Verification of Producer-Consumer Synchronization in GPU Programs

Rahul Sharma
Stanford University
sharmar@cs.stanford.edu

Michael Bauer
NVIDIA Research
mbauer@nvidia.com

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Abstract

Previous efforts to formally verify code written for GPUs have focused solely on kernels written within the traditional data-parallel GPU programming model. No previous work has considered the higher performance, but more complex, *warp-specialized* kernels based on producer-consumer *named barriers* available on current hardware. In this work we present the first formal operational semantics for named barriers and define what it means for a warp-specialized kernel to be correct. We give algorithms for verifying the correctness of warp-specialized kernels and prove that they are both sound and complete for the most common class of warp-specialized programs. We also present WEFT, a verification tool for checking warp-specialized code. Using WEFT, we discover several non-trivial bugs in production warp-specialized kernels.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.1.3 [*Programming Techniques*]: Parallel Programming

Keywords Verification; GPUs; data races; synchronization; deadlock; barrier recycling; warp specialization; named barriers

1. Introduction

GPUs are now an established general purpose computing platform for applications that require significant computational resources and memory bandwidth. While the potential gains of GPUs in performance and energy efficiency are high, writing correct GPU programs is still a challenging task for many programmers. The interaction of a complex GPU memory hierarchy, including different on-chip software and hardware managed caches, coupled with the large number of threads to consider, makes it easy to introduce data races and other correctness bugs into GPU kernels.

There have been several previous attempts at writing tools capable of verifying the correctness of GPU kernels [5, 8, 11, 12, 18–21]. While each of these tools has attacked the problem of verifying GPU kernels in a different way, they have all assumed the standard data-parallel GPU programming model supported by CUDA[22] and OpenCL[15]. In this model, kernels commonly execute using a streaming paradigm: load data on-chip, perform a computation on the data, and write data back off-chip. To coordinate the execution of these phases, a barrier is used to synchronize *all* the threads in a *threadblock*(CUDA)/*workgroup*(OpenCL). While a threadblock-wide barrier is sufficient for streaming data-parallel kernels in which all threads perform roughly the same computation, it is limiting for kernels in which threads within the same threadblock perform different computations.

Recently, programming systems such as CudaDMA[6] and Singe[7] have demonstrated the viability of *warp specialization* as an alternative to the standard data-parallel programming model for targeting GPUs. Warp-specialized kernels assign different computations to *warps* (groups of 32 threads) within the same threadblock in order to achieve important performance goals (e.g. maximizing memory bandwidth in CudaDMA, or fitting extremely large working sets on-chip in Singe). To perform synchronization between different warps, warp-specialized kernels use the producer-consumer *named barriers* available in PTX[1] on NVIDIA GPUs. Named barriers are not currently exposed in the CUDA programming model and are only available via the use of inline PTX assembly code. Named barriers are implemented directly in hardware and support a richer set of synchronization patterns than can be achieved with the standard threadblock-wide barrier used by CUDA and OpenCL. Specifically, named barriers allow warp-specialized kernels to encode producer-consumer relationships between arbitrary subsets of warps in a threadblock. Importantly, producers do not block on a named barrier, and can continue executing after arriving on a named barrier.

While named barriers allow warp-specialized kernels to achieve up to 4X performance improvements in some cases over optimized data-parallel kernels[6, 7], they have considerably more complex semantics which are challenging to verify. Specifically, there are three important properties to check for warp-specialized kernels.

- Deadlock Freedom: checking that the use of named barriers does not result in deadlocks.
- Safe Barrier Recycling: Named barriers are a limited physical resource and it is important to check that IDs of named barriers are properly re-used.
- Race Freedom: checking that shared memory accesses synchronized by named barriers are race free.

To the best of our knowledge, no current GPU verification tools are capable of checking code containing named barriers. In this work, we present the necessary theoretical framework, algorithms, and implementation strategy for constructing a verifier capable of efficiently checking the above correctness criteria for production warp-specialized code.

Using our techniques, we develop WEFT[1], a verifier which is sound, complete, and efficient. Soundness ensures that developers never have to write tests to check the three properties mentioned

[1] Available at `https://github.com/lightsighter/Weft`

PLDI'15, June 13–17, 2015, Portland, OR, USA.
Copyright © 2015 ACM 978-1-4503-3468-6/15/06...$15.00.
http://dx.doi.org/10.1145/2737924.2737962

previously; in fact, WEFT has discovered latent bugs that have persisted for many years. Completeness guarantees that all violations reported by WEFT are actual bugs; there are no false positives. Finally, WEFT is efficient enough to be practical for verifying real codes; WEFT runs on complex, warp-specialized kernels consisting of thousands of lines of code and with billions of potential races within a few minutes.

The rest of this paper is organized as follows. In Section 2, we provide the necessary background for understanding named barriers and the construction of warp-specialized kernels. We also provide a concrete example of an interesting warp-specialized kernel from a real application that demonstrates the need for formal verification. Each of the remaining sections describe one of our primary technical contributions.

- We introduce a core language for warp-specialized kernels that contain named barriers. We give a formal operational semantics for this language and show how it models the necessary properties for checking correctness of warp-specialized kernels (Section 3). These semantics are a proposed formalization of the English description of named barriers in the PTX manual [1] and can be re-used by future tools that attempt to verify programs that use named barriers.
- We formally define the correctness properties for warp-specialized kernels and provide algorithms for verifying these properties. Our algorithms (Section 4) are sound (do not miss any errors), complete (all reported errors are true errors), and efficient.
- We describe our implementation of WEFT, a verification tool for checking warp-specialized code containing named barriers. We describe the important optimizations that permit WEFT to scale to kernels that execute hundreds of millions of program statements. Using WEFT we check the correctness of all warp-specialized kernels of which we are aware and discover several non-trivial bugs (Section 5).

Section 6 covers related work and Section 7 concludes.

2. Background and Motivation

In this section we briefly cover the necessary details for understanding GPU programming (Section 2.1) and warp specialization (Section 2.2). We then introduce a motivating warp-specialized kernel that illustrates the need for verification of code using named barriers (Section 2.3).

2.1 GPU Architecture and Programming

While there are several different frameworks currently available for programming GPUs, they all provide variations on the same data-parallel programming model. For the rest of this work, we use CUDA as a proxy for this model as it is the only interface that supports the named barrier primitives necessary for constructing warp-specialized kernels.

When a CUDA kernel is launched on a GPU, a collection of threadblocks or *cooperative thread arrays* (CTAs for the remainder of this paper) is created to execute the program. Each CTA is composed of up to 1024 threads. The hardware inside of the GPU dynamically assigns CTAs to one of the *streaming multiprocessors* (SMs) inside of the GPU. To communicate between threads within the CTA, CUDA provides an on-chip software-managed *shared memory* visible to all the threads within the same CTA. To coordinate access to shared memory, CUDA supports the `syncthreads` primitive, which performs a CTA-wide blocking barrier. Synchronizing between threads within the same CTA is the only synchronization supported by CUDA because threads within the same CTA are the only ones guaranteed to be executing concurrently.

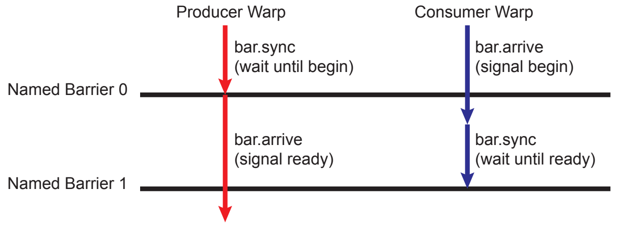

Figure 1. Producer-consumer named barriers example.

While logically CTAs represent a flat collection of threads, in practice the threads of a CTA are organized into groups of 32, referred to as *warps*. The hardware within an SM makes scheduling decisions at the granularity of warps. When a warp is scheduled, all threads within the warp execute the same instruction. In the case of a branch instruction, the SM first executes the warp for the not taken branch with the threads that took the branch masked off. The SM then executes the taken branch with the complementary set of threads in the warp masked off. The resulting serialized execution of different branches within a warp is referred to as *branch divergence* and is a common source of performance degradation. The crucial insight for enabling warp-specialized programs is that there is no penalty if threads in different warps diverge. As long as all threads within a warp continue to execute the same instruction stream, there is no performance degradation.

2.2 Warp Specialization and Named Barriers

A warp-specialized kernel is one in which individual warps are assigned different computations via control divergence contingent upon warp IDs. There are a number of advantages to specializing warps. For example, CudaDMA[6] specializes warps into *compute* and *DMA* warps to optimize the loading of data into shared memory. By separating memory operations from arithmetic, both instruction streams can be better optimized by the CUDA compiler. Alternatively, the Singe compiler[7] specializes warps to handle the mapping of fine-grained static dataflow graphs onto CTAs. Consequently, Singe can leverage task parallelism on GPUs in the absence of significant data-parallelism, and fit very large working sets into on-chip memories by blocking data for the GPU register file.

Similar to data-parallel CUDA kernels, warp-specialized kernels communicate through shared memory. However, unlike data-parallel kernels, communication in warp-specialized kernels is asymmetric, with one or more warps acting as producers, and one or more other warps acting as consumers. In CudaDMA, DMA warps act as producers while compute warps act as consumers. For Singe kernels, dataflow edges between operations assigned to different warps define producer-consumer relationships between pairs of warps. To handle producer-consumer synchronization, NVIDIA GPUs support hardware *named barriers* in PTX[1]. PTX provides two instructions for arriving at a named barrier: `sync` and `arrive`. A `sync` instruction causes the threads in a warp arriving at the barrier to block until the barrier completes. Alternatively, an `arrive` instruction allows a warp to register arrival at a barrier and immediately continue executing without blocking. Both `sync` and `arrive` instructions require the total number of threads participating in the barrier to be specified. Importantly, the number of participants can be less than the total number of threads in the CTA.

Using both the `sync` and `arrive` instructions (via inline PTX assembly), CUDA applications can encode producer-consumer synchronization. Figure 1 demonstrates the usage of two named barriers to encode a producer-consumer relationship between a pair of warps. A producer warp initially blocks on named barrier **0** using a `sync` instruction to wait until it is safe to write data into shared memory. At some point the consumer warp signals that it is

```
__global__ void __launch_bounds__(64,1) example_deadlock(void) {
  assert(warpSize == 32);
  assert(blockDim.x == 64);
  int warp_id = threadIdx.x / 32;
  if (warp_id == 0) {
  // bar.sync (barrier name), (participants);
    asm volatile(”bar.sync 0, 64;”);
    asm volatile(”bar.arrive 1, 64;”);
  } else {
    asm volatile(”bar.sync 1, 64;”);
    asm volatile(”bar.arrive 0, 64;”);
  }
}
```

Listing 1. Example of deadlock using named barriers.

safe to write by performing an `arrive` instruction on named barrier **0**. Since it issued a non-blocking `arrive` instruction, the consumer warp can perform additional work while waiting for the producer warp to generate the data. Eventually, the consumer warp blocks on named barrier **1** waiting for the data to be ready. The producer warp signals that the data is ready by arriving on named barrier **1** with a non-blocking `arrive` so that it can continue executing.

Each SM on current NVIDIA GPUs contains 16 physical named barriers numbered 0 through 15. Once a named barrier completes, it is immediately re-initialized so that it can be used again. We refer to each use of a named barrier as a *generation* of the named barrier. Different generations of a named barrier can have different numbers of participants.

The use of named barriers can create many interesting failure modes for kernels that will not normally arise within the standard CUDA programming model. For example, named barriers introduce the possibility of deadlock. Listing 1 shows a simple warp-specialized kernel which deadlocks for all CTAs. Each of the two warps in this example block waiting on the other warp to arrive on a different named barrier, resulting in a cyclic synchronization dependence. It is important to note that deadlock is a problem specific to warp-specialized code. It is impossible to create such circular waits using the traditional `syncthreads` primitive because all invocations map to named barrier 0.[2]

2.3 Motivating Kernel

To motivate the need for verification of warp-specialized kernels, we give an example from the Heptane chemistry kernel emitted by the Singe DSL compiler[7]. This kernel is currently deployed in a production version of the combustion simulation S3D[9], which is being used for doing advanced energy research on Titan, the number two supercomputer in the world[26]. Figure 2 illustrates how part of the computation represented as a static dataflow graph is mapped onto four warps. All computations (nodes in the dataflow graph) represent warp-wide data-parallel operations consisting of straight-line code. Where dataflow edges cross warp boundaries, data must be communicated through shared memory. All cross-warp dataflow edges (labeled and shown in red) are assigned a named barrier to use when synchronizing access to shared memory for communicating data.

There are two important properties to be observed regarding this example kernel. First, notice the re-use of named barrier **2** in Figure 2. To avoid conflicting arrivals at named barrier **2** from participants of different generations, a *happens-before relationship*[16] must be established between the completion of the previous generation of a named barrier and all participants of the next generation. From Figure 2, we know barrier **2** completes before operation D is run. The happens-before relationships established by the paths $D \rightarrow E \rightarrow F \rightarrow I$ and $D \rightarrow E \rightarrow G$, using named barriers **1** and **3**, ensure the participating warps in the next generation of barrier **2** (warps 1 and 2) both have a happens-before relationship with the previous generation of barrier **2**; therefore barrier **2** can be safely recycled. Note that the path $A \rightarrow B \rightarrow F \rightarrow I$ is insufficient for establishing a happens-before relationship because warp 0 performs a non-blocking arrive on the first generation of named barrier **2** and therefore cannot guarantee that the barrier actually completed. For the Heptane kernel to run correctly, all barriers must successfully execute and be properly re-used. Informally, we refer to a kernel that meets these criteria as *well-synchronized* (we give a formal definition of well-synchronized in Section 4). In the case of the Heptane chemistry kernel, which consists of more than 10K lines of CUDA code, checking for the well-synchronized property by hand is impractical.

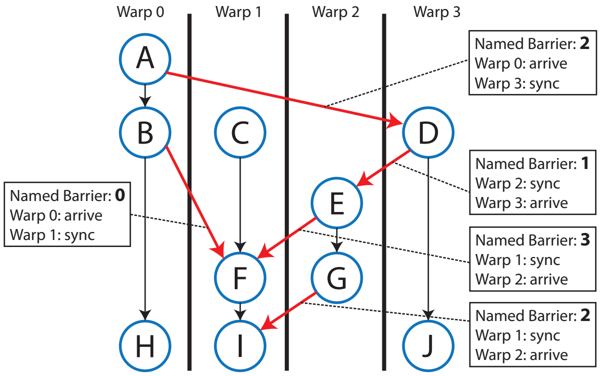

Figure 2. Motivating warp specialization example.

The second important property of the Singe kernel is that the synchronization pattern is completely static. While there is significant parallelism in the kernel, the use of named barriers constrains the synchronization so that it is both deterministic and known at compile-time. Furthermore, each thread executes straight-line code with no dynamic branches or loops. These properties are important as the undecidability result of [25] shows that it is impossible to obtain complete solutions in the presence of arbitrary control flow and synchronization. In practice, we found that all warp-specialized kernels we considered consisted entirely of code with only statically-analyzable branches, loops, and synchronization patterns. While this may seem unusual for general purpose code, it is common for many GPU kernels due to the high cost of dynamic branching associated with the in-order instruction pipelines of current GPUs.[3] We also found that all accesses to global memory and shared memory could be completely understood statically as functions of CTA and thread ID. Consequently, the operational semantics we present in Section 3 are for kernels with statically analyzable control flow and data flow, which is sufficient for handling all of the warp-specialized code known to us.

3. Operational Semantics

Before describing our formal language, we first scope the domain of our problem. We are interested in formally checking for deadlock-freedom, proper named barrier recycling, and shared memory data race freedom within individual CTAs. We make no claim of handling inter-CTA synchronization[4] used by persistent *uber-kernels* such as OptiX[24]. We also do not consider software-

[2] Using `syncthreads` inside of warp-divergent conditional statements is illegal in the CUDA programming model and has undefined semantics, but in practice can also result in deadlock on some architectures.

[3] It is also simple to extend our results to programs with loops and dynamic branches if these constructs do not contain synchronization operations.

[4] Which is not officially supported by the CUDA programming model.

level intra-CTA synchronization which can be constructed with atomic primitives (but are at least an order of magnitude slower than named barriers). We view the verification of atomics as an orthogonal problem to the verification of named barrier usage. We rely on a symmetry assumption that all CTAs execute the same program (albeit with different input data) and therefore we only need to model and verify a single CTA to establish correctness. The validation is limited to the accesses to a GPU's shared memory, which has a well-defined semantics due to the software-managed nature of the cache. We do not attempt to reason about data passed between threads through global memory, both because of the weak semantics [3] and because global memory is not used for communication in any of the warp-specialized kernels of which we are aware.

3.1 Syntax

A GPU program has an arbitrary number of non-interfering CTAs. Each CTA has N (typically between 32 and 1024) threads that can synchronize for access to shared memory. We denote a thread by P and a CTA by T. We use $P_1 \; || P_2 \; || \dots \; || \; P_N$ to denote a CTA with threads $P_1, P_2, \dots, P_N$. Variables in shared memory locations are denoted by g. For simplicity, we assume that all variables occupy 64 bits. We consider abstract *thread programs* for each thread, where everything has been abstracted away except the instructions required to reason about synchronization and shared memory accesses. Each thread program has a separate thread identifier denoted by *id*. We use i to range over thread identifiers. The hardware provides B (typically 16) named barriers and b refers to a specific barrier name.

A thread program has the following grammar:

$$\begin{array}{rcl} P & ::= & \texttt{return} \mid c; P \\ c & ::= & \texttt{read}\; g \mid \texttt{write}\; g \\ & & \mid \texttt{arrive}\; b\; n \mid \texttt{sync}\; b\; n \end{array}$$

A thread program is a sequence of *commands* (straight line code). In each command c a thread can either read/write from a shared memory location or perform a synchronization operation. Blocking synchronization operations are performed by `sync` and non-blocking operations by `arrive`. The read and write commands are treated as no-ops in this section. They are only useful for detecting data races and play no role in the semantics of named barriers. For *synchronization commands* (`sync` and `arrive`), the first argument b represents the name (ID) of a named barrier, and the second argument n represents the expected number of threads to register at this generation of the barrier. Successive commands are separated by a semicolon. A thread terminates by executing a `return`.

In this syntax, the standard barrier `syncthreads` is expressed as `sync` $0\; N$: a sync across all threads in a CTA on barrier 0. 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. We say that a program point η and command c_η *correspond* if η is the program point just after c_η. The first program point of thread i is denoted by η_I^i and the last by η_F^i. We omit the superscripts when the thread is clear from the context.

Our semantics does not assume nor require *warp-synchronous* execution. Warp-synchronous execution is the assumption that all threads within a warp will execute in lock-step. This assumption has traditionally held for past GPU architectures, but it is not standardized and may be invalidated by future designs. We can easily model warp-synchronous execution in our language by inserting a `sync` command across all threads in a warp after every original command in a program, thereby ensuring that the threads within a warp execute in lockstep.

3.2 State

We first define a *state* s of a CTA. It consists of the following:

1. An *enabled map* $\mathcal{E}$ that maps thread identifiers to booleans signifying whether the thread is *enabled* or not. Threads are *disabled* when they block on a barrier.
2. A *barrier map* $\mathcal{B}$ that maps barrier names to a triple consisting of a list $\mathcal{I}$ of threads that have synced at the barrier, a list $\mathcal{A}$ of threads that have arrived at the barrier, and the *thread count*, describing the number of threads the barrier is expecting to register if it is configured.

The thread count is configured by the first thread that reaches a barrier, hence the need for all synchronization commands to specify the expected number of participants. The thread count of unconfigured barriers is denoted by $\perp$. An empty list of thread identifiers is denoted by $[]$ and "::" adds a thread to a list. The number of elements in a list $\mathcal{L}$ is denoted by $|\mathcal{L}|$.

For a map A, we use $A[x/y]$ to denote the map that agrees with A on all inputs except y and maps y to x. Similarly, $A[x/Y]$ denotes a map that agrees with A on all inputs that are not in Y and maps all $y \in Y$ to x. The function $ite(e_1, e_2, e_3)$ stands for " if e_1 then e_2 else e_3".

The initial state I has $\forall i.\mathcal{E}(i) = \textbf{true}$ and $\forall b.\mathcal{B}(b) = ([], [], \perp)$. Plainly stated: all threads are enabled and ready to execute, no thread has registered at any barrier, and all barriers are unconfigured. We use *done* to denote a CTA with no more commands to execute.

3.3 Semantics

We describe the small step operational semantics of a CTA T with threads $P_1, \dots, P_N$ executing in parallel. Recall that a state s has two components $\mathcal{E}$ and $\mathcal{B}$. The rules have the following form:

$$\mathcal{E}, \mathcal{B}, T \rightsquigarrow \mathcal{E}', \mathcal{B}', T'$$

$$\mathcal{E}, \mathcal{B}, i, P \rightsquigarrow \mathcal{E}', \mathcal{B}', i, P'$$

For the two kinds of rules, a CTA/thread program executing in some state takes a single step, which results in a new CTA/thread program and a new state. Here i denotes the thread ID of the thread program (equivalent to `threadIdx.x` for one dimensional CTAs, which are common for warp-specialized kernels). We now discuss the operational semantics rules for each of the commands starting with the execution of a CTA.

$$\frac{\begin{array}{c}\forall b.\Big(\mathcal{B}(b) = ([], [], \perp) \vee \big(\mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \wedge |\mathcal{I}| + |\mathcal{A}| < n\big)\Big) \\ \mathcal{E}, \mathcal{B}, i, P_i \rightsquigarrow \mathcal{E}', \mathcal{B}', i, P_i'\end{array}}{\mathcal{E}, \mathcal{B}, P_1 || \dots || P_i || \dots || P_N \rightsquigarrow \mathcal{E}', \mathcal{B}', P_1 || \dots || P_i' || \dots || P_N}$$

The first antecedent says that for all barriers b, either b is unconfigured or needs to register more threads. In this situation, we nondeterministically choose a thread in the CTA and execute it for one step. If we have N thread programs executing concurrently, any one of these can make an evaluation step from P_i to P_i' with some side-effects on the state.

Next, if some barrier has registered the required number of threads, then wake up all the threads blocked at the barrier and *recycle* the barrier.

$$\frac{\begin{array}{c}\mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \quad |\mathcal{I}| + |\mathcal{A}| = n \\ T = P_1 \; || \; \dots \; || \; P_N \quad \forall i.P_i = ite(i \in \mathcal{I}, \texttt{sync}\; b\; n; P_i', P_i') \\ T' = P_1' \; || \; \dots \; || \; P_N' \quad \mathcal{E}' = \mathcal{E}[\textbf{true}/\mathcal{I}]\end{array}}{\mathcal{E}, \mathcal{B}, T \rightsquigarrow \mathcal{E}', \mathcal{B}[([], [], \perp)/b], T'}$$

If the correct number of threads (n) have registered at b, then enable all threads that were blocked on b (contained in $\mathcal{I}$), change b to an unconfigured state, and update the control of all threads in $\mathcal{I}$. This rule recycles a barrier by returning it to an unconfigured state.

The execution terminates when all threads execute a `return`.

$$\frac{}{\mathcal{E}, \mathcal{B}, \mathtt{return} \mid\mid \ldots \mid\mid \mathtt{return} \rightsquigarrow \mathcal{E}, \mathcal{B}, \mathit{done}}$$

The hardware does not care about the state of the barriers when the CTA exits as the barriers are reset before starting a new CTA.

The first synchronization operation configures the barrier and determines the number of threads required.

$$\frac{\mathcal{B}(b) = ([], [], \bot) \qquad \mathcal{B}' = \mathcal{B}[([], [] :: \mathit{id}, n)/b]}{\mathcal{E}, \mathcal{B}, \mathit{id}, \mathtt{arrive}\ b\ n; c \rightsquigarrow \mathcal{E}, \mathcal{B}', \mathit{id}, c}$$

The arriving thread configures the barrier with the thread count (n) and gets added to the list of arrives. Programs with an invalid thread count (more than the maximum threads permitted in a CTA[5]) can be rejected by a preprocessing phase before execution begins.

If the first thread to register with a barrier is a `sync`, the semantics are similar to `arrive`. The thread updates the enabled map and is added to the list of blocked threads.

$$\frac{\begin{array}{c}\mathcal{E}(\mathit{id}) = \mathbf{true} \quad \mathcal{E}' = \mathcal{E}[\mathbf{false}/\mathit{id}] \\ \mathcal{B}(b) = ([], [], \bot) \quad \mathcal{B}' = \mathcal{B}[([] :: \mathit{id}, [], n)/b]\end{array}}{\mathcal{E}, \mathcal{B}, \mathit{id}, \mathtt{sync}\ b\ n; c \rightsquigarrow \mathcal{E}', \mathcal{B}', \mathit{id}, \mathtt{sync}\ b\ n; c}$$

On executing a non-blocking arrive at the barrier the control and the barrier map are updated.

$$\frac{\mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \quad 0 < |\mathcal{I}| + |\mathcal{A}| < n}{\mathcal{E}, \mathcal{B}, \mathit{id}, \mathtt{arrive}\ b\ n; c \rightsquigarrow \mathcal{E}, \mathcal{B}[(\mathcal{I}, \mathcal{A} :: \mathit{id}, n)/b], \mathit{id}, c}$$

On reaching a `sync`, the operation is slightly different from `arrive`: the thread is disabled and gets added to the list of blocked threads. The control remains unchanged.

$$\frac{\begin{array}{c}\mathcal{E}(\mathit{id}) = \mathbf{true} \quad \mathcal{E}' = \mathcal{E}[\mathbf{false}/\mathit{id}] \quad \mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \\ \mathcal{B}' = \mathcal{B}[(\mathcal{I} :: \mathit{id}, \mathcal{A}, n)/b] \quad 0 < |\mathcal{I}| + |\mathcal{A}| < n\end{array}}{\mathcal{E}, \mathcal{B}, \mathit{id}, \mathtt{sync}\ b\ n; c \rightsquigarrow \mathcal{E}', \mathcal{B}', \mathit{id}, \mathtt{sync}\ b\ n; c}$$

Note that for each generation of a named barrier, the expected number of participants must match for each thread program that registers with the named barrier.

If too many threads reach a barrier then transition to the error state `err`:

$$\frac{\mathcal{E}(\mathit{id}) = \mathbf{true} \quad \mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \quad |\mathcal{I}| + |\mathcal{A}| = n}{\mathcal{E}, \mathcal{B}, \mathit{id}, \mathtt{sync}\ b\ n; c \rightsquigarrow \mathtt{err}, \mathit{id}, \mathtt{sync}\ b\ n; c}$$

When the barrier has already registered the number of threads it was supposed to register then any subsequent attempts to register causes the execution to go to the error state. Also, if there is a mismatch on thread count then transition to the error state.

$$\frac{\mathcal{E}(\mathit{id}) = \mathbf{true} \quad \mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \quad n \neq m}{\mathcal{E}, \mathcal{B}, \mathit{id}, \mathtt{sync}\ b\ m; c \rightsquigarrow \mathtt{err}, \mathit{id}, \mathtt{sync}\ b\ m; c}$$

The error productions for `arrive` (not shown) are identical.

Finally, if any thread produces an error then the execution of the CTA terminates in an error state `err`.

$$\frac{\mathcal{E}, \mathcal{B}, i, P_i \rightsquigarrow \mathtt{err}, i, P_i}{\mathcal{E}, \mathcal{B}, P_1 \mid\mid \ldots \mid\mid P_i \mid\mid \ldots \mid\mid P_N \rightsquigarrow \mathtt{err}, P_1 \mid\mid \ldots \mid\mid P_i \mid\mid \ldots \mid\mid P_N}$$

These error productions ensure that an execution either reaches *done*, goes to `err`, or deadlocks. No other outcome is possible.

4. Algorithm

Our approach to verifying the correctness of a CTA involves guessing happens-before relationships from a concrete execution, performing a static analysis to ensure that the guessed relationships hold for all possible executions, and using the relationships to prove the important correctness properties. We begin by providing formal definitions for the correctness properties that we intend to verify.

4.1 Preliminaries

Recall that a state s has two components $\mathcal{E}$ and $\mathcal{B}$. We use (s, T) to abbreviate $\mathcal{E}, \mathcal{B}, T$. A *configuration* C is a pair of a state s and a CTA T. First we need to define execution traces.

Definition 1. *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.

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

As is standard, $C \rightsquigarrow C'$ denotes that C evaluates to C' in one step; $C \rightsquigarrow^* C'$ denotes that C evaluates to C' in zero or more steps, and $C \rightsquigarrow^m C'$ denotes that C evaluates to C' in m steps.

We also need a notion of time t which says at what step a command is executed in a trace. Time is needed to reason about the order in which commands are executed. 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* $t(\tau, \eta^i) = n$ *if* τ *has a prefix* $(s, T) \rightsquigarrow^{n-1} (s_1, P_1^{(1)} \mid\mid \ldots \mid\mid P_i^{(1)} \mid\mid \ldots \mid\mid P_N^{(1)}) \rightsquigarrow (s_2, P_1^{(2)} \mid\mid \ldots \mid\mid P_i^{(2)} \mid\mid \ldots \mid\mid P_N^{(2)})$ *and* $P_i^{(1)} = c_{\eta^i}; P_i^{(2)}$.

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

Definition 4. *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 `sync` commands that synchronize together are included in the happens-before relation R.

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

Definition 5. $\mathit{Gen}(\tau)(c_\eta) = n$ *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.*

Definition 6. *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* $\mathit{Gen}(\tau_1)(c) = \mathit{Gen}(\tau_2)(c) \neq 0$.

This definition needs to be suitably generalized for arbitrary starting states.

Definition 7. *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* $\mathit{Gen}(\tau_1)(c) = \mathit{Gen}(\tau_2)(c) \neq 0$.

[5] Technically, our semantics also do not permit CTAs with only 1 thread, but this class of programs trivially satisfy our correctness criteria.

```
asm volatile("bar.sync 0, 64;");
if (warp_id == 0) {
  g[lane_id] = w;
  asm volatile("bar.arrive 1, 64;");
} else {
  asm volatile("bar.sync 1, 64;");
  x = g[lane_id];
}
asm volatile("bar.sync 0, 64;");
if (warp_id == 0) {
  asm volatile("bar.sync 1, 64;");
   y = g[lane_id];
} else {
   g[lane_id] = z;
   asm volatile("bar.arrive 1, 64;");
}
```

Listing 2. CUDA snippet for the working example.

```
P                         Q
sync 0 64; (1)            sync 0 64; (1)
write g_0;                sync 1 64; (1)
arrive 1 64; (1)          read g_0;
sync 0 64; (2)            sync 0 64; (2)
sync 1 64; (2)            write g_0;
read g_0                  arrive 1 64 (2)
return                    return
```

Figure 3. Thread programs generated from Listing 2; P has thread ID 0 and Q has 32.

Note that the non-zero check ensures that no trace of a well-synchronized configuration can deadlock or go to an error. Therefore a check for well synchronization subsumes both safe barrier recycling and deadlock freedom. It also ensures that the synchronization behavior is deterministic: in all traces the same commands synchronize together.

Next we define race freedom.

Definition 8. *A well-synchronized CTA T is data race free if for all traces τ_1 and τ_2 starting from (I, T), if $t(\tau_1, \eta_1) < t(\tau_1, \eta_2)$, $t(\tau_2, \eta_1) > t(\tau_2, \eta_2)$, and c_{η_1} and c_{η_2} access the same shared variable g, then c_{η_1} and c_{η_2} are both* `read`.

If two commands accessing the same shared variable g do not have a happens-before relationship between them, then they must both be reads, otherwise there is a data race.

Finally, we have the standard definitions of soundness and completeness.

Definition 9. *An algorithm $\mathcal{D}$ is sound for property Ψ if for all CTA T, $\mathcal{D}(T) \Rightarrow \Psi(T)$.*

Definition 10. *An algorithm $\mathcal{D}$ is complete for property Ψ if for all CTA T, $\neg\mathcal{D}(T) \Rightarrow \neg\Psi(T)$.*

4.2 Property Checking

We describe our algorithms for checking if a kernel is well-synchronized and data race free using the example in Listing 2. The listing shows a CUDA program snippet for a kernel with 2 warps (64 threads). Figure 3 shows the thread programs associated with thread ID 0 and thread ID 32, denoted by P and Q respectively. (We discuss the translation from a CUDA program to a thread program in Section 5.1.) Both P and Q have the same `lane_id`, zero, given by `threadID`$\%32$. In this example, only the threads with the same `lane_id` can have data races, therefore we focus on P and Q; the execution of threads in other lanes mirrors these two threads. If `j` is a statically known constant, then each location `g[j]` in shared memory is treated as a separate variable `g_j` in the thread programs. In this case `j` equals `lane_id`.

```
WELLSYNC($T$: CTA, $\tau$: Trace) : Bool
Return true iff $T$ is well-synchronized.

 1: assume $\tau \equiv (I, T), \ldots, (F, done)$
 2: $G := Gen(\tau)$
 3: $R := \emptyset$
 4: for each $(c_1; c_2) \in T$ do
 5:     $R := R \cup \{(c_1, c_2)\}$
 6: end for
 7: for each $c_1 \equiv$ arrive $b\ n$ do
 8:     for each $c_2 \equiv$ sync $b\ n$ and $G(c_1) = G(c_2)$ do
 9:         $R := R \cup \{(c_1, c_2)\}$
10:     end for
11: end for
12: for each $c_1 \equiv$ sync $b\ n$ do
13:     for each $c_2 \equiv$ sync $b\ n$ and $G(c_1) = G(c_2)$ do
14:         $R := R \cup \{(c_1, c_2), (c_2, c_1)\}$
15:     end for
16: end for
17: $R :=$ Transitive closure of $R$
18: for each $c_1 \equiv$ sync $b\ n$ and $G(c_1) = k$ and each $c_2$ with
    barrier $b$ and $G(c_2) = k + 1$ and $c_3; c_2 \in T$ do
19:     if $(c_1, c_3) \notin R$ then
20:         return false
21:     end if
22: end for
23: return true
```

Figure 4. The algorithm for checking well synchronization.

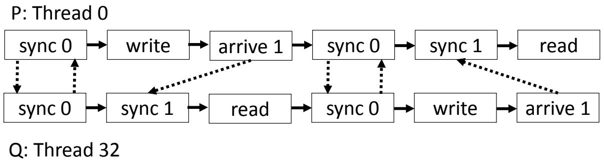

Figure 5. A path between two commands of P and Q signifies a happens-before relationship.

We statically simulate one possible execution of the CTA to generate a trace τ. If this execution deadlocks or throws an error then we have found a violation of the well synchronization property. Otherwise, we have a trace τ that reaches *done*. For Figure 3, we can have the following trace. Suppose P and Q first synchronize on barrier 0, then P writes to g, registers on barrier 1 and blocks on barrier 0. Next, Q blocks on barrier 1 which is recycled, reads from g, recycles barrier 0, writes to g and registers on barrier 1. The unblocked thread P now recycles barrier 1 and reads g.

From this trace τ we can extract $Gen(\tau)$ which assigns non-zero generations to all synchronization commands. The generation IDs are shown in parentheses with the synchronization commands of the thread programs in Figure 3. For example, the first command of all threads synchronize on barrier 0 creating the first generation of barrier 0. Then the fourth command of all threads synchronize again on barrier 0 and this time the assigned generation ID is 2.

To ensure well synchronization, we statically check that all traces assign the same generations to commands as $Gen(\tau)$. Our algorithm is shown in Figure 4. We first use $Gen(\tau)$ to construct a static happens-before relation R. The relation starts empty (line 3). We add successive commands to R. For GPUs the successive commands are guaranteed to execute in order. Figure 5 shows a graphical representation of the relation R for P and Q: the edges represent the tuples in R. The graph starts with no edges and lines 4-6 add the solid edges, corresponding to instruction ordering relations within individual threads.

Next, we add the edges corresponding to the inter-thread happens-before relations. If c_1 is an `arrive` and c_2 is a `sync` such that the two commands are in the same generation, then add (c_1, c_2) to R. Now, for c_1 and c_2 corresponding to `sync` in the same generation, add (c_1, c_2) and (c_2, c_1) to R. Lines 7 to 16 perform these steps and result in the dashed edges in Figure 5. There are cycles in this figure because the happens-before relation includes the commands that execute simultaneously (e.g. sync on the same barrier). Finally, we compute the transitive closure of R (line 17) thus yielding the full static happens-before relation. There is a path between c_1 and c_2 in Figure 5 iff $(c_1, c_2) \in R$.

After constructing the happens-before relation for all program commands, lines 18 to 23 check that there exists happens-before relationships between successive generations of the same barrier. For example, for Figure 3 we need to establish happens-before relationships between line 1 of P and line 4 of Q, between line 2 of Q and line 5 of P, etc. We can observe that for all checks made for commands (c_1, c_3) on line 19, there is a path between the two commands in Figure 5. On line 19, we check for (c_1, c_3) instead of (c_1, c_2) because we need to check that the program point just *before* c_2 happens after c_1. Intuitively, we have shown that successive generations are separated by a happens-before relationship and therefore the CTA is well synchronized. We now establish this formally. Our main result is the following:

Lemma 1. *For well-synchronized configurations the static happens-before relation as constructed in Figure 4 is sound and precise.*

The soundness of R is direct: all tuples added to R are sound because there is no out-of-order execution of commands within individual thread programs and because the well synchronization property imposes barrier generations in all traces. The precise part requires an induction on the size of the programs and the observation that the tuples in R are the only restrictions on the ordering in the executions imposed by the semantics.

Ideally, we could claim that if the algorithm in Figure 4 succeeds, then R ensures that $Gen(\tau)$ is maintained in all executions and the kernel is well-synchronized. Unfortunately, this reasoning is circular because only well-synchronized programs are guaranteed to have a sound R and here we are using R to ensure that generations are ordered. Fortunately, we have the following soundness result based on a different proof approach.

Theorem 1. *If $\tau \equiv (I, T) \rightsquigarrow^* (F, done)$ and* WELLSYNC*(T,τ) returns true then T is well-synchronized.*

We induct on suffixes of the given execution that reaches *done*. In the base case, the program *done* is vacuously well-synchronized. In the induction step, we have a well-synchronized configuration (s_1, T_1) and suppose in the given trace $(s_2, T_2) \rightsquigarrow (s_1, T_1)$. Then the checks in WELLSYNC ensure that the new generation introduced in going from T_1 to T_2 has a happens-before relationship with generations in T_1 and (s_2, T_2) is well-synchronized.

Completeness of WELLSYNC follows because well-synchronized programs have a precise R. The time complexity of WELLSYNC is dominated by the cost of computing the transitive closure, which for a CTA with n commands can be $O(n^3)$ in the worst case. In Section 5 we describe the essential optimizations required for efficiently computing the transitive closure in practice.

If we succeed in proving that T is well-synchronized, then we have proven that all executions of T are deadlock free and recycle named barriers safely. Moreover, as a side-effect of this procedure, Lemma 1 provides a sound and precise static happens-before relation R. Once we have this relation, obtaining a sound and complete algorithm for checking data races is direct. For two commands c_1 and c_2 that are accessing the same shared memory location with at least one write, we report a race if $(c_1, c_2) \notin R$ and $(c_2, c_1) \notin R$. For Figure 3, we need to check that there are happens-before relationships between the write in P and the write in Q, between the write in P and the read in Q and vice versa. From Figure 5, we observe that there is a path between any two accesses to the shared memory and therefore programs P and Q in Figure 3 are race-free.

5. Verification using WEFT

We now describe WEFT[6], our verification tool that uses the algorithm developed in Section 4 to verify the correctness of real warp-specialized kernels. We first describe our implementation of WEFT and then cover the results of using WEFT to verify 26 warp-specialized kernels that use the CudaDMA library or were emitted by the Singe compiler.

5.1 Emulation of GPU Programs in WEFT

WEFT takes as input the PTX assembly for a GPU program and begins by translating the program into the formal language specified in Section 3. WEFT performs this translation by emulating one CTA (usually the first CTA in a kernel) until it terminates, deadlocks, or encounters an error. To perform the emulation, WEFT initializes basic registers such as the CTA ID and thread IDs, but assumes nothing about the input arguments passed to the kernel. All threads in the CTA are emulated concurrently so that WEFT can accurately model blocking on named barriers. WEFT will usually continue emulating an individual thread until it blocks on a named barrier, and then resumes the thread after the barrier has completed.

In addition to the normal emulation mode, WEFT also supports a separate mode for emulating warp-synchronous execution where threads in the same warp are executed in lock-step. This permits WEFT to model current architectures that exhibit the behavior that any thread in a warp arriving at a named barrier is equivalent to the entire warp arriving [1]. This emulation mode also allows WEFT to validate kernels which rely on a warp-synchronous execution assumption for correctness. It is important to note that users must explicitly opt into this mode of emulation as it is not guaranteed to hold for future GPU architectures [14].

WEFT maintains a separate program for every thread in the CTA. Whenever a PTX instruction that directly corresponds to a command in our formal language is encountered (e.g. `bar.sync`) the command is recorded in the program for the thread being emulated. Since WEFT actually simulates PTX, it can unroll all statically bound loops and evaluate static branch conditions to automatically generate the necessary straight-line code. WEFT is also sophisticated enough to track data exchanged through shared memory or using shuffle instructions on the Kepler architecture.

Emulation of a CTA continues until one of three conditions is met: all threads successfully return, a deadlock is detected where no thread can make forward progress, or WEFT is unable to emulate a required instruction. In the second case, WEFT will report the deadlock and how it occurred to the user. The third case arises because WEFT doesn't have access to the inputs to the CTA and cannot reason about dynamic data stored in the framebuffer memory. WEFT is smart enough to elide irrelevant instructions for which it doesn't have inputs (e.g. most floating point math), but it cannot soundly ignore instructions which impact the translation to our formal language. If WEFT encounters a conditional statement, shared memory access, or barrier statement which cannot be emulated because the inputs depend on dynamic parameters to the CTA, then WEFT will raise an error indicating that the kernel cannot be validated by our approach. For all warp-specialized kernels of which we are aware, WEFT successfully generates straight-line thread programs.

[6] In weaving, a weft is used to maintain the alignment of threads in a warp.

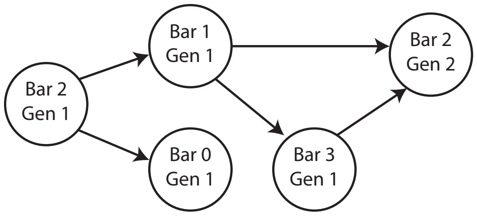

Figure 6. Barrier dependence graph for Figure 2.

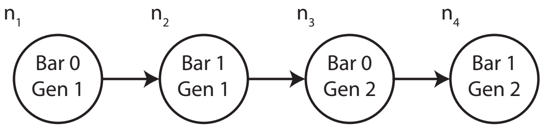

Figure 7. Barrier dependence graph for Figure 5.

5.2 Optimized WEFT Verification Algorithm

After WEFT has generated the thread programs for a CTA, it proceeds to check that the programs are well-synchronized and shared memory race-free. Our initial version of WEFT used a direct implementation of the algorithm in Figure 4 to check if a kernel was well synchronized. Unfortunately, this approach did not scale to many of the kernels we tested. The cause of the scaling problem was the transitive closure computation over the relation R, which is cubic in the number of commands; many of the kernels we considered contained tens to hundreds of millions of commands. To allow WEFT to scale to these kernels, we employ a modified but equivalent algorithm, with more sophisticated data structures to reduce both memory and computation costs.

The first step that WEFT takes is to construct a *barrier dependence graph* which captures the happens-before relationship between barriers. WEFT omits all `read` and `write` commands from the thread programs and simulates one execution of the CTA with only the synchronization commands. Next, WEFT uses the algorithm in Figure 4 to determine if the CTA is well-synchronized. Since memory accesses do not affect synchronization, they can be safely ignored without compromising the soundness or completeness of WELLSYNC. This simplification considerably reduces the number of commands to consider and makes the computation tractable for all of the kernels of which we are aware.

After verifying that the CTA is well-synchronized, WEFT constructs the barrier dependence graph. Each completed barrier is converted to a single node in the graph and is assigned a generation. We call the nodes of the graph *dynamic barriers*. We use c_i^P to denote the i^{th} command of program P. The trace τ of Section 4.2 creates the following nodes for Figure 3: n_1 has c_1^P, c_1^Q, generation 1; n_2 has c_3^P, c_2^Q, generation 1; n_3 has c_4^P, c_4^Q, generation 2; n_4 has c_5^P, c_6^Q, generation 2. Edges are added to the graph by traversing each thread program both forward and backwards to establish happens-before and happens-after relationships between pairs of dynamic barriers. Figures 6 and 7 show the computed barrier dependence graphs for the earlier examples of Figures 2 and 5 respectively. Note that the barrier dependence graph of a well-synchronized CTA is always a directed acyclic graph.

After verifying that a kernel is well-synchronized and generating the barrier dependence graph, WEFT proceeds to check if a kernel is race-free. Due to the large number of commands WEFT cannot compute the standard transitive closure of Figure 4 and it cannot traverse the graph of all commands. In order to scale to kernels requiring hundreds of billions of race tests, WEFT performs constant time race tests by computing the *latest happens-before* and *earliest happens-after* points reachable in every thread program from every program point. Figure 8 gives a visual depiction of these relationships for a given analysis point with respect to two other threads.

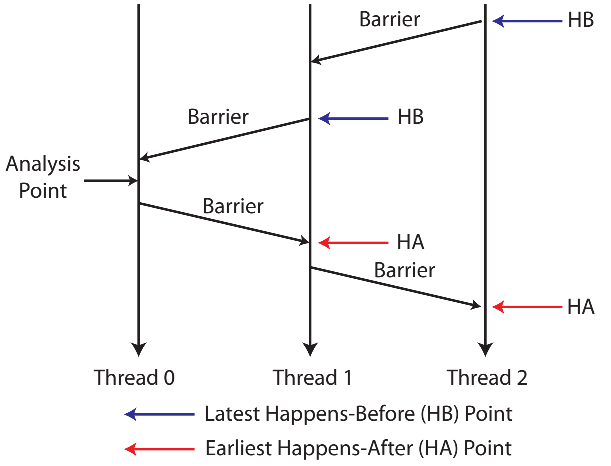

Figure 8. Weft latest/earliest happens-before/after relationships.

By leveraging the latest/earliest happens-before/after points, it is easy to determine if a race exists by performing a simple look-up at the program point of one command in the race test. For a given command c, the commands between the latest happens-before and the earliest happens-after are the commands that can run in parallel with c. For example, from Figure 5, the latest happens before point for the second command of P (c_2^P) is the first command of Q (c_1^Q). The command c_1^Q is guaranteed to execute before c_2^P and the commands after c_1^Q can run in parallel with c_2^P. The earliest happens-after point of c_2^P is c_3^Q as this `read` is guaranteed to execute after c_2^P and commands before it can run in parallel with c_2^P. Given the *barrier interval* (c_1^Q, c_3^Q), to know whether a command c_k^Q can run in parallel with c_2^P, we just need to check whether c_k^Q belongs to this barrier interval or not, i.e., whether $k \in (1, 3)$.

The crucial insight for scalability is that WEFT only needs to perform latest/earliest happens-before/after reachability analysis for all the nodes in the barrier dependence graph and not for every program point in the thread programs. Individual program points can later easily determine their latest/earliest happens-before/after points by using the results from the dynamic barriers that immediately precede and follow within the same thread program. This greatly reduces both computation time and overall memory usage.

WEFT first computes the earliest/latest happens-after/before points reachable in every thread program from each node in the barrier dependence graph. Each dynamic barrier initializes its reachability points with its participants and sets the remaining values to $\perp$. The transitive closure of reachability points is then computed over the barrier dependence graph. WEFT avoids recomputation by using a topological sort of the barrier nodes to guide the computation, requiring at worst $O(D^2)$ time for a kernel with D dynamic barrier nodes. Since D is usually much smaller than the total number of commands, this approach is considerably faster. Furthermore, WEFT can leverage the knowledge that a fixed number of dynamic barriers (usually 16) are live at a time to further improve performance. If the kernel contains N thread programs, the results only require $O(DN)$ space to store, which is independent of the total number of kernel commands. This computation produces the barrier intervals for the synchronization commands. For example, the generated barrier interval of $c_1^P \in n_1$ in thread Q is (c_0^Q, c_2^Q), where the 0^{th} command is just a placeholder for the beginning of a program. Similarly the barrier interval for $c_3^P \in n_2$ is (c_1^Q, c_3^Q).

For individual program commands, WEFT computes the earliest/latest happens-after/before relationships based on the results

Kernel Name	Dynamic Barriers	Shared Addresses	Commands (Thousands)	Race Tests (Millions)	Programs (threads)	Memory Usage (MB)	Verification Time (s)	Races (B,H)
saxpy_single	8192	512	3670	1878	320	3645	8.42	(no,no)
saxpy_double	8192	1024	3670	939	384	4298	8.99	(no,no)
sgemv_vecsingle	257	128	2142	17246	160	247	46.42	(no,no)
sgemv_vecdouble	514	256	4285	34492	192	498	95.19	(no,no)
sgemv_vecmanual	258	1024	8446	34490	160	814	147.81	(no,no)
sgemv_bothsingle	514	4128	1287	4370	288	207	10.15	(no,no)
sgemv_bothdouble	1028	8256	2573	8740	448	507	25.30	(no,no)
sgemv_bothmanual	516	8256	1320	2185	416	305	7.74	(no,no)
rtm_1phase_single	215	633	376	130	160	67	0.43	(**Yes**,no)
rtm_1phase_manual	214	1265	372	74	160	67	0.26	(**Yes**,no)
rtm_2phase_single	215	633	387	136	160	70	0.39	(**Yes**,**Yes**)
rtm_2phase_manual	214	1265	382	77	160	69	0.32	(**Yes**,**Yes**)
rtm_2phase_quad	208	1265	371	71	160	68	0.28	(**Yes**,**Yes**)

Figure 9. Verification results for the CudaDMA kernels. Races are classified as (benign,harmful).

Kernel Name	Dynamic Barriers	Shared Addresses	Commands (Thousands)	Race Tests (Millions)	Programs (threads)	Memory Usage (MB)	Verification Time (s)	Races (B,H)
dme_diff_fermi	352	1930	2912	169264	320	600	398.00	(no,no)
dme_diff_kepler	352	1920	1075	255	320	411	1.40	(no,no)
dme_visc_kepler	128	1920	2058	1038	480	385	3.24	(no,no)
dme_chem_fermi	864	5536	5156	40463	512	1917	53.74	(no,no)
dme_chem_kepler	1760	3072	3866	2486	256	1143	11.91	(no,**Yes**)
hept_diff_kepler	416	1664	1490	633	416	723	2.60	(no,no)
hept_visc_fermi	128	1690	5897	168972	416	693	416.92	(no,no)
hept_visc_kepler	128	1664	2955	2530	416	419	5.22	(no,no)
hept_chem_fermi	992	5192	7572	83762	640	3330	135.03	(no,no)
hept_chem_kepler	928	6144	5241	2375	512	2153	11.82	(no,**Yes**)
prf_diff_kepler	672	3712	6414	5729	928	5099	22.18	(no,no)
prf_visc_fermi	4	3776	901	1737	1024	114	2.64	(no,no)
prf_visc_kepler	128	3712	14213	26706	1024	2123	68.43	(no,no)

Figure 10. Verification results for the Singe kernels. Races are classified as (benign,harmful).

stored in the barrier dependence graph. For each command, WEFT consults the barriers in the same thread immediately before/after the command with different physical barrier names to determine the latest/earliest happens before/after for each thread program. For example, the barrier interval of `write` c_2^P is computed using the barrier intervals of nodes n_1 and n_2. If there are S commands in the program, this requires $O(S)$ time to compute because there are a constant number of dynamic barriers to consider (the maximum number of live barriers). For a kernel with N thread programs, the result of this computation requires $O(SN)$ memory to store. While this is large, for most kernels it requires at most tens of gigabytes of data which is within the range of many machines today.

After establishing the earliest happens-after and latest happens-before relationships for each program command, WEFT examines all pairs of shared memory accesses to a common address in which at least one access is a write and determines if there is a race. Each race test is a constant time look-up since each command knows the earliest happens-after and latest happens-before command that can be reached (if any) in every other thread program. As we will show in Section 5.3, this approach allows WEFT to validate very large and complex warp-specialized kernels in a reasonable amount of time and memory.

5.3 Experiments

We evaluated WEFT by using it to validate 13 warp-specialized kernels that use the CudaDMA library and 13 warp-specialized kernels that were emitted by the Singe compiler. The CudaDMA kernels ranged from 65-1561 lines of CUDA while the Singe kernels ranged from 1684-13245 lines of CUDA. All our experiments were run on an Intel Xeon X5680 Sandy Bridge processor clocked at 3.33 GHz with 48 GB of memory divided across two NUMA domains. The performance of WEFT is primarily limited either by memory latency or memory bandwidth depending on both the processor on which it is being run and the kernel being verified. WEFT is multi-threaded to take advantage of the considerable amount of memory-level parallelism available in our verification algorithm. All our experiments using WEFT were run with four threads (two per NUMA domain) as we found this to be the ideal settings for maximizing memory system performance on our target machine.

Figures 9 and 10 show the verification results for the CudaDMA and Singe kernels respectively. WEFT reports statistics on each of the kernels including the number of dynamic barriers, total shared memory addresses that are used, total commands in the formal language from Section 3 across all threads, the number of thread programs, and the total race tests that were performed. Performance is measured by the time required to verify each kernel and the total memory required to perform the verification. WEFT reports whether the kernel is well-synchronized and race-free. Not surprisingly, all the kernels we examined were well-synchronized and therefore we omit these results from the tables. Kernels that are not well-synchronized tend to hang or crash frequently. While it is easy to cause such bugs to manifest, it is much more difficult to diagnose their root cause. In the future, we anticipate WEFT will be especially valuable in giving feedback for debugging warp-specialized kernels that either deadlock or crash due to a violation of the well-synchronized property during development.

```
1  __global__ void two_phase(...) {
2    __shared__ float2 s_PQ[...];
3    float2 *PQy_buf = s_PQ + PQY_BUF_OFFSET;
4    ...
5    if (tid < COMPUTE_THREADS_PER_CTA) {
6      // Compute warps, main loop
7      for (int iz = 0; iz < NZ; iz++) {
8        ...
9        // Start next transfer to shared memory
10       dma_ld_pq.start_async_dma(); //barrier arrive
11       // Still reading old version in PQy_buf
12       for (int j = 1; j < R; j++) {
13         const float2 v1 = PQy_buf[PQ_index-j], v2 = PQy_buf[PQ_index+j];
14         q_Pxyl += c_x[j] * (v1.x + v2.x); q_Qxyl += c_x[j] * (v1.y + v2.y);
15       }
16     }
17   } else { // DMA warps }
18 }
```

Listing 3. Write-after-read bug in two-phase RTM.

The performance results show that WEFT is capable of scaling to very large kernels containing up to 8K dynamic barriers and 14 million commands. WEFT is able to verify most kernels in a few minutes or less (within the time of a standard coffee break), making it practical for use by real developers. WEFT also saves the time the developers would normally invest in writing tests that uncover deadlocks and data races; since WEFT is sound there is no need to test for the properties that it checks and the testing effort can be focused on other properties such as functional correctness. Moreover, since WEFT is complete, all errors reported by WEFT are real errors and worth investigating. Finally, since the time complexity is a small polynomial function, the tool execution time is predictable. In the worst case, WEFT required seven minutes to verify the `hept_visc_fermi` kernel. This kernel makes extensive use of warp-synchronous programming to exchange thousands of constants through the same shared memory locations, requiring more than 168 billion race tests. Note that the use of the new shuffle instructions available on the Kepler architecture by the `dme_diff_kepler` kernel reduces the number of tests by three orders of magnitude. The `prf_diff_kepler` kernel uses the most memory (5.1 GB), but this usage is well within the memory limits of most modern laptop and desktop machines.

WEFT found several different data races in seven of the kernels from both the CudaDMA and Singe suites. These races can be classified into two categories: *benign* and *harmful*. All of the reverse time migration (RTM) kernels from the CudaDMA test suite had an instance of a benign data race where the developer intentionally had multiple threads load the same value from global memory and write it to the same location in shared memory during the initialization loops of the kernel. The resulting code was simpler to understand and maintain and caused negligible performance degradation. The WEFT tool does not distinguish between benign and harmful races and correctly reports all the competing writes to the same shared memory locations as races. Since benign data races can cause performance degradation, it is still important to report them to the developer even if they do not break correctness.

The two-phase RTM kernels from the CudaDMA suite also contained a bug which resulted in a number of harmful write-after-read data races. Listing 3 shows the pertinent parts of code that cause the data race in all of the two-phase RTM kernels. Initially, a shared memory buffer called `s_PQ` is allocated and an alias called `PQy_buf` is created as a pointer to the middle of the buffer (lines 2-3). Inside of the main loop of the compute warps, the placement of the `start_async_dma` call (line 10) comes before the last use of the shared memory buffer through the `PQy_buf` pointer (line 13), resulting in a write-after-read race as the DMA warps begin writing into shared memory before the last read. Interestingly, this resolved a two-year old non-determinism bug filed against the two-phase kernels, which persisted because the single phase kernels achieved higher performance.

Two Singe kernels also contained write-after-read data races. Both the DME and Heptane chemistry kernels for the Kepler architecture contained instances where writes could occur before previous reads had completed. The cause of these races was a bug in the heuristic that the Singe compiler uses for solving the NP-complete bin-packing problem for managing dataflow transfers through the limited space in shared memory. Interestingly, the bug only manifested on the Kepler architecture where the additional register file space permitted a more aggressive mapping strategy that stressed the bin-packing solver. Despite extensive testing, as far as we know, these write-after-read races have never manifested in practice on current GPU architectures due to the large number of intermediate instructions. However, there is no guarantee that the bug in the Singe compiler would not have manifested on future GPU architectures, further underscoring the need for verification of complex warp-specialized kernels.

6. Discussion and Related Work

There is considerable literature on the verification of concurrent programs and some recent work on GPU verification. Existing work on GPU verification has focused on general-purpose verification of GPU kernels and the need to handle various programming constructs provided by GPU models. Our work is largely orthogonal: we focus on obtaining complete and efficient solutions for handling the verification of named barriers by sacrificing expressiveness. We do not operate at the source level and we do not aim to provide a general-purpose GPU verification tool. Instead, we focus specifically on the problems associated with producer-consumer named barriers, which no previous work addresses. Therefore, we believe this work is largely complementary to the existing work on verification of GPU programs. Furthermore, we feel that this work provides a complete solution for the problems associated with named barriers and could easily be integrated into a more general purpose GPU verification framework.

There are many other verification tools which have attempted to reason about the synchronization schemes of GPU kernels. GPUVerify [5, 8, 11, 12] is a general-purpose verifier for GPU kernels. In the GPUVerify model, a CTA-wide barrier (e.g. `syncthreads`) is the only synchronization mechanism. This restriction permits GPUVerify to perform a *two-thread reduction*: a round robin scheduling of threads is sufficient to detect all data races. A similar reduction is also used by PUG [19]. While this reduction can considerably reduce the cost of the analysis and simplifies the implementation of the verifier, it does not have sufficient power to handle the full generality of named barriers. GPUVerify can also reason about warp-synchronous programming (as can WEFT) and atomics (which WEFT does not) [4].

There are many different tools that use symbolic execution [10, 13, 20, 21] to search for bugs in GPU kernels. To the best of our knowledge none of these tools has any support for reasoning about named barriers. However, we see no fundamental reason that they could not be modified to understand named barrier semantics.

GPU Amplify [18] relies on a combination of static and dynamic analysis to search for data races in GPU kernels. We believe that our work subsumes and generalizes their approach for searching for shared memory races. In particular, their approach can be seen as a special case instance of WEFT that requires that all kernels use only a single named barrier for CTA-wide synchronization.

In some ways, named barrier semantics are related to other kinds of barriers. Java Phasers have similar semantics to named barriers, but are implemented in software[23]. Other (non-exhaustive) work on simple barriers such as those used for distributed programming include [2, 17]. While there are some similarities, the fundamental implications of named barriers being a hardware resource instead of a software object give them a much stricter semantics which is therefore important to verify.

7. Conclusion

As GPU usage continues to expand into new application domains with more task parallelism and larger working sets, warp specialization will become increasingly important for handling challenging kernels that do not fit the standard data-parallel paradigm. The named barriers available on NVIDIA GPUs make warp specialization possible, but require more complex code with potentially difficult to discern bugs.

We have presented WEFT, a verifier for kernels using named barriers that is sound, complete, and efficient. Soundness ensures that developers do not have to write any tests to check for synchronization errors. Completeness ensures that all violations reported by WEFT are actual bugs; there are no false alarms. WEFT runs on large, complex, warp-specialized kernels requiring up to billions of checks for race conditions and completes in a few minutes. Using WEFT we validated 26 kernels and found several non-trivial bugs.

Acknowledgements

We wish to thank Michael Garland, Vinod Grover, Divya Gupta, Manolis Papadakis, Sean Treichler, and the anonymous reviewers for their valuable comments and feedback. This material is based on research sponsored by a fellowship from Microsoft, Los Alamos National Laboratories Subcontract No. 173315-1 through the U.S. Department of Energy under Contract No. DE-AC52-06NA25396, and NSF grant CCF-1160904.

References

[1] Parallel thread execution ISA version 4.1. `http://docs.nvidia.com/cuda/parallel-thread-execution/index.html`, 2014.

[2] A. Aiken and D. Gay. Barrier inference. In *POPL*, pages 342–354, 1998.

[3] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In *ASPLOS*, pages 577–591, 2015.

[4] E. Bardsley and A. F. Donaldson. Warps and atomics: Beyond barrier synchronization in the verification of GPU kernels. In *NFM*, pages 230–245, 2014.

[5] E. Bardsley, A. Betts, N. Chong, P. Collingbourne, P. Deligiannis, A. F. Donaldson, J. Ketema, D. Liew, and S. Qadeer. Engineering a static verification tool for GPU kernels. In *CAV*, pages 226–242, 2014.

[6] M. Bauer, H. Cook, and B. Khailany. CudaDMA: optimizing GPU memory bandwidth via warp specialization. SC '11, 2011.

[7] M. Bauer, S. Treichler, and A. Aiken. Singe: Leveraging warp specialization for high performance on GPUs. PPoPP '14, 2014.

[8] A. Betts, N. Chong, A. F. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: a verifier for GPU kernels. In *OOPSLA*, pages 113–132, 2012.

[9] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science and Discovery*, page 015001, 2009.

[10] W. Chiang, G. Gopalakrishnan, G. Li, and Z. Rakamaric. Formal analysis of GPU programs with atomics via conflict-directed delay-bounding. In *NFM*, pages 213–228, 2013.

[11] N. Chong, A. F. Donaldson, P. H. J. Kelly, J. Ketema, and S. Qadeer. Barrier invariants: a shared state abstraction for the analysis of data-dependent GPU kernels. In *OOPSLA*, pages 605–622, 2013.

[12] N. Chong, A. F. Donaldson, and J. Ketema. A sound and complete abstraction for reasoning about parallel prefix sums. In *POPL*, pages 397–410, 2014.

[13] P. Collingbourne, C. Cadar, and P. H. J. Kelly. Symbolic crosschecking of floating-point and SIMD code. In *EuroSys*, pages 315–328, 2011.

[14] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. Gpus and the future of parallel computing. *IEEE Micro*, 31(5):7–17, 2011.

[15] Khronos. The OpenCL Specification, Version 1.0. The Khronos OpenCL Working Group, December 2008.

[16] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[17] D.-K. Le, W.-N. Chin, and Y. M. Teo. Verification of static and dynamic barrier synchronization using bounded permissions. In *ICFEM*, pages 231–248, 2013.

[18] A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner. Verifying GPU kernels by test amplification. In *PLDI*, pages 383–394, 2012.

[19] G. Li and G. Gopalakrishnan. Scalable smt-based verification of GPU kernel functions. In *FSE*, pages 187–196, 2010.

[20] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. GKLEE: concolic verification and test generation for GPUs. In *PPOPP*, pages 215–224, 2012.

[21] P. Li, G. Li, and G. Gopalakrishnan. Parametric flows: automated behavior equivalencing for symbolic analysis of races in CUDA programs. In *SC*, page 29, 2012.

[22] NVIDIA. CUDA programming guide 6.0. `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`, August 2014.

[23] Oracle. Class phaser. `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Phaser.html`, 2014.

[24] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010.

[25] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.

[26] Top500. Top 500 supercomputers. `http://www.top500.org`, 2014.