Formal Verification of Parallel Prefix Sum

Conference Paper

Author(s):
Mohsen, Safari; Oortwijn, Wytse; Joosten, Sebastiaan J.C.; Huisman, Marieke

Publication date:
2020

Permanent link:
https://doi.org/10.3929/ethz-b-000465636

Originally published in:
Lecture Notes in Computer Science 12229, https://doi.org/10.1007/978-3-030-55754-6_10

Formal Verification of Parallel Prefix Sum

Mohsen Safari[1](✉), Wytse Oortwijn[2], Sebastiaan Joosten[1], and Marieke Huisman[1]

[1] Formal Methods and Tools, University of Twente, Enschede, The Netherlands
{m.safari,s.j.c.joosten,m.huisman}@utwente.nl
[2] Department of Computer Science, ETH Zurich, Zurich, Switzerland
woortwijn@inf.ethz.ch

Abstract. With the advent of dedicated hardware for multicore programming, parallel algorithms have become omnipresent. For example, various algorithms have been proposed for the parallel computation of a prefix sum in the literature. As the prefix sum is a basic building block for many other multicore algorithms, such as sorting, its correctness is of utmost importance. This means, the algorithm should be functionally correct, and the implementation should be thread and memory safe.

In this paper, we use deductive program verification based on permission-based separation logic, as supported by VerCors, to show correctness of the two most frequently used *parallel* in-place prefix sum algorithms for an *arbitrary array size*. Interestingly, the correctness proof for the second algorithm reuses the auxiliary lemmas that we needed to create the first proof. To the best of our knowledge, this paper is the first *tool-supported* verification of functional correctness of the two parallel in-place prefix sum algorithms which does not make any assumption about the size of the input array.

Keywords: GPU verification · Deductive verification · Separation logic

1 Introduction

With many emerging parallel computing paradigms and architectures, investigating how to parallelize algorithms to optimize performance has become an active research area. General Purpose Graphics Processing Units (GPGPUs) are a promising new parallel architecture, where many threads cooperate together, executing the same instructions, but on different data.

One of the algorithms for which several parallel (GPU-based) implementations have been proposed is the prefix sum algorithm [4,9,15,20]. It takes an array of integers and, for each element, it computes the sum of the previous elements. The prefix sum algorithm is used in many other algorithms, e.g. in radix sort, quick sort, to solve recurrences, and in tridiagonal linear systems; see Blelloch [4]. Blelloch introduced a parallel in-place prefix sum algorithm and Harris [12] adapted it for GPUs. Kogge-Stone [15] proposed a different parallel in-place

R. Lee et al. (Eds.): NFM 2020, LNCS 12229, pp. 170–186, 2020.
https://doi.org/10.1007/978-3-030-55754-6_10

prefix sum algorithm and Horn [13] adapted it for GPUs. These two parallel versions [4,15] are the most used in practice and are available as a primitive operation in many libraries (e.g., AMD APP SDK[1], NVIDIA CUDA SDK[2]).

The GPU-based implementations of these two algorithms are widely used, even as a building block for other algorithms (e.g., sorting). Therefore, the correctness of these algorithms is of utmost importance. This means that the algorithms must be memory and thread safe (i.e. free of data races), *and* that they must be functionally correct, i.e. it actually produces the result we expect. Concretely, in this case functional correctness means that the result must be the prefix sum of the input. In general, proving functional correctness of parallel programs is a difficult task. In particular, proving the functional correctness of these two parallel prefix sum algorithms is challenging for several reasons. First, both algorithms are in-place, i.e. we need to reason about values that are unstable and change during the algorithm. Second, the computational pattern of the algorithms makes it complex to reason about the final result. Therefore, it is a challenge to find suitable properties to relate the internal computation steps in the algorithms to the final result. In particular, in Blelloch's algorithm, there are two independent, but closely related phases with different computation pattern in each phase, which makes the verification harder. As a result, establishing functional correctness of the two algorithms is non-trivial.

For the verification, we use deductive verification, a static approach that does not require running the programs. Intermediate annotations are added to capture the intermediate properties of the program. Then, using a proof system, the annotated code is translated into proof obligations which are discharged to an automated theorem prover; in our case Z3.

To prove memory safety and functional correctness of two parallel prefix sum algorithms, we use VerCors [5], which is a verification tool for reasoning about the correctness of concurrent programs. First, we show how to verify the correctness of Blelloch's algorithm. An important feature of our verification is that it is a non-trivial example of how ghost code[3] helps to reason about in-place algorithms. Second, we show how we can verify a different parallel in-place prefix sum algorithm, Kogge-Stone, using the same approach as the first verification. This demonstrates that the verification setup introduced in this paper (approach, operations and lemma) is not specific to this particular case study and can be used in other verifications. To the best of our knowledge, this is the only *tool-supported* verification of data race-freedom and functional correctness of the two most used *parallel* prefix sum algorithms for any *arbitrary size of input*. Note that none of the existing other approaches to analyse GPU applications is able to verify similar properties. Most approaches are dynamic [11,17–19,21], and only aim to find bugs. Other existing static verification techniques [3,10,14,16] either require a bound on the input size, or they do not fully model all aspects of GPU

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

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

[3] Ghost code is not part of the algorithm and is used purely for verification purposes.

programming, such as the use of barriers. Furthermore, our work enables the verification of other complicated parallel algorithms, such as stream compaction and radix sort, that are built on top of the prefix sum algorithms.

Contributions. The main contributions of this paper are:

1. We show the parallel prefix sum algorithm by Blelloch is data race-free and functionally correct for any arbitrary size of input, using deductive approach.
2. We show the lemmas used to verify the first algorithm are general enough to prove data race-freedom and functional correctness of a different algorithm, Kogge-Stone, for any arbitrary size of input.

Organization. Section 2 explains the necessary background, i.e., it introduces VerCors, the two prefix sum algorithms verified in this paper, and their encoding in VerCors. Section 3 and Sect. 4 describe how to specify and verify the correctness of the prefix sum algorithms by Blelloch and Kogge-Stone, respectively. Section 5 discusses related work and Sect. 6 concludes the paper.

2 Background

This section briefly describes VerCors and explains both parallel prefix sum algorithms. In particular, it briefly discusses the VerCors verifier and its underlying logic. We describe the prefix sum problem and then we explain the parallel algorithms proposed by Blelloch and Kogge-Stone to solve this problem. In addition, we discuss the pseudocode of the algorithms as we encoded in VerCors.

2.1 VerCors

VerCors is a verifier to specify and verify (concurrent and parallel) programs written in a high-level language such as (subsets of) Java, C, OpenCL, OpenMP and PVL, where PVL is VerCors' internal language for prototyping new features. VerCors can be used to verify memory safety (e.g., race freedom) and functional correctness of programs. The program logic behind VerCors is based on permission-based separation logic [1,7]. Therefore, the programs are annotated with pre/post-conditions in permission-based separation logic [2,8]. Permissions are used to capture which memory locations may be accessed by which threads. Permissions are written as fractional values in the interval (0, 1] (cf. Boyland [8]): any fraction in the interval (0, 1) indicates a read permission, while 1 indicates a write permission. A write permission can be split into multiple read permissions and read permissions can be added up, and transformed into a write permission if they add up to 1. Blom et al. [6] show how to reason about GPU kernels including barriers. We illustrate the logic to verify a GPU kernel by an example.

List. 1. A simple annotated GPU program

```
 1  /*@ context_everywhere array != NULL && array.length == size;
 2       requires tid != 0 ==> Perm(array[tid-1], read);
 3       requires tid == 0 ==> Perm(array[size-1], read);
 4       ensures Perm(array[tid], 1);
 5       ensures tid != 0 ==> array[tid] == \old(array[tid-1]);
 6       ensures tid == 0 ==> array[tid] == \old(array[size-1]); @*/
 7  __kernel void rightRotation(int array[], int size) {
 8      int temp;
 9    int tid = get_global_id(0);  // get the index
10      if (tid != 0) { temp = array[tid-1]; } else { temp = array[size
             -1]; }
11
12    /*@ requires (tid != 0 ==> Perm(array[tid-1], read)) **
13                   (tid == 0 ==> Perm(array[size-1], read));
14         ensures Perm(array[tid], 1); @*/
15    barrier(CLK_GLOBAL_MEM_FENCE);
16    array[tid] = temp;
```

Verification Example. List 1 shows a specification of a simple kernel that rotates the elements of an array to the right[4]. To specify permissions, we use predicates $Perm(L,\pi)$ where L is a heap location and π a fractional value in the interval (0, 1][5]. Preconditions and postconditions, keywords 'requires' and 'ensures', respectively (lines 2–6), should hold at the beginning and the end of the function, respectively. The keyword 'context_everywhere' is used as an invariant (line 1) that must hold throughout the function. As preconditions, each thread has read permission to its left neighbor (except thread 0 which has read permission to the last index) in lines 2–3. The postconditions indicate each thread has write permission to its location (line 4) and the result of the function as right rotation of all elements (lines 5–6). Each thread first reads its left location (lines 10). Then it synchronizes in the barrier (line 15). When a thread invokes a barrier, it has to fulfill the barrier preconditions, and then it can assume the barrier postconditions. Additionally, it has to be shown that the barrier only redistributes the resources that are available by the threads upon entering the barrier. In this case, each thread gives up read permission on its left location and obtains write permission on its own location (lines 12–14). After that, each thread writes the value read before to its own location (line 16). Note that, we use && for logical conjunction (line 1) and $**$ as separating conjunction in separation logic (lines 12–13). Moreover, the keyword \old is used for an expression to refer to the value of that expression before entering a function (lines 5–6). The OpenCL example is translated into the PVL language of VerCors, using two parallel nested blocks. The outer block indicates the number of workgroups and the inner one shows the number of threads per workgroup (see [6] for more details). In this case study, we reason at the level of the PVL

[4] We assume there is one workgroup and 'size' threads inside it.

[5] The keywords 'read' and 'write' can also be used instead of fractions in VerCors.

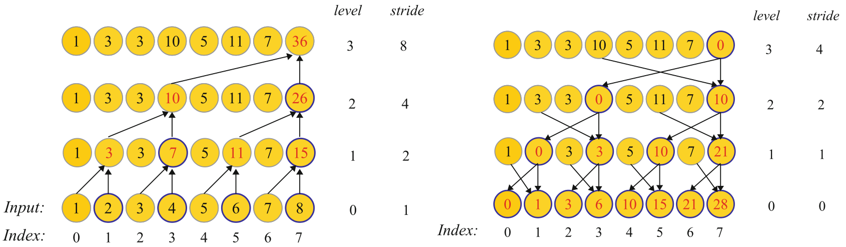


Fig. 1. After the up-sweep phase (left) and the down-sweep phase (right) in Blelloch's algorithm (two arrows coming to a circle indicates summation and one arrow indicates replacement, red color values show the effect of computations and circles with thick border are *indicators* as in Algorithm 1). (Color figure online)

encoding directly, but it is straightforward to adapt this to the verification of the OpenCL kernel.

2.2 Prefix Sum Algorithms

Given an array of integers, the prefix sum of the array is another array with the same size such that each element is the summation of all previous elements. We define an algorithm as an (inclusive) prefix sum if it satisfies the following:

- INPUT: An array *Input* of integers of size N.
- OUTPUT: An array *Output* of size N such that $Output[i] = \sum_{t=0}^{i} Input[t]$ for $0 \leq i < N$.

In the exclusive prefix sum algorithm, where the ith element is excluded from the summation, the output will be:

- OUTPUT: An array *Output* of size N such that $Output[i] = \sum_{t=0}^{i-1} Input[t]$ for $0 \leq i < N$.

Blelloch's Parallel Prefix Sum. Blelloch's algorithm [4] consists of two phases: up-sweep and down-sweep. Figure 1 illustrates both up and down-sweep phases visually, and Algorithm 1 shows the encoding of the in-place algorithm in VerCors. The up-sweep part in the figure corresponds to lines 2–8 of the algorithm and the down-sweep part corresponds to lines 12–23. Therefore, each iteration in the up/down phases in Algorithm 1 (lines 3–8/16–23) correspond to different levels in Fig. 1. We suppose that at the beginning of Algorithm 1, the input and output array have the same values. There is a variable, *stride*, which initially is 1 (line 2) and it is updated in both phases (lines 8 and 23). In the figure, the input values are at level 0 in the up-sweep phase. As we can see, in each iteration of the up-sweep, each pair is summed up at each level. As a result, the last element at the highest level is the summation of the input values. In the

Algorithm 1. Blelloch's Prefix Sum Algorithm

```
1:  function EXCLUSIVE_PREFIXSUM(int[] $Input$, int[] $Output$, int $tid$, int $N$)
2:      int $indicator = 2 \times tid + 1$; int $stride = 1$;
3:      while $stride < N$ do
4:          if $indicator < N$ && $indicator \geq stride$ then
5:              $Output[indicator] = Output[indicator] + Output[indicator - stride]$;
6:          Barrier($tid$);
7:          $indicator = 2 \times indicator + 1$;
8:          $stride = 2 \times stride$;
9:
10:     Barrier($tid$);
11:
12:     $indicator = N \times tid + N$ - $1$; $stride = N \ / \ 2$;
13:     int $temporary$;
14:     if $indicator < N$ then
15:         $Output[indicator] = 0$;
16:     while $stride \geq 1$ do
17:         if $indicator < N$ && $indicator \geq stride$ then
18:             $temporary = Output[indicator]$;
19:             $Output[indicator] = Output[indicator] + Output[indicator - stride]$;
20:             $Output[indicator - stride] = temporary$;
21:         Barrier(tid);
22:         $indicator = (indicator$ - $1) \ / \ 2$;
23:         $stride = stride \ / \ 2$;
```

down-sweep phase, we first set the last element to 0. Then, we use the partial sums calculated from the up-sweep to compute the prefix sum of the input as indicated at the lowest level in down-sweep. Note that in order to synchronize threads at each level of both phases, a barrier is needed (lines 6 and 21). There is also a barrier between up-sweep and down sweep (line 10). The main purpose of having this barrier is for a specification to redistribute the threads permissions.

Kogge-Stone's Parallel Prefix Sum. In contrast to Blelloch's algorithm, Kogge-Stone's [15] algorithm consists of one phase. Algorithm 2 illustrates the encoding and Fig. 2 illustrates the algorithm visually. The levels in the figure correspond to lines 2–11 of the algorithm. In the figure, the lowest level are the input values. As we can see, at each level, each thread (*tid*) sums up elements in locations *tid* and $tid - offset$. Since threads need current values before updating, in the algorithm, we use an auxiliary variable, *temp*, and a barrier (line 7). The threads are synchronized at each level by another barrier (line 10). As a result, at the highest level, where *offset* exceeds the length of the array, the values are the prefix sum of the values in the input array.

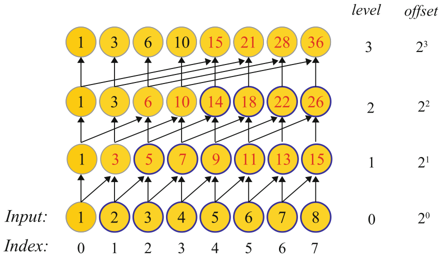


Fig. 2. Kogge-Stone's prefix sum algorithm (two arrows coming to a circle indicates summation and one arrow indicates replacement, red color values show the effect of computations and circles with thick border show *tid* ≥ *offset* as in Algorithm 2). (Color figure online)

Algorithm 2. Kogge-Stone's Prefix Sum Algorithm

```
function INCLUSIVE_PREFIXSUM(int[] $Input$, int[] $Output$, int $tid$, int $N$)
    int $offset = 1$; int $temp$;
    while $offset < N$ do
        $temp = Output[tid]$;
        if $tid \geq offset$ then
            $temp = Output[tid - offset] + temp$;
        Barrier($tid$);
        if $tid \geq offset$ then
            $Output[tid] = temp$;
        Barrier($tid$);
        $offset = 2 \times offset$;
```

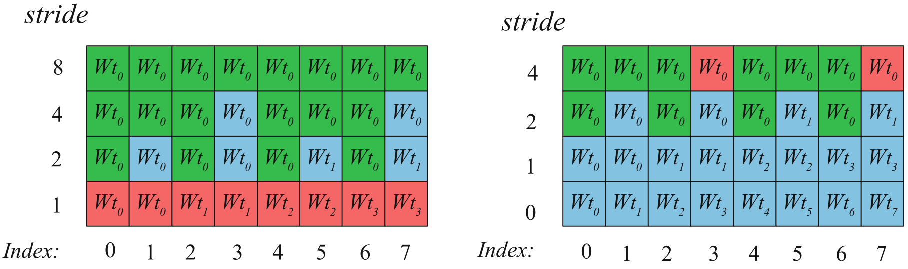
3 Verification of Blelloch's Algorithm

In this section, we explain how we verify Blelloch's parallel prefix sum algorithm. We first discuss how to prove data race-freedom and then functional correctness. Instead of presenting the full specification, we explain the main ideas and verification steps by pictures and refer to Appendix A for the crucial annotations[6].

3.1 Data Race-Freedom

To show that the algorithm is data race-free, we need to specify permissions over resources that are shared among threads. Algorithm 1 has two arrays for input and output. Thus, we specify how threads can read or write from these two arrays. In the input array, each thread (*tid*) only needs read access to location *tid*. The situation is more complicated for the output array. Figure 3 visualizes the permission scheme of threads for the output array graphically. The red elements indicate the initial permissions for both phases. In the up-sweep, each

[6] The source code is available at https://github.com/Safari1991/Prefixsum.



stride								
8	Wt_0	Wt_0	Wt_0	Wt_0	Wt_0	Wt_0	Wt_0	Wt_0
4	Wt_0	Wt_0	Wt_0	Wt_0	Wt_0	Wt_0	Wt_0	Wt_0
2	Wt_0	Wt_0	Wt_0	Wt_0	Wt_0	Wt_1	Wt_0	Wt_1
1	Wt_0	Wt_0	Wt_1	Wt_1	Wt_2	Wt_2	Wt_3	Wt_3
Index:	0	1	2	3	4	5	6	7

stride								
4	Wt_0	Wt_0	Wt_0	Wt_0	Wt_0	Wt_0	Wt_0	Wt_0
2	Wt_0	Wt_0	Wt_0	Wt_0	Wt_0	Wt_1	Wt_0	Wt_1
1	Wt_0	Wt_0	Wt_1	Wt_1	Wt_2	Wt_2	Wt_3	Wt_3
0	Wt_0	Wt_1	Wt_2	Wt_3	Wt_4	Wt_5	Wt_6	Wt_7
Index:	0	1	2	3	4	5	6	7

Fig. 3. Permission patterns for array of length 8: (left) up-sweep and (right) down sweep phases of Blelloch's algorithm (W_{t_i} indicates thread i has write permission, red color indicates initial permissions of active threads, blue shows changes in permission pattern and green shows lost permissions which assigned to thread 0. (Color figure online)

thread needs write access to $indicator$ and $indicator - stride$ (line 5 in Algorithm 1). Since initially, $indicator$ and $stride$ are $2 \times tid + 1$ and 1, respectively, we specify write access for each thread to locations $2 \times tid + 1$ and $2 \times tid$, indicated by red color in Fig. 3 (left). Then, in each iteration, $indicator$ and $stride$ are updated. Therefore, in the barrier of up-sweep (line 6), we change the permissions according to the new values of $indicator$ and $stride$, as shown in blue.

Note that, in each iteration some threads lose permissions, since $indicator$ exceeds the array length (N). According to this scheme, at the end of up-sweep, no threads have permissions left to access elements of the output array due to $indicator > N$ (blue color disappears). However, we need the same pattern of permissions in down-sweep, and in the barrier between up and down sweep (line 10), we cannot invent permissions, but we can only redistribute the current permissions. To solve this, we specify that one random thread (thread 0) collects the lost permissions in each iteration (indicated by green). As we can see, at the end of up-sweep, thread 0 has write permission to all locations in the array.

In the down-sweep phase, Fig. 3 (right), we have the same permission pattern in reverse direction. In down-sweep, thread 0 is the only one whose $indicator$ initially is in the bound of the output size (i.e, $indicator$ is $N \times tid + N - 1$). Thus, initially, thread 0 has write access to $indicator$ and $indicator - stride$ (indicated in red). Note that, at the beginning of this phase we update $stride$ to $N/2$. Thread 0 also has write permission for the rest of elements (indicated by green color), since we need the permissions to redistribute them in the barrier of down-sweep (line 21). As we can see, when we move down, the permission scheme changes according to $indicator$ and $stride$. In the end, each thread (tid) has write permission to its own location (tid) of the output array. In this way threads can safely compute the prefix sum in parallel.

3.2 Functional Correctness

To verify functional correctness, we show that at the end of this algorithm, the output array contains the prefix sum of the input array. Proving functional correctness of this algorithm is particularly challenging because:

1. The algorithm is in-place; which means the elements change in each iteration.
2. There are two phases in the algorithm, each with different computations.
3. The intermediate steps are non-trivial, and non-trivial invariants have to be proven to conclude that indeed the prefix sum is proven.

To overcome the above challenges, we keep track of the values in each iteration of the algorithm. For this history of values, we use ghost variables (i.e., for each iteration in both phases, we assign the current values of the output array to a ghost variable of type sequence). Moreover, we need to specify invariants that relate the computations in up-sweep and down-sweep. If we look at the only values that change in Fig. 1 (red-colored values), we notice that in up-sweep (left) the sum of those values equals the sum of the values in the input array in each iteration. Further, in the down-sweep (right), the red values at each level are the prefix sum of the red values at the corresponding level in the up-sweep. Therefore, our general strategy to tackle the above challenges is:

1. Define different ghost variables in both up-sweep and down-sweep to keep a history of values.
2. Define mathematical functions to update the ghost variables (according to actual computations) in each iteration of the algorithm.
3. Prove functional correctness over the ghost variables using two invariants:
   – In up-sweep, the sum of values that change in each iteration equals the sum of the values in the input array.
   – In down-sweep, the values that change at each level are the prefix sum of the values that change at the corresponding level in up-sweep.
4. Relate the ghost variables to the actual arrays; i.e., prove that the elements in the ghost variables capture the same elements as in the actual arrays.

Up-Sweep Ghost Variables. We go through the steps above to show functional correctness of the algorithm. First, in the up-sweep phase, we define two ghost variables: one to keep track of all values in each iteration as a full history (*f_hist* with type sequence of sequences), and one to keep history of the only values that change as a partial history (*p_hist* with type sequence of sequences). We define two different ghost variables because *p_hist* is used to show preservation of the above two invariants, while *f_hist* is used to prove that the ghost variable in down-sweep is capturing the elements in the output array. Initially, these two ghost variables contain the values in the input array.

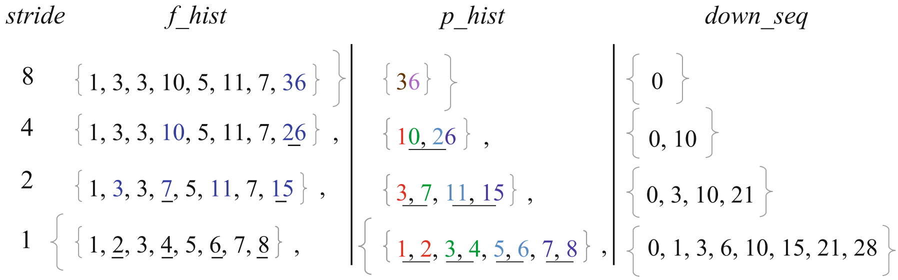


Fig. 4. Ghost variables: (left) Building *f_hist* by applying *Build_full_history* to *f_hist_prev_lvl*, blue color indicates how value changes, (middle) Building *p_hist* by applying *Build_partial_history* to *p_hist_prev_lvl*, colors show combination of each pair and (right) creating *down_seq* by applying *p_sum* to *p_hist_lvl*. (Color figure online)

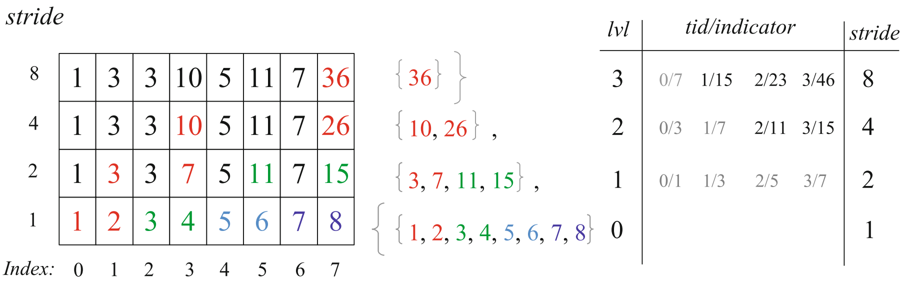
The next step is to define mathematical functions over these ghost variables to update them in the same way as the actual computations do over the actual arrays. To update *f_hist* in each iteration of up-sweep, we must add a new sequence of current values in the output array to the chain of sequences in *f_hist*. Therefore, we define a *Build_full_history* function as shown in List 2. The function takes the previous level in *f_hist*, named as *f_hist_prev_lvl*, the *stride* and an integer i. The integer i, starts from 0 and increases up to the length of *f_hist_prev_lvl*, indicates the location of elements in *f_hist_prev_lvl* to be updated. The *Build_full_history* function goes through all elements and updates the elements if the condition $(i\%(2 \times stride)) == (2 \times stride - 1)$ && $(i \geq stride)$ holds (lines 11–13), otherwise it keeps the elements unchanged (lines 14–15). Note that, this is a recursive function that captures the same computation as in the algorithm, but over the ghost variable. The postconditions (lines 2–8) specify that the result is either the sum of two elements (according to *stride*) if the condition holds (lines 3–5) or unchanged (lines 6–8) otherwise. By applying this function (to *f_hist_prev_lvl*), in each iteration of the algorithm, a full history of values is created like a matrix as sequence of sequences (Fig. 4 (left)). In the figure, the underlined elements show the locations where the condition (in *Build_full_history*) holds and the blue ones show how the values change according to *stride*.

List. 2. The *Build_full_history* function

```
/*@ requires stride > 0 && stride < |f_hist_prev_lvl|;
    ensures |\result| == |f_hist_prev_lvl|-i;
    ensures (\forall int j; j ≥0 && j <|\result|; ((i <|f_hist_prev_lvl|) &&
            ((i+j)≥stride) && (((i+j)%(2×stride)) == (2×stride-1))) ==>
            \result[j] == f_hist_prev_lvl[i+j] + f_hist_prev_lvl[i+j-stride]);
    ensures (\forall int j; j ≥0 && j <|\result|; ((i <|f_hist_prev_lvl|) &&
            (((i+j)<stride) || (((i+j)%(2×stride)) != (2×stride-1)))) ==>
            \result[j] == f_hist_prev_lvl[i+j]); @*/
static pure seq<int> Build_full_history(seq<int> f_hist_prev_lvl, int stride,
   int i) = i <|f_hist_prev_lvl| ? (
      ((i%(2×stride)) == (2×stride-1) && (i ≥ stride) ?
       seq<int> {f_hist_prev_lvl[i] + f_hist_prev_lvl[i-stride]} +
                Build_full_history(f_hist_prev_lvl, stride, i+1) :
       seq<int> {f_hist_prev_lvl[i]} +
                Build_full_history(f_hist_prev_lvl, stride, i+1) )) : seq<int> {};
```

To update *p_hist*, which keeps only the values that change during the iterations, we define a *Build_partial_history* function (see List 3). It takes the previous sequence, *p_hist_prev_lvl*, as an argument, and it creates a sequence that contains the values that changed according to the actual computation by summin up each pair of elements (lines 4–5). Note that, the function uses operations `head` and `tail`, where `head` returns the first element of a sequence and `tail` returns a new sequence by eliminating the first element. Figure 4 (middle) shows the result of applying *Build_partial_history* to *p_hist_prev_lvl*.

Down-Sweep Ghost Variables. Next, in down-sweep, we define a ghost variable, *down_seq*, as a sequence to keep the values that change only in the current iteration. In this way, we can show that the values that change in down-sweep are in fact the exclusive prefix sum of the values changed in up sweep in each iteration. To update *down_seq* in each iteration of down-sweep, we define a function, *epsum* (List 4), and we apply it to the corresponding level of *p_hist*, shown as *p_hist_lvl* in the function. The argument i is initially 0. Note that the `intsum` operation sums all elements in a sequence and `take`(`xs`, `i`), returns the i first elements of a sequence `xs`. The *epsum* function calculates the exclusive prefix sum for each element in *p_hist_lvl* and returns it as a sequence to update *down_seq*. As an example, Fig. 4 (right) shows how *down_seq* is updated in each iteration. As we can see, the elements in *down_seq* are the exclusive prefix sum of the elements in *p_hist* at each level. Hence, it is the exclusive prefix sum of the lowest level which is the input array.



stride								
8	1	3	3	10	5	11	7	36
4	1	3	3	10	5	11	7	26
2	1	3	3	7	5	11	7	15
1	1	2	3	4	5	6	7	8
Index:	0	1	2	3	4	5	6	7

$\{36\}$

$\{10, 26\}$,

$\{3, 7, 11, 15\}$,

$\{1, 2, 3, 4, 5, 6, 7, 8\}$

lvl	*tid/indicator*				*stride*
3	0/7	1/15	2/23	3/46	8
2	0/3	1/7	2/11	3/15	4
1	0/1	1/3	2/5	3/7	2
0					1

Fig. 5. Relation between Output (left) and *p_hist* (middle) according to active threads (grey color) in the table (right): $Output[indicator] == p\_hist[lvl - 1][2 \times tid + 1]$ and $Output[indicator - stride] == p\_hist[lvl - 1][2 \times tid]$ $(lvl > 0)$.

List. 3. The *Build_partial_history* function

```
    //@ requires |p_hist_prev_lvl| ≥ 0;
    static pure seq<int> Build_partial_history(seq<int> p_hist_prev_lvl) =
        1 < |p_hist_prev_lvl| ?
            seq<int> {head(p_hist_prev_lvl) + head(tail(p_hist_prev_lvl))} +
            Build_partial_history(tail(tail(p_hist_prev_lvl))) : p_hist_prev_lvl;
```

List. 4. The *epsum* function

```
    /*@ requires 0 ≤ i && i ≤ |p_hist_lvl|;
        ensures |\result| == |p_hist_lvl|-i;
        ensures (\forall int j; j ≥0 && j <|\result|;
                 \result[j] == intsum(take(p_hist_lvl, i+j))); @*/
    static pure seq<int> epsum(seq<int> p_hist_lvl, int i) =
        i <|p_hist_lvl| ? seq<int> {intsum(take(p_hist_lvl, i))} + epsum(
             p_hist_lvl, i+1) :
        seq<int> { };
```

Relating Ghost Variables and Concrete Variables. We proved functional correctness over the ghost variables, but we need to prove it against the actual arrays. Therefore, the last step is to relate them. First of all, It is trivial to relate the levels in *f_hist* to the output array, because of the postconditions in List 2 (lines 2–8), but we should relate the output array and *p_hist*. Figure 5 indicates the relationship between the output array and *p_hist*, according to *tid* and *indicator*, where gray colors (in the table) indicate the active threads in each iteration. The loop of the algorithm starts from level 1. We update the values in the output array according to the current values. Correspondingly, the values are created in *p_hist* according to the previous level. The *indicator* and *stride* are also updated in each iteration. In the output array and *p_hist*, the same colors belong to one thread according to *tid*, *indicator* and *stride*. The invariants that we have in each iteration of up-sweep is $Output[indicator] == p\_hist[lvl - 1][2 \times tid + 1]$ and $Output[indicator - stride] == p\_hist[lvl - 1][2 \times tid]$. To prove them as loop invariants in VerCors, we need some smaller steps and prove a property:

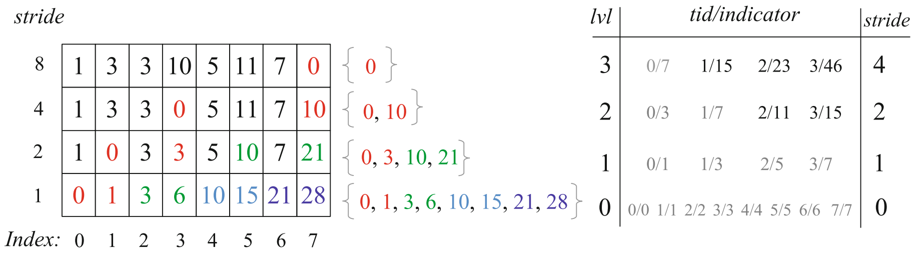


Fig. 6. Relation between the actual array, *Output*, (left) and the ghost variable, *down_seq* (middle) according to active threads (grey color) in the table (right).

Property 1. For any sequence xs:
$\forall i . 0 \leq i < |xs| \rightarrow \mathit{Build\_partial\_history}(xs)[i] == xs[2 \times i] + xs[2 \times i + 1]$.

Using this property and the invariants, we can establish the relation between the output array and *p_hist*. The invariants that hold in each iteration of down-sweep is $\mathit{Output}[\mathit{indicator}] == \mathit{down\_seq}[\mathit{tid}]$ and $\mathit{Output}[\mathit{indicator} - \mathit{stride}] == \mathit{p\_hist}[\mathit{lvl}][2 \times \mathit{tid}]$ (see Fig. 6, for an example). Again, the gray colors indicate the active threads and the same colors (in ghost and array) belong to one thread. To prove the invariants in the tool, we first prove these two properties:

Property 2. For any sequence xs:
$\forall i . 0 \leq i < |xs|/2 \rightarrow \mathit{epsum}(\mathit{Build\_partial\_history}(xs))[i] == \mathit{epsum}(xs)[2 \times i]$.

Property 3. For any sequence xs:
$\forall i . 0 \leq i < |xs|/2 \rightarrow \mathit{epsum}(xs)[2 \times i + 1] == \mathit{epsum}(xs)[2 \times i] + xs[2 \times i]$.

As in up-sweep, by using the invariants, the two properties and several intermediate small steps, we can establish the relation between *down_seq* and the output array. We refer to the implementation for further proof details.

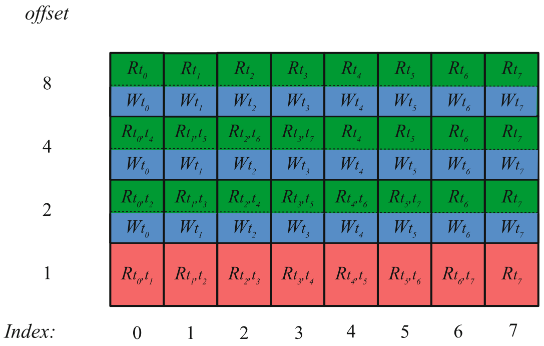
4 Verification of Kogge-Stone's Algorithm

This section explains the verification of Kogge-Stone's parallel prefix sum algorithm. We discuss how to verify this algorithm using the same approach as before. Again, we first discuss data race-freedom and then functional correctness. We only present the main ideas and refer to Appendix B for crucial annotations[7].

4.1 Data Race-Freedom

To verify data race freedom of this algorithm, we need to specify permissions over the output array. Figure 7 shows the permission pattern in each iteration.

[7] The source code is available at https://github.com/Safari1991/Prefixsum.



offset								
8	Rt_0	Rt_1	Rt_2	Rt_3	Rt_4	Rt_5	Rt_6	Rt_7
	Wt_0	Wt_1	Wt_2	Wt_3	Wt_4	Wt_5	Wt_6	Wt_7
4	Rt_0,t_4	Rt_1,t_5	Rt_2,t_6	Rt_3,t_7	Rt_4	Rt_5	Rt_6	Rt_7
	Wt_0	Wt_1	Wt_2	Wt_3	Wt_4	Wt_5	Wt_6	Wt_7
2	Rt_0,t_2	Rt_1,t_3	Rt_2,t_4	Rt_3,t_5	Rt_4,t_6	Rt_5,t_7	Rt_6	Rt_7
	Wt_0	Wt_1	Wt_2	Wt_3	Wt_4	Wt_5	Wt_6	Wt_7
1	Rt_0,t_1	Rt_1,t_2	Rt_2,t_3	Rt_3,t_4	Rt_4,t_5	Rt_5,t_6	Rt_6,t_7	Rt_7
Index:	0	1	2	3	4	5	6	7

Fig. 7. Permissions in Kogge-Stone's algorithm; R_{t_i,t_j} indicates read permission by threads i and j, W_{t_i} indicates write permission by thread i, red color shows initial permissions, blue/green show how the permissions change in the first/second barrier. (Color figure online)

As in Algorithm 2, each thread (*tid*) first needs read permission to locations *tid* and *tid* – *offset* (lines 4 and 6). Since *offset* initially is 1, each thread (*tid*) needs read permission to its own (*tid*) and its left (*tid* − *1*) locations as indicated by the red color in Fig. 7. Then, in the first barrier (line 7), each thread gives up read permissions and obtains write permission to its location to store the results of the computation in line 9 (as shown in blue in Fig. 7). Finally, threads reach the second barrier (line 10) and we change the permissions according to the new value of *offset* for the next iteration. This is indicated in green in the figure. This pattern is repeated by each iteration of the algorithm. At the end of this algorithm, since *offset* is greater than all *tids*, each thread only has read permission to its own location (*tid*).

4.2 Functional Correctness

Next, we briefly discuss how to verify functional correctness of the algorithm. The difference between this algorithm and the previous one is that first, Kogge-Stone is an inclusive prefix sum algorithm and second, there is only one phase. Having one phase makes it easier to verify functional correctness, even though this algorithm is in-place as well. We could reuse the functions and operations we defined for the earlier verification. Since this algorithm is for an inclusive prefix sum, first of all, we slightly change the definition of *epsum* to be an inclusive prefix sum (*ipsum*). The strategy to verify this algorithm is the same as before, i.e., we define a ghost variable to capture the elements in the output array and a function to update this ghost variable in the same way as the actual computation does. Then, we prove functional correctness over this ghost variable by using a suitable property. Finally, we relate the ghost variable to the output array.

As we can see in Fig. 2, in each iteration, the values from index 0 up to index *offset* are actually the inclusive prefix sum of the input array. We use this property as a loop invariant to show that at the end of the algorithm, we have the prefix sum of the input array. Thus, we define a ghost variable, *temp_seq*, and we update it inside the loop according to the *partial_prefixsum* function in List 5. This function captures the same computation as in the algorithm. We can see from the postcondition of the function (lines 4–6 in List 5) that if *index* (and the corresponding *tid*) is less than *offset*, then the second `intsum` returns 0, and the first `intsum` returns the prefix sum up to *index*[8]. Thus, in each iteration for *tid* less than *offset* the result will be the prefix sum in *temp_seq*. Therefore, in the end, when *offset* is the length of the input (and output) array, all values in the ghost variable are the prefix sum of the values in the input array.

List. 5. The *partial_prefixsum* function

```
1      /*@ requires |input_seq| ≥ 0 && index ≥ 0 && index ≤ |input_seq|;
2          requires offset > 0 && offset ≤ 2×|input_seq|;
3          ensures |\result| == |input_seq| - index;
4          ensures (\forall int j; 0≤ j && j <|\result|; \result[j] ==
5                  intsum(take(input_seq, index+j+1)) -
6                  intsum(take(input_seq, index+j+1-offset))); @*/
7      static pure seq<int> partial_prefixsum(seq<int> input_seq, int index,
           int offset) =
8        index < |input_seq| ? seq<int> {intsum(take(input_seq, index+1)) -
9          intsum(take(input_seq, index+1-offset))} +
10         partial_prefixsum(input_seq, index+1, offset) : seq<int> { };
```

As we use *offset* in the function and from the postcondition that we defined, VerCors can infer that in each iteration for *tid* less than *offset*, *temp_seq* and the output array have the same values (specified by a loop invariant). Thus, we conclude that Kogge-Stone's algorithm indeed computes the prefix sum.

5 Related Work

There are a few approaches to reason about GPGPU programs which mostly focus on finding data races. In dynamic approach, programs are instrumented, and then memory accesses are recorded by running them, trying to identify data races (e.g., cuda-memcheck [18], Oclgrind [19] and GRace [21]). This is a simple technique to apply, but since it depends on concrete inputs, it does not guarantee the absence of data races. An improvement over this approach is dynamic symbolic execution where concrete and symbolic (concolic) execution is used, such as GKLEE [17] and KLEE-CL [11]. There are also several static approaches to verify data race-freedom of GPGPU programs. In static approaches, we use logic and theorem provers to guarantee the absence of data races. The key of this approach is using invariants to prove data race-freedom. In addition to VerCors, tools such as PUG [16] and GPUVerify [3] are based on this approach. Except VerCors and VeriFast [14], none of these tools can reason about functional correctness of parallel programs. VeriFast is a verification tool based on static approach to

[8] Note that, the *partial_prefixsum* is a recursive function. In lines 4–6, for the final result, j is 0 and the parameter of `take` will be *index* + *1*, which means the first *index* + *1* elements (i.e., starting from 0 it becomes up to element *index*).

prove functional correctness of single-threaded and multithreaded C and Java programs, but not able to reason about GPGPU programs.

The closest related work to our paper is by Chong et al. [10] where they verify data race-freedom and propose a method to verify functional correctness of Blelloch's and Kogge-Stone's algorithm along with two other parallel prefix sum algorithms for all inputs *up to fixed sizes*. They show that if a parallel prefix sum algorithm is proven to be data race-free, then the correctness can be established by generating one test case. Therefore, they use GPUVerify to prove data race-freedom of 4 parallel prefix sum algorithms. Their approach is applicable for any parallel prefix sum algorithm with other operations and types instead of summation and integers. Comparing VerCors to their tool, GPUVerify benefits from more automation, while we need to specify the annotations manually. However, since GPUVerify is based on model-checking approaches, to verify even data race-freedom of GPU programs, the input size must be bounded. As a result, they only show functional correctness for *a fixed input size* (a realistic size for current GPUs). In this paper, we verified data race-freedom and also functional correctness of the two algorithms for *any arbitrary size of input*. We believe that it should be no problem to also prove the other two algorithms.

6 Conclusion

This paper shows how we verify data race-freedom and functional correctness of the two most widely-used parallel prefix sum algorithms, Blelloch's and Kogge-Stone's algorithm, for *an arbitrary input size* by encoding the algorithms into VerCors verifier. Proving functional correctness of Blelloch's algorithm is challenging for multiple reasons. First, the algorithm is in-place. Second, it consists of two independent, but related phases and third, it is non-trivial to relate the computations in both phases to conclude the desired end result (i.e., that it establishes a prefix sum). We overcome these challenges by introducing ghost variables and defining suitable functions that mimic the computations on the ghost variables. Moreover, we prove suitable properties that help us to reason about the algorithm. The verification of Kogge-Stone's algorithm is not as hard as the first one, since there is only one phase and the property that we define is straightforward. We benefit from functions, operations and properties that are defined in the earlier verification and reuse them in the second verification.

As future work, we plan to verify more complicated parallel algorithms that use the prefix sum algorithm internally, such as stream compaction and sorting algorithms. We also would like to investigate how to further automate the process of proof creation. We believe that a substantial part of the required annotations, in particular those related to permissions, can be generated automatically. In addition, we plan to add a CUDA front-end to the tool.

References

1. Amighi, A., Haack, C., Huisman, M., Hurlin, C.: Permission-based separation logic for multithreaded Java programs. LMCS **11**(1), 2–65 (2015)

2. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_6
3. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: OOPSLA, pp. 113–132. ACM (2012)
4. Blelloch, G.E.: Prefix Sums and their Applications, Synthesis of Parallel Algorithms. Morgan Kaufmann Publishers Inc., San Francisco (1993)
5. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 102–110. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_7
6. Blom, S., Huisman, M., Mihelčić, M.: Specification and verification of GPGPU programs. Sci. Comput. Program. **95**, 376–388 (2014)
7. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL, pp. 259–270 (2005)
8. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44898-5_4
9. Brent, R.P., Kung, H.T.: A regular layout for parallel adders. IEEE Trans. Comput. **3**, 260–264 (1982)
10. Chong, N., Donaldson, A.F., Ketema, J.: A sound and complete abstraction for reasoning about parallel prefix sums. In: ACM SIGPLAN Notices, vol. 49, pp. 397–409. ACM (2014)
11. Collingbourne, P., Cadar, C., Kelly, P.H.J.: Symbolic testing of OpenCL code. In: Eder, K., Lourenço, J., Shehory, O. (eds.) HVC 2011. LNCS, vol. 7261, pp. 203–218. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34188-5_18
12. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with CUDA. GPU Gems **3**(39), 851–876 (2007)
13. Horn, D.: Stream reduction operations for GPGPU applications. GPU Gems **2**(36), 573–589 (2005)
14. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4
15. Kogge, P.M., Stone, H.S.: A parallel algorithm for the efficient solution of a general class of recurrence equations. IEEE Trans. Comput. **100**(8), 786–793 (1973)
16. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: SIGSOFT FSE 2010, Santa Fe, NM, USA, pp. 187–196. ACM (2010)
17. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: concolic verification and test generation for GPUs. In: ACM SIGPLAN Notices, vol. 47, pp. 215–224. ACM (2012)
18. Nvidia: Cuda-memcheck: User manual (version 10) (2019). https://developer.nvidia.com/cuda-memcheck
19. Price, J., McIntosh-Smith, S.: Oclgrind: an extensible OpenCL device simulator. In: Proceedings of the 3rd International Workshop on OpenCL, p. 12. ACM (2015)
20. Sklansky, J.: Conditional-sum addition logic. IRE Trans. Electron. Comput. **2**, 226–231 (1960)
21. Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: GRace: a low-overhead mechanism for detecting data races in GPU programs. ACM SIGPLAN Not. **46**(8), 135–146 (2011)