Optimization, Specification and Verification of the Prefix Sum Program in an OpenCL Environment

Thijs Wiefferink
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
t.w.wiefferink@student.utwente.nl

ABSTRACT

The Prefix Sum is an algorithm used as a building block for various other algorithms, for example radix sort, quicksort and lexically comparing strings. Implementing the Prefix Sum algorithm on the CPU is trivial, but a parallel approach with OpenCL is more complicated. An implementation in OpenCL has been made, and optimized to minimize branch divergence by comparing two different storage solutions. These storage solutions have been benchmarked in order to show the difference in execution time. In addition to performance, verification of the algorithm is important. The two aspects that need to be verified are the absence of data races and functional correctness. This paper describes a specification that covers the absence of data races, by using Permission-Based Separation Logic. The first part of this specification (the up-sweep phase of the algorithm) has been verified using the VerCors tool.

Keywords

Prefix Sum, VerCors, Permission-Based Separation Logic, OpenCL, GPU, Formal verification

1. INTRODUCTION

1.1 GPU properties and verification

GPUs are powerful devices designed to rapidly manipulate and alter memory to accelerate the creation of images, while for example playing a game. However, the GPU is also being used more and more for general purpose computing, which is a use case normally handled by the CPU. An advantage of using the GPU instead of the CPU is the efficient parallel execution on large data sets. When using a GPU for general computing, one has to select an API for the communication with the GPU. Widespread APIs include CUDA, DirectCompute and OpenCL [1] (Open Computing Language). Of these APIs OpenCL is very desireable due to its hardware vendor independency and open source nature. OpenCL is a language specification made by the Khronos Group, which can be used on a wide range of devices, including GPUs of AMD and NVIDIA, and CPUs of Intel and AMD.

Programs that are parallel in nature have an advantage when executed on the GPU instead of the CPU, because of the large number of threads available on the GPU. However, there are some challenges to overcome before a GPU can be used as a general computing device. The first challenge is the prevention of 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 program functionally correct, which checks if the program actually accomplishes the task it is written for. The first aspect is harder to prove than in a singlethreaded program because of the different possible interleavings of thread executions. Verifying programs is useful for use in safety critical systems, such as in the airplane industry. Also for industries like the car and medical equipment industry the verification of programs is required for certain aspects. A third challenge with implementing a program on the GPU is minimizing branch divergence [8].

To overcome the challenges mentioned above, the first step is to make a formal specification of a program to describe exactly what should happen. The second step is verifying this specification. A specification has been made with Permission-Based Separation Logic [6], which is a way to describe exactly what the result of a program should be. The specification is used to describe read and write accesses to parts of the memory, which can in turn be used to prove the program data race free after successful verification. It can also be used to describe the result of the program, which in turn could be used to verify that the program is functionally correct, that is however out of scope for this research. To prove that the program adheres to the Permission-Based Separation Logic specification the VerCors [2] tool has been used. VerCors is capable of proving that the OpenCL programs match their specification, which includes the absence of data races and the functional correctness of the program.

1.2 Prefix Sums

The research goal is to verify a Prefix Sum program by using Permission-Based Separation Logic and the VerCors tool. The prefix Sum program will be written in OpenCL, and benchmarked to check the performance. Prefix Sum is an algorithm also known as Scan. 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.

23^{th} *Twente Student Conference on IT* June 22^{st}, 2015, Enschede, The Netherlands.

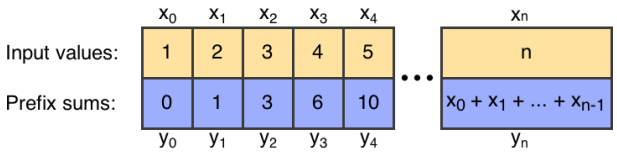

Figure 1: Prefix Sum description

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 [4]. 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.

The algorithm to calculate prefix sums can be structured in such a way that large amounts of data can be processed in parallel. There are existing algorithms that are multi threaded and meant for the GPU. Chapter 39 *Parallel Prefix Sum (Scan) with CUDA* of the book GPU Gems 3 [11] has a step by step approach for implementing an efficient Prefix Sum program in CUDA.

2. RESEARCH GOAL

2.1 Problem Statement

The problem with programs on the GPU is that data races can occur, which are hard to detect with simple testing. It is possible that testing never shows a data race problem, but that it still occurs after long term usage. Another problem is to prove that the end result of a program is always correct, which also requires the absence of data races. The only way to be sure a program is functionally correct and data race free is to validate these two aspects of the program. In order to do that the GPU program has to be formally specified in Permission-Based Separation Logic, which is a challenge for bigger programs like the Prefix Sum used in this case study. A challenge with the implementation of the Prefix Sum algorithm is avoiding branch divergence [8], which would slow down the implementation considerably. Branch divergence occurs when threads that execute the kernel are forced to execute different instructions, disturbing the SIMT (Single-Instruction, Multiple-Thread) principle explained in Section 3.1.

2.2 Research Questions

The research addresses the following research questions based on the problem statement:

1. How can branch divergence be avoided in the implementation of the Prefix Sum?
2. How can the Prefix Sum program be formally specified with Permission-Based Separation Logic?
3. How can the Prefix Sum program be proven to have no data races?

3. BACKGROUND

3.1 GPU execution model

Programs in OpenCL on the GPU have a different execution model than normal programs running on a CPU. The GPU has work items, these are single threads. These work items are bundled into work groups, which all consist of a known number of work items for a certain kernel execution. Because the size of a work group is limited (determined by hardware and drivers), a GPU program uses multiple work groups to execute a large task. Threads in the same work group can easily be synchronized with a barrier, which when reached halt the execution until all threads arrived. But threads in different work groups cannot be synchronized easily. The number of threads in a work group is called the *local size*. The total number of threads used for execution is called *global size*, and is normally determined by the size of the input of the program. To illustrate this model we will look at the vector addition program, visualized in Figure 2. This program takes two arrays of the same size as input, and has an output array which is the sum of the two input arrays. With this program we will use one thread for each addition. In the example in Figure 2 we have an input size of 8, therefore we take a *global size* of 8. In this example a *local size* of 2 is drawn, which would lead to four work groups of two work items each. Note that, normally the *local size* is much higher. The maximum *local size* for the NVIDIA GTX 770 GPU, which is used for this research, is 1024. This number can be different for other GPUs. The *global size* is basically unlimited, but is indirectly limited by the available memory and the way you can split a certain task amongst threads.

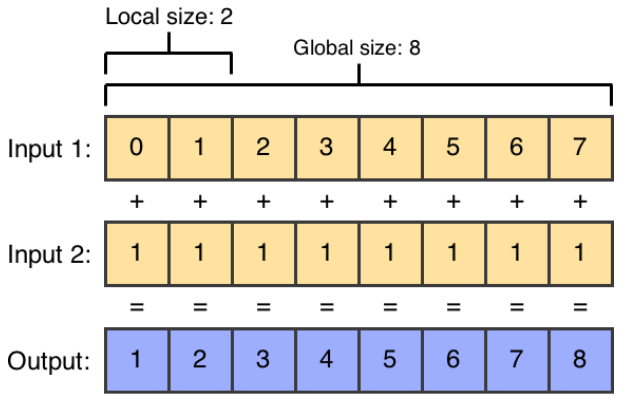

Figure 2: Vector addition example

The placement of work items into work groups is also important for the actual execution of a program. GPUs use a SIMT (Single-Instruction, Multiple-Thread) architecture. In this architecture a single program (set of instructions) is used by multiple threads and executed at the same time. The only difference between two threads is that they work on different parts of the data. When using the vector addition of Figure 2 as example, the addition of one cell of input 1 and 2 would be done by a single thread. To determine which thread performs the addition for which position in the vector, the *global thread id* is used. The *global id* is a number that is different for each thread and ranges from zero to the *global size* minus one. Therefore the *global id* can be used as an index in the data set, in this case the thread with *global id* 0 would perform the addition of the first values of the input arrays, and place the result in the first slot of the output array.

The SIMT execution model and work group separation of threads are the two most important aspects of the OpenCL architecture for the planned case study. The paper *An Introduction to the OpenCL Programming Model* [13] explains the above concepts in more detail and also explains other aspects of the OpenCL programming model.

3.2 Permission-Based Separation Logic

For the specification of the GPU kernel Permission-Based Separation Logic has been used. This specification language started with Hoare logic [9] and has been extended

to Separation Logic [12] to make it possible to reason about programs with pointers. After that it has been extended with permissions by Bornat et al. [7]. A full permission, represented by 1, indicates a write permission, and a fraction between 0 and 1 denotes a read permission. The permissions for a certain memory location can be split to allow multiple threads to read, and can also be joined which can give write permission if it adds up to 1. This system allows different threads to read the same data, but prevents them from writing at the same time.

The specification as implemented in VerCors has a couple of constructs to define the program structure. The first line of the specification. `class Ref {`, serves as identification of a class. After that methods can be placed in the class, with a signature like `void prefixSum(...)`, such a method represents the kernel in this case. There is a special block to indicate parallel executed threads, to use in kernels: `par threads(int t=0..N;;true)`, which is a parallel block. A parallel block looks similar to a for loop, but instead represents threads of a kernel. The thread number, often called `tid` can be referenced by `t` in the specification.

The specification language has a couple of basic clauses to specify the pre- and postconditions of the kernel, a thread and a barrier. A precondition can be expressed using `requires <boolean>;` which ensures that this property holds when the code is run. Requirements of the kernel are assumed to be provided by the host code, in order to be able to use these permissions inside the kernel. A postcondition can be expressed using `ensures <boolean>;` which ensures that after the code has been run, the condition holds. These statements could be used to express the desired end result of the kernel. Apart from this there is a loop invariant to express statements that should hold before, during and after a loop: `loop invariant <boolean>;`. There is also a logical implication in the language written as: `<boolean> ==> <boolean>;`.

To reason about read and write permissions to variables and array indices, there is a special specification construct: `Perm(<variable or array position>, <write|read>)`. This predicate will give a boolean result, indicating if the permission queried for has been met. To reason about a range of conditions the following construct can be used: `(\forall* <variable declare>; <variable bounds>; <boolean condition>);`. This is useful for example to give a thread permission to access a complete column of a matrix, which would result in a statement similar to this: `(\forall* int i; i>0 && i<10; Perm(array[0][i], write));`.

More detailed information about Permission-Based Separation Logic and how it is used to reason about GPU programs can be found in *Specification and Verification of GPGPU Programs* [6] by S. C. C. Blom, M. Huisman and M. Mihelčić.

4. IMPLEMENTATION

4.1 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 Algorithm 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.

The algorithm implemented for this research is made by Blelloch [3], and can be used concurrently. The description of Nguyen [11] has been followed to implement the algorithm. The *Simple-OpenCL* library [10] of O. A. Huguet

Algorithm 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]
```

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 3. 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 4 shows the end result of this phase.

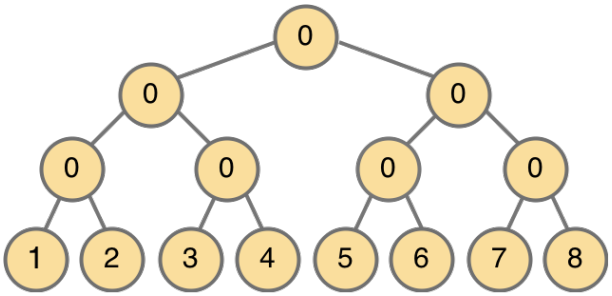

Figure 3: Tree at the start of the up-sweep phase

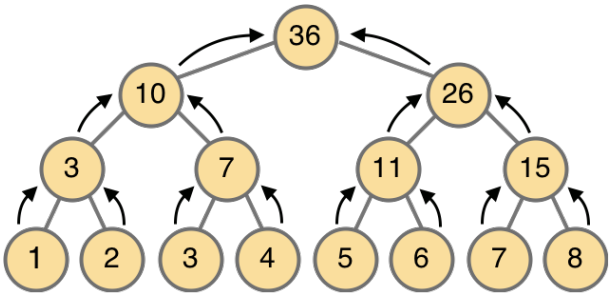

Figure 4: Tree after the up-sweep phase

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 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 5 illustrates the first step, the right node will get the value $0 + 10$, the left node will get value 0. Figure 6 shows the result of the second step, and Figure 7 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.

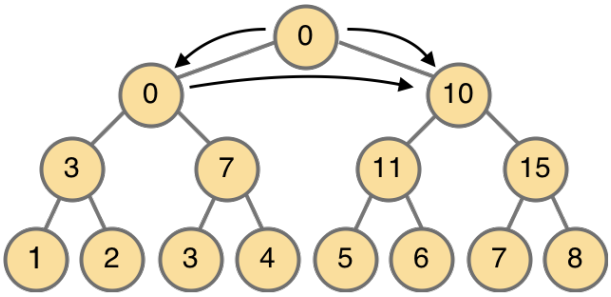

Figure 5: The first step of the down-sweep phase

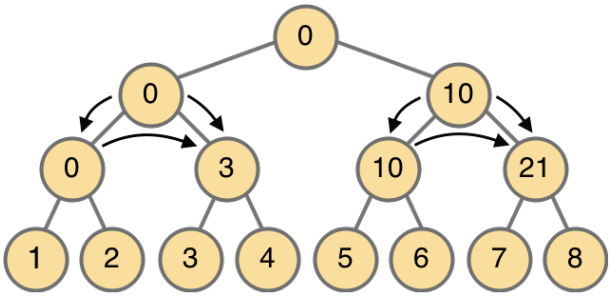

Figure 6: The second step of the down-sweep phase

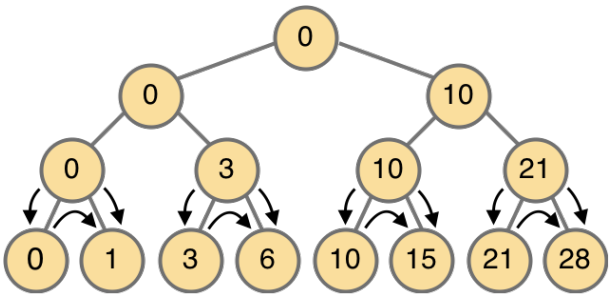

Figure 7: The final step of the down-sweep phase

4.2 Two-dimensional Arrays As Storage

Now that the algorithm of the Prefix Sum has been explained in Section 4.1, 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 4 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 4.1. 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 [8]. 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). A performance comparison will be given in Section 4.4.

4.3 Algorithmic description

The algorithm with the two-dimensional arrays works as described in Algorithm 2 (up-sweep phase) and Algorithm 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: $log_2(n) - 1$.

Algorithm 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]
```

Algorithm 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]
```

4.4 Performance Comparison

The impact of the change in storage type as explained in Section 4.2 has been measured by testing the time it took to run a kernel with both implementations. The test has been performed with all input sizes from 2 until 1024, that are a power of 2. The maximum of 1024 is chosen because this is the maximum number of threads the used graphics card allows in one workgroup. The test has been repeated 10 times, the average result has been used. Figure 8 shows the results of this test, for the first approach with the tree in an array (called one-dimensional array), and the second approach with an two-dimensional array. The difference between the tested kernels in the range of 2 until 32 is almost none, but from 64 until 1024 the difference is significant. At 1024 the 1-dimensional array takes 270 milliseconds to run, at which the 2-dimensional version takes 120 milliseconds to run. This means that the prediction was correct, the two-dimensional storage of the tree is better for the performance for these input sizes. However be aware that this optimization is a trade off between memory usage and speed. The two-dimensional array might be faster, but it does use 5 times as much memory as the one-dimensional array version. With higher input sizes it

might be better to use the one-dimensional version since the memory of the GPU might run out otherwise.

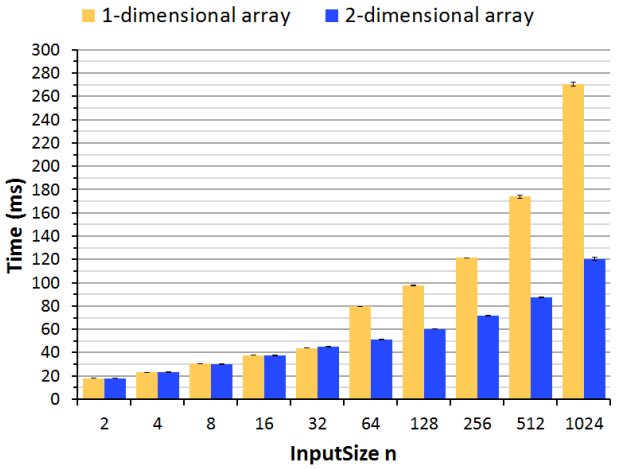

Figure 8: Comparison of kernel execution speed (average of 10 runs)

4.5 Extending to multiple workgroups

The current algorithm is suitable for kernels that make use of one workgroup, but cannot be ran on multiple workgroups. The problem is that the workgroups require synchronization to correctly compute the result with this algorithm, and synchronization is not trivial between workgroups. Because of this limitation the above algorithm can only compute the prefix sum for an input size that is at maximum two times the maximum workgroup size (which is typically around 1024).

In order to extend the described algorithm of Section 4.1 to work with bigger input sizes, and therefore multiple workgroups, it needs some modifications. The basic idea to calculate the prefix sums of a bigger input is the following:

1. Split the input into parts that a workgroup can do individually.
2. Calculate the prefix sums of the different parts (using the algorithm from Section 4.3), save the total sum of a part for later.
3. Calculate the prefix sum on the array with all sums saved at step 2.
4. Add the sums calculated at step 3 to the partial prefix sums calculated at step 2, this results in a completed prefix sum for the complete input sequence.

The above general idea as described by H. Nguyen [11] needs some refinement to be sufficient for all input sizes. One problem of the algorithm is that the array with the sums of the inputs, as saved in step 2, can still become to large to handle by one workgroup. Because of this the same algorithm would need to be applied again, but now on the array with the sums. To fix this problem the algorithm can be made recursive, using the same idea over and over again until the size of the sums array is small enough to compute with one workgroup. The description of the recursive Prefix Sum algorithm to calculate it for unlimited input sizes can be found below.

The workgroupsize stands for the maximum number of work items that can be used in one workgroup, inputSize stands for the size of the input array, padded to a power of 2. The workgroupsize is different for each GPU model, and can be determined by the host code before running the kernel. The algorithm starts at item 1, and ends at either item 1b or item 2e.

Calculate Prefix Sum for n input values:

1. Base case when $n \leq$ (workgroupsize$*$2):
   (a) Calculate the Prefix Sum with a single workgroup computation.
   (b) Return the result.
2. Recursive case when $n >$ (workgroupsize$*$2):
   (a) Split the input into parts that are small enough to be handled by a single workgroup.
   (b) Calculate the prefix sum for each of these parts (uses 1 kernel with multiple workgroups, each workgroup handling a part) and save the full sum of each part into an array (total is known after the upsweep phase).
   (c) Calculate the prefix sum for the array with the sums of the parts (contains 1 sum for each workgroup, as added in step 2b), this is a recursive call, therefore starts at item 1 again and might require a recursive case again. This will always call the method with a smaller n as the current call, since even if the graphic card only allows 1 item per workgroup, it will be divided by 2.
   (d) Add the values calculated in step 2c to the output of step 2b, workgroup synchronization is not required for this step because these additions operate completely disjoint.
   (e) Return the result.

The above algorithm has been fully implemented using OpenCL and C++ (for source code check [14]). This implementation will serve as a basis for the specification, providing the structure of the kernel.

5. SPECIFICATION

5.1 General

After the implementation, the created kernel has been specified. The code of the kernel has been written in OpenCL, but to use the code and specification in the VerCors tool it has to be transformed to PVL. This is required because currently VerCors does not support OpenCL code as input. PVL is a toy language, which supports a subset of C. Only the kernel will be specified and verified, the host code is a step to consider for future research. In Algorithm 4 the specified Prefix Sum kernel can be found. The specification only focuses on proving the program data race free, the functional correctness of the program is a next step to be considered for future research. The specification is for a single workgroup algorithm of the Prefix Sum, multiple workgroups would require extra logic for building the sums array, and a verification of an adder kernel. The specification is explained in the next section.

5.2 Kernel Specifications

The first line of the specification in Algorithm 4 declares the name of the class, and is used to serve as start of the program. On line 18 the kernel method is declared. The N represents the input size of the algorithm and the H represents the number of levels required in the binary tree for the computation. The input array contains the input numbers for the prefix sum, and has size N. The output array will be used to give back the result, and is also size N. The temporary array is given the width N and height H, and will be used to store the tree required for the calculation described in Section 4.1.

The specifications from line 3 until line 16 are the pre- and postconditions for the complete kernel. These specifications declare a static input size as N=32, and the height of the tree for this input size as H=6. The relation between N and H is described by H=$log2(N)+1$, but proving such a definition is not trivial for the VerCors tool. If specifying such a condition was possible then the specification could be proven for more input sizes, instead of only for one static number (32 in this case). These specifications requires read permissions for the complete input array, since the kernel will need to read from the input. It also requires write permission for all positions of the *temp* array. These permissions are supplied by the host when invoking the kernel.

5.3 Thread Specifications

Inside the method that represents the kernel there is a parallel block (line 20), this represents the threads that run the kernel. The parallel block specifies that there are N threads, with numbers 0 until N-1. The specification uses N threads, but actually there are just N/2 threads required for the program. This is due to a more complicated array indexing that using half of the threads would require, which is currently unsupported by the VerCors tool. Before the opening curly brace of the parallel block there are specifications for single threads. The defined N and H are repeated, which will be seen more often in the specification.

Each thread with an id less than N/2 requires read permission to 2 different locations in the input array, namely $t*2$ and $t*2+1$. This is to enable the thread to copy the input values it will need for the first step of the up-sweep phase as seen in Figure 4. The indexing for the *temp* array is the same, with an additional index 0, representing the lowest level of the tree. Next to these permissions the remainder of the *temp* array will be divided amongst all threads, which all get a complete column without the lowest level. This permission situation is represented in Figure 9. In this figure the horizontal plane represents the N dimension of the array, and the vertical plane represents the H dimension. The number in the cells indicate the number of the thread that has write access to the cell.

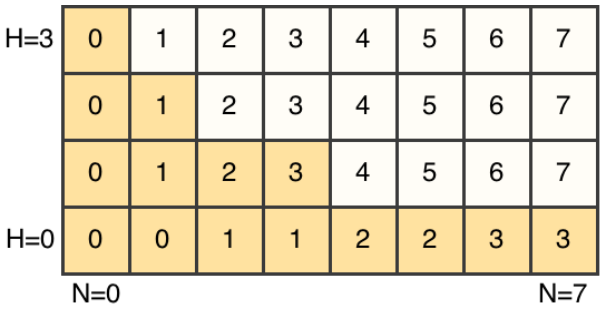

Figure 9: Permissions for the up-sweep phase

Line 40 until line 46 are to copy the input values to the lowest level of the tree. Only the first half of the threads will be doing this, all handling 2 values. Because of the permission distribution mentioned in the last paragraph these threads can immediately do the first up-sweep step, which will be written to the first locations of the level above (N/2 slots will be used on that level).

Line 54 until line 73 specify the up-sweep part of the algorithm. The while-loop has a couple of loop invariants, the first three restate N and H, indicate the bounds of the thread numbers, and indicate the bounds of the level (which is our loop variable). The loop invariant at line 51 specifies that this thread should have permission to write in all locations from *level* until H-1, at column t, as visualized in Figure 9. Since *level* starts at 1, and all threads start with their complete column except the lowest level, this will be correct before the loop starts. Inside the loop, first the variable *level* is increased, then a barrier is enforced, and after that a step of the up-sweep phase happens. The barrier transfers the write permissions that the threads used for the previous step to the threads that will be reading from those location in the current step. This means that the access pattern shown for row *H=0* as shown in Figure 9 is applied to the rows above one by one, except for the topmost row.

Before the down-sweep phase the root has to be set to 0, in this case it is better to set the complete top row to 0, because this prevents branch divergence as mentioned in Section 4.2. After the up-sweep phase each thread still has write permissions to their own slot on the top level, because of that it can set the complete top row to 0. Then the down-sweep happens on line 84 until line 105. The permissions for this phase are slightly different, but similar in many aspects.

At the end, the result of the algorithm is copied to the output array, this happens on line 108. For this operation there is no need for a barrier or special permissions, since the used permissions for the *temp* array are exactly the ones the thread receives in the last iteration. The permission for the *output* array is already provided to the thread at the start.

6. VERIFICATION

To verify the specification explained in Section 5 the VerCors tool has been used. VerCors translates a program to a Common Object Language, and after that verifies it by using the Silicon backend. More information about the architecture can be found in the paper *The VerCors Tool for Verification of Concurrent Programs* by S. C. C. Blom and M. Huisman [5].

Immediately trying to verify the complete specification in the VerCors tool has a very low chance of getting through, that is why the specification has been done step by step. Each time a certain line of the specification has been added, then verified, if it succeeds the next one will be added, otherwise it has to be refined or rewritten. The specification as described in Section 5 has been verified until the up-sweep phase, which is as shown in Algorithm 4, excluding the ensures predicates of the kernel and thread.

7. RESULTS

In the implementation phase of this research a case of branch divergence in an algorithm of the Prefix Sum has been found. This case of branch divergence has been avoided by changing the storage type of the tree required for the computation to a two-dimensional array. Before this change a part of the GPU threads would be idle in certain iterations of the algorithm, which causes performance problems, after this change all threads execute the same instructions (although some threads do not perform useful calculations). The impact of this change has been benchmarked by measuring the execution time of the kernel. The two-dimensional array storage has an execution time of 120 milliseconds for an input size of 1024 values, instead of 270 milliseconds for the one-dimensional array (see Section 4.2 and Section 4.4 for details). The downside of the alternative storage of the tree is the extra memory

it uses, which makes these two solutions a trade off for speed and memory usage.

The Prefix Sum kernel using the two-dimensional array has been fully specified for detection of data races. These specifications entail read/write permissions for locations in the used arrays, to ensure that threads never have access for writing to the same location, and to ensure that threads do not read from a location that another thread has write permission for. These specifications are written with Permission-Based Separation Logic, which can be used to specify everything required to prove the absence of data races (covered in this paper) and the functional correctness (out of scope for this research).

In order to verify the specification the tool VerCors has been used. This tool translates the specification to an intermediate language from the source language, and after that proves all predicates. Currently the up-sweep phase has been verified by VerCors, consisting of Algorithm 4, excluding the ensures predicates of the kernel and thread. This result ensures that the up-sweep phase of the Prefix Sum algorithm is data race free.

8. FUTURE WORK

In addition to the specification for proving the absence of data races a specification could be made that also entails the functional correctness of the Prefix Sum program. The functional correctness of a program is the other half of showing that the program is completely correct, that is why expanding into that direction would offer a complete specification of the Prefix Sum kernel. The specifications for the functional correctness would need to state the output of the algorithm conform with the description in Section 1.2 and Figure 1.

Apart from verifying the GPU kernel of the Prefix Sum, the code running on the host that launches the kernel could also be specified. This would also allow for verification of the calculations required to run all kernels for the multiple workgroups algorithm. This would specifically be the verification of the calculation of the required *global size* and *local size*, and the splitting of the input array according to these values.

The tool support for verifying GPU kernels could also be improved upon. For example by looking into automatic generation of certain repetitive specifications, like defining the input size of the given specification. Another automatic generation topic could be the boundaries of the thread numbers as specified in the parallel blocks. These numbers never change inside the block, and could therefore be repeated until the end of the block. Apart from these examples there might be more cases where automatic generation of specifications could help, and improve the accessibility of the VerCors tool.

9. REFERENCES

[1] OpenCL the open standard for parallel programming of heterogeneous systems. `https://www.khronos.org/opencl/`. Accessed: 13 March 2015.

[2] The VerCors toolset. `https://fmt.ewi.utwente.nl/redmine/projects/vercors-verifier/wiki`. Accessed: 14 March 2015.

[3] G. E. Blelloch. Scans as primitive parallel operations. *Computers, IEEE Transactions on*, 38(11):1526–1538, 1989.

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

[5] S. C. C. Blom and M. Huisman. The VerCors Tool for Verification of Concurrent Programs. In *Proceedings of the 19th International Symposium on Formal Methods, FM 2014, Singapore*, volume 8442 of *Lecture Notes in Computer Science*, pages 127–131, Berlin, 2014. Springer Verlag.

[6] S. C. C. Blom, M. Huisman, and M. Mihelčić. Specification and verification of GPGPU programs. *Science of Computer Programming*, 95, Part 3(0):376 – 388, 2014. Special Section: ACM SAC-SVT 2013 + Bytecode 2013.

[7] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *ACM SIGPLAN Notices*, volume 40, pages 259–270. ACM, 2005.

[8] 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.

[9] C. A. R. Hoare. An axiomatic basis for computer programming. *ACM*, 12(10):576–580, Oct. 1969.

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

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

[12] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[13] J. Thompson and K. Schlachter. An introduction to the OpenCL programming model. 2012.

[14] T. W. Wiefferink. Prefix sum specification and OpenCL/C++ source. `http://fmt.ewi.utwente.nl/education/bachelor/233`.

Algorithm 4: Part 1 of the kernel specifications: The up-sweep

```
class Ref {
    // Requires/Ensures for the complete kernel
    requires N==32 && H==6; // Using static numbers because the ratio is hard to
                            // use correcly in the tool (H == log2(N)+1)
    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;
            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(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;
                barrier(local) {
                    requires N==32 && H==6;
                    requires t>=0 && t<N;
                    requires level>=1 && level<H;
                    requires Perm(temp[level-1][t], write));

                    ensures N==32 && H==6;
                    ensures level>=1 && level<H;
                    ensures t>=0 && t<N;
                    ensures Perm(temp[level][t], write));
                    ensures t<(N/2) ==> Perm(temp[level-1][t*2], write));
                    ensures t<(N/2) ==> Perm(temp[level-1][t*2+1], write));
                }
                // Do next upsweep step
                if(t < (N/2)) {
                    temp[level][t] = temp[level-1][t*2] + temp[level-1][t*2+1];
                }
            }
```

Algorithm 5: Part 2 of the kernel specifications: The down-sweep

```
74            // Set the root to 0 (does the complete top row to save a barrier/if-statement)
75            temp[H-1][t] = 0;
76            // Down-sweep 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 down-sweep 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 }
```