GPU Schedulers: How Fair Is Fair Enough?

Tyler Sorensen
Imperial College London, UK
t.sorensen15@imperial.ac.uk

Hugues Evrard
Imperial College London, UK
h.evrard@imperial.ac.uk

Alastair F. Donaldson
Imperial College London, UK
alastair.donaldson@imperial.ac.uk

Abstract

Blocking synchronisation idioms, e.g. mutexes and barriers, play an important role in concurrent programming. However, systems with semi-fair schedulers, e.g. graphics processing units (GPUs), are becoming increasingly common. Such schedulers provide varying degrees of fairness, guaranteeing enough to allow some, but not all, blocking idioms. While a number of applications that use blocking idioms do run on today's GPUs, reasoning about liveness properties of such applications is difficult as documentation is scarce and scattered.

In this work, we aim to clarify fairness properties of semi-fair schedulers. To do this, we define a general temporal logic formula, based on weak fairness, parameterised by a predicate that enables fairness per-thread at certain points of an execution. We then define fairness properties for three GPU schedulers: HSA, OpenCL, and occupancy-bound execution. We examine existing GPU applications and show that none of the above schedulers are strong enough to provide the fairness properties required by these applications. It hence appears that existing GPU scheduler descriptions do not entirely capture the fairness properties that are provided on current GPUs. Thus, we present two new schedulers that aim to support existing GPU applications. We analyse the behaviour of common blocking idioms under each scheduler and show that one of our new schedulers allows a more natural implementation of a GPU protocol.

2012 ACM Subject Classification Software and its engineering → Semantics, Software and its engineering → Scheduling, Computing methodologies → Graphics processors

Keywords and phrases GPU scheduling, Blocking synchronisation, GPU semantics

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2018.23

Acknowledgements We are grateful to Jeroen Ketema, John Wickerson, and Qiyi Tang for feedback and insightful discussions around this work. We are grateful to Joseph Greathouse for pointing out additional applications that require HSA progress guarantees. We thank the CONCUR reviewers for their thorough evaluations and feedback. This work was supported by the EPSRC, through an Early Career Fellowship (EP/N026314/1), the IRIS Programme Grant (EP/R006865/1) and a gift from Intel Corporation.

1 Introduction

The *scheduler* of a concurrent system is responsible for the placement of virtual threads onto hardware resources. There are often insufficient resources for all threads to execute in parallel, and it is the job of the scheduler to dictate resource sharing, potentially influencing the temporal semantics of concurrent programs. For example, consider a two threaded

© Tyler Sorensen, Hugues Evrard, and Alastair F. Donaldson;
licensed under Creative Commons License CC-BY
29th International Conference on Concurrency Theory (CONCUR 2018).
Editors: Sven Schewe and Lijun Zhang; Article No. 23; pp. 23:1–23:17

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

program where thread 0 waits for thread 1 to set a flag. If the scheduler never allows thread 1 to execute then the program will hang due to starvation. Thus, to reason about liveness properties, developers must understand the *fairness* guarantees provided by the scheduler.

GPUs are highly parallel co-processors found in many devices, from mobile phones to super computers. While these devices initially accelerated graphics computations, the last ten years have seen a strong push towards supporting general purpose computation on GPUs. Today, through programming models such as OpenCL [11], CUDA [15], and HSA [8] we have mature frameworks to execute C-like programs on GPUs.

Current GPU programming models offer a compelling case study for schedulers for three reasons: (1) some blocking idioms, e.g. barriers, are known to hang due to starvation on current GPUs [20]; (2) other blocking idioms, e.g. mutexes, run without starvation on current GPUs; yet (3) documentation for some GPU programming models explicitly states that no guarantees are provided, while others state only minimal guarantees that are insufficient to ensure starvation-freedom even for mutexes. Because GPU schedulers are embedded in closed proprietary frameworks, we do not look at concrete scheduling logic, but instead aim to derive formal fairness guarantees from prose documentation and observed behaviours.

GPUs have a hierarchical programming model: in OpenCL threads are partitioned into *workgroups*. Threads in the same workgroup can synchronise via intrinsic instructions (e.g. the OpenCL `barrier` instruction [10, p. 99]). Yet, despite practical use cases (see Section 4), there are no such intrinsics for inter-workgroup synchronisation. Instead, inter-workgroup synchronisation is achieved by building constructs, e.g. mutexes, using finer-grained primitives, e.g. atomic read-modify-write instructions (RMWs). However, reasoning about such constructs is difficult as inter-workgroup thread interactions are relatively unstudied, especially in relation to fairness. Given this, we focus only on inter-workgroup interactions and threads will be assumed to be in disjoint workgroups. Under this constraint, it is cumbersome to use the word *workgroup*. Because we can think of a workgroup as being a "composite thread", we henceforth use the word *thread* to mean *workgroup*.

The unfair OpenCL scheduler and non-blocking programs. OpenCL is a programming model for parallel systems with wide support for GPUs. Due to scheduling concerns, OpenCL disallows all blocking synchronisation, stating [11, p. 29]: "A conforming implementation may choose to serialize the [threads] so a correct algorithm cannot assume that [threads] will execute in parallel. There is no safe and portable way to synchronize across the independent execution of [threads] . . . ." Such weak guarantees are acceptable for many GPU programs, e.g. matrix multiplication, as they are *non-blocking*. That is, these programs will terminate under an *unfair* scheduler, i.e. a scheduler that provides no fairness guarantees.

Blocking idioms and fair schedulers. On the other hand, there are many useful *blocking* synchronisation idioms, which require fairness properties from the scheduler to ensure starvation-freedom. Three common examples of blocking idioms considered throughout this work, *barrier*, *mutex* and *producer-consumer (PC)*, are described in Table 1. Intuitively, a *fair* scheduler provides the guarantee that any thread that is able to execute will eventually execute. Fair schedulers are able to guarantee starvation-freedom for the idioms of Table 1.

1.1 Semi-fair schedulers: HSA and occupancy-bound execution

We have described two schedulers: fair and unfair, under which starvation-freedom for blocking idioms is either always or never guaranteed. However, some GPU programming models have *semi-fair* schedulers, under which starvation-freedom is guaranteed for only

Table 1 Blocking synchronisation constructs considered in this work.

Barrier	**Mutex**	**Producer-consumer (PC)**
Aligns the execution of all participating threads: a thread waits at the barrier until all threads have reached the barrier. Blocking, as a thread waiting at the barrier relies on the other threads to make enough progress to also reach the barrier.	Provides mutual exclusion for a critical section. A thread *acquires* the mutex before executing the critical section, ensuring exclusive access. Upon leaving, the mutex is *released*. Blocking, as a thread waiting to acquire relies on the the thread in the critical section to eventually release the mutex.	Provides a handshake between threads. A *producer* thread prepares some data and then sets a flag. A *consumer* thread waits until the flag value is observed and then reads the data. Blocking, as the consumer thread relies on the the producer thread to eventually set the flag.

Table 2 Blocking synchronisation idioms guaranteed starvation-freedom under various schedulers.

	fair	HSA	OBE	unfair (e.g. OpenCL)
barrier	yes	no	occupancy-limited	no
mutex	yes	no	yes	no
PC	yes	one-way	occupancy-limited	no

some blocking idioms. We describe two such schedulers and informally analyse the idioms of Table 1 under these schedulers (summarised in Table 2). If starvation-freedom is guaranteed for all threads executing idiom i under scheduler s then we say that i is *allowed* under s.

Similar to OpenCL, Heterogeneous System Architecture (HSA) is a parallel programming model designed to efficiently target GPUs [8]. Unlike OpenCL however, the HSA scheduler conditionally allows blocking between threads based on *thread ids*, a unique contiguous natural number assigned to each thread. Namely, thread B can block thread A, if: "[thread] A comes after B in [thread] flattened id order" [8, p. 46]. Under this scheduler: a barrier is *not* allowed, as all threads wait on all other threads regardless of id; a mutex is *not* allowed, as the ids of threads are not considered when acquiring or releasing the mutex; PC is *conditionally* allowed *if* the producer has a lower id than the consumer.

Occupancy-bound execution (OBE) is a pragmatic GPU execution model that aims to capture the guarantees that current GPUs have been shown experimentally to provide [20]. While OBE is not officially supported, many GPU programs (discussed in Section 4) depend on its guarantees. OBE guarantees fairness among the threads that are currently *occupant* (i.e., are actively executing) on the hardware resources. The fairness properties are described as [20, p. 5]: "A [thread that has executed at least one instruction] is guaranteed to eventually be scheduled for further execution on the GPU." Under this scheduler: a barrier is *not* allowed, as all threads wait on all other threads regardless of whether they have been scheduled previously; a mutex *is* allowed, as a thread that has previously acquired a mutex will be fairly scheduled such that it eventually releases the mutex; PC is *not* allowed, as there is no guarantee that the producer will be scheduled relative to the consumer.

While general barrier and PC idioms are not allowed under OBE, constrained variants have been shown to be allowed by using an occupancy discovery protocol [20] (described in Section 5.1). The protocol works by identifying a subset of threads that have been observed

to take an execution step, i.e. it *discovers* a set of co-occupant threads[1]. Barrier and PC idioms are then able to synchronise threads that have been discovered; thus we say OBE allows *occupancy-limited* variants of these idioms.

It is worth noticing that the two variants of PC shown in Table 2 (occupancy-limited and one-way) are incomparable. That is, one-way is not occupancy-limited, as the OBE scheduler makes no guarantees about threads with lower ids being scheduled before threads with higher ids. Similarly, occupancy-limited is not one-way, as the OBE scheduler allows bi-directional PC synchronisation if both threads have been observed to be co-occupant.

▶ Remark (CUDA). Like OpenCL, CUDA gives no scheduling guarantees, stating [15, p. 11]: "[Threads] are required to execute independently: It must be possible to execute them in any order, in parallel or in series." Still, some CUDA programs rely on OBE or HSA guarantees (see Section 4). The recent version 9 of CUDA introduces *cooperative groups* [15, app. C], which provide primitive barriers between programmer specified threads. Because only barriers are provided, we do not consider cooperative groups. Indeed, we aim to reason about fine-grained fairness guarantees, as required by general blocking synchronisation.

1.2 Contributions and outline

The results of Table 2 raise the following points, which we aim to address:

1. The temporal correctness of common blocking idioms varies under different GPU schedulers; however, we are unaware of any formal scheduler descriptions that are able to validate these observations.
2. Two GPU models, HSA and OBE, have schedulers that are incomparable. However, for each scheduler, there are real programs that rely on its scheduling guarantees. Thus, neither of these schedulers captures all of the guarantees observed on today's GPUs.

To address (1), we develop a formalisation, based on weak fairness, for describing the fairness guarantees of semi-fair schedulers. This formula is parameterised by a *thread fairness criterion* (TFC), a predicate over a thread and the program state, that can be tuned to provide a desired degree of fairness. We illustrate our ideas by defining thread fairness criteria for HSA and OBE (Section 3).

To address (2), we first substantiate the claim by examining blocking GPU programs that run on current GPUs. We show that there are real programs that rely on HSA guarantees, as well as programs that rely on OBE guarantees (Section 4). That is, neither the HSA nor the OBE schedulers entirely capture guarantees on which existing GPUs applications rely.

Thus, we define fairness properties for two new schedulers: *HSA+OBE*, a combination of HSA and OBE, and *LOBE* (linear OBE), an intuitive strengthening of OBE based on contiguous thread ids. Both provide the guarantees required by current programs (Section 5), however we argue that LOBE corresponds to a more intuitive scheduler. We then present an optimisation to the occupancy discovery protocol [20] that exploits exclusive LOBE guarantees (Section 5.1). The schedulers we discuss are summarised in Figure 1.

To summarise, our contributions are as follows:

- We formalise the notion of semi-fair schedulers using a temporal logic formula and use this definition to describe the HSA and OBE GPU schedulers (Section 3).

[1] Some prior work uses *co-occupant* to describe the over-subscription of threads (workgroups) on a physical GPU core. In this work, we use *co-occupant* to mean threads (workgroups) that are logically executed in parallel on the GPU, potentially spanning many physical cores.

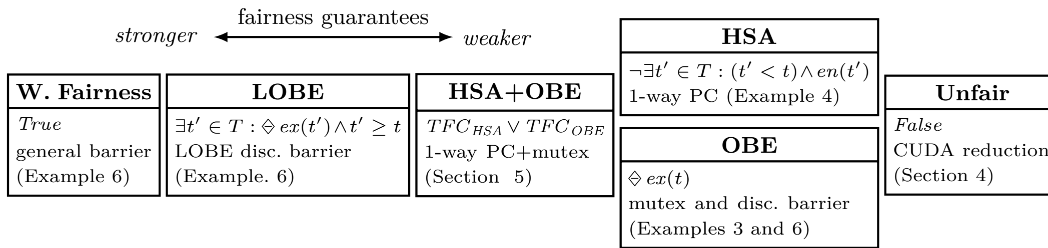

Figure 1 The semi-fair schedulers we define in this work from strongest to weakest. The first line in the box shows the scheduler's TFC (over a thread t), followed by the idiom(s) allowed under the scheduler and where the idiom is analysed. For any scheduler: any idiom to the right of a scheduler is allowed by the scheduler and any idiom to the left is disallowed. HSA and OBE are vertically aligned as they are not comparable.

- We examine blocking GPU applications and show that no existing GPU scheduler definition is strong enough to describe the guarantees required by all such programs (Section 4).
- We present two new semi-fair schedulers that meet the requirements of current blocking GPU programs: HSA+OBE and LOBE (Section 5). LOBE is shown to provide a more natural implementation of a GPU protocol (Section 5.1).

We have discussed related work on programming models and GPU schedules above, while applications that depend on specific schedulers are surveyed in Section 4.

2 Background

2.1 GPU programming

GPU programming models provide a hierarchical thread organisation. A GPU program, or a *kernel*, is executed by a set of workgroups, which we refer to as threads for convenience. Threads have access to a thread id, which can be used to partition inputs of data-parallel programs. We assume these constraints, which are common to GPU applications:

1. *Termination*: Programs are expected to terminate under a fair scheduler. GPU programs generally terminate, and in fact, they get killed by the OS if they execute for too long [19].
2. *Static thread count*: while dynamic thread creation has recently become available, e.g. nested parallelism [11, p. 30], we believe static parallelism should be studied first.
3. *Deterministic threads*: we assume that the scheduler is the only source of nondeterminism; the computation performed by a thread depends only on the program input and the order in which threads interleave. This is the case for all GPU programs examined.
4. *Enabled threads*: we assume all threads are *enabled*, i.e. able to be executed, at the beginning of the program and do not cease to be enabled until they terminate. While some systems contain scheduler-aware intrinsics, e.g. condition variables [3], GPU programming models do not. As a result, the idioms of Table 1 are implemented using atomic operations and busy-waiting, which do not change whether a thread is enabled or not.
5. *Sequential consistency*: while GPUs have relaxed memory models (e.g. see [18]), we believe scheduling under the interleaving model should be understood first.

2.2 Formal program reasoning

A *sequential* program is a sequence of instructions and its behaviour can be reasoned about by step-wise execution of instructions. We do not provide instruction-level semantics, but examples can be found in the literature (e.g. for GPUs, see [4]). A *concurrent* program is the

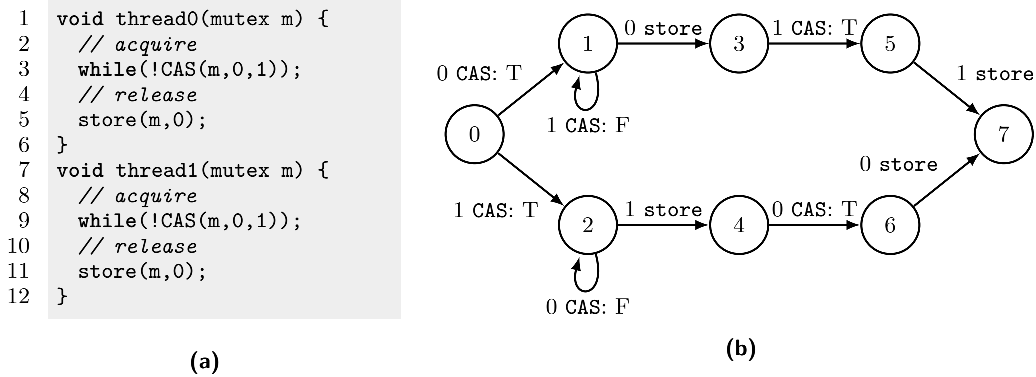

Figure 2 Two threaded mutex idiom (a) program code and (b) corresponding LTS.

parallel composition of n sequential programs, for some $n > 1$. The set $T = \{0, 1, \ldots, n-1\}$ provides a unique id for each thread, often called the *tid*. The behaviour of a concurrent program is defined by all possible *interleavings* of atomic (i.e. indivisible) instructions executed by the threads. Let A be the set of available atomic instructions.

For example, Figure 2a shows two sequential programs, `thread0` (with tid of 0) and `thread1` (with tid of 1), which both have access to a shared mutex object (initially 0). The set A of atomic instructions is {`CAS(m,old,new)`, `store(m,v)`}, whose semantics are as follows:

- `CAS(m,old,new)` – atomically checks whether the value of `m` is equal to `old`. If so, updates the value to `new` and returns true (T). Otherwise returns false (F).
- `store(m,v)` – atomically stores the value of `v` to `m`.

Using these two instructions, Figure 2a implements a simple mutex idiom, in which each thread loops trying to acquire a mutex (via the `CAS` instruction), and then immediately releases the mutex (via the `store` instruction). While other mutex implementations exist, e.g. see [7, ch. 7.2], the blocking behaviour shown in Figure 2a is idiomatic to mutexes.

Labelled transition systems. To reason about concurrent programs, we can use use a *labelled transition system* (LTS). Formally, an LTS L is a 4-tuple $(S, I, L, \rightarrow)$ where

- S is a finite set of states, with $I \subseteq S$ the set of initial states. A state contains values for all program variables and a program counter for each thread.
- $L \subseteq T \times A$ is a set of labels. A label is a pair (t, a) consisting of a thread id $t \in T$ and an atomic instruction $a \in A$.
- $\rightarrow \subseteq S \times L \times S$ is a transition relation. For convenience, given $(p, (t, a), q) \in \rightarrow$, we write $p \xrightarrow{t} q$. This is not ambiguous as we consider only per-thread deterministic programs. We use dot notation to refer to members of the tuple; e.g., given $\alpha \in \rightarrow$, we write $\alpha.t$ to refer to the thread id component of α.

Given a concurrent program, the LTS can be constructed iteratively. A start state s is created with program initial values (0 unless stated otherwise). For each thread $t \in T$, the next instruction $a \in A$ is executed to create state s' to explore. L is updated to include (t, a) and $(s, (t, a), s')$ is added to $\rightarrow$. This process iterates until there are no more states to explore. We show the LTS for the program of Figure 2a in Figure 2b. For ease of presentation, we omit state program values; labels show the thread id followed by the atomic action. If the action has a return value (e.g. `CAS`), it is shown following the action.

For a thread id $t \in T$ and state $p \in S$, we say that t is *enabled* in p, and write $en(p,t)$, if there exists a state $q \in S$ such that $p \xrightarrow{t} q$. We call p a *terminal* state, and write $terminal(p)$, if no thread is enabled in p; that is, $\neg en(p,t)$ holds for all $t \in T$. Intuitively, $en(p,t)$ states that it is possible for a thread t to take a step at state p and $terminal(p)$ states that all threads have completed execution at state p. The program constraints of Section 2.1 ensure that all threads are enabled until their termination.

Program executions and temporal logic. The executions E of a concurrent program are all possible *paths* through its LTS. Formally, a path $z \in E$ is a (possibly infinite) sequence of transitions: $\alpha_0\alpha_1\ldots$, with each $\alpha_i \in \rightarrow$, such that: the path starts in an initial state, i.e. $\alpha_0.p \in I$; adjacent transitions are connected, i.e. $\alpha_i.q = \alpha_{i+1}.p$; and if the path is finite, with n transitions, then it leads to a terminal state, i.e. $terminal(\alpha_{n-1}.q)$.

▶ Remark (Infinite paths). Because we assume programs terminate under fair scheduling (Section 2.1), infinite paths are discarded by the fair scheduler, but not necessarily by semi-fair schedulers. Indeed, these infinite paths allow us to distinguish semi-fair schedulers.

Given a path z and a transition α_i in z, $pre(\alpha_i, z)$ is used to denote the transitions up to, and including, i of z, that is, $\alpha_0\alpha_1\ldots\alpha_i$. Similarly, $post(\alpha_i, z)$ is used to denote the (potentially infinite) transitions of z from α_i, that is, $\alpha_i\alpha_{i+1}\ldots$. For convenience, we use $en(\alpha,t)$ to denote $en(\alpha.p,t)$ and $terminal(\alpha)$ to denote $terminal(\alpha.q)$. Finally, we define a new predicate $ex(\alpha,t)$ which holds if and only if $t = \alpha.t$. Intuitively, $ex(\alpha,t)$ indicates that thread t executes the transition α.

The notion of executions E over an LTS allows reasoning about *liveness* properties of programs. However, the full LTS may yield paths that realistic schedulers would exclude, illustrated in Example 1. Thus, fairness properties, provided by the scheduler, are modelled as a *filter* over the paths in E.

▶ **Example 1** (Mutex without fairness). The two-threaded mutex LTS given in Figure 2b shows that it is possible for a thread to loop indefinitely waiting to acquire the mutex if the other thread is in the critical section, as seen in states 1 and 2. Developers with experience writing concurrent programs for traditional CPUs know that on most systems, these non-terminating paths do not occur in practice!

Fairness filters and liveness properties can be expressed using *temporal logic*. For a path z and a transition α in z, temporal logic allows reasoning over $post(\alpha,z)$ and $pre(\alpha,z)$, i.e. reasoning about future and past behaviours. Following the classic definitions of fairness, linear time temporal logic (LTL), is used in this work (see, e.g. [2, ch. 5] for an in-depth treatment of LTL). For ease of presentation, a less common operator, $\Diamond\!\!\!-$, from past-time temporal logic (which has the same expressiveness as LTL [12]) is used. Temporal operators are evaluated with respect to z (a path) and α (a transition) in z. They take a formula ϕ, which is either another temporal formula or a transition predicate, ranging over $\alpha.p$, $\alpha.t$, or $\alpha.q$ (e.g. *terminal*). The three temporal operators used in this work are:

- The global operator $\Box$, which states that ϕ must hold for all $\alpha' \in post(\alpha,z)$.
- The future operator $\Diamond$, which states that ϕ must hold for at least one $\alpha' \in post(\alpha,z)$.
- The past operator $\Diamond\!\!\!-$, which states that ϕ must hold for at least at one $\alpha' \in pre(\alpha,z)$.

To show that a liveness property f holds for a program with executions E, it is sufficient to show that f holds for all pairs (z,α) such that $z \in E$ and α is the first transition in z. For example, one important liveness property is eventual termination: $\Diamond terminal$. Applying this

formula to the LTS of Figure 2b, a counter-example (i.e. a path that does not terminate) is easily found: $0 \xrightarrow{1} 2, (2 \xrightarrow{0} 2)^\omega$. In this path, thread 0 loops indefinitely trying to acquire the mutex. Infinite paths are expressed using ω-regular expressions [2, ch. 4.3].

However, many systems reliably execute mutex programs similar to Figure 2a. Such systems have *fair* schedulers, which do not allow the infinite paths described above. A fairness guarantee is expressed as a temporal predicate on paths and is used to filter out problematic paths before a liveness property, e.g. eventual termination, is considered.

In this work, *weak fairness* [2, p. 258] is considered, which is typically expressed as:

$$\forall t \in T : \Diamond\Box en(t) \implies \Box\Diamond ex(t) \tag{1}$$

Recall that en and ex are both evaluated over a transition α and a tid t. In this case, both predicates are partially evaluated with respected to a t. Intuitively, weak fairness states that if a thread *is able to* execute, then it will *eventually* execute.

▶ **Example 2** (Mutex with weak fairness)**.** We now return to the task of proving termination for the LTS of Figure 2b. If the scheduler provides weak fairness, then we can discard all paths that do not satisfy the weak fairness definition. Namely, the two problematic paths are: $0 \xrightarrow{1} 2, (2 \xrightarrow{0} 2)^\omega$ and $0 \xrightarrow{0} 1, (1 \xrightarrow{1} 1)^\omega$. Neither path satisfies weak fairness: in both cases the thread that can break the cycle is always enabled, yet it is not eventually executed once the infinite cycle begins. Thus, if executed on system which provides weak fairness, the program of Figure 2a is guaranteed to eventually terminate.

3 Formalising semi-fairness

We now detail our formalism for reasoning about fairness properties for semi-fair schedulers. Semi-fairness is parameterised by a thread predicate called the *thread fairness criterion*, or TFC. Intuitively, the TFC states a condition which, if satisfied by a thread t, guarantees fair execution for t.

Formally an execution is semi-fair with respect to a TFC if the following holds:

$$\forall t \in T : \Diamond\Box(en(t) \wedge TFC(t)) \implies \Box\Diamond ex(t) \tag{2}$$

The formula is similar to weak fairness (Eq. 1), but in order for a thread t to be guaranteed eventual execution, not only must t be enabled, but the TFC for t must also hold. Semi-fairness for different schedulers, e.g. HSA and OBE, can be instantiated by using different TFCs, which in turn will yield different liveness properties for programs under these schedulers, e.g. as shown in Table 2.

The weaker the TFC is, the stronger the fairness condition is. Semi-fairness with the the weakest TFC, i.e. true, yields classic weak fairness. Conversely, semi-fairness with the strongest TFC, i.e. false, yields no fairness.

Formalising a specific notion of semi-fairness now simply requires a TFC. We illustrate this by defining TFCs to describe the semi-fair guarantees provided by the OBE and HSA GPU schedulers, introduced informally in Section 1.

Formalising OBE semi-fairness. The prose definition for the OBE scheduler fits this formalism nicely, as it describes the per-thread condition for fair scheduling: once a thread has been scheduled (i.e. executed an instruction), it will continue to be fairly scheduled. This is

straightforward to encode in a TFC using the $\Diamond\!\!\!\!-$ temporal logic operator (see Section 2.2), which holds for a given predicate if that predicate has held at least once in the past. Thus the TFC for the OBE scheduler can be stated formally as follows:

$$TFC_{OBE}(t) = \Diamond\!\!\!\!-\, ex(t) \tag{3}$$

Formalising HSA semi-fairness. A TFC for the HSA scheduler is less straightforward because the prose documentation is given in terms of relative allowed blocking behaviours, rather than in terms of thread-level fairness. Recall the definition from Section 1: thread B can block thread A if: “[thread] A comes after B in [thread] flattened id order” [8, p. 46]. Searching the documentation further, another snippet phrased closer to a TFC is found, stating [8, p. 28]: “[Thread] $i + j$ might start after [thread] i finishes, so it is not valid for a [thread] to wait on an instruction performed by a later [thread].” We assume here that j refers to any positive integer. Because these prose documentation snippets do not discuss fairness explicitly, it is difficult to directly extract a TFC. We make a best-effort attempt following this reasoning: (1) if thread i is fairly scheduled, no thread with id greater than i is guaranteed to be fairly scheduled; and (2) threads that are not enabled (i.e. they have terminated) have no need to be fairly scheduled. Using these two points, we can derive a TFC for HSA: a thread is guaranteed to be fairly scheduled if there does not exist another thread that has a lower id and is enabled. Formally:

$$TFC_{HSA}(t) = \neg\exists t' \in T : (t' < t) \wedge en(t') \tag{4}$$

Although this TFC is somewhat removed from the prose snippets in the HSA documentation, this formal definition has value in enabling precise discussions about fairness. For example, we can increase confidence in our definition by showing that the idioms informally analysed in Section 1 behave as expected; see Examples 3, 4 and 6. Our formalism for HSA provides few progress guarantees, and as we discuss in Section 4, current GPUs appear to offer stronger guarantees than HSA. The HSA programming model may offer such weak guarantees to allow for a variety of devices to be targeted by this programming model and also to allow flexibility in future framework implementations.

▶ **Example 3** (Mutex with semi-fairness). Here we analyse the mutex LTS of Figure 2b under OBE and HSA semi-fairness guarantees. Recall the two problematic paths (causing starvation) are: $0 \xrightarrow{0} 1, (1 \xrightarrow{1} 1)^\omega$. and $0 \xrightarrow{1} 2, (2 \xrightarrow{0} 2)^\omega$

- **OBE**: In both problematic paths, one thread t acquires the mutex, and the other thread t' spins indefinitely. However, thread t has executed an instruction (acquiring the mutex) and is thus guaranteed eventual execution under OBE; the problematic paths violate this guarantee as thread t never executes after it acquires. Therefore both paths are discarded, guaranteeing starvation-freedom for mutexes under OBE.
- **HSA**: The second problematic path: $0 \xrightarrow{1} 2, (2 \xrightarrow{0} 2)^\omega$, cannot be discarded as thread 0 waits for thread 1 to release. Thread 1 does not have the lowest id of the enabled threads, thus there is no guarantee of eventual execution. Therefore starvation-freedom for mutexes cannot be guaranteed under HSA.

▶ **Example 4** (Producer-consumer with semi-fairness). Figure 3 illustrates a two-threaded producer-consumer program. We use a new atomic instruction, `load`, which simply reads a value from memory (the return value is given on the LTS edges). Thread 0 produces a value via `x0` and then spins, waiting to consume a value via `x1`. Thread 1 is similar, but with

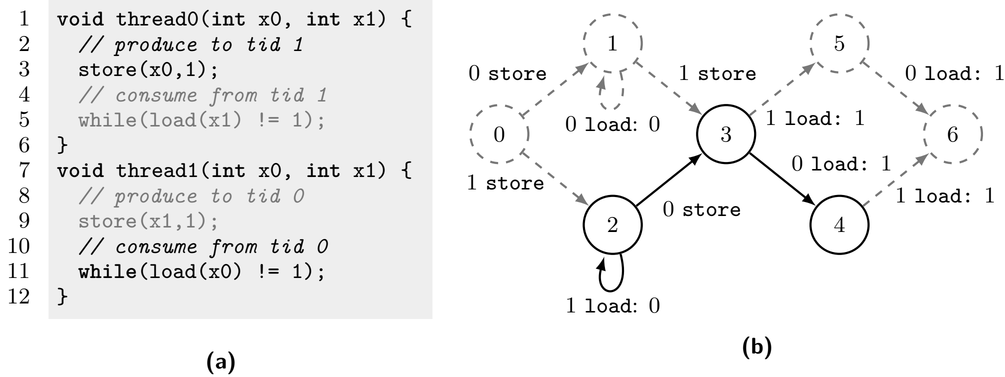

Figure 3 Two threaded PC idiom (a) code and (b) LTS. Omitting in (a) lines in gray and in (b) states and transitions in gray and dashed lines yields the one-way variant of this idiom.

the variables swapped. A subset of this program, omitting lines 4, 5, 8, and 9, shows the *one-way* producer-consumer idiom, where threads only consume from threads with lower ids, i.e., only thread 1 consumes from thread 0. The LTS for the one-way variant omits states 0, 1, 5, and 6 and the start state changes to state 2.

There are two problematic paths for the general test, in which one of the threads spins indefinitely waiting for the other thread to produce a value: $0 \xrightarrow{0} 1, (1 \xrightarrow{0} 1)^\omega$, and $0 \xrightarrow{1} 2, (2 \xrightarrow{1} 2)^\omega$. For the one-way variant, there is one problematic path: $(2 \xrightarrow{1} 2)^\omega$. We now analyse this program under OBE and HSA semi-fairness.

- **OBE**: Consider the problematic path $0 \xrightarrow{1} 2, (2 \xrightarrow{1} 2)^\omega$. Because thread 0 has not executed an instruction, OBE does not guarantee eventual execution for thread 0 and thus this path cannot be discarded. Similar reasoning shows that the problematic path for the one-way variant cannot be discarded either. Thus, neither general nor one-way producer consumer idioms are allowed under OBE.
- **HSA**: Consider the problematic path $0 \xrightarrow{0} 1, (1 \xrightarrow{0} 1)^\omega$. Because thread 1 does not have the lowest id of the enabled threads, HSA does not guarantee eventual execution for thread 1 and this path cannot be discarded. On the other hand, consider the problematic path for the one-way variant: $(2 \xrightarrow{1} 2)^\omega$. Because thread 0 has the lowest id of the enabled threads, HSA guarantees thread 0 will eventually execute, thus causing this path to be invalid. Therefore, general producer-consumer is not allowed under HSA, but one-way producer-consumer, following increasing order of thread ids, is allowed.

4 Inter-workgroup synchronisation in the wild

We now examine existing GPU applications to determine what scheduling guarantees they assume. This provides a basis for understanding (1) what scheduling guarantees are actually provided by existing GPUs, as these applications run without issues on current devices, and (2) the utility of schedulers, i.e. whether their fairness guarantees are exploited in current applications. We limit our exploration to GPU applications that use inter-workgroup synchronisation, and we perform a best-effort search through popular works in this domain. We manually examined the programs, searching for the idioms in Table 1, and relate them to the corresponding scheduler under which they are guaranteed to not starve.

OBE programs. We begin by looking at applications that assume guarantees from the OBE scheduler. The most prevalent example seems to be the occupancy-limited barrier (see Section 1). That is, developers use a priori knowledge about how many threads can be

simultaneously occupant on a given GPU, and only run the program with at most that many threads. The first work on such barriers is a 2010 paper by Xiao and Feng [22]. This work has many citations, many of which describe applications that use the barrier. Additionally, a barrier implementation following this work appears in the popular CUDA library CUB [14]. Thus, the OBE scheduler guarantees appear to be well-tested and useful on current GPUs.

In 2012, Kupta et al. present the *persistent thread model* [6], which more clearly characterises the scheduling guarantees required by the Xiao and Feng Barrier and proposes work-stealing as a potential use case under this model. This work again, has many citations describing use cases. One such work-stealing application that requires OBE scheduling guarantees was published in a 2011 GPU programming cookbook *GPU Computing Gems* [9, ch. 35]. Recent interest in barriers appears in graph analytic applications (e.g. BFS, SSSP), where the 2016 IRGL application benchmark suite is reported to have competitive performance in this domain and uses both barriers and mutexes [17].

HSA programs. We found only four applications that use the one-way PC idiom: two scan implementations, a sparse triangular solve (SpTRSV) application, and a sparse matrix vector multiplication (SpMV) application. While there are few applications in this category, we argue that they are important, as they appear in vendor-endorsed libraries.

The two scan applications, one found in the popular Nvidia CUB GPU library [14] and the second presented in [23], use a straightforward one-way PC idiom. Both scans work by computing workgroup-local scans on independent chunks of a large array. Threads compute chunks according to their thread id, e.g. thread 0 processes the first chunk. A thread t then passes its local sum to its immediate neighbour, thread $t+1$, who spins while waiting for this value. The neighbour factors in this sum and then passes an updated value to its neighbour, and so forth.

The SpMV application, presented in [5], has several workgroups cooperate to calculate the result for a single row. Before any cooperation, the result must first be initialised, which is performed by the workgroup with the lowest id out of the cooperating workgroups. The other workgroups spin, waiting for the initialisation. This algorithm is implemented in the clSPARSE library [1], a joint project between AMD and Vratis.

The SpTRSV application, presented in [13], allows multiple producers to accumulate data to send to a consumer. However, in the triangular solver system, all producers will have lower ids than the relative consumers. Thus the PC idiom remains one-way.

OpenCL programs. We also searched for applications that contain non-trivial inter-workgroup synchronisation and are non-blocking, and thus guaranteed starvation-freedom under any scheduler. By non-trivial synchronisation, we mean inter-workgroup interactions that cannot be achieved by a single atomic read-modify-write (RMW) instruction. While we found examples of non-blocking data-structures (e.g. in the work-stealing example of [9, ch. 35]), the top level loop was blocking as threads without work waited on other threads to complete work. Interestingly, we found only one application that appeared to be globally non-blocking: a reduction application in the CUDA SDK [16], called `threadFenceReduction`, in which the final workgroup to finish local computations also does a final reduction over all other local computations.

5 Unified GPU semi-fairness.

The exploration of applications in Section 4 shows that there are current applications that rely on either HSA or OBE guarantees, and that these applications run without starvation on current GPUs. Hence, it appears that current GPUs provide stronger fairness guarantees than either HSA or OBE describe. In this section, we propose new semi-fairness guarantees that unify HSA and OBE guarantees, and as such, potentially provide a more accurate description of current GPUs scheduling guarantees.

HSA+OBE semi-fairness. A straightforward approach to create a unified fairness property from two existing semi-fair properties is to create a new TFC defined as the disjunction of the two existing TFCs. Thus, threads guaranteed fairness under either existing scheduler are guaranteed fairness under the unified scheduler. We can do this with the HSA and OBE semi-fair schedulers to create a new unified semi-fairness condition, called HSA+OBE, i.e.,

$$TFC_{HSA+OBE}(t) = TFC_{HSA}(t) \vee TFC_{OBE}(t) \quad (5)$$

Thinking about the set of programs for which a scheduler guarantees starvation-freedom, let P_{HSA} be the set of programs allowed under HSA, with P_{OBE} and $P_{HSA+OBE}$ defined similarly. We note that $P_{HSA} \cup P_{OBE} \subset P_{HSA+OBE}$; that is, there are programs in $P_{HSA+OBE}$ that are neither in P_{HSA} nor P_{OBE}. For example, consider a program that uses one-way producer-consumer synchronisation and also a mutex. This program is not allowed under the OBE or HSA scheduler in isolation, but is allowed under the semi-fair scheduler defined as their disjunction. However, this idiom combination seems contrived as the applications discussed in Section 4 that exploit the one-way PC idiom do not require mutexes.

LOBE semi-fairness. The HSA+OBE fairness guarantees are useful for reasoning about existing applications, but these guarantees do not seem like they would naturally be provided by a system scheduler implementation. Namely, HSA+OBE guarantees fairness to (1) the thread with the lowest id that has not terminated (thanks to HSA) and (2) threads that have taken an execution step (thanks to OBE). For example, it might allow relative fair scheduling only between threads 0, 23, 29, and 42, if they were scheduled at least once in the past. Thus, HSA+OBE allows for "gaps", where threads with relative fairness do not have contiguous ids. We believe a more intuitive scheduler would guarantee that threads with relative fairness have contiguous ids.

Given these intuitions, we describe a new semi-fair guarantee, which we call *LOBE* (linear occupancy-bound execution). Similar to OBE, LOBE guarantees fair scheduling to any thread that has taken a step. Additionally, LOBE guarantees fair scheduling to any thread t if another thread t' (1) has taken a step, and (2) has an id greater than or equal to t (hence the word *linear*). Formally, the LOBE TFC can be written:

$$TFC_{LOBE}(t) = \exists t' \in T : \Diamond\, ex(t') \wedge t' \geq t \quad (6)$$

We will now show that LOBE is a unified scheduler, i.e. any program allowed under HSA or OBE is allowed under LOBE. It is sufficient to show that $TFC_{OBE} \implies TFC_{LOBE}$ and $TFC_{HSA} \implies TFC_{LOBE}$. First, we consider $TFC_{OBE} \implies TFC_{LOBE}$: this is trivial as the comparison check in TFC_{LOBE} includes equality, thus any thread that has taken a step is guaranteed to be fairly scheduled.

Considering now $TFC_{HSA} \implies TFC_{LOBE}$: we first recall a property of executions from Section 2.2, namely that an execution either ends in a state where all threads have terminated, or it is infinite. Thus, at an arbitrary non-terminal point in an execution, some

thread t must take a step. If t has the lowest id of the enabled threads, then both LOBE and HSA guarantee that t will be fairly executed. If t does not have the lowest id of the enabled threads, then LOBE guarantees that *all* threads with lower ids than t will be fairly executed, including the thread with the lowest id of the enabled threads, thus satisfying the fairness constraint of HSA.

5.1 LOBE discovery protocol

Because the $TFC_{HSA+OBE}$ is defined as the disjunction of TFC_{HSA} and TFC_{OBE}, the reasoning in Section 5 is sufficient to show that LOBE fairness guarantees are at least as strong as HSA+OBE. A practical GPU program is now discussed for which correctness relies on the stronger guarantees provided by LOBE compared to HSA+OBE. This example shows that (1) LOBE guarantees are strictly stronger than HSA+OBE, and (2) fairness guarantees exclusive to LOBE can be useful in GPU applications.

Our example is a modified version of the discovery protocol from [20], which dynamically discovers threads that are guaranteed to be co-occupant, and are thus guaranteed relative fairness by OBE. The protocol works using a virtual *poll*, in which threads have a short time window to indicate, using shared memory, that they are co-occupant. The protocol acts as a filter: discovered co-occupant threads execute a program, and undiscovered threads exit without performing any meaningful computation. Because only co-occupant threads execute the program, OBE guarantees that blocking idioms such as barriers can be used reliably.

GPU programs are often data-parallel, and threads use their ids to efficiently partition arrays; thus having contiguous ids is vital. Because OBE fairness does not consider thread ids, in order to provide contiguous ids, the discovery protocol dynamically assigns *new* ids to discovered threads. While functionally this approach is sound, there are two immediate drawbacks: (1) programs must be written using new thread ids, which can require intrusive changes, and (2) the native thread id assignment on GPUs may be optimised by the driver for memory accesses in data-parallel programs; using new ids would forgo these optimisations. Exploiting the scheduling guarantees of LOBE, we modify the discovery protocol to preserve native thread ids and also ensuring contiguous ids.

▶ **Example 5** (thread ids and data locality)**.** It is possible that the protocol discovers four threads (with tids 2-5) and creates the following mapping for their new dynamic ids: $\{(5 \to 0), (2 \to 1), (3 \to 3), (4 \to 4)\}$. The GPU runtime might have natively assigned threads 2 and 3 to one processor (often called a compute unit on GPUs) and threads 4 and 5 to another. Because these compute units often have caches, data-locality between threads on the same compute unit could offer performance benefits [21]. In data-parallel programs, there is often data-locality between threads with consecutive ids. Thus, in our example mapping, the (native) threads, 2 and 5 could not exploit data locality, as their new ids are consecutive, but their native ids are not.

While it may seem straightforward to remap the ids of discovered threads to facilitate data locality, as is done in [21], we note that this depends on the ability to query the physical core id of a thread. Nvidia provides this functionality in CUDA, which is exploited in [21], but OpenCL offers no support for such a feature. Thus, the relation between thread ids and data locality is hidden by the OpenCL framework. We assume the natively assigned ids take data locality into account and that dynamically assigned ids might not be as efficient.

We show the algorithm for the discovery protocol in Algorithm 1. The changes we make to exploit LOBE guarantees are indicated by dashed boxes for removed code and solid boxes for added code. We first describe the original protocol. The algorithm has two phases, both protected by the mutex m. The first phase is the *polling phase* (lines 2-10), where threads are able to indicate that they are currently occupant (i.e. executing). The

Algorithm 1 Occupancy discovery protocol. Applying LOBE optimisation removes the code in dashed boxes and adds the code in solid boxes.

1: **function** DISCOVERY_PROTOCOL($open, count,$ $id_map,$ m)
2: Lock(m)
3: **if** $open$ $\vee(tid < count)$ **then**
4: $id_map[tid] \leftarrow count$
5: $count \leftarrow count + 1$
6: $count \leftarrow max(count, tid + 1)$
7: Unlock(m)
8: **else**
9: Unlock(m)
10: **return** $False$
11: Lock(m)
12: **if** $open$ **then**
13: $open \leftarrow False$
14: Unlock(m)
15: **return** $True$

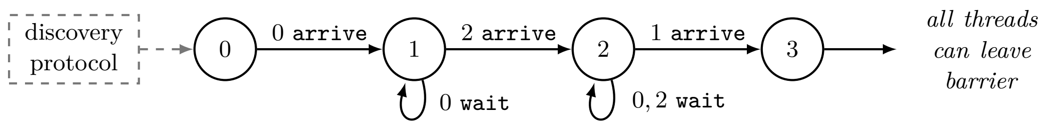

Figure 4 Sub-LTS of a barrier, with an optional discovery protocol preamble.

open shared variable is initialised to true to indicate that the poll is open. A thread first checks whether the poll is open (line 3). If so, then the thread marks itself as discovered; this involves obtaining a new id (line 4) and incrementing the number of discovered threads, via the shared variable *count* (line 5). The thread can then continue to the closing phase (starting line 11). If the poll was not open, the thread indicates that it was not discovered by returning false (lines 8-10). In the closing phase, a thread checks to see if the poll is open; if so, the thread closes the poll and no other threads can be discovered at this point (lines 12-13). All threads who enter the closing phase have been discovered to be co-occupant, thus they return true (line 15). The number of discovered threads will be stored in *count*.

We can optimise this protocol by exploiting fairness guarantees of LOBE. In particular, because LOBE guarantees that threads are fairly scheduled in contiguous id order, the protocol can allow a thread with a higher id to *discover* all threads with lower ids. As a result, threads are able to keep their native ids, although the number of discovered threads is still dynamic. The optimisation to the discovery protocol is simple: first the *id_map*, which originally mapped threads to their new dynamic ids is not needed (lines 1 and 4). Next, the number of discovered threads is no longer based on how many threads were observed to poll, but rather on the highest id of the discovered threads (line 6). Finally, even if the poll is closed, a thread entering the poll may have been discovered by a thread with a higher id; this is now reflected by each thread comparing its id with *count* (line 3). In Example 6, we show that a barrier prefaced by the LOBE optimised protocol is not allowed under HSA+OBE guarantees, and thus illustrate that LOBE fairness guarantees are strictly stronger than HSA+OBE.

▶ **Example 6** (Barriers under semi-fairness). We now analyse the behaviour of barriers, with optional discovery protocols, under our semi-fair schedulers. Figure 4 shows a subset of an LTS for a barrier idiom that synchronises three threads with tids 0, 1, and 2. For the sake of clarity, instead of using atomic actions that correspond to concrete GPU atomic instructions, we use abstract instructions `arrive` and `wait`, which correspond to a thread marking its arrival at the barrier and a thread waiting at the barrier, respectively.

The sub-LTS shows one possible interleaving of threads arriving at the barrier, in the order 0, 1, 2. The final thread to arrive (thread 1) allows all threads to leave. The sub-LTS shows the various spin-waiting scenarios that can occur in a barrier at states 1 and 2. A discovery protocol can optionally be used before the barrier synchronisation.

We analyse the sub-LTS using the LOBE optimised discovery protocol (Section 5.1) here. A similar analysis for the general barrier and original discovery protocol is done in Appendix A. Recall that the LOBE discovery protocol discovers a thread if it has seen a step from a thread with an equal or greater id. In our example with three threads, the fewest behaviours the protocol is guaranteed to have seen is a step by thread 2, denoted: $DP \xrightarrow{2} 0$.

- **HSA+OBE**: consider the starvation path: $DP \xrightarrow{2} 0, 0 \xrightarrow{0} 1, 1 \xrightarrow{0} 2, (2 \xrightarrow{0} 2, 2 \xrightarrow{2} 2)^\omega$. This path cannot be disallowed by HSA+OBE as at state 2, HSA+OBE guarantees fair scheduling for the thread with the lowest id (thread 0) and any threads that have taken a step (threads 0 and 2). This path requires fair execution from thread 1 to break the starvation loop. Thus, barrier synchronisation using the LOBE discovery protocol is not allowed under HSA+OBE.
- **LOBE**: The above starvation path is disallowed by LOBE, as LOBE guarantees fair execution for any thread t that has executed *and* any thread with a lower id than t. Thus, at state 0, the LOBE discovery protocol has observed a step from thread 2, we are guaranteed fair scheduling for threads 2, 1, and 0. Thus barriers with LOBE discovery protocol are allowed under LOBE.

6 Conclusion

While general purpose usage of GPUs is on the rise, current GPU programming models provide loose scheduling fairness guarantees in English prose. In practice, GPUs feature *semi-fair* schedulers. Our goal is to clarify the fairness guarantees that GPU programmers can rely on, or at least the ones they assume. To this aim, we have introduced a formalism that combines the classic weak fairness with a thread fairness criterion (TFC), enabling fairness to be specified at a per-thread level. We have illustrated this formalism by defining the TFC for HSA (from its specification) and OBE (from its description based on empirical evidence) and by reasoning with such TFCs on three classic concurrent programming idioms: barrier, mutex and producer-consumer.

We notice that while some popular existing GPU programs rely on either HSA or OBE guarantees, these two models are not comparable, hence current GPUs must support stronger guarantees that neither HSA nor OBE entirely capture. Our formalism lets easily combine the TFCs of HSA and OBE to define the HSA+OBE scheduler; and we additionally craft the LOBE scheduler which offers slightly stronger fairness guarantees than HSA+OBE. We illustrate that LOBE guarantees can be useful by showing a GPU protocol optimisation for which other GPU semi-fair schedulers do not guarantee starvation-freedom, but LOBE does.

References

1 clSPARSE. Retrieved June 2018 from `https://github.com/clMathLibraries/clSPARSE`.

2 Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

3 Blaise Barney. POSIX threads programming: Condition variables. (visited January 2018). URL: `https://computing.llnl.gov/tutorials/pthreads/#ConditionVariables`.

4. Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. The design and implementation of a verification technique for gpu kernels. *TOPLAS*, 37(3):10:1–10:49, 2015.
5. M. Daga and J. L. Greathouse. Structural agnostic SpMV: Adapting CSR-adaptive for irregular matrices. In *HiPC*, pages 64–74. IEEE, 2015.
6. Kshitij Gupta, Jeff Stuart, and John D. Owens. A study of persistent threads style GPU programming for GPGPU workloads. In *InPar*, pages 1–14, 2012.
7. Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., 2008.
8. HSA Foundation. HSA programmer's reference manual: HSAIL virtual ISA and programming model, compiler writer, and object format (BRIG). (rev 1.1.1), March 2017. URL: `http://www.hsafoundation.com/standards/`.
9. Wen-mei W. Hwu. *GPU Computing Gems Jade Edition.* Morgan Kaufmann, 2011.
10. Khronos Group. The OpenCL C specification version 2.0 (rev. 33), May 2017. URL: `https://www.khronos.org/registry/OpenCL/specs/opencl-2.0-openclc.pdf`.
11. Khronos Group. The OpenCL specification version: 2.2 (rev. 2.2-7), May 2018. URL: `https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf`.
12. Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. The glory of the past. In *Logics of Programs*, pages 196–218. Springer Berlin Heidelberg, 1985.
13. Weifeng Liu, Ang Li, Jonathan Hogg, Iain S. Duff, and Brian Vinter. A synchronization-free algorithm for parallel sparse triangular solves. In *Euro-Par*, pages 617–630. Springer, 2016.
14. Nvidia. CUB. (visited January 2018). URL: `http://nvlabs.github.io/cub/`.
15. Nvidia. CUDA C programming guide, version 9.1, January 2018. URL: `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.
16. Nvidia. CUDA Code Samples, 2018. URL: `https://developer.nvidia.com/cuda-code-samples`.
17. Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on GPUs. In *OOPSLA*, pages 1–19, 2016.
18. Tyler Sorensen and Alastair F. Donaldson. Exposing errors related to weak memory in GPU applications. In *PLDI*, pages 100–113. ACM, 2016.
19. Tyler Sorensen and Alastair F. Donaldson. The hitchhiker's guide to cross-platform OpenCL application development. In *IWOCL*, pages 2:1–2:12, 2016.
20. Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. Portable inter-workgroup barrier synchronisation for GPUs. In *OOPSLA*, pages 39–58, 2016.
21. Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *ICS*, pages 119–130. ACM, 2015.
22. Shucai Xiao and Wu-chun Feng. Inter-block GPU communication via fast barrier synchronization. In *IPDPS*, pages 1–12, 2010.
23. Shengen Yan, Guoping Long, and Yunquan Zhang. Streamscan: Fast scan algorithms for GPUs without global barrier synchronization. In *PPoPP*, pages 229–238. ACM, 2013.

A Barrier example cont.

We continue the analysis of the barrier sub-LTS of Figure 4 that was started in Example 6. That is, we analyse the general barrier (i.e. with no discovery protocol) and the the barrier using the original discovery protocol (as described in Section 5.1).

- **general barrier**:
  - *LOBE* - The starvation path $0 \xrightarrow{0} 1, (1 \xrightarrow{0} 1)^\omega$ is not disallowed by LOBE, as LOBE cannot guarantee fair execution for any thread other than thread 0 at state 1 where the infinite starvation path begins. Thus, general barriers are not allowed under LOBE. Because LOBE is stronger than HSA+OBE, HSA and OBE, we know that the general barrier is not allowed under these schedulers either.
- **original discovery protocol:** This discovery protocol ensures that all three threads, i.e. threads 0, 1, and 2, have taken a step before state 0. We denote this transition as $DP \xrightarrow{0,1,2} 0$.
  - *HSA* - The starvation path $DP \xrightarrow{0,1,2} 0, 0 \xrightarrow{0} 1, (1 \xrightarrow{0} 1)^\omega$ is not disallowed by HSA, as HSA only guarantees fair execution to the lowest enabled thread (i.e. thread 0). To break this starvation loop in the sub LTS, thread 2 would need fairness guarantees. Thus barriers using the original discovery protocol are not allowed under HSA.
  - *OBE* - Because the original discovery protocol guarantees all threads have taken a step before the barrier execution (i.e. state 0), OBE guarantees all three threads fair scheduling. Thus all starvation loops in the sub LTS are guaranteed to be broken, and the barrier using the original discovery protocol is allowed under OBE. Because HSA+OBE and LOBE are stronger than OBE, this synchronisation idiom is also allowed under those schedulers.