Master's Thesis

Irregular Segmented Look-Back Scans in Accelerate

Author: Jason van den Hurk (6087590)
Email: j.r.vandenhurk@students.uu.nl
Master: Computing Science
Track: Programming Technology

Daily Supervisor:	First Supervisor:	Second Supervisor:
Ivo Gabe de Wolff MSc	Dr. Trevor McDonell	Prof. dr. Gabriele Keller
i.g.dewolff@uu.nl	t.l.mcdonell@uu.nl	g.k.keller@uu.nl

Department of Information and Computing Science
July 5th, 2022

Utrecht University

Abstract

Scan operations are a common building block in various parallel algorithms. Scan operations are usually significantly faster on GPUs when dealing with large datasets, due to their parallelism. In 2016, D. Merrill and M. Garland released a single-pass decoupled look-back scan, which was significantly faster then the state-of-the art at that time.

We extend this single-pass decoupled look-back scan to support irregular segmented scans, where we scan over a ragged array using a separate array with flags. This is then implemented into the Accelerate framework, a framework for parallel array computations in Haskell. The performance of the algorithm was compared to the original irregular segmented scan implementation in Accelerate. The changed single-pass decoupled look-back performed significantly better than the original when dealing with large datasets.

Contents

1 Introduction

A *scan* operation is a common operation used in many algorithms. It is an operation that takes in an array, some operator, and optionally some sort of initial element and then outputs an array with the operator applied to the elements stepwise: with the output depending on the combination of the preceding elements. For example, an implementation in Haskell would look like the following:

```haskell
scan :: (b -> a -> b) -> b -> [a] -> [b]
scan op initial xs = initial : (case ls of
                                 []   -> []
                                 x:xs -> scan op (op initial x) xs)
```

However, this is only one particular kind of scan. There are instead several kinds of scans, with different use cases depending on the problem being solved.

Even though this function seems to be inherently sequential, if an associative operator is used, scans can be executed by parallel algorithms. These scan operations have been a vital part of several parallel algorithms over the years. Even back in 1989, the potential of this operation was recognized by E. Blelloch, who made it a primitive building block for a multitude of parallel algorithms [3].

When the development of *Graphics Processing Units* (GPUs) became widespread, scan operations again became a key tool for several algorithms. Throughout the years, several scan algorithms were designed for GPUs, which tried to improve the performance of these operations [6].

Many researchers have spent a lot of time and effort into trying to make scan operations fast, as scan operations are important building blocks. In 2016, look-back algorithms were created by D. Merrill and M. Garland, significantly improving performance over other scan implementations [11]. These improvements were achieved by utilizing the memory bandwidth of the GPU better, and by allowing redundant computations to disassociate from global memory latency.

In parallel with the developments in these algorithms, advances were made in the design of frameworks to help programmers program GPUs easier, by providing easy-to-use abstractions of low-level primitives. These developments allowed simpler access to GPUs, which allowed for easier use of these massively parallel structures, and allowed more programmers to use GPUs to optimize their programs. A framework like that is Accelerate, which is an embedded language within Haskell, designed to make parallel algorithms easier to write and execute on the GPU, or multicore CPUs [4].

The look-back algorithm of D. Merrill and M. Garland was a significant improvement for scans as it was generally faster and reduced the effects of memory latency other algorithms had problems with. However, it is limited to only the class of scans we introduced above. The main question this thesis will answer is how to generalize and optimize the look-back scan to support another class of scans: irregular segmented scans.

In summary, the main contributions of the thesis are the following:

- An irregular segmented scan based on the look-back algorithm
- The implementation of such a scan in a GPU framework, such as Accelerate
- Benchmarks assessing the irregular segmented scan and several possible optimizations in a framework such as Accelerate

2 Background

2.1 Parallel computing

While ordinary CPUs are still making significant gains in single-core performance, various predictions exist that the performance of electronics in general will no longer double within fixed costs every two years [17]. This is due to various reasons, such as transistors that will become as small as a few atoms, which are running into physical limits. As such, other techniques to speed up computing have been developing, one of these being *parallel computing.*

Parallel computing can be understood as using multiple computing resources simultaneous to solve some sort of computational problem [1]. This can circumvent the problem that single-core performance doubling is no longer feasible, as you could instead increase the number of CPU cores or use other more specialized architectures. This brings its own set of problems with it, as synchronization between the various processors will be required in various cases. One such specialized architecture take will be explained further are Graphics Processing Units.

2.2 Graphics Processing Units & CUDA

Graphics Processing Units (GPUs) have seen an uptick in use in calculation-heavy computer programs in the last few years. This uptick is partly due to their unique architecture, by allowing many parallel cores to work on a problem at the same time. While a CPU may have somewhere between 2 and 32 cores, a modern GPU can have thousand of data-parallel cores. This allows certain parallel algorithms to run much faster than on a CPU, at the cost of flexibility in the execution. This is due to the fact that data-parallel cores can usually only differ in the data they process, but cannot differ on the current instruction to execute [14]. This limitation is by design: GPUs are designed for tasks related to graphics and excel at that, but that makes it harder to apply to other domains. However, CPUs are general purpose and designed to be as good as possible for all tasks.

2.2.1 Thread hierarchy

There are various ways of programming a GPU, which usually depends on the manufacturer of the graphics card. Graphics cards from the NVIDIA Corporation can be programmed using the CUDA toolkit [7]. Within CUDA, a group of threads that perform the same code is called a *thread group*, while multiple thread groups are combined into a *thread block*, see Figure 1. This hierarchy is similar to other architectures, but with different names.

These thread blocks are scheduled by the scheduler of the GPU and executed on the GPU. Thread blocks should be independent of each other: it should be possible to

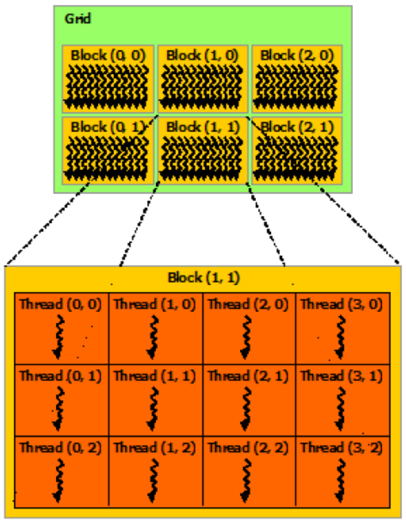

Figure 1: The thread hierarchy of CUDA. Original by the NVIDIA Corporation [13]

execute thread blocks in any order, be it sequential or parallel. Some communication between threads within a thread block is possible, such as some synchronization primitives and the usage of shared memory. Otherwise, only global memory can be used to communicate between threads [13]. The main reasons for this are that it is difficult to communicate efficiently between thread blocks, as well as difficulties in scaling communication to the many thread blocks that can run simultaneously. These thread groups are then grouped into a *grid*, which represents the groups needed to fully process all your inputs.

2.2.2 Memory hierarchy

Within the programming model of CUDA, it is important to understand the memory hierarchy and the distinction between *shared memory* and *global memory*, which is shown in Figure 2. Shared memory is memory that is only available to the current thread block. Due to the memory being very close to the chip executing the code, it is very fast with latency comparable to register access [18].

Global memory doesn't have the limitation that it is only accessible within the current thread block. Instead, every thread on the GPU can access every part of global memory. However, this comes at a trade-off: significantly reduced speed, increased latency and heavier synchronization [13]. As such, efficient algorithms usually try to avoid reading and writing into global memory as much as possible. Instead, they generally copy their section of data into shared memory to perform the various operations needed. After the result is calculated, the data is copied back into global memory from the shared memory. Shared memory can be seen as similar to CPU caches, where keeping as much relevant data in the CPU caches is more efficient than having to resort to the global memory.

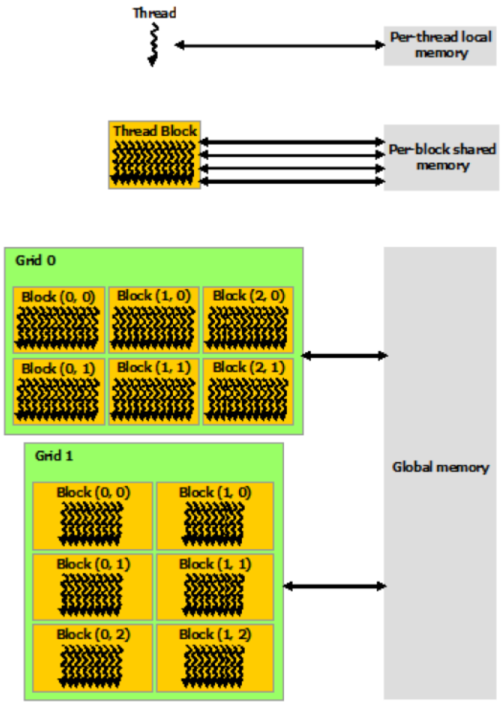

Figure 2: The memory hierarchy of CUDA. Original by the NVIDIA Corporation [13]

2.3 Scans

In 1989, E. Blelloch set the stage for *scan* operations to become a primitive building block for a multitude of parallel algorithms [3], such as the line-of-sight problem and Radix-Sort [2]. The *scan* operator takes a binary associative operator, an initial element, and an input array and returns an array, where the binary associative operator is applied to each element and the result of the operation on the previous elements. For example, given the input $x = [x_0..x_{m-1}]$, an initial element n and some binary associative operator $\oplus$, the result would be

$$[n \oplus x_0,\ n \oplus x_0 \oplus x_1,\ \ldots,\ n \oplus x_0 \oplus \ldots \oplus x_{m-1}] \tag{1}$$

Today, the scan operation is still highly relevant, for example in cooperative allocation within dynamic and irregular data structures [2, 11].

These parallel scan algorithms work in a variety of ways. An example of such an algorithm is reduce-then-scan [12], which is shown in Figure 3. Reduce-then-scan first splits up the elements into blocks. Each of these blocks is then reduced to a single value using the reduce operation. Then, these values are put into a scan operation, which results in an array of intermediate values, representing the value of the scan operation between each block. At last, a scan is done over all blocks, with the corresponding intermediate value added.

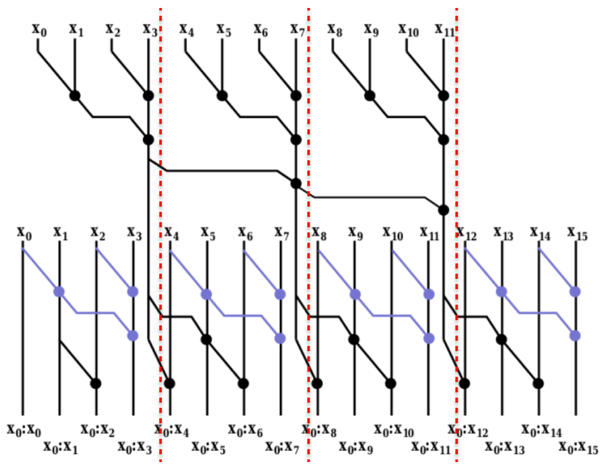
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11}
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15}
$x_0{:}x_0$ $x_0{:}x_1$ $x_0{:}x_2$ $x_0{:}x_3$ $x_0{:}x_4$ $x_0{:}x_5$ $x_0{:}x_6$ $x_0{:}x_7$ $x_0{:}x_8$ $x_0{:}x_9$ $x_0{:}x_{10}$ $x_0{:}x_{11}$ $x_0{:}x_{12}$ $x_0{:}x_{13}$ $x_0{:}x_{14}$ $x_0{:}x_{15}$

Figure 3: An example of a reduce-then-scan execution. Each block is separated by a red dotted line. Figure was edited, original made by D. Merrill and M. Garland [11]

2.3.1 Performance

A key property of a scan operation is the number of *passes* it requires. A pass here is a series of operations which have to finish for all blocks before the algorithm is able to continue. For example, in a reduce-then-scan, the first pass is the reduction of all blocks. Only after all blocks have been reduced, the second pass can start with a scan over those reduced values. Again, only after the scan over the reduced values is completely done, the final scan can be computed to incorporate those reduced values. As such, reduce-then-scan is a 3-pass algorithm. In general having multiple passes is disadvantageous for performance, as blocks cannot progress until other blocks have finished.

For scans, the performance is generally bound by memory access in contrast to the binary operations [11]. As such, reducing the number of reads and writes in global memory is of high importance to make a practical and well-performing algorithm. This is quantified by the amount of data movement: the number of reads and writes in global memory. Several scan algorithms for GPUs have a data-movement of $3n$: 2 global memory reads and a single global memory write per element. For example, the reduce-then-scan has this data-movement of $3n$ [11]. This $3n$ data-movement can be seen in Figure 3, where the first read happens in the reduce phase, and then the second read happens in the scan incorporating the reduced values. At last, the result has to be written back, which results in a total of $3n$ data-movement. $3n$ data-movement is however not the minimum: a scan can be done using only one read and one write per element [19].

2.3.2 Segmented Scans

While scans work on 1-dimensional arrays, we have to define what it means to scan matrices. For example, when dealing with matrices, we could define a scan to perform a 1-dimensional scan for each inner row within the matrix. The implementation then becomes fairly simple: just re-use the existing scan and apply it to each inner row.

However, this method has a couple of downsides. First, if there is any overhead for starting the scan, this is now significantly increased as this overhead is applied for each row in the inner dimension. Secondly, when the inner dimension is small it might mean that not all resources of the processor are utilized. For example, if there are fewer elements than CUDA threads, some threads will not be used, reducing the throughput of the algorithm.

A potential solution for these downsides would be to create a scan that after each inner row 'resets' its value back to the initial element. A matrix could then be represented in memory as a flat array, and a single scan could be started to scan the whole matrix. This is more commonly called a *regular segmented scan.* Regular in this case referring to the regular segment sizes of the scan.

However, it is not required for matrices to have the same number of elements in the inner dimension. Instead, a matrix could be a ragged matrix, with an irregular number of elements in each row. To solve this problem in a single scan, an additional array would be needed to keep track of where these rows start and end. This could be done with a second array with flags, where each truthful value represents the start of a new segment, and such the end of the previous. At each new segment, the scan resets and the scan continues with the initial element [3]. As an example, given the matrix:

$$\begin{bmatrix} x_0 & x_1 & x_2 \\ x_3 & & \\ x_4 & x_5 & \end{bmatrix} \tag{2}$$

the flag array would be $[true, false, false, true, true, false]$. With the flat representation of matrix arrays, the result of this scan would be:

$$[n \oplus x_0, n \oplus x_0 \oplus x_1, n \oplus x_0 \oplus x_1 \oplus x_2, n \oplus x_3, n \oplus x_4, n \oplus x_4 \oplus x_5] \tag{3}$$

which represented in matrix form would be:

$$\begin{bmatrix} n \oplus x_0 & n \oplus x_0 \oplus x_1 & n \oplus x_0 \oplus x_1 \oplus x_2 \\ n \oplus x_3 & & \\ n \oplus x_4 & n \oplus x_4 \oplus x_5 & \end{bmatrix} \tag{4}$$

This is also called an *irregular segmented scan*, and is the class of scans that this thesis will dive further into.

The implementation of segmented scans is varied. Segmented scans can be implemented by changing the operation of a normal scan into a segmented operation which operates on pairs of flags and values [15, 16]. This is called *operator lifting*. First, the flag for a specific element is combined into a tuple of $(flag, element)$. Then, the combine function $\oplus$ is lifted to operator $\oplus^{seg}$ to operate on these flag-value pairs [15, 16]:

$$(f_x, x) \oplus^{seg} (f_y, y) = (f_x | f_y, \text{if } f_y \text{ then } y \text{ else } x \oplus y) \tag{5}$$

and a non-segmented scan is executed with this new operator. This approach has its upsides and downsides. The big upside is that the implementation of segmented scans is fairly easy: the previous scan algorithms can be reused and only the pairing and operator lifting have to be implemented. However, downsides exist as well. Since normal scans are used, there is no possibility for optimizations that possibly exist for segmented scans. It also changes the external interface of scan methods [16], as well as not allowing for certain optimizations. As such, some implementations chose to make segmented scan a separate implementation, so that the operation can stay the same [16].

2.4 Look-back

For non-segmented scans, getting $2n$ global data-movement can improve the performance of parallel scan operations. The chained-scan algorithm is designed for this, however it suffered severely from serial dependencies between adjacent processors [19]. The algorithm was later improved to reduce the serial dependencies between adjacent processors, which became the single-pass parallel prefix scan with decoupled look-back [11].

The algorithm works in a couple of steps, with each of these steps taken by their respective processor. Figure 4 shows a visual representation of these steps. In context of GPUs such as explained in Section 2.2, a "processor" is a thread block. Each thread block then takes care of a certain number of elements, which can vary depending on the number of threads in a block and the number of elements a single thread processes. The steps the algorithm then takes are as follows:

1. Initialization: Initialize various fields in global memory, including the aggregate, inclusive prefix, and the current status.
2. Synchronization: Make sure that each processor has a consistent view of the global memory fields.
3. Compute the aggregate: Calculate the aggregate of the partition that is assigned to the respective processor. Then update the aggregate field in global memory and set the status to "aggregate available"

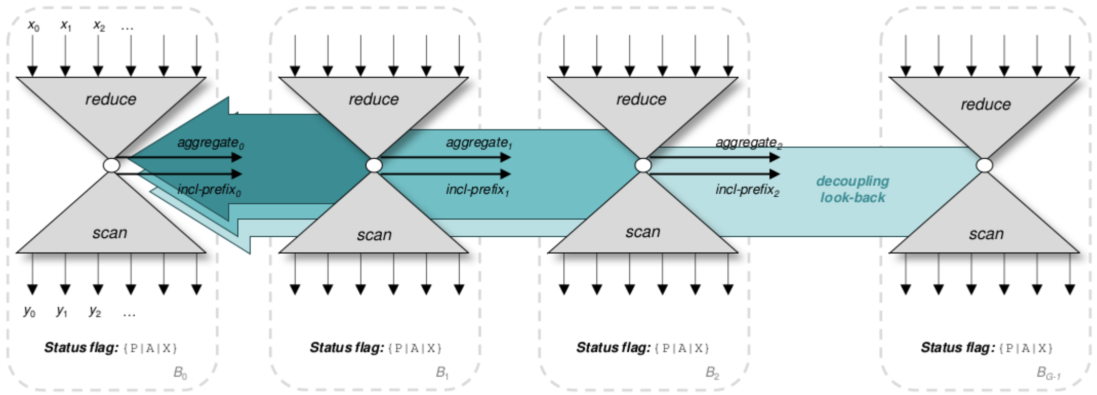

Figure 4: A single-pass parallel prefix scan with look-back. Original from D. Merrill and M. Garland [11]

4. Compute the exclusive prefix: An attempt will be made to calculate the exclusive prefix. Look back at the predecessors. We now have three possible options:
    - The predecessor does not have an aggregate yet. Block until it has an aggregate.
    - There is an aggregate, but not an inclusive prefix. Then add the aggregate to the current exclusive prefix, and repeat these steps for the predecessor.
    - There is an inclusive prefix. Add the inclusive prefix to the current exclusive prefix, and go to the next step
5. Save the exclusive prefix: If we have a complete exclusive prefix, add our aggregate, save it and set the status to “Inclusive prefix available”.
6. Integrate the exclusive prefix: Go though all the elements of the partition and incorporate the exclusive prefix.

This does mean that to algorithm does some redundant computations. A single scan and a reduce operation are performed, and computing the exclusive prefix does redundant work when an aggregate is available but not an inclusive prefix. However, if there is latency from the propagation of the exclusive scan values from a predecessor, this can now be circumvented by using the aggregate and an inclusive prefix of the second predecessor. This has the capability to significantly improve the performance of scan operations. Depending on the use-case, the speedup is between 1.1x and 2.3x [11].

This scan is a single-pass scan, which consequently means that no step described has a dependency on a previous step having to finish for all blocks. Instead, only one or more predecessor blocks have to finish. This can reduce the time waiting for other blocks to finish.

With this scan, there is no need for a separate output array, instead the new elements can be placed back into the original input array [11], also known as having *in-place updates*. This is useful as the choice can be made to not make a separate output array, reducing the memory needed to perform the scan. This is not a unique property of single-pass look-back scans, but it is a useful property.

While this is an advantage, the algorithm does violate the principle of independent thread blocks. For example, if the GPU scheduler schedules the last thread block first, without scheduling the previous blocks, the algorithm would get stuck in an infinite loop while trying to get the prefix of a predecessor. This is circumvented using a technique involving atomics. An atomic counter is kept in global memory that is incremented once when a thread block starts. The counter at that moment is then used to determine what the thread block index is instead of the index from the scheduler. This prevents a thread block being scheduled before their predecessors are scheduled.

With segmented scans, the theoretical minimal global data-movement is $3n$ since the n flags have to be read as well. Current algorithms, such as reduce-then-scan, have a data-movement of $4n$ for segmented versions, which is higher then the theoretical $3n$. As such, improvements can be made here, such as implementing a single-pass look-back algorithm that works on irregular segmented scans with $3n$ data-movement. There already is a version of this algorithm for multi-dimensional scans [5], but none for segmented scans.

2.5 Accelerate

Accelerate is an embedded language in Haskell, which is used to define array computations in Haskell. It is then possible to compile these array computations for multiple architectures, such as GPUs with CUDA, or x64 CPUs. Since Accelerate is a runtime compiler, this compilation is done on-the-fly during execution of the program. This on-the-fly compilation has its up- and downsides. The downside is that the compilation of the program can take some time, which is added to the run-time. This is partly mitigated by caching the programs so that compilation only happens once. The upside is that since compilation happens at run-time, information about the current system is available, which could be used for optimizations [4].

Accelerate is a combinator language, a language that allows creating of a program by combining certain parallel primitive operations. Certain operators within Accelerate are primitives, such as the *map* or the *scan* operation. With these parallel primitives, new parallel functions can be created. For example, a function that calculates the dot product of two vectors can be created as follows, using the *fold* primitive with the prelude function *zipWith*:

```haskell
dotp :: Acc (Vector Float) -> Acc (Vector Float) -> Acc (Scalar Float)
```

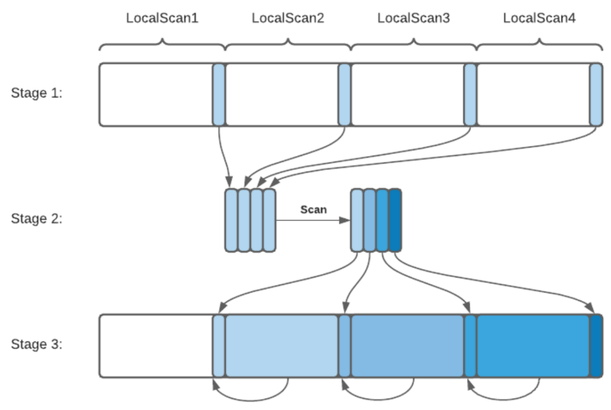

Figure 5: The scan-then-propagate algorithm implemented in Accelerate. Original by M. Clausen [5]

```
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

This function is then optimized by a system of fusion, which combines the underlying primitives to reduce the number of computations, as well as removing intermediate data-structures. This is done to significantly increase performance [9].

To achieve the best possible performance, it is important to make a program which works well with techniques such as SIMD. Accelerate forces the use of flat data-parallelism, and disallows nested, irregular data-parallelism in its type system. This allows Accelerate to statically determine at compile-time that a program cannot be optimized correctly, and reject it during compilation [4].

To actually compile the Accelerate code to the CPU or GPU, Accelerate first compiles the DSL to LLVM IR. This LLVM IR is then compiled and optimized using the LLVM compiler to either native CPU code, or to PTX to run on GPUs. The generation of LLVM IR is type-safe, which makes it easier to write the compilation to LLVM in Accelerate [10].

Accelerate supports various forms of scans in its prelude, including single dimensional scans and multi dimensional scans for both non-segmented as well as irregular segmented scans. However, support for irregular segmented scan is not considered a primitive, but instead implemented using non-segmented scans, as will be explained in Section 2.5.3. An overview of the currently implemented primitive scans are given in Section 2.5.1 and Section 2.5.2.

2.5.1 Single dimension scans

For single dimensions scans, a scan-then-propagate algorithm is used, shown in Figure 5. A scan-then-propagate algorithm works by first splitting up the input into blocks, which are then individually scanned. Then, the last element is placed into a global array, which is then scanned over to get all the aggregates. Lastly, those aggregates are propagated into the blocks, resulting in a fully scanned array. This process has a data-movement of $4n$. First, a single read and write in the first scan, and then another read and write in the subsequent incorporation of the aggregates.

2.5.2 Multi-dimensional scans

Accelerate supports multi-dimensional arrays, which are represented in memory as flat arrays. Currently, normal and segmented scans perform a scan operation per row in the inner-most dimension. However, this scan operation is not done using a scan algorithm that can use multiple thread groups, such as scan-then-propagate, but instead do all the work cooperatively within a single thread group. For example, a scan operation over an array of dimension $2 \times 1.000.000$ will only start two thread groups, as a thread group is started per row in the inner dimension. This can significantly affect performance, as now both thread groups have to deal with 1.000.000 elements. Instead, splitting these 1.000.000 elements over many more thread groups would be faster, since many GPUs can run more than 2 thread groups at the same time. This could significantly improve performance for situations where the inner rows of a matrix are big but there are few of them.

2.5.3 Irregular segmented scans

While the previous scans discussed were implemented in Accelerate as primitives, irregular segmented scans in Accelerate are implemented using normal scans and operator lifting. This makes the implementation easy, but prevents any big optimizations.

As an example of such an optimization, we look at the scan-then-propagate algorithm. In the case where there is a flag in one of the blocks, we only have to incorporate the aggregate into any elements before the first flag. Depending on the implementation, this might save a significant amount of work, for example when the flag is in the first position, as this would allow skipping the incorporation of the aggregate completely.

3 Segmented Look-Back scans

With knowledge of look-back scans, as well as segmented scans, it is now possible to look at the combination of these techniques and what advantages combining them brings. We start out by giving a general overview of using operator lifting, and showing where improvements can be gained with a separate irregular segmented look-back scan. Then, we show the full irregular segmented look-back scan algorithm with all these improvements. At last, we explain the implementation within Accelerate in detail, with some example code.

3.1 Operator lifting

As discussed in Section 2.5.3, operator lifting is one way of implementing a irregular segmented look-back scan. The big upside being that no changes have to be made to the original algorithm, as operator lifting can turn any scan into a irregular segmented scan. We simply lift the operator as discussed before, and every step of the algorithm will now work for irregular segmented scans.

However, when looking at the steps as were discussed in Section 2.4, we can spot some possible improvements versus operator lifting. The first improvement can be spotted when looking at step 3. Assume that some block that is being processed has a segment border anywhere. Since the exclusive and inclusive prefix values of this block will now no longer be influenced by any previous block, we can set the status to “inclusive prefix available” instead of “aggregate available”. This will cause other blocks to have a more complete predecessor available sooner, which should improve performance. This doesn’t happen in the operator lifting approach, since no changes are made to the algorithm.

This advantage can be quantified by looking at the percentage of blocks that have at least a segment border. If more blocks have at least one segment border, more blocks can go to the “inclusive prefix available” status without having to look back, increasing performance.

More improvements can be spotted if we allow changing the algorithm a bit. For example, in step 4 the operator lifting will retrieve both the aggregate element as well as the aggregate flag. However, with a separate implementation you could forgo the loading of the aggregate flag, as if there was a flag the status is set to “inclusive prefix available” as by the previous improvement. As such, any situation where two aggregates are added even if there was a flag in between is avoided. Other improvements include only setting the status to “inclusive prefix available” once, by skipping step 5 if the status is already “inclusive prefix available”, and only incorporating the prefix up until the first flag.

3.2 General overview

Now that the improvements over operator lifting have been identified, the changes can be incorporated into the look-back algorithm. In general, large parts of the single-pass look-back scan algorithm stays the same as was described in Section 2.4. For clarity we will repeat the complete algorithm here, marking all changes in **bold**.

1. Initialization: Initialize various fields in global memory, including the aggregate, inclusive prefix, and the current status.
2. Synchronization: Make sure that each processor has a consistent view of the global memory fields.
3. Compute the aggregate: Calculate the aggregate of the partition that is assigned to the respective processor. Then update the aggregate field in global memory and set the status to "aggregate available" **if there was no flag in this partition. If there was a flag, set the status to "Inclusive prefix available".**
4. Compute the exclusive prefix: An attempt will be made to calculate the exclusive prefix. Look back at the predecessors. We now have three possible options:
   - The predecessor does not have an aggregate yet. Block until it has an aggregate.
   - There is an aggregate, but not an inclusive prefix. Then add the aggregate to the current exclusive prefix, and repeat these steps for the predecessor.
   - There is an inclusive prefix. Add the inclusive prefix to the current exclusive prefix, and go to the next step
5. Save the inclusive prefix: If we have a complete exclusive prefix **and no flag in the partition**, add our aggregate, save it and set the status to "Inclusive prefix available".
6. Integrate the exclusive prefix: Go though the elements of the partition **up to the first flag** and incorporate the exclusive prefix.

The change in step 3 is not only for performance reasons, but is also required to make the algorithm correct. If this change wasn't made, step 4 could add two aggregates together even though there is a segment border in between, which would cause incorrect results. The same applies to step 5: if there was a flag in the partition we already set the inclusive prefix to the correct value. Setting the value to the calculated inclusive prefix is not incorrect but redundant.

The last change is in step 6: we only have to incorporate the exclusive prefix up to the first flag. After the first flag, we reset and any values before it aren't relevant anymore.

Another small optimizations can be found when a new segment starts at the beginning of a block. If this is the case, we can skip the look-back process fully, as we don't require a previous aggregate. However, this is only the case when segments start on the first element of a block, and as such this probably doesn't increase performance by a lot.

Looking at the differences between the two implementations, we hypothesize that the direct implementation should have better performance than the operator lifting implementation, with this performance increase being bigger when the percentage of blocks having at least one segment border is higher. This is further discussed in Section 4.2.1, where the performance of these implementations is discussed.

3.3 Implementation

In order to implement the segmented look-back algorithm within Accelerate, several changes had to be made. These changes include a new primitive to be supported by Accelerate, code generation of this primitive in several different cases, and more. As such, these changes will be discussed in the next several subsections. The full implementation of these changes can be found on Github[1].

3.3.1 Segscan Primitive

As explained in Section 2.5, Accelerate considers several functions primitives, and with these primitives new functions can be created. To allow specific code generation for segmented scans, we start out by adding a datatype for this new primitive as shown in Listing 1.

```haskell
data PreOpenAcc (acc :: Type -> Type -> Type) aenv a where

-- [...]

  SegScan     :: IntegralType i
    -> Direction
    -> Fun              aenv (e -> e -> e)            -- combination function
    -> Maybe       (Exp aenv e)                        -- initial value
    -> acc              aenv (Array (sh, Int) e)      -- array to scan
    -> acc              aenv (Segments i)             -- segment descriptor
    -> PreOpenAcc acc aenv (Array (sh, Int) e)
```

Listing 1: SegScan primitive in Accelerate

In short, to construct a segmented scan, a left-to-right or right-to-left direction is required for the scan, a combinator function, an optional initial value for exclusive

[1] https://github.com/Jasonoro/accelerate-llvm/blob/b43deace9e6cfda7ad9222db96e4a54728b7f5fc/accelerate-llvm-ptx/src/Data/Array/Accelerate/LLVM/PTX/CodeGen/Segscan.hs

segmented scans, the actual array to scan over, and an array of segments which should be of an integral type. With this information, we can construct a new array, of the same shape as the input array.

After construction of this new primitive, all functions that pattern match on the primitives within Accelerate have to be changed to support the new SegScan primitive. As these are usual trivial copies with a few small additions of the already existing Scan primitive, these have been omitted for brevity.

We now come to the actual code generation of Accelerate. Since the code generation of Accelerate can be verbose, due to for example not supporting arrays of structs or pairs, the following parts will be presented in C++ pseudocode. Within the pseudocode the type parameter T will represent the generic type of the input elements. The full code listings can be found in the Appendix.

3.3.2 Getting the block ID

The algorithm starts out by getting a unique, atomically increasing, block ID. This is required as we cannot use the block ID from the GPU scheduler, as there are no guarantees that block n is scheduled before block m, if $n < m$. However, since the algorithm relies on having block n available before block m can finish, this could cause a deadlock if left untouched.

```
1  // arrBlockId is an integer in global memory
2  int bd = blockDim.x;
3  int tid = threadIdx.x;
4  __shared__ int blockIdShared = NULL;
5  int last = bd - 1;
6  if (tid == last) {
7        blockIdShared = atomicAdd(&arrBlockId, 1);
8  }
9  // Sync the threads, as we need the value in shared memory
10 // before we continue
11 __syncthreads();
12 int s0 = blockIdShared;
```

Listing 2: Getting an unique block ID. Accelerate specific code can be found in Listing 8 in the Appendix

The code starts out on line 2 and 3, with grabbing the dimensions of the block as well as the current thread ID. Then, a piece of shared memory is created to hold the atomically increasing block ID. On line 7, the last thread in the block then grabs a pointer to the global integer, and atomically increments it, which returns the value that was in the global integer before the increment. At last, we save this block ID to the integer in shared memory on line 7, so that all other threads can read it at line 12. Some synchronization is required before that, as the thread needs to finish writing to the shared integer before reading can occur, otherwise the value might still be 'NULL'.

3.3.3 Loading data from global memory

To actually perform the scan, data has to be loaded from global memory. This is a relatively straightforward process for scans that are performed left-to-right, but is a bit harder for scans that perform their operations right-to-left. First, the algorithm starts out by calculating the indices that have to be read as well as written to. Then, these indices are loaded from global memory, putting them in a single array.

```
// array_size is an integer representing the length of the flattened array
// global_elements is the array with the elements in global memory
// global_segments is the array with the segments in global memory
const int ELEMENTS_PER_THREAD = 7;
int indexToStartAt = ELEMENTS_PER_THREAD * s0 * bd;
int index;
if (direction == LEFT_TO_RIGHT) {
      index = indexToStartAt + (ELEMENTS_PER_THREAD * tid);
} else {
      index = array_size - indexToStartAt - tid - 1;
}
// Check if this index is within the bounds of the array
if (valid(index)) {
      T* elements = new T[ELEMENTS_PER_THREAD];
      int* segments = new int[ELEMENTS_PER_THREAD];
      for (int i = 0; i < ELEMENTS_PER_THREAD; i++) {
            if (direction == LEFT_TO_RIGHT) {
                  elements[i] = global_elements[index + i];
                  segments[i] = global_segments[index + i];
            }
            else {
                  elements[i] = global_elements[index - i];
                  segments[i] = global_segments[index - i];
            }
      }
      // Rest of algorithm continues here within the if statement...
```

Listing 3: Loading the data from global memory. Accelerate specific code can be found in Listing 9 in the Appendix

3.3.4 Block-level scan & setting global status

With this data of global memory, a block-level scan is performed using SHFL instructions. This block-level scan returns a pair of the scanned input elements, which is different for each thread, but also the index of the first flag found within the scan. This value is needed for two reasons. First, if this value is set we directly set the status to “inclusive status available”. Second, we require this value to determine if a certain thread has to add the previous block aggregate to its elements.

```
// combine is the combination function that is used
// tmpAgg and tmpStatus are global arrays used to communicate
// the aggregates and statuses of the blocks

// The SegscanBlock returns the new element values, and the first flag
// that occurred in this block. The first flag value will be used
// later on to know when we have to stop adding the aggregate
// of the previous block to our elements.
std::pair <T*, int> blockResult =
      SegscanBlock(direction, combine, elements, segments);
bool hadFlag = blockResult.second != INT_MAX;
bool isFirstBlock = s0 == 0;
int status;
// We now have done block-level scans, so now we need to
// incorporate previous block(s) into our application.
// Communication is done over global memory
if (hadFlag || isFirstBlock) {
      status = INCLUSIVE_KNOWN;
}
else {
      status = AGGREGATE_AVAILABLE;
}
// The last thread also writes its result, the aggregate for this thread block,
// and it's corresponding status to the temporary array. This is only
// necessary for full blocks in a multi-block scan; the final
// partially-full tile does not have a successor block.
if (tid == last) {
      T lastElement = blockResult.first[ELEMENTS_PER_THREAD - 1];
      tmpAgg[s0] = std::make_pair(lastElement, lastElement);
      __threadfence_grid();
      tmpStatus[s0] = status;
}
// Wait until all threads are at this point. This sync is not strictly
// necessary, but it forces the GPU to write to the global array first.
// This seemed to improve performance somewhat
__syncthreads();
```

Listing 4: Performing a block level scan and setting the global status. Accelerate specific code can be found in Listing 10 in the Appendix

There are a few important bits here that require some additional explanation. First, the ‘tmpAgg’ and ‘tmpStatus’ arrays. These are arrays within global memory that store the current aggregate as well as the status for all blocks. The status array can have one of three values: either ‘initialized’ when the scan was started but no aggregate is available, ‘aggregate available’ when there is an aggregate available but the previous blocks haven’t been incorporated yet, or ‘inclusive known’ when the full inclusive prefix is known.

Second, the aggregate arrays stores a pair of values of the element: the first element being the aggregate without incorporation of previous blocks and the second being the value with that incorporation. This is to preserve atomicity when changing the status from ‘aggregate available’ to ‘inclusive known’. If this value wasn’t a pair, it could happen that the value would be overwritten before the status is set to ‘inclusive known’, potentially causing incorrect values.

One last point is the use of the “threadfence_grid” synchronization primitive. This function makes sure that all writes before the fence are observed before any writes after it. This prevents any other block from reading an updated status before the aggregate is actually available.

3.3.5 Look-back

With our block-level elements scanned, and this (partial) result published in memory, we can continue with the main part of the algorithm, which is determining what value we have to add to our local elements. This look-back is split into two parts: first the steps to take if a previous block isn’t available and second the complete look-back.

```
__global__ std::pair<T, int> WaitForAvailable(
        int blockId, T aggregate, T(*combine)(T, T)
) {
        while (true) {
                int status = tmpStatus[blockId];
                if (status != SCAN_INITIALIZED) {
                        T aggregateToAdd;
                        if (status == INCLUSIVE_KNOWN) {
                                aggregateToAdd = tmpAgg[s0].second;
                        }
                        else {
                                aggregateToAdd = tmpAgg[s0].first;
                        }
                        T newAggregate;
                        if (aggregate == NULL) {
                                newAggregate = aggregateToAdd;
                        }
                        else {
                                newAggregate = combine(aggregate, aggregateToAdd)
                        }
                        return std::make_pair(newAggregate, status);
                }
        }
}
```

Listing 5: Waiting for a predecessor block to become available. Accelerate specific code can be found in Listing 11 in the Appendix

The first part is a busy-wait function that waits for a specific block to become available, either with an inclusive or exclusive status. If the status of the selected block is still 'initialized', we simply check again until the status changed to either inclusive or exclusive. Once we read that we have a inclusive or exclusive status, we grab that aggregate, add it to the previous aggregate and return that from the function.

There is a small difference between the segmented version of the look-back and the non-segmented one. For the non-segmented version, we don't need to return the previous status from this function. This is because we can always add the inclusive prefix and take that path, but this cannot be done for the segmented version. If we were to do this for the segmented version, we could add an inclusive prefix of a block with a segment, and the look-back would continue to the block before that, which would be incorrect as a flag was passed. As such, we return the previous status so that this info can be used in the main look-back loop.

```
T result;
if (s0 == 0) {
      result = blockResult.first;
}
else {
      bool done = false;
      int currentBlock = s0 - 1;
      T currentAggregate = NULL;
      while(!done) {
            if (__all_sync(aggStatus[currentBlock] == INCLUSIVE_KNOWN)) {
                  done = true;
                  T agg = tmpAgg[currentBlock].second;
                  if (currentAggregate == NULL) {
                        currentAggregate = agg;
                  }
                  else {
                        currentAggregate = combine(currentAggregate, agg);
                  }
            }
            else {
                  std::pair<T, int> res =
                        WaitForAvailable(currentBlock, currentAggregate, combine);
                  currentAggregate = res.first;
                  currentBlock -= 1;
                  if (res.second == INCLUSIVE_KNOWN) {
                        done = true;
                  }
            }
      }
      result = currentAggregate;
}
for (int i = 0; i < ELEMENTS_PER_THREAD; i++) {
      // Only add the aggregate until the first flag in this block
      if (blockRes.second >= i + tid * ELEMENTS_PER_THREAD) {
            blockResult[i] = combine(result, blockResult[i]);
      }
}
```

Listing 6: The main look-back loop. Accelerate specific code can be found in Listing 12 in the Appendix

The second part of the look-back is the main look-back loop. We start out by reading the previous blocks status, and if it's inclusive we add that to our previous aggregate and stop the look-back. Since we initialize the scan with a ‘NULL’ value, we need to make sure this combine function is only called if we do not have a ‘NULL’ value. After we added that aggregate we return from the while loop, so that we can use the found value.

If all threads didn’t have an inclusive value, we instead go into the busy-wait loop explained above. Once this is done, we take that value and add it to the current aggregate. Then, if this was an exclusive value, we continue with the block before. If instead it was an inclusive value, we stop the loop by setting the ‘done’ variable to true.

After finding the look-back value, that value has to be incorporated into our current elements. However, this only needs to happen until our first flag. As such, we make sure to check for this, and only combine the aggregate into our elements when there hasn’t been a flag yet.

3.3.6 Sharing aggregate & writing to global memory

Now that all threads within the block have the correct values, we can share that aggregate with other blocks. We start out by writing the last value of the last thread of the block into global memory, if there was no flag in this block. If there was a flag, we have already written our inclusive value, so writing again would not do anything and so we skip that.

```
if (status == AGGREGATE_AVAILABLE && tid == last) {
      tmpAgg[s0] = std::make_pair(
            blockRes.first, blockResult[ELEMENTS_PER_THREAD - 1]
      );
      __threadfence_grid();
      tmpStatus[s0] = INCLUSIVE_KNOWN;
}
```

Listing 7: Sharing the aggregate. Accelerate specific code can be found in Listing 13

We then write the results of the scan back into global memory, using the same technique as in Section 3.3.3.

4 Experiments and Results

4.1 Experiments

To accurately measure the performance of the constructed segmented scan algorithm, various benchmarks were performed comparing the segmented single-pass look-back scan with the original implementation using operator lifting with a 3-pass scan, and with a single-pass look-back scan using operator lifting. All benchmarks were done on a AMD Ryzen Threadripper 2950X, with an NVIDIA GeForce RTX 2080 Ti, and 64GB of RAM. The OS used was Ubuntu 20.04.2, with CUDA version 10.1 and driver version 470.129.06.

First, we measure the impact of the number of elements to scan on the running time of the various algorithms. This is done by summing the value 1 and having a flag every 10.000 elements. Second, we measure the impact of the number of flags on the various algorithms. This is done with the same program, just with a constant 5.000.000 integers. The segments are then evenly spaced with a certain number of elements between segments, which is varied over the benchmarks in powers of 10.

During the benchmarks 3 'cold-runs' are first done, which constitute running the program without measuring its results. This is done to reduce variance of the running times, for example due to caching behavior. After these cold runs have finished, 10 'hot-runs' are done to measure the running time of the GPU kernel. Only the individual kernel times are measured: copying of the data into GPU memory, or time between kernels is not measured. This is done to try to benchmark these algorithms as accurately as possible, hopefully reducing variance between runs as much as possible. This brings with it a small advantage within the comparison for the original 3-pass scan, as starting up a kernel takes a bit of time which is ignored in the results.

4.1.1 Performance in relation to segment lengths

As hypothesized in Section 3.1, we expect the running time of the algorithm to decrease if the number of blocks that have at least one segment is larger. As such, we do additional measurement on the same test case as above, but with the segments placed so that every nth block has a segment, where n doubles every time. There is one exception, which is the last measurement that has no segments at all. Otherwise, the same benchmarking techniques applies as explained above.

4.2 Results

4.2.1 Segmented scan

First, looking at the general comparison of running times between the several algorithms in Figure 6, it is clear that the specialized 1-pass scan outperforms the

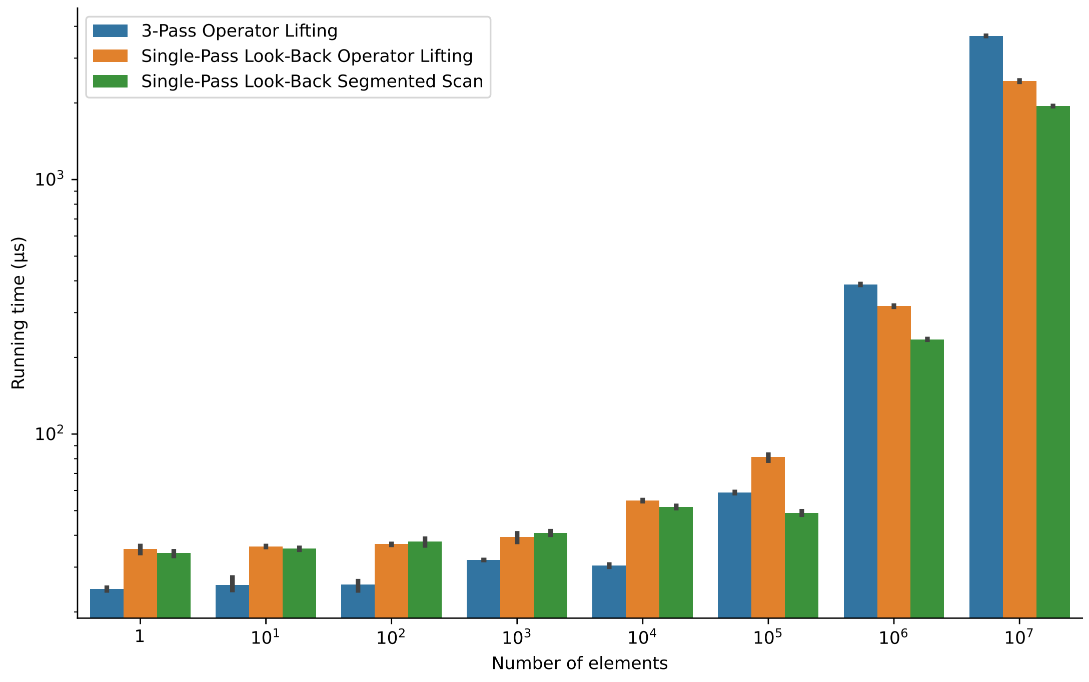

Figure 6: A logarithmic comparison between the original 3-pass scan with lifted operator, the 1-pass with lifted operator and the specialized 1-pass segmented scan, varying the number of elements with constant segment length of 10.000

other algorithms when the number of elements is larger then 10^5. This can also be seen in Figure 7, where the values are normalized taking the original 3-pass scan as baseline.

Continuing to the results in Figure 8, we see that for each variant of segment length, the single-pass look-back segmented scan is the best performing algorithm when there are 5.000.000 elements, but that the running time does change. The original 3-pass scan with operator lifting performs the worst, with an running time of at least double that of the single-pass segmented look-back when there are 5.000.000 elements.

Another interesting observation is that while the operator lifting algorithms have a fairly constant execution time over the different segment lengths, the single-pass segmented look-back seems to have a lower execution time with smaller segment lengths. We show this difference in more detail in Figure 9, where it can clearly be seen that the more blocks have a segment, the quicker the algorithm is.

This confirms our hypothesis as described in Section 3.1. There is however one 'strange' result, which is that a flag every block is slower then a flag every two blocks. Why this happens is unclear and would have to be further researched.

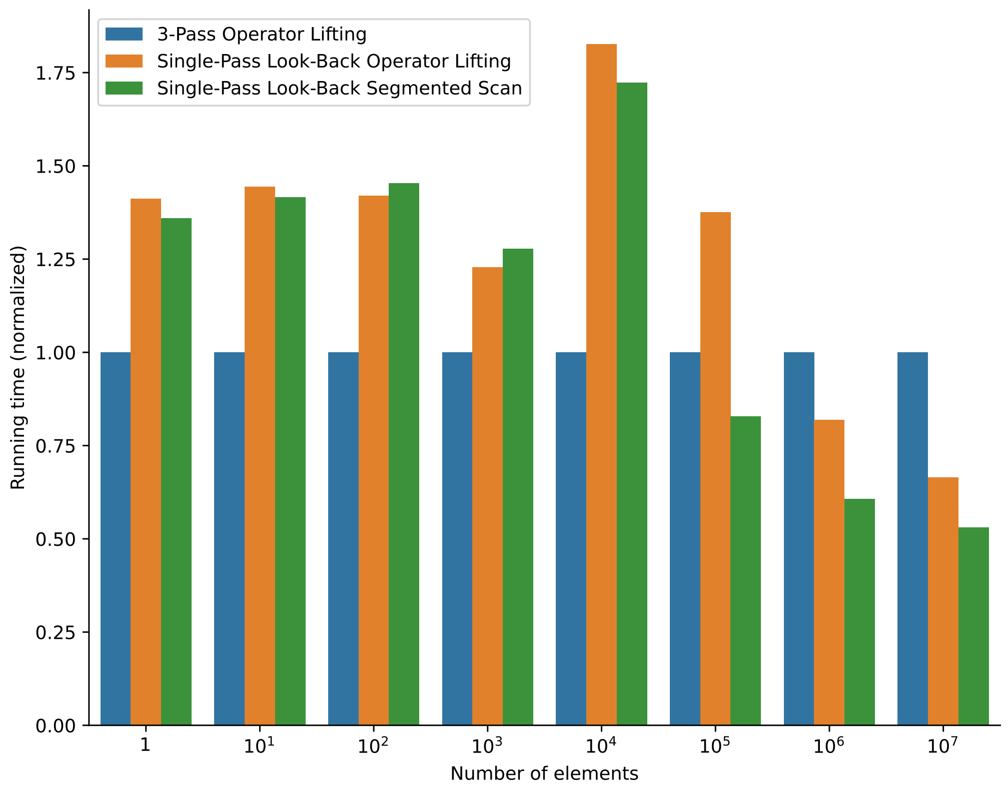

Figure 7: A normalized comparison between the original 3-pass scan with lifted operator, the 1-pass with lifted operator and the specialized 1-pass segmented scan, varying the number of elements with constant segment length of 10.000

4.2.2 Normal scan

Since a segmented look-back scan has been implemented, it was trivial to convert this implementation to a normal look-back scan such as described in the original paper by D. Merrill and M. Garland [11]. The implementation was benchmarked against the 3-pass scan that was already present within Accelerate. With 5.000.000 elements, this resulted in a reduction of running time from 1222 μs to 644 μs. This is a significant improvement for a large number of elements when compared to the current implementation.

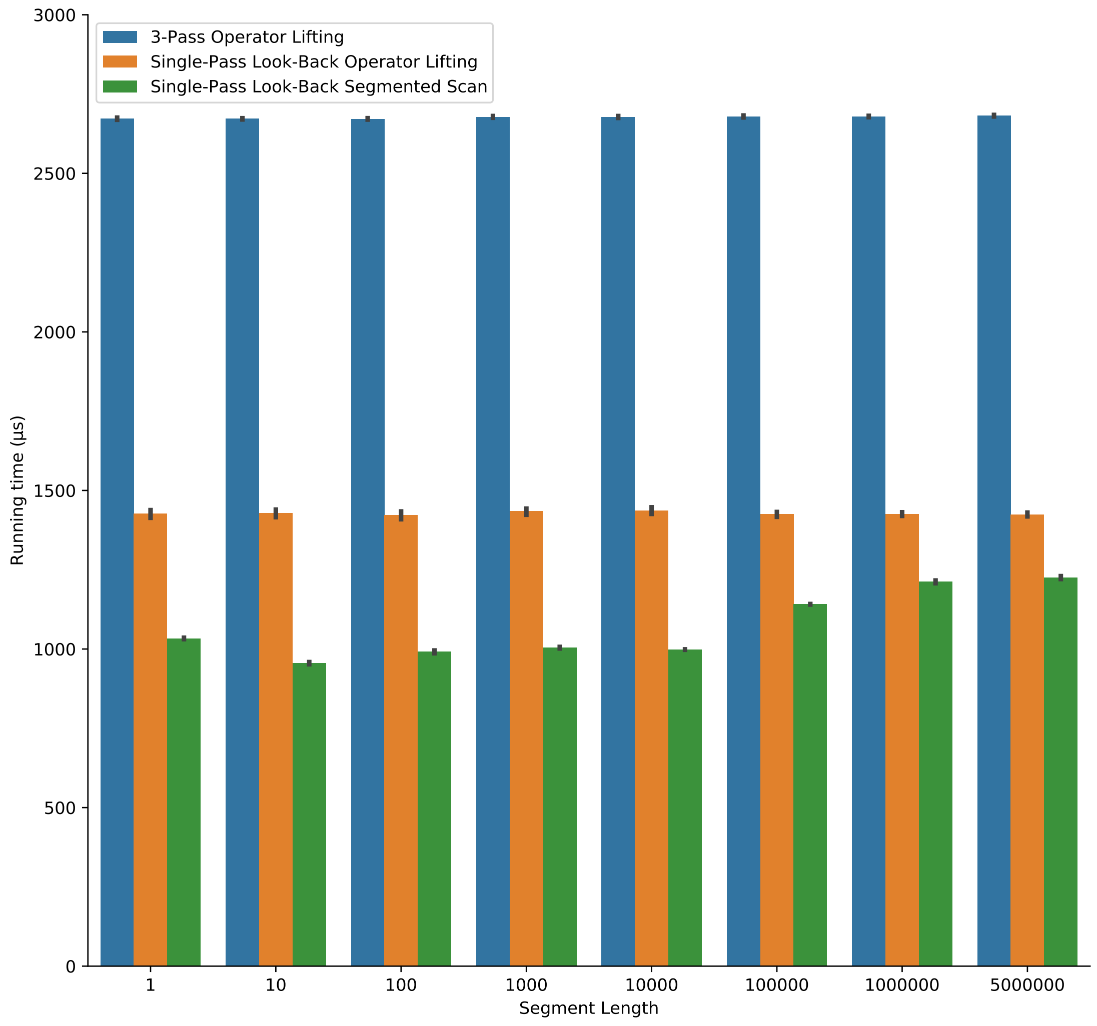

Figure 8: A comparison between the original 3-pass scan with lifted operator, the 1-pass with lifted operator and the specialized 1-pass segmented scan, varying the segment lengths with a constant 5.000.000 elements

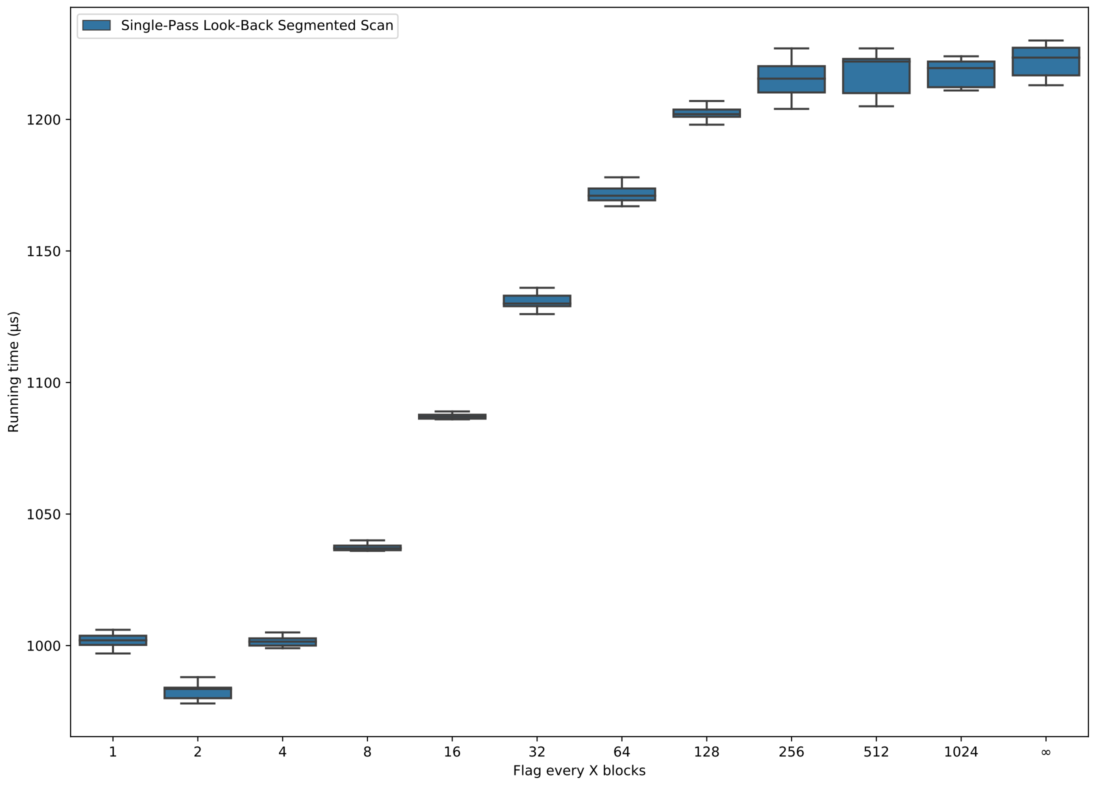

Figure 9: A comparison of running time when varying the number of blocks with a segment with a constant 5.000.000 elements

5 Future work

While the results of the segmented look-back are good, several improvements can still be made and others can be researched. Below we will discuss a couple of improvements that could be implemented and researched, and what their possible impact would be.

5.1 Memory coalescing

When interacting with global memory, it is important for performance to read and write to memory coalesced to improve the performance of these operations [8, 13]. As seen in Section 3.3.3 and Section 3.3.6, global memory reads and writes are currently not coalesced, which probably causes decreased performance. However, since every thread needs to have sequential elements available, some additional shuffling of the elements over either shared memory or using SHFL instructions would be needed to transpose the input so that all threads get their needed elements. This causes some additional overhead, so benchmarking would be required to look whether this decreases overall running time.

5.2 Optimal elements per thread

In the current implementation, every thread combines 7 elements locally, before sharing this information in a block-wide scan. However, the optimal number of elements will depend on factors such as the available registers of the GPU, the speed of global memory, the speed of individual threads, and much more. As such, researching what the optimal number of elements is for certain GPUs and workloads could help improve the performance of this algorithm. Since Accelerate compiles the CUDA kernels at run-time, it is able to use the information about the available GPU to optimize this variable at compile time.

5.3 Fast tracking inclusive values

As discussed in Section 4.2.1, when there are more blocks with segments, the running time of the segmented single-pass algorithm is lower. This happens when a previous block can fast-track setting the inclusive value due to not having to look-back first. However, it might be possible to do a pass before the actual look-back algorithm, finding the blocks that have at least one segment and publishing those inclusive results. After this the normal executing of the algorithm would resume.

Another fast-track option would be to see if we could detect that a block has a segment before the full block scan is done. For example, if there is a flag 10 elements before the end, we would only have to scan the last 10 elements and no intra-warp communication would be needed before we could publish the inclusive prefix.

However, this might cause additional overhead and extensive benchmarking would be required to see whether this would improve performance.

5.4 Flag packing

Currently, flags are represented by 32-bit integers, with the smallest supported possibility being 8-bits for a simple true of false value. Instead, we could pack 8 flags into a single 8-bit value, significantly reducing the memory footprint needed for the flags as well as reducing the amount and size of the reads from global memory. However, packing flags will have a negative effect on performance if memory coalescing is implemented, so this tradeoff would have to be considered. At the moment this kind of packing is not supported within Accelerate, and the current type requirements for arrays of segments prevent doing this. As such, to make these improvements several changes would have to be made to Accelerate to fully support flag packing.

5.5 Fusion

Accelerate has a concept of *fusion*, which is a system that can combine primitives into a single piece of code. For example, if there is a combination of primitives such as `map g . fold f z`, the `map` operation can be fused into the `fold` by applying `g` before the reduction in the fold is performed [9]. This is useful as it reduces the amount of GPU kernels that have to run, as well as not needing an array to save the result of the map.

Previously, fusion could only be performed on the input of scans, as the scan operation takes several passes which complicates fusion. However with the change into a single-pass scan there might now be more possibilities for fusion. However, where to fuse within the algorithm and how this can be done as efficiently as possible is still an open problem.

6 Conclusion

A number of contributions were made in this thesis, in relation to irregular segmented scans. First and foremost, a segmented look-back scan was created based on the original algorithm by D. Merrill and M. Garland. The created algorithm has the necessary changes to support irregular segmented scans, which is the first look-back scan to do so as far as the author is aware.

Secondly, an actual implementation was made within Accelerate. This implementation was used for benchmarking, and can be used by the users of Accelerate as well. The benchmarking was explained in Section 4.2 which showed various positive results. First, the created algorithm was significantly faster when there are at least 10^5 elements, and the created algorithm was faster than simply using operator lifting on the original algorithm. The speedup was over 50% in the best-case scenario. Comparisons were also made regarding the effect of segment lengths onto the running time of the algorithm, showing that the created algorithms running time is dependent on the number of blocks that have at least one segment. The more blocks that have at least one segment, the lower the running time.

Combining everything, the constructed algorithm improved the running time on large input sets significantly, with only a small impact on overhead when there are few elements.

References

[1] Blaise Barney et al. “Introduction to parallel computing”. In: *Lawrence Livermore National Laboratory* 6.13 (2010), p. 10.

[2] G.E. Blelloch. *Prefix sums and their applications.* Tech. rep. School of Computer Science, Carnegie Mellon University, 1990.

[3] G.E. Blelloch. “Scans as primitive parallel operations”. In: *IEEE Transactions on Computers* 38.11 (1989), pp. 1526–1538. DOI: `10.1109/12.42122`.

[4] Manuel M.T. Chakravarty et al. “Accelerating Haskell Array Codes with Multicore GPUs”. In: *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming.* DAMP ’11. Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 3–14. ISBN: 9781450304863. DOI: `10.1145/1926354.1926358`. URL: `https://doi.org/10.1145/1926354.1926358`.

[5] Morten Clausen. “Regular Segmented Single-pass Scan in Futhark”. MA thesis. University of Copenhagen, 2021.

[6] Yuri Dotsenko et al. “Fast scan algorithms on graphics processors”. In: *Proceedings of the 22nd annual international conference on Supercomputing.* 2008, pp. 205–213.

[7] Jayshree Ghorpade. “GPGPU Processing in CUDA Architecture”. In: *Advanced Computing: An International Journal* 3.1 (Jan. 2012), pp. 105–120. ISSN: 2229-726X. DOI: `10.5121/acij.2012.3109`. URL: `http://dx.doi.org/10.5121/acij.2012.3109`.

[8] Mark Harris. *How to Access Global Memory Efficiently in CUDA C/C++ Kernels.* 2013. URL: `https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/` (visited on 05/06/2022).

[9] Trevor L. McDonell et al. “Optimising Purely Functional GPU Programs”. In: *SIGPLAN Not.* 48.9 (Sept. 2013), pp. 49–60. ISSN: 0362-1340. DOI: `10.1145/2544174.2500595`. URL: `https://doi.org/10.1145/2544174.2500595`.

[10] Trevor L. McDonell et al. “Type-Safe Runtime Code Generation: Accelerate to LLVM”. In: *SIGPLAN Not.* 50.12 (Aug. 2015), pp. 201–212. ISSN: 0362-1340. DOI: `10.1145/2887747.2804313`. URL: `https://doi.org/10.1145/2887747.2804313`.

[11] Duane Merrill and Michael Garland. “Single-pass parallel prefix scan with decoupled look-back”. In: *NVIDIA, Tech. Rep. NVR-2016-002* (2016).

[12] Duane Merrill and Andrew Grimshaw. “Parallel scan for stream architectures”. In: (2009).

[13] Nvidia Corporation. *Programming Guide :: CUDA Toolkit Documentation.* 2022. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html` (visited on 01/04/2022).

[14] John D Owens et al. “GPU computing”. In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899.

[15] Jacob T. Schwartz. "Ultracomputers". In: *ACM Trans. Program. Lang. Syst.* 2.4 (Oct. 1980), pp. 484–521. ISSN: 0164-0925. DOI: `10.1145/357114.357116`. URL: `https://doi.org/10.1145/357114.357116`.

[16] Shubhabrata Sengupta, Mark Harris, Michael Garland, et al. "Efficient parallel scan algorithms for GPUs". In: *NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003* 1.1 (2008), pp. 1–17.

[17] John Shalf. "The future of computing beyond Moore's law". In: *Philosophical Transactions of the Royal Society A* 378.2166 (2020), p. 20190061.

[18] Henry Wong et al. "Demystifying GPU microarchitecture through microbenchmarking". In: *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE. 2010, pp. 235–246.

[19] Shengen Yan, Guoping Long, and Yunquan Zhang. "StreamScan: Fast Scan Algorithms for GPUs without Global Barrier Synchronization". In: *SIGPLAN Not.* 48.8 (Feb. 2013), pp. 229–238. ISSN: 0362-1340. DOI: `10.1145/2517327.2442539`. URL: `https://doi.org/10.1145/2517327.2442539`.

Appendix

Getting the block ID

```haskell
-- arrBlockId is an integer array in global memory with size 1
bd <- blockDim
tid <- threadIdx
-- Create a piece of shared memory to save the incremental block id
-- The reason for not just using the CUDA block id is that there are
-- no guarantees that block n is scheduled before m
-- when n < m. This could cause a deadlock since we rely on having
-- the value of block n before we can finish processing m
blockIdShared <- dynamicSharedMem (TupRsingle scalarTypeInt) TypeInt
                                (liftInt 1) (liftInt 0)
last <- A.sub numType bd (liftInt32 1)
when (A.eq singleType tid last) $ do
    bIdPtr <- instr' $ GetElementPtr (asPtr defaultAddrSpace
                        (op integralType (irArrayData arrBlockId)))
                        [op integralType (liftInt 0)]
    -- Increment the counter atomically using the last thread
    bidT    <- instr' $ AtomicRMW (IntegralNumType TypeInt)
                         NonVolatile RMW.Add bIdPtr
                         (op integralType (liftInt 1))
                         (CrossThread, AcquireRelease)
    -- Save it to the shared memory so all blocks have access to it
    writeArray TypeInt blockIdShared (liftInt 0) (ir integralType bidT)

-- Sync the threads, as we need the value in shared memory
-- before we continue
__syncthreads
-- Grab the block id from shared memory
s0  <- readArray TypeInt blockIdShared (liftInt 0)
```

Listing 8: Getting an unique block ID

Loading data from global memory

```haskell
      -- Index this thread block starts at
      inf <- A.mul numType (liftInt elementsPerThread) =<< A.mul numType s0 bd'

      -- index i0 is the index this thread will read from
      i0 <- case dir of
        LeftToRight -> A.add numType inf =<< A.mul numType (liftInt elementsPerThre
        RightToLeft -> do x <- A.sub numType sz inf
                          y <- A.sub numType x =<< A.mul numType (liftInt elementsP
                          z <- A.sub numType y (liftInt 1)
                          return z

      -- index j* is the index that we write to. Recall that for exclusive scans
      -- the output array is one larger than the input; the initial element will
      -- be written into this spot by thread 0 of the first thread block.
      j0    <- case mseed of
                  Nothing -> return i0
                  Just _  -> case dir of
                               LeftToRight -> A.add numType i0 (liftInt 1)
                               RightToLeft -> return i0

      -- If this thread has input, read data and participate in thread-block scan
      let valid i = case dir of
                      LeftToRight -> A.lt  singleType i sz
                      RightToLeft -> A.gte singleType i (liftInt 0)


      when (valid i0) $ do
        let indexFunc = \x y -> case dir of
                          LeftToRight -> A.add numType x (liftInt y)
                          RightToLeft -> A.sub numType x (liftInt y)
        x0 <- app1 (delayedLinearIndex arrIn) =<< (indexFunc i0 0)
        x1 <- app1 (delayedLinearIndex arrIn) =<< (indexFunc i0 1)
        x2 <- app1 (delayedLinearIndex arrIn) =<< (indexFunc i0 2)
        x3 <- app1 (delayedLinearIndex arrIn) =<< (indexFunc i0 3)
        x4 <- app1 (delayedLinearIndex arrIn) =<< (indexFunc i0 4)
        x5 <- app1 (delayedLinearIndex arrIn) =<< (indexFunc i0 5)
        x6 <- app1 (delayedLinearIndex arrIn) =<< (indexFunc i0 6)
```

Listing 9: Loading the data from global memory

Block level scan & setting global status

```haskell
-- The segscanBlock returns the new element values, and the first flag
-- that occured in this block. The first flag value will be used
-- later on to know when we have to stop adding the aggregate
-- of the previous block to our elements.
-- The result of this is a pair of (values, firstFlag)
blockRes <- if ((typeToTuple tp) `TupRpair` TupRsingle scalarTypeInt32,
              A.gte singleType n bd')
    -- Arguments: Direction, Device properties, Type of scanned element
    -- Typeproof that segments are integer, combine function, elements left
    -- Elements to scan, segments to scan with
    then segscanBlock dir dev tp intTy combine Nothing   resSeeded resSegments
    else segscanBlock dir dev tp intTy combine (Just n') resSeeded resSegments

-- We now have done block-level scans, so now we need to
-- incorportate previous block(s) into our application.
-- Communication is done over global memory
let hadFlag = A.neq singleType (A.snd blockRes) (liftInt32 maxBound)
let isFirstBlock = A.eq singleType s0 (liftInt 0)
status <- if (TupRsingle scalarTypeInt32, hadFlag `lor'` isFirstBlock)
          then return (liftInt32 scan_inclusive_known)
          else return (liftInt32 scan_exclusive_known)

-- The last thread also writes its result, the aggregate for this thread block,
-- and it's corresponding status---to the temporary array. This is only
-- necessary for full blocks in a multi-block scan; the final
-- partially-full tile does not have a successor block.
last <- A.sub numType bd (liftInt32 1)
when (A.eq singleType tid last) $ do
    -- Write the final tuple element to the global memory array
    writeArray TypeInt arrTmpAgg s0
          (A.pair (last $ A.fst blockRes) (last $ A.fst blockRes))
    __threadfence_grid
    -- And write the status to another global memory array
    writeArray TypeInt arrTmpStatus s0 status

-- Wait until all threads are at this point. This sync is not strictly
-- necessary, but it forces the GPU to write to the global array first.
-- This seemed to improve performance somewhat
__syncthreads
```

Listing 10: Performing a block-level scan and sharing this aggregate

Look-back - Wait for available

```haskell
-- The block we start looking back to, which would be the (blockId - 1)
lookAtBlock <- A.sub numType s0 (liftInt 1)
let waitForAvailable = while (whileTp `TupRpair` TupRsingle scalarTypeInt32)
(\(unPair4 -> (done, _, _, _)) -> A.eq singleType done (liftInt32 0))
(\(unPair4 -> (done,blockId,agg,_)) -> do
  previousStatus <- readArray TypeInt arrTmpStatus blockId
  let statusIsInit = A.eq singleType previousStatus (liftInt32 scan_initialized)
  if (whileTp `TupRpair` TupRsingle scalarTypeInt32, statusIsInit)
  then do
    return $ pair4 done blockId agg (liftInt32 0)
  else do
    -- In the normal single-pass look-back scan, we do not have to check
    -- whether or not the status is inclusive or exclusive, since we
    -- can always just use the combine function. This is a bit different
    -- with this segmented version: if we do not check wheter the status
    -- is inclusive or exclusive, we might add an inclusive prefix that
    -- actually had an flag in it, which wouldn't be good
    previousAggs <- readArray TypeInt arrTmpAgg blockId
    let (prevAgg, _) = A.unpair previousAggs
    newAggregate <- if (tp, A.eq singleType blockId lookAtBlock)
      then return prevAgg
      else case dir of
        LeftToRight -> app2 combine prevAgg agg
        RightToLeft -> app2 combine agg prevAgg
    return $ pair4 (liftInt32 1) blockId newAggregate previousStatus
)
```

Listing 11: Wait for a predecessor block to become available

Look-back - Main loop

```haskell
res <- if (typeToTuple tp, A.gt singleType s0 (liftInt 0))
then do
r <- while (whileTp)
  (\(unPair3 -> (done,_,_))          -> A.eq singleType done (liftInt32 0))
  (\(unPair3 -> (done,blockId,agg)) -> do
    __threadfence_block
    previousStatus <- readArray TypeInt arrTmpStatus blockId
    statusIsInclusive <- A.eq singleType previousStatus
                            (liftInt32 scan_inclusive_known)
    let allThreadsInclusvie = __all_sync (liftWord32 maxBound)
                                          statusIsInclusive
    if (whileTp, allThreadsInclusvie)
      then do
      previousAggs <- readArray TypeInt arrTmpAgg blockId
      let (prevAgg, inclusivePrefix) = A.unpair previousAggs
      newAggregate <- if (tp, A.eq singleType blockId lookAtBlock)
        then return inclusivePrefix
        else case dir of
          LeftToRight -> app2 combine inclusivePrefix agg
          RightToLeft -> app2 combine agg inclusivePrefix
      return $ pair3 (liftInt32 1) blockId newAggregate
      else do
      if (whileTp, A.gt singleType blockId (liftInt 0))
      then do
        waitR <- waitForAvailable $ pair4 done blockId agg (liftInt32 0)
        let (_, _, newAggregate, statusFound) = unPair4 waitR
        prevBlock <- A.sub numType blockId (liftInt 1)
        newDone <- if (TupRsingle scalarTypeInt32,
             A.eq singleType statusFound (liftInt32 scan_inclusive_known))
        then return $ liftInt32 1
        else return $ liftInt32 0
        return $ pair3 newDone prevBlock newAggregate
      else return $ pair3 done lookAtBlock (undefT tp)
  )
  (pair3 (liftInt32 0) lookAtBlock (undefT tp))
-- We're done, so unpack the value and find out what aggregate we have to add
let (_, _, aggregateToAdd) = unPair3 r
-- Add the aggregate until we hit the first flag
let combine' = \x index ->
  if (tp `TupRpair` TupRsingle scalarTypeInt,
        flip (A.lt singleType) (A.snd blockRes) =<< A.add numType index =<<
```

```
          A.mul numType tid (liftInt32 $ P.fromIntegral elementsPerThread))
  then do
    res <- case dir of
    LeftToRight -> app2 combine aggregateToAdd x
    RightToLeft -> app2 combine x aggregateToAdd
    return $ A.pair res (liftInt 0)
  else do
    return $ A.pair x (liftInt 1)
applyToTupleZip combine' tp (A.fst blockRes) $ pairGrouped
(liftInt32 0, liftInt32 1, liftInt32 2, liftInt32 3,
 liftInt32 4, liftInt32 5, liftInt32 6)
else return $ A.fst blockRes
```

Listing 12: Main look-back loop to get an inclusive prefix

Sharing aggregate

```haskell
-- If we only set an exclusive status, we now need to set the inclusive status.
when (A.eq singleType status (liftInt32 scan_exclusive_known)) $
when (A.eq singleType tid last) $ do
  writeArray TypeInt arrTmpAgg s0
    (A.pair (lastTuple $ A.fst blockRes) (lastTuple res))
  __threadfence_grid
  writeArray TypeInt arrTmpStatus s0 (liftInt32 scan_inclusive_known)
```

Listing 13: Share the computed aggregate in global memory

Writing to global memory

```haskell
let (res1, res2, res3, res4, res5, res6, res7) = unpairGrouped res
let writeArray' = flip $ writeArray TypeInt arrOut
writeArray' res1 =<< (indexFunc j0 0)
writeArray' res2 =<< (indexFunc j0 1)
writeArray' res3 =<< (indexFunc j0 2)
writeArray' res4 =<< (indexFunc j0 3)
writeArray' res5 =<< (indexFunc j0 4)
writeArray' res6 =<< (indexFunc j0 5)
writeArray' res7 =<< (indexFunc j0 6)
```

Listing 14: Writing the result of the scan back to global memory