Formal Verification of Parallel Stream Compaction and Summed-Area Table Algorithms

Mohsen Safari(✉) and Marieke Huisman

Formal Methods and Tools, University of Twente, Enschede, The Netherlands
{m.safari,m.huisman}@utwente.nl

Abstract. Dedicated many-core processors such as GPGPUs, enable programmers to design and implement parallel algorithms to optimize performance. The stream compaction and summed-area table algorithms are two examples where parallel versions have been proposed in the literature with substantial speed ups compared to sequential counterparts.

Since these two algorithms are widely used, their correctness is of the utmost importance, i.e., the algorithms must be functionally correct and their implementations must be memory safe. These algorithms use the parallel prefix sum algorithm internally. In our previous work, we verified two parallel prefix sum algorithms. In this paper, we show how we can reuse a verified sub-function (i.e., prefix sum) to prove more complicated algorithms (i.e., stream compaction and summed area table) in a modular way with less effort. Moreover, we demonstrate that it is feasible in practice to verify larger case studies by building the verification of the complicated algorithm on top of the basic one.

To show the correctness of the algorithms, we use deductive program verification based on permission-based separation logic, which is supported by the program verifier VerCors. To the best of our knowledge, we are the first to verify *functional correctness* of the *parallel* stream compaction and summed-area table algorithms for an arbitrary array size, using *tool support*.

Keywords: GPU verification · Deductive verification · Separation logic

1 Introduction

Many parallel algorithms have been proposed for optimizing performance by exploiting the new parallel architectures, and parallelizing sequential algorithms is an active area of research. General Purpose Graphics Processing Units (GPGPUs) are one of the promising parallel architectures, where many threads execute the same instructions, but on different data (known as SIMD). Stream compaction and summed-area table [11] algorithms are two examples where the parallel (GPU-based) implementations [3,12–14,24] outperform the sequential (CPU-based) counterparts.

Stream compaction reduces an input array to a smaller array by removing undesired elements. This is an important primitive operation on GPUs, because a variety of applications such as collision detection and sparse matrix compression rely on it. The reduction in size by eliminating undesired elements is useful because (1) the computation can be done more efficiently by not wasting the computation power on undesired elements and, (2) it greatly reduces the transfer costs between the CPU and GPU, especially for applications where data transfer between CPU and GPU is frequent.

A summed-area table is a two-dimensional (2D) table generated from a 2D input array where each entry in the table is the sum of all values in the square defined by the entry location and the upper-left corner in the input array. Generating such a table is useful in computer graphics and image processing [13].

Since these two algorithms are widely used in practice (also as a building block in other applications), their correctness is of the utmost importance. This means not only that the algorithms should be memory and thread safe (e.g., free of data races[1]), *but also* that they should be functionally correct, i.e., they should actually produce the result we expect. Concretely, functional correctness for stream compaction means that the result must be the compacted input array with exactly the desired elements. In case of the summed-area table, functional correctness means that the result must be a table, the same size as the input, where each entry contains the sum of all elements in the square defined by the entry location and the upper-left corner in the input.

The two algorithms exploit the prefix sum algorithm, which takes an array of integers and, for each element, computes the sum of the previous elements. In our previous work [23], we verified data race-freedom and functional correctness of two parallel prefix sum algorithms.

In this paper, we investigate (1) how we can profit from already verified sub-functions (i.e., prefix sum) to prove the stream compaction and summed-area table algorithms; i.e., how much effort is needed to adapt the specifications from [23] for the verification of these two algorithms; and (2) how much time is needed to verify them in comparison to the verification of our previous work on prefix sum. We believe such case studies are important to gain more insight in the effort that is needed to verify complex algorithms and how more automation can be added to the verification process. In general, proving functional correctness of parallel algorithms is a challenging task. In particular, proving functional correctness of these two algorithms is challenging because (1) in the stream compaction algorithm, the input of the prefix sum sub-function is an array of flag and the output is used as indices of elements in another array. Therefore, additional properties should be proved to reason about the prefix sum result to be safely used as indices; and (2) in the summed-area table algorithm, in addition to the prefix sum, the transposition operation is used intermittently. Due to these intermediate steps, we should store the manipulated values and

[1] A data race is a situation when two or more threads may access the same memory location simultaneously, and at least one of them is a write.

establish a formal relation between the values of each step in order to reason about the final result (i.e, output).

To prove memory safety and functional correctness of the stream compaction and summed-area table algorithms, we use VerCors [5], which is a deductive program verifier for concurrent programs. Deductive program verification is a static approach to verify program properties by augmenting the source code with pre- and postconditions. To guide verification, intermediate annotations are added to capture the intermediate properties of the program. Then, in our case, the annotated code is translated into proof obligations (via Viper [20]), which are discharged to an automated theorem prover; the SMT solver Z3 [19].

To the best of our knowledge, this is the only *tool-supported* verification of *data race-freedom* and *functional correctness* of the two parallel algorithms for any *arbitrary size of input*. None of the existing other approaches to analyze GPU applications is able to verify similar properties. Most approaches are dynamic [10,18,21,22,25], and only aim to find bugs. Other existing static verification techniques [2,9,15,17] either require a bound on the input size, or they do not fully model all aspects of GPU programming, such as the use of barriers. We show that the verification of larger case studies is feasible, by adding the verification of the more complex algorithm on top of the basic one, not only in theory, but also in practice using tool support. Moreover, our work enables the verification of other parallel algorithms that are built on top of the stream compaction and summed-area table algorithms, such as collision detection and box filtering.

Contributions. The main contributions of this paper are:

1. We provide a tool-supported proof of data race-freedom and functional correctness of the parallel stream compaction algorithm for any input size.
2. We show that the parallel summed-area table algorithm is data race-free and functionally correct for arbitrary input sizes using tool support.
3. We demonstrate how much effort and time needed in practice to verify complicated algorithms by reusing verified algorithms in a layered manner.

Organization. Section 2 discusses related work and Sect. 3 explains the necessary background. Sections 4 and 5 describe how to specify and verify the correctness of the stream compaction and summed-area table algorithms, respectively. Section 6 concludes the paper.

2 Related Work

GPGPU programming is becoming more popular because of its potential to increase the performance of programs. However, it is also highly error-prone due to its inherent parallization. Therefore, the demand for guaranteeing correctness of GPGPU programs is growing. There are only a few approaches to reason about GPGPU programs; most of them focus on finding data races.

In dynamic analysis, a program is instrumented to record memory accesses. Then, by running the instrumented program, data races might be identified (e.g., cuda-memcheck [21], Oclgrind [22] and GRace [25]). This technique depends on concrete inputs and cannot guarantee data race-freedom. Dynamic symbolic execution is a combination of static and dynamic analysis to combine concrete and symbolic inputs to find data races (e.g., GKLEE [18] and KLEE-CL [10]).

Static approaches analyse the complete state space of a program without running it. Deductive program verification as in VerCors, is a static approach, where a program is annotated with intermediate (invariant) properties. Tools such as PUG [17] and GPUVerify [2] use static analysis, but require less annotations. Except VerCors and VeriFast [15], none of these tools can reason about functional correctness of parallel programs. VeriFast aims at proving functional correctness of single-threaded and multi-threaded C and Java programs, but it is not specifically tailored to reason about GPGPU programs.

There is no previous work on formally verifying the parallel stream compaction and summed-area table algorithms on GPUs. To verify these two algorithms, the parallel prefix sum algorithms need to be verified, which has been done by Chong et al. [9] in addition to our previous work [23]. Chong et al. verify data race-freedom and propose a method to verify functional correctness of four different parallel prefix sum algorithms for a fixed input size. 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. Then they use GPUVerify to prove data race-freedom of the parallel prefix sum algorithms for a fixed input size. In our previous work [23], we prove data-race freedom and functional correctness of two different parallel prefix sum algorithms using the VerCors verifier for any arbitrary size of input. We benefit from ghost variables to reason about in-place prefix sum algorithms. In our opinion, the advantages of our approach is that our verification approach for the prefix sum can be reused for these two new algorithms, while Chong would not be able to reuse his prefix sum approach to verify the parallel stream compaction and summed-area table algorithms.

3 Background

This section describes the program verifier VerCors and the logic behind it by illustrating an example. It then briefly discusses the parallel prefix sum algorithm which is used in both parallel stream compaction and summed-area table algorithms.

3.1 VerCors

VerCors[2] 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., data race-freedom) and

[2] The tool is available at: https://github.com/utwente-fmt/vercors.

List. 1. A simple annotated OpenCL program

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

functional correctness of programs. The programs are annotated with pre- and postconditions in permission-based separation logic [1,7]. Permissions are used to capture which heap memory locations may be accessed by which threads, and are used to guarantee thread safety. 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.

Blom et al. [6] show how to use permission-based separation logic to reason about GPU kernels including barriers. We illustrate this logic by an example. Listing 1 shows a specification of a kernel that rotates the elements of an array to the left[3]. It is specified by a thread-level specification. To specify permissions, we use predicate $Perm(L, \pi)$ where L is a heap location and π a fractional value in the interval (0, 1][4]. Pre- and postconditions, (denoted by keywords `requires` and `ensures`, respectively in lines 2–5), must hold at the beginning and the end of the function, respectively. The keyword `context_everywhere` is used to specify an invariant (line 1) that must hold throughout the function. As preconditions, each thread has read permission to its right neighbor (except thread "size-1" which has read permission to the first index) in line 2. The postconditions indicate (1) each thread has write permission to its location (line 3) and (2) the result of the function as a left rotation of all elements (lines 4–5). Each thread is responsible for one array location and it first reads its right location (line 9). Then it synchronizes in the barrier (line 13). When a thread reaches a barrier, it has to fulfill the barrier preconditions, and then it may assume the barrier postconditions. Thus barrier postconditions must follow from barrier preconditions. In this case, each thread gives up read permission on its right location and obtains write permission on its "own" location at index *tid* (lines 11–12). After that, each thread writes the value read before to its "own" location (line

[3] We assume there is one workgroup and "size" threads inside it.

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

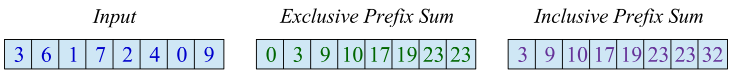

Fig. 1. An example of exclusive and inclusive prefix sum of an input.

14). Note that the keyword `\old` is used for an expression to refer to the value of that expression before entering a function (lines 4–5). The OpenCL example (Listing 1) 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 encoding directly, but it is straightforward to adapt this to the verification of the OpenCL kernel.

3.2 Prefix Sum

Figure 1 illustrates an example of the prefix sum operation. This operation is a basic block used in both stream compaction and summed-area table algorithms, defined as: given an array of integers, the prefix sum of the array is another array with the same size such that each element is the sum of all previous elements. An (inclusive) prefix sum algorithm has the following input and output:

– 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 is as follows:

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

Blelloch [4] introduced an exclusive parallel in-place prefix sum algorithm and Kogge-Stone [16] proposed an inclusive parallel in-place prefix sum algorithm. These two parallel versions are frequently used in practice (as a primitive operation in libraries AMD APP SDK[5], and NVIDIA CUDA SDK[6]).

4 Verification of Parallel Stream Compaction Algorithm

This section describes the stream compaction algorithm and how we verify it. First, we explain the algorithm and its encoding in VerCors. Then, we prove data

[5] http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk.
[6] https://developer.nvidia.com/gpu-computing-sdk.

Algorithm 1. Stream Compaction Algorithm

```
1: function STREAM_COMPACTION(int[] $Input$, int[] $Output$, int[] $Flag$, int[] $ExPre$, int $N$)
2:   Par($tid = 0..\ N$)
3:     EXCLUSIVE_PREFIXSUM($Flag$, $ExPre$, $tid$, $N$);
4:     Barrier($tid$);
5:     if  $Flag[tid] == 1$ then
6:         $Output[ExPre[tid]] = Input[tid]$;
```

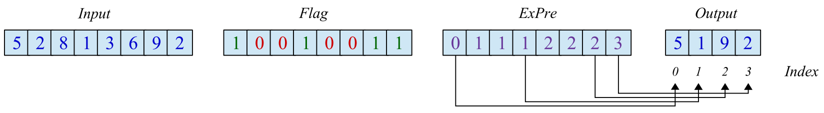

Fig. 2. An example of stream compaction of size 8.

race-freedom and we show how we prove functional correctness of the algorithm. Moreover, we show how we can reuse the verified prefix sum from our previous work [23] to reason about the stream compaction algorithm. We explain the main ideas mostly by using pictures instead of presenting the full specification[7].

4.1 Stream Compaction Algorithm

Given an array of integers as input and an array of booleans that flag which elements are desired, stream compaction returns an array that holds only those elements of the input whose flags are true. An algorithm is a stream compaction if it satisfies the following:

- INPUT: two arrays, *Input* of integers and *Flag* of booleans of size N.
- OUTPUT: an array *Output* of size M $(M \leq N)$ such that
  - $\forall j.\ 0 \leq j < M\colon Output[j] = t \Rightarrow \exists i.\ 0 \leq i < N\colon Input[i] = t \ \wedge\ Flag[i]$.
  - $\forall i.\ 0 \leq i < N\colon Input[i] = t \ \wedge\ Flag[i] \Rightarrow \exists j.\ 0 \leq j < M\colon Output[j] = t$.
  - $\forall i,j.\ 0 \leq i,j < N\colon (\,Flag[i] \ \wedge\ Flag[j] \ \wedge\ i < j \iff (\exists k,l.\ 0 \leq k,l < M\colon$ $Output[k] = Input[i] \ \wedge\ Output[l] = Input[j] \ \wedge\ k < l)\,)$.

Algorithm 1 shows the pseudocode of the parallel algorithm and Fig. 2 presents an example of stream compaction. Initially we have an input and a flag array (implemented as integers of zeros and ones). To keep the flagged elements and discard the rest, first we calculate the exclusive prefix sum (from [4]) of the flag array. Interestingly, for the elements whose flags are 1, the exclusive prefix sum indicates their location (index) in the output array. In the implementation, the input of the prefix sum function is *Flag* and the output is stored in *ExPre* (line 3). Then all threads are synchronized by the barrier in line 4, after which all the desired elements are stored in the output array (lines 5–6).

[7] The full specification is available at https://github.com/Safari1991/Prefixsum-Applications.

4.2 Data Race-Freedom

To prove data race-freedom, we specify how threads access shared resources by adding permission annotations to the code. In Algorithm 1, we have several arrays that are shared among threads. There are three locations in the algorithm where permissions can be redistributed: before Algorithm 1 as preconditions, in the exclusive prefix sum function as postconditions and in the barrier (redistribution of permissions). Figure 3 visualizes the permission pattern for those shared arrays, which reflects the permission annotations in the code according to these three locations. The explanation of the permission patterns in each array in these three locations is as follows:

- *Input*: since each thread (*tid*) only needs read permission (line 6 in Algorithm 1), we define each thread to have read permissions to its "own" location at index *tid* throughout the algorithm (Fig. 3). This also ensures that the values in *Input* cannot be changed.
- *Flag*: since *Flag* is the input of the exclusive prefix sum function, its permission pattern at the beginning of Algorithm 1 must match the permission preconditions of the exclusive prefix sum function. Thus, following the preconditions of this function (see [23]), we define the permissions such that each thread (*tid*) has read permissions to its "own" location (Fig. 3: left). The exclusive prefix sum function returns the same permissions for *Flag* in its postconditions (Fig. 3: middle). Since, each thread needs read permission in line 5 of Algorithm 1, we keep the same permission pattern in the barrier as well (Fig. 3: right).
- *ExPre*: since *ExPre* is the output of the exclusive prefix sum function, the permission pattern at the beginning of Algorithm 1 should match the permission preconditions of the exclusive prefix sum function (specified in [23]). Thus, each thread ($tid <$ half *ExPre* size) has write permissions to locations $2 \times tid$ and $2 \times tid + 1$ (Fig. 3: left). As postcondition of the exclusive prefix sum function (specified in [23]), each thread has write permission to its "own" location in *ExPre* (Fig. 3: middle). Since each thread only needs read permission in line 6 of Algorithm 1, we change the permission pattern from write to read in the barrier (Fig. 3: right).
- *Output*: it is only used in line 6 of Algorithm 1 and its permissions are according to the values in *ExPre*. Thus, the initial permissions for *Output* can be arbitrary and in the barrier, we specify the permissions such that each thread (*tid*) has write permission in location $ExPre[tid]$ if its flag is 1 (indicated by t_f in Fig. 3: right).

4.3 Functional Correctness

Proving functional correctness of the parallel stream compaction algorithm consists of two parts. First, we prove that the elements in the exclusive prefix sum function (*ExPre*) are in the range of the output, thus they can be used safely as indices in *Output* (i.e., line 6 in Algorithm 1). Second, we prove that *Output*

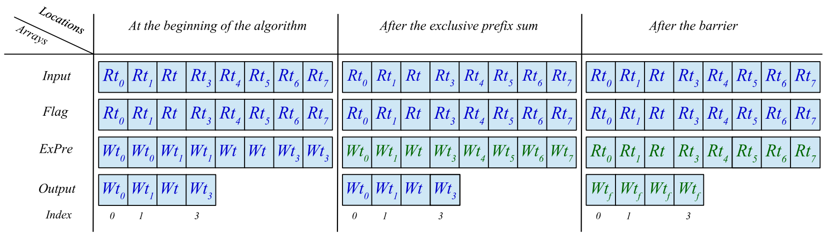
Locations
Arrays
At the beginning of the algorithm
After the exclusive prefix sum
After the barrier
Input
Flag
ExPre
Output
Index

Fig. 3. Permission pattern of arrays in stream compaction algorithm corresponding to Fig. 2; Rt_i/ Wt_i means thread i has read/write permission. Green color indicates permission changes. (Color figure online)

contains all the elements whose flags are 1, and does not contain any elements whose flags are not 1. Moreover, the order of desired elements, the ones whose flags are 1, in *Input* must be the same as in *Output*.

To prove both parts, we use ghost variables[8], defined as sequences. There are some advantages of using ghost variables as sequences: (1) it is not required to define permissions over sequences; (2) we can define pure functions over sequences to mimic the actual computations over concrete variables; (3) we can easily prove desired properties (e.g., functional properties) over ghost sequences; and (4) ghost variables can act as histories for concrete variables whose values might change during the program. This gives us a global view (of program states) of how the concrete variables change to their final value. Concretely, we define two ghost variables, *inp_seq* and *flag_seq* as sequences of integers to capture all values in arrays *Input* and *Flag*, respectively. Since values in *Input* and *Flag* do not change during the algorithm[9], *inp_seq* and *flag_seq* are always the same as *Input* and *Flag*[10].

First, to reuse of the exclusive prefix sum specification (line 3 in Algorithm 1) from our previous work [23], we should consider two points: (1) the input to the exclusive prefix sum (*Flag*) in this paper is restricted to 0 and 1; and (2) the elements in the exclusive prefix sum function (*ExPre*) should be safely usable as indices in *Output* (i.e., line 6 in Algorithm 1). Therefore, we use VerCors to prove some suitable properties to reason about the values of the prefix sum of the flag. For space reasons, we show the properties without discussing the proofs here. The first property that we prove in VerCors is that the sum of a sequence of zeros and ones is non-negative[11]:

Property 4.1

$$(\forall i. 0 \leq i < |flag\_seq| \colon flag\_seq[i] = 0 \; vee \; flag\_seq[i] = 1) \Rightarrow$$
$$intsum(flag\_seq) \geq 0.$$

[8] Ghost variables are not part of the algorithm and are used only for verification purposes.

[9] Note that threads only have read permissions over *Input* and *Flag*.

[10] Thus, properties for *inp_seq* and *flag_seq* also hold for *Input* and *Flag*.

[11] The `intsum` operation sums up all elements in a sequence.

List. 2. The *filter* function

```
/*@ requires |inp_seq| == |flag_seq|;
    requires (\forall int i; 0 ≤ i && i < |flag_seq|; flag_seq[i]==0 || flag_seq[i]==1);
    ensures |\result| == intsum(flag_seq);
    ensures 0 ≤ |\result| && |\result| ≤ |flag_seq|; @*/
static pure seq<int> filter(seq<int> inp_seq, seq<int> flag_seq) = |inp_seq|>0 ?
  head(flag_seq)==1 ? seq<int>{head(inp_seq)} + filter(tail(inp_seq), tail(flag_seq))
  : filter(tail(inp_seq), tail(flag_seq)) : seq<int>{};
```

We need Property 4.1 since the prefix sum for each element is the sum of all previous elements. We benefit from the first property to prove in VerCors that all the elements in the exclusive prefix sum of a sequence *flag_seq* (only zeros and ones) are greater than or equal to zero and less than or equal to the sum of elements in *flag_seq*[12]:

Property 4.2

$$\begin{gathered}(\forall i. 0 \leq i < |flag\_seq|\colon flag\_seq[i] = 0 \ \vee \ flag\_seq[i] = 1) \Rightarrow \\ (\forall i. 0 \leq i < |epsum(flag\_seq)|\colon epsum(flag\_seq)[i] \geq 0 \ \wedge \\ epsum(flag\_seq)[i] \leq intsum(flag\_seq)).\end{gathered}$$

This gives the lower and upper bound of elements in the prefix sum, which are used as indices in *Output*. This property is not sufficient to prove that the elements are in the range of *Output* due to two reasons. First, an element in the prefix sum can be as large as the sum of ones in the flag. Hence, it might exceed *Output* size which is in the range 0 to *intsum*(*flag_seq*) − *1*. Second, we only use the elements in the prefix sum whose flags are 1. Property 4.2 does not specify those elements explicitly. Therefore, we prove another property in VerCors to explicitly specify the elements in the prefix sum whose flags are 1 as follows:

Property 4.3

$$\begin{gathered}(\forall i. 0 \leq i < |flag\_seq|\colon flag\_seq[i] = 0 \ \vee \ flag\_seq[i] = 1) \Rightarrow \\ (\forall i. 0 \leq i < |epsum(flag\_seq)| \ \wedge \ flag\_seq[i] = 1\colon \\ (epsum(flag\_seq)[i] \geq 0 \ \wedge \ epsum(flag\_seq)[i] < intsum(flag\_seq))).\end{gathered}$$

Property 4.3 guarantees that the elements in the prefix sum whose flags are 1 are truly in the range of *Output*, and can be used safely as indices. Moreover, it has been proven in [23] that *epsum*(*flag_seq*) is equal to the result of the prefix sum function (i.e., *ExPre*).

Second, we reason about the final values in *Output*, using the following steps:

1. Define a ghost variable as a sequence.
2. Define a mathematical function that updates the ghost variable according to the actual computation of the algorithm.

[12] The `epsum` operation of a sequence returns an exclusive prefix sum of that sequence.

List. 3. The proof steps to relate *out_seq* to *Output* array

```
1 seq<int> out_seq = filter(inp_seq, flag_seq);
2 assert |out_seq| == intsum(flag_seq); // by line 4 in Listing 2
3 if(flag_seq[tid] == 1)
4    // applying Property 4.4
5    assert inp_seq[tid] == filter(inp_seq, flag_seq)[epsum(flag_seq)[tid]];
6    assert out_seq == filter(inp_seq, flag_seq); // by line 1
7    assert inp_seq[tid] == out_seq[epsum(flag_seq)[tid]]; // by lines 5−6
8    assert Output[ExPre[tid]] == Input[tid]; // by lines 5−6 in Algorithm 1
9    assert Output[ExPre[tid]] == out_seq[epsum(flag_seq)[tid]]; // by lines 7−8
```

3. Prove functional correctness over the ghost variables by defining a suitable property.
4. Relate the ghost variable to the concrete variable; i.e., prove that the elements in the ghost sequence are the same as in the actual array.

Following this approach, we define a ghost variable, *out_seq*, as a sequence of integers and a mathematical function, *filter*, as shown in Listing 2. This function computes the compacted list of an input sequence, *inp_seq*, by filtering it according to a flag sequence, *flag_seq* (where `head` returns the first element of a sequence and `tail` returns a new sequence by eliminating the first element). Thus, for each element in *inp_seq*, this function checks its flag to either add it to the result (line 6) or discard it (line 7). The function specification has two preconditions: (1) the length of both sequences is the same (line 1) and (2) each element in *flag_seq* is either 0 or 1 (lines 2). The postcondition states that the length of the compacted list (result) is the sum of all elements in *flag_seq* (line 3) which is at most the same length as *flag_seq* (line 4). We apply the *filter* function to *inp_seq* and *flag_seq* (as ghost statements) at the end of Algorithm 1 to update *out_seq*.

To reason about the values in *out_seq* and relate it to *inp_seq* and *flag_seq* we prove the following property in VerCors:

Property 4.4

$$\begin{aligned}&(\forall i.0 \leq i < |flag\_seq|\colon flag\_seq[i] = 0 \ \vee \ flag\_seq[i] = 1) \Rightarrow\\ &(\forall i.0 \leq i < |epsum(flag\_seq)| \ \wedge \ flag\_seq[i] = 1\colon\\ &\ (inp\_seq[i] = filter(inp\_seq, flag\_seq)[epsum(flag\_seq)[i]])).\end{aligned}$$

From Property 4.4, we can prove in VerCors that all elements in *inp_seq* (and *Input*) whose flags are 1 are in *out_seq* and the order is also preserved. Since we specify that the length of *out_seq* is the sum of all elements in the flag, which is the number of ones (line 4 in Listing 2), we also prove that there are no elements in *out_seq* whose flags are not 1.

The last step is to relate *out_seq* to *Output*. Listing 3 shows the proof steps which are located at the end of Algorithm 1. Through some smaller steps, and using Property 4.4 we prove in VerCors that *out_seq* and *Output* is the same

Algorithm 2. Summed-Area Table Algorithm

```
function SUMMED_AREA_TABLE(int[][] $Input$, int[][] $Temp$, int[][] $Output$, int $N$)
 for(int $i = 0$; $i < N$; $i++$)
  Par($tid = 0..\ N$)                                    // First Prefix Sum
   INCLUSIVE_PREFIXSUM($Input[i]$, $Temp[i]$, $inp\_seq[i]$, $tmp1\_seq[i]$, $tid$, $N$);
 Properties 1 and 2 (Table 1) hold here
 Par($tidX = 0..\ N$, $tidY = 0..\ N$)                    // First Transposition
  int $temporary = Temp[tidX][tidY]$;
  Barrier($tidX$, $tidY$);
  $Temp[tidY][tidX] = temporary$;
 $tmp2\_seq = transpose(tmp1\_seq, 0, N)$;
 Properties 3 and 4 (Table 1) hold here
 for(int $i = 0$; $i < N$; $i++$)
  Par($tid = 0..\ N$)                                    // Second Prefix Sum
   INCLUSIVE_PREFIXSUM($Temp[i]$, $Output[i]$, $tmp1\_seq[i]$, $tmp3\_seq[i]$, $tid$, $N$);
 Properties 5 and 6 (Table 1) hold here
 Par($tidX = 0..\ N$, $tidY = 0..\ N$)                    // Second Transposition
  int $temporary = Output[tidX][tidY]$;
  Barrier($tidX$, $tidY$);
  $Output[tidY][tidX] = temporary$;
 $out\_seq = transpose(tmp3\_seq, 0, N)$;
 Properties 7 and 8 (Table 1) hold here
```

(line 9). Note that for each tid, $epsum(flag\_seq)[tid]$ is equal to $ExPre[tid]$ as proven in [23].

As we can see in this verification, we could reuse the specification of the verified prefix sum algorithm, by proving some more properties. We should note that the time we spent to verify the stream compaction algorithm is much less than the verification of the prefix sum algorithm.

5 Verification of Parallel Summed-Area Table Algorithm

This section discusses the summed-area table algorithm and its verification. As above, after describing the algorithm and its encoding in VerCors, we first prove data-race freedom and then explain how we prove functional correctness. We also show how we reuse the verified prefix sum from [23] in the verification of the summed-area table algorithm. Again, we only show the main ideas[13].

5.1 Summed-Area Table Algorithm

Given a 2D array of integers, the summed-area table is a 2D array with the same size where each entry in the output is the sum of all values in the square defined by the entry location and the upper-left corner in the input. The algorithm's input and output are specified in the following way:

[13] The verified specification is available at https://github.com/Safari1991/Prefixsum-Applications.

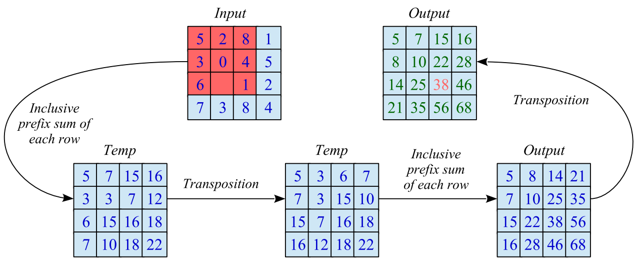
Input
Output
Inclusive prefix sum of each row
Transposition
Temp
Temp
Transposition
Inclusive prefix sum of each row
Output

Fig. 4. An example of summed-area table of size 4 × 4.

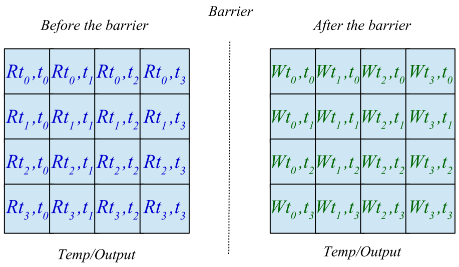

Fig. 5. Permission pattern of matrix Temp/Output before and after the barrier in the transposition phase; $Rt_i, t_j / Wt_i, t_j$ means thread (i, j) has read/write permission.

- INPUT: a 2D array *Input* of integers of size $N \times M$.
- OUTPUT: a 2D array *Output* of size $N \times M$ such that

$$Output[i][j] = \sum_{t=0}^{i} \sum_{k=0}^{j} Input[t][k] for 0 \leq i < N and 0 \leq j < M.$$

Algorithm 2 shows the (annotated) pseudocode of the parallel algorithm and Fig. 4 shows an example for the summed-area table algorithm. For example, the red element 38 in *Output* is the sum of the elements in the red square in *Input*. We apply the inclusive prefix sum (from [16]) to each row of *Input* and store it in *Temp* (lines 2–4). Then, we transpose the *Temp* matrix (lines 6–9). Thereafter, we apply again the inclusive prefix sum to each row of *Temp* (lines 12–14). Finally, we transpose the matrix again, resulting in matrix *Output* (lines 16–19). The parallel transpositions after each prefix sum are determined by creating 2D thread blocks for each element of the matrix (lines 6–9 and 16–19) where each thread (*tidI*, *tidJ*) stores its value into location (*tidJ*, *tidI*) by first writing into a *temporary* variable (lines 7 and 17) and then synchronizing in the barrier (lines 8 and 18).

5.2 Data Race-Freedom

Since data race-freedom of the parallel inclusive prefix sum has been verified in our previous work [23], we only show data-race freedom of the transposition

List. 4. The *transpose* function

```
/*@ requires |xs| == N && (\forall int j; 0 ≤ j && j < N; |xs[j]| == N);
    requires i ≥ 0 && i ≤ N;
    ensures |\result| == N − i;
    ensures (\forall int j; 0 ≤ j && j < N − i; |\result[j]| == N);
    ensures (\forall int j; 0 ≤ j && j < N − i;
                (\forall int k; 0 ≤ k && k < N; \result[j][k] == xs[k][i+j])); @*/
static pure seq<int> transpose(seq<seq<int>> xs, int i, int N) = i < N ?
    seq<seq<int>> {transpose_helper(xs, 0, i, N)} + transpose(xs, i+1, N) :
    seq<int> {};

/*@ requires |xs| == |N| && (\forall int j; 0 ≤ j && j < N; |xs[j]| == N);
    requires k ≥ 0 && k ≤ N && i ≥ 0 && i < N;
    ensures |\result| == N − k;
    ensures (\forall int j ; 0≤j && j<|\result|; \result[j] == xs[k + j][i]); @*/
static pure seq<int> transpose_helper(seq<seq<int>> xs, int k, int i, int N) =
    k < N ? seq<int> {xs[k][i]} + transpose_helper(xs, k+1, i, N) : seq<int> {};
```

phases in Algorithm 2. As an example, Fig. 5 illustrates the permission pattern in a matrix *Temp* (and also *Output*) of size 4×4. Before the barrier (lines 7 and 14 in Algorithm 2) each thread ($tidI$, $tidJ$) has read permission in location ($tidI$, $tidJ$) in *Temp* (and also *Output*). In the barrier, the permission pattern changes such that each thread ($tidI$, $tidJ$) has write permission to location ($tidJ$, $tidI$). In this way, each thread ($tidI$, $tidJ$) can read its value from location ($tidI$, $tidJ$) (before the barrier) and write that value into location ($tidJ$, $tidI$) (after the barrier) safely.

5.3 Functional Correctness

Next, we discuss functional correctness of the parallel summed-area table algorithm. The approach to verify this algorithm is the same as before. First of all, we define two ghost variables: *inp_seq* is a sequence of sequences that captures the elements in *Input*, and *tmp1_seq* stores the inclusive prefix sum of elements in *Input* (see Fig. 6: step 1).

After applying the first prefix sum function, Properties 1 and 2 from Table 1 hold (line 5 in Algorithm 2). Property 1 specifies that *tmp1_seq* contains the inclusive prefix sum of the elements in *inp_seq*[14]. Property 2 shows the relation between *tmp1_seq*, and the actual array, *Temp*. We obtain these properties from the postconditions of the verified inclusive prefix sum (see [23]).

Now, we define a mathematical function *transpose*, as shown in Listing 4. This function computes the transposition of a sequence of sequences. The *transpose* function creates a sequence for each column i (starting from 0) as a new row i in the result, using a helper function (*transpose_helper*) to collect the *ith* elements of each row (in the input sequence)[15]. The preconditions of both functions

[14] The `ipsum` operation of a sequence returns an inclusive prefix sum of that sequence.
[15] Both functions are recursive and they are invoked with i and k equal to zero.

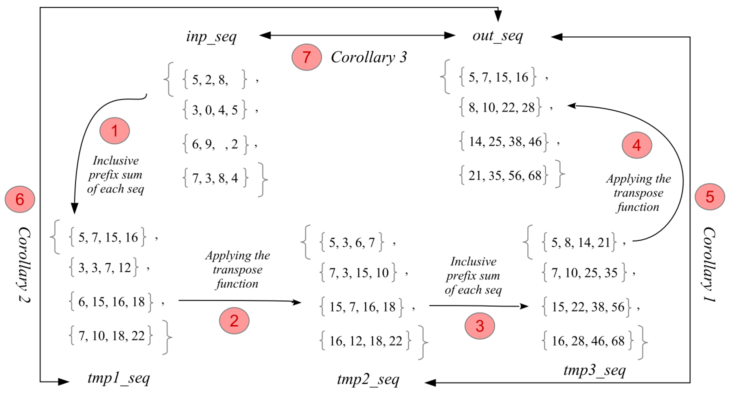

Fig. 6. An example of phases in the summed-area table algorithm over sequences as ghost variables. The sequences and functions capture the same arrays and computations as in Fig. 4.

specify the length of the input sequence and the range of the parameter's integer variables (lines 1–2 and 10–11). The postcondition of the *transpose_helper* function (lines 12–13) indicates that the result has the same size as each row and the return sequence contains the *ith* element of each row in the input sequence. The postcondition of the *transpose* function (lines 3–6) indicates that the result has the same size and indeed is the transposition of the input sequence.

Table 1. List of all properties in the summed-area table algorithm. Even numbers indicate the relation between the ghost and concrete variables. Odd numbers shows the relation between the ghost variables before and after each phase of the algorithm.

No.	Mathematical description of properties
1	$(\forall i.0 \le i < \lvert inp\_seq\rvert$: $(\forall j.0 \le j < \lvert inp\_seq[i]\rvert$: $tmp1\_seq[i][j] = ipsum(inp\_seq[i])[j] = \sum_{t=0}^{j} inp\_seq[i][t]))$
2	$(\forall i.0 \le i < \lvert tmp1\_seq\rvert$: $(\forall j.0 \le j < \lvert tmp1\_seq[i]\rvert$: $tmp1\_seq[i][j] = Temp[i][j]))$
3	$(\forall i.0 \le i < \lvert tmp1\_seq\rvert$: $(\forall j.0 \le j < \lvert tmp1\_seq[i]\rvert$: $tmp2\_seq[i][j] = tmp1\_seq[j][i]))$
4	$(\forall i.0 \le i < \lvert tmp2\_seq\rvert$: $(\forall j.0 \le j < \lvert tmp2\_seq[i]\rvert$: $tmp2\_seq[i][j] = Temp[i][j]))$
5	$(\forall i.0 \le i < \lvert tmp2\_seq\rvert$:$(\forall j.0 \le j < \lvert tmp2\_seq[i]\rvert$: $tmp3\_seq[i][j] = ipsum(tmp2\_seq[i])[j] = \sum_{t=0}^{j} tmp2\_seq[i][t]))$
6	$(\forall i.0 \le i < \lvert tmp3\_seq\rvert$: $(\forall j.0 \le j < \lvert tmp3\_seq[i]\rvert$: $tmp3\_seq[i][j] = Output[i][j]))$
7	$(\forall i.0 \le i < \lvert tmp3\_seq\rvert$: $(\forall j.0 \le j < \lvert tmp3\_seq[i]\rvert$: $out\_seq[i][j] = tmp3\_seq[j][i]))$
8	$(\forall i.0 \le i < \lvert out\_seq\rvert$: $(\forall j.0 \le j < \lvert out\_seq[i]\rvert$: $out\_seq[i][j] = Output[i][j]))$

We apply the *transpose* function to *tmp1_seq* right after the first transposition computation of the algorithm (line 10 in Algorithm 2). We store the result in a different sequence as *tmp2_seq*. Figure 6: step 2, illustrates this phase of the algorithm over the sequences. From the postcondition of the *transpose* function we have Properties 3 and 4 (Table 1) in line 11 of Algorithm 2. Property 3 shows that *tmp2_seq* is the transposition of *tmp1_seq* and Property 4 relates the ghost variable, *tmp2_seq* to the actual array *Temp*.

Then we have the second prefix sum function to compute the inclusive prefix sum of elements in *Temp* and store it in *Output*. We define a ghost variable, *tmp3_seq*, to store the inclusive prefix sum of elements in *tmp2_seq*, which is the same as *Temp* (lines 14 in Algorithm 2). Figure 6: step 3, shows this phase against the ghost variables. From the postcondition of the verified inclusive prefix sum (see [23]), it follows that Properties 5 and 6 (Table 1) hold in line 15 of Algorithm 2. Property 5 specifies that *tmp3_seq* contains the inclusive prefix sum of the elements in *tmp2_seq*. Property 6 shows the relation between *tmp3_seq* and *Output*.

The last phase of the algorithm is the second transposition, but this time for *Output*. Therefore, we now apply the transposition function to the *tmp3_seq* ghost variable and store the result in another ghost variable, *out_seq* in line 20 of Algorithm 2 (see Fig. 6: step 4). At this point (line 21 in Algorithm 2) we have Properties 7 and 8 (Table 1). Property 7 indicates that *out_seq* is the transposition of *tmp3_seq* and Property 8 relates *out_seq* to *Output*.

As we can see in Property 8, we relate the final result between the ghost variable, *out_seq* and the actual array *Output*, but we still should reason about the values in *out_seq* (and correspondingly *Output*). To accomplish this, we prove several corollaries following from the properties in Table 1:

Corollary 1. *From Properties 7 and 5 we have:*

$$(\forall i. 0 \le i < |out\_seq| : (\forall j. 0 \le j < |out\_seq[i]|: out\_seq[i][j] = \sum_{t=0}^{i} tmp2\_seq[j][t])).$$

Corollary 2. *From Corollary 1 and Property 3 we have:*

$$(\forall i. 0 \le i < |out\_seq| : (\forall j. 0 \le j < |out\_seq[i]|: out\_seq[i][j] = \sum_{t=0}^{i} tmp1\_seq[t][j])).$$

Corollary 3. *From Corollary 2 and Property 1 we have:*

$$(\forall i. 0 \le i < |out\_seq| : (\forall j. 0 \le j < |out\_seq[i]|: \\ out\_seq[i][j] = \sum_{t_1=0}^{i} \sum_{t_2=0}^{j} inp\_seq[t_1][t_2])).$$

Corollary 1 relates *out_seq* to *tmp2_seq* (Fig. 6: step 5). Corollary 2 shows the relation between the ghost variables *out_seq* and *tmp1_seq* (Fig. 6: step 6). Corollary 3 proves the relation between *inp_seq* and *out_seq* (Fig. 6: step 7). As *inp_seq* and *out_seq* are the same as *Input* and *Output* (Property 8), respectively, we conclude functional correctness.

In this verification, we could easily reuse the specification of the verified prefix sum algorithm in a straightforward way without proving more properties. As a consequence, the verification of the summed-area table algorithm takes much less time than the verification of the prefix sum algorithm. Moreover, we verified the parallel transposition, which is also a primitive operation in GPGPUs, and thus its verification can be reused for the verification of other algorithms that use this operation.

6 Conclusion

In this paper, we have proven data race-freedom and functional correctness of the parallel stream compaction and summed-area table algorithms, for an arbitrary input size by encoding the algorithms into the VerCors verifier. The two algorithms are widely used as primitive operations in other algorithms (e.g., collision detection and sparse matrix compression). Proving functional correctness of both algorithms is challenging because both use other parallel algorithms as sub-routine (e.g., prefix sum and transposition). To overcome these challenges, we reuse previous work on the verification of parallel prefix sum algorithms. It is straightforward to reuse the verification of prefix sum in the summed-area table algorithm. However, the transposition operation is used intermittently in addition to the prefix sum sub-routine. We should establish a formal relation between each of these intermediate steps in order to reason about the final result. In the stream compaction algorithm, since the input to the prefix sum sub-routine is a flag array, we should prove more properties of the prefix sum. Moreover, we define ghost variables and suitable functions that mimic the actual computations in the algorithms.

The complete verification of both algorithms took 2 weeks, whereas in comparison the verification of prefix sum took a couple of months. This shows that less effort is needed to verify complicated algorithms by reusing verified subroutines in practice. Therefore, we believe that now it will be a minimal effort to verify even more complex algorithms that are built on top of the stream compaction and summed-area table algorithms. As future work, we would like to investigate how a substantial part of the required annotations, in particular those related to permissions, can be generated automatically. In addition, based on our verifications, we plan to develop a library of general properties for common GPGPU sub-routines in VerCors.

References

1. Amighi, A., Haack, C., Huisman, M., Hurlin, C.: Permission-based separation logic for multithreaded Java programs. LMCS **11**(1), 1–66 (2015)
2. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: OOPSLA, pp. 113–132. ACM (2012)
3. Billeter, M., Olsson, O., Assarsson, U.: Efficient stream compaction on wide SIMD many-core architectures. In: Proceedings of the Conference on High Performance Graphics, vol. 2009, pp. 159–166 (2009)
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. 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)
10. 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
11. Crow, F.C.: Summed-area tables for texture mapping. In: Proceedings of the 11th annual conference on Computer graphics and interactive techniques, pp. 207–212 (1984)
12. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with CUDA. GPU Gems **3**(39), 851–876 (2007)
13. Hensley, J., Scheuermann, T., Coombe, G., Singh, M., Lastra, A.: Fast summed-area table generation and its applications. In: Computer Graphics Forum, vol. 24, pp. 547–555. Wiley Online Library (2005)
14. Horn, D.: Stream reduction operations for GPGPU applications. GPU Gems **2**(36), 573–589 (2005)
15. 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
16. 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)
17. 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)
18. 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)
19. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
20. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
21. Nvidia: Cuda-memcheck: User manual (version 10) (2019). https://developer.nvidia.com/cuda-memcheck
22. Price, J., McIntosh-Smith, S.: Oclgrind: an extensible OpenCL device simulator. In: Proceedings of the 3rd International Workshop on OpenCL, p. 12. ACM (2015)

23. Safari, M., Oortwijn, W., Joosten, S., Huisman, M.: Formal verification of parallel prefix sum. In: Lee, R., Jha, S., Mavridou, A. (eds.) NFM 2020. LNCS, vol. 12229, pp. 170–186. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-55754-6_10
24. Sengupta, S., Lefohn, A., Owens, J.: A work-efficient step-efficient prefix sum algorithm. In: Proceedings of the Workshop on Edge Computing Using New Commodity Architecture, May 2006
25. 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)