Verification of the Prefix Sum Program in an OpenCL Environment

Thijs Wiefferink
thijs@wiefferink.me, s1366564

University of Twente

Abstract

todo: write

Keywords

todo: write

Preface

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 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 [5].

Contents

1 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 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[5] 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 [1]. In the field of specifying and verifying GPU programs the Prefix Sum is a suitable next step, because it will be a bigger and more complex example of verifying a GPU program.

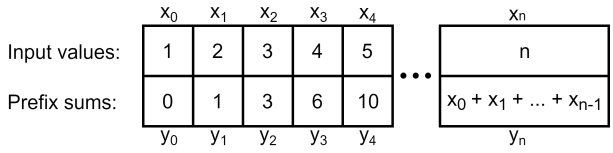

Figure 1: Prefix Sum description

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 [4].

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[5].

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 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). 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 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

todo: introduce chapters

2 Prefix sum algorithm

The sections below are repeated from the previous project[5], 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

```
1 result[0] := 0
2 for a := 1 to n do
3     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 [4] has been followed to implement the algorithm. The *Simple-OpenCL* library [3] 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.

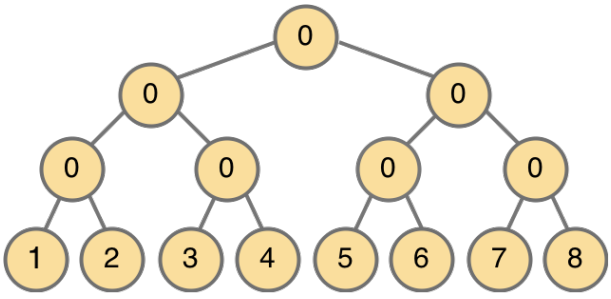

Figure 2: Tree at the start of the up-sweep phase

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

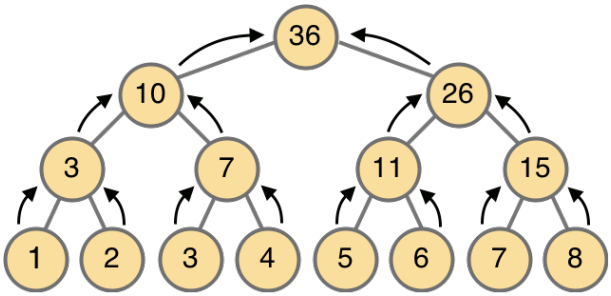

Figure 3: Tree after the up-sweep phase

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.

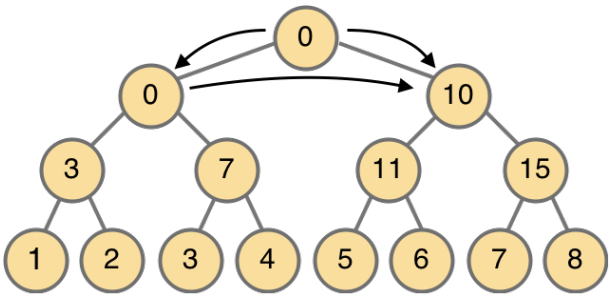

Figure 4: The first step of the down-sweep phase

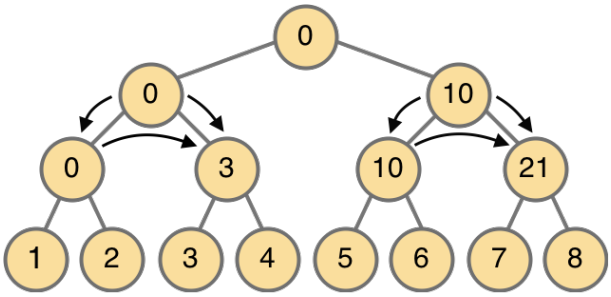

Figure 5: The second step of the down-sweep phase

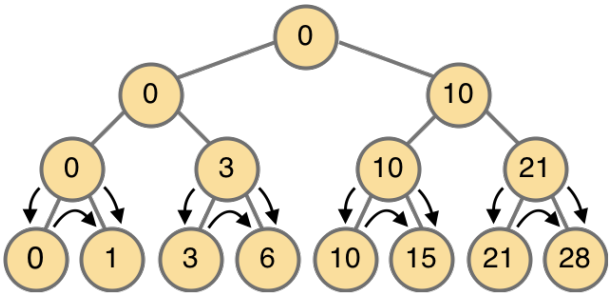

Figure 6: The final step of the down-sweep phase

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 [2]. 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.

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;
```

```
47            loop_invariant N==32 && H==6;
48            loop_invariant t>=0 && t<N;
49            loop_invariant level>=1 && level<H;
50            loop_invariant (\forall* int i; level<=i && i<H; Perm(tree[i][t], write));
51            loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2], write));
52            loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2+1],
                  write));
53            while((level+1)<H) {
54                level = level+1;
55                barrier(local) {
56                    requires N==32 && H==6;
57                    requires t>=0 && t<N;
58                    requires level>=1 && level<H;
59                    requires Perm(tree[level-1][t], write));
60
61                    ensures N==32 && H==6;
62                    ensures level>=1 && level<H;
63                    ensures t>=0 && t<N;
64                    ensures Perm(tree[level][t], write));
65                    ensures t<(N/2) ==> Perm(tree[level-1][t*2], write));
66                    ensures t<(N/2) ==> Perm(tree[level-1][t*2+1], write));
67                }
68                // Do next upsweep step
69                if(t < (N/2)) {
70                    tree[level][t] = tree[level-1][t*2] + tree[level-1][t*2+1];
71                }
72            }
73
74            // Set the root to 0
75            tree[H-1][t] = 0;
76            // Downsweep phase
77            int level=H-1;
78            loop_invariant N==32 && H==6;
79            loop_invariant t>=0 && t<N;
80            loop_invariant level>=1 && level<H;
81            loop_invariant (\forall* int i; level<i && i<H; Perm(tree[i][t], write));
82            loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2], write));
83            loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2+1],
                  write));
84            while((level-1)>0) {
85                level = level-1;
86                barrier(local) {
87                    requires N==32 && H==6;
88                    requires t>=0 && t<N;
89                    requires level>=1 && level<H;
90                    requires t<(N/2) ==> Perm(tree[level][t*2], write);
91                    requires t<(N/2) ==> Perm(tree[level][t*2+1], write);
92
93                    ensures N==32 && H==6;
94                    ensures level>=1 && level<H;
95                    ensures t>=0 && t<N;
96                    ensures Perm(tree[level][t], write);
97                    ensures t<(N/2) ==> Perm(tree[level-1][t*2], write);
98                    ensures t<(N/2) ==> Perm(tree[level-1][t*2+1], write);
99                }
100               // Do next downsweep step
101               if(t < (N/2)) {
102                   tree[level-1][t*2+1] = tree[level-1][t*2] + tree[level][t]; // Set right child
103                   tree[level-1][t*2] = tree[level][t]; // Set left child
104               }
105           }
106
107           // Copy the result from the tree to the output array
108           if(t < (N/2)) {
109               output[t*2] = tree[0][t*2];
110               output[t*2+1] = tree[0][t*2+1];
111           }
112       }
113   }
114 }
```

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.

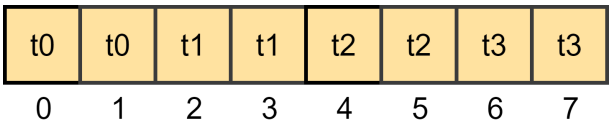

Figure 7: Permissions of the input array

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 $t * 2 + 1$.

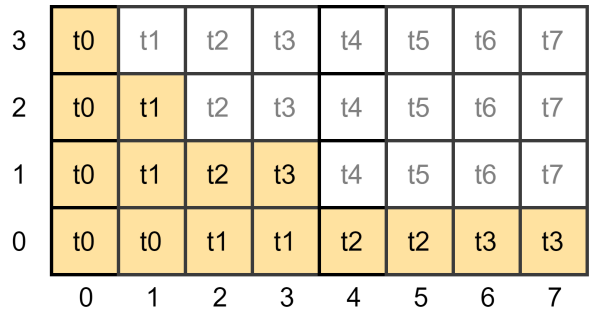

Figure 8: Permissions of the tree array before starting the upsweep

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.

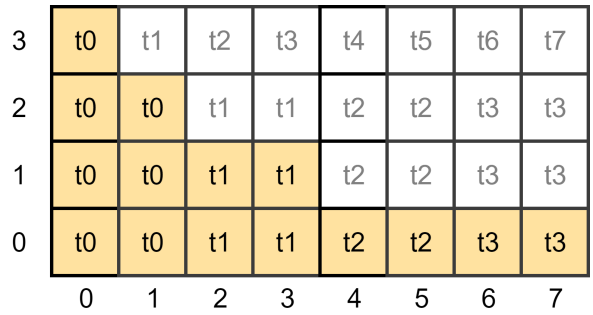

Figure 9: Permissions of the tree after upsweep

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.

Listing 5: Upsweep barrier permissions clauses (before merging)

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

Listing 6: Upsweep barrier permissions clauses (after merging)

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

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.

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.

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. 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));`. 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

```
1 class Ref {
2     // Requires/Ensures for complete kernel
3     requires N==32 && H==6;
4     requires tree != null;
5     requires (\forall* int i; 0<=i && i<N; Perm(input[i], read));
6     requires (\forall* int i; 0<=i && i<N; Perm(output[i], write));
7     requires (\forall* int i; 0<=i && i<N;
8                 (\forall* int j; 0<=j && j<H; Perm(tree[j][i], write))
9             );
```

```
10
11      ensures N==32 && H==6;
12      ensures tree != null;
13      ensures (\forall* int i; 0<=i && i<N; Perm(input[i], read));
14      ensures (\forall* int i; 0<=i && i<N; Perm(output[i], write));
15      ensures (\forall* int i; 0<=i && i<N;
16                 (\forall* int j; 0<=j && j<H; Perm(tree[j][i], write))
17              );
18
19      // Kernel method
20      void test(int N, int H, int[N] input, int[N] output, int[H][N] tree) {
21
22          // Define threads
23          par tst (int t=0..N;;true)
24              requires N==32 && H==6;
25              requires tree != null;
26              requires t<(N/2) ==> Perm(input[t*2], read);
27              requires t<(N/2) ==> Perm(input[t*2+1], read);
28              requires t<(N/2) ==> Perm(output[t*2], write);
29              requires t<(N/2) ==> Perm(output[t*2+1], write);
30              requires t<(N/2) ==> Perm(tree[0][t*2], write);
31              requires t<(N/2) ==> Perm(tree[0][t*2+1], write);
32              requires (\forall* int i; 1<=i && i<H; Perm(tree[i][t], write));
33
34              ensures N==32 && H==6;
35              ensures t<(N/2) ==> Perm(input[t*2], read);
36              ensures t<(N/2) ==> Perm(input[t*2+1], read);
37              ensures t<(N/2) ==> Perm(output[t*2], write);
38              ensures t<(N/2) ==> Perm(output[t*2+1], write);
39              ensures t<(N/2) ==> Perm(tree[0][t*2], write);
40              ensures t<(N/2) ==> Perm(tree[0][t*2+1], write);
41              ensures (\forall* int i; 1<=i && i<H; Perm(tree[i][t], write));
42
43          // Thread code
44          {
45              if(t < (N/2)) {
46                  // Input copy
47                  tree[0][t*2]   = input[t*2];
48                  tree[0][t*2+1] = input[t*2+1];
49                  // First step of up-sweep
50                  tree[1][t] = tree[0][t*2] + tree[0][t*2+1];
51              }
52
53              // Remaining steps of the upsweep phase
54              int level=1;
55              loop_invariant N==32 && H==6;
56              loop_invariant tree != null;
57              loop_invariant level>=1 && level<H;
58              loop_invariant t>=0 && t<N;
59              loop_invariant (\forall* int i; level<=i && i<H; Perm(tree[i][t], write));
60              loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2], write));
61              loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2+1],
                    write));
62              while((level+1)<H) {
63                  level = level+1;
64                  assert Perm(tree[level-1][t], write);
65                  barrier(local) {
66                      requires tree != null;
67                      requires N==32 && H==6;
68                      requires level>=1 && level<H;
69                      requires t>=0 && t<N;
70                      requires Perm(tree[level-1][t], write);
71
72                      ensures tree != null;
73                      ensures N==32 && H==6;
74                      ensures level>=1 && level<H;
75                      ensures t>=0 && t<N;
76                      ensures t<(N/2) ==> (\forall* int k; 0<=k && k<2; Perm(tree[level-1][t*2+k],
                            write));
77                  }
78                  // Do next up-sweep step
79                  if(t < (N/2)) {
80                      assert tree != null;
81                      assert N==32 && H==6;
```

```
82                  assert level>=1 && level<H;
83                  assert t>=0 && t<N;
84                  assert (\forall* int k; 0<=k && k<2; Perm(tree[level-1][t*2+k], write));
85                  tree[level][t] = tree[level-1][t*2+id(0)] + tree[level-1][t*2+id(1)];
86              }
87          }
88
89          // Set the root to zero, preparation for down-sweep
90          tree[H-1][t] = 0;
91
92          if(t < (N/2) && level-1>0) {
93              // First step of downsweep
94              tree[level-1][t*2+1] = tree[level-1][t*2] + tree[level][t]; // Right child
95              tree[level-1][t*2] = tree[level][t];                        // Left child
96          }
97
98          // Remaining steps of the downsweep
99          loop_invariant N==32 && H==6;
100         loop_invariant level>=1 && level<H;
101         loop_invariant t>=0 && t<N;
102         loop_invariant tree != null;
103         loop_invariant (\forall* int i; level<=i && i<H; Perm(tree[i][t], write));
104         loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2], write));
105         loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2+1],
                write));
106         while((level-1)>0) {
107             level = level-1;
108             assert t<(N/2) ==> Perm(tree[level][t*2+id(0)], write);
109             assert t<(N/2) ==> Perm(tree[level][t*2+id(1)], write);
110             barrier(local) {
111                 requires N==32 && H==6;
112                 requires t>=0 && t<N;
113                 requires level>=1 && level<H;
114                 requires t<(N/2) ==> (\forall* int k; 0<=k && k<2; Perm(tree[level][t*2+k],
                        write));
115                 requires tree != null;
116
117                 ensures N==32 && H==6;
118                 ensures level>=1 && level<H;
119                 ensures t>=0 && t<N;
120                 ensures Perm(tree[level][t], write);
121                 ensures tree != null;
122             }
123             // Do next down-sweep step
124             if(t < (N/2)) {
125                 assert tree != null;
126                 assert N==32 && H==6;
127                 assert level>=1 && level<H;
128                 assert t>=0 && t<N;
129                 assert Perm(tree[level][t], write);
130                 tree[level-1][t*2+1] = tree[level-1][t*2] + tree[level][t]; // Right child
131                 tree[level-1][t*2] = tree[level][t];                        // Left child
132             }
133         }
134         // Copy the result from the tree to the output array
135         if(t < (N/2)) {
136             output[t*2] = tree[0][t*2];
137             output[t*2+1] = tree[0][t*2+1];
138         }
139     }
140 }
141 }
```

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 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.

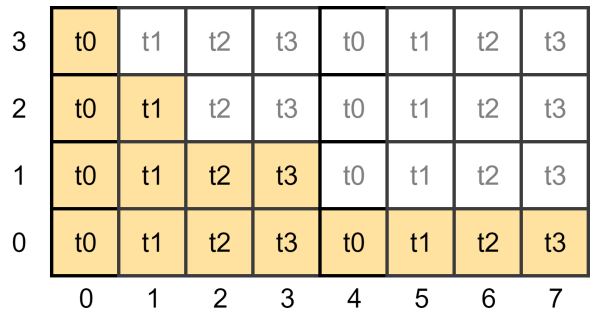

Figure 10: Permissions of the tree before the upsweep, updated for usage with half the number of threads

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.

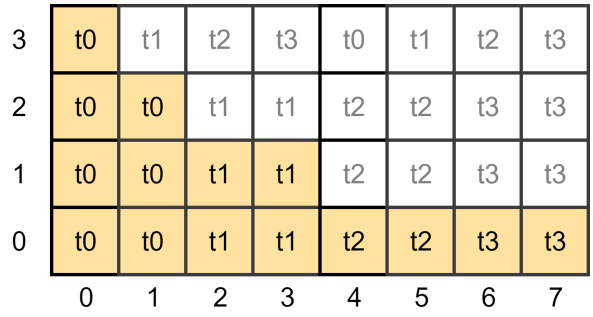

Figure 11: Permissions of the tree after the upsweep, updated for usage with half the number of threads

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.

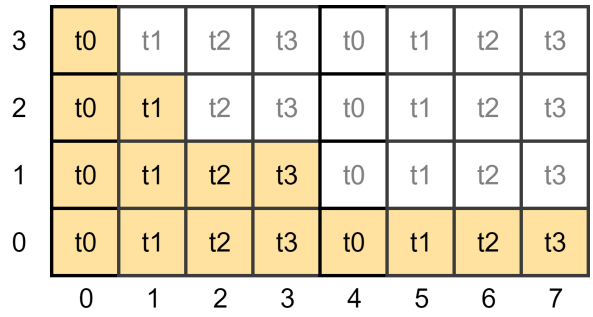

Figure 12: Permissions of the tree after the downsweep, updated for usage with half the number of threads

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.

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 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.

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.

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);
    requires (\forall* int i; 0<=i && i<N*2; Perm(input[i], read));
    requires (\forall* int i; 0<=i && i<N*2; Perm(output[i], write));
    requires (\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)
```

```
16          requires Perm(input[t], read);
17          requires Perm(input[t+N], read);
18
19          requires Perm(output[t], write);
20          requires Perm(output[t+N], write);
21
22          requires (\forall* int k; 0<=k && k<2;
23              (\forall* int i; 0<=i && i<H; Perm(tree[i][k*N+t], write))
24          );
25      {
26          ////////// INPUT COPY
27          tree[0][t] = input[t];
28          tree[0][t+N] = input[t+N];
29
30          ////////// UPSWEEP
31          int level=0;
32          loop_invariant level>=0 && level<H;
33          loop_invariant t>=0 && t<N;
34          // Old permisison layout going away, top row excluded: (t1 | t2 | t3 | t4 | ...)
35          loop_invariant (\forall* int k; 0<=k && k<2;
36              (\forall* int i; level<=i && i<H; Perm(tree[i][k*N+t], write))
37          );
38          // New permission layout appearing, lowest row already has it: (t1 | t1 | t2 | t2 | ...)
39          loop_invariant (\forall* int k; 0<=k && k<2;
40              (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2+k], write))
41          );
42          while((level+1)<H) {
43              level = level+1;
44              barrier(runThreads) {
45                  context t>=0 && t<N;
46                  context level>=1 && level<H;
47
48                  requires Perm(tree[level-1][t], write);
49                  requires Perm(tree[level-1][t+N], write);
50
51                  ensures (\forall* int k; 0<=k && k<2; Perm(tree[level-1][t*2+k], write));
52              }
53              tree[level][t] = tree[level-1][t*2] + tree[level-1][t*2+1];
54          }
55
56          ////////// DOWNSWEEP
57          tree[H-1][t] = 0; // Set the root of the tree to zero
58
59          loop_invariant level>=0 && level<H;
60          loop_invariant t>=0 && t<N;
61          // Old permission layout coming back, top row already has it: (t1 | t2 | t3 | t4 | ...)
62          loop_invariant (\forall* int k; 0<=k && k<2;
63              (\forall* int i; level<=i && i<H; Perm(tree[i][k*N+t], write))
64          );
65          // New permission layout going away, bottom row excluded: (t1 | t1 | t2 | t2 | ...)
66          loop_invariant (\forall* int k; 0<=k && k<2;
67              (\forall* int i; 0<=i && i<level; Perm(tree[i][t*2+k], write))
68          );
69          while(level>0) {
70              // Do next down-sweep step
71              tree[level-1][t*2+1] = tree[level-1][t*2] + tree[level][t]; // Right child
72              tree[level-1][t*2] = tree[level][t];                        // Left child
73
74              level = level-1;
75
76              barrier(runThreads) {
77                  context t>=0 && t<N;
78                  context level>=0 && level<H;
79
80                  requires Perm(tree[level][t*2], write);
81                  requires Perm(tree[level][t*2+1], write);
82
83                  ensures Perm(tree[level][t], write);
84                  ensures Perm(tree[level][t+N], write);
85              }
86          }
87
88          ////////// OUTPUT COPY
89          output[t] = tree[0][t];
```

```
90              output[t+N] = tree[0][t+N];
91          }
92      }
93  }
```

todo: Write about:

1. Discussions with Stefan
2. \matrix and \array (removed tree!=null)

todo: Steps:

1. Show version of previous research
2. Extend to downsweep and program ensures (Permissions V2, show the encountered problems)
3. Half threads version (HalfThreads, show encountered problems)
4. Explain functional ideas (new permissions layouts)

4 Verifying functionality

todo: verification process of the functionality of the program

5 Discussion

todo: summary

5.1 Verification performance

todo: Current performance, options and solutions for the future

1. Scaling with classes
2. Scaling with methods
3. Improvements in Z3
4. Better hardware
5. Better progress during verification

5.2 Results

todo: result description: verified

5.3 Related work

todo: similar verification projects

6 Conclusion

todo: summary

6.1 Research question answer

todo: works, not for huge programs though

6.2 Limitations and problems

todo: tool limitations, change code for verification process

6.3 Research value

todo: do larger programs, improve tool, concurrency

6.4 Future work

1. Larger programs
2. Improve tool
3. Handle more functional details: overflow, underflow, etc.

7 References

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

[2] 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.

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

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

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