Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Formal verification of parallel prefix sum and stream compaction algorithms in CUDA ☆

Mohsen Safari *, Marieke Huisman

Formal Methods and Tools, University of Twente, Enschede, the Netherlands

A R T I C L E   I N F O

Article history:
Received 16 April 2021
Received in revised form 11 November 2021
Accepted 23 February 2022
Available online xxxx

Keywords:
GPU verification
CUDA
Deductive verification
Separation logic
Prefix sum
Stream compaction

A B S T R A C T

GPUs are an important part of any High Performance Computing (HPC) architecture. To make optimal use of the specifics of a GPU architecture, we need programming models that naturally support the parallel execution model of a GPU. CUDA and OpenCL are two widely used examples of such programming models. Furthermore, we also need to redesign algorithms such that they adhere to this parallel programming model, and we need to be able to prove the correctness of these redesigned algorithms.

In this paper we study two examples of such parallelized algorithms, and we discuss how to prove their correctness (data race freedom and (partial) functional correctness) using the VerCors program verifier. First of all, we prove the correctness of two parallel algorithms solving the prefix sum problem. Second, we show how such a prefix sum algorithm is used as a basic block in a stream compaction algorithm, and we prove correctness of this stream compaction algorithm, taking advantage of the earlier correctness proof for the prefix sum algorithm.

The proofs as described in this paper are developed over the CUDA implementations of these algorithms. In earlier work, we had already shown correctness of a more high-level version of the algorithm. This paper discusses how we add support to reason about CUDA programs in VerCors, and it then shows how we can redo the verification at the level of the CUDA code. We also discuss some practical challenges that we had to address to prove correctness of the actual CUDA-level verifications.

© 2022 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

As software is becoming an integral part of our daily lives, the development of techniques to ensure the reliability and correctness of this software is essential. In particular with the advent of multi-core and many-core hardware such as Graphics Processing Units (GPUs), this has become even more challenging. GPUs naturally support parallel execution, with many threads cooperating together, executing the same instructions, but on different data (known as the Single Instruction Multiple Data (SIMD) programming model). This parallel execution can substantially improve the performance, but also provides a risk for the reliability of the software. This paper illustrates how program verification can be used to provide such correctness guarantees on some well-known parallel GPU algorithms.

☆ This work is supported by NWO grant 639.023.710 for the Mercedes project.

* Corresponding author.
E-mail addresses: m.safari@utwente.nl (M. Safari), m.huisman@utwente.nl (M. Huisman).

https://doi.org/10.1016/j.tcs.2022.02.027
0304-3975/© 2022 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Please cite this article as: M. Safari and M. Huisman, Formal verification of parallel prefix sum and stream compaction algorithms in CUDA, Theoretical Computer Science, https://doi.org/10.1016/j.tcs.2022.02.027

More concretely, this paper contributes the following. First of all, we introduce CUDA [21], a well-known programming model for GPU architectures, and we discuss how we add verification support for CUDA programs in the VerCors program verifier [6]. The VerCors program verifier provides support for deductive verification of concurrent programs, using permission-based separation logic. In particular, VerCors support a GPU-specific variant of permission-based separation logic [7]. The VerCors verifier already supported this logic for its own built-in language, called PVL. This paper discusses how we extend this support to the commonly used CUDA programming framework, and how it can be used to prove memory and thread safety (i.e., absence of data races[1]), as well as (partial) functional correctness of CUDA implementations.[2]

Second, we illustrate how this support to reason about CUDA programs is used to verify the CUDA implementation of some well-known parallel (GPU-based) algorithms. We first discuss the verification of two algorithms that provide a solution to the prefix sum problem [10,17,5,27]. These algorithms take an array of integers and, for each element, they compute the sum of the previous elements. Algorithms that solve the prefix sum problem are an important building stone for other parallel algorithms, such as stream compaction, summed-area table, radix sort, quick sort, etc; see Blelloch [5]. In addition, we also discuss the verification of one algorithm that uses such a prefix sum algorithm, namely stream compaction [15,14,26,4], and for which parallel versions are known to outperform their sequential (CPU-based) counterparts. The verification of the stream compaction algorithm crucially depends on the verification of the prefix sum algorithm. Stream compaction reduces an input array to a smaller array by removing undesired elements, and many other 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 the CPU and GPU is frequent.

There are only a few approaches that support reasoning about GPU programs, see e.g. [7,19,12,18,3]; most of these focus on finding data races. In general, proving functional correctness of parallel GPU programs is a difficult task due to the hierarchical structure of threads and different levels of memory accesses. In particular, in the verifications discussed in this paper, we had to address several challenges, for which we needed the powerful tool support as provided by the VerCors program verifier. First, both prefix sum algorithms are in-place, i.e. we need to reason about values that are unstable and that change during the algorithm. Moreover, the computational pattern of the prefix sum algorithms makes it complex to reason about the final result, and it was a challenge to find suitable properties that relate the internal computation steps in the algorithms to the final result. For the verification of the stream compaction algorithm, the main challenge was to take advantage of the specification of the prefix sum algorithm. In particular, the input of the prefix sum is of a restricted format, the input array just contains flags and therefore the output can be used as indices of elements in another array. To take advantage of this, several additional properties had to be proved to show that the prefix sum result can indeed be safely used as array indices.

To summarize, this paper provides the following contributions:

- Support to reason about CUDA programs added to the VerCors program verifier
- A correctness proof for both data race freedom and functional correctness of two CUDA implementations of parallel algorithms that solve the prefix sum problem.
- A correctness proof for both data race freedom and functional correctness of a parallel stream compaction algorithm, implemented in CUDA, and using one of the prefix sum algorithms.

To the best of our knowledge, this is the only *tool-supported* verification of data race freedom and functional correctness of the prefix sum and stream compaction algorithms, implemented in CUDA.

This paper is an extended version of our papers presented at NFM 2020 [25] and ICTAC 2020 [24]. In the NFM 2020 paper we proved the correctness of the two parallel prefix sum algorithms at the pseudocode level. In the ICTAC 2020 paper we proved the parallel stream compaction algorithm at the pseudocode level. In this paper, we add CUDA verification support to the VerCors program verifier, and then redo the verification for the actual CUDA implementations.

This paper is organized as follows. After some background information on GPUs, CUDA and the VerCors verifier, Section 3 describes how we add support for CUDA verification in VerCors. Then, Section 4 shows how we prove correctness of two parallel prefix sum algorithms, implemented in CUDA. Section 5 presents the verification of stream compaction algorithm, also implemented in CUDA. Finally, Section 6 presents related work, and Section 7 concludes the paper, and discusses future work.

2. Background

This section first explains GPU hardware and the CUDA programming platform briefly. Then it describes the VerCors verifier and its underlying logic.

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

[2] This paper does not consider termination proofs of CUDA programs.

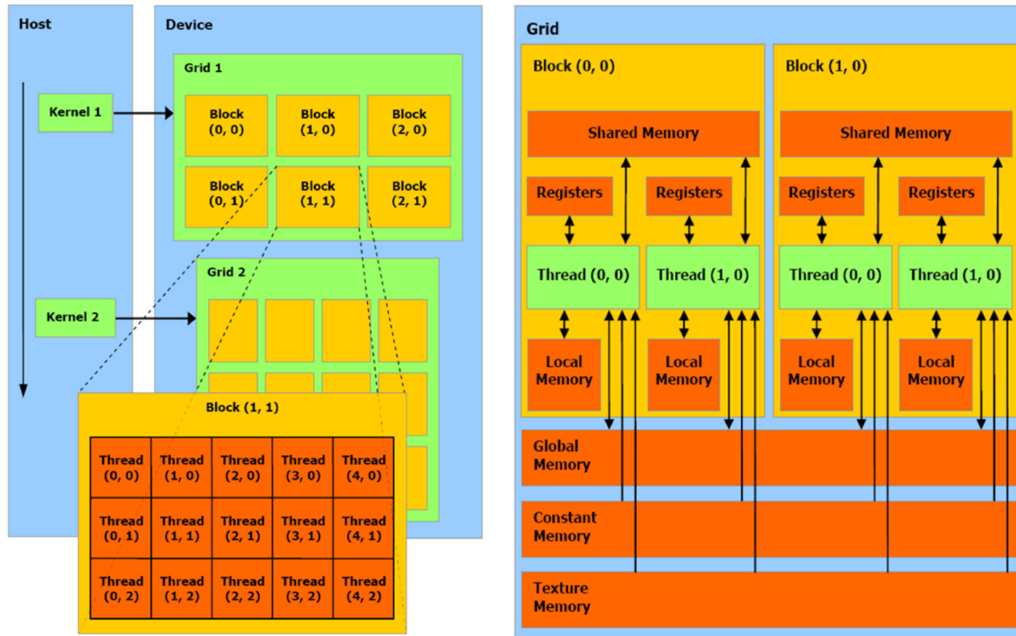

Fig. 1. GPGPU architecture and programming model.

2.1. GPU and CUDA programming

Fig. 1 gives an overview of the GPU architecture and its programming model.[3] GPUs have many simpler, but more efficient cores than CPUs that can run many threads simultaneously. A GPU has several MultiProcessors (MPs) and each MP has some Streaming Processors (SPs). GPUs have a hierarchical memory structure. The fastest memory is a register, which is local to a single thread. Each MP has a shared memory, which is the second-fastest memory. Threads from one block cannot access the shared memory of other blocks. Finally, the slowest memory is global memory. This is accessible by all threads and by the CPU and thus can be used for communication between all threads and from and to the host.

CUDA allows software developers to implement parallel algorithms on NVIDIA GPUs. A CUDA program is a CPU-GPU program. It means part of the program runs on CPU and the other part runs on the GPU. The CPU part is called *host* and the GPU part is called *kernel*. The typical workflow in a CUDA program is that we copy data from the host to the *device* (GPU), and then, we call the kernel from the host, running on the device. Each thread executes the kernel code on its own part of the data. Finally, we copy data back from the device to the host.

In the CUDA programming model, programmers can define threads in a hierarchical level as grids and blocks. A block indicates a number of threads running on one MP. A grid shows the total number of blocks to run on one GPU. Both grids and blocks can be defined in one, two or three dimensions by the programmer; however, the GPU scheduler decides how to assign thread blocks to MPs. CUDA provides a mechanism for barrier synchronization amongst the threads within a block, but there is no programming primitive for inter-block synchronization. Thus, inter-block synchronization can be achieved using the host side. In addition, there are also primitive atomic operations which can be used to avoid read/write inconsistencies in global and shared memories.

2.2. VerCors program verifier

VerCors is a program verifier to specify and verify concurrent and parallel programs. It supports high-level languages such as (subsets of) Java, CUDA, OpenCL, OpenMP and PVL, where PVL is VerCors' internal language for prototyping new features. VerCors can be used to verify memory safety and functional correctness of programs.

Fig. 2 shows the high-level architecture of VerCors. To use VerCors, programs should be annotated with pre/post-conditions using permission-based separation logic [9,2,8]. Then, VerCors encodes the annotated programs via several program transformation steps into the intermediate representation language (Silver) of the Viper framework [20,28]. Finally the back end of Viper (Silicon) translates the annotated program into proof obligations which are sent to an automated theorem prover; in our case Z3 [13].

[3] The figure is taken from NVIDIA CUDA C Programming Guide: http://docs.nvidia.com/cuda/-cuda-c-programming-guide/index.html.

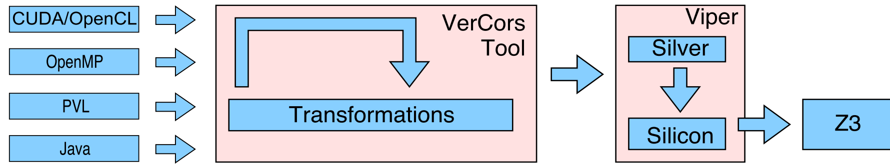

Fig. 2. VerCors tool set overall architecture.

List 1 Parallel left rotation in PVL.

```
1  context_everywhere array != NULL && array.length == size;
2  requires (\forall* int i; i >= 0 && i < size; Perm(array[i], 1));
3  ensures (\forall* int i; i >= 0 && i < size; Perm(array[i], 1));
4  ensures (\forall int i; i >= 0 && i < size; (i != size-1 ==> array[i] == \old(array[i+1]))
5          && (i == size-1 ==> array[i] == \old(array[0])) );
6  void leftRotation(int[] array, int size){
7    par threadBlock (int tid = 0 .. size)
8    requires tid != size-1 ==> Perm(array[tid+1], 1\2);
9    requires tid == size-1 ==> Perm(array[0], 1\2);
10   requires Perm(array[tid], 1\2);
11   ensures Perm(array[tid], 1);
12   ensures tid != size-1 ==> array[tid] == \old(array[tid+1]);
13   ensures tid == size-1 ==> array[tid] == \old(array[0]);
14   {
15     int temp;
16     if (tid != size-1){ temp = array[tid+1]; }
17     else{ temp = array[0]; }
18     barrier(threadBlock)
19       requires tid != size-1 ==> Perm(array[tid+1], 1\2);
20       requires tid == size-1 ==> Perm(array[0], 1\2);
21       requires Perm(array[tid], 1\2);
22       ensures Perm(array[tid], 1);
23       {}
24     array[tid] = temp;
25   }
26 }
```

In the annotations, 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 [9]): 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. The soundness of the program logic ensures that in each memory location, the total number of permissions among all threads accessing this location does not exceed 1. Thus, verified programs ensure the absence of data races. The next section gives an example that shows how to specify and verify a program using the VerCors verifier.

3. CUDA verification in VerCors

This section first shows how to verify parallel programs written in PVL using VerCors. Then, we discuss how we add support to reason about CUDA programs to VerCors, by transforming CUDA programs into the parallel programming constructs of PVL. Finally, we present an example verification of a CUDA program using VerCors.

3.1. Parallel PVL verification

To verify parallel algorithms, PVL defines several parallel programming constructs (i.e., parallel blocks, barriers, atomics, etc). List. 1 gives an example of a parallel block that uses a barrier to synchronize the threads. It contains a function named “leftRotation” that rotates the elements of an array to the left. Inside the function, there is a parallel block named “threadBlock” (lines 7-25). The keyword `par` is used to define the parallel block, followed by an arbitrary name for the block. The par block first specifies the number of threads in the parallel block, as well as a name for the thread identifier. In this example, we have “size” threads in the range from 0 to “size-1” and “tid” is used to refer to each thread (line 7).

The body of the parallel block is executed by each thread. Therefore, each thread (“tid”) stores its right neighbor in a temporary location (i.e., “temp”), except thread “size-1” which stores the first element in the array (lines 16-17). Then all

List 2 General template of CUDA thread hierarchies in PVL.

```
1
2  requires (\forall* int i; i >= 0 && i < gcount;
3           (\forall* int j; j >= 0 && j < bsize; pre(i, j)));
4  ensures (\forall* int i; i >= 0 && i < gcount;
5           (\forall* int j; j >= 0 && j < bsize; post(i, j)));
6  void HostCode(){
7    ...
8    par Grid (int gid = 0 .. gcount)
9    requires (\forall* int k; k >= 0 && k < bsize; pre(gid, k));
10   ensures (\forall* int k; k >= 0 && k < bsize; post(gid, k));
11   {
12     par Block (int tid = 0 .. bsize)
13     requires pre(tid);
14     ensures post(tid);
15     {
16       ...
17     }
18   }
19 }
```

threads synchronize at the barrier (line 18). The keyword `barrier` and the name of the parallel block as an argument (e.g., "threadBlock" in the example) are used to define a barrier in PVL. After that, each thread writes the value read before the barrier into its own location at index "tid" in the array (line 24).

To verify this function in VerCors, we annotate the barrier, in addition to the function and the parallel block. To specify permissions, we use predicate `Perm(L, π)` 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, 8-13), must hold at the beginning and the end of the function (or par block), respectively. The keyword `context_everywhere` is used to specify an invariant (line 1) that must hold throughout the function (including the par block). As precondition of the function, we have write permission over all locations in the array (line 2). At the beginning of the parallel block, each thread reads from its right neighbor, except thread "size-1" which reads from location 0 (lines 16-17). Therefore, we specify read permissions as precondition of the parallel block in lines 8-9. Since after the barrier each thread ("tid") writes into its own location at index "tid", we change the permissions in the barrier such that each thread has write permissions to its own location (lines 19-22). When a thread reaches the barrier, it has to fulfill the barrier preconditions, and then it may assume the barrier postconditions. Moreover, the barrier postconditions must follow from the barrier preconditions. Therefore, each thread has initially read permission in its own location as well (line 10). As a result, the accumulation of available permissions in each location of the array is 1 and we can change it into write permission in the barrier. Note that the body of the barrier is empty (line 23).

As postcondition of the parallel block (1) first each thread has write permission to its own location (this comes from the postcondition of the barrier) in line 11 and (2) the elements are truly shifted to the left (lines 12-13). From the postcondition of the parallel block, we can establish the corresponding postcondition for the function (lines 3-5). 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). Moreover, `\forall*` indicates universal separating conjunction over permission predicates and `\forall` denotes standard universal conjunction over logical predicates.

3.2. CUDA verification

CUDA to PVL transformation To be able to verify CUDA programs in VerCors, we transform a CUDA program into a PVL program internally in the tool. The main challenge to support verification of CUDA programs is to handle the hierarchical structure to define threads in CUDA (i.e., grids and blocks). To address this, we map a CUDA kernel into two nested PVL parallel blocks. List. 2 illustrates the general template resulting from the transformation.

As we can see, the outer parallel block (i.e., Grid) indicates the number of blocks in a grid and the inner one (i.e., Block) indicates the number of threads per block. The two nested parallel blocks encode the CUDA kernel that runs on a GPU and everything outside the two nested parallel blocks encodes the host code that runs on the CPU. To verify the CUDA kernel part, the user only needs to add thread-level annotations for the inner parallel block. The specifications for the outer block are inferred by VerCors as separation conjunction over all threads in a block (lines 9-10 in List. 2). Similarly, the specification for the whole kernel is inferred by VerCors as separation conjunction over all thread blocks in the grid (lines 2-5 in List. 2).

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

List 3 Parallel left rotation in CUDA.

```
1  /*@ context_everywhere array != null && array.length == size;
2      requires threadIdx.x != size-1 ? \pointer_index(array, threadIdx.x+1, 1\2)
3                                : \pointer_index(array, 0, 1\2);
4      requires \pointer_index(array, threadIdx.x, 1\2)
5      ensures \pointer_index(array, threadIdx.x, 1);
6      ensures threadIdx.x != size-1 ==> array[threadIdx.x] == \old(array[threadIdx.x+1]);
7      ensures threadIdx.x == size-1 ==> array[threadIdx.x] == \old(array[0]); @*/
8  __global__ void CUDALeftRotation(int *array, int size) {
9    int temp;
10   int tid = threadIdx.x;  // get the thread id
11   if (tid != size-1) { temp = array[tid+1]; } else { temp = array[0]; }
12
13   /*@ requires tid != size-1 ? \pointer_index(array, tid+1, 1\2)
14                            : \pointer_index(array, 0, 1\2);
15       requires \pointer_index(array, threadIdx.x, 1\2)
16       ensures \pointer_index(array, tid, 1); @*/
17   __syncthreads();
18   array[tid] = temp;
19 }
```

Moreover, the barrier and atomic operations in CUDA are transformed into the corresponding ones in PVL. Note that the barrier always synchronizes threads inside the inner block. We refer to [7,1] for more details about reasoning of GPU programs with barriers and atomic operations.

CUDA verification example List. 3 shows how we verify the kernel part of the same left rotation program written in CUDA. We assume there is only one thread block in the grid and there are "size" threads in that block. As there are pointers in CUDA, we use `\pointer_index`(S, idx, π) in the specification to specify permission over a specific location *idx* that pointer *S* points to.[5] This CUDA example is transformed in PVL, which returns (in essence) the same annotated PVL program as in List. 1.

CUDA verification challenges A major challenge that we encounter when we verify CUDA programs is the abundance of nested quantified expressions in the specification, because the extracted pre- and postconditions of the grids and kernel are quantified over the number of threads per block and the number of blocks in a grid (lines 2-5 in List. 2). These nested universal quantifiers make the reasoning about the generated proof obligations more difficult for the underlying theorem prover Z3.

To mitigate this problem, we had to extend the VerCors tool implementation with many simplification rules that during the transformation process syntacticly replace these complicated expressions by simpler ones. For instance, for an array *arr* the tool applies a simplification rule that replaces

$$(\backslash forall * int\ i; i >= 0\ \&\&\ i < gcount;\ (\forall * int\ j; j >= 0\ \&\&\ j < bsize;\ \texttt{Perm}(arr, bsize \times i + j)))$$

by

$$bsize > 0 \Rightarrow (\backslash forall * int\ i; i >= 0\ \&\&\ i < gcount \times bsize;\ \texttt{Perm}(arr, i)).$$

The situation becomes even more complicated when there are non-linear arithmetic access patterns to an array (e.g., $2 \times tid + 1$), which requires many specialized simplification rules to be added, for all different access patterns. It is future work to investigate if we can find more general simplification rules that can address a large range of array access patterns.

4. Prefix sum

In this section, we explain the prefix sum problem and discuss two parallel solutions to this problem implemented in CUDA. Then, we show how to verify data race-freedom and functional correctness of both algorithms. Instead of presenting the full specification, we explain the main ideas and verification steps.[6] We end the section with a discussion of verification challenges that were introduced by actually verifying the CUDA implementation, rather than the PVL pseudocode.

[5] We use the predicate `\pointer`$((S_0, \ldots, S_n), \ell, \pi)$ to indicate that all array references $S_0, \ldots, S_n$ have length ℓ, and that the current thread has permission $\pi \in (0, 1]$ for them.

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

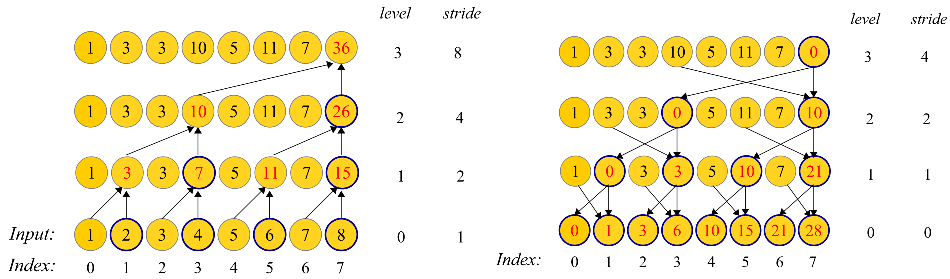

Fig. 3. After the up-sweep phase (left) and the down-sweep phase (right) in Blelloch's algorithm (two arrows coming to a circle indicate summation and one arrow indicates replacement, red color values show the effect of computations and circles with thick border are *indicators* as in Algorithm 1). (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

4.1. Prefix sum problem

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. The prefix sum problem is to find an algorithm such that 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 $0 \le i < N$.

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 $0 \le i < N$.

Blelloch [5] introduced an *exclusive parallel in-place* algorithm that solves the prefix sum problem. Kogge-Stone [17] proposed an *inclusive parallel in-place prefix sum* algorithm. These two parallel versions are frequently used in practice (for example as a primitive operation in libraries AMD APP SDK,[7] and NVIDIA CUDA SDK[8]).

4.2. Blelloch's parallel prefix sum

Blelloch's algorithm [5] consists of two phases: up-sweep and down-sweep. Fig. 3 illustrates both the up and down-sweep phases visually, and Algorithm 1 shows the implementation of the in-place algorithm in CUDA. The up-sweep phase is implemented in lines 3-8 of the algorithm and the down-sweep phase is implemented in lines 11-20. Each iteration in the up/down phases in Algorithm 1 (lines 3-8/11-20) corresponds to a different level in Fig. 3. We assume 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 which is updated in both phases (lines 8 and 20). 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, two nodes are summed up at each level (line 5). As a result, the last element at the highest level is the sum of the input values. In the down-sweep phase, we first set the last element to 0 (lines 11-12). Then, we use the partial sums calculated during the up-sweep to compute the prefix sum of the input. The complete prefix sum is computed as the lowest level of the down-sweep (lines 14-17). Note that in order to synchronize threads at each level of both phases, a barrier is needed (lines 6 and 18). There is also a barrier between up-sweep and down sweep (line 9).

4.2.1. 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. Fig. 4 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 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 the red color in Fig. 4 (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.

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

8 https://developer.nvidia.com/gpu-computing-sdk.

Algorithm 1 Blelloch's Prefix Sum Algorithm in CUDA.

```
1: __global__ void exclusivePrefixsumKernel(int* Input, int* Output, int N)
2: int tid = threadIdx.x; int indicator = 2 × tid + 1; int stride = 1;
3: while stride < N do
4:     if indicator < N && indicator ≥ stride then
5:         Output[indicator] = Output[indicator] + Output[indicator − stride];
6:     __syncthreads();
7:     indicator = 2 × indicator + 1;
8:     stride = 2 × stride;
9: __syncthreads();
10: indicator = N × tid + N − 1; stride = N / 2;
11: if indicator < N then
12:     Output[indicator] = 0;
13: while stride ≥ 1 do
14:     if indicator < N && indicator ≥ stride then
15:         int temporary = Output[indicator];
16:         Output[indicator] = Output[indicator] + Output[indicator − stride];
17:         Output[indicator − stride] = temporary;
18:     __syncthreads();
19:     indicator = (indicator − 1) / 2;
20:     stride = stride / 2;
```

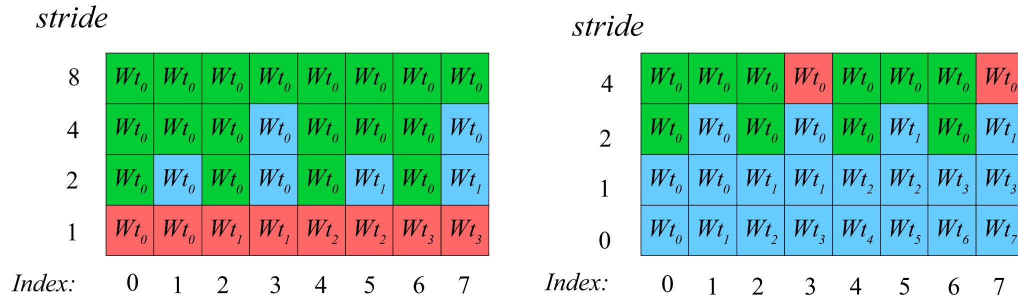

Fig. 4. Permission patterns for array of length 8: (left) up-sweep and (right) down sweep phases of Blelloch's algorithm (W_{t_i} indicates thread i has write permission, red color indicates initial permissions of active threads, blue shows changes in permission pattern and green shows lost permissions which assigned to thread 0.

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 9), 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 the up-sweep, thread 0 has write permission to all locations in the array.

In the down-sweep phase, Fig. 4 (right), we have the same permission pattern in reverse direction. In the down-sweep phase, 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 18). 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.

4.2.2. 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; i.e., the elements change in each iteration.
2. There are two phases, 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-

List 4 The *Build_full_history* function.

```
1  /*@ requires stride > 0 && stride < |f_hist_prev_lvl|;
2      ensures |\result| == |f_hist_prev_lvl|-i;
3      ensures (\forall int j; j≥0 && j<|\result|; ((i<|f_hist_prev_lvl|) &&
4              ((i+j)≥stride) && (((i+j)%(2×stride)) == (2×stride-1))) ==>
5              \result[j] == f_hist_prev_lvl[i+j] + f_hist_prev_lvl[i+j-stride]);
6      ensures (\forall int j; j≥0 && j<|\result|; ((i<|f_hist_prev_lvl|) &&
7              (((i+j)<stride) || (((i+j)%(2×stride)) != (2×stride-1)))) ==>
8              \result[j] == f_hist_prev_lvl[i+j]); @*/
9  static pure seq<int> Build_full_history(seq<int> f_hist_prev_lvl, int stride,
10     int i) = i<|f_hist_prev_lvl| ? (
11        ((i%(2×stride)) == (2×stride-1) && (i ≥ stride) ?
12          seq<int> {f_hist_prev_lvl[i] + f_hist_prev_lvl[i-stride]} +
13                Build_full_history(f_hist_prev_lvl, stride, i+1) :
14          seq<int> {f_hist_prev_lvl[i]} +
15                Build_full_history(f_hist_prev_lvl, stride, i+1) )) : seq<int> {};
```

sweep. If we look at the only values that change in Fig. 3 (the 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:
   (a) In the up-sweep, the sum of values that change in each iteration equals the sum of the values in the input array.
   (b) In the 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 the 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.

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. 4. 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. 5 (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*.

To update *p_hist*, which keeps only the values that change during the iterations, we define a *Build_partial_history* function (see List. 5). 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 summing 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. Fig. 5 (middle) shows the result of applying *Build_partial_history* to *p_hist_prev_lvl*.

stride	f_hist	p_hist	down_seq
8	{1, 3, 3, 10, 5, 11, 7, 36}	{36}	{0}
4	{1, 3, 3, 10, 5, 11, 7, 26},	{10, 26},	{0, 10}
2	{1, 3, 3, 7, 5, 11, 7, 15},	{3, 7, 11, 15},	{0, 3, 10, 21}
1	{1, 2, 3, 4, 5, 6, 7, 8},	{1, 2, 3, 4, 5, 6, 7, 8},	{0, 1, 3, 6, 10, 15, 21, 28}

Fig. 5. 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*.

List 5 The *Build_partial_history* function.

```
1  //@ requires |p_hist_prev_lvl| ≥ 0;
2  static pure seq<int> Build_partial_history(seq<int> p_hist_prev_lvl) =
3    1 < |p_hist_prev_lvl| ?
4      seq<int> {head(p_hist_prev_lvl) + head(tail(p_hist_prev_lvl))} +
5      Build_partial_history(tail(tail(p_hist_prev_lvl))) : p_hist_prev_lvl;
```

List 6 The *epsum* function.

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

Down-sweep ghost variables Next, in the down-sweep phase, 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 the down-sweep are in fact the exclusive prefix sum of the values changed in the up sweep in the corresponding iteration. To update *down_seq* in each iteration of down-sweep, we define a function, *epsum* (List. 6), 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. 5 (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.

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. 4 (lines 2-8). However we should also relate the output array and *p_hist* and this is more challenging. Fig. 6 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 are $Output[indicator] = p_hist[lvl-1][2 \times tid + 1]$ 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:

Property 1. *For any sequence xs:*
$(\forall i. 0 \le i < |xs|: Build_partial_history(xs)[i] = xs[2 \times i] + xs[2 \times i + 1])$.

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 the down-sweep phase are $Output[indicator] = down_seq[tid]$ and $Output[indicator - stride] =$

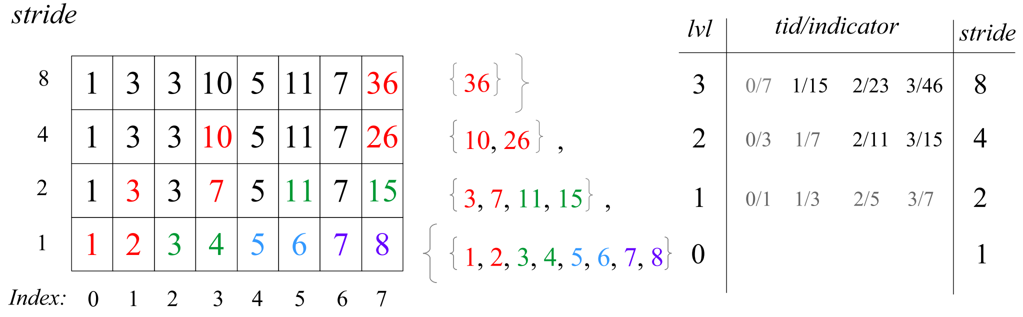

Fig. 6. Relation between Output (left) and *p_hist* (middle) according to active threads (grey color) in the table (right): $Output[indicator] = p_hist[lvl-1][2 \times tid + 1]$ and $Output[indicator - stride] = p_hist[lvl-1][2 \times tid]$ ($lvl > 0$).

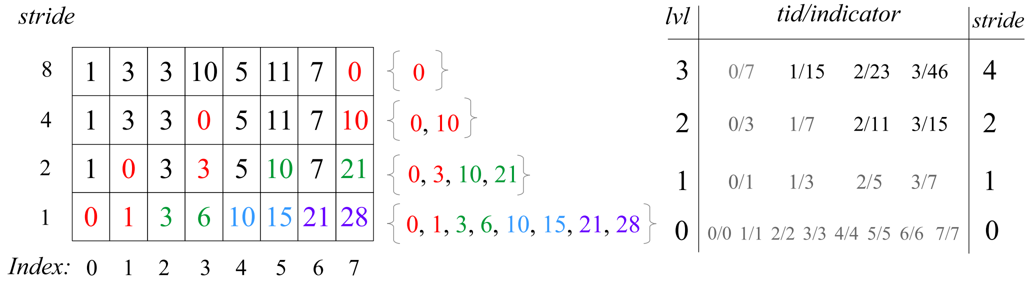

Fig. 7. Relation between the actual array, *Output*, (left) and the ghost variable, *down_seq* (middle) according to active threads (grey color) in the table (right).

$p_hist[lvl][2 \times tid]$ (see Fig. 7, 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:*
$(\forall i. 0 \leq i < |xs|/2: epsum(Build_partial_history(xs))[i] = epsum(xs)[2 \times i])$.

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

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 verified implementation for further proof details.[9]

4.3. Kogge-Stone's parallel prefix sum

In contrast to Blelloch's algorithm, Kogge-Stone's [17] algorithm consists of one phase. Algorithm 2 illustrates the encoding and Fig. 8 illustrates the algorithm visually. The levels in the figure correspond to the loop in lines 3-11 of the algorithm. In the figure, the lowest level is 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.

4.3.1. Data race-freedom

To verify data race freedom of this algorithm, we need to specify permissions over the output array. Fig. 9 shows the permission pattern in each iteration. 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. 9. 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. 9). 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

[9] The full specification is available at https://github.com/Safari1991/Prefixsum-StreamCompaction/blob/main/Blelloch.cu.

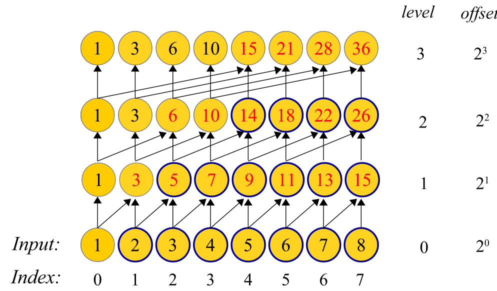

Fig. 8. Kogge-Stone's prefix sum algorithm (two arrows coming to a circle indicate summation and one arrow indicates replacement, red color values show the effect of computations and circles with thick border show $tid \geq offset$ as in Algorithm 2).

Algorithm 2 Kogge-Stone's Prefix Sum Algorithm in CUDA.

```
1: __global__ void inclusivePrefixsumKernel(int* Input, int* Output, int N)
2: int tid = threadIdx.x; int offset = 1;
3: while offset < N do
4:     int temp = Output[tid];
5:     if tid ≥ offset then
6:         temp = Output[tid − offset] + temp;
7:     __syncthreads();
8:     if tid ≥ offset then
9:         Output[tid] = temp;
10:    __syncthreads();
11:    offset = 2 × offset;
```

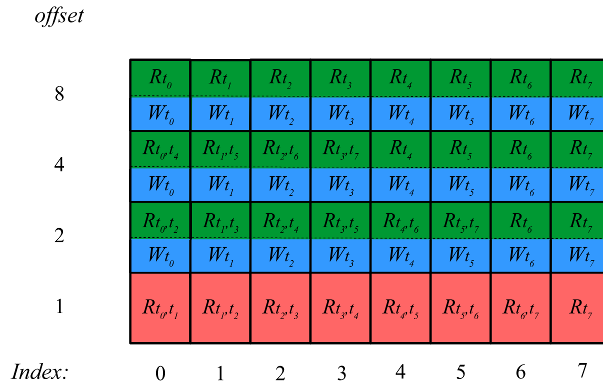

Fig. 9. Permissions in Kogge-Stone's algorithm; R_{t_i,t_j} indicates read permission by threads i and j, W_{t_i} indicates write permission by thread i, red color shows initial permissions, blue/green show how the permissions change in the first/second barrier.

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*).

4.3.2. Functional correctness

Next, we discuss how to verify functional correctness of the algorithm. The difference between this algorithm and the Blelloch's algorithm 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 several of 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 (as *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 in every iteration.

As we can see in Fig. 8, 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. 7. This function captures the same computation as in the algorithm. We can see from the postcondition of the function (lines 4-6 in List. 7) 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*.[10] 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 7 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.

4.4. Verification challenges for CUDA

During the verification of the CUDA implementations of the two prefix sum algorithms in VerCors, we encountered several challenges. First, after defining the ghost variables inside the kernel, we had to specify that the initial values in those sequences are the same as the input. Unfortunately, in our current version of the VerCors verifier, it is not possible to define pure functions to initialize the sequences according to the concrete input, because pure functions only can be used for sequences and not pointers and arrays. The second problem that we encountered is when we need to re-establish the relation between the ghost variables and concrete ones after a barrier. We prove the relation before the barrier, but as the permission pattern changes in the barrier, we should re-establish the relation again. Unfortunately, again with the current version of the tool, we cannot specify this in the barrier. The reason for this is that the current version of the VerCors verifier has a set-up for ghost variables that is not well-tailored to C programs (as CUDA is a variant of the C support in VerCors). There are no fundamental reasons, but adjusting this requires a major reorganization of the tool's internals. Therefore, as a temporary workaround we added a few explicit assumptions in the CUDA programs, which specify the relevant properties about the ghost variables. It should be stressed that when the PVL pseudocode version of the algorithm was verified, these properties all could be proven, thus we see no fundamental problem with adding those assumptions, it is just a practical temporary workaround.

As we use non-linear access patterns to an array (i.e., $2 \times tid$ and $2 \times tid + 1$) in the prefix sum algorithms, the underlying theorem prover Z3 has a hard time to prove or refute complicated proof obligations with nested quantifiers, as also mentioned above. However, to be able to benefit from the synchronization algorithm on the GPU, the two prefix sum examples are implemented in such a way that the entire algorithm resides in one kernel. The consequence of this design choice is that we can only have one thread block, which is restricted to a limited number of threads. That means, the input size is restricted to that limit. However, the advantage of this for verification is that instead of using $threadIdx.x + blockIdx.x \times blockDim.x$, we can use *threadIdx.x* as thread identifiers. This simplifies proof obligations and mitigates the problem in Z3. To be able to do this, we explicitly specify that the number of thread blocks is one in the contract of the kernel. In this way, we further simplify the generated proof obligations in Z3.

In general, we find that specifying the number of thread blocks and threads per block explicitly in the contract simplifies the complexity, as we can replace thread block variable (e.g., *gcount*) and size of blocks (*bsize*) by concrete values in the proof. Note that these are necessarily user-specified parameters when invoking a CUDA kernel.

5. Stream compaction

So far we verified two parallel prefix sum solutions, Blelloch's and Kogge-Stone's algorithms. Next, we prove the partial correctness of parallel stream compaction as an application of the prefix sum. To verify the stream compaction algorithm, we explain how to reuse the verified Blelloch's prefix sum algorithm. Again, we first show how to prove data race-freedom, and then we discuss functional correctness. We explain the main ideas mostly by using pictures instead of presenting the

[10] 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*).

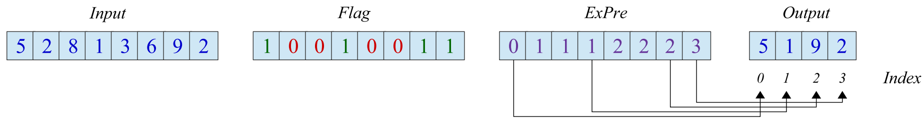

Fig. 10. An example of stream compaction of size 8.

Algorithm 3 Stream Compaction Algorithm in CUDA.

```
1: __global__ void streamCompactionKernel(int* Input, int* Output, int* Flag, int* ExPre, int N)
2: exclusivePrefixsumKernel(Flag, ExPre, N);
3: __syncthreads();
4: if Flag[tid] == 1 then
5:     Output[ExPre[tid]] = Input[tid];
```

full specification.[11] We end the section with a discussion of verification challenges that were introduced by actually verifying the CUDA implementation, rather than the PVL pseudocode.

5.1. Stream compaction problem

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. The stream compaction problem is to find an algorithm such that 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
  - $\forall j.\ 0 \leq j < M$: $Output[j] = t \Rightarrow \exists i.\ 0 \leq i < N$: $Input[i] = t \ \wedge \ Flag[i]$.
  - $\forall i.\ 0 \leq i < N$: $Input[i] = t \ \wedge \ Flag[i] \Rightarrow \exists j.\ 0 \leq j < M$: $Output[j] = t$.
  - $\forall i, j.\ 0 \leq i, j < N$: $(\ Flag[i] \ \wedge \ Flag[j] \ \wedge \ i < j \iff$ $(\exists k, l.\ 0 \leq k, l < M$: $Output[k] = Input[i] \ \wedge$ $Output[l] = Input[j] \ \wedge \ k < l)\ )$.

Algorithm 3 shows the pseudocode of the parallel algorithm and Fig. 10 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 (e.g., Blelloch's algorithm) 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 3, after which all the desired elements are stored in the output array (lines 4-5).

5.2. Data race-freedom

Again, to prove data race-freedom, we specify how threads access shared resources by adding permission annotations to the code. In Algorithm 3, we have several arrays that are shared among threads. There are three locations in the algorithm where permissions can be redistributed: before Algorithm 3 as preconditions, in the exclusive prefix sum function as postconditions and in the barrier (redistribution of permissions). Fig. 11 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 5 in Algorithm 3), we define each thread to have read permissions to its "own" location at index *tid* throughout the algorithm (Fig. 11). 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 3 must match the permission preconditions of the exclusive prefix sum function (i.e., Algorithm 1). Thus, following the preconditions of Algorithm 1, we define the permissions such that each thread (*tid*) has read permissions to its "own" location (Fig. 11: left). The exclusive prefix sum function returns the same permissions for *Flag* in its postconditions (Fig. 11: middle). Since, each thread needs read permission in line 4 of Algorithm 3, we keep the same permission pattern in the barrier (line 3) as well (Fig. 11: right).

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

Arrays \ Locations	At the beginning of the algorithm	After the exclusive prefix sum	After the barrier
Input	Rt_0 Rt_1 Rt_2 Rt_3 Rt_4 Rt_5 Rt_6 Rt_7	Rt_0 Rt_1 Rt_2 Rt_3 Rt_4 Rt_5 Rt_6 Rt_7	Rt_0 Rt_1 Rt_2 Rt_3 Rt_4 Rt_5 Rt_6 Rt_7
Flag	Rt_0 Rt_1 Rt_2 Rt_3 Rt_4 Rt_5 Rt_6 Rt_7	Rt_0 Rt_1 Rt_2 Rt_3 Rt_4 Rt_5 Rt_6 Rt_7	Rt_0 Rt_1 Rt_2 Rt_3 Rt_4 Rt_5 Rt_6 Rt_7
ExPre	Wt_0 Wt_0 Wt_1 Wt_1 Wt_2 Wt_2 Wt_3 Wt_3	Wt_0 Wt_1 Wt_2 Wt_3 Wt_4 Wt_5 Wt_6 Wt_7	Rt_0 Rt_1 Rt_2 Rt_3 Rt_4 Rt_5 Rt_6 Rt_7
Output	Wt_0 Wt_1 Wt_2 Wt_3	Wt_0 Wt_1 Wt_2 Wt_3	Wt_f Wt_f Wt_f Wt_f
Index	0 1 2 3	0 1 2 3	0 1 2 3

Fig. 11. Permission pattern of arrays in stream compaction algorithm corresponding to Fig. 10; Rt_i/Wt_i means thread i has read/write permission. Green color indicates permission changes.

- *ExPre*: since *ExPre* is the output of the exclusive prefix sum function (i.e., Algorithm 1), the permission pattern at the beginning of Algorithm 3 should match the permission preconditions of Algorithm 1. Thus, each thread (*tid* < half *ExPre* size) has write permissions to locations $2 \times tid$ and $2 \times tid + 1$ (Fig. 11: left). As postcondition of the exclusive prefix sum function (i.e., Algorithm 1), each thread has write permission to its "own" location in *ExPre* (Fig. 11: middle). Since each thread only needs read permission in line 5 of Algorithm 3, we change the permission pattern from write to read in the barrier (Fig. 11: right).
- *Output*: it is only used in line 5 of Algorithm 3 and its permissions are according to the values in *ExPre*. Thus, the initial permissions for *Output* can be arbitrary and in the barrier (line 3), 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. 11: right).

5.3. 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 3). Second, we prove that *Output* 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*.

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,[12] *inp_seq* and *flag_seq* are always the same as *Input* and *Flag*.[13]

First, to reuse of the exclusive prefix sum specification (line 2 in Algorithm 3) from Algorithm 1, we should consider two points: (1) the input to the exclusive prefix sum (*Flag*) in the stream compaction algorithm 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 5 in Algorithm 3). Therefore, we use VerCors to prove some more 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:

Property 4. *For any sequence $flag_seq$ (with only zeros and ones):*

$$(\forall i. 0 \leq i < |flag_seq|: flag_seq[i] = 0 \ \vee \ flag_seq[i] = 1) \Rightarrow$$
$$intsum(flag_seq) \geq 0.$$

We need Property 4 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*:

Property 5. *For any sequence $flag_seq$ (with only zeros and ones):*

$$(\forall i. 0 \leq i < |flag_seq|: flag_seq[i] = 0 \ \vee \ flag_seq[i] = 1) \Rightarrow$$
$$(\forall i. 0 \leq i < |epsum(flag_seq)|: epsum(flag_seq)[i] \geq 0 \ \wedge$$
$$epsum(flag_seq)[i] \leq intsum(flag_seq)).$$

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

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

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

List 8 The *filter* function.

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

List 9 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 List. 8
3  if(flag_seq[tid] == 1)
4    // applying Property 7
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 4-5 in Algorithm 3
9    assert Output[ExPre[tid]] == out_seq[epsum(flag_seq)[tid]]; // by lines 7-8
```

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 5 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 6. *For any sequence $flag_seq$ (with only zeros and ones):*

$$(\forall i.0 \leq i < |flag_seq|: flag_seq[i] = 0 \;\vee\; flag_seq[i] = 1) \Rightarrow \\ (\forall i.0 \leq i < |epsum(flag_seq)| \;\wedge\; flag_seq[i] = 1: \\ (epsum(flag_seq)[i] \geq 0 \;\wedge\; epsum(flag_seq)[i] < intsum(flag_seq))).$$

Property 6 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, we already proved that *epsum*(*flag_seq*) is equal to the result of the exclusive prefix sum function (i.e., *ExPre*).

Second, we define a ghost variable, *out_seq*, as a sequence of integers and a mathematical function, *filter*, as shown in List. 8. This function computes the compacted list of an input sequence, *inp_seq*, by filtering it according to a flag sequence, *flag_seq*. 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 3 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 7. *For any equal sized sequences $input_seq$ and $flag_seq$:*

$$(\forall i.0 \leq i < |flag_seq|: flag_seq[i] = 0 \;\vee\; flag_seq[i] = 1) \Rightarrow \\ (\forall i.0 \leq i < |epsum(flag_seq)| \;\wedge\; flag_seq[i] = 1: \\ (inp_seq[i] = filter(inp_seq, flag_seq)[epsum(flag_seq)[i]])).$$

From Property 7, 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 List. 8), 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*. List. 9 shows the proof steps which are located at the end of Algorithm 3. Through some smaller steps, and using Property 7 we prove in VerCors that *out_seq* and *Output* is the same (line 9). Note that, we proved that for each *tid*, $epsum(flag_seq)[tid]$ is equal to $ExPre[tid]$.

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 exclusive prefix sum algorithm.

5.4. Verification challenges for CUDA

Since the CUDA implementation of stream compaction reuses the prefix sum implementation, the complexity of the annotated CUDA file increases. As a result, we encounter non-termination problems with the verification in the tool, as Z3 cannot prove nor refute the properties when they are all in one file. To solve this problem, we split up the code over several files. First of all, we removed the host part and only preserved the kernel part. This does not make any difference for the verification as there is only one parallel block and the algorithm completely runs inside the kernel. Second, to keep the verification effort manageable, we prove all lemmas in a separate file and only use their specification during the kernel verification. Again, this does not affect what is proven, but it does have a significant impact on the verification time.

6. Related work

There are only a few approaches to reason about GPU programs. Those 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 [22], Oclgrind [23] and GRace [29]). 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 [19] and KLEE-CL [12]. There are also several static approaches to verify data race-freedom of GPU 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 [18] and GPUVerify [3] are based on this approach. VeriFast is a static verification tool to prove functional correctness of single-threaded and multithreaded C and Java programs, but it is not able to reason about GPU programs. Except VerCors and VeriFast [16], none of these tools can reason about functional correctness of concurrent programs.

There is no previous work on formally verifying the parallel stream compaction algorithm on GPUs. The closest related work to the verification of prefix sum algorithms is by Chong et al. [11]. 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. 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, to verify even data race-freedom of GPU programs in GPUVerify, 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.

In our opinion, the advantage of our approach is that our verification approach for the prefix sum can be reused for these two new algorithms, while Chong might not be able to reuse his prefix sum approach to verify the parallel stream compaction algorithm.

7. Conclusion

This paper shows how we verify data race-freedom and functional correctness of CUDA implementations of two parallel prefix sum algorithms, Blelloch's and Kogge-Stone's algorithm, using the VerCors verifier. Furthermore, we have proven the correctness of the parallel stream compaction algorithm on top of the verified prefix sum. To prove these algorithms, we added CUDA support to the 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 address 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 Blelloch's algorithm, since there is only one phase. We benefit from functions, operations and properties that are defined in the earlier verification and reuse them in the second verification. 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 verification of the presented algorithms.

As future work, we 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. Moreover, we plan to verify more CUDA implementations of parallel algorithms in VerCors.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] A. Amighi, S. Darabi, S. Blom, M. Huisman, Specification and verification of atomic operations in GPGPU programs, in: SEFM 2015 Collocated Workshops, Springer, 2015, pp. 69–83.
[2] J. Berdine, C. Calcagno, P.W. O'hearn, Smallfoot: modular automatic assertion checking with separation logic, in: International Symposium on Formal Methods for Components and Objects, Springer, 2005, pp. 115–137.
[3] A. Betts, N. Chong, A. Donaldson, S. Qadeer, P. Thomson, GPUVerify: a verifier for GPU kernels, in: OOPSLA, ACM, 2012, pp. 113–132.
[4] M. Billeter, O. Olsson, U. Assarsson, Efficient stream compaction on wide simd many-core architectures, in: Proceedings of the Conference on High Performance Graphics 2009, 2009, pp. 159–166.
[5] G.E. Blelloch, Prefix sums and their applications, in: Synthesis of Parallel Algorithms, Morgan Kaufmann Publishers Inc., San Francisco, 1993.
[6] S. Blom, S. Darabi, M. Huisman, W. Oortwijn, The vercors tool set: verification of parallel and concurrent software, in: International Conference on Integrated Formal Methods, Springer, 2017, pp. 102–110.
[7] S. Blom, M. Huisman, M. Mihelčić, Specification and verification of GPGPU programs, Sci. Comput. Program. 95 (2014) 376–388.
[8] R. Bornat, C. Calcagno, P. O'Hearn, M. Parkinson, Permission accounting in separation logic, in: POPL, 2005, pp. 259–270.
[9] J. Boyland, Checking interference with fractional permissions, in: International Static Analysis Symposium, Springer, 2003, pp. 55–72.
[10] R.P. Brent, H.T. Kung, A regular layout for parallel adders, IEEE Comput. Archit. Lett. 31 (1982) 260–264.
[11] N. Chong, A.F. Donaldson, J. Ketema, A sound and complete abstraction for reasoning about parallel prefix sums, in: ACM SIGPLAN Notices, ACM, 2014, pp. 397–409.
[12] P. Collingbourne, C. Cadar, P.H. Kelly, Symbolic testing of OpenCL code, in: Haifa Verification Conference, Springer, 2011, pp. 203–218.
[13] L. De Moura, N. Bjørner, Z3: an efficient smt solver, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2008, pp. 337–340.
[14] M. Harris, S. Sengupta, J.D. Owens, Parallel prefix sum (scan) with CUDA, in: GPU Gems 3, 2007, pp. 851–876.
[15] D. Horn, Stream reduction operations for GPGPU applications, in: GPU Gems 2, 2005, pp. 573–589.
[16] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, F. Piessens, Verifast: a powerful, sound, predictable, fast verifier for C and Java, in: NASA Formal Methods Symposium, Springer, 2011, pp. 41–55.
[17] P.M. Kogge, H.S. Stone, A parallel algorithm for the efficient solution of a general class of recurrence equations, IEEE Trans. Comput. 100 (1973) 786–793.
[18] G. Li, G. Gopalakrishnan, Scalable SMT-based verification of GPU kernel functions, in: SIGSOFT FSE 2010, Santa Fe, NM, USA, ACM, 2010, pp. 187–196.
[19] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, S.P. Rajan, GKLEE: concolic verification and test generation for GPUs, in: ACM SIGPLAN Notices, ACM, 2012, pp. 215–224.
[20] P. Müller, M. Schwerhoff, A.J. Summers, Viper: a verification infrastructure for permission-based reasoning, in: International Conference on Verification, Model Checking, and Abstract Interpretation, Springer, 2016, pp. 41–62.
[21] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with cuda: is cuda the parallel programming model that application developers have been waiting for?, Queue 6 (2008) 40–53.
[22] Nvidia, Cuda-memcheck: user manual (version 10), https://developer.nvidia.com/cuda-memcheck, 2019.
[23] J. Price, S. McIntosh-Smith, Oclgrind: an extensible OpenCL device simulator, in: Proceedings of the 3rd International Workshop on OpenCL, ACM, 2015, p. 12.
[24] M. Safari, M. Huisman, Formal verification of parallel stream compaction and summed-area table algorithms, in: International Colloquium on Theoretical Aspects of Computing, Springer, 2020, pp. 181–199.
[25] M. Safari, W. Oortwijn, S. Joosten, M. Huisman, Formal verification of parallel prefix sum, in: NASA Formal Methods Symposium, Springer, 2020, pp. 170–186.
[26] S. Sengupta, A. Lefohn, J. Owens, A work-efficient step-efficient prefix sum algorithm, 2006.
[27] J. Sklansky, Conditional-sum addition logic, IEEE Trans. Electron. Comput. 2 (1960) 226–231.
[28] Viper project website, http://www.pm.inf.ethz.ch/research/viper, 2016.
[29] M. Zheng, V.T. Ravi, F. Qin, G. Agrawal, GRace: a low-overhead mechanism for detecting data races in GPU programs, ACM SIGPLAN Not. 46 (2011) 135–146.