,],[\,],\bot) \vee (\mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \wedge |\mathcal{I}| + |\mathcal{A}| < n)\big) \qquad \mathcal{E}, \mathcal{B}, i, P_i \rightsquigarrow \mathcal{E}', \mathcal{B}', i, P_i'}{\mathcal{E}, \mathcal{B}, P_1 \parallel \ldots \parallel P_i \parallel \ldots \parallel P_N \rightsquigarrow \mathcal{E}', \mathcal{B}', P_1 \parallel \ldots \parallel P_i' \parallel \ldots \parallel P_N} \textsc{Cta-Step}$$

$$\frac{\mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \qquad |\mathcal{I}| + |\mathcal{A}| = n \qquad T = P_1 \parallel \ldots \parallel P_N \qquad \forall i.\, P_i = ite(i \in \mathcal{I},\ \textbf{wait\_nb}\ b\ n; P_i',\ P_i') \qquad T' = P_1' \parallel \ldots \parallel P_N' \qquad \mathcal{E}' = \mathcal{E}[\textbf{true}/\mathcal{I}]}{\mathcal{E}, \mathcal{B}, T \rightsquigarrow \mathcal{E}', \mathcal{B}[([\,],[\,],\bot)/b], T'} \textsc{Cta-Recycle}$$

$$\frac{}{\mathcal{E}, \mathcal{B}, \textbf{return} \parallel \ldots \parallel \textbf{return} \rightsquigarrow \mathcal{E}, \mathcal{B}, \mathit{done}} \textsc{Cta-Done}$$

$$\frac{\mathcal{B}(b) = ([\,],[\,],\bot) \qquad \mathcal{B}' = \mathcal{B}[([\,], [\,] :: i, n)/b]}{\mathcal{E}, \mathcal{B}, i, \textbf{arrive\_nb}\ b\ n; c \rightsquigarrow \mathcal{E}, \mathcal{B}', i, c} \textsc{Arrive-Init}$$

$$\frac{\mathcal{E}(i) = \textbf{true} \qquad \mathcal{E}' = \mathcal{E}[\textbf{false}/i] \qquad \mathcal{B}(b) = ([\,],[\,],\bot) \qquad \mathcal{B}' = \mathcal{B}[([\,] :: i, [\,], n)/b]}{\mathcal{E}, \mathcal{B}, i, \textbf{wait\_nb}\ b\ n; c \rightsquigarrow \mathcal{E}', \mathcal{B}', i, \textbf{wait\_nb}\ b\ n; c} \textsc{Wait-Init}$$

$$\frac{\mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \qquad 0 < |\mathcal{I}| + |\mathcal{A}| < n}{\mathcal{E}, \mathcal{B}, i, \textbf{arrive\_nb}\ b\ n; c \rightsquigarrow \mathcal{E}, \mathcal{B}[(\mathcal{I}, \mathcal{A} :: i, n)/b], i, c} \textsc{Arrive}$$

$$\frac{\mathcal{E}(i) = \textbf{true} \qquad \mathcal{E}' = \mathcal{E}[\textbf{false}/i] \qquad \mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \qquad \mathcal{B}' = \mathcal{B}[(\mathcal{I} :: i, \mathcal{A}, n)/b] \qquad 0 < |\mathcal{I}| + |\mathcal{A}| < n}{\mathcal{E}, \mathcal{B}, i, \textbf{wait\_nb}\ b\ n; c \rightsquigarrow \mathcal{E}', \mathcal{B}', i, \textbf{wait\_nb}\ b\ n; c} \textsc{Wait}$$

$$\frac{\mathcal{E}(i) = \textbf{true} \qquad \mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \qquad |\mathcal{I}| + |\mathcal{A}| = n}{\mathcal{E}, \mathcal{B}, i, \textbf{wait\_nb}\ b\ n; c \rightsquigarrow \textbf{err}, i, \textbf{wait\_nb}\ b\ n; c} \textsc{Wait-Full}$$

$$\frac{\mathcal{E}(i) = \textbf{true} \qquad \mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \qquad n \neq m}{\mathcal{E}, \mathcal{B}, i, \textbf{wait\_nb}\ b\ m; c \rightsquigarrow \textbf{err}, i, \textbf{wait\_nb}\ b\ m; c} \textsc{Wait-Mismatch}$$

$$\frac{\mathcal{E}(i) = \textbf{true} \qquad \mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \qquad |\mathcal{I}| + |\mathcal{A}| = n}{\mathcal{E}, \mathcal{B}, i, \textbf{arrive\_nb}\ b\ n; c \rightsquigarrow \textbf{err}, i, \textbf{arrive\_nb}\ b\ n; c} \textsc{Arrive-Full}$$

$$\frac{\mathcal{E}(i) = \textbf{true} \qquad \mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \qquad n \neq m}{\mathcal{E}, \mathcal{B}, i, \textbf{arrive\_nb}\ b\ m; c \rightsquigarrow \textbf{err}, i, \textbf{arrive\_nb}\ b\ m; c} \textsc{Arrive-Mismatch}$$

$$\frac{\mathcal{E}, \mathcal{B}, i, P_i \rightsquigarrow \textbf{err}, i, P_i}{\mathcal{E}, \mathcal{B}, P_1 \parallel \ldots \parallel P_i \parallel \ldots \parallel P_N \rightsquigarrow \textbf{err}, P_1 \parallel \ldots \parallel P_i \parallel \ldots \parallel P_N} \textsc{Cta-Err}$$

Fig. 2. Small-step operational semantics of the named-barrier language. The first four rules step a CTA; the remaining rules step an individual thread. Here $ite(e_1, e_2, e_3)$ is "if e_1 then e_2 else e_3", $A[x/y]$ updates map A at y to x (and $A[x/Y]$ at every $y \in Y$), and $\mathcal{L} :: i$ appends i to list $\mathcal{L}$. The error productions for **arrive_nb** mirror those for **wait_nb**.

if η is the program point just after c_η. If η is the program point just after the command c_η, then we define a quantity $t(\tau, \eta)$ which provides the step at which c_η is executed in the trace τ.

Definition 3 (Time). *Time $t(\tau, \eta^i) = n$ if τ has a prefix $(s, T) \rightsquigarrow^{n-1} (s_1, P_1^{(1)} \parallel \ldots \parallel P_i^{(1)} \parallel \ldots \parallel P_N^{(1)}) \rightsquigarrow (s_2, P_1^{(2)} \parallel \ldots \parallel P_i^{(2)} \parallel \ldots \parallel P_N^{(2)})$ and $P_i^{(1)} = c_{\eta^i}; P_i^{(2)}$.*

This definition ensures that for **read**, **write**, and **arrive_nb**, the time $t(\tau, \cdot)$ provides the execution step when these are executed in τ. For **wait_nb** it provides the step when the corresponding barrier is recycled. Using time, we can define a *happens-before* relationship.

Definition 4 (Sound and Precise Happens-Before). *For a configuration (s, T), a happens-before relation R is* sound *and* precise *if for all pairs of commands (c_{η_1}, c_{η_2}) we have $R(c_{\eta_1}, c_{\eta_2})$ iff for all traces τ starting from (s, T) we have $t(\tau, \eta_1) \leq t(\tau, \eta_2)$.*

This happens-before relation is slightly non-standard: it includes the commands that execute simultaneously (note the $\leq$). For example, all **wait_nb** commands that synchronize together are included in the happens-before relation R. The fact that this relation is over $\leq$ has important implications for the race-detection algorithm discussed next.

To define well synchronization, we need to define *generations*. For a trace τ, $Gen(\tau)$ maps a synchronization command to a generation ID observed in the trace τ. For example, a generation ID of 2 for a command **wait_nb** 0 n indicates that this command was used to register on barrier 0 after this barrier had been recycled once previously. The generation ID of unexecuted commands is set to 0.

Definition 5 (Generation). *$Gen(\tau)(c_\eta) = n$ if $t(\tau, \eta) = m$, c_η is an operation on barrier b, and the first m steps of τ contain n recyclings of barrier b.*

CTAs with the same generation mapping for all traces are called *well-synchronized*. Well-synchronization is a deep property of barrier programs that enables significantly more efficient program analysis than common model-checking-based approaches for verification of concurrent programs.

Definition 6 (Well-Synchronized CTA). *A CTA T is* well-synchronized *if for any two traces τ_1 and τ_2 that start from (I, T), for all synchronization commands c, we have $Gen(\tau_1)(c) = Gen(\tau_2)(c) \neq 0$.*

This definition is directly generalized to arbitrary starting states.

Definition 7 (Well-Synchronized Configuration). *A configuration (s, T) is* well-synchronized *if for any two traces τ_1 and τ_2 that start from (s, T), for all synchronization commands c, we have $Gen(\tau_1)(c) = Gen(\tau_2)(c) \neq 0$.*

We let the predicate $\mathrm{WS}(T)$ and $\mathrm{WS}(s, T)$ to refer to when a CTA and a configuration are well-synchronized.

2.3 Checking Well Synchronization

Given the definition of well-synchronization (Definition 6), we want an algorithm to check when a CTA T is indeed well-synchronized, and in the process, extract information from the well-synchronization analysis to aid in race-detection. The algorithm for checking well-synchronization is shown in Algorithm 1. The algorithm simulates an arbitrary execution of T. With this execution, it constructs a candidate happens-before relation R, and then uses R to check well-synchronization. The key idea behind the check is to ensure that there are synchronization edges that enforce a strict ($<$) dependence between operations at successive generations of the same barrier. If these dependencies are present, then no concurrent interleaving may cause the observed generation of a barrier to change across traces.

It is important to note that this presentation of the algorithm slightly deviates from the original presentation in the Weft paper. In particular, we add the check of $c_3; c_2 \in T$, and check all operations

Algorithm 1 Check whether a CTA is well-synchronized.

```
1:  function WellSync(T : CTA) : Bool
2:      τ ← simulate an arbitrary execution of T from the initial state.
3:      if τ deadlocks or errs then
4:          return false
5:      G ← Gen(τ)
6:      R ← ∅
7:      R ← ∅
8:      for each c1 ; c2 ∈ T do
9:          R ← R ∪ {(c1, c2)}
10:     for each c1 ≡ arrive_nb b n, c2 ≡ sync_nb b n s.t. G(c1) = G(c2) do
11:         R ← R ∪ {(c1, c2)}
12:     for each c1 ≡ sync_nb b n, c2 ≡ sync_nb b n s.t. G(c1) = G(c2) do
13:         R ← R ∪ {(c1, c2), (c2, c1)}
14:     R ← transitive closure of R
15:     for each (c1, c2) ∈ T using barrier b and generations G(c1) = g ∧ G(c2) = g + 1 do
16:         if c3; c2 ∉ T then
17:             return false
18:         if c3; c2 ∈ T ∧ (c1, c3) ∉ R ∧ c1 ≠ c3 then
19:             return false
20:     return true
```

using a barrier b instead of just **sync_nb** operations. These modifications to the algorithm are required to prove the critical results discussed next.

The following theorems are discussed in the Weft paper. We will describe each, then spend time discussing the concrete proof strategy for each.

Theorem 1. *For well-synchronized configurations, the happens-before relation R computed by Algorithm 1 is both sound and precise.*

With Theorem 1, race-detection using R is trivial. If two shared memory operations on the same memory location (where at least one is not a read) do not have a happens-before edge between them in R, then the two operations race.

Theorem 2. *If WellSync(T) returns true then T is well-synchronized.*

2.3.1 Soundess of R. Because T is known to be well-synchronized, the soundness of R is straightforward to argue. Concretely, showing soundness involves showing that if $(c_1, c_2) \in R$, $\forall \tau : t(\tau, c_1) \leq t(\tau, c_2)$. The case of thread-order edges is trivial, as the operational semantics execute instructions in thread-order. For edges between **arrive_nb** and **sync_nb** operations, we know the generations observed by Algorithm 1 are the only possible generations. Therefore, the operational semantics force all **sync_nb** instructions to complete after all **arrive_nb**'s on the barrier have completed. A similar argument applies for edges between **sync_nb** operations, as these instructions all complete during the recycle step of the semantics.

2.3.2 Preciseness of R. Proving preciseness of R is not as straightforward. The concrete statement that must be shown is that if $\forall \tau, c_1, c_2 : t(\tau, c_1) \leq t(\tau, c_2) \implies (c_1, c_2) \in R$. The Weft paper describes a proof strategy based on induction on the size of programs. However, this strategy is not correct, as the induction over only well-synchronized programs does not allow all well-synchronized programs

to be created (a well-synchronized program does not necessarily stay well-synchronized after a new instruction is added).

The proof is trivial for the cases where $c_1 = c_2$ and c_1, c_2 are on the same thread. The interesting case is when c_1, c_2 are on different threads. The proof in this case proceeds by proving the contrapositive, where $(c_1, c_2) \notin R \implies \exists \tau : t(\tau, c_2) < t(\tau, c_1)$. The following proof strategy was proposed by Claude.

The intuition behind showing that such a trace exists is to construct a sub-trace that has executed all instructions $< c_1$. If this construction holds, it directly provides a witness to prove the claim.

The key idea is to define the set $C = \{c \mid (c_1, c) \notin R \wedge c \neq c_1\}$, containing all instructions that are not strictly after c_1. We will show that we can construct a sub-trace that executes all instructions in C and no-other instructions, which will act as the witness trace. This proof is done by induction, which will show that if a trace starts in a configuration that has only executed instructions in C, and there exist remaining instructions in C, then there exists a step that the trace can take to stay in C.

This will be done by maintaining the additional invariants that:

- Every thread has not yet run a command not in C.
- Every parked thread is parked at a command in C.

To show that such a step always exists, there are a few cases:

- A barrier is full: This case directly falls out of the invariant that every parked thread is parked at a command in C.
- A barrier is not full: In this case, we must take an interleave step that steps a thread. By the maintained invariant, all threads have not yet run an instruction that is not in C (though the next instruction some threads may run could not be in C). Case again on whether there is an enabled thread that can run an instruction in C:
  - If such a thread exists, it can be stepped, which either retires the instruction or sleeps the thread. Either case preserves both invariants. An error step cannot be taken because that would contradict the assumption that the program is well-synchronized.
  - If no thread exists, that means all threads with an instruction in C are disabled. We must show that this case is not possible. To do so, imagine successfully completing the trace by running any remaining threads; this must be possible becuase $\mathrm{WS}(T)$. To proceed, some other instruction must have completed the barriers that the threads blocked in C are stuck on. Because $\mathrm{WS}(T)$, all observed generations in the trace are the only possible generations. Consider one of the blocked instructions that is the first in the successful trace to be recycled and wake up, and one of the instructions that unblocked it but did not execute in the trace until it got blocked. The range of steps between these instructions in the trace must all be instructions that are not in C, because we know that all threads capable of executing instructions in C are blocked. The unblocking instruction would be a sync or an arrive, and in each case would have an edge in R, due to the edge addition rules, because the sync/arrive must have the same generation as the blocked instruction to wake it. Because this edge exists in R, the unblocking instruction must have actually then been in C! This is a contradiction, as all threads that are capable of running instructions in C have parked, which then causes this case to become impossible.

With this lemma complete, we can construct a trace from the initial state that executes all instructions in C, which includes c_2 but not c_1. Since all traces of a well-synchronized program complete, there must exist a suffix of this trace that runs the entire program to completion. Put together, this is a complete trace that has $c_2 < c_1$, which is the proof obligation.

2.3.3 Soundness of Algorithm 1. The soundness of Algorithm 1 is the trickiest proof from the Weft paper, and the strategy originally presented in the paper does not work. The goal is to show that if WellSync(T) = true then WS(T). This statement is difficult to prove directly, and we first need to introduce some definitions and helper theorems to make the full proof go through. The induction strategy for this proof was proposed by Claude.

First, for a trace τ, define the set $\tau[b, g]$ which refers to the sub-trace of τ that references all instructions using barrier b at generation g.

Now, we define the relation Conforms(τ, τ'), which holds when

(1) $\forall c \in \tau', G(\tau)(c) = G(\tau')(c)$
(2) For every generation g in τ', consider $\tau[b, g]$, and consider the state S_g of the barrier b before either the recycle step of g or the end of τ. At every step of τ' on g, the state in τ' must *agree* with S_g. This means that the arrival count of b must be $\leq S_g$, and the expected arrival count must either be $\perp$ or the same as in S_g, and the parked threads must be a subset of the parked threads in S_g.
(3) If $(x, y) \in R$ (computed by the algorithm) and $y \in \tau'$, then $t(x, \tau') \leq t(y, \tau')$.

The idea behind the conformance relation is to show that steps of an arbitrary trace τ' are still agreeing with the steps made by τ, the trace simulated by Algorithm 1. We will leverage the conformance invariant to show the following helper theorem:

Theorem 3. *If WellSync(T) =* ***true*** *along trace τ, then for all (potentially incomplete) traces τ' of T, Conforms(τ, τ').*

Proof. This theorem can be shown by forwards induction on the length of τ'. The base case holds trivially.

In the induction step, τ' has been extended by some transition σ. Note that here we assume σ exists; we will show in a separate theorem that σ must exist. We will perform a case analysis on σ and show that each step either preserves conformance or is not possible. In this document, we will not describe every case, but describe the illustrative cases. In each case, we will show how each aspect of conformance holds, in order.

Case Arrive-Configure: σ is an arrive-configure executing some instruction $c =$ **arrive_nb** b n. We'll first show $G(\tau)(c) = G(\tau')(c)$. Suppose that $G(\tau)(c) = g$ and that there are k recyclings of b so far in τ'. We'll show that k must equal g.

Suppose $k > g$, meaning that c executes later than it should in τ'. Therefore, generation g in τ' must have already completed. Because $G(\tau)(c) = g$, b required n registrants to complete, and if c isn't one of them, then another instruction c' where $G(\tau)(c') \neq g$ but $G(\tau')(c') = g$, taking c's place. However, because of assumption (1) of conformance of τ' without σ, $G(\tau)(c') = G(\tau')(c')$, which is a contradiction.

Suppose $k < g$, meaning that c executes earlier than it should in τ'. If $k < g$, then by the pigeonhole principle, there must be at least one instruction c' with $G(\tau)(c') = k$ that can't be included in generation k in τ'. However, from the check in Algorithm 1, we know that the happens-before edge $(c', c_3) \in R$, where $c_3; c \in T$. Because of thread-order execution, c_3 must have already executed for c to execute. By assumption 3 of conformance of τ' without σ, if $(c', c_3) \in R$ and c_3 has already executed, then c' must have already executed, which is a contradiction. *Important Note:* If Algorithm 1 only checked $(c_1, c_2) \in R$, we would not be able to derive this contradiction.

Therefore, $k = g$, upholding component (1) of conformance.

Now, we'll show component (2) of conformance. Because we know the generation of c is the same in both τ and τ', we know the arrival step must remain consistent (same expected count, arrival count is consistent) because c registered at g in the successful trace τ.

Finally, component (3). No edges are added into R yet from just an arrive other than the thread-order edge, which uphold (3) trivially.

Case Recycle: In the recycle case, all parked threads at a barrier are re-enabled and have their pending **sync_nb** instruction retired. Because of conformance, all parked threads are the same waiters in S_g, so their generations after retirement are the same as in τ, maintaining (1).

Because the barrier returns to an unconfigured state, (2) is maintained.

For (3), edges are added to R from all **arrive_nb**'s at the same generation to all **sync_nb** instructions that just retired. These edges satisfy (3), as the arrives must have already executed in τ' for the recycle to occur. Then, edges are added between all **sync_nb** instructions getting retired, which also satisfies (3).

Case Arrive-Mismatch: The goal in this case is to show that this error production cannot occur in conforming traces. From conformance, we know that all other operations on this barrier (so far, of which there is at least one) have also executed in τ. Suppose the number of recyclings of b in τ' is k. Clearly, c cannot have generation k in τ, because that would mean τ errored, which isn't true. Therefore, $G(\tau)(c) \neq k$, meaning either $G(\tau)(c) < k \vee G(\tau)(c) > k$.

Suppose $G(\tau)(c) < k$, meaning that c executed later than it should have. This case is similar to the previous "late" argument about why a late execution violates conformance.

Suppose $G(\tau)(c) > k$, meaning that c executed earlier than it should have. This case is similar to prior arguments, where the dependence edges in R forbid early execution.

□

Theorem 4. *All complete traces τ' such that Conforms(τ, τ') are successful.*

Proof. Consider a complete trace τ' such that Conforms(τ, τ') but τ' is stuck. By conformance, all generations match between τ and τ', and all parked threads running **sync_nb** operations are parked at the right generation. Therefore, the way that a deadlock can occur is that there exists a thread parked on a **sync_nb** instruction c while at least one of the instructions that registers on c''s barrier has not yet executed. There may be multiple stuck threads; in this case, let c be any parked **sync_nb** that has the smallest execution time in τ. For one of the registrants on c's barrier to not have run, it must be stuck behind some other instruction $c' =$ **sync_nb**. Because the registrant r must execute before c, $t(\tau, r) \leq t(\tau, c)$. But because r is stuck behind e, $t(\tau, e) < t(\tau, r) \leq t(\tau, c)$. This is a contradiction, because c was supposed to be the parked **sync_nb** with the smallest execution time in τ. □

Theorem 5. *If WellSync(T) =* ***true*** *then WS(T).*

Proof. Follows from the combination of Theorem 3 and Theorem 4. □

2.3.4 Completeness of Algorithm 1. The goal of this case is to show that if WellSync(T) = false then T is not well-synchronized. We will prove this by the contrapositive, by showing that if WS(T) then WellSync(T) = true.

The proof will proceed by showing that the algorithm cannot return false in the two cases that it may. There are two cases of the proof; in both, we will assume that the algorithm returns false and will show how this is a contradiction.

- Case τ deadlocks or errors: Trivially, T is not well-synchronized.
- Case $c_3; c_2 \notin T$: In this case, it is clear how to construct a trace of the program that observes a different G, as the predecessor-less instruction can choose to run whenever. Thus, T cannot be well-synchronized.
- Case $(c_1, c_3) \notin R$: This case is more involved. First, we know that R is sound and precise because WS(T). We know there exists τ where $G(\tau)(c_1) = g \wedge G(\tau)(c_2) = g + 1$, otherwise

WellSync(T) would have returned false earlier. Because $(c_1, c_3) \notin R$, we can apply the same strategy as in Section 2.3.2 to construct a trace τ' where $t(\tau', c_3) < t(\tau', c_1)$. Take this subtrace right before the execution of c_1. Because c_2 uses the same barrier as c_1, and c_3 has executed, it is now possible for c_2 to observe a generation of g instead of $g + 1$, and result in an error or progression on the wrong generation. If this was possible, then T would not be well-synchronized, which is a contradiction. However, just this construction is not sufficient for when c_1 is a **sync_nb**, because c_1 could still register before c_3 and then retire after c_3. Fortunately, the construction used in Section 2.3.2 allows for an even stronger construction where c_1 is gauranteed to not even register before c_3 in the trace, ensuring that c_2 sees an earlier generation. In particular, we consider the configuration where all instructions in C (using the notation from the previous section) have executed, and the next instruction of every thread is an instruction not in C, and no syncs in C are parked.

3 Shared-Memory Barriers

We now discuss the extensions to Weft to model shared-memory barriers, a new interface and synchronization mechanism available on NVIDIA GPUs since the Hopper architectural generation.

3.1 Background

Shared-memory barriers (or *mbarriers*) are directly integrated with on-chip accelerators like the Tensor Core and Tensor Memory Accelerator, and act as the user-facing synchronization mechanism; operations dispatched to on-chip accelerators trigger mbarriers to notify that the accelerator has completed a batch of work. Mbarriers diverge from named-barriers in several ways:

- The number of mbarriers is limited only by the number of shared-memory locations, rather than a fixed set of 15 named-barriers.
- Mbarriers are initialized once by a single thread, and maintain the same initialized expected arrival count across all generations.
- The most salient difference is the explicit exposure of barrier generations to the synchronization primitives themselves, and that waits do not contribute to the advancing of a barrier. These changes add new complexity to analysis and programming with mbarriers, and permit a wider variety of interleavings.

In particular, mbarrier wait operations accept a *phase* argument, which is either 0 or 1. If the argument phase to a wait operation matches the parity of the barrier's current generation, the wait parks and disables the thread, just like a named-barrier. However, if the argument phase does not match the parity of the barrier's current generation, the wait operation returns immediately. As discussed previously, these wait operations also do not affect when the mbarrier resolves and gets recycled, means wait operations can significantly drift if not properly synchronized. We discuss next the formal modeling of these barriers, and show how the Weft infrastructure can be adapted without significant modification to prove strong results about mbarrier programs.

3.2 Language Extensions

The extended language for Weft with mbarriers is shown in Figure 3. The general structure of the language is unchanged, with the only additions being a namespace of mbarrier identifiers, instructions to manipulate mbarriers, and a separate mapping $\mathcal{B}_M$ to manage mbarrier state. The operational semantics of the mbarrier extensions are shown in Figure 3, along with extended / modified rules from the named-barrier language. The structure of the semantics is the same as with named-barriers, except the program state now consists of separate state for both named-barriers and mbarriers.

$$
\begin{array}{rll}
nb \in \{0, \ldots, NB-1\} & & \text{named barrier names} \\
sb \in \{0, \ldots, SB-1\} & & \text{shared barrier names} \\
i \in \{1, \ldots, N\} & & \text{thread identifiers} \\
l & & \text{shared memory locations} \\
n \in \mathbb{N} & & \text{natural numbers} \\
ph \in \{0, 1\} & & \text{mbarrier phase bits} \\
T ::= P_1 \parallel P_2 \parallel \cdots \parallel P_N & & \text{CTAs} \\
P ::= \textbf{return} \mid c\,;P & & \text{thread programs} \\
c ::= \textbf{read}\ l \mid \textbf{write}\ l & & \text{commands} \\
\quad \mid \textbf{arrive\_nb}\ nb\ n \mid \textbf{sync\_nb}\ nb\ n & & \\
\quad \mid \textbf{init\_mb}\ sb\ n & & \text{mbarrier initialization} \\
\quad \mid \textbf{arrive\_mb}\ sb & & \text{mbarrier arrivals} \\
\quad \mid \textbf{wait\_mb}\ sb\ ph & & \text{mbarrier waiting (with phase)} \\
S ::= (\mathcal{E}, \mathcal{B}_N, \mathcal{B}_M) & & \text{states} \\
\mathcal{E} : \{1, \ldots, N\} \to \{\textbf{true}, \textbf{false}\} & & \text{enabled map} \\
\mathcal{B}_N : \{0, \ldots, NB-1\} \mapsto (\mathcal{I}, \mathcal{A}, (\mathbb{N} \cup \{\bot\})) & & \text{named barrier map} \\
\mathcal{B}_M : \{0, \ldots, SB-1\} \mapsto (\mathcal{I}, \mathcal{A}, \mathbb{N}, \{0, 1\}) & & \text{shared barrier state} \\
\mathcal{I}, \mathcal{A} ::= [\,] \mid i :: \mathcal{I} & & \text{synced / arrived lists}
\end{array}
$$

Fig. 3. The syntax of Weft extended with support for shared-memory barriers (mbarriers).

The handling of initialization is straightforward, and the arrival logic for mbarriers is analagous to arrivals for named-barriers. The divergence between the two forms of barriers can be seen in the rules for waits. The rule for when a wait parks is similar to named-barriers, but contains the predicate that the barrier's current generation parity matches the phase used as an argument to the wait operation. When this phase doesn't match, the thread immediately retires the wait instruction. This can occur when the user intentionally does not match the argument phase to the barrier's current phase, or the barrier advances before a thread reaches the wait instruction. These semantics show how the application must carefully choose the phase argument for each barrier wait to ensure that the desired synchronization occurs; incorrect phase management can result in wait operations not performing the intended synchronization.

3.3 Well Synchronization

3.3.1 Generations. Fortunately, the property of well-synchronization also directly applies to mbarrier programs. Before stating the algorithm, we first must revisit the definition of generation for mbarrier operations. The definition of generation for **arrive_mb** operations is the same, but the definition for **wait_mb** operations differs:

$$
G(\tau)(\textbf{wait\_mb}\ b\ p) = \begin{cases} g & \text{if } g \equiv p \pmod 2 \\ g-1 & \text{otherwise} \end{cases}
$$

where g is the number of recyclings of b strictly before the wait operation in τ. The reason for this case is because of the phase-mismatch property of mbarriers: when the current phase of a barrier doesn't match the argument phase, the argument phase is assumed to refer to the previous generation instead of the current generation.

$$\frac{\begin{array}{c}\forall nb.\ \big(\mathcal{B}_N(nb) = ([\,],[\,],\bot) \vee (\mathcal{B}_N(nb) = (\mathcal{I},\mathcal{A},n) \wedge |\mathcal{I}| + |\mathcal{A}| < n)\big) \\ \forall sb \in \mathrm{dom}(\mathcal{B}_M).\ \mathcal{B}_M(sb) = (\mathcal{I},\mathcal{A},n,ph) \wedge |\mathcal{A}| < n \\ \mathcal{E}, \mathcal{B}_N, \mathcal{B}_M, i, P_i \rightsquigarrow \mathcal{E}', \mathcal{B}'_N, \mathcal{B}'_M, i, P'_i\end{array}}{\mathcal{E}, \mathcal{B}_N, \mathcal{B}_M, P_1 \parallel \ldots \parallel P_i \parallel \ldots \parallel P_N \rightsquigarrow \mathcal{E}', \mathcal{B}'_N, \mathcal{B}'_M, P_1 \parallel \ldots \parallel P'_i \parallel \ldots \parallel P_N}\ \textsc{Cta-Step}$$

$$\frac{sb \notin \mathrm{dom}(\mathcal{B}_M) \qquad \mathcal{B}'_M = \mathcal{B}_M[([\,],[\,],n,0)/sb]}{\mathcal{E}, \mathcal{B}_N, \mathcal{B}_M, i, \textbf{init\_mb}\ sb\ n; c \rightsquigarrow \mathcal{E}, \mathcal{B}_N, \mathcal{B}'_M, i, c}\ \textsc{MB-Init}$$

$$\frac{sb \in \mathrm{dom}(\mathcal{B}_M)}{\mathcal{E}, \mathcal{B}_N, \mathcal{B}_M, i, \textbf{init\_mb}\ sb\ n; c \rightsquigarrow \textbf{err}, i, \textbf{init\_mb}\ sb\ n; c}\ \textsc{MB-Init-Err}$$

$$\frac{\mathcal{B}_M(sb) = (\mathcal{I},\mathcal{A},n,ph) \qquad \mathcal{B}'_M = \mathcal{B}_M[(\mathcal{I},\mathcal{A} :: i,n,ph)/sb]}{\mathcal{E}, \mathcal{B}_N, \mathcal{B}_M, i, \textbf{arrive\_mb}\ sb; c \rightsquigarrow \mathcal{E}, \mathcal{B}_N, \mathcal{B}'_M, i, c}\ \textsc{MB-Arrive}$$

$$\frac{sb \notin \mathrm{dom}(\mathcal{B}_M)}{\mathcal{E}, \mathcal{B}_N, \mathcal{B}_M, i, \textbf{arrive\_mb}\ sb; c \rightsquigarrow \textbf{err}, i, \textbf{arrive\_mb}\ sb; c}\ \textsc{MB-Arrive-Err}$$

$$\frac{\begin{array}{c}\mathcal{E}(i) = \textbf{true} \\ \mathcal{E}' = \mathcal{E}[\textbf{false}/i] \qquad \mathcal{B}_M(sb) = (\mathcal{I},\mathcal{A},n,ph) \qquad \mathcal{B}'_M = \mathcal{B}_M[(\mathcal{I} :: i,\mathcal{A},n,ph)/sb]\end{array}}{\mathcal{E}, \mathcal{B}_N, \mathcal{B}_M, i, \textbf{wait\_mb}\ sb\ ph; c \rightsquigarrow \mathcal{E}', \mathcal{B}_N, \mathcal{B}'_M, i, \textbf{wait\_mb}\ sb\ ph; c}\ \textsc{MB-Wait-Block}$$

$$\frac{\mathcal{B}_M(sb) = (\mathcal{I},\mathcal{A},n,ph') \qquad ph \neq ph'}{\mathcal{E}, \mathcal{B}_N, \mathcal{B}_M, i, \textbf{wait\_mb}\ sb\ ph; c \rightsquigarrow \mathcal{E}, \mathcal{B}_N, \mathcal{B}_M, i, c}\ \textsc{MB-Wait-Pass}$$

$$\frac{sb \notin \mathrm{dom}(\mathcal{B}_M)}{\mathcal{E}, \mathcal{B}_N, \mathcal{B}_M, i, \textbf{wait\_mb}\ sb\ ph; c \rightsquigarrow \textbf{err}, i, \textbf{wait\_mb}\ sb\ ph; c}\ \textsc{MB-Wait-Err}$$

$$\frac{\begin{array}{c}\mathcal{B}_M(sb) = (\mathcal{I},\mathcal{A},n,ph) \qquad |\mathcal{A}| = n \qquad \forall i \in \mathcal{I}, \mathcal{E}(i) = \textbf{false} \qquad T = P_1 \parallel \ldots \parallel P_N \\ \forall i.\ P_i = ite(i \in \mathcal{I},\ \textbf{wait\_mb}\ sb\ ph; P'_i,\ P'_i) \qquad T' = P'_1 \parallel \ldots \parallel P'_N \qquad \mathcal{E}' = \mathcal{E}[\textbf{true}/\mathcal{I}]\end{array}}{\mathcal{E}, \mathcal{B}_N, \mathcal{B}_M, T \rightsquigarrow \mathcal{E}', \mathcal{B}_N, \mathcal{B}_M[([\,],[\,],n,ph \oplus 1)/sb], T'}\ \textsc{MB-Recycle}$$

Fig. 4. Operational semantics for the mbarrier fragment of Weft.

3.3.2 Algorithm. The extended algorithm for checking well-synchronization of mbarrier programs is shown in Algorithm 2. The algorithm maintains a similar structure as Algorithm 1 and keeps several of the same checks, which still apply to named-barriers and mbarrier arrivals. The non-trivial deviation of the algorithm is for handling mbarrier waits, which have to interact with the capability of mbarrier wait operations to observe multiple generations of a barrier at the same time. The first group of checks for **wait_mb** operations are similar to the standard checks in Algorithm 1, ensuring that a **wait_mb** can't observe a generation earlier than it was intended to. The second check ensures that a **wait_mb** can't observe a *later* generation than it was intended to. Because **wait_mb** operations can reference the generation before the current generation, a wait at g can interleave with other mbarrier operations at $g + 1$. However, to ensure that the **wait_mb** can not observe $g + 2$, there must be at least one **arrive_mb** at generation $g + 1$ with a happens-before relationship into the **wait_mb**. Next, we discuss how to extend the proof strategies for Algorithm 1 to the extensions described in Algorithm 2.

3.3.3 Soundness of R. The edges added between **arrive_mb** and **wait_mb** are directly justifiable, as all **arrive_mb** operations must complete before the corresponding **wait_mb** operations can

Algorithm 2 Check whether a CTA is well-synchronized with named-barrier and mbarrier operations.

```
1:  function WellSync+(T : CTA) : Bool
2:      τ ← simulate an arbitrary execution of T from the initial state.
3:      if τ deadlocks or errs then
4:          return false
5:      G ← Gen(τ)
6:      ▷ All instructions that can change a barrier's arrival count. ◁
7:      Reg(b, g) ← { c | c ≡ sync_nb b n ∨ c ≡ arrive_nb b n ∨ c ≡ arrive_mb b, G(c) = g }
8:      ▷ The next four steps are unchanged from Algorithm 1. ◁
9:      R ← ∅
10:     for each c1 ; c2 ∈ T do
11:         R ← R ∪ {(c1, c2)}
12:     for each c1 ≡ arrive_nb nb n, c2 ≡ sync_nb nb n s.t. G(c1) = G(c2) do
13:         R ← R ∪ {(c1, c2)}
14:     for each c1 ≡ sync_nb nb n, c2 ≡ sync_nb nb n s.t. G(c1) = G(c2) do
15:         R ← R ∪ {(c1, c2), (c2, c1)}
16:     ▷ Add edges from mbarrier arrives to mbarrier waits. ◁
17:     for each c1 ≡ arrive_mb sb, c2 ≡ wait_mb sb ph s.t. G(c1) = G(c2) do
18:         R ← R ∪ {(c1, c2)}
19:     R ← transitive closure of R
20:     ▷ Generalize Algorithm 1's check for all instructions that modify arrival counts. ◁
21:     for each (c1, c2) ∈ (Reg(b, g), Reg(b, g + 1)) do
22:         if c3; c2 ∉ T then
23:             return false
24:         if c3; c2 ∈ T ∧ (c1, c3) ∉ R ∧ c1 ≠ c3 then
25:             return false
26:     ▷ Now ensure each wait_mb cannot drift to another generation. ◁
27:     for each w ≡ wait_mb b p with G(w) = g do
28:         ▷ Waits on the first two generations (−1, 0) can't drift further backwards. ◁
29:         if g ≥ 1 then
30:             ▷ Repeat the standard checks from above. ◁
31:             if c3; w ∉ T then
32:                 return false
33:             else if c3; w ∈ T ∧ ∃ c ∈ Reg(b, g − 1) : (c, c3) ∉ R ∧ c ≠ c3 then
34:                 return false
35:         ▷ Check that a wait_mb can't drift too far forward. This check needs to only be performed when
              generation g + 1 completes. ◁
36:         if Reg(b, g + 1) ≠ arrival-count(b) and ∀ c+ ∈ Reg(b, g + 1), (w, c+) ∉ R then
37:             return false
38:     ▷ Duplicate barrier initializations are disallowed. ◁
39:     if a barrier b is initialized multiple times then
40:         return false
41:     ▷ There must be an edge between an initialization and all barrier uses. ◁
42:     for each ci ≡ init_mb b n and each user of b cb do
43:         if cp; cb ∉ T then
44:             return false
45:         else if cp; cb ∈ T ∧ (ci, cp) ∉ R ∧ ci ≠ cp then
46:             return false
47:     return true
```

complete. Note that edges are not added between **wait_mb** operations, as no relationship between their finish times is guaranteed.

3.3.4 Preciseness of R. The same general strategy as used in Section 2.3.2 is applicable again. We refer to the mechanization for the exact invariant maintenance arguments.

3.3.5 Soundness of Algorithm 2. We use the same general strategy as in Section 2.3.3. We will work through some cases in the maintenance of $\text{Conforms}(\tau, \tau')$ that require different reasoning than with named barriers. First, for point (2) of conformance, mbarriers state agreement only requires agreement over the arrival counts. Then, we require the parked waiters to be a subset of all **wait_mb** operations that observed the same generation. We'll now discuss several cases of the proof.

Case Mbarrier-Wait-Block: For (1), we have an instruction $c =$ **wait_mb** $b\ p$ that is observing k generations before it in τ'. We will follow the same strategy as before by showing that $k < g$ and $k > g$ are both contradictions, where $g = G(\tau)(c)$. By being in the wait-block rule, we know $k\%2 = p$.

Suppose $k < g$. This case proceeds in the same way as the other cases, due to the symmetric check for **wait_mb** operations.

Suppose $k > g$. Here, we cannot apply the same pigeonhole-based argument as previously, because **wait_mb** operations do not contribute to the arrival count of b. Instead, we leverage the fact that there must be at least one edge from the arrival set at $g + 1$ into c. Because $k\%2 = p$, k must be at least $g + 2$ in this case for the wait-block rule to fire. Therefore, all arrives at $g + 1$ must have completed. Because these arrives conform, and there is a happens-before edge from at least one of them into c, we have a contradiction.

The proof obligations for (2) and (3) hold directly.

Case Mbarrier-Wait-Pass: Again, for (1), we have an instruction $c =$ **wait_mb** $b\ p$ that observes k previous recyclings of b, and $G(\tau)(c) = g$. Based on the definition of G for **wait_mb** operations with mismatched phases, $G(\tau')(c) = k - 1$. We must show that $k = g + 1$.

Suppose that $k < g + 1$. Because $k\%2 \neq p$, $k \leq g - 1$. This case is a contradiction as there are edges from all instructions at $g - 1$ to c.

Suppose that $k > g + 1$. Similarly, $k \geq g + 3$. This case follows the same logic as the $k > g$ in the wait-block case.

(2) is upheld trivially.

For (3), this wait could only proceed because all previous arrives have completed, so (3) is upheld.

Case Mbarrier-Recycle: This case directly falls out of the modified conformance invariant for mbarriers.

3.3.6 Completeness of Algorithm 2. We will follow the same strategy as in Section 2.3.4, and argue why for a well-synchronized CTA T, Algorithm 2 cannot return false.

Case Wait-Check 1: The same strategy as Section 2.3.4 applies.

Case Wait-Check 2: The same strategy as Section 2.3.4 applies.

Case Wait-Check 3: Using a similar construction as in Section 2.3.4, if no edges of this category existed, we could construct a trace forcing the **wait_mb** to observe generation $g + 2$ instead.

Case Init-Check 1: A CTA with multiple initializations of the same barrier will error, and thus the CTA is not well-synchronized.

Case Init-Check 2: Using a similar construction as in Section 2.3.4, without this edge, we can construct a trace that causes the barrier operation to see an uninitialized barrier, resulting in an error, which means that T was not well-synchronized.

4 Loops

The second major extension of Weft in our work is extending Weft beyond only straight-line programs, and considering programs with *loops*. Standard approaches for verifying concurrent programs with loops involves running algorithms like Weft with a bounded unrolling of the target loop, and providing gaurantees for only the tested unrollings. Alternate approaches involve full model-checkers, which perform exhaustive searches over the potential state space of the loop to show that some predicate always holds. Our goal was to show that a Weft-like approach can yield both soundness and completeness gaurantees for loops where we can show that a loop is well-synchronized and race-free for any number of iterations. We show how to perform such an analysis by carefully choosing a bounded unrolling of the loop that allows us to prove that any number of iterations of the loop are well-synchronized. Our work currently focuses on Weft with named-barriers (the theorems in this section have been mechanized for the named-barrier language only), but we believe the results should lift to the mbarrier language without significant work.

Our core results focus on singly-nested loops, where each thread shares the same loop iteration domain, as commonly found in Tensor Core programs on modern GPUs. The core idea behind our approach is to show that well-synchronized program fragments can be sequentially composed in certain circumstances. We combine this composition with an observation that many barrier programs often only operate over a finite configuration space. Once this configuration space has been explored and proven to be well-synchronized, any number of repetitions of the configuration space is then also well-synchronized. We will build up this result over the course of several theorems.

4.1 The Angelic Lemma

The first major result is the observation that under certain circumstances, well-synchronized programs can be composed while maintaining well-synchronization. The power of this observation is that once a program is known to be well-synchronized, any chosen trace τ of the program is sufficient to reason about, since any other trace τ' is gauranteed to agree on its barrier generations. This argument is encapsulated in the following lemma, dubbed the "angelic lemma". This name comes from a family of program analyses called "angelic program analysis", which analyze a concurrent program by choosing a favorable execution, and then showing that this favorable execution is actually sufficient to reason about any possible execution.

Lemma 1. *Consider a CTA $T = A; B$, where T is the sequential composition of two CTAs A and B. Suppose that $WS(A) \wedge WS(A; B)$. Then, there must exists a trace τ of T where $\tau = \tau_A; \tau_B$, meaning that τ executes all instructions in A before executing any instructions in B.*

Proof. Because $\mathrm{WS}(A)$, $\exists\, \tau_A$ such that τ_A is a successful trace of A. AFSOC $\nexists\, \tau_B : \tau_A; \tau_B$ is not a successful trace of $A; B$. This would imply that τ_A is a sub-trace of $T = A; B$ that is not successful, which violates the assumption that $\mathrm{WS}(A; B)$. □

The power that the angelic lemma offers us is to choose convenient orderings of program fragments we know to be well-synchronized. We will use this capability to choose schedules of loop iterations that allow us to derive assertions about concurrent loop iterations as if they were sequentially composed.

4.2 Defining a Finite Configuration Space

An observation for many high-performance GPU programs that leverage barriers is that their loop iterations operate only over a finite configuration space. This configuration space is usually related to the depth of the software pipeline that is used by the program to hide the latency of memory

operations with compute. We now describe how to precisely define this configuration space for a class of barrier programs.

Consider the Weft language, but annotate a sequence of straightline instructions I as the body of a loop L. From a realistic input program source, this likely means some form of unrolling and canonicalization must be performed. We will show for programs of this form, there exists a computable number of iterations k where after k iterations, every barrier referenced in I is gauranteed to have been recycled at least once, or the program deadlocks.

The following assumptions are made over L:

- All threads t have the same iteration domain of their local fragment of L.
- For each barrier b referenced by L, only a constant arrival count is used at all instructions that reference b.

Let arrivers(b) be the number of arrivals across all threads in an iteration of L that arrive on b. Then, let the function f over barriers be defined as:

$$f(b) = \begin{cases} 1 & \text{if arrivers}(b) = \text{arrival-count}(b), \\ \frac{\text{arrival-count}(b)}{\text{arrivers}(b)} & \text{if arrivers}(b) \text{ divides arrival-count}(b), \\ 1 & \text{if arrival-count}(b) \text{ divides arrivers}(b), \\ \text{LCM}(\text{arrivers}(b), \text{arrival-count}(b)) & \text{otherwise} \end{cases}$$

Lemma 2. *Under the earlier stated assumptions, if $k = LCM(f(b_1), f(b_2), \ldots, f(b_n))$, for all barriers $b_1, b_2, \ldots, b_n$ referenced by L, then after each thread executes k iterations of L, all barriers referenced by L must advance their generations at least once and return to their original state, or the k iterations deadlock.*

Proof. Unroll L k times for each thread to yield a straightline program P for each thread. Simulate an execution of P for each thread. If P deadlocks, then the proof concludes.

If P does not deadlock, then after k iterations, every instruction in P has executed. Case analysis can be done on $f(b)$, but in each case, the execution of all instructions in P will have enough arrivals to force the advancement of the generation of each b, and execute enough arrives to return each barrier to its state upon entry to P. □

Corollary 1. *After k iterations of L, a barrier b referenced in L will have its generation increased by exactly $\frac{k \cdot \text{arrivers}(b)}{\text{arrival-count}(b)}$ times.*

To extend this result to mbarriers, the computation of k is the exact same as done with named barriers, except that the the result function f (or the definition of a new f_m for mbarriers) is doubled. Due to the extra phase bit in the barrier's state, the barrier must be advanced twice before it returns to its initial state. Therefore, $f_m(b) = 2 \cdot f(b)$, and the remaining computations complete in the same way.

4.3 Singly-Nested Loops

We can combine the angelic lemma and the computation of k to define an algorithm for checking whether any number of iterations of the loop body I is well-synchronized. We first consider loops that contain no instructions before or after the loop. So, the programs we want to model can be expressed as I^n, representing n sequential compositions of the straightline loop body I. Note that even though each I is sequentially composed, the concurrent operational semantics of Weft allows for instruction execution to float across iteration boundaries.

4.3.1 Core Lemmas. We first define several helper lemmas that illustrate the intuition as well as the proof strategies for reasoning about loops. The first helper lemma describes the structure of the generations G across multiple iterations of the loop.

Lemma 3. *For some n, assume that $\forall i \in [0, n]$, $WS(I_0^k; \ldots; I_i^k)$, where k is defined as above. Consider the program $I_0^k; \ldots; I_n^k; I_{n+1}^k$. Then, there exists an execution trace T of $I_0^k; \ldots; I_n^k; I_{n+1}^k$ that produces a generation mapping G_T such that $\forall i \in I^k, G_T(I_{n+1}^k(i)) = G_T(I_n^k(i)) + \frac{k \cdot arrivers(b)}{arrival\text{-}count(b)}$.*

Proof. Because $\forall i \in [0, n)$, $\mathrm{WS}(I_0^k; \ldots; I_i^k)$, there exists a trace T of instructions that runs $I_0^k; \ldots; I_n^k$ to completion with recorded generation mapping G_T.

Because each i-long batch of I^k is WS, T can be further constrained to a trace that first executes all instructions in I_0^k, then I_1^k and so-on. By Lemma 2, after each I^k instructions in T, the program state has reset to the same state as before I_0^k executed.

Because the state is the same as before I_n^k executed, and I_{n+1}^k is the same, the same sequence of steps may be taken to execute I_{n+1}^k to completion after all of $I_0^k; \ldots; I_n^k$ have executed and extend T.

This execution produces the same generations as the trace T produced for I_n^k, but offset by $\frac{k \cdot \text{arrivers}(b)}{\text{arrival-count}(b)}$ (Corollary 1). □

Next, we can leverage this lemma to also describe the structure of the happens-before relation R computed by Algorithm 1.

Lemma 4. *For some n, assume that $\forall i \in [0, n]$, $WS(I_0^k; \ldots; I_i^k)$, where k is defined as above. Consider the program $I_0^k; \ldots; I_n^k; I_{n+1}^k$. Then, there exists an execution trace T of $I_0^k; \ldots; I_n^k; I_{n+1}^k$ that produces an observed happens-before relation R_T such that*

(1) $\forall i_1, i_2 \in I^k, (I_n^k(i_1), I_n^k(i_2)) \in R_T \iff (I_{n+1}^k(i_1), I_{n+1}^k(i_2)) \in R_T$.
(2) $\forall i_1, i_2 \in I^k, (I_n^k(i_1), I_{n+1}^k(i_2)) \in R_T \iff (I_{n-1}^k(i_1), I_n^k(i_2)) \in R_T$.

Proof. Goal (1): Apply Lemma 3 to get the corresponding trace T, generation mapping G_T and happens-before relation R_T.

Consider the three cases that a pair of instructions (i_1, i_2) can be in relation R_T (before the transitive closure is applied), assuming that i_1 and i_2 are both from I_{n+1}^k.

If i_1 and i_2 are adjacent on the same thread, this will also be true for I_n^k.

If (i_1, i_2) is added to R to due arrives and syncs on the same generation, i_1 and i_2 have the same generation in I_{n-1}^k. By Corollary 1, i_1 and i_2 must have the same generations in I_n^k, and thus the same edge will be added.

Goal (2): The same line of reasoning as used in Goal (1) can be applied again, but this time the relative generations of instructions in I_{n-1}^k, I_n^k and I_{n+1}^k need to be considered. □

The next step is to use these lemmas in a larger helper lemma, which allows us to extend the well-synchronized property from a prefix of well-synchronized loop iterations to a newly added loop iteration.

Lemma 5. *Assume that for some n, $\forall i \in [0, n]$, $WS(I_0^k; \ldots; I_i^k)$, where k is defined as above. Then, it must be true that $WS(I_0^k; \ldots; I_n^k; I_{n+1}^k)$.*

Proof. Let P refer to the program $I_0^k; \ldots; I_n^k; I_{n+1}^k$

Because $\forall i \in [0, n]$, $\mathrm{WS}(I_0^k; \ldots; I_i^k)$, there exists a trace T of instructions that runs all of I_0^k, then all of I_1^k and so-on. This trace can then be extended to finally run all of I_{n+1}^k to completion.

From this of T, collect the generation mapping G and the happens-before relation R.

The goal of this proof is now to show that Algorithm 1 must return true on T, using the constructed G and R. In particular, we must show that neither of the cases where Algorithm 1 returns false can occur.

Consider each pair of instructions (c_1, c_2) considered by Algorithm 1 with generations $G(c_1) = g$ and $G(c_2) = g + 1$.

It can be shown that for two instructions with generations g and $g + 1$, these instructions may be at most 3 batches of I^k apart. A way to see this is to consider the spans of time where an instruction may have a particular generation within a batch. Each batch has a range of time where an instruction has generation $g-1$, the barrier gets recycled, and then the instruction has generation g. Visually this looks like $[g-1, \ldots, g]; [g, \ldots, g+1]; [g+1, \ldots, g+2]$, as each barrier must be recycled at least once in each batch of I^k. We can perform a large case analysis on which batch of I^k these instructions came from. The cases include when the instructions are both from the well-synchronized prefix, or they span from the well-synchronized prefix into the last batch I^k_{n+1}. In each case, by Lemma 3 and Lemma 4, it can be shown that because the same happens-before relationships occur in the well-synchronized prefix, they must also occur in the last batch, and thus Algorithm 1 must return true. □

4.3.2 Final Results. Using these helper lemmas, we can now state full theorems for reasoning about loops with Weft. In particular, only a constant number of invocations of Algorithm 1 are required to ensure that a loop is well-synchronized for any number of iterations.

Theorem 6. *If* $\forall\, i \in [0, 3 \cdot k]$, $WS(I^i)$, *then* $WS(I^n)$ *for all* n.

Proof. This follows from inductively applying Lemma 5. □

This theorem can be extended to loops that contain prologues and epilogues, where the structure of the loop can be represented as a CTA $P; I^n; E$. The extended theorem is as follows.

Theorem 7. *If* $WS(P) \wedge WS(P; I^k) \wedge \forall\, i \in [0, 3 \cdot k]$, $WS(P; I^i; E)$ *then* $WS(P; I^n; E)$ *for all* n.

Proof. This can be proved with a similar strategy as Theorem 6 and a variant of Lemma 5 that considers the effects of the prologue P and epilogue E. □

These results allow us to conclude in only polynomial time ($O(k \cdot n^3)$) that a loop is well-synchronized (and thus can be determined to be race-free) for any number of iterations. This is in strict contrast to prior approaches, which either require exponential time for exhaustive model-checking, or forgo precision by considering only bounded unrollings.

4.4 Limitations

While promising, the detailed approach has several limitations. First, it has an assumption of loop structure: we assume the loop body is straightline code, which may require non-trivial pre-processing of the source program to satisfy. The second limitation, which is more frustrating, appears in the premises of Theorem 7. Theorem 7 requires $\mathrm{WS}(P) \wedge \mathrm{WS}(P; I^k)$, which seems reasonable, but limits the set of programs our results can be applied to. Consider the following programs:

t_1

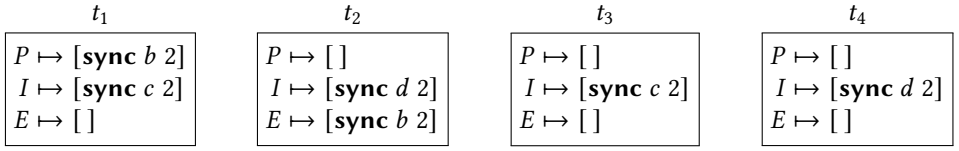
$P \mapsto$ [**sync** b 2]
$I \mapsto$ [ ]
$E \mapsto$ [ ]

t_2

$P \mapsto$ [ ]
$I \mapsto$ [**sync** c 2]
$E \mapsto$ [**sync** b 2]

t_3

$P \mapsto$ [ ]
$I \mapsto$ [**sync** c 2]
$E \mapsto$ [ ]



These programs have the interesting property that the prologues alone are not well-synchronized, but the entire program is well-synchronized! New techniques will be required to lift the requirement of the well-synchronized prologue from our results, as the proofs require the key reasoning step of running a subset of the program to completion.

5 Future Work

There are several interesting directions of future work that should be considered to both complete this work and explore new areas.

5.0.1 Lifting Limitations. Both identified limitations in the prior section should be targeted. The more significant limitation to be lifted is the requirement that the prologue be well-synchronized. It's currently unclear how to perform this analysis without a completely separate proof approach. Perhaps approaches based on thread-partitioning can be explored that maintain a similar style as the angelic execution approach.

5.0.2 More Complex Looping Structures. The approach described in this work targets singly-nested loops. Future extensions to this work should target other kinds of looping structures as well, which are commonly found in end-user programs. Structures of interest include:

- Sequential composition of multiple loops ($P_1; I_1^n; E_1; P_2; I_2^m; E_2; \ldots$).
- Nested loops.
- Programs where some threads don't have a loop while other threads do.

It's not entirely clear to us yet how to handle these cases. Preliminary investigation of sequential composition of loops indicate that a similar strategy as used in Theorem 7 can be used, but the strategy also has the same current limitations. We also performed an initial foray into nested loops following a similar strategy, but found that it was difficult to generalize past a doubly-nested loop. A strategy similar to hierarchical reduction from the software-pipelining literature would be desired to handle arbitrary amounts of nesting. However, most real-world programs do not contain more than a doubly-nested main loop.

5.0.3 CLC. NVIDIA's Cluster Launch Control (CLC) primitives are new primitives introduced in the Blackwell architecture that allow for CTAs to cancel other pending CTAs and "steal" their work. CLC is commonly used in high-performance Tensor Core computations to to implement dynamic tile schedulers that steal pending tiles for the current CTA to execute, avoiding re-initialization overhead of shared memory and barriers. These dynamic schedulers are implemented with `while` loops that poll a CLC-configured barrier until triggers. Identification of this program structure for analysis with our loop-based approach will likely be needed to make progress, as treating the CLC components as opaque will require the analyses to be too conservative. To begin, investigating bounded unrolling of the outer loop of a CLC-based dynamic scheduler will likely allow for our work on mbarrier verification to be applied to real-world Tensor Core kernels with dynamic schedulers.

5.0.4 Implementation. While we have a protoype implementation of these ideas here, the prototype operates over programs in a simple DSL that matches the formalism described in this paper. Further work must be done to consume real programs written by end users to verify their correctness. It's unclear what the best input language to consume is; we have had discussions about taking

CuTeDSL programs or directly lifting PTX into the Weft source language. Both have tradeoffs: CuTeDSL programs have richer structure than PTX and have higher-level mbarrier operations, while PTX programs are more widely accessible but require lifting into a form that Weft could process.

References

[1] Rahul Sharma, Michael Bauer, and Alex Aiken. 2015. Verification of producer-consumer synchronization in GPU programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 88–98. doi:10.1145/2737924.2737962