Barrier Invariants: A Shared State Abstraction for the Analysis of Data-Dependent GPU Kernels *

Nathan Chong Alastair F. Donaldson
Paul H.J. Kelly Jeroen Ketema
Imperial College London
{nyc04,afd,phjk,jketema}@imperial.ac.uk

Shaz Qadeer
Microsoft Research
qadeer@microsoft.com

Abstract

Data-dependent GPU kernels, whose data or control flow are dependent on the input of the program, are difficult to verify because they require reasoning about shared state manipulated by many parallel threads. Existing verification techniques for GPU kernels achieve soundness and scalability by using a two-thread reduction and making the contents of the shared state *nondeterministic* each time threads synchronise at a barrier, to account for all possible thread interactions. This coarse abstraction prohibits verification of data-dependent kernels. We present *barrier invariants*, a novel abstraction technique which allows key properties about the shared state of a kernel to be preserved across barriers during formal reasoning. We have integrated barrier invariants with the GPUVerify tool, and present a detailed case study showing how they can be used to verify three *prefix sum* algorithms, allowing efficient modular verification of a *stream compaction* kernel, a key building block for GPU programming. This analysis goes significantly beyond what is possible using existing verification techniques for GPU kernels.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Verification; GPUs; concurrency; data races.

* This work was supported by the EU FP7 STREP project CARP (project number 287767) and the EPSRC PSL project (EP/I006761/1).

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2509136.2509517

1. Introduction

Graphics processing units (GPUs) are highly parallel processors that are now commonly used in the acceleration of a wide range of computationally intensive tasks. A GPU consists of a large number of processor elements, each equipped with a region of private memory, together with an area of shared memory. GPUs are programmed using *kernels*: a GPU kernel is a function parameterised by a thread identifier, to be executed in parallel by many threads such that each thread executes on a separate processing element. Thread-local data is stored in private memory, and data shared between threads is stored in shared memory. In order to communicate through shared memory, threads can synchronise using *barrier* operations. On reaching a barrier a thread stalls until all threads have reached the barrier and shared memory accesses issued by all threads have completed.[1] The two most widely used programming models for writing GPU kernels are CUDA [30] and OpenCL [22].

Writing GPU kernels is challenging due to concurrency bugs, especially *data races*, where two threads access the same region of shared memory, at least one of the accesses is a write, and there is no intervening barrier synchronisation. As in traditional concurrent software, data races can lead to nondeterministically occurring bugs that are hard to track down and fix. Unlike traditional concurrent software, data races in GPU kernels are rarely benign: they are almost always a result of programmer errors.

Recently, many techniques for formal analysis of GPU kernels have been developed. These techniques have primarily focused on data races, using formal verification [2, 12, 25], dynamic symbolic execution [11, 26, 27] and a combination of static and dynamic analysis [24]. The GPUVerify [2, 12] and PUG [25] tools achieve scalable verification based on the observation that data race-freedom is a *pairwise* property: a data race always occurs between exactly two threads. Data race analysis can thus be performed by con-

[1] In practice, GPU kernels consist of multiple groups of threads. Since barriers, the focus of our paper, allow synchronisation only between threads *within* a group, we restrict our presentation to the single-group case.

sidering the execution of an *arbitrary* pair of threads, using abstraction to over-approximate the effects of other threads. If a kernel is race-free for an arbitrary pair of threads then it must be race-free for all possible pairs of threads. We refer to this approach as the *two-thread reduction*. It avoids the need to use quantifiers to reason about all threads executing a kernel, which is advantageous due to the difficulty of automated reasoning in the presence of quantified formulae [14, 16, 29]. The idea of reducing verification complexity through pairwise reasoning is well-known and has been employed for example in model checking of cache coherence protocols [9, 28, 34].

The soundness and precision of the two-thread reduction hinges on the method used to over-approximate the behaviour of additional threads. The simplest approach is to make *no* assumptions about the behaviour of additional threads, assuming that these threads may update the shared state arbitrarily. In verification terms, this can be achieved by making the contents of the shared state nondeterministic (i.e., by *havocking* the shared state) each time a barrier is reached. Variations on this *adversarial* abstraction are employed by GPUVerify and PUG. The adversarial abstraction has been shown to be effective in the verification of *data-independent* GPU kernels: kernels for which control flow and memory access patterns do not depend on the data stored in shared memory. While a large number of GPU kernels fall into this category, there are important and interesting kernels that exhibit data-dependence. A key family of such kernels use *prefix sum* operations [4, 18] to perform compaction of data [3]; we describe and study these operations and their use extensively in the paper. Data-dependent kernels are difficult to verify because they require reasoning about shared state manipulated by many parallel threads. The access pattern of a single thread may depend on the data or control flow of many other threads, making race checking challenging.

To see why data-dependence hinders verification using adversarial abstraction, consider the following simple kernel, where `A` and `B` are arrays in shared memory, `tid` denotes the id of a thread, and `f` is a side-effect free procedure which may read from the shared state and ensures that for distinct threads s and t that `f(`s`)` $\neq$ `f(`t`)` (a larger, real-world example is presented in Section 2):

```
A[tid] = f(tid); barrier(); B[A[tid]] = tid;
```

The kernel is data-dependent because array `B` is written to at an index which is computed by reading from the shared state. The kernel is clearly race-free. However, consider execution of the kernel fragment with respect to an arbitrary, distinct pair of threads, s and t, using the adversarial abstraction. Because $s \neq t$, execution of `A[tid] = f(tid)` by both threads will not result in a data race on `A`, and will lead to a state in which `A[`s`]` $\neq$ `A[`t`]`. However, at the barrier, the adversarial abstraction dictates that `A` and `B` should be nondeterministically assigned. A possible nondeterministic assignment yields `A[`s`] = A[`t`] = 0`, which causes the statement `B[A[tid]] = tid` to result in a data race on `B` at index 0. This simple kernel is not amenable to verification using the two-thread reduction with adversarial abstraction: the adversarial abstraction is too coarse.

In this paper, we present *barrier invariants*, a novel abstraction technique to allow verification of data-dependent GPU kernels using the two-thread reduction. The idea is that each barrier in a kernel can be annotated with a barrier invariant, stating a property of the shared state that must hold each time the barrier is reached. Barrier invariants retain the scalability of the two-thread reduction, but allow the precision lost by the adversarial abstraction to be recovered: when considering the execution of a barrier by an arbitrary pair of threads, instead of setting the shared state to an arbitrary value, the shared state is set to an arbitrary value *satisfying the barrier invariant*.

A barrier invariant can be added to the above example to capture the fact that the elements of `A` are distinct as follows (where x and y range over thread ids):

```
A[tid] = f(tid);
barrier() invariant(∀x ≠ y : A[x]≠A[y]);
B[A[tid]] = tid;
```

When considering the execution of an arbitrary pair of threads s and t, the invariant is established on entry to the barrier by checking that `A[`s`]` $\neq$ `A[`t`]`: because s and t are arbitrary, this proves that the invariant holds for *all* pairs of threads. After the barrier, it is legitimate to assume that `A[`x`]` $\neq$ `A[`y`]` holds for all pairs of distinct threads x and y and, in particular, for the threads s and t under consideration, and thus the write to `B` at index `A[tid]` is verified to be race-free as required. In fact, after the barrier, it is sufficient to assume the invariant *only* for the pair (s, t). More complex kernels usually require the invariant to be assumed for multiple pairs. However, a small subset of all pairs usually suffices. This is important for kernels with thousands of threads where assuming a barrier invariant for all pairs is not feasible without the use of quantifiers.

In the same way that loop invariants and procedure specifications are fundamental to modular reasoning about sequential programs, we believe that barrier invariants are fundamental to thread-modular reasoning about data-dependent GPU kernels which, until now, has been out-of-scope. Because GPU kernels are executed by large numbers of threads, barrier invariants are still necessary to reason about data-dependent properties even for loop- and call-free kernels.

In summary, the main contributions of our work are:

- A theoretical presentation and proof of soundness for barrier invariants, which allow precise verification of data-dependent GPU kernels;
- A case study using barrier invariants to prove specifications for three distinct prefix sum algorithms, and using these for modular verification of a *stream compaction* kernel, a key building block in GPU programming;

```
// data, out, idx: arrays in shared memory
procedure compact(data, out)
  // (i) test each element with predicate p
  in parallel for each thread tid
    flag[tid] = p(data[tid])
  // (ii) compute indices for compaction
  idx = prescan(flag)
  // (iii) compaction
  in parallel for each thread tid
    if (flag[tid])
      out[idx[tid]] = data[tid]
```

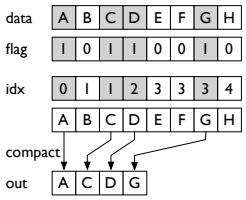


Figure 1. Stream compaction program and example (image courtesy M. Harris [18])

- An implementation of barrier invariants in the GPUVerify tool, making it the first tool capable of verifying data-dependent GPU kernels, a major step forward from the capabilities of existing verifiers [2, 24, 25];
- Experimental results showing that GPUVerify is capable of scalable analysis where existing verification techniques cannot be applied, and where exhaustive symbolic execution is often infeasible [26, 27].

2. Motivating Example: Stream Compaction

Figure 1 presents pseudo-code for a *stream compaction* algorithm: a parallel program that filters an input array with respect to a predicate p. This parallel primitive is commonly used for removing redundant or dead elements from a data set and has many applications in GPU programming, including in parallel breadth first tree traversal, ray tracing and collision detection [3]. For an input array `data`, each thread t tests its respective element `data[`t`]` against p and writes 1 to the temporary array `flag` if the element satisfies p and 0 otherwise. In the *compact* stage, each thread t, for which p`(data[`t`])` holds, must write to an index of output array `out` such that all elements to be kept are written contiguously. This index is the sum of the values in `flag` at indices $0 \leq i < t$.

The index can be computed using an *exclusive prefix sum* operator, also known as a *prescan* [4, 18]. Given an array $[x_1, x_2, \ldots, x_n]$ and an associative operator $\oplus$ with identity e, the prescan operator computes the sums of all prefixes: $[e, x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_{n-1}]$. For example, taking $\oplus$ and e to be $+$ and 0, respectively, the prescan of the array $[3, 1, 7, 0, 4, 1, 6, 3]$ is $[0, 3, 4, 11, 11, 15, 16, 22]$.

A kernel implementing stream compaction is *data-dependent* because the access pattern of a thread is determined by the values of the input array: not just the element that 'belongs' to a thread, but also the `flag` result of all preceding threads. Determining data race-freedom depends on the correctness of the prescan and, in particular, the constraints the operation imposes on the shared array `idx`: if `flag[`s`] = flag[`t`] = 1` for distinct threads s and t (so that both s and t will write to the output array), then race-freedom requires that `idx[`s`]` $\neq$ `idx[`t`]`.

Figure 2 presents an OpenCL kernel for stream compaction, to be executed by a group of `n` threads. The φ annotations that follow `barrier()` statements are barrier invariants that we will use to prove the prescan specification in Section 4. The id of a thread is denoted `tid`, and `data`, `out`, `flag` and `idx` are arrays declared in GPU shared memory. The prescan operation is performed inline, implemented as an algorithm due to Blelloch [4]. Surveying several GPU code repositories (the AMD APP SDK[2], the NVIDIA CUDA SDK[3] and the SHOC,[4] Rodinia [7] and Parboil [33] benchmarks) we found this prescan implementation used frequently.

Although the algorithm works in-place over the `idx` array, it is helpful to see that the correctness comes from viewing the array as a tree. If the length n of `idx` is a power of two, the array can be viewed as a balanced binary tree of depth $\lg n$. The example on the right of Figure 2, due to Blelloch [4], shows the state of `idx` as the algorithm proceeds: colouring the elements that are updated at each iteration shows the tree traversed by the algorithm. Each iteration of the upsweep and downsweep can be identified by the value of `offset`, which gives the 'distance' between vertices at that depth of the tree, and `d`, which denotes the number of active threads. The parallelism of the algorithm is due to the vertices at the same depth being updatable simultaneously. The figure shows the assigned thread (T0, T1, T2, T3) per sub-operation of the upsweep and downsweep. Note that the thread-to-element assignment changes at different depths of the tree.

a. The **upsweep** is a reduction working from the leaves of the tree up to the root. Pairs of vertices (`idx[left]`, `idx[right]`) are summed until the root contains the full reduction and other elements form partial sums.

b. The **downsweep** combines the upsweep partial sums to give the prescan result. Initially, the identity 0 is inserted into the root, marked 'clear' in step (4) of Figure 2. The downsweep then traverses down the tree. At each level of the tree, an active vertex: (i) copies its value to its left child `idx[left]` (the dotted arrow), and (ii) sums its value with the old value of its left child `temp` (this value is a partial sum generated by the upsweep), storing the value in its right child `idx[right]` (the black arrows).

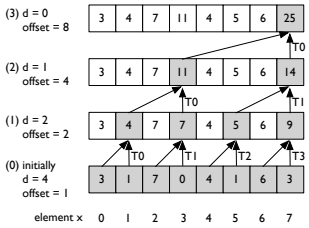
The prescan ensures that `idx`$[x] = \sum_{i=0}^{x-1}$ `flag`$[i]$ for all x. The stream compaction kernel additionally guarantees that the input is non-negative: `flag`$[x] \geq 0$ for all x. Together these imply that the output satisfies a monotonic property: for all $x < y$ we have `idx`$[x] +$ `flag`$[x] \leq$ `idx`$[y]$. Hence, for all $x \neq y$, if `flag`$[x] > 0 \wedge$ `flag`$[y] > 0$ then `idx`$[x] \neq$ `idx`$[y]$, which suffices to prove race-freedom.

[2] `http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/`

[3] `https://developer.nvidia.com/gpu-computing-sdk`

[4] `https://github.com/spaffy/shoc/wiki`

```
// data, out, flag, idx: arrays in shared memory
unsigned offset, d, left, right, temp;

// (i) test each element with predicate p
flag[tid] = p(data[tid]);
// (ii) compute indices for compaction
barrier(); // φ_load
if (tid < n/2) {
  idx[2*tid] = flag[2*tid];
  idx[2*tid + 1] = flag[2*tid + 1];
}

// (ii)(a) upsweep
offset = 1;
for (d = n/2; d > 0; d /= 2) {
  barrier(); // φ_us
  if (tid < d) {
    left = offset * (2 * tid + 1) − 1;
    right = offset * (2 * tid + 2) − 1;
    idx[right] += idx[left];
  }
  offset *= 2;
}

// code continues here
// (ii)(b) downsweep
if (tid == 0) idx[n−1] = 0;

for (d = 1; d < n; d *= 2) {
  offset /= 2;
  barrier(); // φ_ds
  if (tid < d) {
    left = offset * (2 * tid + 1) − 1;
    right = offset * (2 * tid + 2) − 1;
    temp = idx[left];
    idx[left] = idx[right];
    idx[right] += temp;
  }
}
barrier(); // φ_spec

// (iii) compaction
if (flag[tid]) out[idx[tid]] = data[tid];
```



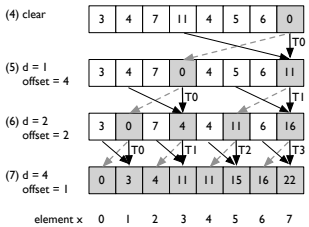


Figure 2. Stream compaction as an OpenCL kernel using n threads with the prescan inline

Blelloch [4] proves that his algorithm satisfies the prescan specification using induction on the pre-order traversal of the tree. Our aim is to prove race-freedom of kernels such as stream compaction which use the prescan algorithm by direct source code analysis. We are not aware of any verification technology that allows direct verification of source code for massively parallel kernels using induction, thus we cannot encode Blelloch's proof of correctness directly.

In Sections 4 and 5 we show that the prescan specification necessary to prove race-freedom of kernels that use this algorithm can be established using *barrier invariants*; we also study the application of barrier invariants to two further prefix sum algorithms.

3. Barrier Invariants

We now present our main novel contribution: *barrier invariants*, which allow precise reasoning about GPU kernels using the two-thread reduction. We first present a simple kernel programming language with barrier invariants and a concrete semantics (Section 3.1). We then present an abstract semantics that employs the two-thread reduction, and prove a soundness result: if a kernel can be proved data race-free with respect to the abstract semantics, then the kernel is race-free with respect to the concrete semantics (Section 3.2). Finally, we discuss practical issues associated with implementing barrier invariants efficiently in the GPUVerify tool (Section 3.3).

Our presentation of barrier invariants could be generalised to an n-thread reduction for any $n \geq 2$. This generalisation is conceptually straightforward but notationally cumbersome. In practice we have not found examples where it is necessary to consider more than two threads simultaneously to prove data race-freedom of a kernel, thus for clarity we make our presentation specific to the two-thread case.

kernel	::=	**threads**: number;
		main: stmt;
stmt	::=	basic_stmt | stmt; stmt | $\mathbf{barrier}_{\mathsf{iexpr}}$
basic_stmt	::=	name := expr | name := sh[expr] | sh[expr] := expr
expr	::=	constant literal | name | expr op expr
name	::=	thread identifier tid | any valid C name
iexpr	::=	constant literal | sh[iexpr] | name | $\overline{\mathsf{name}}$ | iexpr op iexpr

Figure 3. Syntax for our kernel programming language

3.1 Concrete operational semantics for GPU kernels

In prior work we formally presented a GPU kernel programming language [2]. In order to present barrier invariants in a self-contained manner, yet with minimal repetition of prior work, we consider a simple language for straight-line GPU kernels exhibiting no control-flow or procedure calls, according to the syntax of Figure 3. For detailed handling of additional language constructs, which is orthogonal to the issues associated with barrier invariants, see [2]; our implementation of barrier invariants in GPUVerify handles the OpenCL and CUDA languages in full.[5]

A kernel declares the number of threads that will execute (**threads**: number) and a (possibly compound) statement which is the body of the kernel. The set name denotes local variables of a kernel and includes the special read-only variable tid, the unique thread identifier of each thread. Kernel statements allow assignment to local variables, access to shared memory (sh), and barrier synchronisation.

Each **barrier** statement is annotated with an invariant $\varphi \in$ iexpr; the meaning of which is formalised below. A barrier invariant φ is implicitly quantified over all pairs of

[5] It is interesting to note that while verification condition generation for straight-line programs is straightforward (e.g., using weakest preconditions) and does not require invariant abstractions, the same is not true of straight-line GPU kernels. This is because multiple threads are in flight concurrently, updating shared memory; barrier invariants are required to reflect these updates when threads synchronise at barriers.

$$((sh, l, R, W), v := e) \to (sh, l[v \mapsto e^l], R, W) \quad \text{(T-ASSIGN)}$$

$$((sh, l, R, W), v := sh[e]) \to (sh, l[v \mapsto sh(e^l)], R \cup \{e^l\}, W) \quad \text{(T-RD)}$$

$$((sh, l, R, W), sh[e_1] := e_2) \to (sh[e_1^l \mapsto e_2^l], l, R, W \cup \{e_1^l\}) \quad \text{(T-WR)}$$

Figure 4. Rules for thread execution of basic statements

distinct threads, where a local variable v appears directly or as $\overline{v}$ if it refers, respectively, to the v of the first or second thread in a pair. For example, if x is a local variable, the barrier invariant $sh[x + tid] = sh[\overline{x} + \overline{tid}]$ can be read as $\forall\ s \neq t : sh[x_s + s] = sh[x_t + t]$, where x_s and x_t refer to the local variable x of threads s and t, respectively, and where tid is replaced by s and $\overline{tid}$ by t.

We do not allow quantifiers to appear explicitly in barrier invariant expressions. In principle we could drop this restriction, but as discussed in the introduction a key advantage of the two-thread reduction is that it *avoids* the need to reason about quantifiers.

Concrete semantics Let P be a kernel executed by n threads, and let Word refer to the set of all memory words, which we assume provides a representation for Boolean and bit-vector data. A *thread state* for P is a tuple

$$(sh, l, R, W) \in \mathsf{ThreadStates}$$

where: $sh : \mathbb{N} \to \mathsf{Word}$ is the shared memory of the kernel; $l : \mathsf{name} \to \mathsf{Word}$ is the storage for the local variables of the thread; $R, W \subseteq \mathbb{N}$ are *read* and *write* sets recording the shared addresses the thread has accessed since the last barrier. The read and write sets are used in the operational semantics to perform race checking.

A *kernel state* Σ for P is a tuple

$$(sh, (l_0, R_0, W_0), \ldots, (l_{n-1}, R_{n-1}, W_{n-1}))$$

such that $l_t(tid) = t$ for all $0 \leq t < n$, and where sh is the shared memory of the kernel and (sh, l_t, R_t, W_t) is the thread state associated with thread t for all $0 \leq t < n$.

We refer to the shared memory of kernel state Σ as $\Sigma.sh$ and use $\Sigma(t)$ to denote the thread state (sh, l_t, R_t, W_t) of thread t; we also write $\Sigma(t).l$, $\Sigma(t).R$, and $\Sigma(t).W$ to refer to the thread-specific components of this thread state. A state Σ is said to be a valid *initial state* of P if $\Sigma(t).R = \Sigma(t).W = \emptyset$ for all $0 \leq t < n$.

The rules of Figure 4 define the evolution of thread states whilst executing a basic statement. Evaluation of a local expression e given a local store l is denoted e^l. The rules define local variable and shared state updates in a standard manner; in addition, T-RD and T-WR record in R and W, respectively, the shared locations that are read and written.

$$\frac{\forall\ 0 \leq t < n : (\Sigma(t), \mathsf{basic\_stmt}) \to \sigma_t \quad \mathsf{race}(\sigma_0, \ldots, \sigma_{n-1})}{(\Sigma, \mathsf{basic\_stmt}; ss) \to_k error} \quad \text{(K-RACE)}$$

$$\frac{\begin{array}{c}\forall\ 0 \leq t < n : (\Sigma(t), \mathsf{basic\_stmt}) \to \sigma_t \\ \neg\mathsf{race}(\sigma_0, \ldots, \sigma_{n-1}) \\ sh' = \mathsf{merge}(\sigma_0, \ldots, \sigma_{n-1}) \\ \forall\ 0 \leq t < n : \Sigma'(t) = (sh', \sigma_t.l, \sigma_t.R, \sigma_t.W)\end{array}}{(\Sigma, \mathsf{basic\_stmt}; ss) \to_k (\Sigma', ss)} \quad \text{(K-STEP)}$$

$$\frac{\exists\ 0 \leq s \neq t < n : \neg[\![\varphi]\!]_\Sigma^{s,t}}{(\Sigma, \mathbf{barrier}_\varphi; ss) \to_k error} \quad \text{(K-BAR-ERR)}$$

$$\frac{\begin{array}{c}\forall\ 0 \leq s \neq t < n : [\![\varphi]\!]_\Sigma^{s,t} \\ \forall\ 0 \leq t < n : \Sigma'(t) = (\Sigma.sh, \Sigma(t).l, \emptyset, \emptyset)\end{array}}{(\Sigma, \mathbf{barrier}_\varphi; ss) \to_k (\Sigma', ss)} \quad \text{(K-BAR-INV)}$$

Figure 5. Rules for lock-step execution of a kernel

Figure 5 defines the operational semantics of kernels, where $error$ is a designated error state. If execution is guaranteed to abort when a data race occurs then for verification purposes it suffices to consider a single arbitrary schedule of thread execution between a pair of barriers [2, 11, 25, 26]. In Figure 5 threads execute in *lock-step*: all threads execute the first statement of the kernel, then all threads execute the second statement, etc. In rules K-RACE and K-STEP, $\sigma_0, \ldots, \sigma_{t-1}$ are the thread states reached by each thread on executing the given basic statement according to the rules of Figure 4. The race predicate is used to check whether execution of this basic statement would cause a data race, indicated by a conflict between the read/write sets of the threads:

$$\mathsf{race}(\sigma_0, \ldots, \sigma_{n-1}) \triangleq \exists\ 0 \leq s \neq t < n : (\sigma_s.R \cup \sigma_s.W) \cap \sigma_t.W \neq \emptyset\,.$$

If a race occurs, rule K-RACE causes execution to go to $error$. Otherwise, rule K-STEP allows the kernel to transition to a new kernel state where the local store and read/write sets for thread t are taken from thread state σ_t, and where the shared state is derived by merging the shared state associated with each thread state σ_t according to the write sets:

$$\mathsf{merge}(\sigma_0, \ldots, \sigma_{n-1})(z) \triangleq \sigma_t.sh(z)$$

where t is such that $z \in \sigma_t.W$ and $t = 0$ otherwise. Observe that there is at most one t with $z \in \sigma_t.W$ if there was no data race.

On reaching a barrier, K-BAR-ERR and K-BAR-INV check if the associated barrier invariant holds for all distinct pairs of threads. The valuation of barrier invariant φ with respect to state Σ and threads s and t is denoted $[\![\varphi]\!]_\Sigma^{s,t}$, and defined as:

$$
\begin{aligned}
[\![\text{constant literal}]\!]_{\Sigma}^{s,t} &= \text{constant literal} \\
[\![sh[\mathsf{iexpr}]]\!]_{\Sigma}^{s,t} &= \Sigma.sh([\![\mathsf{iexpr}]\!]_{\Sigma}^{s,t}) \\
[\![\mathsf{name}]\!]_{\Sigma}^{s,t} &= \Sigma(s).l(\mathsf{name}) \\
[\![\overline{\mathsf{name}}]\!]_{\Sigma}^{s,t} &= \Sigma(t).l(\mathsf{name}) \\
[\![\mathsf{iexpr}_1\, op\ \mathsf{iexpr}_2]\!]_{\Sigma}^{s,t} &= [\![\mathsf{iexpr}_1]\!]_{\Sigma}^{s,t}\ op\ [\![\mathsf{iexpr}_2]\!]_{\Sigma}^{s,t}
\end{aligned}
$$

Note that a variable v occurring as v, respectively $\overline{v}$, is evaluated in the context of thread s, respectively t. Rule K-BAR-ERR causes execution to go to $error$ if the invariant does not hold for some pair of threads. Otherwise, rule K-BAR-INV resets the read/write sets for all threads and allows execution to proceed beyond the barrier.

3.2 Two-thread reduction with barrier invariants

We now define an abstract semantics for a kernel P with respect to a pair of distinct threads (s,t). Using $\mathcal{A}^{s,t}(P)$ to denote P interpreted with respect to this abstract semantics, we show the following:

Theorem 3.1 (Soundness)**.** *Let P be a kernel executed by n threads. If for every pair $0 \leq s \neq t < n$, no execution of $\mathcal{A}^{s,t}(P)$ from a valid initial state leads to $error$, then no execution of P from a valid initial state leads to $error$.*

Hence, it suffices to prove data race-freedom, and validity of barrier invariants, with respect to the abstract semantics.

Before describing the abstract semantics formally and proving Theorem 3.1, we describe the abstract semantics in an intuitive manner. The semantics models a pair of distinct threads (s,t) executing the kernel. When executing a sequence of statements between two barriers, it is as if s and t are the only threads executing the kernel. They perform local updates and access the shared state, recording all shared locations that are accessed in their read/write sets. The read-/write sets are used to detect data races between s and t: if a data race occurs, execution aborts.

On reaching a barrier, a check is made to determine whether the invariant φ associated with the barrier holds for the pair (s,t). This check must take into account possible updates to the shared state made by additional threads executing the kernel. We handle this by considering whether a shared memory location v was accessed by s or t since the last barrier. There are two key cases:

- **v was accessed by at least one of s,t:** We say that v is a *known* location. In this case it is sound to assume that v has not been modified by any thread $r \notin \{s,t\}$. This is because our analysis considers *all possible pairs of threads*: if r can write to v then because some $x \in \{s,t\}$ accesses v a data race will be detected for the pair (x,r) and the kernel will not be deemed correct.
- **v was not accessed by s or t:** We say that v is an *unknown* location. In this case we must consider that v could have been modified by some thread $r \notin \{s,t\}$. A safe approximation is to assume that v contains an *arbitrary* value when evaluating the barrier invariant.

If there exists an assignment to unknown locations yielding a state in which φ does not hold for (s,t), execution aborts: there may be a concrete execution that would lead to this state. Thus with the two-thread reduction, user-provided barrier invariants are not taken on trust — they are **checked**.

If φ can be shown to hold for the pair (s,t) for all assignments to unknown locations then, because (s,t) is arbitrary, it is sound to assume φ holds for *all* pairs of distinct threads. Abstract execution for the pair (s,t) continues after the barrier by transitioning to an abstract state in which:

- the local stores for (s,t) and the values of *known* locations are preserved;
- the read and write sets for (s,t) are cleared;
- φ is assumed to hold for all pairs of distinct threads.

Armed with this intuition, we now formally present the abstract semantics.

Abstract semantics A thread state is defined as in Section 3.1. An *abstract kernel state* T for P with respect to threads s and t is a tuple $(sh, (l_s, R_s, W_s), (l_t, R_t, W_t))$. An abstract kernel state is similar to a (concrete) kernel state, except that only the states of threads s and t are represented. We use $T(s)$ and $T(t)$ to denote the thread states of s and t. We write $\mathsf{known}^{s,t}(T)$ for the set of shared locations collectively accessed by s and t since the last barrier:

$$\mathsf{known}^{s,t}(T) \triangleq T(s).R \cup T(s).W \cup T(t).R \cup T(t).W\,.$$

The key aspects of abstract kernel execution are defined by the rules of Figure 6; we omit abstract versions of rules K-RACE and K-STEP of Figure 5 which are defined in the obvious way by restricting these rules to two threads.

On reaching a barrier, A-BAR-ERR and A-BAR-INV check whether the barrier invariant φ holds in all concrete states that agree with the current abstract state on the local stores of s and t and on the values of *known* shared locations. Formally, we define a concretisation operator $\gamma^{s,t}$ yielding the set of such concrete states given an abstract state T:

$$
\begin{aligned}
\gamma^{s,t}(T) \triangleq \{&\Sigma \text{ a concrete kernel state} \\
&\mid \Sigma(s).l = T(s).l \wedge \Sigma(t).l = T(t).l \\
&\wedge \textstyle\bigwedge_{v \in \mathsf{known}^{s,t}(T)} (\Sigma.sh(v) = T.sh(v))\}
\end{aligned}
$$

Note that the state of the read/write sets of s and t are not preserved by $\gamma^{s,t}$; our semantics does not require this.

Rule A-BAR-ERR aborts if there exists a concrete state $\Sigma \in \gamma^{s,t}(T)$ such that the barrier invariant does not hold in Σ for the pair (s,t). If the barrier invariant holds for (s,t) in *every* state of $\gamma^{s,t}(T)$ then execution proceeds past the barrier by rule A-BAR-INV. A concrete state $\Sigma' \in \gamma^{s,t}(T)$ is chosen such that the barrier invariant φ holds in Σ' for *every*

$$\frac{\Sigma \in \gamma^{s,t}(T) \qquad \neg[\![\varphi]\!]^{s,t}_{\Sigma}}{(T, \mathbf{barrier}_{\varphi}; ss) \rightarrow_k \mathit{error}} \text{ (A-BAR-ERR)}$$

$$\frac{\begin{array}{c}\forall\, \Sigma \in \gamma^{s,t}(T) : [\![\varphi]\!]^{s,t}_{\Sigma} \qquad \Sigma' \in \gamma^{s,t}(T) \\ \forall\, 0 \leq x \neq y < n : [\![\varphi]\!]^{x,y}_{\Sigma'} \qquad T' = \alpha^{s,t}(\Sigma')\end{array}}{(T, \mathbf{barrier}_{\varphi}; ss) \rightarrow_k (T', ss)} \text{ (A-BAR-INV)}$$

Figure 6. Barrier rules for two-threaded execution

pair of distinct threads (x, y). The abstract state T' resulting from barrier synchronisation is obtained by projecting Σ' with respect to threads s and t while emptying the read/read sets of s and t and discarding the local stores and read/write sets of all other threads. Formally this is described using an abstraction operator $\alpha^{s,t}$:

$$\alpha^{s,t}(\Sigma) \triangleq (\Sigma.sh, (\Sigma(s).l, \emptyset, \emptyset), (\Sigma(t).l, \emptyset, \emptyset)) \,.$$

We explain in Section 3.3 how the universal quantifiers used by rule A-BAR-INV are eliminated in practice.

Theorem 3.1 now follows by contradiction assuming that P has a race or violates a barrier invariant while $\mathcal{A}^{s,t}(P)$ does not. The argument hinges on the fact that any step of P not ending in *error* can be mimicked by $\mathcal{A}^{s,t}(P)$, due to the lack of data races and barrier invariant violations up to the step. We present a formal proof in Appendix A.

3.3 Integrating barrier invariants with GPUVerify

We have implemented support for barrier invariants in GPUVerify [2], our verification tool for OpenCL and CUDA kernels which is built on top of the Boogie verifier [1] and Z3 SMT solver [15]. GPUVerify transforms a kernel annotated with barrier invariants into a sequential program that models the execution of two threads using the abstract semantics of Section 3.2. Our implementation fully supports loops, conditionals and procedures using lock-step predicated execution as described in [2, 12]. We describe what we believe are the most interesting aspects associated with integrating barrier invariants with GPUVerify.

Quantifier elimination Rule A-BAR-INV of Figure 6 uses quantifiers to (a) consider all concretisations of the current abstract state when checking the barrier invariant, and (b) select a successor state in which the barrier invariant holds for all pairs of distinct threads. To avoid the need to reason directly about quantifiers we now explain how quantifiers are eliminated in practice.

Elimination of the quantifier for concretisation Because barrier invariant expressions are quantifier-free, a barrier invariant φ refers to a finite, typically small, number of shared memory locations. Let d be the largest number of syntactically distinct shared memory accesses appearing in any single barrier invariant in a given kernel. We introduce d auxiliary variables, $u_1, \ldots, u_d$, which track at most d *unknown* shared memory addresses. At the start of execution, and after each barrier, these variables are set nondeterministically to reflect the fact that neither thread has accessed any shared memory location since the last barrier. After each shared memory access to a location computed from expression e, a statement $\mathit{assume}(\bigwedge_{1 \leq i \leq d} u_i \neq e)$ is inserted to reflect the fact that this location is now in the *known* set. For a barrier invariant φ, let $e_1, \ldots, e_f$ be the syntactically distinct sub-expressions of φ that refer to the shared state, where $f \leq d$. For ease of explanation, assume that there is no nesting between these sub-expressions. Let φ' be identical to φ except that each occurrence of e_i is replaced with $\texttt{ite}(e_i = u_i, \star, e_i)$, where `ite` is the *if-then-else* operator — this evaluates to e_i *unless* e_i is equal to the ith *unknown* location, in which case evaluation is nondeterministic. The formula φ' is then checked for the pair of threads under consideration. This transformation ensures that verification only succeeds if the truth of a barrier invariant is independent of unknown shared locations, which is what rules A-BAR-ERR and A-BAR-INV of Figure 6 require. We handle nested sub-expressions via a straightforward extension of this method.

As noted above, this approach relies on barrier invariants being quantifier-free. If a barrier with invariant φ follows a loop and the loop iterates a large or statically unbounded number of times then, without using quantifiers, it may not be practical or possible, respectively, for φ to refer to the set of all shared locations accessed during loop execution. We have not thus far found a practical example where this limitation is problematic.

Elimination of the quantifier for successor state selection Rule A-BAR-INV allows execution to continue after a barrier by choosing a successor state in which the barrier invariant is assumed to hold for all pairs of distinct threads. Clearly it is sound to weaken this assumption to the case where the invariant holds only for a selected subset of thread pairs that are relevant to the statements following the barrier. However, if too small a subset is chosen, the assumption following the barrier may not be strong enough to prove data race-freedom of statements following the barrier, or to establish successive barrier invariants.

In practice we have found that it is possible to derive a small subset of pairs sufficient for verification to succeed, avoiding the need for quantifiers. GPUVerify requires a set of *instantiation expressions* to be provided with each barrier invariant; eliminating the need for quantification. For example, to state that a barrier invariant should be assumed only for the pair of threads under consideration and the pair consisting of their immediate right neighbours (if they exist), we would specify the set $\{(tid, \overline{tid}), (tid + 1, \overline{tid} + 1)\}$. In Section 4.1 we illustrate a set of instantiation expressions for one of the Blelloch prescan invariants.

Modular and staged verification GPUVerify supports reasoning about kernels with loops and procedure calls through

loop invariants and procedure contracts; we have implemented front-end support for this using the CLANG/LLVM compiler framework. For data-dependent kernels, it may be necessary for barrier invariant-style expressions of the form iexpr to occur in loop invariants, referring to the shared state and variables of the form $\overline{v}$. In this case, the loop invariant is established by checking that such an expression holds for the pair of threads (s, t) under consideration. However, because threads do not necessarily synchronise at the head of a loop, during loop abstraction such a loop invariant may only be instantiated for the pair (s, t), and not for additional pairs of threads. A similar argument applies when assuming that a post-condition of a procedure holds when replacing a procedure with its specification.

GPUVerify also supports *staged* verification: if race-freedom of a procedure can be established using a simple set of barrier invariants then it is sound to prove a post-condition of the procedure using a richer set of barrier invariants under the *assumption* of race-freedom. Thus the richer barrier invariants and the post-condition can be checked without the burden of simultaneously performing data race analysis.

In Section 5 we discuss how modular and staged analysis are used in verifying race-freedom for the stream compaction kernel of Section 2.

4. Barrier Invariants for Stream Compaction

We are now equipped to tackle the data-dependent stream compaction kernel of Figure 2. After establishing some notation, we explain how barrier invariants are derived for the Blelloch prescan which is at the core of the stream compaction kernel (Section 4.1). The required barrier invariants are intricate, capturing the key properties of Blelloch's algorithm. We do *not* envisage automatic inference of such invariants, and argue that manual specification of invariants for key library functions such as prefix sums is worthwhile due to their widespread use. Our verification technique automatically checks barrier invariants, thus these complex assertions are *not* taken on trust by the tool (this is in contrast to related work on thread contracts [21], see Section 6).

We briefly discuss two further widely used prefix sum algorithms for which we have derived barrier invariants (Section 4.2), and provide some insights from our experience deriving barrier invariants for these examples.

Preliminaries For presentation purposes we separate the `idx` array used by both the upsweep and downsweep phases into two arrays: a `sum` array used in the tree reduction of the upsweep and a `prescan` array used by the downsweep that will contain the expected output of the prescan. In our experiments (Section 5), verification is performed on unmodified source code, which does not make use of this simplification and instead introduces `sum` as a *ghost variable* (an auxiliary variable used only for verification): the upsweep and downsweep work over the same array and we take a snapshot of the array in-between the two loops and store it in `sum`.

The specification we will prove is that the output is monotonic: for all threads $s < t$, $\texttt{prescan}[s] + \texttt{flag}[s] \leq \texttt{prescan}[t]$. For unsigned (i.e., non-negative) inputs we can use barrier invariants to prove this specification under the assumption that addition of unsigned integers does not overflow. Without this assumption the specification does not hold: for sufficiently large inputs, the additions performed during the prescan will overflow, leading to unexpected results in the `idx` array, and thus erroneous accesses to the `out` array. In applications, stream compaction is used under the implicit assumption that the inputs will not lead to overflow, thus our assumption is pragmatic. In Section 5 we discuss how this assumption is realised in practice for our experimental evaluation.

As discussed in Section 2, although the prescan algorithm works in-place over the same `idx` array, we can view the upsweep and downsweep as working over a logical tree where we can identify each iteration of the upsweep and downsweep by the value of the `offset` variable, which gives the 'distance' between vertices at that depth of the tree. In Figure 7(c) we show that a vertex of the tree can be specified as an (element, offset)-coordinate (x, θ) using the predicate $\mathsf{isvertex}(x, \theta) \triangleq (x+1) \bmod \theta = 0$. For example, $(5, 2)$ is a vertex as element $x = 5$ is updated when offset $\theta = 2$, while $(5, 4)$ is not a vertex. We will use the indexing functions $\mathsf{ai}(tid, \theta) \triangleq \theta(2{\cdot}tid{+}1){-}1$ and $\mathsf{bi}(tid, \theta) \triangleq \theta(2{\cdot}tid{+}2){-}1$, which give the indices of the left and right child vertices for a thread tid at offset θ. For example, $\mathsf{ai}(1, 2) = 5$ and $\mathsf{bi}(1, 2) = 7$ (see Figures 2(a) and (b)).

We note that the thread-private variables `d` and `offset` are used in both the upsweep and downsweep loops and additionally are 'uniform' between threads: all threads agree on the value of these variables at each barrier. This property is easily encoded using the barrier invariant $\texttt{d} = \overline{\texttt{d}} \wedge \texttt{offset} = \overline{\texttt{offset}}$ and is important because the barrier invariants that follow use the `offset` variable as a measure of progress of the upsweep and downsweep. GPUVerify infers such *uniform* invariants automatically using static analysis. The variable `offset` is always a power of two and ranges from 1 to n in the upsweep loop and back from n to 1 in the downsweep loop.

4.1 Barrier invariants for the Blelloch prescan

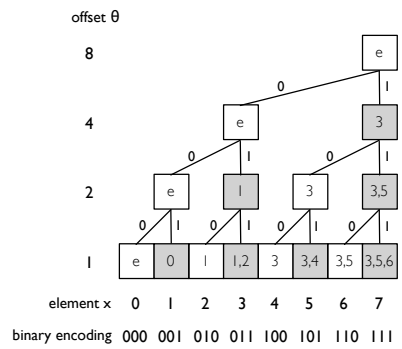
We use barrier invariants to establish equalities between the shared arrays `flag`, `sum` and `prescan`: the φ_{us} invariant, in the upsweep loop, gives equalities between elements of `sum` and `flag`, while the φ_{ds} invariant, in the downsweep loop, gives equalities between elements of `prescan` and `sum`. The tables in Figures 7(a) and (b) give the set of equalities given by the barrier invariants for $n = 8$ at the end of their respective loops. The monotonic specification, expressed as the invariant φ_{spec} for the final barrier, is then a combination of the relevant earlier equalities. We discuss each barrier invariant in turn.

offset	1	2	4	8
$\texttt{sum}[0] = \texttt{flag}[0]$				
$\texttt{sum}[1] = \texttt{flag}[1]$	$+\,\texttt{sum}[0]$			
$\texttt{sum}[2] = \texttt{flag}[2]$				
$\texttt{sum}[3] = \texttt{flag}[3]$	$+\,\texttt{sum}[2]$	$+\,\texttt{sum}[1]$		
$\texttt{sum}[4] = \texttt{flag}[4]$				
$\texttt{sum}[5] = \texttt{flag}[5]$	$+\,\texttt{sum}[4]$			
$\texttt{sum}[6] = \texttt{flag}[6]$				
$\texttt{sum}[7] = \texttt{flag}[7]$	$+\,\texttt{sum}[6]$	$+\,\texttt{sum}[5]$	$+\,\texttt{sum}[3]$	

(a) Upsweep equalities given by φ_{us}

$\texttt{prescan}[0] = e$	$\texttt{prescan}[4] = \texttt{sum}[3]$
$\texttt{prescan}[1] = \texttt{sum}[0]$	$\texttt{prescan}[5] = \texttt{sum}[3] + \texttt{sum}[4]$
$\texttt{prescan}[2] = \texttt{sum}[1]$	$\texttt{prescan}[6] = \texttt{sum}[3] + \texttt{sum}[5]$
$\texttt{prescan}[3] = \texttt{sum}[1] + \texttt{sum}[2]$	$\texttt{prescan}[7] = \texttt{sum}[3] + \texttt{sum}[5] + \texttt{sum}[6]$

(b) Downsweep equalities given by φ_{ds}



(c) Tree structure of the algorithm. Right child vertices are shaded. Each vertex (x, θ) is labelled with the summations formed as the downsweep proceeds, where $a, b, \ldots$ denotes $\texttt{prescan}[x] = \texttt{sum}[a] + \texttt{sum}[b] + \cdots$ and e denotes the identity.

Figure 7. Upsweep and downsweep equalities for $n = 8$ and the tree structure of the algorithm

Load invariant The stream compaction kernel is defined over n threads, however, only $n/2$ threads are required for the prescan. After a parallel test of each element using n threads, we synchronise with a barrier so $n/2$ threads can continue with the prescan. The state that we need to carry through this barrier is

$$\varphi_{\text{load}} \triangleq \texttt{sum}[tid] = \texttt{flag}[tid]\,.$$

Upsweep invariant The upsweep is a reduction working from the leaves of a tree up to its root. At each iteration of the upsweep, each parent vertex is given the sum of its left and right child vertices. The upsweep proceeds until the root of the tree contains the full reduction and other elements form partial sums: each vertex of the tree will contain the sum of the leaf vertices below it in the tree.

In Figure 7(c) we show the tree structure for the concrete case where $n = 8$ and we shade right child vertices (for the moment, ignore the labels of each vertex, which will be used in the downsweep). The upsweep loop has the loop invariant:

$$(\texttt{d} = 4 \wedge \texttt{offset} = 1) \vee (\texttt{d} = 2 \wedge \texttt{offset} = 2)$$
$$\vee\, (\texttt{d} = 1 \wedge \texttt{offset} = 4) \vee (\texttt{d} = 0 \wedge \texttt{offset} = 8)$$

where each conjunct characterises a loop iteration (and the last conjunct corresponds to the final time the loop head is evaluated and the loop exits). By considering the state of the array `sum` at the start of each iteration we can construct a barrier invariant φ_{us} for the barrier inside the loop body. The columns of Figure 7(a) summarise the per-element equalities formed by the upsweep as the `offset` changes.

Informally, the summations of an element x at offset θ can be found by traversing the tree from the vertex (x, θ) to the leaf vertex $(x, 1)$ of the element: if a right child vertex (x, θ') is encountered the term $\texttt{sum}[x - \theta']$ is added to the summation. Furthermore, an index $x - \theta'$ is a summation term iff $\mathsf{isvertex}(x, 2\theta')$ holds. This gives us the following per-element invariant upsweep, which captures the state of any vertex (x, θ) of the tree.

$$\mathsf{upsweep}(x, \theta) \triangleq \texttt{sum}[x] = \texttt{flag}[x] + \sum_{\substack{\theta' \in \{2^j \mid 0 \le j < \lg \theta\} \\ \mathsf{isvertex}(x, 2\theta')}} \texttt{sum}[x - \theta']\,.$$

Using this per-element invariant, we define the barrier invariant φ_{us} by considering the elements of `sum` known to a thread tid in each iteration of the upsweep loop with offset θ. At loop entry, when $\texttt{offset} = 1$, each thread $tid < n/2$ knows about exactly two elements: $\mathsf{ai}(tid, 1)$ and $\mathsf{bi}(tid, 1)$. In subsequent loop iterations we have an uneven distribution of elements over threads as more threads become disabled while the upsweep proceeds. For each previous `offset` value $\theta'(< \theta)$, each thread $tid < n/\theta'$ will continue to know about its left child at that depth of the tree, i.e., $\mathsf{ai}(tid, \theta'/2)$, because this vertex will have no further summation terms. For the current `offset` value θ, each thread $tid < n/\theta$, i.e., each thread active in the iteration of the upsweep just completed, will know about its left and right vertices: $\mathsf{ai}(tid, \theta/2)$ and $\mathsf{bi}(tid, \theta/2)$.

Thus φ_{us} is defined as

$$tid < n/2 \Rightarrow (\mathsf{upsweep}(\mathsf{ai}(tid, 1), 1) \wedge \mathsf{upsweep}(\mathsf{bi}(tid, 1), 1))$$

in the case $\theta = 1$, and as

$$\bigwedge_{\theta' \in \{2^i \mid 1 \le i \le \lg \theta\}} (tid < n/\theta' \Rightarrow \mathsf{upsweep}(\mathsf{ai}(tid, \theta'/2), \theta)) \wedge (tid < n/\theta \Rightarrow \mathsf{upsweep}(\mathsf{bi}(tid, \theta/2), \theta))$$

otherwise.

Downsweep invariant The downsweep combines the partial sums formed in the upsweep to give the expected prescan result. For presentation purposes we show the downsweep operating over an array `prescan` different from the array `sum` used in the upsweep, where initialisation is such that $\texttt{prescan}[x] = \texttt{sum}[x]$ for all elements x.

The downsweep traverses the tree from the root to the leaves after clearing the root with the identity element e. At each iteration of the downsweep, each vertex (i) copies its value to its left child and (ii) sums its value with the old value of the left child into the right child. This means that a vertex is only summed into when it is a right child; otherwise, it receives the value of its parent vertex.

In Figure 7(c) we consider the concrete case where $n = 8$. We shade right child vertices and label each vertex with the summation terms of each vertex after the downsweep has processed that level of the tree. For example, the root $(7, 8)$ is labelled with e to denote that it is equal to the identity (i.e., $\texttt{prescan}[7] = e$), when $\texttt{offset} = 8$.

At the next downsweep iteration, when $\texttt{offset} = 4$, only $(7, 8)$ is a vertex. The vertex copies its value e into its left child $(3, 4)$ and sums this value with the previous value of its left child, i.e., $\texttt{sum}[7] + \texttt{sum}[3] = e + \texttt{sum}[3] = \texttt{sum}[3]$, storing the result in its right child $(7, 4)$. In the figure, we write this summation by labelling the vertex $(7, 4)$ with '3'.

In the next iteration, when $\texttt{offset} = 2$, vertex $(7, 4)$ copies its value $\texttt{sum}[3]$ into its left child $(5, 2)$ and sums this value with the previous value of its left child, i.e., $\texttt{prescan}[7] + \texttt{sum}[5] = \texttt{sum}[3] + \texttt{sum}[5]$ into its right child $(7, 2)$. Hence, in the figure, the left $(5, 2)$ and right $(7, 2)$ child vertices are respectively labelled '3' and '$3, 5$'. The remaining vertices are similarly computed.

Now consider the leaf vertices of the tree, which are the final output of the prescan. We note that the number of terms in the summation for an element x is the number of right child vertices in the path from the root vertex to the leaf vertex. For example, element 5 has two right child vertices on the path from the root vertex $(7, 8)$ to its leaf vertex.

Informally, the summations of an element x at offset θ can be found by traversing the tree from the vertex (x, θ) to the root and gathering terms: if a right child vertex (x', θ') is encountered then add the term $\texttt{sum}[x' - \theta']$ to the summation. This observation leads to the following invariant, which defines the elements of `prescan` in terms of the offset θ:

$$\begin{aligned}\mathsf{downsweep}(x, \theta) &\triangleq \texttt{prescan}[x] \\ &= \begin{cases} \sum_{\theta' \in B(x,\theta)} \texttt{sum}[y(x, \theta')] & \text{if } \mathsf{isvertex}(x, 2\theta) \\ \texttt{sum}[x] & \text{otherwise} \end{cases}\end{aligned}$$

where

$$B(x, \theta) \triangleq \{2^i \mid \lg \theta \leq i < \lg n \wedge \mathsf{bit}(x, i) = 1\}$$

and

$$y(x, \theta') \triangleq x - \theta' + \sum_{0 \leq j < \lg \theta', \mathsf{bit}(x,j)=0} 2^j$$

We exploit the fact that the binary encoding of x may be interpreted as the path from the root to the element leaf vertex: using 0 to mean left and 1 to mean right as we read from the most-to-least significant bit (see also Figure 7(c)).

Finally, we define the invariant φ_{ds} by considering the elements of `prescan` accessed by the thread tid in each iteration of the downsweep loop with offset θ:

$$\begin{aligned}\varphi_{\text{ds}} \triangleq \textstyle\bigwedge_{0 \leq i < \lg n} \big(&tid < n/2^{i+1} \wedge \theta \geq \lfloor 2^i/2 \rfloor \\ &\Rightarrow \mathsf{downsweep}(\mathsf{ai}(tid, 2^i), \theta)\big) \\ \wedge \textstyle\bigwedge_{0 \leq i < \lg n} \big(&tid < n/2^{i+1} \wedge \theta \bowtie \lfloor 2^i/2 \rfloor \\ &\Rightarrow \mathsf{downsweep}(\mathsf{bi}(tid, 2^i), \theta)\big)\end{aligned}$$

where $\bowtie$ is defined as $\geq$ if $2^i = n/2$, and as $=$ otherwise.

Specification invariant The result of the prescan is expressed in the final barrier invariant:

$$\begin{aligned}\varphi_{\text{spec}} \triangleq tid < \overline{tid} \Rightarrow (&\texttt{prescan}[tid] + \texttt{flag}[tid] \\ &\leq \texttt{prescan}[\overline{tid}])\end{aligned}$$

This is a simple consequence of the equalities formed by φ_{us} and φ_{ds}, i.e. at $\theta = n$ for φ_{us} and $\theta = 0$ for φ_{ds}. For example, consider the case $tid = 1$ and $\overline{tid} = 5$. Then we must show $\texttt{prescan}[1] + \texttt{flag}[1] \leq \texttt{prescan}[5]$. By the downsweep equalities, given in Figure 7(b), we can rewrite this equation as: $\texttt{sum}[0] + \texttt{flag}[1] \leq \texttt{sum}[3] + \texttt{sum}[4]$. Then, by the upsweep equalities, given in Figure 7(a), and cancellations, this becomes $0 \leq \texttt{sum}[2] + \texttt{flag}[3] + \texttt{sum}[4]$, which holds given unsigned (and thus non-negative) inputs.

Instantiation expressions As discussed in Section 3.3, GPUVerify requires a set of instantiation expressions to be specified for each barrier invariant. As an example, we briefly discuss instantiation expressions for the upsweep barrier invariant, φ_{us}.

In the upsweep of Figure 2, we see at each iteration that a thread tid uses the results produced by threads $2 \cdot tid$ and $2 \cdot tid + 1$ from the previous iteration. For example, thread 1 at offset 2 uses the sums formed by 2 and 3 when the upsweep was at offset 1. Therefore, it suffices to instantiate φ_{us} for threads $2 \cdot tid$ and $2 \cdot tid + 1$ after the upsweep barrier; instantiating the invariant for further threads does not add any information that is useful for verification.

4.2 Other prefix sum algorithms

We have derived barrier invariants for two further prefix sum algorithms which, through our survey of the CUDA and AMD SDKs and SHOC, Rodinia and Parboil benchmarks (see Section 2), we found to be widely used in GPU kernel programming: *Brent-Kung* [5] and *Kogge-Stone* [23]. We briefly outline the process of barrier invariant discovery for these algorithms which can also be used to perform stream compaction. The source code for these examples, annotated with barrier invariants, is available online.[6]

[6] `http://multicore.doc.ic.ac.uk/tools/GPUVerify/OOPSLA13`

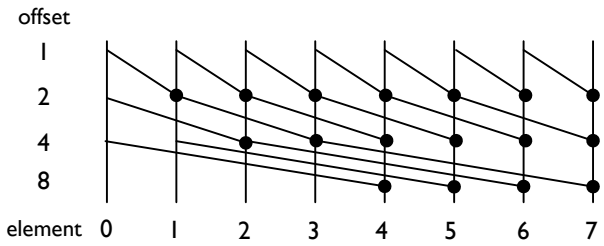


Figure 8. Circuit for Kogge-Stone prefix sum

Brent-Kung The Brent-Kung algorithm performs an inclusive prefix sum (also known as a *scan*) meaning that the output is the sum of all inclusive-prefixes: i.e., $\texttt{idx}[x] = \sum_{i=0}^{x} \texttt{flag}[i]$ for all x. Like the Blelloch prescan, the Brent-Kung scan consists of upsweep and downsweep loops. The barrier invariants for this algorithm follow a similar pattern to those for Blelloch: we establish equalities for the upsweep and downsweep loop that can be combined into the monotonic specification. The Brent-Kung upsweep matches the Blelloch upsweep so we reuse the same upsweep barrier invariant. The downsweep then collects partial sums to form the final result. Unlike the Blelloch downsweep, which operates over a logical tree, the Brent-Kung downsweep works over a logical *forest*, which requires significantly different barrier invariants.

Kogge-Stone Figure 8 describes the Kogge-Stone inclusive prefix sum as a circuit diagram for $n = 8$ elements. There is a wire for each input and data flows top-down through the diagram. Each node $\bullet$ performs the binary associative operator on its two inputs and produces an output that passes downward and also optionally across the circuit (through a diagonal wire). The algorithm works by summing array elements from successively larger power-of-two offsets. In this algorithm we observe that the circuit always adds adjacent summation intervals. Initially, at offset 1, we have for each element x that $\texttt{idx}[x] = \texttt{flag}[x] = \sum_{i=x}^{x} \texttt{flag}[i]$. After the first iteration, at offset 2, each element $x \geq 1$ has $\texttt{idx}[x] = \sum_{i=x-1}^{x} \texttt{flag}[i]$. After the second iteration, at offset 4, each element $x \geq 2$ has $\texttt{idx}[x] = \sum_{i=x-3}^{x} \texttt{flag}[i]$. For example, element 3 will have summed $\texttt{idx}[3] = \sum_{i=2}^{3} \texttt{flag}[i]$ and $\texttt{idx}[1] = \sum_{i=0}^{1} \texttt{flag}[i]$ to become $\sum_{i=0}^{3} \texttt{flag}[i]$. In general, we see that each add has the form $\sum_{i=a}^{b} \texttt{flag}[i] + \sum_{i=b+1}^{c} \texttt{flag}[i] = \sum_{i=a}^{c} \texttt{flag}[i]$.

The GPU kernel implementation of Kogge-Stone assigns one thread to each array element. Due to the tightly coupled nature of threads, we found it difficult to design a compact barrier invariant to summarise the state of the `idx` array, contrary to what we were able to do for the Blelloch and Brent-Kung prefix sums. However, using the above observation regarding the summations taking place, we were able to apply abstract interpretation [13] to the source code using a domain of summation intervals, so that each element $\texttt{idx}[x]$ is represented by an abstract element (a, b) denoting $\sum_{i=a}^{b} \texttt{flag}[i]$. Addition of adjacent intervals is defined as $(a, b) \oplus (b+1, c) = (a, c)$. This enables us to prove the scan specification for this algorithm for any associative operator using simple barrier invariants.

4.3 Experience

The barrier invariants we have derived for the Blelloch, Brent-Kung and Kogge-Stone prefix sum algorithms are intricate. Their derivation was non-trivial and in the process we made numerous errors and encountered numerous omissions. To identify such problems we relied heavily on the fact that our method does not take user-supplied barrier invariants on trust, but checks that they do indeed hold.

The process of barrier invariant derivation required (a) a thorough understanding of how these algorithms operate and why they are functionally correct, and (b) a strong intuition for writing inductive invariants, gained through significant prior experience using verification tools such as Boogie [1]. Due to (b), we speculate that writing barrier invariants for complex algorithms is likely to be beyond the scope of general GPU software developers who do not have a background in verification. Nevertheless, we believe there is significant value in applying barrier invariants to the verification of important library functions such as prefix sums, due to the wide use of such functions and the fact that the verification effort need only be undertaken once. The specification for the library function may be useful in *automatic* verification of a client application, as we demonstrate for the stream compaction example in our experimental evaluation (Section 5).

There was some degree of re-use between barrier invariants in our examples. The *upsweep* phases of Blelloch and Brent-Kung are identical, allowing identical barrier invariants, and the process of deriving the Blelloch downsweep invariant yielded insights which made derivation of the Brent-Kung downsweep invariant relatively straightforward. The Kogge-Stone algorithm, on the other hand, required a different approach, as discussed in Section 4.2. Having identified a barrier invariant, we found that the necessary instantiation expressions (as briefly discussed at the end of Section 4) were straightforward to identify.

5. Experimental results

We now present verification results for analysing the stream compaction kernel and the Blelloch, Brent-Kung and Kogge-Stone prefix sum algorithms discussed in Sections 2 and 4. We evaluate the effectiveness of GPUVerify in (a) performing modular verification of the stream compaction kernel using a specification for the prefix sum phase, and (b) the scalability of verifying the prefix sums using barrier invariants.

We compare GPUVerify with two other GPU kernel analysis tools, GKLEE [26] and GKLEE$_p$ [27], which are based on the KLEE dynamic symbolic execution engine [6]. GKLEE searches for bugs in GPU kernels through exhaustive path analysis using an explicit representation of threads.

GKLEE$_p$ tries to improve on the performance of GKLEE by reasoning about representative pairs of threads, introducing fresh pairs to the analysis only when necessary. While the tools are not designed for verification, they can perform brute-force verification via exhaustive path exploration.

We also attempted to provide a performance comparison with KLEE-CL [11], another KLEE-based tool, but were unable to do so due to bugs in KLEE-CL, which we have reported to the developers. Because GKLEE and KLEE-CL share many similarities, we expect that a comparison with KLEE-CL after bug-fixing will yield similar results. We do not compare with existing verification tools, PUG [25], the tool of Leung et al. [24], and GPUVerify prior to this work [2], as they do not support reasoning about data-dependent kernels.

All experiments were performed on a compute cluster using nodes with Intel Xeon EP-2620 cores at 2GHz with 16GB RAM running RedHat Linux 6.3, rev. 2784 of Boogie, and Z3 v4.3.1. We used a timeout of 3 hours for each experiment. Times reported are averages over ten runs. GPUVerify and our benchmarks are available online to make our results reproducible.

Verification strategy We apply GPUVerify using a modular and staged verification strategy, as discussed in Section 3.3, exploiting the fact that the prefix sum phase of stream compaction is outlined into a procedure. Race-freedom for the stream compaction is verified by summarising the prefix sum procedure by its monotonic specification. We consider three implementations of prefix sum, using the Blelloch, Brent-Kung and Kogge-Stone algorithms (see Section 4). In each case, we verify race-freedom of the prefix sum using trivial barrier invariants. This is because the prefix sums themselves are *data-independent*: data-dependence arises when the result of a prefix sum is used for array indexing during stream compaction. Having established race-freedom for the prefix sum, we use the barrier invariants described in Section 4 to prove the monotonic specification, with race checking disabled.

The experimental results we present for GPUVerify all use this staged approach; we found that staging these verification phases always accelerated verification, yielding speedups of up to $4\times$. GKLEE and GKLEE$_p$ are based on exhaustive path exploration, thus do not support modular and staged analysis.

Verifying stream compaction using the prefix sum specification We considered verifying race-freedom for stream compaction, assuming the prefix sum specification, for all power-of-two thread counts from 2 to 2^{31}. GPUVerify supports specification-based reasoning directly. Using GKLEE and GKLEE$_p$ a procedure can be summarised through the use of *assume* commands, one per thread.

Figure 9 shows for each thread count up to 2^{15} the time in seconds taken for analysis with GPUVerify, GKLEE and GKLEE$_p$. For each thread count and tool we show the time taken for analysis, in seconds, averaged over 10 runs. Below the time we show the variation observed between runs. Timeouts are indicated by 'TO'.

In all cases analysis with GPUVerify succeeded in *less than two seconds*, illustrating the power of modular analysis using the two-thread reduction; we obtained similar results for thread counts up to 2^{31}. GKLEE is capable of analysis of up to 8 threads within our resource limits, and GKLEE$_p$ scales further to 256 threads. Because GKLEE and GKLEE$_p$ use an explicit thread representation it is unsurprising that analysis does not scale to larger thread counts. We emphasise that GKLEE and GKLEE$_p$ were not designed for this sort of analysis; nevertheless, they provide the only source for comparison. For large thread counts, significant variation in analysis time is observed using GKLEE$_p$. We attribute this to various sources of nondeterminism arising from the KLEE execution environment [6].

Verifying prefix sum specifications For stream compaction we are interested in performing a prefix sum using the + operator with identity 0. Because, as discussed in Section 4, the specification does not hold in the case where addition overflows, we have added to GPUVerify support for a non-overflowing bit-vector addition operator, which performs addition of an n-bit bit-vector using $n+1$ bits, and then injects an *assume* statement to restrict analysis to the case where addition did not overflow, indicated by the top bit of the result. It was not possible to add similar support directly to GKLEE, thus for our experiments with GKLEE we model bit-vector addition as saturating.

The monotonic specification also holds with equivalent barrier invariants when the operator is changed to *max* or | (bitwise-or). It is instructive to see how the verification cost varies according to the operator used.

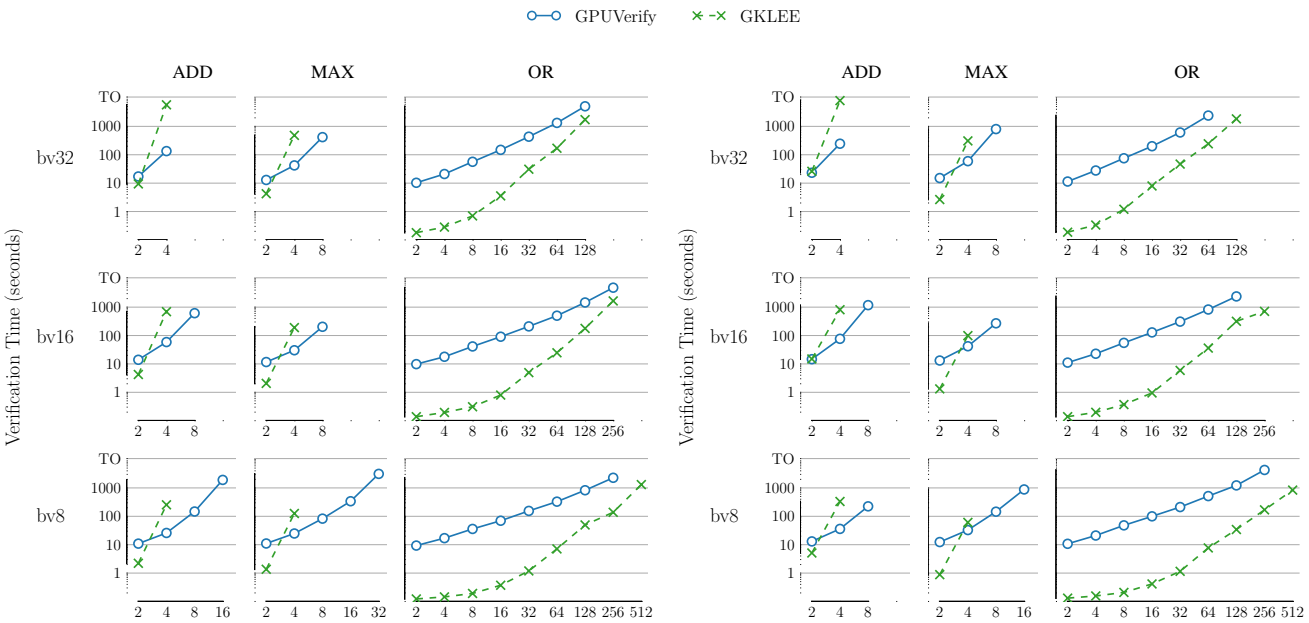
We found that GKLEE$_p$ could not be used to verify the monotonic specification for these kernels: analysis led to false positive reports of the monotonic post-condition failing. This is because, as mentioned briefly in [27], the thread reduction employed by GKLEE$_p$ relies on a form of shared state abstraction similar to the *adversarial* abstraction discussed in Section 1. GKLEE on the other hand does not attempt to perform thread reduction and so does not use any shared state abstraction, hence we are able to meaningfully compare GPUVerify with GKLEE.

The graphs of Figure 10 show times for verifying (a) the Blelloch prescan algorithm and (b) the Brent-Kung scan algorithm for increasing thread counts using GPUVerify (indicated by circles) and GKLEE (indicated by crosses). We consider each of the operators + (ADD), *max* (MAX) and | (OR), and consider 8-, 16- and 32-bit integer representations: bv*X* indicates that integers are represented using *X*-bit bit-vectors.

For GPUVerify, each data-point is the total time to verify the algorithm using our staged verification strategy: i.e., the time to verify the algorithm is race-free (using trivial bar-

#Threads	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768
GPUVerify	1.47 ±0.04	1.43 ±0.01	1.43 ±0.05	1.47 ±0.04	1.39 ±0.02	1.40 ±0.02	1.38 ±0.02	1.38 ±0.02	1.41 ±0.02	1.38 ±0.02	1.39 ±0.03	1.40 ±0.02	1.41 ±0.02	1.41 ±0.02
GKLEE	0.80 ±0.08	76.95 ±0.27	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
GKLEE$_p$	0.57 ±0.01	1.03 ±0.04	3.38 ±0.06	9.40 ±0.19	39.44 ±0.97	220.89 ±5.01	1838.71 ±110.81	TO	TO	TO	TO	TO	TO	TO

Figure 9. Experimental results for modular analysis of the stream compaction kernel using GPUVerify, GKLEE and GKLEE$_p$. Analysis times, in seconds, are averages over 10 runs, and the variation (95% confidence interval) between runs is also shown. Timeouts are indicated by 'TO'.



(a) Blelloch prescan

(b) Brent-Kung scan

Figure 10. Verification results for (a) Blelloch and (b) Brent-Kung prefix sums using concrete operators of differing bit-widths

rier invariants) plus the time to verify the monotonic specification (with race checking disabled). For GKLEE, each data-point is the time taken for exhaustive path analysis. For each tool, absence of a data point indicates that a timeout occurred, or that the memory limit of our platform was reached.

The results show that for both Blelloch and Brent-Kung with the + and *max*, the two-thread reduction afforded by barrier invariants allows GPUVerify to scale to larger thread counts than with GKLEE, overcoming the overhead associated with barrier invariants observed for small thread counts. The scalability of GPUVerify is particularly noticeable when integers are represented using 8-bits. For larger bit-widths the scalability of both tools degrades.

For the | operator, which leads to simpler bit-vector reasoning, both GPUVerify and GKLEE perform significantly better than with + or *max* for both the Blelloch and Brent-Kung algorithms. GKLEE performs extremely well for small thread counts, outperforming GPUVerify by two orders of magnitude. As the thread count increases, this performance gap closes to one order of magnitude or less, and the scalability of GPUVerify appears to be almost linear (note that although the y-axis is plotted to a log scale, the number of threads on the x-axis also grows exponentially). Nevertheless, in four cases GKLEE is able to verify one larger thread configuration than GPUVerify, before both tools exhaust resource limits. Looking at the log files created by GKLEE, we found that analysis for the | operator required consideration of just a single execution path regardless of the thread count, while analysis using + and *max* required exploring a number of paths exponentially proportional to the thread count. This is because the application of saturating addition, or the *max* function, leads to a branch in the LLVM intermediate representation on which GKLEE operates. In each case the branch is data-dependent on the operands of the function,

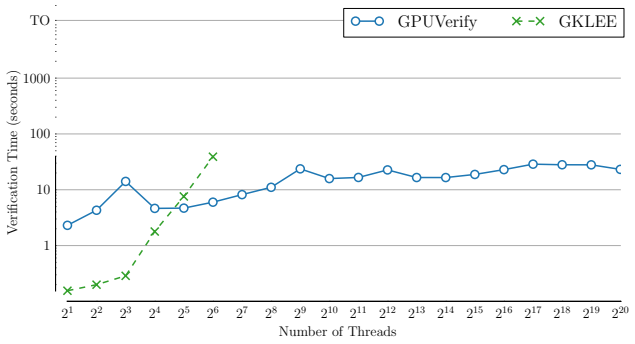


Figure 11. Verification results for Kogge-Stone prefix sum

so each time this function is applied GKLEE must fork execution to consider all successive execution states that can be reached via both branch outcomes. The | function does not lead to a branch, so this exponential path explosion is avoided. This is not an issue for GPUVerify, which does not perform a per-path analysis.

As explained in Section 4.2, we applied source-level abstract interpretation to the Kogge-Stone prefix sum algorithm. This leads to a description that is independent of the specific associative operator that is used. Figure 11 plots verification times using GPUVerify and GKLEE for abstract Kogge-Stone for varying thread counts, where integers are represented using 32-bit bit-vectors. Data points for GPUVerify and GKLEE are represented using circles and crosses, respectively.

The graph shows that verification with GPUVerify scales extremely well: we were able to verify the example for all power-of-two thread counts up to 2^{31} (thread counts up to 2^{20} are shown in the figure), and that analysis appears to be insensitive to the number of threads. Using GKLEE, analysis was possible for up to 64 threads within our 3 hour timeout. This demonstrates that scalability that can be achieved by combining the two-thread abstraction with additional abstract reasoning.

Scaling verification for Blelloch and Brent-Kung using an abstract operator Inspired by the excellent scalability results for the abstract Kogge-Stone algorithm (Figure 11), we considered the use of abstraction to overcome the scalability limitations associated with bit-vector reasoning for the Blelloch and Brent-Kung algorithms.

For non-negative integers x, y and z and an operator $\oplus \in \{+, max, |\}$ with identity 0, the barrier invariants for Blelloch and Brent-Kung, and the monotonic specification, rely on the following properties: $x \oplus (y \oplus z) = (x \oplus y) \oplus z$ (associativity), $x \oplus e = e \oplus x = x$ (identity), and $max(x, y) \leq x \oplus y$ (upperbound). Motivated by this, we extended GPUVerify with an abstract binary operator $\oplus$ over bit-vectors, which maps to a Boogie uninterpreted function together with a set of axioms encoding the associativity, identity and upperbound properties. Our hypothesis was that by abstracting from the complexity of bit-vector addition we could prove the prescan specification with 32-bit bit vectors for larger thread counts.

Unfortunately, specifying these axioms involved the use of quantifiers, and we found that the quantified associativity axiom led to problems with the quantifier instantiation approach of Z3, which uses E-matching [14]. If a formula has a large expression that involves $\oplus$, the associativity axiom can be instantiated for any sub-tree of the form $t_1 \oplus (t_2 \oplus t_3)$ or $t_1 \oplus (t_2 \oplus t_3)$, and instantiation leads to duplication of each of the sub-trees t_1, t_2 and t_3 in the formula. Thus direct use of quantifiers in describing properties of $\oplus$ led to rapid memory exhaustion.

We used *triggers* [14] to carefully design a set of quantified axioms for $\oplus$ which allow the barrier invariants for the upsweep and downsweep phases of the prefix sum algorithms to be checked. We have not managed to craft triggers suitable for proving the final monotonic specification; we plan to investigate this further in future work.

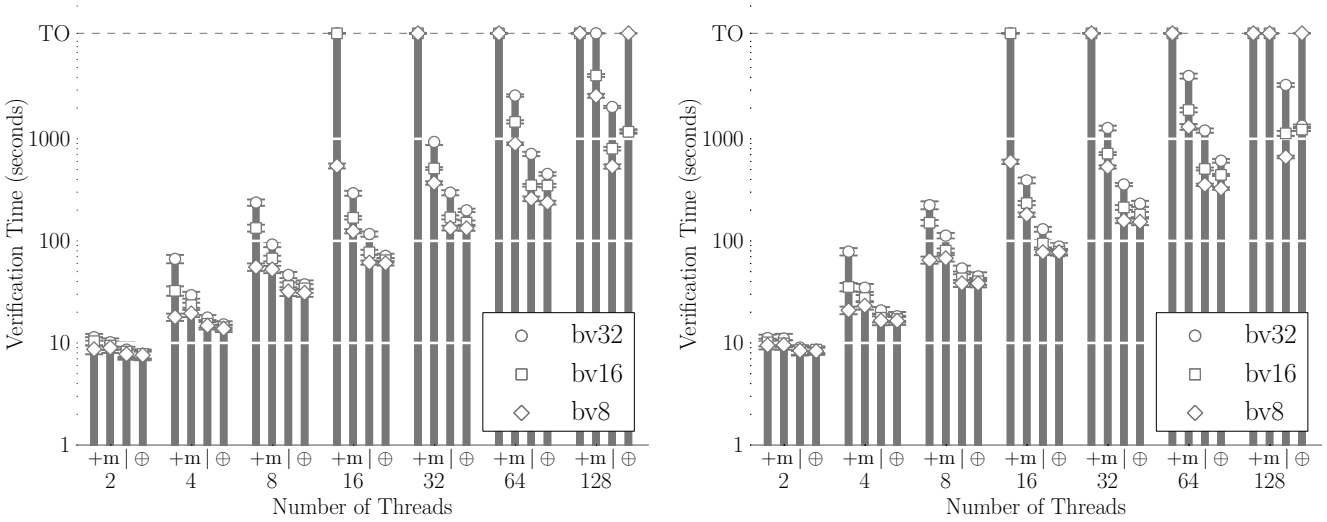
Figure 12 shows verification times for checking the barrier invariants φ_{load}, φ_{us} and φ_{ds} (see Section 4) but *not* the final specification φ_{spec}, for $\{8, 16, 32\}$-bit bit vectors, with respect to concrete operators +, *max*, |, and using the abstract operator $\oplus$. The results show that the abstract operator scales significantly better than + and *max*, allowing the (pre)scans to be verified for 32-bit bit vectors for up to 64 threads, compared with 8 threads using +. Furthermore, because all assumptions made regarding $\oplus$ hold for the concrete operators, verification of the (pre)scans using $\oplus$ alone establishes that these barrier invariants hold for all three operators simultaneously.

We could not apply GKLEE using the abstract operator since the tool does not support the use of uninterpreted functions and quantifiers.

6. Related Work

Formal analysis of GPU kernels We have discussed related work on formal analysis for GPU kernels [2, 11, 12, 24–27] in the introduction and through experimental comparison in Section 5. Among this work, the closest to GPUVerify is PUG [25]. It would be possible in principle to integrate barrier invariants with the PUG verification method in a similar manner to the approach taken here.

A recent method for functional verification of GPU kernels uses permission-based separation logic [19]. Here, a kernel must be annotated (currently manually) with an assignment of read or write permissions to memory locations on a per-thread basis. Race-freedom is proven by showing that write permissions are exclusive. To reason about communication at barriers, a *barrier specification* is required to state how permissions are exchanged between threads at a barrier. Barrier specifications differ from barrier invariants in that they talk only about permissions, not about the data on which the kernel operates. The approach of [19] does not



(a) Blelloch prescan

(b) Brent-Kung scan

Figure 12. Verifying *upsweep* and *downsweep* Blelloch/Brent-Kung using concrete and abstract (⊕) operators

employ any thread-reduction abstraction; instead, quantifiers are used to reason about all threads. This method has not yet been automated, so a systematic comparison with GPUVerify for realistic examples is not yet possible.

Protocol verification A reduction to two processes, similar to the two-thread reduction employed in GPU kernel verification, is at the heart of a method for verifying cache coherence protocols known as CMP [9], which was inspired by the foundational work of McMillan [28]. With CMP, verification of a protocol for an arbitrary number of processes is performed by model checking a system where a small number of processes are explicitly represented and a highly nondeterministic 'other' process over-approximates the possible behaviours of the remaining processes. The unconstrained nature of the 'other' process can lead to spurious counterexamples, which must be eliminated either by introducing additional explicit processes, or by adding *non-interference lemmas* so that the actions of the 'other' process more precisely reflect the possible actions of processes in the concrete system. The CMP method has been extended and generalised with *message flows* and *message flow invariants* [34], which aid in the automatic derivation of non-interference lemmas by capturing large classes of permissible interactions between processes.

Our approach uses the same high-level proof idea as the CMP method: we consider a small number of threads (two), and our default adversarial abstraction models the possible actions of all other threads, analogously to the 'other' process. The purpose of barrier invariants is to refine the adversarial abstraction so that the possible behaviours of additional threads are represented more precisely, thus barrier invariants can be seen as analogues to non-interference lemmas. However, the techniques aim to solve different problems: non-interference lemmas in the CMP method describe interaction sequences between processes in a message-based protocol, while barrier invariants for GPU kernel verification capture properties of the shared state in shared memory parallel programs, and thus there are many technical differences in the manner by which the high-level proof technique is applied in practice.

Other techniques for verification of data-parallel programs A technique for proving race-freedom of data parallel programs using *thread contracts* is presented in [21]. With this approach the user specifies a *coordination strategy*: a logical annotation describing how threads will access shared memory when executing a parallel region of code. An SMT solver is used to show that adherence to the coordination strategy guarantees race-freedom. However, the work does *not* address the problem of proving that a parallel program obeys its coordination strategy; if an erroneous coordination strategy is specified then the associated race-freedom proof cannot be trusted. The authors show how assertions can be generated to check coordination strategies at runtime, which is useful but provides no guarantees. Barrier invariants can be viewed as a kind of coordination strategy tailored towards analysis of GPU kernels. A key difference between our contribution and that of [21] is that our verification method checks the validity of barrier invariants as well as using them to prove race-freedom: if erroneous barrier invariants are provided then the verification attempt will fail.

Collective loop invariants are proposed in [32] to allow verification of data parallel programs using symbolic execution, with applications to MPI. In the context of MPI, a collective loop invariant is an assertion specified with respect to a set of processes $\mathcal{I}$ and a set of loop heads $\mathcal{L}$, and is required to hold in any state where every process $p \in \mathcal{I}$ is

at a loop head in $\mathcal{L}$. This facilitates reasoning about parallel programs where threads do not necessarily synchronise at loop heads. Like barrier invariants, collective loop invariants establish properties over sets of processes/threads, but otherwise the techniques are orthogonal: a barrier invariant is designed to capture shared state properties when all threads in a GPU kernel synchronise at the *same* barrier; collective loop invariants aim to capture the system state in MPI programs where processes do *not* frequently synchronise. We believe there is scope for process-modular reasoning about MPI programs using barrier invariants, complementing the strengths of collective loop invariants.

Thread modular reasoning There has been much work on thread-modular reasoning for general-purpose concurrent programs, notably the Owicki-Gries [31] and rely-guarantee [20] techniques.

The two-thread reduction can be viewed as a form of thread-modular reasoning: for a GPU kernel executed by n threads, each thread is (implicitly) verified with respect to $n - 1$ environment abstractions. In each such environment abstraction, a single additional thread is represented for purposes of data race analysis, and the actions of further threads are considered by modelling the shared state abstractly. Barrier invariants refine this environment model by providing a more precise representation of the shared state.

The two-thread reduction with barrier invariants exploits the structure of data-parallel programs where barriers are the only means of synchronisation. This leads to more compact specifications than are possible using traditional thread-modular techniques which must take account of arbitrary thread synchronisation.

Staged verification In Section 3.3 we discussed the use of staged verification in GPUVerify. Forms of staged analysis are often used in program verification, usually for collaboration between techniques. For example, it is common to derive simple program invariants using a sound abstract interpreter, and then to assume these invariants when applying a more heavy-weight verification method (see, e.g., [17]). The use of assumptions in collaborative verification and testing has been the subject of recent work [10].

7. Conclusions and Future Work

We have presented barrier invariants, a method for facilitating race analysis of data-dependent GPU kernels using the two-thread reduction. We have demonstrated the application of barrier invariants through a detailed case-study of stream compaction, and shown that our implementation facilitates practical verification of this important kernel for relatively large thread counts.

In future work we plan to apply barrier invariants more widely, verifying other data-dependent kernels such as radix sort, collision detection, eigenvalue computation and graph colouring. Based on this experience we will investigate techniques for automatically inferring auxiliary barrier invariants, so that users can focus on the key invariants that are specific to the algorithm in hand. We also plan to investigate the application of barrier invariants to other data parallel programming models that employ barrier synchronisation, including OpenMP and MPI, and to consider verification of kernels that use atomic operations to avoid barrier synchronisation (building on existing results in this area using GKLEE [8]).

Our experiments show that employing abstraction to avoid reasoning directly about arithmetic is promising, but our results are incomplete due to the challenges associated with quantifier instantiation. We believe that progress on this problem could have wide application in software verification.

Acknowledgments

We are grateful to the following people for their insightful comments on various drafts of this work: Adam Betts, Peter Collingbourne, Derek Graham, Marieke Huisman, Samin Ishtiaq, Kim Jarvis, Jael Kriener, Rustan Leino, Curtis Madsen, Alastair Reid, Murali Talupur and John Wickerson.

We are also grateful to Ganesh Gopalakrishnan and Peng Li for assisting us in using the GKLEE and GKLEE$_p$ tools.

References

[1] M. Barnett et al. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

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

[3] M. Billeter, O. Olsson, and U. Assarsson. Efficient stream compaction on wide SIMD many-core architectures. In *HPG*, pages 159–166, 2009.

[4] G. E. Blelloch. Prefix sums and their applications. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1990.

[5] R. P. Brent and H.-T. Kung. A regular layout for parallel adders. *IEEE Trans. Computers*, 31(3):260–264, 1982.

[6] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[7] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization*, pages 44–54, 2009.

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

[9] C.-T. Chou, P. K. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In *FMCAD*, pages 382–398, 2004.

[10] M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In *FM*, pages 132–146, 2012.

[11] P. Collingbourne, C. Cadar, and P. H. J. Kelly. Symbolic testing of OpenCL code. In *HVC*, pages 203–218, 2011.

[12] P. Collingbourne, A. F. Donaldson, J. Ketema, and S. Qadeer. Interleaving and lock-step semantics for analysis and verification of GPU kernels. In *ESOP*, pages 270–289, 2013.

[13] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[14] L. M. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *CADE*, pages 183–198, 2007.

[15] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[16] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[17] A. F. Donaldson, L. Haller, and D. Kroening. Strengthening induction-based race checking with lightweight static analysis. In *VMCAI*, pages 169–183, 2011.

[18] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison-Wesley, 2007.

[19] M. Huisman and M. Mihelčić. Specification and verification of GPGPU programs using permission-based separation logic. In *BYTECODE*, 2013.

[20] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4): 596–619, 1983.

[21] R. Karmani, P. Madhusudan, and B. Moore. Thread contracts for safe parallelism. In *PPoPP*, pages 125–134, 2011.

[22] Khronos OpenCL Working Group. The OpenCL specification, version 1.2, 2012.

[23] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Computers*, C-22(8):786–793, 1973.

[24] A. Leung, M. Gupta, Y. Agarwal, et al. Verifying GPU kernels by test amplification. In *PLDI*, pages 383–394, 2012.

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

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

[27] P. Li, G. Li, and G. Gopalakrishnan. Parametric flows: Automated behavior equivalencing for symbolic analysis of races in CUDA programs. In *SC*, pages 29:1–29:10, 2012.

[28] K. McMillan. Verification of infinite state systems by compositional model checking. In *CHARME*, pages 219–234, 1999.

[29] M. Moskal. Programming with triggers. In *SMT*, pages 20–29, 2009.

[30] NVIDIA. CUDA C programming guide, version 5.0, 2012.

[31] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976.

[32] S. F. Siegel and T. K. Zirkel. Loop invariant symbolic execution for parallel programs. In *VMCAI*, pages 412–427, 2012.

[33] J. Stratton et al. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, UIUC, 2012.

[34] M. Talupur and M. R. Tuttle. Going with the flow: Parameterized verification using message flows. In *FMCAD*, pages 1–8, 2008.

A. Proof of Theorem 3.1

Define the projection $\pi^{s,t}(\Sigma)$ as

$$(\Sigma.sh, \pi^s(\Sigma), \pi^t(\Sigma))$$

where $\pi^x(\Sigma) \triangleq (\Sigma(x).l, \Sigma(x).R, \Sigma(x).W)$. We have:

Proof (Theorem 3.1). The proof is by contradiction. Thus, suppose no execution of $\mathcal{A}^{s,t}(P)$ for any pair of threads s and t leads $error$, but that P either has a race or violates a barrier invariant. Hence, there is an execution of P from an initial state to a state where either (a) K-RACE or (b) K-BAR-ERR applies. In both cases, observe that this is due to two threads, say s and t.

Suppose $\rho = \Sigma_1, \Sigma_2, \ldots, \Sigma_n$ is the sequence of kernel states successively assumed by P during the erroneous execution with exception of the final K-RACE or K-BAR-ERR step. We will show by induction that

$$\pi(\rho) = \pi^{s,t}(\Sigma_1), \pi^{s,t}(\Sigma_2), \ldots, \pi^{s,t}(\Sigma_n)$$

is a sequence of successive kernel states of an execution of $\mathcal{A}^{s,t}(P)$ whose execution steps are exactly the execution steps of the problematic execution with each K-BAR-INV step replaced by A-BAR-INV; from this, it follows that either a K-RACE or A-BAR-ERR step is possible from $\pi^{s,t}(\Sigma_n)$, contradicting that no execution of $\mathcal{A}^{s,t}(P)$ leads to $error$.

The base case of the induction is trivial. For the successor case, consider K-STEP and K-BAR-INV in turn. For each K-STEP step from Σ_i to Σ_{i+1} a K-STEP step is possible from $\pi^{s,t}(\Sigma_i)$ to $\pi^{s,t}(\Sigma_{i+1})$, as no data races occur in ρ and, hence, the part of the state Σ_i employed by s and t is completely captured by $\pi^{s,t}(\Sigma_i)$. Similarly, for each K-BAR-INV step from Σ_i to Σ_{i+1} it follows that an A-BAR-INV step is possible from $\pi^{s,t}(\Sigma_i)$ to $\pi^{s,t}(\Sigma_{i+1})$: No execution of $\mathcal{A}^{s,t}(P)$ ends in error and, hence, for all $\Sigma \in \gamma^{s,t}(T)$ it must hold that $[\![\varphi]\!]^{s,t}_{\Sigma}$. Moreover, as $\Sigma_i \in \gamma^{s,t}(\pi^{s,t}(\Sigma_i))$ by definition of $\gamma^{s,t}$ and $\pi^{s,t}$, it follows by occurrence of K-BAR-INV in ρ that $\pi^{s,t}(\Sigma_{i+1})$ can be used as T' (observe here that Σ_{i+1} is equal to Σ_i with the read/write sets of all threads cleared). $\square$