Let's sort this out: GPGPU Verification of Radix Sort

Dré van Oorschot
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
d.h.m.a.vanoorschot@student.utwente.nl

ABSTRACT

This paper shows how the VerCors verification toolset can be used to prove data race freedom and functional correctness of a parallel radix sort algorithm for GPUs. This is a widely used standard sorting implementation for GPGPU programming frameworks and therefore its correctness is of utmost importance. Additionally, it presents the usefulness of VerCors as well as its current limitations to the scientific community.

Keywords

GPGPU, radix sort, VerCors.

1. INTRODUCTION

Sorting is the act of putting elements in an array in a certain order and is a fundamental problem in computer science. Several efficient solutions to the sorting problem exist such as merge sort and quick sort. These algorithms work on the basis of reordering elements by comparing said elements with one another.
An alternative approach is to count the occurrence of different elements and reorder the elements based on the amount of occurrences. Radix sort and counting sort are examples of algorithms that make use of this principle.
Parallel implementations of sorting algorithms on Graphics Processing Units (GPUs) have been made and often perform better than their sequential CPU-bound counterparts [1, 5]. The use case of GPUs are far wider than only graphics computations. Special General Purpose GPUs (GPGPUs) have been developed that are designed to also do parallel computing tasks that are not graphics related. GPGPU programs are in general more powerful than similar CPU-bound programs when the compute task entails highly parallel computations. This is due to the high core and thread count of GPUs [7].
Verifying sorting algorithms is important due to their wide usage. A lot of programming languages have built-in sorting functions. Incorrect implementations are therefore impermissible. However, verification of parallel programs is hard due to the exponential state space of the problem. This paper focuses on verifying data race freedom and functional correctness of a parallel radix sort algorithm for GPUs.

32nd *Twente Student Conference on IT* Jan. 31st, 2020, Enschede, The Netherlands.

Organisation. The paper starts by outlining some background on radix sort, GPGPU programming, and the VerCors verification toolset in Sections 2.1, 2.2, and 2.3 respectively. This is followed by the central problem statement containing the research questions of this paper in Section 3. The research questions posed in the problem statements are answered in Sections 4 through 6. Finally, the paper is concluded with Future Work, Related Work, and Conclusion in Sections 7, 8, and 9 respectively.

2. BACKGROUND

This section discusses background information needed to understand the paper. It explains the functioning of radix sort as well as how the steps used in radix sort can be parallelized. Further more it explains the basics of the VerCors verification tool and of GPGPU programming.

2.1 Radix Sort

Radix sort is not a comparison sort but a counting sort. It divides every key into equally sized chunks and repeatedly sorts the keys for every chunk in three steps: count, prefix sum and reorder [6].

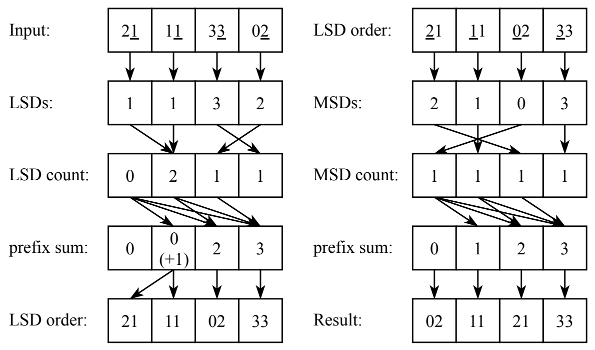


Figure 1: Radix sort example using a radix of 4

The size of the chunks is determined by the chosen radix (or base). The example in Figure 1 uses a radix of 4, meaning the four possible values of a key chunk are 0, 1, 2, and 3. Moreover, the example can be sorted with two passes of the algorithm by first sorting the input array on the Least Significant Digit (LSD), followed by a sort on the Most Significant Digit (MSD). The sorting of the LSD is explained in more detail.
The input of the example consists of the following values:

$$input = [21, 11, 33, 02] \tag{1}$$

Taking the LSD of all the values of the input array results in the following input for the first pass of the algorithm:

$$LSDs = [1, 1, 3, 2] \tag{2}$$

The first step is to count the number of occurrences of every key: 0 never occurs, 1 occurs twice, 2 occurs once and 3 occurs once. The frequency of every number is stored on the index of the corresponding number. The result is the following count:

$$c = [0, 2, 1, 1] \tag{3}$$

The second step is to take the exclusive prefix sum of the count to get the initial starting index of every key in the result array. The exclusive prefix sum of a value in an array is the sum of every value that preceded said value in the array or more formally: let c be an array of initial values, and p the array of prefix sums. Every value in p can be calculated with the following formula:

$$p[i] = \sum_{0 \leq j < i} c[j] \tag{4}$$

In this example, the reslt is the following prefix sum:

$$p = [0, c[0], c[0] + c[1], c[0] + c[1] + c[2]] = [0, 0, 2, 3] \tag{5}$$

The last step is to reorder the elements of the input such that the LSDs of the input are in order. Referring back to the input, the first value that needs to be reordered is 21. 21 has an LSD of 1. The value at index 1 of the prefix sum is 0. This means 21 needs to be inserted at index 0 of the result. Additionally, the value at index 1 in the prefix sum is incremented, so that the next value with an LSD of 1 gets inserted right next to 21.
Performing this routine for each value of the input results in an array whose LSDs are sorted in ascending order.

$$LSDorder = [21, 11, 02, 33] \tag{6}$$

Performing this entire algorithm on the MSD and using the above LSD order as input will result in a sorted array. Note that in the example 21 and 11 have retained their original order, meaning that this sort is stable. This is important because the result of previous passes of the algorithm should persist in later passes of the algorithm.
The three steps of a radix sort pass can be parallelized in a GPGPU implementation of radix sort.
Parallel Count. An approach to parallelize the counting of keys is to have a global counter and let all kernels do atomic increments on this global counter. This is not an optimal solution because different kernels may block one another by wanting to increment the same value.

Algorithm 1 Parallel Count Algorithm

```
tempCounts[ ][ ]
function ParCount(tid, partition[ ], output[ ])
    for x ∈ partition do
        tempCounts[tid][x] ← tempCounts[tid][x] + 1
    end for
    barrier(tid)
    if tid $\leq$ radix then
        for count ∈ tempCounts do
            output[tid] ← output[tid] + count[tid]
        end for
    end if
end function
```

Another more optimal approach is shown in Algorithm 1. In this approach, a local count for all kernels is kept, which are all contained in the two dimensional array *tempCounts*. When every thread is done, all local counts can be summed into a global count which could possibly also sum all the counters in parallel by having a kernel for every index of the count.

Parallel Prefix Sum. The parallel prefix sum can be calculated either inclusively or exclusively. Blelloch [2] proposed a parallel algorithm to solve the exclusive prefix sum problem. Kogge-Stone [8] proposed a parallel algorithm to solve the inclusive prefix sum problem. Both these algorithms have been verified for both functional correctness and data race freedom in VerCors[1]. Correctness of the algorithm is assumed in this paper.

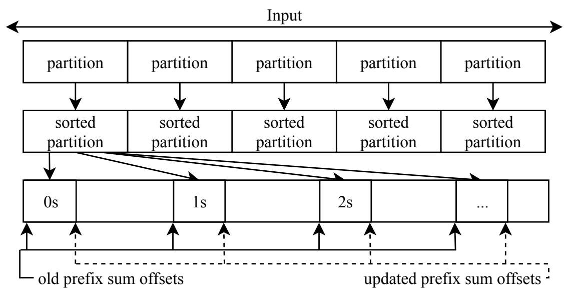


Figure 2: The parallel reorder step in radix sort

Parallel Reorder. Reordering happens in two steps as can be seen in Figure 2. The input is split up into equal partitions. Each partition is sorted in parallel. Once every partition is sorted, each partition can be inserted into the result sequentially. Since all values in a partition are in order, all values in a partition can be inserted in parallel on the offset provided by the prefix sum calculated earlier and the offset in the partition. The prefix sum can then be updated based on the amount of each value that has been inserted to provide offsets for the next partition.

Algorithm 2 Parallel Reorder Algorithm

```
function ParallelReorder(tid, input[ ], output[ ], prefixsum[ ])
    for x ∈ input do
        if getKeyChunk(x, radix, iteration) = tid then
            output[prefixsum[tid]] ← x
            prefixsum[tid] ← prefixsum[tid] + 1
        end if
    end for
end function
```

However, this paper uses the approach as outlined in Algorithm 2. This approach takes the entire input at once and has an amount of parallel threads equal to the radix. Every thread is responsible for looking up one specific key. When found, the thread adds it to the result array corresponding to the index presented by the prefix sum and update the prefix sum accordingly.
The upside of the second approach is that it is simpler to implement while still requiring no atomic operations. This implementation was therefore chosen to be verified first. It should be noted that the second approach is a lot slower for large inputs compared to the first approach; In the first approach, the amount of threads scale with the input size while in the second approach, it scales with the size of the radix which is usually small.

2.2 GPGPU programming

To understand the GPU implementation of radix sort and to convert this implementation to a model that VerCors understands, it is key to get familiar with common GPU

[1]The source code of the parallel prefix sum algorithm can be found at: `https://github.com/Safari1991/prefixsum`.

programming languages. This section will explain general GPGPU programming concepts and relate them to OpenCL and CUDA.
GPGPU programs are written in the form of kernels. A kernel is a function or program that is executed by each thread. Kernels are invoked by a host program running on the CPU that provides instructions to a GPU. The amount of threads is specified when invoking a kernel. The kernel then gets executed on the amount of specified threads.
Both in CUDA [7] and OpenCL [10], kernels are void functions. A common workflow for kernel functions is therefore to initialise the data on the host program, copy that data to the GPU, execute the kernel function on the GPU and copy the modified data back to the host program.
To bring structure to a GPGPU program and addressability to threads running kernels, threads are assigned to workers. A worker contains a single thread executing a kernel. In OpenCL, when a worker has finished a kernel, it collects a new one from the work queue until no more kernels are left [10]. Workers are grouped in work-groups. Additionally, CUDA features a model called a grid where work-groups can be laid out in a one, two or three dimensional structure [7].
Workers and work-groups have access to different memory:

- Global memory storing data accesible by all workers. It is the largest in size but slowest in operation.
- Shared memory storing data belonging to a work-group that only workers belonging to said work-group can access. It is small in size but fast in operation.
- Local memory storing data belonging to an individual worker. Usually local variables of a kernel are stored here.

Additionally, OpenCL features a memory type called constant memory that is the same as global memory but is read-only [10]. Figure 3 shows the relation between framework structure and the different memory spaces.

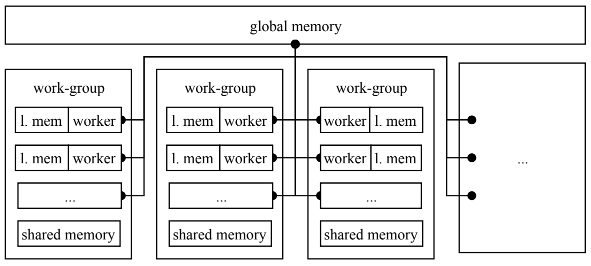


Figure 3: Memory model used in GPGPU frameworks

2.3 VerCors

VerCors is a tool for mechanised verification for different concurrency models such as heterogeneous concurrency (e.g. concurrent Java programs) and homogeneous concurrency (e.g. parallel GPU programs) [3]. It has verification support for a subset of languages including Java, C, OpenCL and its own internal language, PVL.
The tool uses permission-based separation logic for verification, which is an extension of Hoare logic that provides reasoning about where data is stored in memory [3]. This enables reasoning about what part of memory is accessible by each thread. Permissions are represented as a fraction in the interval (0, 1] where the interval (0, 1) represents a read permission and 1 represents a write permission. Additionally, several read permissions can be added up into a write permission and alternatively, a write permission can be divided up into several read permissions. Every variable has a set of permissions that at any time in the program is owned by a certain thread or function. Permissions to a variable can be taken and given back between threads and functions. When a permission model is specified on a program such that there is no unpermitted access to a variable and no permissions on a variable overlap between threads or functions, the program is proven to be data race free.
Verification happens by using the concept of programming by contract. Programs can be annotated with preconditions and postconditions, and loop and class invariants. These annotations are to be written in permission-based separation logic. If a program is annotated correctly in every step of the program, the verifier is able to infer and verify the specified specifications of the program.
To understand this paper, it is important to at least understand the notion of ghost variables and sequences. Ghost variables are variables that are declared in the contract of a function. As the name suggests, these variables have no influence on the functionality of the algorithm and should solely be used for verification purposes. Because ghost variables act in the background, permission modeling is not required for these variables.
Sequences are very similar to arrays, but there are some key differences between the two. Firstly, sequences are immutable. Secondly, they are more expressive than arrays. For example, unlike arrays, sequences support slicing, concatenation, and support a head and tail function. They are therefore very useful for verification helper functions and ghost variables[2].

3. PROBLEM STATEMENT

The central research question for this paper is:
How can the VerCors verification toolset verify data race freedom and functional correctness of the parallel radix sort algorithm?
This research question poses a problem that was too large to solve in the allotted time for this paper, so instead the paper focuses on the verification of the subroutines used for the algorithm.
The central research question has been split up in several sub-questions. There are two sub-questions regarding the verification problem:

RQ 1.1 How can VerCors verify data race freedom of the count step and the reorder step in parallel radix sort?
RQ 1.2 How can VerCors verify functional correctness of the count step and the reorder step in parallel radix sort?

During the production of this paper, several issues were encountered related to the VerCors tool, spawning the following sub-question:

RQ 2.1 What are the current limitations in the VerCors verification toolset?

RQ 1.1 is answered in Sections 4.1 and 5.1 for the count step and reorder step respectively. A possible approach to answer **RQ 1.2** is presented in Section 4.2 for the count step. Section 5.2 presents the specifications needed to answer **RQ 1.2** regarding the reorder step. A further approach to answer the central research question is outlined in Section 7. Finally, **RQ 2.1** is answered in Section 6.

4. PARALLEL COUNT VERIFICATION

This section discusses how VerCors has been used to prove data race freedom of the Parallel Count algorithm by outlining the permission model for the algorithm. It also

[2]For more (albeit outdated) information on sequences see: `https://github.com/utwente-fmt/vercors/wiki/Axiomatic-Data-Types`

discusses a possible approach to prove functional correctness of the algorithm by outlining the specifications that need to be proven as well as how these conditions could be proven using VerCors.

4.1 Data Race Freedom

To prove data race freedom, a permission flow needs to be defined for the algorithm. This section describes the general idea of this process. The details of the process can be found in the source code[3].

The function as a whole revolves around the following three parameters and requires the following initial permissions:

- The input is an array of integers on which the counting will be performed. Read permission is required on the entire array.
- The output is an array of integers with a length equal to the radix. It is used to store the final count values of the algorithm. Write permission is required on the entire array.
- *tempCounts* is a two dimensional array of integers of which the outer array contains one array per thread to keep count of a partition of the input. These inner arrays have a length equal to the radix. Write permission is required on the entire array.

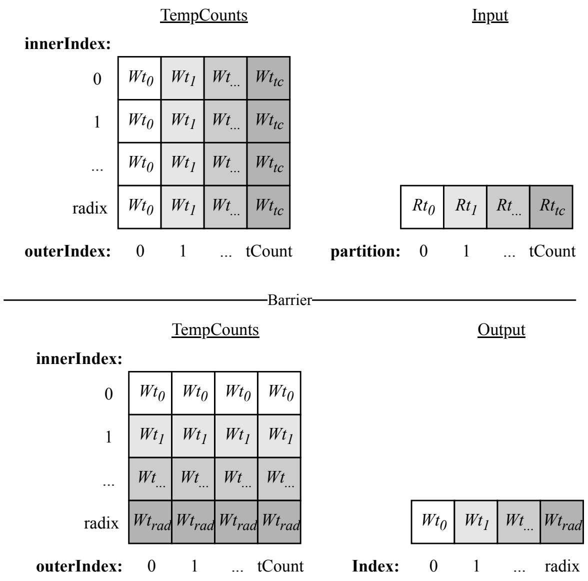

TempCounts
Input
innerIndex:
0
1
...
radix
Wt_0 Wt_1 $Wt_{...}$ Wt_{tc}
Rt_0 Rt_1 $Rt_{...}$ Rt_{tc}
outerIndex: 0 1 ... tCount
partition: 0 1 ... tCount
Barrier
TempCounts
Output
innerIndex:
0
1
...
radix
Wt_0 Wt_1 $Wt_{...}$ Wt_{rad}
outerIndex: 0 1 ... tCount
Index: 0 1 ... radix

Figure 4: Permission model for the parallel count algorithm for both before and after the barrier. Wt_i represents a write permission for thread i on a variable and Rt_i represents a read permission for thread i on a variable.

The algorithm itself can be defined by a section before and after the barrier as can be seen in Figure 4. Before the barrier, the input is divided up in equal partitions. Each thread has read permission on a specific partition and write permission on one specific inner array of *tempCounts*.

The algorithm synchronises in the barrier when every thread has calculated the count of its assigned partition of the input. Permissions are redistributed in the barrier. All threads drop their read permissions on the input as the input is not used anymore. Each thread is assigned a specific thread id. If that thread id falls within the radix, the thread gets write permission on the variable at the index (matching its thread id) of the output and write permission on the variable at that index in every inner array of the *tempCounts*. A thread can now sum all the local counts of a specific value and write the result to the correct location in the output. Although read permission on *tempCounts* would technically be sufficient, write permission is necessary to prevent a permission leak. A permission leak occurs when function requires a certain permission from the function caller but does not ensure that this permission is returned back to the caller [9].

Although the algorithm presented works for any input size, a few assumptions have been made to simplify the proof:

- $threadCount \geq radix$. Otherwise there are too little threads to sum all the values to the output after the barrier.
- $partitionsize * threadcount = |input|$. Otherwise some partitions may be partially empty or not the entire input will be processed.

4.2 Functional Correctness

This section does not present the verification of the functional correctness of the parallel count algorithm, but rather presents a possible approach on how to use VerCors to solve this problem. To prove functional correctness the following two postconditions will need to be proven:

$$\sum_{x \in (c_0,\ldots,c_n)} x = |(a_0, ..., a_m)| \tag{7}$$

$$\sum_{x_i \in (c_0,\ldots,c_n)} (x_i * i) = \sum_{y \in (a_0,\ldots,a_n)} y \tag{8}$$

Where $(a_0, ..., a_n)$ is the input array and $(c_0, ..., c_n)$ is the count result.

Intuition. The first condition ensures that all the input values have been processed and are present in the count output. If an input value got skipped or counted double, the sum of the counts would not match the length of the input array.

However, the first conditions does not ensure that every value is counted at the correct index of the output. For example, the condition would still hold if the value 3 would be processed by incrementing the output value at index 2. This is where the second condition becomes important. It makes sure that the count output is a proper reduction from the input. If a number were to be miscount, it would be multiplied by the wrong number on the left hand side of the condition and lead to a violation of the condition.

Conversely, a count output could be produced such that it contains miscounts, but still satisfies the second condition. For example, if the algorithm were to count a single 2 as two 1's, the condition would still hold. However, in that case, it would violate the first condition.

Approach in VerCors. Verifying the first condition consists of two parts; the parallel counting step before the barrier and the merging step after the barrier. For the part before the barrier, the condition needs to be verified for all the individual arrays in *tempCounts*.

For reasons discussed in Section 6, it is not wise to try and verify properties directly on *tempCounts*. Instead, a ghost variable could be created that is of type seq<seq <**int**>> and perform verification on this variable. One also needs to make sure it is at all times consistent with *tempCounts*.

Additionally, a helper function is required that can sum up sequences. This function can be used on each individual sub-sequence of the ghost variable to create a loop invariant that either verifies that after each loop iteration, the

[3]The source code of the parallel count algorithm can be found at: `http://bit.ly/2G3wHrS`

sum of the sub-sequence is equal to the number of completed iterations or verifies that the current sum of the sub-sequence is one higher compared to the last iteration. With such a loop invariant in place, it can be verified that after the loop that each the sum of each sub-sequence is equal to the length of each partition of the input. The algorithm has been correctly verified till the point directly before the barrier using this helper function and the two dimensional ghost sequence.

Another helper function is required after the barrier to verify the merge step of the algorithm. The helper function should be able to sum all values at a certain index of the inner sequences in a certain range from the beginning of the outer sequence. So for example, the function call rangeSum(xss, 0, 10) should return the sum of all the 0th elements of the first 10 inner sequences.

This function can be used as a loop invariant for the loop after the barrier. After each loop iteration of thread k, it should be verified that the current value of the output count at index k is equal to the sum of all the values at index k of the first n^{th} inner sequences where n is the amount of completed loop iterations. Or put more formally:

$$\sum_{i=0}^{n} prefixsum[i][k] = output[k] \tag{9}$$

Where k is the thread id and n the amount of completed loop iterations.

If the loop invariant holds, it can be verified that after the loop, each value of the output matches the total sum of all values at a certain index of the inner sequences and thus the first condition holds.

The second condition can be verified in a similar way as the first condition before the barrier. However, a different helper function will be used that calculates the sum of each value in a sequence multiplied by its index to produce a result similar to the left hand side of the second condition. For the right hand side of the condition, the sequence sum function can be reused.

The second condition may be directly verifiable after the loop in the part after the barrier. This may be the case as at this point in the algorithm, it has been verified that the individual counts of $tempCounts$ satisfy the second condition as well as that these counts are correctly merged into the output (this has been verified by the first condition). This means that it can directly be inferred that the second condition also holds for the final output.

5. PARALLEL REORDER VERIFICATION

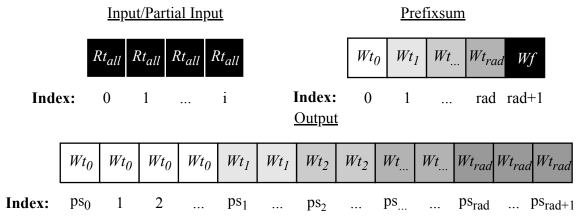
This section discusses how VerCors has been used to prove data race freedom of the Parallel Reorder algorithm by outlining the permission model for the algorithm. It also provides possible specifications that could be used to prove functional correctness of the algorithm.

5.1 Data Race Freedom

To prove data race freedom, a permission flow needs to be defined for the algorithm. This section describes the general idea of this process. The details of the process can be found in the source code[4].

the function as a whole revolves around the following parameters and require the following initial permissions:

- The input is an integer array. It contains the values that need to be reordered. Read permission is required on the entire array.
- The partial input is an integer array. It contains the relevant part of each value of the input based on the radix and current iteration of the radix sort algorithm. It is used to match up the original input with the relevant prefix sum index. Read permission is required on the entire array.
- The prefix sum is an integer array and has length equal to $radix + 1$. It is used to determine the index of an input value in the output. Because the prefix sum needs to be updated after every output insertion, write permission is required on the entire array.
- The output is an integer array. Write permission is required on the entire array.



Input/Partial Input
Rt_{all}
Rt_{all}
Rt_{all}
Rt_{all}
Index: 0 1 ... i
Prefixsum
Wt_0
Wt_1
$Wt_{...}$
Wt_{rad}
Wf
Index: 0 1 ... rad rad+1
Output
Wt_0 Wt_0 Wt_0 Wt_0 Wt_1 Wt_1 Wt_2 Wt_2 $Wt_{...}$ $Wt_{...}$ Wt_{rad} Wt_{rad} Wt_{rad}
Index: ps_0 1 2 ... ps_1 ... ps_2 ... $ps_{...}$ ... ps_{rad} ... ps_{rad+1}

Figure 5: Permission model for the parallel reorder algorithm. Wt_i represents a write permission for thread i on a variable and Rt_i represents a read permission for thread i on a variable. t_{all} represents permission for all threads and f represent an undistributed permission. ps_i indicates the value stored at index i in the prefix sum copy.

The algorithm consists of one parallel block, so permissions only need to be distributed at the beginning of the algorithm. The parallel block spawns an amount of threads equal to the radix. Figure 5 presents the permission distribution among all threads involved in the algorithm.

All threads get read permission on both the input and partial input. This is allowed because parallel reads cannot cause data races; the order of threads that access the variable is irrelevant for the final result. Furthermore, each thread has write permission on the variable in the prefix sum whose index matches the threads id.

Similarly, each thread has write permission on a range specified by the prefix sum. More specifically, the values in the prefix sum are used as range indicators on the output. For example, thread 0 has write permission from indices *prefixsum*[*0*] till *prefixsum*[*1*] of the output and thread 1 has write permission from indices *prefixsum*[*1*] till *prefixsum*[*2*] of the output and so on. This is also the reason why the prefix sum has one extra value appended. The value contains the value of the last index of the output and only serves as a permission range indicator. It is not used anywhere else in the program which is why no thread has any permissions over the value.

However, using the prefix sum as a range indicator for permissions is not feasible as the values of the prefix sum get updated overtime. To solve this issue, a copy of the prefix sum is created as a ghost variable that remains unchanged for the entire algorithm to keep the permission ranges in place.

5.2 Functional Correctness

To prove functional correctness of the reorder step, the postcondtions of the parallel prefix sum method become preconditions for the parallel reorder algorithm, to ensure that the prefix sum parameter of the algorithm is correct. Additionally, the following two postconditions should

[4]the source code of the parallel reorder algorithm can be found at: `http://bit.ly/3amS45n`

hold:

$$\forall x_i, y_j \in (c_0, ..., c_n)(i \geq j \rightarrow x_i \geq y_j) \tag{10}$$

$$(a_0, ..., a_n) \mapsto (b_0, ..., b_n) \tag{11}$$

Where $(a_0, ..., a_n)$ and $(b_0, ..., b_n)$ are the input and output array respectively and $(c_0, ..., c_n)$ are the relevant digits taken from the output for a certain iteration of the radix sort algorithm with a certain radix.

Intuition. The first condition implies that any two digits from the partial output are correctly ordered. If this holds for all digits, the output must be ordered. However, for this condition to hold, it is not necessary that the output is a rearrangement of the input. For example an input of (3, 2, 1) being sorted to (1, 1, 1) would verify.

The second condition makes sure that the input matches the output by stating that the input should be a bijection of the output. Together with the first condition, the specifications makes sure that the algorithm creates an ordered bijection, which means that the output is sorted.

It should be noted that VerCors does not have a built-in bijection function. However, VerCors does support multisets in the form of bags. Two equal multisets form a bijection and a multiset taken from an array also forms a bijection. So a possible approach could be to convert both the input and output to a bag and assert that they are equal to prove they form a bijection.

6. VERCORS LIMITATIONS

Although the VerCors verification toolset is diverse and PVL expressive, it does have its limitations. Some of these limitations were encountered during the production of this paper. This section highlights some of these limitations.

Non-termination of verification. The more complex a verification case becomes, the longer it takes for VerCors to prove or disprove certain properties of an algorithm. In some cases, this can lead to non-termination of the verification. It becomes clear that at some point VerCors lacks information to prove the given specification, but is also unable to disprove the specification, which results in the verification running indefinitely. An example of such a case occurs during attempts to prove data race freedom of the entire parallel radix sort algorithm[5].

This can become problematic for two reasons. Firstly, it becomes hard to determine whether a verification that inherently takes a long time to verify has become non-terminating and can therefore leave a researcher in doubt. Secondly, the tool gives no feedback when it has become un-terminating which makes it difficult in some cases to pinpoint the cause of the problem.

Multiple read permissions on one variable. In some verification problems it may occur that multiple read permissions are granted to different threads. Although this should be allowed as no data races can occur in interleaving reads on a variable, the tool may still complain about insufficient permission. An example of such a case can be seen in Listing 1.

When running Listing 1, VerCors will give a warning that there are no triggers available for a quantifier at an undefined position. The quantifier referred to in this case is the **forall** on line 6. The trigger is necessary to be able to reference the same variable in different threads. Usually this is not a problem either because a variable is only referenced by one thread at the time or because additional specifications give enough context for VerCors to still be able to infer variable access among different threads. However, this is not always the case as is the case in Listing 1, which will report that there is insufficient permission to grant read permission on the xs array on line 6.

[5]The source code for the verification of data race freedom of the parallel radix sort algorithm is available at: `http://bit.ly/2G3wLrA`

```
 1  context_everywhere xs != null;
 2  context_everywhere t > 0;
 3  context (\forall* int i;  i >= 0 && i < xs.length;
        Perm(xs[i], 1/2));
 4  void parRead(int[] xs, int t) {
 5    par parRead(int tid=0..t)
 6    context (\forall* int i;  i >= 0 && i < xs.length;
          Perm(xs[i], 1/(2*t)));
 7    {
 8      //some Code
 9    }
10  }
```

Listing 1: VerCors example where multiple threads with read permission on the same variable fails

Two dimensional arrays. The inner arrays of a two dimensional arrays are not of the same type as the outer array. When comparing two dimensional integer arrays to flat integer arrays, their types are Array<Cell<Integer>> and Array<Integer>. While semantically a flat array is of the same type as an inner array of a two dimensional array, PVL treats them as type Array and type Cell respectively, meaning they are syntactically of a different type.

In most cases this does not present problems as a Cell object has the same functionality as an Array object such as indexing and length querying. However, in more advanced use cases it can become a problem, two of which are presented in Listing 2.

```
1  void foo(int [][]  xss) {
2      bar(xss [0]) ;
3      xss [0]  = new int[xss[0].length];
4  }
5  void bar(int[] xs) {
6    //some Code
7  }
```

Listing 2: VerCors example where two dimensional arrays can cause errors

The first problem is presented in line 2 of Listing 2. Calling the bar() function on an inner array of xss will throw a type Error. A possible work around is to copy over all the values from xss to a new temporary flat array in a loop and feed that to the bar() function. However, this makes verifying specifications on this kind of code unnecessarily complicated.

The second problem is presented in line 3 of Listing 2. Beside this operation would not be allowed because a Cell type is replaced with an Array type, this operation highlights another problem: A variable of type Array<Cell< Integer>> is immutable meaning that it is only allowed to alter values within a Cell object and not allowed to replace a Cell object in its entirety. The type differentiation in two dimensional arrays makes working with 2 dimensional a complex endeavour.

Using expressions in array indexing. VerCors in some cases is unable to properly process expressions used for the indexing of arrays. An example of this can be seen in Listing 3. The first expression is taken from the source code of the parallel count algorithm. The second is an alternative notation for the same expression. The two expressions are logically equivalent. The difference is that the first expression does its index calculations when calculating the range of the **forall** expression whereas the second expression indicates a set range and calculates the required offset as part of the indexing expression.

```
1 requires (\forall int i; i >= (partitionSize * tid)
      && i < (partitionSize * (tid + 1)); input[i] >= 0
       && input[i] < radix);
2 requires (\forall int i; i >= 0 && i < partitionSize;
      input[(tid * partitionSize) + i] >= 0 && input[(
      tid * partitionSize) + i] < radix);
```

Listing 3: Two logically equivalent VerCors specifications

However, VerCors will only be able to verify the first expression and give a permission error for the second expression. While the solution to solving these kind of indexing issues is trivial by moving the index expression to the range indication of the **\forall** expression, it is important to be aware of this issue when using the tool.

7. FUTURE WORK

Although this paper answers all the sub-questions in Section 3, there is still work to be done to answer the central research question. This section outlines the next logical steps that can be taken towards answering the central research question.

Firstly, data race freedom of the entire parallel radix sort algorithm needs to be verified. The PVL source code presented in this paper verifies the algorithm until and including the prefix sum step of the algorithm. The current problem is that the algorithm has become too large for VerCors to handle. However, a new tool has been added to the VerCors toolset called SplitVerify which should alleviate this problem. SplitVerify splits up every function in a PVL source file to separate files and verifies every function separately. Using SplitVerify may enable the parallel radix sort algorithm to be verified on data race freedom.

Secondly, functional correctness of both the parallel count and reorder step need to be verified before the functional correctness of the entire parallel radix sort algorithm can be verified. Section 4.2 provides an approach on how this can be done in VerCors for the parallel count step and Section 5.2 outlines possible specifications to use to prove functional correctness of the parallel reorder step.

8. RELATED WORK

Radix sort is an actively researched topic. There are several publications of implementations of parallel radix sort implemented on GPUs. However, these publications often only seem to involve performance analysis without a focus on their correctness [11]. Publications concerning verification and correctness only seem to concern sequential CPU implementations of the algorithm. For example, de Gouw et al. verified a sequential Java implementation of radix sort by annotating the code with JML and providing the code to a semi automated theorem solver called KeY [4]. It seems the verification of parallel radix sort, especially GPU implementations, is a novel topic. Further research into this topic as suggested in Section 7 seems therefore relevant.

9. CONCLUSION

This paper shows how VerCors can be used to verify data race freedom of both the parallel count algorithm and the parallel reorder algorithm. Both of these algorithms are used in parallel radix sort and are therefore important steps towards verifying data race freedom of parallel radix sort. Additionally, this paper presents a possible approach on how to verify functional correctness of the parallel count algorithm with VerCors. Although the central research question remains unanswered in this paper, it does present a road map on the next logical steps to take to answer this question. Lastly, this paper presents some practical issues that are currently present in the VerCors verification toolset.

10. REFERENCES

[1] D. Basu. Parallel radix sort on the GPU using C AMP, Feb 2013.

[2] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990.

[3] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The VerCors tool set: Verification of parallel and concurrent software. *Lecture Notes in Computer Science Integrated Formal Methods*, page 102–110, 2017.

[4] S. de Gouw, F. S. De Boer, and J. Rot. *Verification of counting sort and radix sort*, volume 10001 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2016.

[5] Z. Dominik, P. Marcin, W. Maciej, and W. Kazimierz. Comparison of hybrid sorting algorithms implemented on different parallel hardware platforms. *Computer Science*, 14(4):679, 2013.

[6] T. Harada and L. Howes. Introduction to GPU radix sort. 2011.

[7] R. Inam. *An Introduction to GPGPU Programming - CUDA Architecture*. Mälardalen University, Mälardalen Real-Time Research Centre, 2010.

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

[9] W. Oortwijn. A brief introduction to VerCors, Apr 2018.

[10] M. Scarpino. A gentle introduction to OpenCL, Aug 2011.

[11] D. P. Singh, I. Joshi, and J. Choudhary. Survey of GPU based sorting algorithms. *International Journal of Parallel Programming*, 46(6):1017–1034, 2018.