GPGPU Verification: Correctness of Odd-Even Transposition Sort Algorithm

Yernar Kumashev
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
y.kumashev@student.utwente.nl

ABSTRACT

With the increasing popularity of concurrent programming, many parallel algorithms have been proposed, that are not yet verified. The use of unverified parallel programs in production can result in wrong outputs from the programs leading to undesired situations. This paper discusses the verification of data race freedom and functional correctness of one of the parallel algorithms, the Odd-Even Transposition Sort algorithm, that was not yet verified to the best of our knowledge. The first step is to use Permission-Based Separation Logic supported by verification tool VerCors, to verify that the algorithm is data-race free. The Permission-Based Separation Logic is an extension of Hoare logic and Separation Logic that helps to reason about computer program correctness with the use of pre/post conditions. The next step is to verify the functional correctness of the algorithm, meaning that it outputs the sorted input array. The verification of functional correctness is achieved by adding more annotations to the Permission-Based Separation Logic annotations used to prove data-race freedom.

1. INTRODUCTION

In the fast-paced world, it's no longer performance-wise sufficient to run algorithms on a single CPU. Inputs to functions are enormous and more efficient algorithms are required. Parallel programming is an excellent solution due to the fact that the single task is split into many sub-tasks and executed simultaneously on many cores of GPUs[1]. Such efficient task management helps to achieve significant performance improvements compared to having a single processor running tasks in sequential order. An excellent example of significant performance improvements achieved with the use of parallel programming is the parallel implementation of the Odd-Even Transposition Sort[7], which is verified in this research paper. A sequential implementation of the algorithm runs in O(N^2) time when the parallel implementation runs in O(N) time, where N is the length of the array to be sorted. The importance of sorting algorithms is obvious since they are used not only in almost every web application but in other algorithms as well, such as search and merge algorithms. Therefore, it is important to verify the correctness of parallel sorting algorithms. This research aims to verify the absence of data races[2] and functional correctness[3] of the Odd-Even Transposition Sort. The algorithm will be verified using deductive program verification, which is a verification approach where the source code is appended with pre/post conditions. Pre/post conditions added to the code are based on Permission-Based Separation Logic[1] supported by VerCors[5]. The VerCors tool and the Permission-Based Separation Logic are explained in more details in Section 2.4.

[1]General Processing Units is responsible for splitting a complex problem into large amount of simple tasks performed simultaneously. GPUs are ideal and widely used for graphics in computer games.

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

2. BACKGROUND

This section gives background information about General Purpose GPUs. In addition, it explains in more details the parallel implementation of the Odd-Even Transposition Sort. Finally, there is a brief discussion of the VerCors tool and its logic.

2.1 General Purpose GPUs

If a programmer writes any code in any programming language, it's most likely the written code will run on CPUs (Central Processing Units), due to low popularity of concurrency support by programming languages because of inheritance anomaly[4] [12]. However, the CPUs are not the only processors that can execute code on the computer. There are also Graphics Processing Units (GPUs) that were designed to provide high-quality graphics on computers since they were very cheap and able to manipulate massive data processes in a short time due to its ability to execute commands in parallel. Eventually, people started using GPUs for other programming tasks. Such popularity of GPUs in the computer industry developed a new area of GPGPU(General Purpose GPU) programming. Only a few popular programming languages support GPU programming, such as OpenCL, CUDA, Halide. OpenCL stands out compared to other languages, because it's open-source, meaning it's free and anyone can write and run code on any computers and hardware vendor-independent, unlike CUDA that was created by NVIDIA and works only on NVIDIA hardware. Only the OpenCL GPU kernel programming model will be explained since the CUDA

[2]Data race occurs when 2 or more threads access the same memory location in shared memory simultaneously, when at least one of them has a write permission, leading to memory corruption.

[3]Functional correctness in this research means that the algorithm returns a sorted array.

[4]Inheritance anomaly is the result of conflicts that occur often between inheritance and synchronization constraints of a concurrent object.

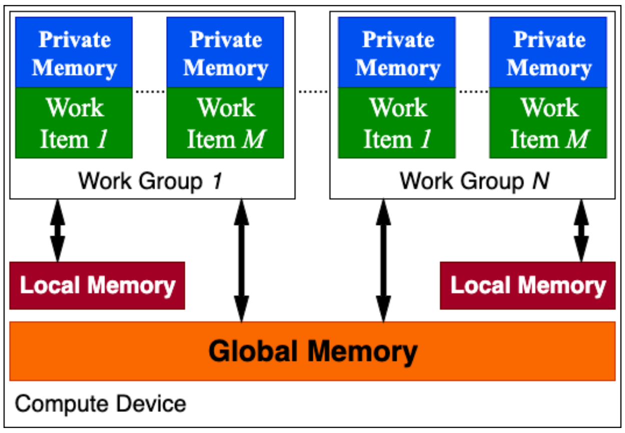

Figure 1. Memory layout of OpenCL GPU

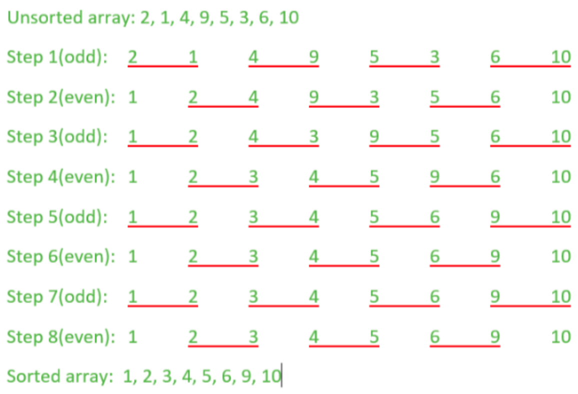
Unsorted array: 2, 1, 4, 9, 5, 3, 6, 10
Step 1(odd): 2 1 4 9 5 3 6 10
Step 2(even): 1 2 4 9 3 5 6 10
Step 3(odd): 1 2 4 3 9 5 6 10
Step 4(even): 1 2 3 4 5 9 6 10
Step 5(odd): 1 2 3 4 5 6 9 10
Step 6(even): 1 2 3 4 5 6 9 10
Step 7(odd): 1 2 3 4 5 6 9 10
Step 8(even): 1 2 3 4 5 6 9 10
Sorted array: 1, 2, 3, 4, 5, 6, 9, 10

Figure 2. Odd-Even Transposition Sort algorithm illustration
Every red line in each step indicate a thread that compares 2 values on top of it and swaps them, if they are out-of-order

model is roughly identical[3]. The memory layout is illustrated in Figure 1. A kernel is an instruction to be executed by work groups. A work group consists of work items (threads). Threads have access to private memory, that other threads can't access and also to the work group memory (local memory), just like other threads inside of that work group. Data in the global memory are accessible by all the threads running the kernel. A thread has access to the unique thread id, that allows it to act individually. The synchronization between threads can be achieved using barriers, such that each thread waits until all other threads reach the same barrier.

2.2 Odd-Even Transposition Sort Algorithm

Odd-Even Transposition Sort is a parallel algorithm with O(N) run-time complexity. It uses the same technique as Bubble Sort by swapping 2 neighbouring elements in the array if they are in the wrong order. However, multi-threading is not supported by the Bubble Sort algorithm. For parallelizing Bubble Sort the odd and even phases are required where:
• **Odd Phase:** Compare every odd element with the next element in the array and swap if it is in the wrong order.
• **Even Phase:** Every even element is compared with the next element and swapped if it is in the wrong order.
Both phases alternate until no more swaps are made. In each iteration every pair of elements are compared and swapped by threads. Figure 2 illustrates how the algorithm sorts the array of size 8.

2.3 Parallel implementation on GPUs

Algorithm 1 shows the pseudo-code of the parallel implementation of the Odd-Even Transposition Sort. This algorithm will be executed by exactly half the size of an input array threads. The memory access by a thread depends on the thread id. Synchronization between threads is achieved using Barriers. Barriers can be found in the pseudo-code in lines 8 and 12 since threads must wait for other threads to be finished in the current phase and only when all threads reached Barrier, they can proceed to the next phase. Even and Odd phases are depicted in lines 5-7 and 9-11, respectively. In every Odd-Even iteration, the *isSorted* Boolean variable is set to true and if any swaps were made during the phase, this variable is set to false, meaning that another Odd-Even iteration must be executed. When no swaps were made, the algorithm terminates, returning the sorted array.

Algorithm 1 Parallel Odd-Even Transposition Sort Algorithm

```
 1: function PARALLEL-OETS(int[] arr, int tid)
 2:     $isSorted = false$
 3:     while $isSorted == false$ do
 4:         $isSorted = true$
 5:         if $arr[tid * 2] > arr[tid * 2 + 1]$ then
 6:             swap $arr[tid * 2]$ and $arr[tid * 2 + 1]$.
 7:             $isSorted = false$
 8:         Barrier(tid)
 9:         if $arr[tid * 2 + 1] > arr[tid * 2 + 2]$ then
10:             swap $arr[tid * 2 + 1]$ and $arr[tid * 2 + 2]$.
11:             $isSorted = false$
12:         Barrier(tid)
```

2.4 VerCors

VerCors is a tool for static verification of concurrent programs. Static verification means that it does not execute the source code of the input program it is verifying, but analyzes if all the specified requirements are satisfied by the program. VerCors supports subsets of high-level languages like Java, C, OpenCL, OpenMP, PVL. PVL (Prototypal Verification Language) is a toy language that was made for VerCors to prototype new verification features. VerCors can be used to verify that the algorithm is data race free and functionally correct. The logic behind VerCors verification is based on Permission-Based Separation Logic and can be achieved by adding annotations to the program. Permission-Based Separation Logic is an extension of the Hoare logic[8] and Separation Logic[13] made for reasoning about computer program correctness.To prevent a thread from invalidation of properties of other threads, a strong notion of ownership in the form of permission is used in Permission-Based Separation Logic. Reading from or writing to shared memory for each thread is possible only if they have permissions to do so. Permissions can be specified using "write"/"read" keywords or with any decimal between 0 and 1 (excluding 0 and 1) for a read permission and with 1 for a write permission. Explicit ownership handling guarantees the data-race freedom and memory safety of the program.
List 1 illustrates a sample program written in PVL to show how Permission-Based Separation Logic handled in VerCors. Given class *Counter* with integer property *val* and a function *incr* that increases the *val* by given number n, with 3 pre/post conditions (Lines 4-6 in List 1). The first annotation is a pre-condition, specifying that the thread

can call this function only if it has a write permission.

List. 1. VerCors Permission-Based Separation Logic

```
class Counter {
    int val;
    /*
    requires Perm(this.val, 1);
    ensures Perm(this.val, 1);
    ensures this.val == \old(this.val) + n;
    */
    void incr(int n) {
        this.val = this.val + n;
    }
}
```

The second annotation is a post-condition, ensuring that the thread who made the call still has a write permission when the function terminates. The final annotation is another post-condition, checking that the *val* was indeed incremented by n. Checking the program with VerCors compiler without all 3 annotations will result in 'AssignmentFailed' error. If each thread has correct permission throughout the whole program, that means the program is data race free. In this example, the first 2 annotations handle permissions correctly resulting in data race freedom verification. The last annotation verifies the functional correctness of the algorithm, by specifying how exactly the value has changed.
Ghost variables play an important role in the verification of the functional correctness. We use ghost variables to explicitly keep track of program history. They are part of the verification and not the program. The proof of functional correctness is achieved first on ghost variables, then ghost variables are related to concrete variables in the program.

3. METHODOLOGY

3.1 Research Questions

The research aims to answer 2 research questions.

1. How to prove that the Odd-Even Transposition Sort algorithm is data race free?

2. How to prove the functional correctness of the Odd-Even Transposition Sort algorithm?

3.2 Approach

For verification of the data race absence in the Odd-Even Transposition Sort algorithm, the VerCors verification tool is used. The pseudo-code of the parallel implementation of the algorithm illustrated in Algorithm 1 is converted to PVL. To verify that the algorithm is data race free, the PVL code is augmented with pre/post-conditions in Permission-Based Separation Logic and successfully verified with VerCors verifier. The verification of the functional correctness can be achieved by adding more annotations to already exiting annotations written to verify data race freedom.

4. ALGORITHM VERIFICATION

In the following paragraph, the verification of data-race freedom and functional correctness of the algorithm is described. Small code snippets will be used to have a better understanding of verification steps.

4.1 Data-Race Freedom

To prove that the algorithm is data race free, the read and write permissions need to be specified for each thread, so the situation when 2 or more threads modify the same shared memory location does not occur. Algorithm 1 is in-place and every thread has permissions to both read from and write to 2 elements in shared memory location during each phase, therefore we don't use read permission at all. A number of threads needed for sorting an input array equal to half the length of the input array, because each thread during even or odd phase has access to 2 shared memory locations. Each thread has a unique identifier *tid* that allows all threads access and modify different elements in the input array. The sample PVL code with annotations specifying permissions for threads is shown in List 2.

List. 2. Permission scheme for data-race freedom verification.

```
 1  /*
 2  loop_invariant tid*2 < arr.length ==> Perm(arr[tid
        *2], write);
 3  loop_invariant tid*2 +1 < arr.length ==> Perm(arr[tid
        *2+1], write); */
 4  while(!isSorted){
 5      isSorted = true;
 6    //even
 7    if(tid*2+1 < arr.length && arr[tid*2] > arr[tid
         *2+1]) {
 8        int temp = arr[tid*2];
 9          arr[tid*2] = arr[tid*2+1];
10          arr[tid*2+1] = temp;
11        isSorted = false;
12    }
13    barrier(fwd) { /*
14      requires tid*2 < arr.length ==> Perm(arr[tid*2],
           write);
15      requires tid*2 +1 < arr.length ==> Perm(arr[tid
           *2+1], write);
16      ensures tid*2 +1 < arr.length ==> Perm(arr[tid
           *2+1], write);
17      ensures tid*2+2 < arr.length ==> Perm(arr[tid
           *2+2], write); */
18    }
19      //odd
20      ....
21      barrier(fwd) { /*
22      requires tid*2+1 < arr.length ==> Perm(arr[(tid*2)
           +1], write);
23      requires (tid*2+2 < arr.length) ==> Perm(arr[tid
           *2+2], write);
24      ensures tid*2 < arr.length ==> Perm(arr[(tid*2)],
           write);
25      ensures tid*2+1 < arr.length ==> Perm(arr[(tid*2)
           +1], write); */
26    }
27  }
```

The if-statement for odd phase (Line 20 in List 2) is skipped due to it's complete equivalence to if-statement in even phase with only difference in a reference to elements in the shared memory. During the even phase, each thread has a write permission to *arr[tid*2]* and *arr[tid*2 +1]* (Lines 2-3) where *arr* is the input array. This annotation is written as *loop_invariant*[5], since thread needs to acquire the same write permission that it had when entering the loop iteration, when loop iteration ends. In the odd phase, each thread loses write permission to *arr[tid*2]*, but still

[5]Invariant that must hold on every iteration of the loop.

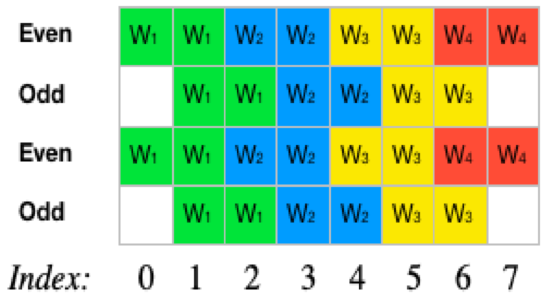

Figure 3. Permission distribution over threads. Illustration of thread allocation for 4 threads in an array of length 8. This figure demonstrates what elements each thread has access based on its *tid*, depending on the phase its in. Wi represents thread with *tid* equals i that has write permissions for 2 elements. Each thread is depicted with a different colour.

	$[2, 1, 4, 9, 5, 3, 6, 10]$
Initial Inversion	$[1, 0, 1, 3, 1, 0, 0, 0]$
	$[1, 2, 4, 9, 3, 5, 6, 10]$
Inversion after even phase	$[0, 0, 1, 3, 0, 0, 0, 0]$
	$[1, 2, 4, 3, 9, 5, 6, 10]$
Inversion after odd phase	$[0, 0, 1, 0, 2, 0, 0, 0]$

Figure 4. Example inversion change after even and odd phases.

has a write permission to *arr[tid*2 +1]* and gains new write permission to *arr[tid*2 +2]* (Lines 16-17), therefore we update the permission scheme in the barrier. Barriers are used in the algorithm for synchronization, so when the threads that are finished with their tasks early can wait for other threads to finish. With the help of the first barrier (Lines 13-18 in List 2), the permissions of the threads are updated by shifting indicators one position to the right, preparing threads for the odd phase. The second barrier (Lines 21-26 in List 2) is needed to shift the indicator one position to the left, so when the even phase starts, there is no conflict and each thread has access to the right elements in the shared memory. Figure 3 visualizes thread permission pattern in more details. Boundary checks of indicators are important and taken into account since as shown in Figure 3, when the input array has even length, the last thread is idle during the odd phase. The absence of boundary checks will invalidate verification.

4.2 Functional correctness

To verify functional correctness of Odd-Even Transposition Sort algorithm, we need to verify that the algorithm returns a sorted array when it terminates. For verification, the concept of inversion[6][p. 41] is used. Inversion is one way of measuring the level of disorder in an unsorted array. Given input array *arr* inversion count for an element at index i is the number of smaller elements from *arr[i+1]* to *arr[arr.length-1]*. Inversion is an array that contains the inversion count for all elements in the input array. The following equation is how the inversion array can be found:

$$\forall i.\ 0 \le i < |xs| \sum (\forall j.\ i < j < |xs| \ \ (xs[j] < xs[i])?\ 1 : 0)$$

For example, for the input array in Figure 1 ([2, 1, 4, 9, 5, 3, 6, 10]), inversion is [1,0,1,3,1,0,0,0]. It can be counted that the number 9 at index 3 has inversion count 3, because there are 3 numbers (specifically 5,3,6) smaller than 9 to its right, and number 6 at index 6 has inversion count that equals 0, because there are no smaller values to its right. Figure 4 displays the initial inversion sequence of an input array and then the updated inversion sequence after odd phase and even phase. If we observe how the inversion sequence is changing after each step, it can be concluded that when the inversion count for an element equals 0 then there is no bigger number to its right. Another observations is that if sum of all numbers in inversion sequence equals 0 then all the elements in the input array are in the right order. Finally, if *isSorted* equals true meaning that the algorithm is terminated then the intSum[6] of inversion sequence equals 0. These observations provide 3 different properties that need to be verified to prove the functional correctness of the Odd-Even Transposition Sort algorithm. The following steps are needed to verify the algorithm:

1. Define ghost variables to store the history of values.
2. Define functions to update the ghost variables.
3. Prove observed properties over the ghost variables.
4. Relate ghost variables with concrete variables in the program.

Ghost variables.
The first ghost variable is a sequence named *hist_seq* that stores the same elements in the same order as in the input array. This ghost variable is mainly created to keep the history of the input array and it is updated together with the input array, so there is no necessity in helper function. The second ghost variable stores inversion count for each element in an *inversion* sequence. Every element with index i from 0 to *arr.length* from the input array has its inversion count stored in *inversion[i]*. Updating *inversion* sequence is more complicated compared to updating *hist_seq*, therefore 2 functions were created: one to count the number of inversions for a single element and another one to count inversion count for all elements in the array with the help of the first function. The first function is called *Count_inversion* and depicted in List 3. The function receives a copy of the input array in *hist_seq*, *currIndex* that tells the function from which position to start counting inversion and i that helps to navigate through the *hist_seq*, because *Count_inversion* is a recursive function. The first post-conditions in line 5-6 in List 3, verify if the result equals 0 then all the numbers to the right from the *currIndex* must be bigger or equal to the element at *currIndex* and vice versa. If the result is more than 0 then there is at least 1 element smaller than the element at *currIndex* and vice versa as specified in lines 7-8 in List 3.

List. 3. Recursive function to calculate the number of inversion for an element at *currIndex*

```
/*@
```

[6]intSum is a helper function that sums all the elements in a given sequence.

```
2  requires 0 <= currIndex && currIndex < |histSeq|;
3  requires 0 <= i && i <= |histSeq|;
4  ensures 0 <= \result && \result <= |histSeq| - i;
5  ensures (\result == 0) ==> (\forall int j; i <= j &&
       j < |histSeq|; histSeq[j] >= histSeq[currIndex]);
6  ensures (\forall int j; i <= j && j < |histSeq|;
       histSeq[j] >= histSeq[currIndex]) ==> (\result ==
        0);
7  ensures (\result > 0) ==> (\exists int k; i <= k && k
        < |histSeq|; histSeq[k] < histSeq[currIndex]);
8  ensures (\exists int k; i <= k && k < |histSeq|;
       histSeq[k] < histSeq[currIndex]) ==> (\result >
       0); @*/
9  static pure int Count_inversion (seq<int> histSeq,
       int currIndex, int i) =
10   i < |histSeq| ? (histSeq[currIndex] > histSeq[i] ? 1
         + Count_inversion(histSeq, currIndex, i+1) :
11     Count_inversion(histSeq, currIndex, i+1)) : 0;
```

To update the *inversion* ghost variable another recursive function is needed that calculates inversion count to all the elements in the *hist_seq*. This function is named *Count_all _inversions* and shown in List 4. Similar post-conditions as in *Count_inversion* are specified for this function, only difference is that in *Count_all_inversions* we apply result checks to all the elements in *hist_seq* instead of a single one. Example outputs of the *Count_all_inversions* function are depicted in Figure 4.
The next step after defining functions to update ghost variables, is to verify the properties over ghost variables. *Count_all_inversions* function annotations help to verify the following property:
Property 1 For any sequence xs:

$$\forall i.\ 0 \leq i < |xs|\ [xs[i] == 0 \longrightarrow \forall j.\ i < j < |xs|$$

$$hist\_seq[i] <= hist\_seq[j]]$$

Property 1 states that for every element in the *inversion* sequence, if the inversion count of any element equals 0, then there is no smaller element to its right in *hist_seq*. This property is verified in the post-condition of the current function in line 5 in List 4 and the post-condition that proves the same property vice versa is shown in line 6 in List 4. Another post-conditions verifying the situation when the inversion count is more than 0 are in lines 6 and 7 and were added to strengthen the verification of the Property 1.

List. 4. Recursive function to calculate the inversion count for all elements in `hist_seq`

```
1  /*
2  requires 0 <= i && i <= |hist_seq|;
3  ensures |\result| == |hist_seq| - i;
4  ensures (\forall int x; x >= 0 && x < |\result|; (\
       result[x] >= 0 && \result[x] <= |hist_seq| - i));
5  ensures (\forall int x; x >= 0 && x < |\result|; \
       result[x] == 0 ==> (\forall int j; x+i+1 <= j &&
       j < |hist_seq|; hist_seq[j] >= hist_seq[x+i]) );
6  ensures (\forall int x; x >= 0 && x < |\result|; (\
       forall int j; x+i+1 <= j && j < |hist_seq|;
       hist_seq[j] >= hist_seq[x+i]) ==> \result[x] == 0
        );
7  ensures (\forall int x; x >= 0 && x < |\result|; \
       result[x] > 0 ==> (\exists int k; x+i+1 <= k && k
        < |hist_seq|; hist_seq[k] < hist_seq[x+i]) );
8  ensures (\forall int x; x >= 0 && x < |\result|; (\
       exists int k; x+i+1 <= k && k < |hist_seq|;
       hist_seq[k] < hist_seq[x+i]) ==> \result[x] > 0);
       */
9  static pure seq<int> Count_all_inversions (seq<int>
       hist_seq, int i) = i < |hist_seq| ? seq<int> {
       Count_inversion(hist_seq, i, i+1)} +
       Count_all_inversions(hist_seq, i+1) :seq<int>{};
```

Another property to be verified from the observation is:
Property 2 For any sequence xs:

$$[\forall i.\ 0 \leq i < |xs|\ [xs[i] == 0] \longrightarrow \forall j.\ i < j < |xs|$$

$$hist\_seq[i] <= hist\_seq[j]$$

Property 2 claim that if every element in the inversion sequence equals 0, then for every number in *hist_seq* there is no bigger number to the right of it. In other words, if intSum of inversion sequence equals 0, then the numbers in *hist_seq* are all in the right order. To verify this property another function illustrated in List 5 was defined. The function named lemma receives a *hist_seq* as *xs* sequence and *inversion* as a *ys* sequence. The post-condition in line 4 in List 5, verifies this property.

List. 5. Recursive function to calculate the inversion count for each element in `hist_seq`

```
1  /*
2  requires |xs| == |ys|;
3  requires (\forall int j; j >= 0 && j < |ys|; ys[j] ==
        0 ==> (\forall int k; k >= j+1 && k < |xs|; xs[j
      ] <= xs[k]));
4  ensures intsum(ys) == 0 ==> (\forall int j; j >= 0 &&
        j < |ys|; (\forall int k; k >= j+1 && k < |xs|;
      xs[j] <= xs[k])); */
5  void lemma(seq<int> xs, seq<int> ys){}
```

The final property is:
Property 3 For any sequence xs:

$$isSorted == true \longrightarrow [\forall i.\ 0 \leq i < |xs|\ [xs[i] == 0]$$

Due to the fact that the algorithm is terminated only if the Boolean *isSorted* equals true, the property to be verified is if *isSorted* equals true, then the intSum of an *inversion* sequence must be 0. This is the final property before achieving complete verification of the functional correctness of the algorithm over ghost variables. However, because of the short amount of time allocated for this project, the proof of final property has not been achieved.
Relating ghost variable to a concrete program variable.
The *hist_seq* ghost variable is the only variable we can relate to a program variable because *inversion* ghost variable is created only to achieve the verification of the algorithm and cannot be not related to any variable in the program. To relate *hist_seq* with arr which is an input array, we specify both pre and post conditions such that whenever any thread has access to a shared memory location, it verifies if the number in the shared memory equals to the value in the *hist_seq*. The annotation is added as an *loop_invariant* and demonstrated in Line 4 in List 6.

List. 6. Recursive function to calculate the inversion count for each element in `hist_seq`

```
1  /*
2  loop_invariant Perm(arr[tid*2], write);
3  loop_invariant Perm(arr[tid*2+1], write);
4  loop_invariant (\forall* int i; 0 <= i && i < arr.
      length/2; hist_seq[i*2] == arr[i*2] && hist_seq[i
      *2+1] == arr[i*2+1]); */
5  while(!isSorted){...}
```

5. FUTURE WORK.

Due to the short amount of time assigned to this paper, some steps still need to be taken to prove the functional correctness of the algorithm. Currently, Property 3 is untouched and it is one of the hardest properties to prove. The source code can be found in footnote[7]. The function to update inversion inside the while-loop is already written, but annotations failed verification check, so these annotations need to be fixed and added to the program as a loop-invariant.

6. RELATED WORK.

VerCors, PUG[10], GPUVerify[4] are few of several verification tools that support static verification of the GPGPU programs. However, only the VerCors tool allows to reason about functional correctness of the GPGPU programs. VeriFast[9] is similar to VerCors verification tool, with only difference that VeriFast can only verify single-threaded and multi-threaded programs written in Java and C.
There are many proposed parallel sorting algorithms, most popular are Bitonic Sort, Radix Sort and Merge Sort [2]. Correctness proof of Bitonic Sort has been achieved by proving axioms and inference rules defined based on the observations of the algorithm.[11] However, this proof is only formal and has not been verified using a verification tool. To the best of our knowledge, small amount of research is done regarding the parallel sorting algorithm verification. This is mainly, due to its complexity and lack of supported features in verification tools. Despite the fact, that there is not much work regarding parallel sorting algorithm verification, there is a prefix sum verification of data race freedom and functional correctness achieved using VerCors, that can be used to verify Radix Sort[14].

7. CONCLUSION

This paper shows the verification of Odd-Even Transposition Sort for data race freedom and functional correctness. First using VerCors and Permission-Based Separation Logic the algorithm was verified to be data race free. Next step using the same VerCors verifier, verification of functional correctness of the algorithm was attempted, but due to time constraints it is not finished. For verification of functional correctness, the concept of inversion was used and 3 properties were defined based on observations. Property 1 and 2 were proven over ghost variables, but Property 3 still needs to be verified. We believe that this paper is the first attempt to verify Odd-Even Transposition Sort and the concept of inversion might be useful not only to verify Odd-Even Transposition Sort algorithm, but to verify other parallel sorting algorithms as well. VerCors aims to automate the process of proof creation and this paper is a small contribution to this goal.

References

[1] A. Amighi, C. Haack, M. Huisman, and C. Hurlin. Permission-based separation logic for multi-threaded java programs. *Logical methods in computer science*, 11(3):2, 2 2015. eemcs-eprint-25324.

[2] H. S. Azad. *Advances in GPU Research and Practic.* Morgan Kauffman, 2017.

[3] E. Bardsley and A. F. Donaldson. Warps and atomics: Beyond barrier synchronization in the verification of gpu kernels. 2009.

[4] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. Gpuverify: a verifier for gpu kernels. pages 113–132, 2012.

[5] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The vercors tool set: Verification of parallel and concurrent software. In N. Polikarpova and S. Schneider, editors, *Integrated Formal Methods*, Lecture Notes in Computer Science, pages 102–110. Springer, 2017.

[6] T. H. Cormen, C. E. Leiserson, and C. S. Ronald Linn Rivest. *Introduction to Algorithms.* The MIT Press, 2009.

[7] A. N. Habermann. *Parallel neighbor-sort (or the glory of the induction principle).* Carnegie Mellon University, 1972.

[8] C. A. R. Hoare. An axiomatic basis for computer programming. 1969.

[9] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. in: Nasa formal methods symposium. pages 41–55, 2011.

[10] G. Li and G. Gopalakrishnan. Scalable smt-based verification of gpu kernel functions. in: Sigsoft fse. pages 187–196, 2010.

[11] H. Lutfiyya and B. McMillin. Formal generation of executable assertions for a fault-tolerant parallel bitonic sort. 1991.

[12] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages.

[13] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. 2002.

[14] M. Safari. Formal verification of parallel prefix sum. 2019.

[7] $https://figshare.com/articles/oddEven\_pvl/11725848$