Verification of the Parallel Prefix Sum Program with VerCors

Thijs Wiefferink
thijs@wiefferink.me, s1366564

University of Twente

Abstract

This project ~~continues the verification of~~ proves a parallel Prefix Sum ~~program. The~~ algorithm data race free using VerCors. The used Prefix Sum program is intended for running on a GPU using OpenCL~~, the used algorithm is suitable for this purpose~~. The verification is done using ~~Permissions-Based~~ Permissions Based Separation Logic and PVL (a toy language for verification purposes) to create a specification and VerCors to verify this specification. ~~The~~ By specifying permissions the Prefix Sum program has been proven data race free in this project. ~~The result~~ This is a step towards verifying larger and more complex programs with VerCors, while identifying usability and performance problems in the tool. The results of the program has not yet been verified, but a specification has been made to start with this functional verification. During the project improvements have been made to VerCors by ~~the author Stefan Blom~~ its developers to support this research and make future ~~use~~ usage easier.

Keywords

VerCors, Permission-Based Separation Logic, GPU, OpenCL, verification, specification

~~Preface~~

Contents

1 Introduction

This project is the continuation of my bachelor thesis project, in which the verification of the prefix sum algorithm has been started. Since only the 'data race free' part has been proven for the first part of the algorithm (the upsweep), the goal of this project is to prove the complete algorithm data race free, and additionally try to prove the functionality of the algorithm.

The necessary background information will be included in this report to understand the goal and results of the project, but full details of the Bachelor project can be read in the paper of that project [7].

2 ~~Introduction~~

This chapter describes the research domain, shows the problem that is solved, introduces the research questions and explains the approach.

1.1 Research domain

In this section the context information of this research project is described.

1.1.1 GPU computing

A graphics processing unit (GPU) is a device designed to rapidly manipulate and alter memory to accelerate the creation of images, for example while watching a video or playing a game. However, GPUs are also used more for general purpose computing, which is traditionally handled by the central processing unit (CPU). GPUs are better than CPUs doing parallel execution on large data sets. For example increasing the brightness of an image is easily done by a GPU, since this operation can be done in parallel on all pixels of the image. GPUs are however also used for physics calculations or mining crypto currencies. When using a GPU for general computing an API has to be used, I have chosen OpenCL for the previous research project because of its hardware vendor independency and open source nature.

Running a parallel computation on a GPU brings a couple of challenges. The first challenge is preventing data races. A data race is the situation in a program where multiple threads are accessing the same memory location, with at least one of them writing to the location. The second challenge is verifying the correctness of the functionality of the program. Verifying both of these aspects is useful for safety critical systems.

~~The previous research project has started~~ My Bachelor thesis research project made a start with the verification process to show that a program (specifically the Prefix Sum algorithm) has no data races. Since only the first half of the program could be verified in the given time for the project the verification has been continued in this project.

1.1.2 Prefix Sum

To show what the prefix sum is and how it is calculated, this section repeats the information from the previous project[7] below.

The algorithm computes the sums of all possible prefixes of an input array. In Figure 1, a mathematical representation of the prefix sum is illustrated, x represents the input array, x_0 indicates the first element from the input array, where n represents the size of the input array, and y is the output array containing the prefix sums. Each number y_a in the output array is the sum of all numbers $x_b \in x$ for which the condition $b < a$ holds.

The Prefix Sum algorithm is an interesting case study because it is a building block for a lot of other algorithms. For example radix sort and quicksort can be implemented using Prefix Sums, but it can also be used to lexically compare strings of characters or to search for regular expressions [2]. In the field of specifying and verifying GPU programs the Prefix Sum is a suitable next step,

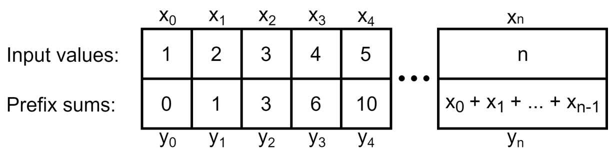

Figure 1: Prefix Sum description

because it will be a bigger and more complex example of verifying a GPU program.

The algorithm to calculate prefix sums can be structured in such a way that large amounts of data can be processed in parallel. A multi threaded algorithm meant for the GPU has been implemented in the previous project. The verification process continues with this same algorithm. The implemented version of the Prefix Sum is based on Chapter 39 *Parallel Prefix Sum (Scan) with CUDA* of the book GPU Gems 3 [6].

1.1.3 Verification

For the verification of the Prefix Sum program Permission-Based Separation Logic is used to specify the behavior of the program, and the tool VerCors is used to verify that the code matches the specification. A description of Permission-Based Separation Logic can be found in the previous project[7].

1.2 Problem statement

Making sure a program has no data races, and always gives correct results is extremely hard, if not impossible with traditional testing. For a single-threaded application testing all inputs should prove that the program is correct, but for multi-threaded applications this does not prove anything about data races. In order to verify the correctness of a program it ~~has to~~ could be specified in Permission-Based Separation Logic, which is a challenge for bigger programs like the Prefix Sum. VerCors, the tool used to verify the specification, will get slower when a larger specification is used. This makes using the tool correctly and efficiently a challenge. During the previous project a number of problems in the tools have been identified that blocked the verification, during this project this will be the case as well.

1.3 Research questions

The following research questions have been formed from the problem statement:

1. How can the Prefix Sum program be proven to have no data races?
2. How can the Prefix Sum program be proven to give the correct result?
3. What are the limitations of VerCors for verifying GPU programs?

The first research question has partially been proven by the previous research, which will be used as a starting point. To answer it fully the specification of the Prefix Sum algorithm has to be extended to the complete program (add the downsweep ~~and final permissions~~permissions, ensures clauses for the threads and the ensures clauses of the kernel). For the second research question the specification created for the first question has to be extended with information about the results. The third question is to review the VerCors tool, which will be done during the verification process of the first two questions.

1.4 Approach

First the last specification of the previous project will be tested in the last version of VerCors, to ensure it still works after all changes in the tool. After that works, the verification of the read/write permissions can be continued for the downsweep phase of the program. I expect that the specification should not be hard, because it uses much of the same patterns of the upsweep phase, but getting VerCors to actually verify it will take time. After the downsweep phase is specified and verified, the `ensures` clause of the complete program can be added, which might give some trouble to proof as well. While adding more specifications I expect the verification time to increase. ~~Hopefully this does rise to a time that makes iterating through different versions of the specification too slow.~~

After the read/write permissions have been verified, the functional specification and verification can be started. To speed up this process it is probably best to remove the downsweep phase at first, and just start with the upsweep phase. This will keep the iteration time low and should make it easy to rapidly expand the specification. Doing the verification of the downsweep phase might get slow due to the verification time of VerCors. Adding the final ensures for the complete program might be the hardest task, since ~~the~~ everything needs to be enabled at that point.

To answer the third research question notes will be made during the verification process to keep track of tool improvements and limitations. These will be written down to provide future work for the tool, or notes for creating specifications that are supported by VerCors.

When encountering a problem during the verification, Stefan Blom (author of VerCors) will be contacted to see if there is a problem in VerCors and if/how that could be solved.

1.5 Report structure

Firstly Chapter 2 explains the prefix sum algorithm to know the program that will be verified. After that the verification of the permissions of the program will be done in Chapter 3. Chapter 4 builds on the permissions with the functional verification. The results are discussed in Chapter 5 and the conclusions are drawn in Chapter 6.

2 Prefix sum algorithm

The sections below are repeated from the previous project[7], describing the algorithm that is used for calculating the Prefix Sum in parallel.

2.1 Prefix Sum algorithm

The basic single thread Prefix Sum algorithm is simple, but cannot be used with multiple threads. The trivial way to compute Prefix Sums would be to compute it as described in Listing 1. In the loop body of this algorithm it depends on knowing the result of the previous sum, because of this data dependency the algorithm cannot be used to calculate Prefix Sums concurrently.

Listing 1: Single thread Prefix Sum

```
result[0] := 0
for a := 1 to n do
result[a] := input[a-1] + result[a-1]
```

The algorithm implemented for this research is made by Blelloch ~~[?]~~[?], and can be used concurrently. The description of Nguyen [6] has been followed to implement the algorithm. The *Simple-OpenCL* library [5] of O. A. Huguet and C. G. Marin has been used to implement the OpenCL kernels. The algorithm by Blelloch performs the Prefix Sum calculation in two phases, the up-sweep phase and down-sweep phase. The algorithm uses a balanced binary tree for data storage, therefore a tree with $log_2(n)+1$ levels is required to accommodate for an input of size n. If the input size is not a power of 2, then the input will be padded with zeros until it is; this is necessary because the algorithm requires a binary tree. The tree has $d = log_2(n)+1$ levels, and each level d has 2^d nodes. At the start the input values will be placed in the n leaves at the bottom of the tree, see Figure 2. The up-sweep phase traverses the tree from the leaves to the root, level by level, and computes partial sums in the nodes of the tree. At each level the sum of 2 nodes is computed and placed in the node above, at the end the root node contains the sum of all elements in the input. Figure 3 shows the end result of this phase.

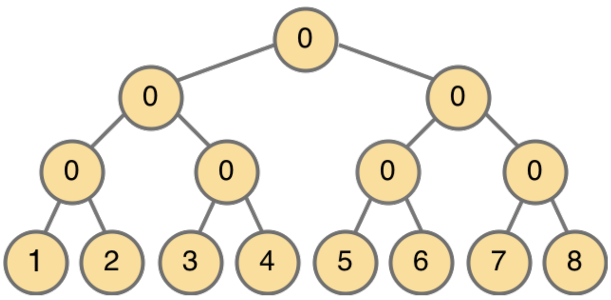

Figure 2: Tree at the start of the up-sweep phase

After this first phase the second phase will start, called the down-sweep phase. This phase starts by inserting zero at the root of the tree, after that it traverses

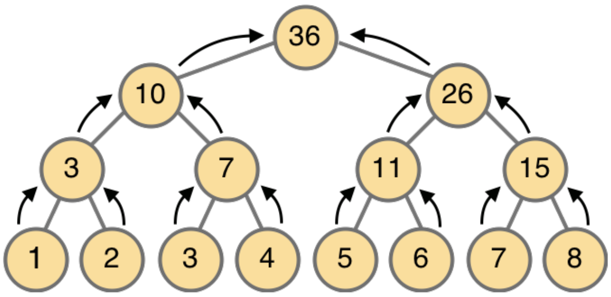

Figure 3: Tree after the up-sweep phase

the tree from the root to the leaves. On each level the right child will be set to the sum of the left child and the current node, and the left child will be set to the value of the current node. This way the zero that has been inserted at the root will travel to the leftmost leaf, and intermediate sums will travel to the right, and get added to form the final result. Figure 4 illustrates the first step, the right node will get the value $0 + 10$, the left node will get value 0. Figure 5 shows the result of the second step, and Figure 6 shows the end result, with an exclusive prefix sum in the leaves of the tree. An exclusive prefix sum means that each output value is the sum of all inputs with a lower index, instead of a lower or the same index as with an inclusive prefix sum.

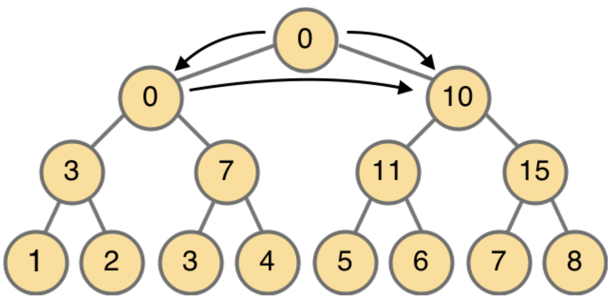

Figure 4: The first step of the down-sweep phase

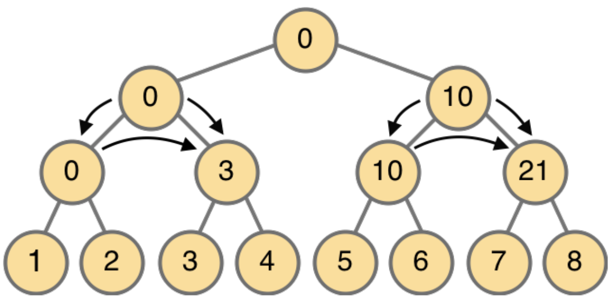

Figure 5: The second step of the down-sweep phase

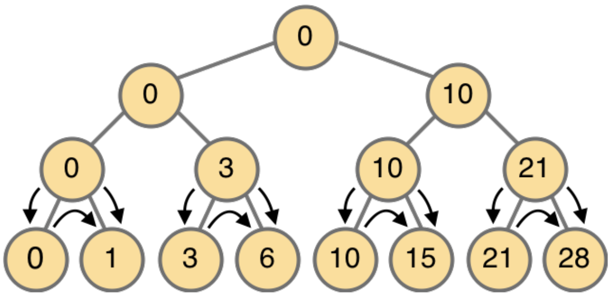

Figure 6: The final step of the down-sweep phase

2.2 Two-dimensional Arrays As Storage

Now that the algorithm of the Prefix Sum has been explained in Section 2.1, the storage of the tree in memory will be looked at. A common way to store a binary tree in an array is to have the root node at index $i = 1$, with the left child at $i * 2$ and the right child at $i * 2 + 1$. Such a storage solution would require an array of twice the size of the input, which is the minimum required size for this algorithm to work. The disadvantage of this storage type is that at each level of the tree in the up- and down-sweep phase only a part of the threads running the kernel are active. To illustrate, in the example of Figure 3 about the up-sweep phase, 4 threads would be required to run this kernel, with 4 of them active on the lowest level, then 2 on the level above, and 1 for the highest level. To make the correct threads active and idle, the kernel has code that shuts off certain threads in certain iterations of the up-sweep phase. This code causes branch divergence, which means that certain threads of a kernel are running different code as other threads. Because threads run different code, the SIMT principle is disturbed.

To prevent the problem mentioned above a different version of the Prefix Sum algorithm has been implemented that uses another storage solution for the binary tree. The algorithm has a two-dimensional array, the first array stores an array for each level of the tree, and those levels have values for each node of the tree. The two-dimensional array has a height of $log_2(n)+1$, and a width the same as the inputsize n. The nodes of a tree are aligned to the left in the level arrays, which leaves blank spots on all levels except the lowest one. Because of these blank spots we can now let all threads do the calculation as explained in Section 2.1. The threads that normally would have been idle will now perform operations on the blank spots of the two-dimensional array, which do not interfere with the actually useful calculations. This change has a positive effect on the performance of the kernel because of reduced branch divergence [4]. The kernel of the one-dimensional array would need to be split each time it is executed, since there are threads executing different code. But this is not the case for the two-dimensional array version, in which all threads do exactly the same operations (although on different data). The previous research project has benchmarked the two solutions and confirmed that the two-dimensional array has better performance.

2.3 Algorithmic description

The algorithm with the two-dimensional arrays works as described in Listing 2 (up-sweep phase) and Listing 3 (down-sweep phase). The loops at respectively line 2 and line 3 of these algorithms are to indicate that one work item of the GPU will do the calculation inside the loop. The arrays used in the algorithms have their first dimension represent the level of the tree, and their second dimension the node of tree on the given level. The algorithms assume that the values of the nodes of the tree are stored as much to the left as possible, so for example the root of the tree has 0 as the second dimension of the array, and the highest possible number on the first dimension: $log_2(n) - 1$.

Listing 2: Upsweep phase

```
1 for d=1 to log2(n) do
2 for all k=0 to n-1 in parallel do
3 x[d][k] := x[d-1][k*2] + x[d-1][k*2+1]
```

Listing 3: Downsweep phase

```
1 x[(log2(n)-1)*n] := 0
2 for d=log2(n)-1 to 1 do
3 for all k=0 to n-1 in parallel do
4 x[d-1][k*2+1] = x[d-1][k*2] + x[d][k]
5 x[d-1][k*2] = x[d][k]
```

3 Verifying permissions

This chapter describes the process of verifying the read/write permissions of the Prefix Sum program. It starts with the result of the previous research, expands it to the downsweep phase, after which the specification is simplified.

3.1 Starting point

Listing 4 shows the specification as was made during the previous project. This specification has been verified until the end of the upsweep phase. The specification of the downsweep phase has been written down, ~~as well as the ensures for the complete program, but these could not be verified by VerCors yet at that time.~~ but could not yet be verified by VerCors because it was not 100% correct and complete. The specification was missing verification of the permissions required to start the downsweep phase, and was quite verbose.

The specification starts with a class definition. Inside this class definition there is one method, called `prefixSum`. This method will do the prefix sum algorithm on the provided input. The method has a couple parameters, these have the following purpose:

1. **N**: The size of the input (must be a power of two)
2. **H**: $log2(N) + 1$, the height of the temporary array
3. **input**: Array of size `N`, containing the input numbers
4. **output**: Array of size `N`, meant for the result of the algorithm
5. **tree**: Two-dimensional array of size `N*H`, used during the computation

The output and tree array are explicitly given as parameters to the method, since in OpenCL the host (CPU program) would declare the required memory blocks, which get used in the GPU kernel. The kernel method has a couple of conditions, starting with static numbers for `N` and `H`. These are declared like this because VerCors can not handle declaring that `N` should be a power of two, and `H` needs to be $log2(N) + 1$. After this permissions to the complete input, tree and output arrays are given to the method.

Next at line 19 a `par` block is used to declare that the following code will run in parallel on multiple threads, like a GPU does. This block specifies how many threads are used, and provides the `t` variable to reference the number of the current thread. `requires` and `ensures` conditions in the `par` block declare which permissions are assigned to each thread.

Now the code for each thread is specified. It starts by copying the input values to the bottom of the tree array. After that the upsweep starts. The `loop_invariants` declare the permission layout of the tree array, and how it changes while going through the loop. Inside the loop there is a barrier at line 56. This barrier ensures that all GPU threads synchronize, making sure all previous code is executed on all threads. The barrier changes the permissions layout of the tree array, ensuring it is ready for the next row.

Before the downsweep starts, the top row of the tree array is set to zero. After that the loop of the downsweep is listed. The specifications at lines 76 until 105 are not verified by the tool, but were the best guess as done in the Bachelor project.

This specification is a ~~good~~ starting point for completing the read/write permissions verification. The process of extending this specification is described in the next section.

Listing 4: Specification as written in the previous project, only the upsweep part is verified

```
class Ref {
   // Requires/Ensures for the complete kernel
   requires N==32 && H==6;
   requires (\forall* int i; 0<=i && i<N; Perm(input[i], read));
   requires (\forall* int i; 0<=i && i<N; Perm(output[i], write));
   requires (\forall* int i; 0<=i && i<N;
      (\forall* int j; 0<=j && j<H; Perm(tree[j][i], write))
   );
   ensures N==32 && H==6;
   ensures (\forall* int i; 0<=i && i<N; Perm(input[i], read));
   ensures (\forall* int i; 0<=i && i<N; Perm(output[i], write));
   ensures (\forall* int i; 0<=i && i<N;
      (\forall* int j; 1<=j && j<H; Perm(tree[j][i], write))
   );
   ensures (\forall* int i; 0<=i && i<N; Perm(tree[0][i], write));
   // Kernel method
   void prefixSum(int N, int H, int[N] input, int[N] output, int[H][N] tree) {
      // Define N threads (parallel block)
      par threads (int t=0..N;;true)
         requires N==32 && H==6;
         requires t<(N/2) ==> Perm(input[t*2], read);
         requires t<(N/2) ==> Perm(input[t*2+1], read);
         requires t<(N/2) ==> Perm(output[t*2], write);
         requires t<(N/2) ==> Perm(output[t*2+1], write);
         requires t<(N/2) ==> Perm(tree[0][t*2], write);
         requires t<(N/2) ==> Perm(tree[0][t*2+1], write);
         requires (\forall* int j; 1<=j && j<H; Perm(tree[j][t], write));
         ensures N==32 && H==6;
         ensures t<(N/2) ==> Perm(input[t*2], read);
         ensures t<(N/2) ==> Perm(input[t*2+1], read);
         ensures t<(N/2) ==> Perm(output[t*2], write);
         ensures t<(N/2) ==> Perm(output[t*2+1], write);
         ensures t<(N/2) ==> Perm(tree[0][t*2], write);
         ensures t<(N/2) ==> Perm(tree[0][t*2+1], write);
         ensures (\forall* int j; 1<=j && j<H; Perm(tree[j][t], write));
      // Thread code
      {
         // Only use the first half of threads
         if(t < (N/2)) {
            // Input copy
            tree[0][t*2]  = input[t*2];
            tree[0][t*2+1] = input[t*2+1];
            // First step of upsweep
            tree[1][t] = tree[0][t*2] + tree[0][t*2+1];
         }
         int level=1;
         loop_invariant N==32 && H==6;
         loop_invariant t>=0 && t<N;
         loop_invariant level>=1 && level<H;
         loop_invariant (\forall* int i; level<=i && i<H; Perm(tree[i][t], write));
         loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2], write));
         loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2+1],
             write));
         while((level+1)<H) {
            level = level+1;
            barrier(local) {
               requires N==32 && H==6;
               requires t>=0 && t<N;
               requires level>=1 && level<H;
               requires Perm(tree[level-1][t], write));

               ensures N==32 && H==6;
               ensures level>=1 && level<H;
```

```
                    ensures t>=0 && t<N;
                    ensures Perm(tree[level][t], write));
                    ensures t<(N/2) ==> Perm(tree[level-1][t*2], write));
                    ensures t<(N/2) ==> Perm(tree[level-1][t*2+1], write));
                }
                // Do next upsweep step
                if(t < (N/2)) {
                    tree[level][t] = tree[level-1][t*2] + tree[level-1][t*2+1];
                }
            }

            // Set the root to 0
            tree[H-1][t] = 0;
            // Downsweep phase
            int level=H-1;
            loop_invariant N==32 && H==6;
            loop_invariant t>=0 && t<N;
            loop_invariant level>=1 && level<H;
            loop_invariant (\forall* int i; level<i && i<H; Perm(tree[i][t], write));
            loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2], write));
            loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2+1],
                  write));
            while((level-1)>0) {
                level = level-1;
                barrier(local) {
                    requires N==32 && H==6;
                    requires t>=0 && t<N;
                    requires level>=1 && level<H;
                    requires t<(N/2) ==> Perm(tree[level][t*2], write);
                    requires t<(N/2) ==> Perm(tree[level][t*2+1], write);

                    ensures N==32 && H==6;
                    ensures level>=1 && level<H;
                    ensures t>=0 && t<N;
                    ensures Perm(tree[level][t], write);
                    ensures t<(N/2) ==> Perm(tree[level-1][t*2], write);
                    ensures t<(N/2) ==> Perm(tree[level-1][t*2+1], write);
                }
                // Do next downsweep step
                if(t < (N/2)) {
                    tree[level-1][t*2+1] = tree[level-1][t*2] + tree[level][t]; // Set right child
                    tree[level-1][t*2] = tree[level][t]; // Set left child
                }
            }

            // Copy the result from the tree to the output array
            if(t < (N/2)) {
                output[t*2] = tree[0][t*2];
                output[t*2+1] = tree[0][t*2+1];
            }
        }
    }
}
```

3.2 Verifying downsweep array permissions

Listing 7 shows the working specification of all permissions of the `input`, `output` and `temp` array. The specification made in the previous project was quite close, but required some adjustments to verify with VerCors. First the permissions that should be active are explained, after which the required specification updates are shown.

3.2.1 Input array permissions

The input array permissions are shown in Figure 7, the array indices are shown at the bottom, the numbers inside the squares indicate the thread number that has access. The figure shows the input array for an input size of 8. This means that there will be 8 thread, of which only the first 4 will get any permissions. Each thread has access to indexes $t * 2$ and $t * 2 + 1$, where t is the thread number. In the upsweep phase the first half of the threads will start adding inputs together in consecutive pairs, so that is why the input array permissions are distributed like this.

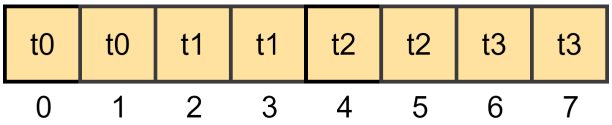

Figure 7: Permissions of the input array

3.2.2 Tree array permissions

The tree array permissions are more complex, and will change during the up and downsweep. At the start of the program the permissions are as shown in Figure 8. This figure shows the tree array with N horizontally and H vertically. The numbers in the squares show which thread has write access. A part of the squares have no background color, these are still included, but serve no actual purpose other than improving performance by reducing branch divergence as shown in the previous research project.

The permissions of the bottom row ($H = 0$) are like the input array, each thread has access to index $t * 2$ and $t * 2 + 1$. In the other rows the indices are simply assigned to their thread number. The first step of the downsweep will start with $tree[1][0] = tree[0][0] + tree[0][1]$, which thread 0 can do because it has write permissions to each of these locations. Thread 1, 2 and 3 have permissions to the next locations, doing the same computation as thread 0 while adjusting their indices using the thread number.

With the initial permissions of the input array and the bottom row of the tree array, the input can be copied to the bottom row of the tree array. The first half of the threads will do this task, each copying two consecutive locations, again $t * 2$ and $t * 2 + 1$.

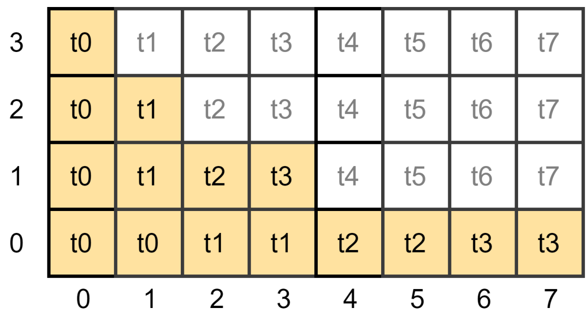

Figure 8: Permissions of the tree array before starting the upsweep

The `barrier()` of the upsweep will update the tree permissions row-by-row. Each iteration of the while loop the permission layout of the bottom row is applied to the next row above, until all but the top-most row has that permission layout.

The resulting permissions are shown in Figure 9.

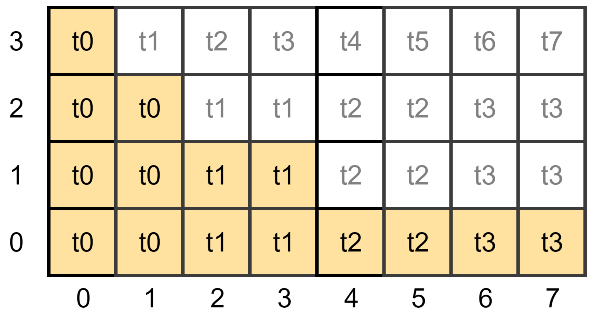

Figure 9: Permissions of the tree after upsweep

After the upsweep the program will set the root of the tree to zero, each thread has still access to the cell of its thread id, so each thread updates one cell.

Now the downsweep can start, for which the starting permissions are as shown in Figure 9. For all rows below the top-most row, the first half of threads will update two cells, at $t * 2$ and $t * 2 + 1$. The first computation can immediately start, because the permissions are already correct. For the following rows the permissions need to be updated row-by-row, reverting the permissions that have been added for the upsweep. After the downsweep the permissions will be back to the situation before the upsweep, as shown in Figure 8.

3.2.3 Output array permissions

At the end of the computation the result is at the bottom row of the tree array and will be copied to the output array. The bottom row of the tree array still has the layout where each thread has access to $t * 2$ and $t * 2 + 1$. So to easily copy the results the output array should have the same permission layout. The resulting permission layout is the same as the input array as shown in Figure 7. Now the first half of threads can copy the result and the Prefix Sum computation is complete.

3.2.4 Specification changes

Below the changes to the specification of the previous research project are explained.

Merged barrier clauses
The barrier of the upsweep phase changes the permissions, as described above in the permissions layout. The original specification used one require, and two ensures in the barrier to accomplish this change, as shown in Listing 5. VerCors could not match these statements correctly through the while loop, to solve this problem the ensures clauses have been merged into one ensure clause, as shown in Listing 6. Both of these options mean exactly the same, but is is hard for VerCors to recognize the original one. Ideally this would be fixed in the tool, but this was not trivial. This alternative notation has been provided by the tool author.

A similar merge was required for the downsweep barrier, in this case two require clauses have been merged into a single one using the same technique.

Listing 5: Upsweep barrier permissions clauses (before merging)

```
requires Perm(tree[level-1][t], write));
ensures t<(N/2) ==> Perm(tree[level-1][t*2], write));
ensures t<(N/2) ==> Perm(tree[level-1][t*2+1], write));
```

Listing 6: Upsweep barrier permissions clauses (after merging)

```
requires Perm(tree[level-1][t], write));
ensures t<(N/2) ==> (\forall* int k; 0<=k && k<2;
    Perm(tree[level-1][t*2+k], write));
```

First step of the downsweep

The first iteration of the downsweep has been extracted from the while loop. This was necessary because the permission layout of the tree array was already correct for the first iteration, so doing a barrier was not necessary yet. Extracting this first iteration leads to code duplication for the downsweep operation, which will be addressed in later specification versions. This change also meant updating the loop_invariant of the downsweep while loop, the `level<i` has been replaced with `level<=i` in the loop_invariant that describes the write permissions of the temp array of all rows except the bottom one.

Array definition workaround

VerCors did not properly recognize the length of the array through the specification, and could sometimes not know that each specified array cell is unique. A workaround for this problem has been provided by Stefan Blom. By adding `tree != null` the tools handles array lengths correctly and knows its cells are unique. ~~Hopefully this workaround can be removed in the future~~Note that this workaround has been removed already, the next section will show a simplified specification where this issue is solved.

Assert statements

The VerCors tools could not get through the while loops correctly anymore after some updates, the underlying problem is in the external library Z3. This problem has been solved by adding `assert <condition>` statements just before and inside the barrier, and by using identity operators in some places. These statements force the tools to prove a certain condition at that point, which provides provides search options into the right direction (where previously the tools would run out of options). At the end the specification works, but got bigger with assert statements and has added complexity with identity operators `id(0)` and `id(1)`.

Simplification failures

A couple of "simplification failure" errors have been encountered during the verification process. This happens when a `\forall*` cannot be transformed to a rule that the internal prover accepts. VerCors provides a language that is easy for the user, but internally the used prover tools only accept a more restrict set of specifications. This means that the specification gets transformed by VerCors into a format that the used prover tools understands. If this transformation cannot find a correct translation, a "simplifications failure" error message is shown. Transformations like these happen with simple matching rules, that replace certain parts with others. For example a simplification failed at some point when instead of writing `0<=i && i<N`, I wrote `i>=0 && i<N`. These expressions mean the same exact thing, but the rules only match the first variant. These kind of problems would really need to be solved before VerCors can be used by anyone to verify their programs.

Cleanup
Inside the barrier of the upsweep there was an unnecessary ensures clause, which has been removed: `ensures Perm(tree[level][t], write));`. This clause was not necessary because the permissions that a barrier does not mention, are assumed to be kept. So only the permission changes need to be in the clauses. The `loop_invariant` of the loop will ensure the other permissions are still available in the next iteration. Additionally the ensures clause of the complete program has been simplified, it does not have a separate clause for the bottom row of the tree array anymore. The split clauses were not necessary, since the program simply has write permissions to all locations in the tree array, so it is pointless to split among multiple clauses.

Verification time
A side effect of all changes is that the time it takes to verify the specification went from around 2 minutes, to 6 minutes. Considering that the Prefix Sum is quite a trivial number of lines of code, this is quite long. This would prevent verifying large programs, because it simply takes too long to run, especially considering that the specification is rarely correct the first run. The author of the tool indicates that each `if` statement causes the verification time to go up, since there are more options to consider. Loops have an even larger impact, which is why this program saw the increase in verification time. The workaround with `assert` and `id()` also do not help performance.

3.2.5 Result

After the changes described above the specification is complete and verifies in VerCors. It proves that the program does not contain data races, which is a good step towards proving the complete program.

Listing 7: Full permissions specification, verified using VerCors

```
class Ref {
   // Requires/Ensures for complete kernel
   requires N==32 && H==6;
   requires tree != null;
   requires (\forall* int i; 0<=i && i<N; Perm(input[i], read));
   requires (\forall* int i; 0<=i && i<N; Perm(output[i], write));
   requires (\forall* int i; 0<=i && i<N;
            (\forall* int j; 0<=j && j<H; Perm(tree[j][i], write))
         );

   ensures N==32 && H==6;
   ensures tree != null;
   ensures (\forall* int i; 0<=i && i<N; Perm(input[i], read));
   ensures (\forall* int i; 0<=i && i<N; Perm(output[i], write));
   ensures (\forall* int i; 0<=i && i<N;
            (\forall* int j; 0<=j && j<H; Perm(tree[j][i], write))
         );

   // Kernel method
```

```
void test(int N, int H, int[N] input, int[N] output, int[H][N] tree) {

    // Define threads
    par tst (int t=0..N;;true)
        requires N==32 && H==6;
        requires tree != null;
        requires t<(N/2) ==> Perm(input[t*2], read);
        requires t<(N/2) ==> Perm(input[t*2+1], read);
        requires t<(N/2) ==> Perm(output[t*2], write);
        requires t<(N/2) ==> Perm(output[t*2+1], write);
        requires t<(N/2) ==> Perm(tree[0][t*2], write);
        requires t<(N/2) ==> Perm(tree[0][t*2+1], write);
        requires (\forall* int i; 1<=i && i<H; Perm(tree[i][t], write));

        ensures N==32 && H==6;
        ensures t<(N/2) ==> Perm(input[t*2], read);
        ensures t<(N/2) ==> Perm(input[t*2+1], read);
        ensures t<(N/2) ==> Perm(output[t*2], write);
        ensures t<(N/2) ==> Perm(output[t*2+1], write);
        ensures t<(N/2) ==> Perm(tree[0][t*2], write);
        ensures t<(N/2) ==> Perm(tree[0][t*2+1], write);
        ensures (\forall* int i; 1<=i && i<H; Perm(tree[i][t], write));

    // Thread code
    {
        if(t < (N/2)) {
            // Input copy
            tree[0][t*2]  = input[t*2];
            tree[0][t*2+1] = input[t*2+1];
            // First step of up-sweep
            tree[1][t] = tree[0][t*2] + tree[0][t*2+1];
        }

        // Remaining steps of the upsweep phase
        int level=1;
        loop_invariant N==32 && H==6;
        loop_invariant tree != null;
        loop_invariant level>=1 && level<H;
        loop_invariant t>=0 && t<N;
        loop_invariant (\forall* int i; level<=i && i<H; Perm(tree[i][t], write));
        loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2], write));
        loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2+1],
            write));
        while((level+1)<H) {
            level = level+1;
            assert Perm(tree[level-1][t], write);
            barrier(local) {
                requires tree != null;
                requires N==32 && H==6;
                requires level>=1 && level<H;
                requires t>=0 && t<N;
                requires Perm(tree[level-1][t], write);

                ensures tree != null;
                ensures N==32 && H==6;
                ensures level>=1 && level<H;
                ensures t>=0 && t<N;
                ensures t<(N/2) ==> (\forall* int k; 0<=k && k<2; Perm(tree[level-1][t*2+k],
                    write));
            }
            // Do next up-sweep step
            if(t < (N/2)) {
                assert tree != null;
                assert N==32 && H==6;
                assert level>=1 && level<H;
                assert t>=0 && t<N;
                assert (\forall* int k; 0<=k && k<2; Perm(tree[level-1][t*2+k], write));
                tree[level][t] = tree[level-1][t*2+id(0)] + tree[level-1][t*2+id(1)];
            }
        }

        // Set the root to zero, preparation for down-sweep
        tree[H-1][t] = 0;
```

```
            if(t < (N/2) && level-1>0) {
                // First step of downsweep
                tree[level-1][t*2+1] = tree[level-1][t*2] + tree[level][t]; // Right child
                tree[level-1][t*2] = tree[level][t];                        // Left child
            }

            // Remaining steps of the downsweep
            loop_invariant N==32 && H==6;
            loop_invariant level>=1 && level<H;
            loop_invariant t>=0 && t<N;
            loop_invariant tree != null;
            loop_invariant (\forall* int i; level<=i && i<H; Perm(tree[i][t], write));
            loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2], write));
            loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2+1], write));
            while((level-1)>0) {
                level = level-1;
                assert t<(N/2) ==> Perm(tree[level][t*2+id(0)], write);
                assert t<(N/2) ==> Perm(tree[level][t*2+id(1)], write);
                barrier(local) {
                    requires N==32 && H==6;
                    requires t>=0 && t<N;
                    requires level>=1 && level<H;
                    requires t<(N/2) ==> (\forall* int k; 0<=k && k<2; Perm(tree[level][t*2+k], write));
                    requires tree != null;

                    ensures N==32 && H==6;
                    ensures level>=1 && level<H;
                    ensures t>=0 && t<N;
                    ensures Perm(tree[level][t], write);
                    ensures tree != null;
                }
                // Do next down-sweep step
                if(t < (N/2)) {
                    assert tree != null;
                    assert N==32 && H==6;
                    assert level>=1 && level<H;
                    assert t>=0 && t<N;
                    assert Perm(tree[level][t], write);
                    tree[level-1][t*2+1] = tree[level-1][t*2] + tree[level][t]; // Right child
                    tree[level-1][t*2] = tree[level][t];                        // Left child
                }
            }
            // Copy the result from the tree to the output array
            if(t < (N/2)) {
                output[t*2] = tree[0][t*2];
                output[t*2+1] = tree[0][t*2+1];
            }
        }
    }
}
```

3.3 Specification simplification

After finishing the specification for the array permissions, it felt like it was too verbose. The first problem is that a lot of clauses start with `t<(N/2)`, and most code blocks ~~are~~ also have the same condition. This condition disables the last half of the threads that are used in the kernel. Actually these threads are not useful at all and normally would not be used for the execution. The only reason the number of threads is ~~double~~ twice than necessary is that VerCors could not handle the correct number of threads at first. The syntax of specifying a GPU kernel program was different, and did not allow to easily use half the number of threads, and arrays did not accept expressions inside their declaration. Later the syntax updated based on the Java language syntax, so it basically is just a class with a method inside. So now it is possible to make a specification with half the number of threads, so that the result is a lot more compact and readable. The resulting specification will be discussed below.

3.3.1 Half the number of threads

To half the number of threads the `N==32` clause has been changed to `N==16`. The `par` block that specifies the number of threads uses the N directly, so the number of threads is now correctly halved. The input, output and tree array specifications are changed to use $N * 2$ as size, because they should stay the same size as before. All of the $t < (N/2) ==>$ constructs have been removed, those are not necessary anymore.

3.3.2 Updated permissions

First of all the permissions of the input and output array have changed, now instead of letting only the first half of the threads copy locations $t*2$ and $t*2+1$, all threads (which are half) copy locations t and $t+N$. The permissions have been changed accordingly.

There were a couple of loop_invariants that did not have a condition that excludes half of the threads, assigning certain permissions to each thread. Because the number of threads halved, this means that not all permissions are assigned anymore. The input and output array keep the same permissions structure, since these already used only the first half of the threads. The tree array has slightly different permissions though. On the rows where previously all threads had one cell, each thread now has t and $t+N$. Figure 10 shows the permissions layout before the upsweep, which can be compared to Figure 8. In the figure you can also see that the initial permissions of the bottom row are not in the $t*2$ and $t*2+1$ layout anymore, the reason for that is explained below.

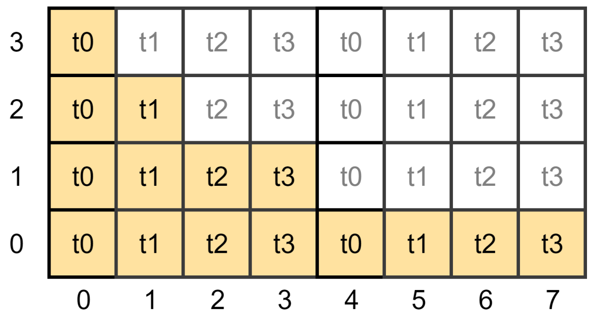

Figure 10: Permissions of the tree before the upsweep, updated for usage with half the number of threads

3.3.3 Merged first iteration

Another improvement in this specification is that the first iterations of the upsweep and downsweep are not outside of the while loop anymore.

For the upsweep this meant the permissions of the bottom row are now in the t and $t+N$ layout at the start. This means that before the first upsweep step can be done, the permissions of the bottom row need to change. Because inside the while loop the barrier comes before the computation, this works correctly. First the barrier assigns new permissions to one row, after that the computation uses that row and the row above to do the upsweep step. After the upsweep the permissions look like shown in Figure 11, which only differs from Figure 9 on the top row, where $t4$ until $t7$ have been replaced (since those threads are not used anymore). This cleans up the specification, removing a duplicate computation and some specification complexity.

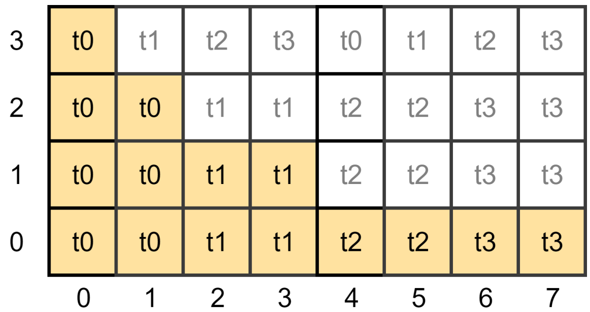

Figure 11: Permissions of the tree after the upsweep, updated for usage with half the number of threads

For the downsweep different changes have been made. The permissions after the upsweep are already correct to do the first downsweep step, so the barrier and computation in the while loop have been swapped. Now the computation happens first, and the barrier as second, which gets the permissions ready for the next iteration. Because of this change the final permissions layout has changed, the bottom row is now also converted to the t and $t + N$ layout. Figure 12 shows the resulting permissions after the downsweep, which can be compared to Figure 8. This new layout has the same permissions on each row, while the previous one had special permissions on the last row. This is because of the updated while loop, that now changes the permissions of all rows.

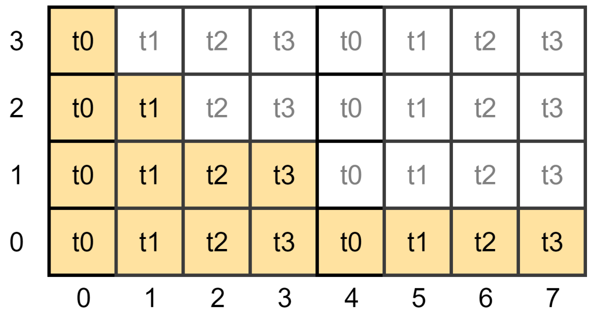

Figure 12: Permissions of the tree after the downsweep, updated for usage with half the number of threads

Finally because the permissions of the last row have changed, copying the result to the output array has been updated. Each thread now copies locations t and $t + N$, because that is where they have permissions.

To be able to use indexing in the form of t and $t + N$ VerCors has been updated by Stefan Blom, this allowed this specification to work.

3.3.4 Compacting: invariant and context keywords

In the previous specification the clause `N==32 && H==6` was repeated constantly, in each loop and barrier. To prevent this kind of repetition the `invariant` keyword is now available on the program level, so that specifying constants is only needed once. This invariant on the kernel level requires and ensures the given condition through the complete program, including loops. This cleans up a lot of clauses, making the specification easier to read and understand.

Another pattern that is common in the previous specification is that a clause in the requires and ensures is the same. For example the barrier would repeat a condition of the loop_invariant as a require and ensure, duplicating this specification. This has been solved by Stefan Blom by adding the `context` keyword in VerCors, which acts as a require and ensure at the same time. This compacted the specification of the barriers a bit, for the constraints of the thread number and the level.

The above changes do not change the actual verification at all, since these options are simple syntactic sugar. They do however improve the user experience of VerCors a lot, keeping the specification a lot more readable.

3.3.5 Cleanup: \matrix and \array

The previous specification contains a bunch of `tree != null` clauses, which were used as a way to ensure certain properties about the used arrays. This was a temporary workaround, which now has a permanent solution with `\array` and `\matrix` as replacements. In the specification `\matrix(tree,H,N*2)` is used to indicate that `tree` is a two-dimensional array (also called matrix) with dimensions `H` by `N*2`. This information is used through the specification by VerCors to know the dimension of the array and other standard array properties. For the input and output array `\array(input,N*2)` is used, specifying an array with one dimension. These changes mean that all `tree != null` clauses have been cleaned up, and make it more obvious what the inputs are. The workaround with `id(0)` and `id(1)` has also been removed, this is not necessary anymore because of improvements in VerCors.

3.3.6 Result

With the changes described above the specification has become smaller, easier to read, and with less workarounds. Sadly the verification time has increased to 36 minutes for the complete specification on my machine (Intel i7-4770K), making it painful to do modifications. Turning off the ensure clauses of the kernel and thread block will get the verification time back to 1 minute and 30 seconds, which is can be worked with.

3.4 Result

After first verifying a straightforward version of the permissions using a verbose specification, a compact version has been made. This last version is a nice and compact verification of the permissions (and therefore absence of data races) of the Prefix Sum, which can serve as a starting point for verifying the functionality. The next step is to find out the changes necessary for the functional verification, which is discussed in the next chapter.

Listing 8: Compacted specification using only half the number of threads

```
class Ref {
    // KERNEL
    invariant N==16 && H==6; // H = log(N)/log(2) + 2, N should be a power of 2
    invariant \matrix(tree,H,N*2);
    invariant \array(input,N*2);
    invariant \array(output,N*2);
    context (\forall* int i; 0<=i && i<N*2; Perm(input[i], read));
    context (\forall* int i; 0<=i && i<N*2; Perm(output[i], write));
    context (\forall* int i; 0<=i && i<N*2;
        (\forall* int j; 0<=j && j<H; Perm(tree[j][i], write))
    );
    void runKernel(int N, int H, int[N*2] input, int[N*2] output, int[H][N*2] tree) {

        ////////// THREADS
        par runThreads (int t=0..N)
            context Perm(input[t], read);
            context Perm(input[t+N], read);

            context Perm(output[t], write);
            context Perm(output[t+N], write);

            context (\forall* int k; 0<=k && k<2;
                (\forall* int i; 0<=i && i<H; Perm(tree[i][k*N+t], write))
            );
        {
            ////////// INPUT COPY
            tree[0][t] = input[t];
            tree[0][t+N] = input[t+N];

            ////////// UPSWEEP
            int level=0;
            loop_invariant level>=0 && level<H;
            loop_invariant t>=0 && t<N;
            // Old permisison layout going away, top row excluded: (t1 | t2 | t3 | t4 | ...)
            loop_invariant (\forall* int k; 0<=k && k<2;
                (\forall* int i; level<=i && i<H; Perm(tree[i][k*N+t], write))
            );
            // New permission layout appearing, lowest row already has it: (t1 | t1 | t2 | t2 | ...)
            loop_invariant (\forall* int k; 0<=k && k<2;
                (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2+k], write))
            );
            while((level+1)<H) {
                level = level+1;
                barrier(runThreads) {
                    context t>=0 && t<N;
                    context level>=1 && level<H;

                    requires Perm(tree[level-1][t], write);
                    requires Perm(tree[level-1][t+N], write);

                    ensures (\forall* int k; 0<=k && k<2; Perm(tree[level-1][t*2+k], write));
                }
                tree[level][t] = tree[level-1][t*2] + tree[level-1][t*2+1];
            }

            ////////// DOWNSWEEP
            tree[H-1][t] = 0; // Set the root of the tree to zero

            loop_invariant level>=0 && level<H;
            loop_invariant t>=0 && t<N;
            // Old permission layout coming back, top row already has it: (t1 | t2 | t3 | t4 | ...)
            loop_invariant (\forall* int k; 0<=k && k<2;
                (\forall* int i; level<=i && i<H; Perm(tree[i][k*N+t], write))
            );
            // New permission layout going away, bottom row excluded: (t1 | t1 | t2 | t2 | ...)
            loop_invariant (\forall* int k; 0<=k && k<2;
                (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2+k], write))
            );
            while(level>0) {
                // Do next down-sweep step
                tree[level-1][t*2+1] = tree[level-1][t*2] + tree[level][t]; // Right child
                tree[level-1][t*2] = tree[level][t];                        // Left child
```

```
            level = level-1;

            barrier(runThreads) {
                context t>=0 && t<N;
                context level>=0 && level<H;

                requires Perm(tree[level][t*2], write);
                requires Perm(tree[level][t*2+1], write);

                ensures Perm(tree[level][t], write);
                ensures Perm(tree[level][t+N], write);
            }
        }

        ////////// OUTPUT COPY
        output[t] = tree[0][t];
        output[t+N] = tree[0][t+N];
    }
  }
}
```

4 Verifying functionality

The result of the previous chapter is a specification that verifies the permissions of the Prefix Sum program, proving that the program is data race free. This does however not prove that the computed end result is actually the Prefix Sum of the inputs. To prove this aspect the specification has to be extended with clauses about the result. This chapter will describe the steps taken to work towards a specification that verifies the permissions and functionality of the Prefix Sum algorithm. In the time slot for this project is was not possible to finish the verification of the functionality, but a couple of steps in the right direction have been taken.

4.1 Approach

For this extension of the specification the plan is once again to do it step-by-step. This needs to stay the way to tackle specifications for now, because writing all specifications at once is rarely getting verified by VerCors. However, this is more the human element than the fault of the tool, often it is hard to come up with all specifications at once, most of the time some simple required clauses are missing to let the tool verify the specification.

The starting point will be the specification of the previous chapter, that specifies all permissions. To make the first additions quicker to test, the downsweep and output copy parts of the specification will be disabled at first. The ensures clauses of the kernel and threads will also be disabled, since these are not correct if the downsweep is disabled. When the upsweep is figured out and verifies, the downsweep can be enabled again. After that the functional specifications can be added to the downsweep, working towards a complete specification.

4.2 Goal

The goal that we are working towards is an ensure clause that specifies that the output array contains a prefix sum of the values in the input array. Such a clause would look like shown in Listing 9. This clause specifies that each cell of the output should contain the sum of all values in the input array with an index between zero (inclusive) and its own index (exclusive). A new construct `\sum` is used to express this condition, which sums all values of a range of indices. A range of indices can be specified using a special syntax `[0 ..i)`. This expression means sum all values starting at 0 (inclusive because of the square bracket), ending at `i` (exclusive because of the round bracket) of the `input` array.

Listing 9: Ensure clause of the program that should be proven eventually

```
ensures (
\forall* int i; 0<=i && i<N*2;
output[i] == \sum([0 ..i), input)
);
```

4.3 Verification steps

Before the condition described in the section above can be proven, a lot of steps needs to be taken. Below these steps are listed:

1. The bottom row of the tree array is the same as the input array.
2. Each upsweep step a row of the tree array contains partial sums of input values.
3. The highest row of the tree array is zero.
4. Each downsweep step a row of the tree array contains updated partial sums of input values, after the last iteration this will already specify that the bottom row contains the prefix sum.
5. The output array contains the prefix sum (condition of Listing 9).

Each of the above steps has some challenges, in terms of writing down the conditions itself and actually verifying it with VerCors. The next sections will discuss each step.

4.3.1 Step 1: tree array bottom row

The first step is to specify that the bottom row of the tree array has the same values as the input array. At the while loop of the upsweep this should be the case, so that is where a loop_invariant will be added describing this condition. Listing 10 shows this clause that is necessary for this.

Listing 10: Bottom row of the tree array is equal to the input array

```
loop_invariant (\forall* int i; 0<=i && i<N*2; tree[0][i] ==
    input[i]);
```

4.3.2 Step 2: upsweep results

During the upsweep the tree array will be updated to contain partial sums of the input values. During iteration of the while loop a loop_invariant should enforce the values of the tree array to be correct. Listing 11 shows this condition. The condition expresses that two cells on a row below have been added to the current row. This condition does not yet hold up at the start of the while loop, which is why the precondition `level>0` has been added.

Listing 11: Upsweep loop_invariant enforcing the results of the computation

```
loop_invariant level>0 ==> tree[level][t] == tree[level-1][t*2] +
    tree[level-1][t*2+1];
```

Currently I'm not sure this is enough to let VerCors proof the end condition, but it should be a good start. It might need to already know that each cell is the sum of a certain range of values from the bottom row of the tree array, which might look like shown in Listing ~~??~~12. One problem is that threads do not have access to all cells at the lower row of the tree array, so this condition would lead to permissions errors. Instead the input array might be used, but I'm unsure if this can be proven using VerCors.

The precondition is necessary because not all threads will have complete results at all levels, as shown earlier the higher rows in the tree array have some dummy values, to increase performance.

Listing 12: Upsweep loop_invariant enforcing sum of inputs

```
((t+1)*2^level < N*2) => tree[level][t] == \sum([t*2^level
    ..(t+1)*2^level), tree[0]);
```

4.3.3 Step 3: highest row to zero

After the upsweep the top row of the tree array will be set to zero, in preparation of the downsweep. At the while loop of the downsweep an invariant about this row will be placed, as shown in Listing 13. With this information the downsweep condition can be written.

Listing 13: Top row of the tree array to zero

```
loop_invariant tree[H-1][t] == 0;
```

4.3.4 Step 4: downsweep results

The next clause should be about the results of the computation of the downsweep loop. Listing 14 shows conditions that could be added to verify this result. The first condition specifies that a value from the current level has been copied to a level below, at twice the thread index. This computation takes the value from a node in the binary tree, and assigns it to its left child. The right child has a different computation, it is set to the sum of the left child and the parent. Because the left child is also updated by the previous computation, the `\old` function is put around the value of the left child in the specification.

Listing 14: Downsweep results loop_invariant

```
loop_invariant tree[level-1][t*2+1] = \old(tree[level-1][t*2]) +
    tree[level][t];
loop_invariant tree[level-1][t*2] = tree[level][t];
```

Specifying the partial sum that should be in the result cells is also possible, the condition for that is shown in Listing 15. Once again threads do not have access to all required cells of the lower row of the tree array, so instead the input array could be used.

Listing 15: Downsweep summed results

```
loop_invariant (t*2^(level-1) < N*2) => tree[level-1][t*2] ==
    \sum([0 ..t*2^(level-1)], tree[0]);
loop_invariant (t*2^(level-1) < N*2) => tree[level-1][t*2+1] ==
    \sum([0 ..t*2^(level-1)+1], tree[0]);
```

4.3.5 Step 5: output results

Finally the results that have appeared on the bottom row of the tree array will be copied to the output array. This result will be verified in an ensures clause of the thread block and the kernel ensure. The condition of the thread block will be as shown in Listing 16.

The conditions the complete kernel will have is shown in Listing 9. With this last information the full program should be verified, in this all works in VerCors.

Listing 16: Ensures clause of the kernel, verifying the output array results

```
output[t] == \sum([0 ..t), input);
output[t+N] == \sum([0 ..t+N), input);
```

4.4 Results

Of the steps that are described above, only the first one is verified by VerCors. Because the verification process takes a long time, verifying and compacting the specification of the permissions took a long time. The descriptions above should however be a good starting point to verify the functionality of the Prefix Sum algorithm with VerCors. It will require more time and effort, and most likely additions to VerCors, but should be possible.

Listing 17: Work in progress functional specification

```
class Ref {

    ////////// KERNEL
    invariant N==16 && H==6;
    invariant \matrix(tree,H,N*2);
    invariant \array(input,N*2);
    invariant \array(output,N*2);
    context (\forall* int i; 0<=i && i<N*2; Perm(input[i], read));
    context (\forall* int i; 0<=i && i<N*2; Perm(output[i], write));
    context (\forall* int i; 0<=i && i<N*2;
        (\forall* int j; 0<=j && j<H; Perm(tree[j][i], write))
    );
    void runKernel(int N, int H, int[N*2] input, int[N*2] output, int[H][N*2] tree) {

        ////////// THREADS
        par runThreads (int t=0..N)
            context Perm(input[t], read);
            context Perm(input[t+N], read);

            context Perm(output[t], write);
            context Perm(output[t+N], write);

            context (\forall* int k; 0<=k && k<2;
```

```
        (\forall* int i; 0<=i && i<H; Perm(tree[i][k*N+t], write))
    );

    ensures output[t] == \sum([0 ..t), input);
    ensures output[t+N] == \sum([0 ..t+N), input);
{
    ////////// INPUT COPY
    tree[0][t] = input[t];
    tree[0][t+N] = input[t+N];

    ////////// UPSWEEP
    int level=0;
    loop_invariant level>=0 && level<H;
    loop_invariant t>=0 && t<N;
    // Permissions
    loop_invariant (\forall* int k; 0<=k && k<2;
        (\forall* int i; level<=i && i<H; Perm(tree[i][k*N+t], write))
    );
    loop_invariant (\forall* int k; 0<=k && k<2;
        (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2+k], 1/2))
    );
    loop_invariant (\forall* int k; 0<=k && k<4;
        (\forall* int i; 0<=i && i<level; t<N/4 ==> Perm(tree[i][t*4+k], 1/2))
    );
    // Functional
    loop_invariant (\forall* int i; 0<=i && i<N*2; tree[0][i] == input[i]);
    loop_invariant ((t+1)*2^level < N*2) => tree[level][t] == \sum([t*2^level
         ..(t+1)*2^level), input);
    while((level+1)<H) {
        level = level+1;
        barrier(runThreads) {
            context t>=0 && t<N;
            context level>=1 && level<H;
            // Permissions
            requires Perm(tree[level-1][t], write);
            requires Perm(tree[level-1][t+N], write);
            ensures (\forall* int k; 0<=k && k<2; Perm(tree[level-1][t*2+k], write));
            // Functional
            ensures ((t+1)*2^level < N*2) => tree[level][t] == \sum([t*2^level
                 ..(t+1)*2^level), input);
        }
        tree[level][t] = tree[level-1][t*2] + tree[level-1][t*2+1];
    }

    ////////// DOWNSWEEP
    tree[H-1][t] = 0; // Set the root of the tree to zero

    loop_invariant level>=0 && level<H;
    loop_invariant t>=0 && t<N;
    // Permissions
    loop_invariant (\forall* int k; 0<=k && k<2;
        (\forall* int i; level<=i && i<H; Perm(tree[i][k*N+t], write))
    );
    loop_invariant (\forall* int k; 0<=k && k<2;
        (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2+k], write))
    );
    // Results
    loop_invariant tree[H-1][t] == 0;
    loop_invariant (t*2^(level-1) < N*2) => tree[level-1][t*2] == \sum([0 ..t*2^(level-1)],
         input);
    loop_invariant (t*2^(level-1) < N*2) => tree[level-1][t*2+1] == \sum([0
         ..t*2^(level-1)+1], input);
    while(level>0) {
        // Do next down-sweep step
        tree[level-1][t*2+1] = tree[level-1][t*2] + tree[level][t]; // Right child
        tree[level-1][t*2] = tree[level][t];                        // Left child

        level = level-1;

        barrier(runThreads) {
            context t>=0 && t<N;
            context level>=0 && level<H;
            // Permissions
            requires Perm(tree[level][t*2], write);
```

```
                requires Perm(tree[level][t*2+1], write);
                ensures Perm(tree[level][t], write);
                ensures Perm(tree[level][t+N], write);
                // Functional
                ensures (t*2^(level-1) < N*2) => tree[level-1][t*2] == \sum([0 ..t*2^(level-1)],
                     input);
                ensures (t*2^(level-1) < N*2) => tree[level-1][t*2+1] == \sum([0
                     ..t*2^(level-1)+1], input);
            }
        }

        ////////// OUTPUT COPY
        output[t] = tree[0][t];
        output[t+N] = tree[0][t+N];
    }
  }
}
```

5 Discussion

Below the results will be discussed and related work will be shown.

5.1 Results

The permissions of the Prefix Sum program have been verified, showing that the program is data race free. The specification is as compact as is reasonable and does not have special workarounds. This shows that parallel programs like the Prefix Sum can be proven data race free with VerCors and its specifications language.

The functional specification has not been proven by VerCors, but an example version has been made to show a direction that could be taken to start the verification.

During the specification and verification of the Prefix Sum program the tool VerCors has been improved using the feedback and issues provided during this project. Stefan Blom always answered quickly to solve problems in VerCors or provide information about usage of it, helping this project a lot.

5.2 Related work

Most of the related work are simply the example provided by VerCors in the `/examples` directory on Github[1]. Most techniques used in the specifications of this project are shown in these examples (of which some are added in response to this project).

The basics of VerCors are described in [3], which introduces the tool. The architecture and a couple more details are shown in [1]. Other verification projects have been done with VerCors, but most using Java instead of the PVL language.

5.3 Limitations and problems

In this section the limitations and problems encountered during this project will be discussed. A first problem is that the code of the algorithm that computes the Prefix Sum has been changed during the specification process. Especially at the start code sometimes had to change to be able to properly create a specification. In my opinion verification should add an extra layer to code, but not force code to change, except when you uncover actual problems in the code. At the end the program has been restored to a version that is close to the original algorithm, so during this project VerCors has improved and allowed the code to stay clean.

A second problem is the performance of VerCors. The tool will run longer when the specification is longer, and this looks exponential with the number of if/while conditions. This will become a problem when trying to prove larger programs. To show the time it takes for VerCors to prove a specification, see Table 1. As shown in the table, it starts with verification times of a couple seconds, increases

[1] `https://github.com/utwente-fmt/vercors`

to minutes when doing the downsweep, and finally for the full program increases to 36 minutes. As long as the verification is in the seconds or at least less than 10 minutes it is doable to verify a program. But if it gets any longer it is simply too slow to be able to iterate quickly enough. This is a severe problem for adoption of VerCors for large programs.

Table 1: Verification times permissions specification, Listing 8

Until specification part	Time
Input copy	7882ms
Upsweep	12745ms
Downsweep	92351ms
Downsweep with thread ensures	1934471ms
Downsweep with thread and kernel ensures	2190647ms

A lot of verification time is added when adding the ensure clauses of the kernel and thread blocks. This time seems unreasonably long, since it is basically proving that the final condition of the downsweep loop matches the ensures clause. This step does add almost 30 minutes to the verification time. And after that it needs to combine the permissions of the individual threads together (which adds 4 minutes to the verification). Hopefully with some debugging of VerCors it is possible to find out why these last steps take that long, to eventually solve this issue.

There are some possibilities for improving the performance of VerCors. A first issue is that only one core is used for the verification, while modern systems usually at least have 4. When verifying programs with multiple methods VerCors can use multiple cores, but in this case it is just one method. Different classes can also be verified in parallel, so verifying a program consisting of multiple classes can be speed up by having a system with a good multi-core processor. To improve the verification times of a single method, the underlying proving libraries like Z3 could be improved. Or the translated specification that VerCors produces and passes to the underlying prover could be improved. It is however not likely that VerCors will ever be able to work linearly depending on the number of if/while conditions, since each such a condition simply introduces a lot more options. The fact that this program is a GPU kernel, which basically has a sort of loop for all threads makes the verification complexity higher, causing longer run times. Regular Java programs will have quicker verification because the additional thread loop is absent.

In conclusion, as long as programs use small methods the verification should be quick enough. Once a program has large methods, like the algorithm shown here it will become a lot harder to prove quick enough to be useful. Further investigation of the performance could be done to see where the bottlenecks are, which is something Stefan Blom already introduced a `--profile` command line flag for.

6 Conclusion

Below the research questions are answered and the value of this project is described. After that future work is discussed, which could build on this project.

6.1 Research questions answers

The first research question asks how the Prefix Sum program can be proven to have no data races. This question has been answered with the specification and verification of the program using VerCors.

The second research question asks how the Prefix Sum program can be proven to give the correct result. This question has not been completely answered, although a specification has been made the verification process with VerCors has not been completed. This will require more time, but has a good chance of succeeding given that problems in the tool can be solved.

The third research question asks about the limitations of VerCors for verifying GPU programs. This question is answered above in the discussion section. Large programs are possible to verify, although methods will need to stay reasonably small. For verifying GPU programs I would not recommend anything that is much larger than this Prefix Sum program in terms of if/while statements, because that would make iterating on the specification really hard. For the final conditions of the Prefix Sum specifications it already took a long time, since often the tool would run for half an hour, only to find out there is a small mistake in the specification.

6.2 Research value

This project has contributed to improving VerCors by finding bugs and asking for extensions. For example the new clauses like `\context` and the top level `\invariant` are also in response to my feedback. Some bugs in the tool that led to accepting wrong specifications with a "verified" result have also been fixed. This is in my opinion critical, since the last thing you want is a verification tool that accepts wrong specifications.

Next to this the work done in this project can be used as example for GPU verification, and might be a stepping stone for verifying other programs of the same size, or even larger.

6.3 Future work

Since the functionality of the Prefix Sum could not be verified fully yet, this would be interesting to look into in the future. Next to this other algorithms could be verified to test the tool further. VerCors could also be improved more, accepting more variants of specifications to make it easier to pick up if you are new with using it. A simple addition to the tool would be automatically adding conditions for the range of the thread variable for GPU programs, since this does never change and is currently repeated all the time.

7 Acknowledgments

I would like to thank Marieke Huisman for providing the initial Bachelor thesis project that got me into program verification and specifically verifying a parallel Prefix Sum program for the GPU. Marieke also has given me the opportunity to do this research project based on the Bachelor project results. Next to that she provided feedback and suggestions about the project. Next I would like to thank Saeed Darabi which guided me through the Bachelor project together with Marieke. Finally I want to thank Stefan Blom a lot for his quick responses to my feedback about VerCors. I submitted bugs and feedback to Stefan and most of the time he had a new VerCors release for me within a day. I also appreciate his suggestions for the specification, he helped a lot to get it to form that VerCors can handle, and helped me with small details I would have otherwise not have thought about.

8 References

[1] A. Amighi, S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. The vercors project: Setting up basecamp. In *Proceedings of the sixth workshop on Programming languages meets program verification*, pages 71–82. ACM, 2012.

[2] G. E. Blelloch. Prefix sums and their applications. 1990.

[3] S. Blom and M. Huisman. The vercors tool for verification of concurrent programs. *FM*, 8442:127–131, 2014.

[4] T. D. Han and T. S. Abdelrahman. Reducing branch divergence in GPU programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1–3:8, New York, NY, USA, 2011. ACM.

[5] O. A. Huguet and C. G. Marin. Simple-OpenCL library. `https://code.google.com/p/simple-opencl/`. Accessed: 05 March 2015.

[6] H. Nguyen. *GPU Gems 3*. Addison-Wesley Professional, first edition, 2007.

[7] T. Wiefferink. Optimization, specification and verification of the prefix sum program in an opencl environment. 2015.