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 **5**
- 1.1 Research domain . . . 5
  - 1.1.1 GPU computing . . . 5
  - 1.1.2 Prefix Sum . . . 5
  - 1.1.3 Verification . . . 6
- 1.2 Problem statement . . . 6
- 1.3 Research questions . . . 6
- 1.4 Approach . . . 7
- 1.5 Report structure . . . 7

2 Prefix sum algorithm **8**
- 2.1 Prefix Sum algorithm . . . 8
- 2.2 Two-dimensional Arrays As Storage . . . 10
- 2.3 Algorithmic description . . . 11

3 Verifying permissions **12**
- 3.1 Starting point . . . 12
- 3.2 Verifying the downsweep array permissions . . . 14

4 Verifying functionality **17**

5 Discussion **18**
- 5.1 Results . . . 18
- 5.2 Related work . . . 18

6 Conclusion **19**
- 6.1 Research question answer . . . 19
- 6.2 Limitations and problems . . . 19
- 6.3 Research value . . . 19
- 6.4 Future work . . . 19

7 References **20**

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.

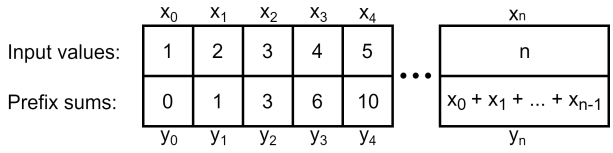

Figure 1: Prefix Sum description

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.

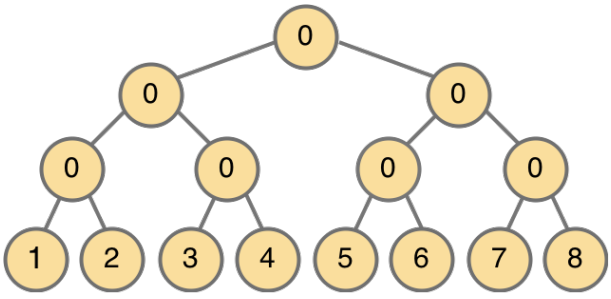

Figure 2: Tree at the start of the up-sweep phase

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

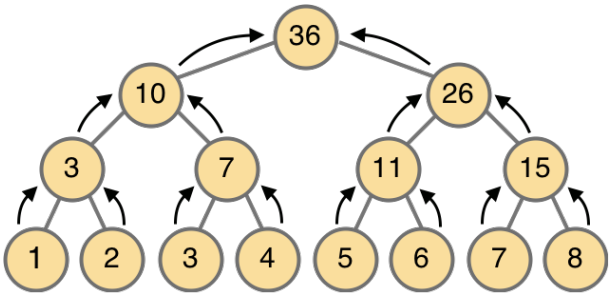

Figure 3: Tree after the up-sweep phase

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.

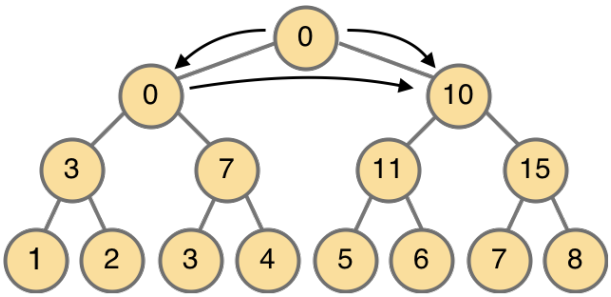

Figure 4: The first step of the down-sweep phase

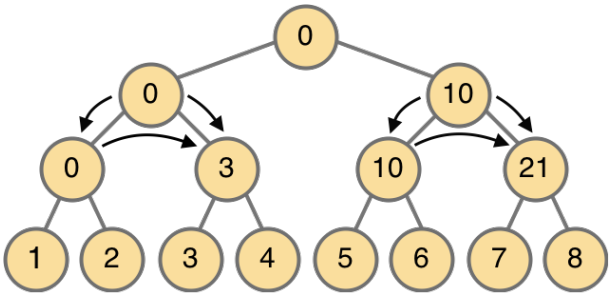

Figure 5: The second step of the down-sweep phase

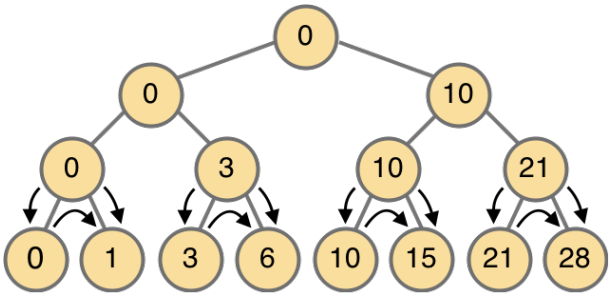

Figure 6: The final step of the down-sweep phase

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(temp[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(temp[j][i], write))
    );
    ensures (\forall* int i; 0<=i && i<N; Perm(temp[0][i], write));
    // Kernel method
    void prefixSum(int N, int H, int[N] input, int[N] output, int[H][N] temp) {
        // 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(temp[0][t*2], write);
            requires t<(N/2) ==> Perm(temp[0][t*2+1], write);
            requires (\forall* int j; 1<=j && j<H; Perm(temp[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(temp[0][t*2], write);
            ensures t<(N/2) ==> Perm(temp[0][t*2+1], write);
            ensures (\forall* int j; 1<=j && j<H; Perm(temp[j][t], write));
        // Thread code
        {
            // Only use the first half of threads
            if(t < (N/2)) {
                // Input copy
                temp[0][t*2]   = input[t*2];
                temp[0][t*2+1] = input[t*2+1];
                // First step of upsweep
                temp[1][t] = temp[0][t*2] + temp[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(temp[i][t], write));
51            loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(temp[i][t*2], write));
52            loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(temp[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(temp[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(temp[level][t], write));
65                    ensures t<(N/2) ==> Perm(temp[level-1][t*2], write));
66                    ensures t<(N/2) ==> Perm(temp[level-1][t*2+1], write));
67                }
68                // Do next upsweep step
69                if(t < (N/2)) {
70                    temp[level][t] = temp[level-1][t*2] + temp[level-1][t*2+1];
71                }
72            }
73
74            // Set the root to 0
75            temp[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(temp[i][t], write));
82            loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(temp[i][t*2], write));
83            loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(temp[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(temp[level][t*2], write);
91                    requires t<(N/2) ==> Perm(temp[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(temp[level][t], write);
97                    ensures t<(N/2) ==> Perm(temp[level-1][t*2], write);
98                    ensures t<(N/2) ==> Perm(temp[level-1][t*2+1], write);
99                }
100               // Do next downsweep step
101               if(t < (N/2)) {
102                   temp[level-1][t*2+1] = temp[level-1][t*2] + temp[level][t]; // Set right child
103                   temp[level-1][t*2] = temp[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] = temp[0][t*2];
110               output[t*2+1] = temp[0][t*2+1];
111           }
112       }
113   }
114 }
```

3.2 Verifying the downsweep array permissions

Listing 5: Specification as written in the previous project, only the upsweep part is verified

```
class Ref {
    // Requires/Ensures for complete kernel
    requires N==32 && H==6;
    requires temp != 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(temp[j][i], write))
             );

    ensures N==32 && H==6;
    ensures temp != 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(temp[j][i], write))
             );

    // Kernel method
    void test(int N, int H, int[N] input, int[N] output, int[H][N] temp) {

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

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

            // Remaining steps of the upsweep phase
            int level=1;
            loop_invariant N==32 && H==6;
            loop_invariant temp != null;
            loop_invariant level>=1 && level<H;
            loop_invariant t>=0 && t<N;
            loop_invariant (\forall* int i; level<=i && i<H; Perm(temp[i][t], write));
            loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(temp[i][t*2], write));
            loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(temp[i][t*2+1],
                 write));
            while((level+1)<H) {
                level = level+1;
                assert Perm(temp[level-1][t], write);
                barrier(local) {
                    // This barrier changes permissions from this situation:
                    // t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 ...
                    // to the following situation:
```

```
69                  // t1 | t1 | t2 | t2 | t3 | t3 | t4 | t4 ...
70                  // for levels 0 to (H-2), top row keeps the old layout
71                  // (t1 means thread 1 has permission for that slot)
72
73                  requires temp != null;
74                  requires N==32 && H==6;
75                  requires level>=1 && level<H;
76                  requires t>=0 && t<N;
77                  requires Perm(temp[level-1][t], write);
78
79                  ensures temp != null;
80                  ensures N==32 && H==6;
81                  ensures level>=1 && level<H;
82                  ensures t>=0 && t<N;
83                  ensures t<(N/2) ==> (\forall* int k; 0<=k && k<2; Perm(temp[level-1][t*2+k],
                        write));
84              }
85              // Do next up-sweep step
86              if(t < (N/2)) {
87                  assert temp != null;
88                  assert N==32 && H==6;
89                  assert level>=1 && level<H;
90                  assert t>=0 && t<N;
91                  assert (\forall* int k; 0<=k && k<2; Perm(temp[level-1][t*2+k], write));
92                  temp[level][t] = temp[level-1][t*2+id(0)] + temp[level-1][t*2+id(1)];
93              }
94          }
95
96          // Set the root to zero, preparation for down-sweep
97          temp[H-1][t] = 0;
98
99          if(t < (N/2) && level-1>0) {
100             // First step of downsweep
101             temp[level-1][t*2+1] = temp[level-1][t*2] + temp[level][t]; // Right child
102             temp[level-1][t*2] = temp[level][t];                        // Left child
103         }
104
105         // Remaining steps of the downsweep
106         loop_invariant N==32 && H==6;
107         loop_invariant level>=1 && level<H;
108         loop_invariant t>=0 && t<N;
109         loop_invariant temp != null;
110         loop_invariant (\forall* int i; level<=i && i<H; Perm(temp[i][t], write));
111         loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(temp[i][t*2], write));
112         loop_invariant t<(N/2) ==> (\forall* int i; 0<=i && i<level; Perm(temp[i][t*2+1],
                write));
113         while((level-1)>0) {
114             level = level-1;
115             assert t<(N/2) ==> Perm(temp[level][t*2+id(0)], write);
116             assert t<(N/2) ==> Perm(temp[level][t*2+id(1)], write);
117             barrier(local) {
118                 // This barrier changes permissions from this situation:
119                 // t1 | t1 | t2 | t2 | t3 | t3 | t4 | t4 ...
120                 // to the following situation:
121                 // t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 ...
122                 // for levels 1 to (H-2), top and bottom rows keep the old layout
123                 // (t1 means thread 1 has permission for that slot)
124
125                 requires N==32 && H==6;
126                 requires t>=0 && t<N;
127                 requires level>=1 && level<H;
128                 requires t<(N/2) ==> (\forall* int k; 0<=k && k<2; Perm(temp[level][t*2+k],
                        write));
129                 requires temp != null;
130
131                 ensures N==32 && H==6;
132                 ensures level>=1 && level<H;
133                 ensures t>=0 && t<N;
134                 ensures Perm(temp[level][t], write);
135                 ensures temp != null;
136             }
137             // Do next down-sweep step
138             if(t < (N/2)) {
139                 assert temp != null;
```

```
140                     assert N==32 && H==6;
141                     assert level>=1 && level<H;
142                     assert t>=0 && t<N;
143                     assert Perm(temp[level][t], write);
144                     temp[level-1][t*2+1] = temp[level-1][t*2] + temp[level][t]; // Right child
145                     temp[level-1][t*2] = temp[level][t];                        // Left child
146                 }
147             }
148             // Copy the result from the tree to the output array
149             if(t < (N/2)) {
150                 output[t*2] = temp[0][t*2];
151                 output[t*2+1] = temp[0][t*2+1];
152             }
153         }
154     }
155 }
```

---

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 Results

todo: result description: verified

5.2 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.