A Sound and Complete Abstraction for Reasoning about Parallel Prefix Sums

Nathan Chong  Alastair F. Donaldson  Jeroen Ketema

Imperial College London
{nyc04,afd,jketema}@imperial.ac.uk

1. Introduction

The prefix sum operation, which given an array \( A \) computes an array \( B \) consisting of all sums of prefixes of \( A \), is an important building block in many high performance computing applications. A key example is stream compaction: suppose a group of \( n \) threads has calculated a number of data items in parallel, each thread \( t \) (\( 1 \leq t \leq n \)) having computed \( d_t \) items; now the threads must write the resulting items to a shared array \( d \) in a compact manner, i.e. thread \( t \) should write its items to a series of \( d_t \) indices of \( d \) starting from position \( d_1 + \cdots + d_{t-1} \). Compacting the data stream by serialising the threads, so that thread 1 writes its results, followed by thread 2, etc., would be slow. Instead, compaction can be performed in parallel via a prefix sum. Each thread \( t \) writes its count \( d_t \) to an array \( d \) at position \( t \), then the threads perform an exclusive parallel prefix sum (defined formally in Section 2) on \( d \) to yield an array \( index \). The prefix sum sets the elements of \( index \) to \([0, d_1, d_1 + d_2, \ldots, d_1 + \cdots + d_n]\), so that a thread \( t \) can write its results to \( d \) compactly starting from position \( index[t] \).

Stream compaction is just one example; the prefix sum operation is defined for any binary associative operator, and prefix sums using various operators have found wide application in computationally intensive tasks. A selection of examples are summarised in Table 1. For several decades the design of efficient parallel prefix sums has been an active research area. Parallel prefix sums were first implemented in hardware circuits as logic for propagating carry bits in adders [34]. The design space for these circuits has been explored and important circuits known in the literature are due to Kogge and Stone [21], Ladner and Fischer [22], and Brent and Kung [5]. More recent work by Sheeran [33] has further explored the design space. In software, prefix sum algorithms have been used in the context of evaluating polynomials in parallel [36] and Blelloch [3] has proposed the use of prefix sums as a primitive parallel operation, giving numerous examples of their application [4]. Prefix sums are now primitives in parallel programming models such as MPI [29] and OpenMP [35].

Recently there has been a great deal of interest in implementing efficient prefix sums for acceleration on massively parallel graphics processing units (GPUs) [2, 7, 12, 16, 26, 31], implemented using GPU programming models such as OpenCL [19] and CUDA [27]. The major benchmark suites for general-purpose GPU programming, SHOC [12], Rodinia [7] and Parboil [37] all include a variety of prefix sum implementations. The thrust\(^1\) and CUDA Data Parallel Primitives (cudpp\(^2\)) libraries implement efficient prefix sum primitives for NVIDIA GPUs.

We present a method for formally verifying that a generic parallel prefix sum implementation—designed to work for any data

---

* This work was supported by the EU FP7 STREP project CARP (project number 287767), the EPSRC PSL project (EP/I006761/1), and an EPSRC First Grant (EP/K011499/1).

\(^1\)http://code.google.com/p/thrust/

\(^2\)http://code.google.com/p/cudpp/
A Sound and Complete Abstraction for Reasoning about Parallel Prefix Sums

2. Background on Prefix Sums

We briefly review the definitions of a prefix sum and an exclusive prefix sum, and give examples of sequential and parallel prefix sum implementations.

Given a set $S$ with an associative binary operation $\oplus$ (i.e., a semigroup), the prefix sum of a list $[s_1, s_2, \ldots, s_n]$ of elements of $S$ is the list:

$$[s_1, s_1 \oplus s_2, \ldots, s_1 \oplus s_2 \oplus \cdots \oplus s_n]$$

consisting of all sums of prefixes, in increasing order of length. For example, if we consider the set of integers under addition, the prefix sum of $[1, 3, 5, 7]$ is $[1, 4, 9, 16]$.

If $(S, \oplus)$ has an identity element 1, so that $(S, \oplus)$ is a monoid, then the exclusive prefix sum of $[s_1, s_2, \ldots, s_n]$ is defined as:

$$[1, s_1, s_1 \oplus s_2, \ldots, s_1 \oplus s_2 \oplus \cdots \oplus s_{n-1}]$$

For example, if $S$ is the set of all four-bit binary numbers and $\oplus$ is the bitwise-or operator with identity 0000 then the exclusive prefix sum of $[0001, 0010, 0100, 1000]$ is $[0000, 0001, 0011, 0111]$.

It is trivial to implement a sequential prefix sum: the C-like program of Figure 1 computes the prefix sum of `in` into `out`.

```c
void prefixSum(const T *in, T *out) {
    out[0] = in[0];
    for(unsigned i = 1; i < n; i++)
        out[i] = out[i-1] @ in[i];
}
```

Figure 1. A sequential prefix sum for inputs of length $n$

Our second main contribution is an extension of our method and theoretical results to the case of barrier-synchronising data-parallel programs, the programming model of GPU kernels. This is a contribution over previous work on the correctness of parallel prefix sums [33, 38] which applies to synchronous parallel hardware described as sequential HASKELL programs, but not to asynchronous concurrent programs. We show that if a data-parallel program implementing a generic prefix sum can be proved free from data races then correctness of the prefix sum can be established by running a single test case using the interval of summations monoid, as in the sequential case. We use this result to design and implement a highly automatic method for verifying parallel prefix sum implementations at the level of GPU kernel source code, using the GPU-Verify tool [1] to prove data race-freedoom.

We present a large experimental evaluation using four different prefix sum algorithms implemented as OpenCL kernels. We show that race-freedom of these kernels can be proven efficiently for all power-of-two element sizes up to $2^{31}$, and that verifying the kernels by running a single test is very fast: using two NVIDIA GPUs, two Intel CPUs (for which OpenCL is also supported), and an ARM GPU, prefix sums on vectors of up to length $10^6$ can be checked in a matter of seconds. We argue that, once race-freedom has been established, the method of Voigtlander [38] can also be used to prove correctness of prefix sum implementations using testing, but show that our method dramatically outperforms Voigtlander’s approach, which requires running $O(n^2)$ tests.

Because we can efficiently prove data race-freedoom of GPU kernels implementing prefix sums for all conceivably useful sizes, and because verification using our method involves running a single test case, taking no longer than using the prefix sum in practice, we claim that we have made a major step towards solving the problem of verifying GPU implementations of generic prefix sums.

---

### Table 1. Some applications of parallel prefix sums

<table>
<thead>
<tr>
<th>Prefix sum application</th>
<th>Data-type</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream compaction [2, 16]</td>
<td>Int</td>
<td>+</td>
</tr>
<tr>
<td>Sorting algorithms [16, 30]</td>
<td>Int</td>
<td>+</td>
</tr>
<tr>
<td>Polynomial interpolation [13]</td>
<td>Float</td>
<td>$^{\times}$</td>
</tr>
<tr>
<td>Line-of-sight calculation [4]</td>
<td>Int</td>
<td>max</td>
</tr>
<tr>
<td>Binary addition [22]</td>
<td>pairs-of-bits</td>
<td>carry operator</td>
</tr>
<tr>
<td>Finite state machine simulation [22]</td>
<td>transition function composition</td>
<td></td>
</tr>
</tbody>
</table>

*Floating point multiplication is not actually associative, but is often treated as such in applications where some error can be tolerated.*
kernel void koggeStone(const local T *in, local T *out) {
    out[tid] = in[tid];
    barrier();
    for (unsigned offset = 1; offset < n; offset *= 2) {
        $T$ temp;
        if (tid ≥ offset) temp = out[tid−offset];
        barrier();
        if (tid ≥ offset) out[tid] = temp ⊕ out[tid];
        barrier();
    }
}

Figure 2. Kogge-Stone prefix sum implemented in OpenCL

| offset = 1 | 1 1 1 1 1 1 1 1 |
| offset = 2 | 1 2 2 2 2 2 2 2 |
| offset = 4 | 1 2 3 4 4 4 4 4 |

Figure 3. Evolution of the output array for the Kogge-Stone kernel of Figure 2 with $n = 8$ for an example input

where $T$ is some data type with associative binary operator $\oplus$. An exclusive prefix sum can be implemented similarly.\(^3\)

Figure 2 shows the Kogge-Stone parallel prefix sum implemented in OpenCL and designed to be executed by a single workgroup of $n$ threads.\(^4\) The koggeStone function is marked kernel to indicate that it is the entry point for a GPU kernel, and arrays in and out have the local qualifier to indicate that they are allocated in GPU memory local to the workgroup. Threads execute asynchronously, and a thread can use $\text{tid}$ to access its unique thread identifier, in the range $\{0, \ldots , n − 1\}$. Threads synchronise by calling $\text{barrier()}$, which causes all threads to wait until every thread reaches the barrier statement. On each iteration of the loop, every thread whose $\text{tid}$ is greater than or equal to $\text{offset}$ sums the value it has already computed with the value previously computed by the thread $\text{offset}$ places to the left. Figure 3 illustrates how the output array evolves when the kernel is instantiated using integer addition and applied to the input $\{1,1,1,1,1,1,1\}$, with $n = 8$.

3. Sequential Computational Model

We present a sequential imperative programming language, and define what it means to implement a correct prefix sum in this language. This provides a simple foundation on which to clearly introduce our interval of summations abstraction, which we do in Section 4. We extend our language, abstraction and theoretical results to apply to data-parallel programs (and in particular GPU kernels) in Section 5.

Syntax and typing \ The syntax for our language is shown in Figure 4, where $e$ ranges over literal values, $\nu$ and $A$ over scalar and array variable names, respectively, and $\text{op}$ over an unspecified set of binary operators. Our presentation can be easily extended to cater for operators of other arities.

\(^3\)For ease of presentation, we assume throughout that the input length $n$ is hard-coded into the program under consideration. In practice, $n$ would be supplied as a parameter.

\(^4\)For readability, we deviate a little from the precise syntax of OpenCL.

| expr $e ::= c$ | literal |
| $\nu$ | variable |
| $A[e]$ | array element |
| $e_1 \text{ op } e_2$ | operator |
| stmt $s ::= e$ | variable assignment |
| $A[e_1] := e_2$ | array element assignment |
| if $(e) \{ s_s \} \text{ else } \{ s_s \}$ | conditional |
| while $(e) \{ s_s \}$ | loop |
| stmts $ss ::= e$ | empty sequence |
| $s; ss$ | sequence |

Figure 4. Syntax

All variables, arrays and literals are typed (assuming standard syntax for variable and type declarations, which we omit). Types for scalar variables and literals are drawn from a set of base types $T$, ranged over by $T$, which includes at least integers and Booleans (denoted Int and Bool respectively)—equipped with the usual literals and operators—and the single-element type Unit. Array variables have type Array$(T)$ which denotes all maps of type Int $\rightarrow T$. For ease of presentation we assume no out-of-bounds errors occur and we do not allow arrays of arrays. The language also omits features such as pointers, procedures and unstructured control flow. In Section 6 we argue that our technique extends to languages with these features.

The typing rules of the language are straightforward and are depicted in Figure 5. As usual there is a context $T$ which specifies the types of variables.

Operational semantics \ Let Var be a set of variables and Arr a set of arrays all of which are assumed to be typed. A variable store $\sigma_v$ is a mapping

$$\text{Var} \rightarrow \{ T \}_{T \in T}$$

such that if $v \in \text{Var}$ is of type $T$, then $\sigma_v(v)$ is of type $T$. An array store $\sigma_A$ is a mapping

$$\text{Arr} \rightarrow \{ \text{Array}(T) \}_{T \in T}$$

such that if $A$ is of type Array$(T)$, then $\sigma_A(A)$ is of type Array$(T)$.

Expressions are evaluated under a variable store and an array store. Denoting the evaluation of an expression $e$ by $\lbrack e \rbrack\sigma_v^{\sigma_A}$, we define:

$\lbrack c \rbrack\sigma_v^{\sigma_A} = c$

$\lbrack A[e] \rbrack\sigma_v^{\sigma_A} = \sigma_A(A)(\lbrack e \rbrack\sigma_v^{\sigma_A})$

$\lbrack e_1 \text{ op } e_2 \rbrack\sigma_v^{\sigma_A} = \lbrack e_1 \rbrack\sigma_v^{\sigma_A} \text{ op } \lbrack e_2 \rbrack\sigma_v^{\sigma_A}$

Figure 6 gives the operational semantics for our language. The semantics is defined over program states $S = (\sigma_v, \sigma_A, ss)$ with $\sigma_v$ a variable store, $\sigma_A$ an array store and $ss$ a sequence of statements. In the figure, $ss_1 \cdot ss_2$ denotes the concatenation of sequences of statements $ss_1$ and $ss_2$. The rules are standard for an imperative language like ours, except that we have both a variable and an array store instead of just a single store. Although this split is not strictly needed here, it eases the extension to data-parallel programs in Section 5 where we shall regard variables as thread-local and arrays as shared among all threads.

Let a program $P$ be a sequence of statements. An initial state of $P$ is any program state with $P$ as the sequence of statements. Given an initial state $S_0$ of $P$, an execution of a program $P$ is a finite or infinite sequence:

$$S_0 \rightarrow S_1 \cdots \rightarrow S_i \cdots$$

with each $S_i$ $(i \geq 0)$ a program state. An execution is maximal if it (a) cannot be extended by applying one of the rules from the
operational semantics or (b) is infinite. We say \( P \) terminates for an initial state \( S \), if all maximal executions starting from \( S \) are finite. Note that the maximal execution is unique in the current case as execution is deterministic; this will no longer be so in the data-parallel case of Section 5. The proof that our type system and operational semantics satisfy the usual safety property—that types are preserved under execution and progress can always be made unless termination has occurred [28, Section 8.3]—is standard.

**Prefix sum algorithms** We now define what it means for a program in our language to implement a prefix sum algorithm. Recall from Section 2 that a prefix sum can be defined with respect to a semigroup, and an exclusive prefix sum with respect to a monoid. Because most prefix sums of interest in practice are over monoids we shall henceforth consider prefix sums over monoids. All the results we present restrict easily to the case of semigroups.

We shall write \( M \) to refer to a monoid with elements \( S_M \in T \), binary operator \( \oplus_M \) (which we assume is a programming language operator) and identity \( 1_M \in S_M \). Moreover, we say that a variable \( v \), respectively an array \( A \), is read-only in \( P \) if no assignment of the form \( v := \ldots \), respectively \( A[e] := \ldots \), occurs in \( P \).

**Definition 3.1.** Let \( M \) be a monoid, \( P \) be a program, \( n \) a natural number, and \( \text{in}, \text{out} \) arrays of type \( Array(S_M) \) such that \( \text{in} \) is read-only in \( P \). The program \( P \) computes an \( M \)-prefix sum of length \( n \) from \( \text{in} \) to \( \text{out} \) for an initial state \( S \) if \( P \) terminates for \( S \) and for each final array store \( \sigma_A \) it holds that \( \sigma_A(\text{out})(k) = \bigoplus_{0 \leq j \leq k} \sigma_A(\text{in})(j) \) (0 \( \leq k < n \)).

The program \( P \) implements an \( M \)-prefix sum of length \( n \) from \( \text{in} \) to \( \text{out} \) if \( P \) computes an \( M \)-prefix sum of length \( n \) from \( \text{in} \) to \( \text{out} \) for every initial state.

Whether a program computes a prefix sum for a given input can be established by running the program; determining whether a program implements a prefix sum amounts to functional verification. Implementation of an exclusive prefix sum is defined analogously.

Designating the array \( \text{in} \) as read-only is for ease of presentation, making it sufficient to consider only the final array store in the above definition and avoiding the need to relate final and initial array stores.

We observe that Definition 3.1 quantifies over all possible final array stores to be able to cover the data-parallel case of Section 5. In the current sequential case the final array store is unique as execution is deterministic.

Henceforth we shall assume that a prefix sum is always from \( \text{in} \) to \( \text{out} \), and shall simply talk about a program implementing an \( M \)-prefix sum of length \( n \), or computing an \( M \)-prefix sum of length \( n \) from an initial state.

### 4. The Interval of Summations Abstraction

We now turn our attention to proving the correctness of generic prefix sums: prefix sums that are designed to be polymorphic, so that they work for any type and operator that together have the properties of a monoid. We present our main theoretical result, that they work for any type and operator that together have the properties of a monoid. We present our main theoretical result, that they work for any type and operator that together have the properties of a monoid.
We call a program that makes use of $S_X$ a generic program. Akin to a generic method in JAVÁ, a template method in C++, or a function with a type class context in HASKELL, a generic program cannot be directly executed: it must first be instantiated with respect to a specific type.

**Definition 4.1.** Let $P$ be a generic program and $M$ a monoid. We write $P[M]$ to denote the program that is identical to $P$ except that every occurrence of $S_X$, $\oplus_X$, $1_X$ is replaced by $S_M$, $\oplus_M$, $1_M$, respectively. We refer to the process of obtaining $P[M]$ from $P$ as a monoid substitution.

If $M$ already occurs in $P$ then we cannot tell, from $P[M]$ alone, which uses of $M$ were already present in $P$ or originated from uses of $X$. We handle this issue as follows, to simplify our formal treatment: if $M$ occurs in $P$ then we choose a monoid $M'$ isomorphic to $M$ such that $M'$ does not occur in $P$, and we define $P[M]$ to be $P[M']$. For ease of presentation we still refer to the monoid $M$ as $M$.

It is easy to see that monoid substitution is well-defined, and that if the abstract program $P$ is well-typed according to the rules of Figure 5 then applying a monoid substitution to $P$ leads to a well-typed program.

**Definition 4.2** (Generic prefix sums). Let $P$ be a generic program. Then $P$ implements a generic prefix sum of length $n$ if $\text{in}$ and $\text{out}$ are of type $\text{Array}(S_X)$ and, for every monoid $M$, $P[M]$ implements an $M$-prefix sum of length $n$.

The interval of summations monoid: The key insight behind our result is the observation that a generic prefix sum can only rely on the properties of a monoid: associativity and the existence of an identity element. Relying on any additional properties specific to a particular operator, such as commutativity, idempotence or distributivity with respect to some other operator, would render the prefix sum inapplicable in general.

Suppose we wish to compute a prefix sum of length $n$ from array $\text{in}$ into array $\text{out}$, with respect to an arbitrary monoid $M$. Thus $\text{in}$ and $\text{out}$ have type $\text{Array}(S_M)$, and the prefix sum operator is $\oplus_M$. If the prefix sum is correctly implemented then at the end of the computation each element of $\text{out}$ must be the sum of a contiguous sequence of elements of $\text{in}$. For example, $\text{out}[0]$ should be equal to $\text{in}[0]$ and $\text{out}[3]$ to $\text{in}[0] \oplus_M \text{in}[1] \oplus_M \text{in}[2] \oplus_M \text{in}[3]$. As computation progresses, these contiguous summations are built up using the $\oplus_M$ operator, initially starting with individual elements of $\text{in}$.

Consider, for any $0 \leq k < n$, how $\text{out}[k]$ could be computed (to give the main intuition in a simple manner we ignore possible uses of the identity element): either (a) the existing value of a variable or array element could be copied into $\text{out}[k]$ or (b) $\text{out}[k]$ could be derived by summing two values of $S_M$, $c$ and $d$, say, using $\oplus_M$. In the latter case

- $c$ and $d$ are summations of contiguous elements of $\text{in}$: $c = \bigoplus_{0 \leq j < i_1} \text{in}[j]$ and $d = \bigoplus_{i_2 < j \leq k} \text{in}[j]$, for some $i_1$ and $i_2$ (where an empty summation is defined to be $1_M$), and
- the summations $c$ and $d$ must ‘kiss’: we must have $i_1 + 1 = i_2$.

If $c$ and $d$ did not have these forms then, using only the laws of associativity and identity, it would not be possible to rewrite $c \oplus_M d$ into the form $\bigoplus_{j \in (i_1, i_2]} \text{in}[j]$, the required value for $\text{out}[k]$. By a similar argument, if $c$ (similarly $d$) is in turn constructed by summing two values $e$ and $f$ then $c$ and $f$ must both be contiguous summations of elements of $\text{in}$ that kiss, otherwise it would not be possible, using just the monoid laws, to rearrange $c \oplus_M f$ into the form of $c$. Continuing this argument we can see that a correct generic prefix sum must compute its result by successively summing partial sums of contiguous input elements that kiss; if a non-contiguous sum is ever constructed then this sum cannot contribute to the final prefix sum result.

We introduce a monoid, the interval of summations monoid, which captures abstractly the notion of contiguous summation intervals, and the operation of summing pairs of kissing intervals.

**Definition 4.3**. The interval of summations monoid I (henceforth referred to as the interval monoid) has the elements

$$I_I = \{(i_1, i_2) \in \text{Int} \times \text{Int} \mid i_1 \leq i_2\} \cup \{I_f, I_T\}$$

and a binary operator $\oplus_I$ defined by:

$$\begin{align*}
  1_I \oplus_I x &= x \oplus_I 1_I = x & \text{for all } x \in I_I \\
  \top \oplus_I x &= \top \oplus_I \top = \top & \text{for all } x \in I_I \\
  (i_1, i_2) \oplus_I (i_3, i_4) &= \begin{cases} 
    (i_1, i_4) & \text{if } i_2 + 1 = i_3 \\
    \top & \text{otherwise}
  \end{cases}
\end{align*}$$

It is easily checked that $I_I$ defines a monoid with identity $I_f$.

The interval monoid abstractly describes summations of elements in with respect to an arbitrary monoid, say, $M$. A pair $(i_1, i_2)$ abstractly describes the summation $\bigoplus_{i_1 \leq i \leq i_2} \text{in}[i]$. The identity $I_f$ represents an empty interval of summations, thus is an abstraction of $1_M$. Finally, the element $\top$ represents all non-empty summations of elements in with respect to $\oplus_M$ that are not known to correspond to contiguous summations.

The $\oplus_I$ operator precisely captures the effect of summing two contiguous summations. For $i_1 \leq i_2 < i_3$,

$$\bigoplus_{I_I, i_1 \leq i \leq i_2} \text{in}[i] \oplus_M \bigoplus_{I_I, i_2 + 1 \leq i \leq i_3} \text{in}[i] = \bigoplus_{I_I, i_1 \leq i \leq i_3} \text{in}[i]$$

and, correspondingly, $(i_1, i_2) \oplus_I (i_3, i_4) = (i_1, i_4)$.

The way $\oplus_I$ treats $I_f$ expresses the fact that adding an empty summation to either side of a summation interval has no effect. Finally, the treatment of $\top$ by the operator captures the argument above: using only the properties of a monoid, it is not possible to transform a non-contiguous summation into a contiguous one by applying the $\oplus_M$ operator. Notice that $\top$ is an absorbing element, or annihilating element, of $I_I$: once a summation has become non-contiguous there can be no return.

Superficially, it may appear that the interval monoid is similar to the abstract domain of intervals that is commonly used in abstract interpretation [11]. The domains are in fact very different: an element $(i, j)$ of our interval of summations domain represents a single concrete value that can be expressed as a summation, whereas an element $[i, j]$ of the traditional domain of intervals represents the set of values $\{i, i + 1, \ldots, j\}$.

We now define a condition on initial program stores which, for a given natural number $n$, ensures that the first $n$ elements of $\text{in}$ are abstracted in the interval monoid by appropriate singleton summation intervals, and that all other array elements and variables of type $S_X$ have values that are not known to be summation intervals.

**Definition 4.4** (Singleton condition for sequential programs). Let $P$ be a generic program with $\text{in}$ of type $\text{Array}(S_X)$. An initial state of $P$ with variable store $\sigma_V$ and array store $\sigma_A$ satisfies the singleton condition for $n$ if

1. for all $v$ of type $S_X$ in $P$, $\sigma_V(v) = \top$;
2. for all $A$ of type $\text{Array}(S_X)$ in $P$ and $k \in \text{Int}$,

$$\sigma_A(A)(k) = \begin{cases} 
  (k, k) & \text{if } A = \text{in} \text{ and } k \in \{0, \ldots, n - 1\} \\
  \top & \text{otherwise}
\end{cases}$$

We can now state our main theorem which shows that, with respect to generic prefix sums, the interval monoid is a sound and complete abstraction: a program implements a generic prefix sum
if and only if it implements a prefix sum when instantiated with the interval monoid. In fact, our result is stronger than this: it shows that correctness of a generic prefix sum can be established by considering the interval monoid instantiation only for those initial stores that satisfy the singleton condition.

Theorem 4.5 (Soundness and completeness). Let $P$ be a generic program and $n$ a natural number. Then:

\[ P \text{ implements a generic prefix sum of length } n. \]

\[ \iff \]

\[ P[I] \text{ computes an I-prefix sum of length } n \text{ for every initial state satisfying the singleton condition for } n. \]

Observe that monoid substitution (Definition 4.1) satisfies the following algebraic laws for any monoid $M$:

\[ (v := e)[M] \equiv v := (e[M]) \]

\[ (A[e_1] := e_2)[M] \equiv A[e_1] := (e_2[M]) \]

\[ (\text{if } e \{ s_{s_1} \text{ else } s_{s_2} \}[M] \equiv \text{if } e \{ s_{s_1}[M] \text{ else } s_{s_2}[M] \} \]

\[ (\text{while } e \{ ss \}[M] \equiv \text{while } (e) \{ ss[M] \} \]

\[ e[M] \equiv e \]

\[ (s; ss)[M] \equiv s[M]; ss[M] \]

where it is immediate by the typing rules and the definition of generic programs that neither $\oplus_X$ nor $1_X$ can occur in either the expression $e_1$ of $A[e_1] := e_2$ or the expression $e$ of a conditional or loop. For the same reason, it follows that if the first statement in $Q$ is a loop or conditional with guard $\epsilon$, then $\epsilon$ will evaluate identically in both $S_I$ and $S_M$. Thus, whatever the form of $Q$, the same rule from the operational semantics applies in both $S_I$ and $S_M$. Let the resulting program states after application of the rule be $S'_I$ and $S'_M$. By the algebraic laws it is now immediate that there exists a unique generic program $Q'$ such that $Q'[I]$ is the sequence of statements of $S'_I$ and $Q'[M]$ is the sequence of statements of $S'_M$.

It remains to show that the conditions on the stores of $S'_I$ and $S'_M$ are satisfied. If the applied rule was one of S-ITE-T, S-ITE-F, S-LOOP-T, S-LOOP-F, this is immediate, as the stores are identical before and after the steps.

For S-ASSIGN there are two cases to consider. If $v$ is not of type $S_X$ in $Q$, it is immediate by the typing rules and the generality of $Q$ that $e$ evaluates identically in both $S_I$ and $S_M$. If $v$ is of type $S_X$ in $Q$, it follows by the typing rules and associativity of $\oplus_X$ that $e$ is an expression of the form $x_0 \oplus_X \cdots \oplus_X x_k$ where for all $x_i$, with $0 \leq i \leq k$ either (a) the identity element $1_X$, (b) a variable or (c) an array element. For each $x_i = A[e'_i]$, the expression $e'_i$ evaluates identically in both $S_I$ and $S_M$ by the typing rules and generality of $Q$. The result is now immediate by the definition of $\oplus_X$ and the fact that (a) for each $x_i = 1_X$ we have $1_X \in Reify_M(1, \sigma_A)$, (b) for each variable $v$ among $x_i$ we have by assumption that $\sigma'_A(v) \in Reify_M(\sigma(v), \sigma_A(\text{in}))$, and (c) for each $A[e'_i]$ among $x_i$ we have $\sigma'_A(A)[\{e'_i \sigma_A(\text{in})\}] \in Reify_M(\sigma(A)(\{e'_i \sigma_A(\text{in})\}))$, where $(\sigma_A, \sigma_A)$ are the stores of $S_I$ and $(\sigma'_A, \sigma'_A)$ those of $S'_M$.

For S-ARRAY observe that $e_1$ evaluates identically in both $S_I$ and $S_M$ by the typing rules and generality of $Q$. Hence, the same array element is updated in each application of the rule. There are now two cases to consider, i.e., $A$ is or is not of type Array($S_X$) in $Q$. These cases are identical to those of S-ASSIGN.

\[ \square \]

Lemma 4.7. Let $P$ be a generic program, $M$ a monoid, and $n$ a natural number. If $S_M$ is an initial state of $P[M]$, then there exists an initial state $S_I$ of $P[I]$ satisfying the singleton condition for $n$ such that $S_M \in Reify_M(S_I)$. Then:

\[ \iff \]

\[ \text{if } S_I \rightarrow_s S'_I, \text{there exists a program state } S'_M \in Reify_M(S'_I) \]

\[ \text{of } P[M] \text{ such that } S_M \rightarrow_S S'_M \]

\[ \text{and } \]

\[ \text{if } S_M \rightarrow_s S'_M, \text{there exists a program state } S'_I \text{ of } P[I] \text{ such that } S_I \rightarrow_s S'_I \text{ and } S_M \in Reify_M(S'_I). \]

Proof. Let $S_M$ be an initial state of $P[M]$. The lemma is immediate by induction on the number of steps applying Lemma 4.6 once we show that an initial state $S_I$ of $P[I]$ exists that satisfies the singleton condition for $n$ such that $S_M \in Reify_M(S_I)$. To this end, write $S_M$ as $(\sigma'_A, \sigma_A, P[M])$ and define $S_I$ as $(\sigma, \sigma_A, P[I])$ with:

\[ \sigma(v) = \begin{cases} \sigma'_A(v) & \text{if } T \neq S_X \\ T & \text{if } T = S_X \end{cases} \]

for all $v$ of type $T$ in $P$.


for all $A$ of type $\text{Array}(T)$ in $P$ and $k \in \mathbb{N}$

\[
\sigma_A(A)(k) = \begin{cases} 
(k, k) & \text{if } A = \text{in} \text{ and } k \in \{0, \ldots, n-1\} \\
\top & \text{if } A = \text{in} \text{ and } k \notin \{0, \ldots, n-1\} \\
\top & \text{if } A \neq \text{in} \text{ and } T = \mathbb{S}_X
\end{cases}
\]

That $S_I$ satisfies the singleton condition for $n$ and that we have $S_M \in \text{Reify}_M(S_I)$ is now immediate. \hfill \square

**Proof (Theorem 4.5).** The $\Leftarrow$-direction is trivial, as $I$ is a monoid. For the $\Rightarrow$-direction, let $M$ be a monoid and $S_M$ an initial state of $P[M]$. By Lemma 4.7 there exists an initial state $S_I$ of $P[I]$ such that $S_I$ satisfies the singleton condition and $S_M \in \text{Reify}(S_I)$. All executions starting from $S_I$ are terminating because $P[I]$ implements an $I$-prefix sum. By Lemma 4.7, for every terminating execution (and thus every execution) $S_I \rightarrow^* S_I'$ there exists a corresponding terminating execution $S_M \rightarrow^* S_M'$ such that $S_M \in \text{Reify}_M(S_I')$. Moreover, these are the only executions of $S_I$: if there would exist a non-terminating or another terminating execution starting from $S_M$, then the induction argument from Lemma 4.7 would yield an additional execution starting from $S_I$. Because $P[I]$ computes an $I$-prefix sum, we have for the array store $\sigma_A$ of $S_I'$ that $\sigma_A(\text{out})(k) = \bigoplus_{i \leq k} \sigma_A(\text{in})(i) = (0, k)$ for all $k \in \{0, \ldots, n-1\}$. Hence, by definition of reification for the array store $\sigma_A'$ of $S_M'$ we have that $\sigma_A'(\text{out})(k) = \bigoplus_{M, i \leq k} \sigma_A'(\text{in})(i)$ for all $k \in \{0, \ldots, n-1\}$. \hfill \square

5. Extension to Data-Parallel Programs

We next consider data-parallel programs, or kernels, in which threads synchronise by means of barriers.

**Syntax, typing and semantics** We extend the language of Figure 4 with a barrier synchronisation statement as follows:

\[
\text{stms} ::= \cdots \mid \text{barrier}
\]

The typing rule for barrier is straightforward:

\[
\Gamma \vdash \text{barrier} : \text{Unit} \quad \text{(T-BARRIER)}
\]

The typing rules for expressions and all other statements are as before (see Figure 5).

Given a finite set of thread identifiers $D \subset \mathbb{N}$, a kernel (program) state is a tuple $(\sigma_A, K)$ with $\sigma_A$ an array store and $K$ a map from $D$ to (variable store, sequence of statements)-pairs such that for each $i \in D$ and variable store $\sigma_i$, $K(i)$ is a statement. The array store $\sigma_A$ represents arrays that are shared among all threads, and $K$ represents the local variables and instruction sequence for each thread. The variable $tid$ represents the identity of a thread, and must occur read-only in every program $P$. Our theoretical presentation does not depend on the existence of $tid$, but the prefix sum implementations we evaluate in Section 7 rely on each thread having a unique identifier.

Figure 7 gives the operational semantics of our kernel programming language, where $\rightarrow_\tau$ is as defined in Figure 6. The semantics is a standard interleaving semantics (see rule K-STEP) with an additional rule for barrier synchronisation (rule K-BARRIER). Rule K-BARRIER checks whether all threads are either at a barrier or have terminated; the additional condition that at least one thread must be at a barrier ensures that rule K-BARRIER and termination are mutually exclusive.

The semantics assumes that statements are executed atomically, so that for example a thread can execute a shared state update such as $A[i] := A[j] + 1$ in a single step. This assumption is not valid in practice: such a statement would involve issuing separate (and possibly multiple) load and store instructions between which other threads could interleave. Even if we refined the semantics to reflect this, we would still need to account for the weak memory models of modern architectures. However, if a program is free from data races, which we define formally below, the effects of this assumption are not visible. The verification technique for GPU implementations of parallel prefix sums, which we present in Section 6, depends on proving that a GPU kernel is free from data races.

Our rules for barrier synchronisation follow the MPI programming model [14] in which it is valid for parallel processes to synchronise at syntactically distinct barriers. In GPU programming models, such as OpenCL and CUDA the rules for barrier synchronisation are stricter, requiring that all threads synchronise at the same barrier, and that if a barrier is inside a loop all threads must have executed the same number of loop iterations on reaching the barrier [19, 27]; precise semantics for barrier synchronisation in GPU kernels have been formally specified [1, 10]. Our results for the looser barrier synchronisation model of MPI makes our technique more widely applicable. Adding stricter conditions for barrier synchronisation does not affect our theoretical results, and the GPUVerify tool used as part of our verification method checks the stricter conditions required in the GPU setting.

Given a kernel program $P$, an initial state of $P$ is any kernel state $(\sigma_A, K)$ where, for every thread $t$, the sequence of statements of $K(t)$ is $P$. An execution of a program $P$ starts from an initial state. Maximal execution and termination for an initial state are defined as for sequential programs. Note that the interleaving nature of the semantics means that there may be multiple maximal executions, contrary to the sequential case.

**Prefix sums and generic prefix sums** Computation and implementation of a prefix sum is defined as in the sequential case (see Definition 3.1) with $P$ interpreted as a kernel program. A generic kernel program and generic prefix sum are also defined as in the sequential case (Definition 4.1).

**Interval monoid** Employing the interval monoid as is (Definition 4.3), we extend the definition of the singleton condition to kernel programs, taking into account that there is now a variable store per thread.

**Definition 5.1** (Singleton condition for kernel programs). Let $P$ be a generic program with $\text{in}$ of type $\text{Array}(X)$. An initial state $(\sigma_A, K)$ of $P[I]$ satisfies the singleton condition for $n$ if

1. for all $t \in D$ and $v$ of type $\mathbb{S}_X$ in $P$, $\sigma_A(v) = \top$ with $\sigma_A$ the variable store of $K(t)$;
2. condition 2 of Definition 4.4 is satisfied

**Soundness and completeness** Our soundness and completeness theorem for sequential programs, Theorem 4.5, can now be stated identically for kernel programs. The proof strategy is the same, using reification of abstract states to concrete states to set up a simulation, using lemmas analogous to Lemma 4.6 and Lemma 4.7. We explain the notable differences in how the proof is set up.

The Reify$_M$ function is adapted to map states of kernel programs $P[I]$ to sets of states of kernel programs $P[M]$. The function treats the array store as before, but must take into account the fact that each thread now has its own variable store and program. Thus, while reification of array stores still operates at the level of the whole program state, reification of the variable store and program component now operates at the level of each individual thread. The definition is left unchanged otherwise.

The proof of Lemma 4.6 translates to the case of kernel programs by observing that

\[\text{barrier}[M] \equiv \text{barrier}\]

A Sound and Complete Abstraction for Reasoning about Parallel Prefix Sums

7
and that by this algebraic law and the algebraic laws from the proof of Lemma 4.6 we have that the same rule will be applied in both the case of $S_I$ and $S_{1I}$, where the proof of Lemma 4.6 is embedded in the treatment of the K-STEP case.

In proving Lemma 4.7 for kernel programs, we must relate an initial state $S_M = (\sigma_A, K')$ of $P[M]$ to an initial state $S_I$ of $P[I]$ that satisfies the singleton condition. The proof is identical to that of Lemma 4.7. The definition of the initial state $S_I$ is easily adapted from the definition in the proof of Lemma 4.7 by applying the definition for the variable store to the variable store of each individual thread; the definition is left unchanged otherwise.

**Data race freedom** We now define what it means for a kernel program to exhibit a data race. We show that if a kernel program is data race-free then, due to the properties of barrier synchronisation, the program computes a deterministic result. The verification technique we present in Section 6 depends on this guarantee of determinism and, as discussed above, establishing race-freeness of a kernel avoids the need to reason about weak memory semantics or the granularity of memory accesses.

Say that we read from an array element $A[i]$ in an execution step if $\sigma_A(A(i))$ is referenced during the evaluation of any of the expressions occurring in the step. Likewise, say that we write to an array element $A[i]$ in an execution step if $\sigma_A(A(i))$ is updated in the step. An array element $A[i]$ is accessed if it is either read from or written to in the execution step. Observe that an array element can be accessed multiple times in a single execution step, e.g., when evaluating $A[tid] := A[tid] \oplus_M 1_M$.

**Definition 5.2.** Let $S_0 \rightarrow^*_k S_n$ be an execution of a kernel program $P$. The execution is said to have a data race if there are execution steps $S_i \rightarrow_k S_{i+1}$ and $S_j \rightarrow_k S_{j+1}$ along the execution such that:

- distinct threads are responsible for these steps,
- a common array element is accessed in both execution steps,
- at least one of the accesses writes to the array element,
- no application of K-BARRIER occurs in between the accesses.

A program $P$ is data race-free if for every initial state $S_0$ of $P$ and execution starting from $S_0$ it holds that the execution does not have a data race.

**Theorem 5.3.** Let $P$ be a generic kernel program and let $M$ be a monoid. Then: $P[M]$ is data race-free $\iff$ $P[M']$ is data race-free for all monoids $M'$.

**Proof:** The direction is trivial, as $M$ is a monoid. For the $\Rightarrow$ direction, observe that neither the control-flow nor the array accesses can be influenced by the choice of $M'$ by the typing rules and genericity of $P$.

We now argue that data race-free kernel programs behave deterministically. Say that a barrier synchronisation occurs during execution of a program whenever the K-BARRIER rule of Figure 7 applies. Suppose that a kernel program is race-free, and consider executing the program from an initial state $S$. Race-freedom means that the execution of one thread cannot depend upon the actions of another until all threads participate in a barrier synchronisation. Thus while the threads may interleave nondeterministically, their individual execution is deterministic and when the first barrier synchronisation occurs (assuming all threads reach a barrier without diverging), the program state $S_I$ is deterministic. By a similar argument, individual thread execution until the next barrier synchronisation is deterministic, leading to a deterministic program state $S_2$ when the second barrier synchronisation occurs. Applying this argument and using induction on program executions, we can prove the following:

**Lemma 5.4.** Let $P$ be a data race-free kernel program and let $S$ be an initial state of $P$. If there exists a finite, maximal execution starting from $S$ with final array store $\sigma_A$, then all executions starting from $S$ are finite and for all maximal executions the final array store is $\sigma_A$.

6. An Automated Verification Technique

Our theoretical results show that to verify functional correctness of a generic parallel prefix sum of length $n$, implemented as a barrier-synchronising program, it suffices to prove that the program is free from data races and then test that the program behaves correctly with respect to the interval monoid for every initial state with $\mathbb{I} = \{0, 0\}, \{1, 1\}, \ldots, \{n-1, n-1\}$. In practice, a library routine for computing a prefix sum takes no input except for $\mathbb{I}$ so that there is just a single input to test.

GPU kernels are barrier synchronising programs that are required to be data race-free: the semantics of an OpenCL or CUDA kernel is undefined if the kernel exhibits data races [19, 27]. Thus our results suggest the following method for verifying functional correctness of GPU kernels that claim to implement generic prefix sums with respect to a single input array:

1. Prove that the GPU kernel is data race-free.
2. Run the interval monoid test to check functional correctness.

Steps 1 and 2 can be conducted independently, but the functional correctness guarantee provided by step 2 is conditional on the result of step 1. We now discuss how we have implemented a practical verification method for generic prefix sums written in OpenCL, based on this approach.

**Representing a generic prefix sum** The OpenCL programming model does not support generic functions directly. However, it is easy to describe a generic prefix sum by writing a kernel that uses a symbol TYPE and a macro OPERATOR as placeholders for the concrete type and operator with respect to which the prefix sum should be executed. A concrete choice of type and operator can be chosen by including a header file that specifies TYPE and OPERATOR.

**Race analysis** Step 1 above can be discharged to any sound verifier for GPU kernels capable of proving race-freedom. For GPU kernels this includes the PUG [24] and GPUVerify [1] tools, which rely on SMT solvers, and the tool described in [23] which performs verification using a combination of dynamic and static analysis. In
our experiments (Section 7) we use the GPUVerify tool because it is actively maintained and publicly available, and is the only tool supporting the multi-platform OpenCL programming model, which allows us to evaluate step 2 above on a number of platforms.

By Theorem 5.3, we can prove race-freedom of a generic prefix sum using any choice of \textsc{type} and \textsc{operator}. In our experiments we use the interval monoid.

**Running the interval monoid test case**  Step 2 above requires an encoding of the interval monoid. Observe that to check a prefix sum of length \( n \) we need only encode elements \((i, j)\) of the interval monoid for \( 0 \leq i \leq j < n \), as well as the \( \top \) and \( \bot \) elements. An element can thus be encoded using \( O(\lg(n)) \) bits, meaning that \( O(n \lg(n)) \) space is required for the test input and result. With this encoding, the interval monoid test case can be executed as a regular OpenCL kernel.

**Soundness, completeness and automation**  Our verification strategy is sound due to the guarantees provided by the version of Theorem 4.5 for kernel programs, together with Theorem 5.3 and Lemma 5.4, and the requirement that a sound method for verifying race-freedom is used. As with any dynamic technique, soundness of the testing phase of our approach depends on the probity of the compiler, driver and hardware implementation of the architecture on which the test is executed. In our experiments we guard against this source of unsoundness by testing with respect to multiple, diverse platforms.

Our verification strategy is only complete if the technique used to prove race freedom is complete. The GPUVerify method which we employ is not complete: the technique uses both thread abstraction and loop invariant abstraction, which can lead to false positive data race reports.

A small amount of manual effort is required in order to prove race-freedom of prefix sum kernels: GPUVerify tries to perform automatic inference of invariants required to prove race-freedom, but often this inference is not strong enough and manual invariants must be supplied by the programmer. However, these invariants are checked by GPUVerify—they are not taken on trust by the tool. In the examples we consider in Section 7 we disabled automatic invariant inference and supplied a small number of relatively simple, non-quantified loop invariants by hand. The dynamic analysis phase of our verification method is fully automatic.

**Handling extended programming language features**  Our sequential language and its data-parallel extension omit real-world language features such as procedures, unstructured control flow, and pointers (with associated aliasing issues). We are confident that our results apply directly in this extended setting under the condition that data of generic type \( S_X \) are never accessed via pointers with different element types. This is important for the testing part of our approach; without this restriction it would be possible to pass our test case by casting the array to a char pointer and writing the desired final result for the interval monoid byte-by-byte. Proving race-freedom in the presence of intricate pointer manipulation can be challenging, but is possible with the right invariants and is supported by GPUVerify. Because GPUVerify operates at the level of control flow graphs using the techniques of [10], unstructured control flow is also supported.

Nevertheless, the prefix sum implementations we have seen in practice do not manipulate pointers in complex ways, call auxiliary procedures or exhibit unstructured control flow.

As discussed in Section 3, the requirement that the input array be read-only is for ease of presentation only and can be dropped in practice, allowing our technique to be used in verifying prefix sums that operate in place.

### Table 2. Characteristics of the prefix sum algorithms considered in this paper where \( n \) is the number of elements

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Depth</th>
<th>Size</th>
<th>Fanout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>( n )</td>
<td>( n )</td>
<td>2</td>
</tr>
<tr>
<td>Kogge-Stone</td>
<td>( \lg n )</td>
<td>( n \lg n - (n - 1) )</td>
<td>2</td>
</tr>
<tr>
<td>Sklansky</td>
<td>( \lg n )</td>
<td>( (n/2) \lg n )</td>
<td>( n )</td>
</tr>
<tr>
<td>Brent-Kung</td>
<td>( (2 \lg n - 1) )</td>
<td>( 2n - \lg n - 2 )</td>
<td>2</td>
</tr>
<tr>
<td>Blelloch</td>
<td>( 2 \lg n )</td>
<td>( 2(n - 1) )</td>
<td>2</td>
</tr>
</tbody>
</table>

### 7. Experimental Evaluation

We demonstrate the effectiveness of our method for prefix sum verification in practice by applying it to four different algorithms implemented as OpenCL kernels.

All materials required to reproduce our experimental results, including implementations all kernels, are available online.\(^5\)

**Details of prefix sum kernels**  We evaluate our technique using four different, well-known prefix sum algorithms: Kogge-Stone [21], Sklansky [34], Brent-Kung [5] and Blelloch [4]. The Blelloch algorithm is an exclusive prefix sum.

Figure 8 provides a circuit diagram description for each algorithm. There is a wire for each input and data flows top-down through the circuit. 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). Table 2 gives a comparison of these algorithms, as well as the straightforward sequential implementation, in terms of circuit characteristics: \( \text{depth} \), the number of levels; \( \text{size} \), the number of nodes; and \( \text{fanout} \), the maximum number of outbound wires per node, as the size of the input (or width) \( n \) of the circuit varies. The depth of a circuit indicates how long it takes for the result to propagate through the circuit, and thus dictates the time taken for prefix sum computation. Size indicates how many gates would be required to build the circuit in hardware, while fanout indicates the required driving capacitance of the output gate. A software implementation of a prefix sum is \textit{work efficient} if the number of operations required is within the same order of magnitude as the sequential version. The size column of Figure 2 indicates that Blelloch and Brent-Kung are work-efficient, while Kogge-Stone and Sklansky are not. A more detailed discussion of these characteristics can be found in [16].

Through a survey of several GPU code repositories (the AMD APP SDK,\(^6\) the NVIDIA CUDA SDK,\(^7\) and the SHOC [12], Rodinia [7] and Parboil [37] benchmarks) we found Kogge-Stone to be the most widely used GPU implementation of a prefix sum in practice, Blelloch to be used where an exclusive prefix sum is required, and Brent-Kung employed several times in one large CUDA kernel that computes eigenvalues of a matrix. Sklansky is the oldest prefix sum algorithm we consider; we did not see it used in practice.

**Experimental setup**  As discussed in Section 6, we use the GPUVerify tool [1] to prove that each kernel is free from data races. These experiments were performed on a Linux machine with a 1.15 GHz AMD Phenom 9600B 4 core processor, using a version of GPUVerify downloaded from the tool web page\(^8\) on 26 June 2013.

We then verify functional correctness for each kernel by running the interval of summations test case. We give results for five different platforms: two NVIDIA GPUs – a 1.16 GHz GeForce GTX 570

\(^{5}\)http://multicore.doc.ic.ac.uk/tools/GPUVerify/
\(^{7}\)https://developer.nvidia.com/gpu-computing-sdk
\(^{8}\)http://multicore.doc.ic.ac.uk/tools/GPUVerify/
and a 1.15 GHz Tesla M2050; two Intel CPUs – a 2.13 GHz 4 core Xeon E5606 and a 2.67 GHz 6 core Xeon X5650; and one ARM GPU – a 533 MHz 4 core Mali T604. These devices exhibit a range of power and performance characteristics. The NVIDIA GPUs offer a large number of parallel processing elements running at a relatively low clock-rate; the Intel CPUs run at a higher clock-rate but exhibit less parallelism; and the ARM GPU is designed for high power-efficiency. We chose to run on multiple platforms to guard against possible unsound results as a result of a particular compiler, driver or hardware configuration. Each experiment was run with a timeout of 1 hour. All the timing results we present are averages over three runs.

In [38], Voigtländer shows for sequential programs that a generic prefix sum is correct for all input lengths if it can be shown to behave correctly for all input lengths with respect to two binary operators over a set of three elements. He shows further that it is sufficient to consider \( O(n^2) \) test inputs: \( n(n+1)/2 \) tests using the first operator and \( n-1 \) tests using the second operator. The result of [38] considers all possible \( n \), but also restricts so that a prefix sum of a specific length \( n \) can be shown correct by testing the corresponding set of inputs. Our Lemma 5.4 allows us to lift this method to race-free barrier-synchronising programs. Thus, by way of comparison, we also tried dynamically verifying the prefix sum kernels by running, for each element size, the set of Voigtländer tests. We discuss Voigtländer’s paper further in the related work (Section 8).

Verification using the interval of summations test case and the Voigtländer tests both require race-freedom to be established, thus the overhead of race checking applies to both approaches.

**Results for proving race-freedom** Table 3 presents verification times, in seconds, for proving race-freedom of each of our prefix sum kernels using GPUVerify, for a selection of power-of-two element counts up to \( 2^{31} \). Times for the power-of-two thread counts not shown are in the same order of magnitude.

The results show that verification time is more-or-less independent of the number of threads executing the kernel. This is because GPUVerify checks race-freedom for a kernel with respect to an arbitrary pair of threads, and reasons about loops using invariants rather than by unrolling. This excellent scalability demonstrates that the data race analysis aspect of our verification method is practical for very large arrays.

As noted in Section 6, we disabled automatic invariant inference in GPUVerify as it did not allow fully automatic verification of our examples. Instead we provided relatively simple, non-quantified loop invariants for each of our kernels in order for verification to succeed. For example, for the loop of the Kogge-Stone kernel shown in Figure 2 we provided the following invariant (no further invariants were necessary):

\[
\text{no_read(out)} \land \text{no_write(out)}
\]

which specifies that no access to the out array is in-flight at the head of the loop. The most intricate invariant we had to specify, for the Blelloch kernel, was:

\[
(d \& (d - 1)) = 0 \land ((\text{offset} = 0 \land d = N) \lor d + \text{offset} = N).
\]

Here, \( d \) and offset are loop counters, with \( d \) increasing in powers-of-two from 1 to \( N \), and offset decreasing in powers-of-two from \( N \) to 1, and finally reaching zero. The conjunct \( (d \& (d - 1)) = 0 \) uses the bitwise-and operator, \&, to compactly express that \( d \) must be a power-of-two or zero.

Counting each top-level conjunct of a loop invariant as a separate invariant, the number of invariants required for verification of each kernel was as follows: (Kogge-Stone, 2); (Skłansky, 1); (Brent-Kung, 4); (Blelloch, 6). The invariants did not have to be tailored for individual thread counts.

**Results for dynamic analysis** The I rows of Table 4 present, for each prefix sum and platform, the time in seconds taken to run the single interval of summations test for power-of-two array sizes from \( 2^9 \) to \( 2^{14} \) and \( 2^{18} \) to \( 2^{20} \) (results in between follow a similar trend). For each array size we also show in the \#Voigtländer tests row the total number of tests that would need to be run to verify the kernel using Voigtländer’s method and the V rows show how long these tests took to run. The times are end-to-end, taking into account device initialisation, creation of memory buffers, compilation of the OpenCL kernel,\(^9\) copying data into memory buffers, running the test and validating the result. For the Voigtländer tests we used a wrapper so that device initialisation, creation of memory buffers and kernel compilation is performed only once for each array size. We note that while in theory an array of length \( n \) can be processed by \( n/2 \) concurrent threads (or \( n \) threads in the case of Kogge-Stone), it is up to the OpenCL runtime to decide how to schedule threads across hardware resources, and that in practice a large number of threads will be launched in a series of waves.

From the I rows it can be seen that running the interval of summations test case is very fast on all platforms, even for arrays of length \( 2^{20} \). It may seem surprising that the runtime does not increase significantly as the array size increases. This is because (a) computing with the interval monoid is very cheap, requiring only integer equality testing and addition by a constant and (b) the expense of online compilation of the OpenCL kernel, which dominates the runtime of these simple tests.

The V rows show that, unsurprisingly, running a quadratic number of tests per array size does not scale well. On all platforms, we found that sizes of \( n \geq 2^{14} \) exhausted our time limit of 1 hour, showing that the Voigtländer method will not scale to large arrays.

**8. Related Work**

**Relationship to results by Voigtländer and Sheeran** The closest work to this paper is work by Voigtländer [38] which proves two interesting results for sequential generic prefix sums.

---

\(^9\) In the OpenCL model, a kernel is compiled at runtime using an online compiler, so that it need not be pre-compiled for any given device [19].
Kogge-Stone 5.7 5.8 5.9 5.8 5.8 6.0 5.6 5.8 5.7 5.6 6.4 6.4 6.6
Sklansky 5.6 6.1 6.5 6.8 6.6 6.7 6.8 6.7 6.5 6.4 7.3 8.1 7.9
Brent-Kung 5.9 6.7 7.4 12.3 11.4 12.9 10.2 10.3 10.7 12.6 14.9 11.1 16.4
Blelloch 6.0 7.4 9.0 12.4 12.6 13.8 12.0 14.1 11.6 12.3 15.5 13.0 14.8

Table 3. Time (in seconds) taken to prove race freedom for prefix sum kernels, for increasing array lengths. An array of length \( n \) is processed by \( n/2 \) threads, except in the case of Kogge-Stone where \( n \) threads are used.

Kogge-Stone V 19.2 76.2 337.6 1463.9 - - - - -
Skansky V 18.5 71.8 320.2 1438.3 - - - - -
Brent-Kung V 19.0 69.3 317.3 1454.3 - - - - -
Blelloch V 19.2 75.4 324.2 1595.3 - - - - -

NVIDIA GeForce GTX 570

Kogge-Stone V 14.6 36.1 160.9 653.0 2796.8 - - - - -
Skansky V 12.2 32.5 149.6 609.3 2601.5 - - - - -
Brent-Kung V 10.9 35.9 165.7 679.9 2931.8 - - - - -
Blelloch V 10.9 36.7 166.0 679.9 2931.8 - - - - -

NVIDIA Tesla M2050

Kogge-Stone V 21.1 109.8 660.8 3467.1 - - - - -
Skansky V 12.2 78.4 404.8 2003.9 - - - - -
Brent-Kung V 12.6 80.4 469.9 2272.5 - - - - -
Blelloch V 12.9 80.7 472.1 2332.5 - - - - -

Intel Xeon X5650

Kogge-Stone V 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3
Skansky V 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3
Brent-Kung V 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3
Blelloch V 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3 1.3

Intel Xeon E5606

Kogge-Stone V 1.9 1.9 2.1 2.1 2.0 2.1 2.1 2.4 2.7 3.2 - - -
Skansky V 1.9 1.9 2.1 2.1 2.0 2.1 2.1 2.2 2.3 2.4 - - -
Brent-Kung V 2.0 2.0 2.1 2.1 2.1 2.1 2.3 2.3 2.5 - - -
Blelloch V 2.0 2.0 2.1 2.1 2.1 2.1 2.4 2.4 2.6 - - -

ARM Mali T604

Kogge-Stone V 287.0 1166.1 - - - - - - - - - - -
Skansky V 287.2 1147.6 - - - - - - - - - - -
Brent-Kung V 287.4 1108.1 - - - - - - - - - - -
Blelloch V 277.0 1105.3 - - - - - - - - - - -

Table 4. Time (in seconds) taken to establish correctness of prefix sum implementations, for increasing array lengths, on NVIDIA GPU, Intel CPU and ARM GPU architectures. The number of test cases associated with Voigtländer’s method for each array length is also shown.

First Voigtländer shows, using relational parametricity, that a generic prefix sum is correct if and only if it behaves correctly for each length \( n \) with respect to integer lists and list concatenation when applied to the input \([0], [1], \ldots, [n - 1]\), that is, it yields the output \([[0], [0], 1], \ldots, [0, \ldots, n - 1]]\). Voigtländer states that this result was inspired by earlier unpublished work by Sheeran; Sheeran confirms this claim when exploiting the result in later work on the design of prefix sum algorithms [33]. We refer to this as the Sheeran result. The result also holds for fixed lengths \( n \), which means the result is similar to our result for sequential prefix sums.
with respect to the interval monoid. However, the Sheeran result cannot be used for practical verification of prefix sums for large \( n \), because \( O(n^2 \log(n)) \) space is required to represent the result.\(^{10}\) Our interval of summations monoid avoids this problem by exploiting the fact that, as argued in Section 4, a correct generic prefix sum should never compute a non-contiguous summation. The interval of summations abstraction thus allows a contiguous summation to be represented precisely and compactly as an interval \((i, j)\) (or as \(1_j\) in the case of an empty summation), and collapses all other summations into the absorbing element \(\top\).

Letting \( L \) denote the monoid of integer lists under list concatenation employed by the Sheeran method, there is an obvious homomorphism \( \theta : L \to I \) defined by:

\[
\theta([x_1, \ldots, x_m]) = \begin{cases} 
1_I & \text{if } m = 0 \\
(x_1, x_m) & \text{if } m > 0 \text{ and } x_{i+1} = x_i + 1 \ (1 \leq i < m) \\
\top & \text{otherwise}
\end{cases}
\]

This homomorphism discards exactly those summations that cannot contribute to a correct generic prefix sum.

Secondly, Voigtländer presents an elegant proof of the “0-1-2-Principle” for parallel prefix computation. In the spirit of the 0-1-Principle of Knuth [20],\(^{11}\) the 0-1-2-Principle states that a prefix sum algorithm is correct on all ternary sequences with every associative operator (defined over the ternary set) if and only if it is correct on any input sequence with an associative operator. In fact, carefully tracing his proof, Voigtländer observes that it suffices to consider just two operators and for every length \( n \) a particular input set of ternary sequences of size \( O(n^2) \) and \( O(n) \) for his first and second operator, respectively. As with the Sheeran result, this result also holds for fixed lengths \( n \). Voigtländer does not present an experimental evaluation of his method in [38]; we believe that our work is the first to do so. As we demonstrated in the experiments of Section 7, verifying a prefix sum by running Voigtländer’s set of tests does not scale to large array sizes.

The focus of [33, 38] is on sequential HASKELL programs describing parallel synchronous hardware. In contrast, we have presented a method for verifying prefix sums implemented as asynchronous barrier-synchronising programs, leading to a practical method for verifying GPU kernel implementations. Due to our imperative, data-parallel setting, we present proofs using a direct simulation argument; we are not aware of any work on using relational parametricity to reason about data-parallel programs and believe this would be challenging.

**Correct-by-derivation prefix sums** An alternative approach by Hinze [17] is to construct prefix sums that are correct by derivation. In Hinze’s work, rather than verifying candidate implementations for functional correctness, new prefix sum circuits are built from a set of basic primitives and combinators. The correctness of these circuits is ensured by a set of algebraic laws derived for these combinators. Similar to the work of Sheeran, prefix sums are given as HASKELL programs describing circuit layouts. We are not aware of any work that translates this approach to a data-parallel setting.

**Formal verification of GPU kernels** Recently a number of techniques for formal or semi-formal analysis of GPU kernels have been proposed [1, 9, 18, 23–25], and formal semantics for GPU kernels have been studied in-depth [10, 15]. The PUG [24] and GPU-Verify [1] methods aim to statically prove race-freedom, while the method described in [23] uses a combination of dynamic and static analysis for race-freedom verification. We employed GPUVerify to prove race-freedom of prefix sum kernels in our experimental evaluation (Section 7). The GKLEE [25] and KLEE-CL [9] methods employ dynamic symbolic execution, based on the KLEE tool [6], to find data races and array bounds errors in CUDA and OpenCL kernels, respectively.

A recent technique for functional verification of OpenCL kernels using permission-based separation logic [18] could, in principle, be applied to generic prefix sum kernels. However, tool support for automation of this method is not yet available.

In prior work we introduced barrier invariants [8] to enable reasoning about data-dependent GPU kernels, and used barrier invariants to statically prove functional properties of Bledloch, Brent-Kung and Kogge-Stone prefix sum kernels. The challenges in making this approach scale provided insights which ultimately led to the interval of summations abstraction, and we used an early formulation of the abstraction to verify the Kogge-Stone kernel in [8]. The method for prefix sum verification presented here significantly outperforms the technique of [8], but the concept of barrier invariants has wider applicability.

9. Conclusions

We have introduced the interval of summations abstraction, investigated theoretical guarantees provided by this abstraction for verifying prefix sum algorithms, and shown that it can be used to design a practical and scalable verification method for GPU kernel implementations of prefix sums. Our experimental results demonstrate that we can prove race-freedom of prefix sum implementations for all conceivably useful array sizes, after which checking functional correctness of a prefix sum boils down to running a single test case, taking no longer than the time required to run the prefix sum in practice. We believe this substantiates our claim that we have made a major step towards solving the problem of functional verification for GPU implementations of generic prefix sums.

Our approach depends on the ability to prove that a data-parallel program behaves deterministically. For models of computation where barriers provide the only means of synchronisation this boils down to proving race-freedom (Lemma 5.4). Our approach thus extends to prefix sums written in programming models such as OpenMP, MPI and pthreads as long as these implementations use barriers exclusively for synchronisation, in favour of locks or atomic operations. Correspondingly, our method cannot be applied to GPU kernel implementations where threads communicate using atomic; in practice we have not seen the use of atomic in prefix sum kernels.

We have considered verification of prefix sums with respect to fixed array sizes. To prove that a prefix sum is correct for all array sizes it would be necessary to prove race-freedom for arbitrary array sizes, and also to prove that the prefix sum would compute the correct result for our interval of summations test case for every array size. While the former is beyond the scope of current tools such as GPUVerify, and the latter obviously cannot be achieved through testing, one could attempt to establish both through (manual or partially automated) mathematical proof.

We deliberately designed the interval of summations abstraction to capture the specific case of prefix sums, due to the importance of this class of algorithms. The abstraction can be used to analyse any algorithm that operates over an abstract monoid as long as the results computed by the algorithm are required to be contiguous summations. We are aware of one further class of such algorithms: reduction operations. The reduction of an array \( [s_1, s_2, \ldots, s_m] \) is the sum \( s_1 \oplus s_2 \oplus \cdots \oplus s_m \). Indeed, a reduction is performed during the first phase of the Bledloch and Brent-Kung prefix sum algorithms.
A natural question is to ask if there are prefix sum algorithms that gain efficiency by exploiting other properties of operators, such as commutativity or idempotence. We are aware of recent work by Sergeev [32] which investigates prefix sum circuits for the XOR operator, exploiting the fact that this operator satisfies the identity $x \oplus y \oplus y = x$. Beyond prefix sums, there is scope for devising abstractions to allow representation-independent reasoning via a canonical test case in less restricted settings. For example, potentially discontiguous summations of monoid elements can be represented using lists [33, 38], and commutativity can be accommodated by switching to a bag representation. However, these representations lose the space-efficiency afforded by the interval of summations abstraction which allows our approach to scale.

Acknowledgments

We are grateful to Josh Berdine, Pantazis Deligiannnis, Samin Ishtiaq, Alan Mycroft, Jakob Grue Simonsen, John Wickerson and our POPL reviewers for their insightful comments on various drafts of this work, and to Akash Lal for helpful discussion.

We thank Daniel Liew (Imperial College), Anton Lokhmotov (ARM) and Dong Ping Zhang (AMD) for their assistance with our experimental evaluation.

Finally, we thank Mary Sheeran and Janis Voigtländer for their work, which inspired our efforts.

References


