Interleaving and Lock-Step Semantics for Analysis and Verification of GPU Kernels[*]

Peter Collingbourne[1,**], Alastair F. Donaldson[1], Jeroen Ketema[1], and Shaz Qadeer[2]

[1] Imperial College London
peter@pcc.me.uk, {afd,jketema}@imperial.ac.uk
[2] Microsoft Research
qadeer@microsoft.com

Abstract. We study semantics of GPU kernels — the parallel programs that run on Graphics Processing Units (GPUs). We provide a novel lock-step execution semantics for GPU kernels represented by *arbitrary* reducible control flow graphs and compare this semantics with a traditional interleaving semantics. We show for terminating kernels that either both semantics compute identical results or both behave erroneously.

The result induces a method that allows GPU kernels with arbitrary reducible control flow graphs to be verified via transformation to a sequential program that employs predicated execution. We implemented this method in the GPUVerify tool and experimentally evaluated it by comparing the tool with the previous version of the tool based on a similar method for *structured programs*, i.e., where control is organised using **if** and **while** statements. The evaluation was based on a set of 163 open source and commercial GPU kernels. Among these kernels, 42 exhibit unstructured control flow which our novel method can handle fully automatically, but the previous method could not. Overall the generality of the new method comes at a modest price: Verification across our benchmark set was 2.25 times slower overall; however, the median slow down across all kernels was 0.77, indicating that our novel technique yielded faster analysis in many cases.

1 Introduction

Graphics Processing Units (GPUs) have recently found application in accelerating general-purpose computations, e.g., in image retrieval [23] and machine learning [3]. If an application exhibits significant parallelism it may be possible to extract the computational core of the application as a *kernel* and offload this kernel to run across the parallel hardware of a GPU, sometimes beating CPU performance by orders of magnitude. Writing kernels for massively parallel GPUs is challenging, requiring coordination of a large number of threads. Data races and mis-synchronisation at barriers (known as *barrier divergence*) can lead to erroneous and non-deterministic program behaviours. Worse, they can lead to bugs which manifest only on some GPU architectures.

Substantial effort has been put into the design of tools for rigorous analysis of GPU kernels [7,16,8,17,15]. In prior work [7], we presented a verification technique and tool, GPUVerify, for analysis of data races and barrier divergence in OpenCL [13] and

[*] This work was supported by the EU FP7 STREP project CARP (project number 287767).

[**] Peter Collingbourne is currently employed at Google.

M. Felleisen and P. Gardner (Eds.): ESOP 2013, LNCS 7792, pp. 270–289, 2013.

CUDA [21] kernels. GPUVerify achieves scalability by reducing verification of a parallel kernel to a *sequential* program verification task. This is achieved by transforming kernels into a form where all threads execute in *lock-step* in a manner that still facilitates detection of data races and barrier divergence arising due to arbitrary thread interleavings.

Semantics and program transformations for lock-step execution have been formally studied for *structured* GPU kernels where control flow is described by **if** and **while** constructs [7]. In this setting, the hierarchical structure of a program gives rise to a simple, recursive algorithm for transforming control flow into *predicated* form so that all threads execute the same sequence of statements. Lock-step semantics for GPU kernels where control flow is described by an *arbitrary* reducible control flow graph (CFG),[1] has *not* been studied. Unlike structured programs, arbitrary CFGs do not necessarily exhibit a hierarchical structure, thus the existing predication-based approach cannot be directly extended. Furthermore, it is not possible in to efficiently pre-process an arbitrary CFG into a structured form [9].

The restriction to structured programs poses a serious limitation to the design of GPU kernel analysis techniques: kernels frequently exhibit unstructured control flow, either directly, e.g., through **switch** statements, or indirectly, through short-circuit evaluation of Boolean expressions. Dealing with CFGs also enables analysis of GPU kernels after compiler optimisations have been applied, bringing the analysis closer to the code actually executed by the GPU. It allows for the reuse of existing compiler infrastructures, such as Clang/LLVM, which use CFGs as their intermediate representation. Reusing compiler infrastructures hugely simplifies tool development, removing the burden of writing a robust front-end for C-like languages.

We present a traditional interleaving semantics and a novel lock-step semantics for GPU kernels described by CFGs. We show that if a GPU kernel is guaranteed to terminate then the kernel is correct with respect to the interleaving semantics if and only if it is correct with respect to the lock-step semantics, where *correct* means that all execution traces are free from data races, barrier divergence, and assertion failures. Our novel lock-step semantics enables the strategy of reducing verification of a multithreaded GPU kernel to verification of a sequential program to be applied to arbitrary GPU kernels, and we have implemented this method in the GPUVerify tool. We present an experimental evaluation, applying our new tool to a set of 163 open source and commercial GPU kernels. In 42 cases these kernels exhibited unstructured control flow either (a) explicitly (e.g., through **switch** statements), or (b) implicitly due to short-circuit evaluation. In the case of (a), these kernels had to be manually simplified to be amenable to analysis using the original version of GPUVerify. In the case of (b), it turned out that the semantics of short-circuit evaluation of logical operators was not handled correctly in GPUVerify. Our new, more general implementation handles all these kernels accurately and automatically. Our results show that GPUVerify continues to perform well: compared to the original version that was limited to structured kernels [7], verification across our

[1] Henceforth, whenever we refer to a CFG we shall always mean a reducible CFG. For a definition of reducibility we refer the reader to [1]. We note that irreducibility is uncommon in practice. In particular, we have never encountered a GPU kernel with an irreducible control flow graph, and whether irreducible control flow is supported at all is implementation-defined in OpenCL [13].

```
__kernel void
scan(__global int *sum) {
  int offset = 1, temp;
  while (offset < TS) {
    if (tid >= offset)
      temp = sum[tid - offset];
    barrier();
    if (tid >= offset)
      sum[tid] = sum[tid] + temp;
    barrier();
    offset *= 2;
  }
}
```

(a) A correct kernel

```
__kernel void
scan(__global int *sum) {
  int offset = 1, temp;
  while (offset <= tid) {
    temp = sum[tid - offset];
    barrier();
    sum[tid] = sum[tid] + temp;
    barrier();
    offset *= 2;
  }
}
```

(b) A kernel with barrier divergence

Fig. 1. Two OpenCL kernels

benchmark set was 2.25 times slower overall, but the median slow down across all kernels was 0.77, indicating that our novel technique yields faster analysis in many cases.

In summary, our main contributions are:

- A novel operational semantics for lock-step execution of GPU kernels with arbitrary reducible control flow.
- A proof-sketch that this semantics is equivalent to a traditional interleaving semantics for terminating GPU kernels.
- A revised implementation of GPUVerify which uses our lock-step semantics to reduce verification of a multithreaded kernel to a sequential verification task.

After presenting a small example to provide some background on GPU kernels and illustrate the problems of data races and barrier divergence (Sect. 2), we present the interleaving semantics (Sect. 3), our novel lock-step semantics (Sect. 4) and a proof-sketch showing that the semantics are equivalent for terminating kernels (Sect. 5). We then discuss the implementation in GPUVerify, and present our experimental results (Sect. 6). We end with related work and conclusions (Sect. 7).

2 A Background Example

We use an example to illustrate the key concepts from GPU programming and provide an informal description as to how predicated lock-step execution works for structured programs. We return to this example when presenting interleaving and lock-step semantics for kernels described as CFGs in Sects. 3 and 4.

Threads, Barriers, and Shared Memory. Figure 1a shows an OpenCL kernel[2] to be executed by TS threads, where TS is a power of two. The kernel implements a *scan* (or *prefix-sum*) operation on the sum array so that at the end of the kernel we have, for all $0 \leq i < TS$, $\mathrm{sum}[i] = \Sigma_{j=0}^{i} \mathbf{old}(\mathrm{sum})[j]$, where $\mathbf{old}(\mathrm{sum})$ refers to the sum array at the start of the kernel. All threads execute this kernel function in parallel,

[2] For ease of presentation we use a slightly simplified version of OpenCL syntax, and we assume that all threads reside in the same work group and that this work group is one dimensional. Our implementation, described in Sect. 6, supports OpenCL in full.

and threads may follow different control paths or access distinct data by querying their unique thread id, tid. Communication is possible via shared memory; the sum array is marked as residing in global shared memory via the __global qualifier. Threads synchronise using a barrier-statement, a collective operation that requires all threads to reach the same syntactic barrier before any thread proceeds past the barrier.

Data Races and Barrier Divergence. Two common defects from which GPU kernels suffer are *data races* and *barrier divergence*. In Fig. 1a, accesses to sum inside the loop are guarded so that on loop iteration i only threads with id at least 2^{i-1} access the sum array. If either of the barriers in the example were omitted the kernel would be prone to a *data race* arising due to thread t_1 reading from $\texttt{sum}[t_1 - \texttt{offset}]$, while thread t_2 writes to $\texttt{sum}[t_2]$, where $t_2 = t_1 - \texttt{offset}$. The kernel of Fig. 1b aims to optimise the original example by reducing branches inside the loop: threads are restricted to only execute the loop body if their id is sufficiently large. This optimisation is *erroneous*; given a barrier inside a loop, the OpenCL standard requires that either all threads or zero threads reach the barrier during a given loop iteration, otherwise *barrier divergence* occurs and behaviour is undefined. In Fig. 1b, thread 0 will not enter the loop at all and thus will *never* reach the first barrier, while all other threads will enter the loop and reach the barrier. Unfortunately, on an NVIDIA 9400M the kernel of Fig. 1b behaves identically to the kernel of Fig. 1a, meaning that this barrier divergence bug would not be detected on this platform. This is problematic because the erroneous kernel code is not portable across architectures which support OpenCL (e.g., the kernel fails to produce correct results with Intel's SDK for OpenCL).

Lock-Step Predicated Execution. We informally describe lock-step execution for structured programs as used by GPUVerify [7] and which we here generalise to CFGs.

To achieve lock-step execution, GPUVerify transforms kernels into *predicated* form [2]. The example of Fig. 2 illustrates the effect of applying predication to the kernel of Fig. 1a. A statement of the form $e \Rightarrow \mathit{stmt}$ is a *predicated* statement which is a no-op if e is false, and has the same effect as *stmt* if e is true. Observe that the **if** statements in the body of the loop have been predicated: the condition (which is the same for both statements) is evaluated into a Boolean variable q, the conditional statements are removed and the statements previously inside the conditionals are predicated by the associated Boolean variable. Predication of the **while** loop is achieved by evaluating the loop condition into a Boolean variable p, predicating all statements in the loop body by p, and recomputing p at the end of the loop body. The loop condition is replaced by a guard which evaluates to false if and only if the predicate variable p is false for *every* thread. Thus all threads continue to execute the loop until each thread is ready to leave the loop; when the loop condition becomes false for a given thread the thread simply performs no-ops during subsequent loop iterations.

```
__kernel void
scan(__global int *sum) {
 bool p, q;
 int offset = 1, temp;
 p = (offset < TS);
 while (∃ t :: t.p) {
   q = (p && tid >= offset);
   q ⇒ temp = sum[tid - offset];
   p ⇒ barrier();
   q ⇒ sum[tid] = sum[tid] + temp;
   p ⇒ barrier();
   p ⇒ offset *= 2;
   p ⇒ p = (offset < TS);
 }
}
```

Fig. 2. Lock-step predicated execution for structured kernel of Fig. 1a

In predicated form, the threads do not exhibit any diverging behaviour due to execution of different branches, and thus the kernel can be regarded as a *sequential*, vector program. GPUVerify exploits this fact to reduce GPU kernel verification to a sequential program verification task. The full technique, described in [7], involves considering lock-step execution of an arbitrary *pair* of threads, rather than all threads.

The example illustrates that predication is easy to perform at the level of structured programs built hierarchically using **if** and **while** statements. However, predication does not directly extend to the unstructured case, and unstructured control flow cannot be efficiently pre-processed into structured form [9]. Hence, we present a program transformation for predicated execution of GPU kernels described as CFGs.

3 Interleaving Semantics for GPU Kernels

We introduce a simple language for describing GPU kernels as CFGs.

3.1 Syntax

A kernel is defined over a set of variables $\mathit{Var} = V_s \uplus V_p$ with V_s the *shared* variables and V_p the *private* variables. Variables take values from a domain D. Kernels are expressed using a syntax that is identical to the core of the Boogie programming language [5], except that it includes an additional **barrier** statement:

$$\begin{aligned}
\mathit{Program} &::= \mathit{Block}^+\\
\mathit{Block} &::= \mathit{BlockId} : \mathit{Stmts}\ \mathbf{goto}\ \mathit{BlockId}^+ \ ;\\
\mathit{Stmts} &::= \varepsilon \mid \mathit{Stmt} \,;\, \mathit{Stmts}\\
\mathit{Stmt} &::= \mathit{Var} := \mathit{Expr} \mid \mathbf{havoc}\ \mathit{Var} \ \mid \mathbf{assume}\, \mathit{Expr} \mid \mathbf{assert}\ \mathit{Expr} \mid \mathbf{skip} \ \mid \mathbf{barrier}
\end{aligned}$$

Here, ε is an empty sequence of statements. The form of expressions is irrelevant, except that we assume (a) *equality testing* $(=)$, (b) the standard Boolean operators, and (c) a ternary operator $\mathit{Expr}_1 \,?\, \mathit{Expr}_2 : \mathit{Expr}_3$, which — like the operator from C — evaluates to the result of Expr_2 if Expr_1 is true and to the result of Expr_3 otherwise.

Thus, a kernel consists of a number of *basic blocks*, with each block consisting of a number of *statements* followed by a **goto** that non-deterministically chooses which block to execute next based on the provided $\mathit{BlockId}$s; non-deterministic choice in combination with **assume**s at the beginning of blocks is used to model branching.

Because gotos only appear at the end of blocks there is a one-to-one correspondence between kernels and CFGs. We assume that all kernels have reducible CFGs, which means that cycles in a CFG are guaranteed to form *natural loops*. A natural loop has a unique *header* node, the single entry point to the loop, and one or more *back edges* going from a loop node to the header [1].

We assume that each block in a kernel is uniquely labelled and that there is a block labelled Start. This is the block from which execution of each thread commences. Moreover, no block is labelled End; instead the occurrence of End in a **goto** signifies that the program may terminate at this point. The first statement of a block is always an **assume** and only variables from V_p appear in the guard of the **assume**. No other **assume**s occur in blocks and the first statement of Start is **assume** true. Observe that any kernel can be easily pre-processed to satisfy these restrictions.

```
$Start$      : $offset := 1$ ;
               goto $W, W_{end}$ ;
$W$          : assume $offset < TS$ ;
               goto $I_1, I'_1$ ;
$I_1$        : assume $tid \geq offset$ ;
               $temp := sum[tid - offset]$ ;
               goto $B_1$ ;
$I'_1$       : assume $tid < offset$ ;
               goto $B_1$ ;
$B_1$        : barrier ;
               goto $I_2, I'_2$ ;
$I_2$        : assume $tid \geq offset$ ;
               $sum[tid] := sum[tid] + temp$ ;
               goto $B_2$ ;
$I'_2$       : assume $tid < offset$ ;
               goto $B_2$ ;
$B_2$        : barrier ;
               goto $W_{last}$ ;
$W_{last}$   : $offset := 2 \cdot offset$ ;
               goto $W, W_{end}$ ;
$W_{end}$    : assume $offset \geq TS$ ;
               goto $End$ ;
```

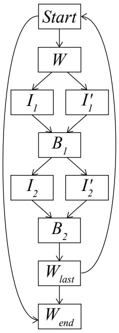

Fig. 3. The kernel of Fig. 1a encoded in our kernel language and its CFG

Figure 3 shows the kernel of Fig. 1a encoded in our simple programming language, where we omit **assume** *true* for brevity. Remark that an array is being used; we could easily add arrays to our GPU kernel semantics but, again for brevity, we do not.

3.2 Operational Semantics

We now define a small-step operational semantics for our kernel programming language, which is based on interleaving the steps taken by individual threads.

Individual Threads. The behaviour of individual threads and the non-barrier statements executed by these threads is presented in Figs. 4a and 4b.

The operational semantics of a thread t is defined in terms of triples $\langle \sigma, \sigma_t, b_t \rangle$, where $\sigma : V_s \nrightarrow D$ is the *shared* store, $\sigma_t : V_p \nrightarrow D$ is the *private* store of thread t, and b_t is the statement or sequence of statements the thread will *reduce* (i.e., execute) next.

In Fig. 4a, $(\sigma, \sigma_t)[v \mapsto val]$ denotes a pair of stores equal to (σ, σ_t) except that v (which we assume occurs in either σ or σ_t) has been updated and is equal to val. The evaluation of an expression e given (σ, σ_t) is denoted $(\sigma, \sigma_t)(e)$. The labels on arrows allow us to observe (a) changes to stores and (b) the state of stores upon termination. A label is omitted when the stores do not change, e.g., in the case of the SKIP rule.

The symbols $\surd$, $\mathcal{E}$, and $\bot$ indicate, resp., *termination*, *error*, and *infeasible*. These are *termination statuses* which signify that a thread (or later kernel) has terminated with that particular status. Below, termination always means termination with status *termination*; termination with status *error* or *infeasible* is indicated explicitly.

The ASSIGN and SKIP rules of Fig. 4a are standard. The HAVOC rule updates the value of a variable v with an arbitrary value from the domain D of v. The ASSERT$_\mathrm{T}$ and ASSUME$_\mathrm{T}$ rules are no-ops if the assumption or assertion $(\sigma, \sigma_t)(e)$ holds. If the assumption or assertion does not hold, ASSERT$_\mathrm{F}$ and ASSUME$_\mathrm{F}$ yield, resp., $\mathcal{E}$ and $\bot$.

In Fig. 4b, s denotes a statement and b denotes the *body of a block*, i.e., a sequence of statements followed by a **goto**. The SEQ$_\mathrm{B}$ and SEQ$_\mathrm{E,I}$ rules define reduction of $s \,;\, b$ in terms of reduction of s. The GOTO and BLOCK rules specify how reduction continues once the end of a block is reached. The END rule specifies termination of a thread.

$$\frac{val = (\sigma, \sigma_t)(e)}{P \vdash \langle \sigma, \sigma_t, v := e \rangle \overset{(\sigma,\sigma_t)}{\rightarrow} (\sigma, \sigma_t)[v \mapsto val]}\ \textsc{Assign}$$

$$\frac{val \in D}{P \vdash \langle \sigma, \sigma_t, \mathbf{havoc}\, v \rangle \overset{(\sigma,\sigma_t)}{\rightarrow} (\sigma, \sigma_t)[v \mapsto val]}\ \textsc{Havoc}$$

$$\frac{a \in \{\mathbf{assert}, \mathbf{assume}\} \qquad (\sigma, \sigma_t)(e)}{P \vdash \langle \sigma, \sigma_t, a\, e \rangle \rightarrow (\sigma, \sigma_t)}\ \begin{array}{l}\textsc{Assert}_{\text{T}}\\ \textsc{Assume}_{\text{T}}\end{array} \qquad \frac{\neg(\sigma, \sigma_t)(e)}{P \vdash \langle \sigma, \sigma_t, \mathbf{assume}\, e \rangle \overset{(\sigma,\sigma_t)}{\rightarrow} \bot}\ \textsc{Assume}_{\text{F}}$$

$$\frac{\neg(\sigma, \sigma_t)(e)}{P \vdash \langle \sigma, \sigma_t, \mathbf{assert}\, e \rangle \overset{(\sigma,\sigma_t)}{\rightarrow} \mathcal{E}}\ \textsc{Assert}_{\text{F}} \qquad \frac{}{P \vdash \langle \sigma, \sigma_t, \mathbf{skip} \rangle \rightarrow (\sigma, \sigma_t)}\ \textsc{Skip}$$

(a) Statement rules

$$\frac{P \vdash \langle \sigma, \sigma_t, s \rangle \overset{(\sigma,\sigma_t)}{\rightarrow} (\tau, \tau_t)}{P \vdash \langle \sigma, \sigma_t, s\,;\, b \rangle \overset{(\sigma,\sigma_t)}{\rightarrow} \langle \tau, \tau_t, b \rangle}\ \textsc{Seq}_{\text{B}} \qquad \frac{P \vdash \langle \sigma, \sigma_t, s \rangle \overset{(\sigma,\sigma_t)}{\rightarrow} e \qquad e \in \{\mathcal{E}, \bot\}}{P \vdash \langle \sigma, \sigma_t, s\,;\, b \rangle \overset{(\sigma,\sigma_t)}{\rightarrow} e}\ \textsc{Seq}_{\text{E,I}}$$

$$\frac{1 \leq i \leq n}{P \vdash \langle \sigma, \sigma_t, \mathbf{goto}\, B_1, \ldots, B_n\,; \rangle \rightarrow \langle \sigma, \sigma_t, B_i \rangle}\ \textsc{Goto} \qquad \frac{(B : b) \in P}{P \vdash \langle \sigma, \sigma_t, B \rangle \rightarrow \langle \sigma, \sigma_t, b \rangle}\ \textsc{Block}$$

$$\frac{}{P \vdash \langle \sigma, \sigma_t, End \rangle \overset{\sigma,\sigma_t}{\rightarrow} \surd}\ \textsc{End}$$

(b) Thread rules

$$\frac{T_{\vec{\sigma}}|_t = \langle \sigma_t, b_t \rangle \qquad P \vdash \langle \sigma, \sigma_t, b_t \rangle \overset{(\sigma,\sigma_t)}{\rightarrow} \langle \tau, \tau_t, c_t \rangle}{P \vdash \langle \sigma, T_{\vec{\sigma}} \rangle \overset{(\sigma,\vec{\sigma})}{\rightarrow} \langle \tau, T_{\vec{\sigma}}[\langle \tau_t, c_t \rangle]_t \rangle}\ \textsc{Thread}_{\text{B}}$$

$$\frac{T_{\vec{\sigma}}|_t = \langle \sigma_t, b_t \rangle \qquad P \vdash \langle \sigma, \sigma_t, b_t \rangle \overset{(\sigma,\sigma_t)}{\rightarrow} \surd}{P \vdash \langle \sigma, T_{\vec{\sigma}} \rangle \overset{(\sigma,\vec{\sigma})}{\rightarrow} \langle \sigma, T_{\vec{\sigma}}[\langle \sigma_t, \surd \rangle]_t \rangle}\ \textsc{Thread}_{\text{T}}$$

$$\frac{T_{\vec{\sigma}}|_t = \langle \sigma_t, b_t \rangle \qquad P \vdash \langle \sigma, \sigma_t, b_t \rangle \overset{(\sigma,\sigma_t)}{\rightarrow} s \qquad s \in \{\mathcal{E}, \bot\}}{P \vdash \langle \sigma, T_{\vec{\sigma}} \rangle \overset{(\sigma,\vec{\sigma})}{\rightarrow} s}\ \textsc{Thread}_{\text{E,I}}$$

$$\frac{\forall\, 1 \leq t \leq TS : T_{\vec{\sigma}}|_t = \langle \sigma_t, \surd \rangle}{P \vdash \langle \sigma, T_{\vec{\sigma}} \rangle \overset{(\sigma,\vec{\sigma})}{\rightarrow} \surd}\ \textsc{Termination}$$

(c) Interleaving rules

$$\frac{T_{\vec{\sigma}}|_t = \langle (\beta_t, \sigma_t), \mathbf{barrier}\, e_t\,;\, b_t \rangle \wedge \neg(\sigma, \sigma_t)(e_t)}{P \vdash \langle \sigma, T_{\vec{\sigma}} \rangle \rightarrow \langle \sigma, T_{\vec{\sigma}}[\langle (\beta_t, \sigma_t), b_t \rangle]_t \rangle}\ \textsc{Barrier}_{\text{Skip}}$$

$$\frac{\forall\, t : T_{\vec{\sigma}}|_t = \langle (\beta_t, \sigma_t), \mathbf{barrier}\, e_t\,;\, b_t \rangle \wedge (\sigma, \sigma_t)(e_t) \qquad \forall\, t_1, t_2 : \beta_{t_1} = \beta_{t_2}}{P \vdash \langle \sigma, T_{\vec{\sigma}} \rangle \rightarrow \langle \sigma, \langle (\beta_1, \sigma_1), b_1 \rangle, \ldots, \langle (\beta_{TS}, \sigma_{TS}), b_{TS} \rangle \rangle}\ \textsc{Barrier}_{\text{S}}$$

$$\frac{\forall\, t : T_{\vec{\sigma}}|_t = \langle (\beta_t, \sigma_t), \mathbf{barrier}\, e_t\,;\, b_t \rangle \wedge (\sigma, \sigma_t)(e_t) \qquad \exists\, t_1, t_2 : \beta_{t_1} \neq \beta_{t_2}}{P \vdash \langle \sigma, T_{\vec{\sigma}} \rangle \overset{(\sigma,\vec{\sigma})}{\rightarrow} \mathcal{E}}\ \textsc{Barrier}_{\text{F}}$$

(d) Synchronisation rules; barrier variables β_{t_1} and β_{t_2} enforce OpenCL conditions **B1** and **B2**

Fig. 4. Interleaving operational semantics

Interleaving. Fig. 4c, we give our interleaving semantics for a kernel P given thread count TS. The semantics is defined over tuples $\langle\sigma, \langle\sigma_1, b_1\rangle, \ldots, \langle\sigma_{TS}, b_{TS}\rangle\rangle$, where σ is the *shared store*, σ_t is the *private store* of thread t, and b_t is the statement or sequence of statements thread t will reduce next. A thread *cannot* access the private store of any other thread, while the shared store is accessible by *all* threads. In the figure, $T_{\vec{\sigma}}$ denotes $(\langle\sigma_1, b_1\rangle, \ldots, \langle\sigma_{TS}, b_{TS}\rangle)$, where $\vec{\sigma} = (\sigma_1, \ldots, \sigma_{TS})$. Moreover, $T_{\vec{\sigma}}|_t$ denotes $\langle\sigma_t, b_t\rangle$ and $T_{\vec{\sigma}}[\langle\sigma', b\rangle]_t$ denotes $T_{\vec{\sigma}}$ with the t-th element replaced by $\langle\sigma', b\rangle$.

The THREAD$_{\text{B}}$ rule defines how a single step is performed by a single thread, cf. the rules in Fig. 4b. The THREAD$_{\text{T}}$ rule defines termination of a single thread, where the thread enters the termination state $\surd$ from which no further reduction is possible. The THREAD$_{\text{E,I}}$ rule specifies that a kernel terminates with status *error* or *infeasible* if one of the threads terminates as such. The TERMINATION rule specifies that a kernel terminates once all threads have terminated. As steps might be possible in multiple threads, the THREAD rules are non-deterministic and, hence, define an interleaving semantics.

We define a *reduction* of a kernel P as sequence of applications of the operational rules where each thread starts reduction from $Start$ and where the *initial* shared store is some σ and the *initial* private store of thread t is some σ_t. A reduction is *maximal* if it is either infinite or if termination with status *termination*, *error*, or *infeasible* has occurred.

Our interleaving semantics effectively has a sequentially consistent memory model, which is not the case for GPUs in practice. However, because our viewpoint is that GPU kernels that exhibit data races should be regarded as erroneous, this is of no consequence.

Barrier Synchronisation. When we define lock-step predicated execution of barriers in Sect. 4 we will need to model execution of a barrier by a thread in a *disabled* state. In preparation for this, let us say that a barrier statement has the form $\mathbf{barrier}\, e$, where e is a Boolean expression. In Sect. 4, e will evaluate to $true$ if and only if the barrier is executed in an enabled state. The notion of thread-enabledness is not relevant to our interleaving semantics: we can view a thread as *always* being enabled. Thus we regard the $\mathbf{barrier}$ syntax of our kernel programming language as short for $\mathbf{barrier}\ true$.

Figure 4d defines the rules for (mis-)synchronisation between threads at barriers. Our aim here is to formalise the conditions for correct barrier synchronisation in OpenCL, which are stated informally in the OpenCL specification as follows [13]:

B1. If $\mathbf{barrier}$ is inside a conditional statement, then all [threads] must enter the conditional if any [thread] enters the conditional statement and executes the barrier.
B2. If $\mathbf{barrier}$ is inside a loop, all [threads] must execute the barrier for each iteration of the loop before any are allowed to continue execution beyond the barrier.

The rules of Fig. 4d capture these conditions using a number of special *barrier variables* that we assume are implicit in definition of each kernel:

- Every thread has a private variable v_{barrier}. We assume that each barrier appearing in the kernel has a unique id. The variable v_{barrier} of each thread t is initialised to a special value $(-)$ different from every barrier id. When t reaches a barrier, v_{barrier} is set to the id of that barrier, and it is reset to $(-)$ after reduction of the barrier.
- For every loop L in the kernel, every thread has a private *loop counter* variable v_L. The variable v_L of each thread t is initialised to zero, incremented each time the header node for L is reduced by t, and reset to zero on exit from L.

The variable v_{barrier} codifies that each thread is synchronising on the same barrier, capturing condition **B1** above. The loop counters codify that each thread must have executed the same number of loop iterations upon synchronisation, capturing **B2**.

In Fig. 4d, we express the private store of a thread t as a pair (β_t, σ_t), where β_t records the barrier variables for the thread and σ_t the values of all other private variables. The BARRIER$_{\mathrm{SKIP}}$ rule specifies that **barrier** e is a no-op if e is false. Although this can never occur for kernels written directly in our kernel programming language, our equivalence proof in Sect. 5 requires this detail to be accounted for.

The BARRIER$_{\mathrm{S}}$ rule specifies that reduction continues beyond a barrier if all threads are at a barrier and the barrier variables agree across threads. The BARRIER$_{\mathrm{F}}$ rule specifies that a kernel should terminate with *error* if the threads have reached barriers with disagreeing barrier variables: this means that one of **B1** or **B2** has been violated and thus *barrier divergence* has occurred.

Data Races. We say that a thread t is *responsible* for a step in a reduction if a THREAD rule (see Fig. 4c) was employed in the step and the premise of the rule was instantiated with t. Moreover, we say that a thread t *accesses* a variable v in a step if t is responsible for the step and if in the step either (a) the value of v is used to evaluate an expression or (b) v is updated. The definition is now as follows:

Definition 3.1. *Let P be a kernel. Then, P* has data race *if there is a maximal reduction ρ of P, distinct threads t and t', and a shared variable v such that: ρ does not end in the* infeasible *status $\perp$; t updates v during ρ; t' accesses v during ρ; no application of* BARRIER$_{\mathrm{S}}$ *occurs between the accesses (i.e., no barrier separates them).*

Terminating and Race Free Kernels. We say that a kernel P is *(successfully) terminating* with respect to the interleaving semantics if all maximal reductions of P are finite and do not end with status *error*. We say that P is *race free* with respect to the interleaving semantics if P has no data races according to Definition 3.1.

4 Lock-Step Semantics for GPU Kernels

We define lock-step execution semantics for GPU kernels represented as arbitrary CFGs in two stages. First, in Sect. 4.1, we present a transformation which turns the program executed by a single thread into a form where control flow is *flattened*: all branches, except for loop back edges, are eliminated. Then, in Sect. 4.2, we use the transformation to express lock-step execution of all threads in a kernel as a sequential *vector* program.

To avoid many corner cases we assume that kernels always synchronise on a barrier immediately preceding termination. This is without loss of generality, as threads implicitly synchronise on kernel termination. In addition, if a block B ends with **goto** $B_1, \ldots, B_n$ then at most one of $B_1, \ldots, B_n$ is a loop head. A kernel can be trivially preprocessed to satisfy these restrictions.

Sort Order. Predication of CFGs involves flattening control flow, rewriting branches by predicating blocks and executing these blocks in a linear order. Intuitively, for a kernel exhibiting control flow corresponding to an **if-then-else** statement s, this linear order must arrange blocks such that statements preceding s occur before the statements inside s, which in turn must precede the statements occurring after s. However, if statements s_1 and s_2 occur, resp., in the **then** and **else** branches of s, then the order in which the blocks associated with s_1 and s_2 appear does not matter.

For arbitrary CFGs without loops any topological sort gives a suitable order: it ensures that if block B is a predecessor of C in the original CFG then B will be executed before C in the predicated program. In the presence of loops the order must ensure that once execution of the blocks in a loop commences this loop will be executed completely before any node outside the loop is executed.

Formally, we require a total order $\leq$ on blocks satisfying the following conditions:

- For all blocks B and C, if there is a path from B to C in the CFG, then $B \leq C$ unless a back edge occurs on the path.
- For all loops L, if $B \leq D$ and $B, D \in L$, then $C \in L$ for all $B \leq C \leq D$.

A total order satisfying the above conditions always can always be computed: Consider any innermost loop of the kernel and perform a topological sort of the blocks in the loop body (disregarding back edges). Replace the loop body by an abstract block. Repeat until no loops remain and perform a topological sort of resulting CFG. The sort order is now the order obtained by the final topological sort where one recursively replaces each abstract node by the nodes it represents, i.e., if $B \leq L \leq D$ with L an abstract node, then for any $C \leq C'$ in the loop body represented by L one defines $B \leq C \leq C' \leq D$.

Considering the kernel of Fig. 3, we have that $L = \{W, I_1, I_1', B_1, I_2, I_2', B_2, W_{last}\}$ is a loop and that $\mathit{Start} \leq W \leq I_1 \leq I_1' \leq B_1 \leq I_2 \leq I_2' \leq B_2 \leq W_{last} \leq W_{end}$ satisfies our requirements; reversing I_1 and I_1', and also I_2 and I_2', is possible.

In what follows we assume that a total order satisfying the above conditions has been chosen, and we refer to this order as the *sort order*. For a block B we use $\mathit{next}(B)$ to denote the block that follows B in the sort order. If B is the final block in the sort order we define $\mathit{next}(B)$ to be End, the block label denoting thread termination.

4.1 Predication of a Single Thread

We now describe how predication of the body of a kernel thread is performed.

Predication of Statements. To predicate statements, we introduce a fresh private variable v_{active} for each thread, to which we assign *BlockId*s; the assigned *BlockId* indicates the block that needs to be executed. If the value of v_{active} is not equal to the block that is currently being executed, all statements in the block will effectively be no-ops. In the case of **barrier** this follows by the BARRIER$_{\text{SKIP}}$ rule of Fig. 4d.

Assuming the *BlockId* of the current block is B, predication of statements is defined in Table 1, except for **assume** statements which are dealt with below at the level of blocks. In the case of **havoc**, the variable v_{havoc} is fresh and private.

Table 1. Predication of statements

Original form	Predicated form
$v := e\,;$	$v := (v_{\text{active}} = B)\ ?\ e : v\,;$
havoc $v\,;$	**havoc** $v_{\text{havoc}}\,;$ $v := (v_{\text{active}} = B)\ ?\ v_{\text{havoc}} : v\,;$
assert $e\,;$	**assert** $(v_{\text{active}} = B) \Rightarrow e\,;$
skip;	**skip**;
barrier;	**barrier** $(v_{\text{active}} = B)\,;$

Table 2. Predication of blocks

Original form	Predicated form
B : **assume** $\mathit{guard}(B)$; ss **goto** $B_1, \ldots, B_n$; (B *is not* the last node of a loop according to the sort order)	B : $\pi(ss)$ $v_{\text{next}} :\in \{B_1, \ldots, B_n\}$; **assume** $(v_{\text{active}} = B)$ $\Rightarrow \bigwedge_{i=1}^{n} ((v_{\text{next}} = B_i) \Rightarrow \mathit{guard}(B_i))$; $v_{\text{active}} := (v_{\text{active}} = B) \; ? \; v_{\text{next}} : v_{\text{active}}$; **goto** $\mathit{next}(B)$;
B : **assume** $\mathit{guard}(B)$; ss **goto** $B_1, \ldots, B_n$; (B *is* the last node of a loop according to the sort order)	B : $\pi(ss)$ $v_{\text{next}} :\in \{B_1, \ldots, B_n\}$; **assume** $(v_{\text{active}} = B)$ $\Rightarrow \bigwedge_{i=1}^{n} ((v_{\text{next}} = B_i) \Rightarrow \mathit{guard}(B_i))$; $v_{\text{active}} := (v_{\text{active}} = B) \; ? \; v_{\text{next}} : v_{\text{active}}$; **goto** $B_{\text{back}}, B_{\text{exit}}$; B_{back} : **assume** $v_{\text{active}} = B_{\text{head}}$; **goto** B_{head} ; B_{exit} : **assume** $v_{\text{active}} \neq B_{\text{head}}$; **goto** $\mathit{next}(B)$;

Predication of Blocks. Let $\pi(s)$ denote the predicated form of a single statement s, and $\pi(ss)$ the pointwise extension to a sequence of statements ss. Predication of blocks is defined by default as in the top row of Table 2 (see also Fig. 5). Here, v_{next} is a fresh, private variable, and $v_{\text{next}} :\in \{B_1, \ldots, B_n\}$ is shorthand for **havoc** v_{next} ; **assume** $\bigvee_{i=1}^{n}(v_{\text{next}} = B_i)$. Furthermore, $\mathit{guard}(B)$ denotes the expression that occurs in the **assume** that is required to occur at the beginning of block B.

At the end of the predicated block, v_{active} is set to the value of the block to be reduced next, while actual reduction continues with block $\mathit{next}(B)$, as specified by the sort order. The **assume** that 'guards' the block to be reduced next is moved into the block currently being reduced. Moving guards does not affect behaviour, but only shortens traces that end in *infeasible*; this is needed to properly handle barrier divergence in lock-step kernels.

The above method does not deal correctly with loops: no block can be executed more than once as no back-edges are occur. As such, we predicate block a B in a special manner if B belongs to a loop L and B occurs last in the sort order among all the blocks of L. Assume B_{head} is the header of L. The block B is predicated as in the bottom row of Table 2, where B_{back} and B_{exit} are fresh (see again Fig. 5). Our definition of the sort order guarantees that B_{head} is always sorted first among the blocks of L. By the introduction of B_{back}, reduction jumps back to B_{head} if L needs to be reduced again, otherwise reduction will continue beyond L by definition of B_{exit}.

Predication of Kernels. Predicating a complete kernel P now consists of three steps: (1) Compute a sort order on blocks as detailed above; (2) Predicate every block with respect to the sort order, according to the rules of Table 2; (3) Insert the assignment $v_{\text{active}} := \mathit{Start}$ at the beginning of $\pi(\mathit{Start})$. The introduction of $v_{\text{active}} := \mathit{Start}$ ensures that the statements from $\pi(\mathit{Start})$ are always reduced first.

```
$B_2$        : $\textbf{barrier}\,(v_{\text{active}} = B_2)$ ;
               $v_{\text{next}} :\in \{W_{last}\}$ ;
               $v_{\text{active}} := (v_{\text{active}} = B_2)\ ?\ v_{\text{next}} : v_{\text{active}}$ ;
               $\textbf{goto}\, W_{last}$ ;
$W_{last}$   : $\mathit{offset} :=$
                   $(v_{\text{active}} = W_{last})\ ?\ (2 \cdot \mathit{offset}) : \mathit{offset}$ ;
               $v_{\text{next}} :\in \{W, W_{end}\}$ ;
               $v_{\text{active}} := (v_{\text{active}} = W_{last})\ ?\ v_{\text{next}} : v_{\text{active}}$ ;
               $\textbf{assume}\,(v_{\text{active}} = W_{last}) \Rightarrow (((v_{\text{next}} = W) \Rightarrow (\mathit{offset} < \mathit{TS}))$
                                                  $\wedge((v_{\text{next}} = W_{end}) \Rightarrow (\mathit{offset} \geq \mathit{TS})))$
               $\textbf{goto}\, W_{\text{back}}, W_{\text{exit}}$ ;
$W_{\text{back}}$ : $\textbf{assume}\, v_{\text{active}} = W$ ;
               $\textbf{goto}\, W$ ;
$W_{\text{exit}}$ : $\textbf{assume}\, v_{\text{active}} \neq W$ ;
               $\textbf{goto}\, W_{end}$ ;
$W_{end}$    : $\textbf{goto}\, \mathit{End}$ ;
```

Fig. 5. Predication of part of the kernel of Fig. 3

4.2 Lock-Step Execution of All Threads

We now use the predication scheme of Sect. 4.1 to define a lock-step execution semantics for kernels. We achieve this by encoding the kernel as a sequential program, each statement of which is a *vector* statement that performs the work of all threads simultaneously. To enable this, we first extend our programming language with these vector statements.

Vector Statements. We extend our language as follows:

$$\mathit{Stmt} ::= \cdots \mid \mathit{Var}^* := \mathit{Expr}^* \mid \textbf{havoc}\ \mathit{Var}^* \mid \mathit{Var} := \psi((\mathit{Expr} \times \mathit{Expr})^*)$$

The vector assignment simultaneously assigns values to multiple variables, where the variables assigned to are assumed to be distinct and where the number of expressions is equal to the number of variables. Similarly, the vector **havoc** havocs multiple variables, which are are assumed to be distinct. The ψ-assignment is used to model simultaneous writes to a shared variable by all threads. It takes a sequence $(e_1, e'_1), \ldots, (e_n, e'_n)$, with each e_i a Boolean, and non-deterministically assigns to the variable v a value from the set $\{\sigma(e'_i) \mid 1 \leq i \leq n \wedge \sigma(e_i)\}$ (if the set is empty, v is left unchanged).

The semantics for the new statements is presented in Fig. 6, where $\langle e_i \rangle_{i=1}^{n}$ denotes $(e_1), \ldots, (e_n)$ and $[v_i \mapsto \mathit{val}_i]_{i=1}^{n}$ denotes $[v_1 \mapsto \mathit{val}_1] \cdots [v_n \mapsto \mathit{val}_n]$.

Lock-Step Execution. To encode a kernel P as a single-threaded program $\phi(P)$ which effectively executes all threads in lock-step, we assume for every *private* variable v from P that there exists a variable v_t in $\phi(P)$ for each $1 \leq t \leq \mathit{TS}$. For each *shared* variable v from P we assume there exists an identical variable in $\phi(P)$. Construction of a lock-step program for P starts from $\pi(P)$ — the predicated version of P.

Statements. The construction for the predicated statements from Table 1 is presented in Table 3a. In the table, ϕ_t denotes a map over expressions which replaces each private variable v by v_t. Note that for every thread t, there exists a variable $v_{\text{active},t}$, as variables

$$\frac{\forall i : val_i = (\sigma, \sigma_t)(e_i)}{P \vdash \langle (\sigma, \sigma_t), \langle v_i \rangle_{i=1}^{n} := \langle e_i \rangle_{i=1}^{n} \rangle \overset{(\sigma,\sigma_t)}{\rightarrow} (\sigma, \sigma_t)[v_i \mapsto val_i]_{i=1}^{n}} \textsc{Assigns}$$

$$\frac{\forall i : val_i \in D}{P \vdash \langle (\sigma, \sigma_t), \mathbf{havoc}\, \langle v_i \rangle_{i=1}^{n} \rangle \overset{(\sigma,\sigma_t)}{\rightarrow} (\sigma, \sigma_t)[v_i \mapsto val_i]_{i=1}^{n}} \textsc{Havocs}$$

$$\frac{\exists i : \sigma(e_i) \wedge val = (\sigma, \sigma_t)(e'_i)}{P \vdash \langle \sigma, v := \psi(\langle e_i, e'_i \rangle_{i=1}^{n}) \rangle \overset{(\sigma,\sigma_t)}{\rightarrow} \sigma[v \mapsto val]} \psi_{\mathrm{T}}$$

$$\frac{\forall\, i : \neg(\sigma, \sigma_t)(e_i)}{P \vdash \langle (\sigma, \sigma_t), v := \psi(\langle e_i, e'_i \rangle_{i=1}^{n}) \rangle \rightarrow (\sigma, \sigma_t)} \psi_{\mathrm{F}}$$

Fig. 6. Operational semantics for vector statements

freshly introduced by the predication scheme of Sect. 4.1 are private. Hence, we always know for each thread which block to reduce next. We discuss each statement in turn.

With respect to assignments, we distinguish between assignments to private and shared variables. For a private variable v, the assignment is replaced by a vector assignment to the variables v_t, where ϕ_t is applied to e as appropriate. For a shared variable v, it is not obvious which value needs to be assigned to v, as there might be multiple threads t with $v_{\text{active},t} = B$; we non-deterministically pick the value from one of the threads with $v_{\text{active},t} = B$, employing a ψ-assignment.

In the case of a **havoc** followed by an assignment, there is again a case distinction between private and shared variables. For a private variable, the **havoc** and assignment are simply replaced by corresponding vector statements. For a shared variable, a vector havoc is used to produce an arbitrary value for each thread, and then the value associated with one of the threads t with $v_{\text{active},t} = B$ is non-deterministically assigned employing ψ.

In the case of **assert**, we test whether $(v_{\text{active},t} = B) \Rightarrow \phi_t(e)$ holds for each thread $1 \leq t \leq TS$. The **skip** statement remains a no-op.

Lock-step execution of a **barrier** statement with condition $v_{\text{active}} = B$ translates to an assertion checking that if $v_{\text{active},t} = B$ holds for *some* thread t then it must hold for *all* threads. We call these assertions *barrier assertions*. We shall sketch in Sect. 5 that checking for barrier divergence in this manner is equivalent to checking for barrier divergence in the interleaving semantics of Sect. 3. However, contrary to the interleaving case, there is no need to consider barrier variables in the lock-step case.

The last three rows of Table 3a consider statements that do not originate from Table 1 but that do occur in blocks: Initially, each $v_{\text{active},t}$ is assigned to $Start$; assignments to v_{active} are vectorised, where $:\in$ is extended in the obvious way to non-deterministically assign values from multiple sets to multiple variables; **assume** is dealt with as **assert**.

Blocks. The lock-step construction for blocks is presented in Table 3b, where $\phi(ss)$ denotes the lock-step form of a sequence of statements.

If a block *is not* sorted last among the blocks of a loop (see the top row of Table 3b), we simply apply the lock-step construction to the statements in the block. If a block *is* sorted last among blocks in a loops L (see the bottom row of Table 3b) then the

Table 3. Lock-step construction

(a) Statements

Predicated form	Lock-step form	
$v := (v_{\text{active}} = B)\ ?\ e : v\,;$	v private	$\langle v_t\rangle_{t=1}^{TS} := \langle (v_{\text{active},t} = B)\ ?\ \phi_t(e) : v_t\rangle_{t=1}^{TS}\,;$
	v shared	$v := \psi(\langle v_{\text{active},t} = B, \phi_t(e)\rangle_{t=1}^{TS})\,;$
$\textbf{havoc}\ v_{\text{havoc}}\,;$ $v := (v_{\text{active}} = B)\ ?\ v_{\text{havoc}} : v\,;$	v private	$\textbf{havoc}\ \langle v_{\text{havoc},t}\rangle_{t=1}^{TS}\,;$ $\langle v_t\rangle_{t=1}^{TS} := \langle (v_{\text{active},t} = B)\ ?\ v_{\text{havoc},t} : v_t\rangle_{t=1}^{TS}\,;$
	v shared	$\textbf{havoc}\ \langle v_{\text{havoc},t}\rangle_{t=1}^{TS}\,;$ $v := \psi(\langle v_{\text{active},t} = B, v_{\text{havoc},t}\rangle_{t=1}^{TS})\,;$
$\textbf{assert}\ (v_{\text{active}} = B) \Rightarrow e\,;$	$\textbf{assert}\ \bigwedge_{t=1}^{TS}((v_{\text{active},t} = B) \Rightarrow \phi_t(e))$	
$\textbf{skip}\,;$	$\textbf{skip}\,;$	
$\textbf{barrier}\ (v_{\text{active}} = B)\,;$	$\textbf{assert}\ \Big(\bigvee_{t=1}^{TS}(v_{\text{active},t} = B)\Big) \Rightarrow \Big(\bigwedge_{t=1}^{TS}(v_{\text{active},t} = B)\Big);$	
$v_{\text{active}} := \mathit{Start}\,;$	$\langle v_{\text{active},t}\rangle_{t=1}^{TS} := \langle \mathit{Start}\rangle_{t=1}^{TS}\,;$	
$v_{\text{next}} :\in \{B_1, \ldots, B_n\}\,;$	$\langle v_{\text{next},t}\rangle_{t=1}^{TS} :\in \langle \{B_1, \ldots, B_n\}\rangle_{t=1}^{TS}\,;$	
$\textbf{assume}\ (v_{\text{active}} = B) \Rightarrow e\,;$	$\textbf{assume}\ \bigwedge_{t=1}^{TS}((v_{\text{active},t} = B) \Rightarrow \phi_t(e))$	

(b) Blocks

Predicated form	Lock-step form
$B\ :\ ss$ $\quad \textbf{goto}\ \mathit{next}(B)\,;$	$B\ :\ \phi(ss)$ $\quad \textbf{goto}\ \mathit{next}(B)\,;$
$B\ :\ ss$ $\quad \textbf{goto}\ B_{\text{back}}, B_{\text{exit}}\,;$ $B_{\text{back}}\ :\ \textbf{assume}\ v_{\text{active}} = B_{\text{head}}\,;$ $\quad \textbf{goto}\ B_{\text{head}}\,;$ $B_{\text{exit}}\ :\ \textbf{assume}\ v_{\text{active}} \neq B_{\text{head}}\,;$ $\quad \textbf{goto}\ \mathit{next}(B)\,;$	$B\ :\ \phi(ss)$ $\quad \textbf{goto}\ B_{\text{back}}, B_{\text{exit}}\,;$ $B_{\text{back}}\ :\ \textbf{assume}\ \bigvee_{t=1}^{TS}(v_{\text{active},t} = B_{\text{head}})\,;$ $\quad \textbf{goto}\ B_{\text{head}}\,;$ $B_{\text{exit}}\ :\ \textbf{assume}\ \bigwedge_{t=1}^{TS}(v_{\text{active},t} \neq B_{\text{head}})\,;$ $\quad \textbf{goto}\ \mathit{next}(B)\,;$

successors of the block in the predicated program are B_{back}, which leads to the loop header, and B_{exit}, which leads to a node outside the loop. Our goal is to enforce the rule that *no* thread should leave the loop until *all* threads are ready to leave the loop, as discussed informally in Sect. 2 and illustrated for structured programs by the guard of the **while** loop in Fig. 2. To achieve this, the bottom row of Table 3b employs an **assume** in B_{back} requiring that $v_{\text{active}} = B_{\text{head}}$ for *some* thread, and an **assume** in B_{exit} requiring $v_{\text{active}} \neq B_{\text{head}}$ for *all* threads. A concrete example is given in Fig. 7.

Lock-Step Semantics and Data Races. Having completed our definition of the lock-step construction $\phi(P)$ for a kernel P, we now say that the lock-step semantics for P is the *interleaving* semantics for $\phi(P)$, with respect to a *single* thread (i.e., with $TS = 1$). Barrier divergence is captured via the introduction of *barrier assertions*. This leaves to define data races in lock-step execution traces.

Say that thread t is *enabled* during a reduction step if the statement being reduced occurs in block B and $v_{\text{active},t} = B$ holds at the point of reduction and let v be a variable. A thread t *reads* v during a reduction step if t is enabled during the step and if the step involves evaluating an expression containing v. A thread t *writes* v during a

```
$B_2$        : $\mathbf{assert}\left(\bigvee_{t=1}^{TS}(v_{\mathrm{active},t} = B_2)\right) \Rightarrow \left(\bigwedge_{t=1}^{TS}(v_{\mathrm{active},t} = B_2)\right)$ ;
               $\langle v_{\mathrm{next},t}\rangle_{t=1}^{TS} :\in \langle\{W_{last}\}\rangle_{t=1}^{TS}$ ;
               $\langle v_{\mathrm{active},t}\rangle_{t=1}^{TS} := \langle(v_{\mathrm{active},t} = B_2)\ ?\ v_{\mathrm{next},t} : v_{\mathrm{active},t}\rangle_{t=1}^{TS}$ ;
               $\mathbf{goto}\, W_{last}$ ;
$W_{last}$   : $\langle \mathit{offset}_t\rangle_{t=1}^{TS} := \langle(v_{\mathrm{active},t} = W_{last})\ ?\ (2 \cdot \mathit{offset}_t) : \mathit{offset}_t\rangle_{t=1}^{TS}$ ;
               $\langle v_{\mathrm{next},t}\rangle_{t=1}^{TS} :\in \langle\{W, W_{end}\}\rangle_{t=1}^{TS}$ ;
               $\mathbf{assume} \bigwedge_{t=1}^{TS}((v_{\mathrm{active},t} = W_{last}) \Rightarrow (((v_{\mathrm{next},t} = W) \Rightarrow (\mathit{offset}_t < TS))$
                                                   $\wedge((v_{\mathrm{next},t} = W_{end}) \Rightarrow (\mathit{offset}_t \geq TS))))$ ;
               $\langle v_{\mathrm{active},t}\rangle_{t=1}^{TS} := \langle(v_{\mathrm{active},t} = W_{last})\ ?\ v_{\mathrm{next},t} : v_{\mathrm{active},t}\rangle_{t=1}^{TS}$ ;
               $\mathbf{goto}\, W_{\mathrm{back}}, W_{\mathrm{exit}}$ ;
$W_{\mathrm{back}}$ : $\mathbf{assume} \bigvee_{t=1}^{TS}(v_{\mathrm{active},t} = W)$ ;
               $\mathbf{goto}\, W$ ;
$W_{\mathrm{exit}}$ : $\mathbf{assume} \bigwedge_{t=1}^{TS}(v_{\mathrm{active},t} \neq W)$ ;
               $\mathbf{goto}\, W_{end}$ ;
$W_{\mathrm{end}}$  : $\mathbf{goto}\, \mathit{End}$ ;
```

Fig. 7. Part of the lock-step program for the kernel of Fig. 3

reduction step if t is enabled during the step and if the statement being reduced is an assignment to v. In the case of a write, if multiple threads are enabled then v will be updated non-deterministically using one of the values supplied by the enabled threads. Nevertheless, we regard *all* enabled threads as having written to v.

A data race in a lock-step program is defined as follows:

Definition 4.1. *Let $\phi(P)$ be the lock-step form of a kernel P. Then, $\phi(P)$* has a data race *if there is a maximal reduction ρ of $\phi(P)$, distinct threads t and t', and a shared variable v such that: ρ does not end in* infeasible*; t writes v during ρ and t' either reads or writes v during ρ; the accesses are not separated by a* barrier assertion *(i.e., no barrier is reduced between the accesses).*

Terminating and Race Free Kernels. We say that a kernel P is *terminating* with respect to the lock-step semantics if all maximal reductions of $\phi(P)$ are finite and do not end with status *error*. We say that P is *race free* with respect to the lock-step semantics if $\phi(P)$ has no data races according to Definition 4.1.

5 Equivalence between Interleaving and Lock-Step Semantics

We can now prove our main result, an equivalence between the interleaving semantics of Sect. 3 and lock-step semantics of Sect. 4. Our result applies to *well-formed* kernels:

Definition 5.1. *A kernel P is* well-formed *if for every block B in P if B ends with* $\mathbf{goto}\, B_1, \ldots, B_n$*, then $\bigvee_{i=1}^{n} guard(B_i)$ is a tautology.*

Well-formedness implies that whenever a thread reduces a **goto**, the guard of *at least one* block that can be reached via the **goto** is guaranteed to hold. Recall from Sect. 3.1 that guards of assume statements refer only to private variables, thus it is not possible for another thread to invalidate the guard of an **assume** between reduction of a **goto** and evaluation of the guard. Well-formedness is guaranteed to hold if the CFG for P is obtained from a kernel written in a C-like language such as OpenCL or CUDA.

Theorem 5.2. *Let P be a well-formed kernel and let $\phi(P)$ be the lock-step version of P. Then, P is race free and terminating with respect to the interleaving semantics iff P is race free and terminating with respect to the lock-step semantics. Moreover, if race-freedom holds then for every terminating reduction of P there exists a terminating reduction of $\phi(P)$, and vice versa, such that every shared variable v has the same value at the end of both reductions.*

To see why well-formedness is required, consider the following kernel, where each thread t has a private variable tid whose value is t and where v is shared and v' is private:

$$\begin{array}{llllll}
\mathit{Start} : & \mathbf{assume}\,\mathit{true} & B_1 : & \mathbf{assume}\,\mathit{tid} = 1 \wedge v' = 5\,; & B_2 : & \mathbf{assume}\,\mathit{tid} \neq 1\,; \\
 & v := 4\,; v' := v\,; & & \mathbf{goto}\,\mathit{End}\,; & & v := 5\,; \\
 & \mathbf{goto}\,B_1, B_2\,; & & & & \mathbf{goto}\,\mathit{End}\,;
\end{array}$$

The interleaving semantics allows for reduction of $\mathbf{assume}\,\mathit{tid} = 1 \wedge v' = 5$ after all assignments in all threads have taken place. Hence, if the assignment of 4 to v by thread 1 is not last among the assignments to v, then $v' = 5$ evaluates to true, and eventually termination occurs with a data race. In the case of lock-step execution and assuming the sort order $\mathit{Start} \leq B_1 \leq B_2$, we have that $\mathbf{assume}\,\mathit{tid} = 1 \wedge v' = 5$ is always reduced immediately after $v := 4\,; v' := v\,;$. Hence, reduction always terminates with *infeasible* and no data race occurs.

That termination is required follows by adapting the counterexamples from [12,11] showing that CUDA hardware does not necessarily schedule threads from a non-terminating kernel in a way that that is fair from an interleaving point-of-view.

The proof of the theorem proceeds by showing that P and its predicated form $\pi(P)$ are stutter equivalent, and then establishing a relationship between $\pi(P)$ and $\phi(P)$.

Equivalence of P and $\pi(P)$. To show that P and $\pi(P)$ are stutter equivalent [14], we define a denotational semantics of kernels in terms of *execution traces* [5], i.e., sequences of tuples $(\sigma, \vec{\sigma}) = (\sigma, \sigma_1, \ldots, \sigma_{TS})$ with σ the shared store and σ_t the private store of thread t.

Definition 5.3. *Let ρ be a maximal reduction. The* denotation *or* execution trace $\mathcal{D}(\rho)$ *of ρ is the sequence of $\twoheadrightarrow$-labels of ρ together with the termination status of ρ if ρ terminates. Let $(b_1, \ldots, b_{TS})$ be a tuple of block labels. The* denotation $\mathcal{D}(b_1, \ldots, b_{TS})$ *of $(b_1, \ldots, b_{TS})$ is the set of denotations of all maximal reductions of $(b_1, \ldots, b_{TS})$ for all initial stores $\sigma, \sigma_1, \ldots, \sigma_t$* not *terminating as* infeasible. *Let P be a kernel. The* denotation $\mathcal{D}(P)$ *of P is $\mathcal{D}(\mathit{Start}, \ldots, \mathit{Start})$.*

Observe that *infeasible* traces are *not* included in the denotations of $(b_1, \ldots, b_{TS})$ and P; these traces do not constitute actual program behaviour.

Stutter equivalence is defined on subsets of variables, where a *restriction* of a store σ to a set of variables V is denoted by $\sigma{\restriction_V}$ and, where given a tuple $(\sigma, \vec{\sigma}) = (\sigma, \sigma_1, \ldots, \sigma_{TS})$, the restriction $(\sigma, \vec{\sigma}){\restriction_V}$ is $(\sigma, \vec{\sigma}){\restriction_V} = (\sigma{\restriction_V}, \sigma_1{\restriction_V}, \ldots, \sigma_{TS}{\restriction_V})$.

Definition 5.4. *Let V be a set of variables. Define the map δ_V over execution traces as the map that replaces every maximal subsequence $(\sigma_1, \vec{\sigma}_1)\,(\sigma_2, \vec{\sigma}_2) \cdots (\sigma_n, \vec{\sigma}_n) \cdots$ where $(\sigma_1, \vec{\sigma}_1){\restriction_V} = (\sigma_2, \vec{\sigma}_2){\restriction_V} = \ldots = (\sigma_n, \vec{\sigma}_n){\restriction_V} = \ldots$ by $(\sigma_1, \vec{\sigma}_1)$.*

Let Σ and T be execution traces. The traces are stutter equivalent *with respect to V, denoted $\Sigma \sim_{\mathrm{st}}^{V} T$, iff:*

- *Σ and T are both finite with equal termination statuses and $\delta_V(\Sigma) = \delta_V(T)$;*
- *Σ and T are both infinite and $\delta_V(\Sigma) = \delta_V(T)$.*

Let P and Q be kernels. The kernels are stutter equivalent *with respect to V, denoted $P \sim_{\mathrm{st}}^{V} Q$, iff for every $\Sigma \in \mathcal{D}(P)$ there is a $T \in \mathcal{D}(Q)$ with $\Sigma \sim_{\mathrm{st}}^{V} T$, and vice versa.*

Theorem 5.5. *If P is a kernel with variables V, then $\pi(P) \sim_{\mathrm{st}}^{V} P$, where $\pi(P)$ is the predicated form of P. A data race occurs in P iff a data race occurs in $\pi(P)$ where, during reduction of neither of the two statements causing the data race, $v_{\mathrm{active}} \neq B$ with B is the block containing the statement.*

The above result follows immediately by a case distinction on the statements that may occur in kernels once we establish the following lemma, which is a direct consequence of our construction and the first requirement on the sort order of blocks.

Lemma 5.6. *Let P be a kernel with variables V. For any thread t and each block B of P, if (σ, σ_t) is a store of t and $(\hat{\sigma}, \hat{\sigma}_t)$ is a store of in t in $\pi(P)$ such that $\hat{\sigma}\restriction_V = \sigma$ and $\hat{\sigma}(v_{\mathrm{active}}) = B$, then*

1. *if the reduction of B is immediately followed by the reduction of a block C, then there exists a reduction of $\pi(B)$ such that v_{active} is equal to C at the end of $\pi(B)$ and eventually $\pi(C)$ is reduced with v_{active} equal to C;*
2. *if the reduction of $\pi(B)$ ends with v_{active} equal to C, then there exists a reduction of B that is immediately followed by the reduction of a block C.*

Soundness and Completeness. Theorem 5.2 is now proved as follows.

Proof (Sketch). For termination and race-freedom of $\phi(P)$, it suffices by Theorem 5.5 to consider $\pi(P)$ — the predicated form of P. Reason by contradiction and construct for a reduction of $\phi(P)$ which is either infinite or has data race, a reduction of $\pi(P)$ that also is either infinite or has a data race: Replace each statement and **goto** from the right-hand columns of Table 3 by a copy of the statement or **goto** in the left-hand column and reduce, where we introduce a copy for each thread. That a reduction of a barrier assertion can be replaced by BARRIER$_{\mathrm{S}}$ follows as no statements from outside loops can be reduced while we are inside a loop (cf. the second requirement on sort order of blocks) and by the guards of blocks having been moved during predication to the end of the block preceding it in execution. The remainder of the theorem follows by permuting steps of different threads so the reverse transformation from above can be applied. □

6 Implementation and Experiments

Implementation in GPUVerify. We have implemented the predication technique described here in GPUVerify [7], a verification tool for OpenCL and CUDA kernels built on top of the Boogie verification engine [6] and Z3 SMT solver [19]. GPUVerify previously employed a predication technique for structured programs. Predication for CFGs has allowed us to build a new front-end for GPUVerify which takes LLVM intermediate representation (IR) as input; IR directly corresponds to a CFG. This allows us to compile OpenCL and CUDA kernels using the Clang/LLVM framework and perform analysis on the resulting IR. Hence, tricky syntactic features of C-like languages are taken care of by Clang/LLVM. Analysing kernels after compilation and optimisation also increases the probity of verification, opening up the opportunity to discover compiler-related bugs.

Experimental Evaluation. To assess the performance overhead in terms of verification time for our novel predication scheme and associated tool chain we compared our new implementation (GPUVerify II) with the original structured one (GPUVerify I).

We compared the tool versions using 163 OpenCL and CUDA kernels drawn from the AMD Accelerated Parallel Processing SDK v2.6 [4] (71 OpenCL kernels), the NVIDIA GPU Computing SDK v2.0 [20] (20 CUDA kernels), Microsoft C++ AMP Sample Projects [18] (20 kernels translated from C++ AMP to CUDA) and Rightware's Basemark CL v1.1 suite [22] (52 OpenCL kernels, provided to us under an academic license). These kernels were used for analysis of GPUVerify I in [7], where several of the kernels had to be manually modified before they could be subjected to analysis: 4 kernels exhibited unstructured control flow due to **switch** statements, and one featured a **do-while** loop which was beyond the scope of the predication scheme of [7]. Furthermore, unstructured control flow arising from short-circuit evaluation of logical operators had been overlooked in GPUVerify I, which affected 30 kernels. In GPUVerify II all kernels are handled uniformly as a consequence of our novel predication scheme in combination with the use of Clang/LLVM, which encodes short-circuit evaluation using unstructured control flow.

All experiments were performed on a PC with a 3.6 GHz Intel i5 CPU, 8 GB RAM running Windows 7 (64-bit), using Z3 v4.1. All times reported are averages over 3 runs. Both tool versions and all our benchmarks, except the commercial Basemark CL kernels, are available online to make our results reproducible.[3]

The majority of our benchmark kernels could be automatically verified by both GPUVerify I and GPUVerify II; 22 kernels were beyond the scope of both tools and resulted in a failed proof attempt. Key to the usability of GPUVerify is its *response time*, the time the tool takes to either report successful verification vs. a failed proof attempt. Comparing GPUVerify I and GPUVerify II we found that across the entire benchmark set the analysis time taken by GPUVerify II was 2.25 times that of GPUVerify I, with GPUVerify II taking on average 2.53 times longer than GPUVerify per kernel. However, the median slow down associated with GPUVerify II was 0.77, i.e., a speed up of 1.3.

The average, median and longest analysis time across all kernels were 4.3, 1.7 and 157 seconds, resp., for GPUVerify I, and 9.6, 1.4 and 300 seconds, resp., for GPUVerify II. For 124 of the 163 kernels (76%), GPUVerify II was marginally (though not significantly) faster than GPUVerify I. For a further 21 kernels (13%) GPUVerify II was up to 50% slower than GPUVerify I. The remaining 18 kernels (11%) caused the slow

[3] `http://multicore.doc.ic.ac.uk/tools/GPUVerify`

down on average. In each case the difference lay in constraint solving times; the SMT queries generated by our CFG-based tool chain can be somewhat more complex than in the structured case. The most dramatic example is a kernel which was verified by GPUVerify I and GPUVerify II in 3 and 202 seconds, resp., a slow-down for GPUVerify II of 70 times. This kernel exhibits a large number of shared memory accesses. In the LLVM IR processed by GPUVerify II these accesses are expressed as many separate, contiguous loads and stores, requiring reasoning about race-freedom between many pairs of operations. The structured approach of GPUVerify I captures these accesses at the abstract syntax tree level, allowing a load/store from/to a contiguous region to be expressed as a single access, significantly simplifying reasoning. This illustrates that there are benefits to working at the higher level of abstract syntax trees, and suggests that optimisations in GPUVerify II to automatically identify and merge contiguous memory accesses might be beneficial.

7 Related Work and Conclusion

Related Work. Interleaving semantics for GPU kernels has been defined by [15,17,12]. These are similar to our semantics except that [15,12] do not give a semantics for barriers. Contrary to our lock-step approach, [15,17] battle the state space explosion due to arbitrary interleavings of threads by considering one particular schedule.

In [11,12], a semantics of CUDA kernels is defined that tries to model NVIDIA hardware as faithfully as possible. The focus is not on predicated execution (although it does figure briefly in [11]), but on so-called *immediate post-dominator re-convergence* [10], a method to continue lock-step execution of threads as soon as possible after branch divergence has occurred between threads.

In addition to the above and similar to us, [12] shows for terminating kernels that CUDA execution of kernels can be faithfully simulated by certain interleaving thread schedules. The reverse is not shown; our analysis is that such a result is difficult to establish due to data races that occur in the examples of [12].

Conclusion. Our lock-step semantics for GPU kernels expressed as arbitrary reducible CFGs enables automated analysis of a wider class of GPU kernels than previous techniques for structured programs, and allows for the analysis of compiled kernel code, after optimisations have been applied. Our soundness and completeness result establishes an equivalence between our lock-step semantics and a traditional semantics based on interleaving, and our implementation in GPUVerify and associated experimental evaluation demonstrate that our approach is practical.

Because our kernel programming language supports non-deterministic choice and havocking of variables it can express an over-approximation of a concrete kernel. In future work we plan to exploit this, investigating the combination of source-level abstraction techniques such as predicate abstraction with our verification method.

The well-formedness restriction of Definition 5.1 means that our equivalence result does not apply to kernels that exhibiting 'dead end' paths. This is relevant if such paths are introduced through under-approximation, e.g., unwinding a loop by a fixed number of iterations in the style of bounded model checking. We plan to investigate whether it is possible to relax these well-formedness conditions under certain circumstances.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Pearson Education, 2nd edn. (2007)
2. Allen, J., Kennedy, K., Porterfield, C., Warren, J.: Conversion of control dependence to data dependence. In: POPL 1983, pp. 177–189 (1983)
3. Alshawabkeh, M., Jang, B., Kaeli, D.: Accelerating the local outlier factor algorithm on a GPU for intrusion detection systems. In: GPGPU-3, pp. 104–110 (2010)
4. AMD: AMD Accelerated Parallel Processing (APP) SDK, http://developer.amd.com/sdks/amdappsdk/pages/default.aspx
5. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: PASTE 2005, pp. 82–87 (2005)
6. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
7. Betts, A., Chong, N., Donaldson, A.F., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: OOPSLA 2012, pp. 113–132 (2012)
8. Collingbourne, P., Cadar, C., Kelly, P.H.J.: Symbolic Testing of OpenCL Code. In: Eder, K., Lourenço, J., Shehory, O. (eds.) HVC 2011. LNCS, vol. 7261, pp. 203–218. Springer, Heidelberg (2012)
9. DeMillo, R.A., Eisenstat, S.C., Lipton, R.J.: Space-time trade-offs in structured programming: An improved combinatorial embedding theorem. J. ACM 27(1), 123–127 (1980)
10. Fung, W.W., Sham, I., Yuan, G., Aamodt, T.M.: Dynamic warp formation and scheduling for efficient GPU control flow. In: MICRO 2007, pp. 407–418 (2007)
11. Habermaier, A.: The model of computation of CUDA and its formal semantics. Tech. Rep. 2011-14, University of Augsburg (2011)
12. Habermaier, A., Knapp, A.: On the Correctness of the SIMT Execution Model of GPUs. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 316–335. Springer, Heidelberg (2012)
13. Khronos Group: The OpenCL specification, version 1.2 (2011)
14. Lamport, L.: What good is temporal logic? In: Information Processing 1983, pp. 657–668 (1983)
15. Leung, A., Gupta, M., Agarwal, Y., Gupta, R., Jhala, R., Lerner, S.: Verifying GPU kernels by test amplification. In: PLDI 2012, pp. 383–394 (2012)
16. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: FSE 2010, pp. 187–196 (2010)
17. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: concolic verification and test generation for GPUs. In: PPoPP 2012, pp. 215–224 (2012)
18. Microsoft Corporation: C++ AMP sample projects for download, http://blogs.msdn.com/b/nativeconcurrency/archive/2012/01/30/c-amp-sample-projects-for-download.aspx
19. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
20. NVIDIA: CUDA Toolkit Release Archive, http://developer.nvidia.com/cuda/cuda-toolkit-archive
21. NVIDIA: NVIDIA CUDA C Programming Guide, Version 4.2 (2012)
22. Rightware Oy: Basemark CL, http://www.rightware.com/en/Benchmarking+Software/Basemark%99+CL
23. Zhu, F., Chen, P., Yang, D., Zhang, W., Chen, H., Zang, B.: A GPU-based high-throughput image retrieval algorithm. In: GPGPU-5, pp. 30–37 (2012)