Implementing Single-Pass Scan in the Futhark Compiler

Andreas Nicolaisen, jtc303
Marco Aslak Persson, bfr555

Abstract

This report presents an algorithm for the single pass parallel scan, and its implementation in the Futhark compiler. The algorithm is based on the one presented by Duane Merrill and Michael Garland in the article "Single-pass Parallel Prefix Scan with Decoupled Look-back"[8]. The algorithm is first introduced at a high level, to give an intuition for how it works, and the reasoning behind its structure. Then we present how this algorithm is implemented inside the Futhark compiler, and report an initial evaluation of its performance. Finally, we discuss several possible improvements that can benefit our implementation.

Contents

1 Introduction

The scan operator, also known as parallel-prefix sum, is a basic block of data-parallel programming, as it appears in the implementation of important algorithms such as radix sort, numeric recurrences, e.g., $x_n = a_n * x_{n-1} + b_n$, solvers of partial differential equations [1], financial algorithms aimed at option pricing [9] and graphics algorithms [4]. Furthermore, scan is required by analysis that is aimed at enhancing the degree of application parallelism that is mapped to hardware [3].

Scan has the following signature:

$$(\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to [\alpha] \tag{1}$$

The first part of the signature is a binary associative operator, the second is the neutral element of that operator, and the third is an input array of the same element type α. The result is an array of the same element type and length as the input array. Given the input list $x = [x_0, \ldots, x_{n-1}]$, a neutral element ne and the operator $\oplus$, the result of the (inclusive) scan is semantically the following:

$$[ne \oplus x_0,\ \ ne \oplus x_0 \oplus x_1,\ \ ne \oplus x_0 \oplus x_1 \oplus x_2,\ \ldots,\ ne \oplus x_0 \oplus \ldots \oplus x_{n-1}] \tag{2}$$

It is important to mention that parallel implementation of scan are work efficient, i.e., the total number of operations performed by all cores is $O(n)$, and even when an infinite number of cores are available, the parallel implementation of scan requires $O(log\ n)$ sequential steps to complete, i.e., depth complexity is $O(log\ n)$.

While the theoretical importance of the scan operator was well understood since the eighties [2], the scan construct was conspicuously missing from the multi-core parallel implementation of large benchmark suites such as Spec2006, and it was not accounted for in compiler analysis—ranging from entirely dynamic [10] to mostly static [12, 11]—aimed at automatic parallelization of legacy sequential code. This is because, when the number of cores is low, one can use simpler, ad-hoc implementations of scan, for example in which each core can scan sequentially (and independent of the others) its chunk of elements, and the difficult part that requires inter-thread communication can be also performed sequentially by one thread, without affecting performance (because the number of cores is low, typically less than 64).

It was only with the advent of highly-parallel commodity hardware such as GPGPUs—whose full utilization requires tens-to-hundred of thousands of hardware threads—that the practical importance of scan as a basic-block of parallel programming was finally demonstrated, because the ad-hoc solutions such as the one mentioned above do not scale well when the number of cores increases. Since scan patterns are not supported by loop-based dependence analysis, and scan patterns are difficult to recognize from sequential implementations, it has become important to support scan as a primitive construct at language level, thus enabling (i) high-level optimizations by means of re-write rules such as fusion, and (ii) efficient code generation for highly-parallel hardware (which are tedious and non-trivial to achieve).

In this report, we present our implementation of a single-pass-scan algorithm, in the Futhark compiler [6]. We will first explain the algorithm at a high level, by breaking it into smaller parts and examining them individually, and explain how they fit together. Then we will go into how it was implemented into the Futhark compiler, and explain how different parts of

the compiler implementation relates to the generated OpenCL code. We'll then present our benchmark results where we achieve a speedup from 11572μs to 4750μs in comparison to the previous Futhark implementation, on a scan with plus as the operator on a 1GiB input. We within get 84.8% of the performence of memcpy on the same input. We will finally discuss the limitations of the algorithm and implementation, and suggest possible improvements.

2 Algorithm - Motivation & High level

2.1 Motivation

Typical GPU implementations of scan use three kernels, for example:

(1) Each block reduces its chunk of consecutive elements and writes its result into a smallish array;

(2) The smallish array is scanned;

(3) Each block scans its chunks of consecutive elements, to which it also adds the prefix computed in step (2).

This approach requires at least three accesses to global memory for each of the to-be-scanned elements: one read from global memory in step (1) and one read and one write from global memory in step (3). Matters are even worse in the current Futhark compiler implementation of scan that requires four global-memory accesses per scanned element: the first phase performs a partial scan of the elements that are processed by a block, and the last stage adds the prefix to each element. This approach is taken because Futhark IR supports a construct that represents the fusion of a map operation with a scan, which is difficult to be supported by the three-access approach without requiring redundant computation of the map operation.

The implementation presented in this report requires only two global-memory accesses per scanned element: if we assume for simplicity that the scanned element is of size one word, then the goal is that our scan should have performance comparable to that of `memcopy`.[1]

[1] The memcopy operation is an absolute upper-bound for performance because any implementation of scan requires at least two global memory accesses and some arithmetic computation.

2.2 High level

Our implementation is based on the algorithm presented by Duane Merrill and Michael Garland[8]. The main idea is to implement the scan within one kernel by communicating the prefixes across blocks in a pipelined fashion, whose efficiency is improved by some (parallel) redundant computation. The main stages of the algorithm are:

(1) A dynamically renaming scheme (based on atomic increment) is applied to blocks to ensure that for any $n < q$, if the block (re)named q is running then block n is already under execution as well. The dynamic renaming prevents deadlocks, which are possible with the original block numbering.[2]

(2) Chunks of M elements per thread are read in coalesced way from global to register memory, where they transformed using the map function. They are then put into correct order, by using the shared memory as a transposition buffer;

(3) Each thread scans sequentially its M elements;

(4) A block level scan processes the last elements of each thread;

(5) The block prefix is published in global memory (i) with status-flag `P`, i.e., full prefix, for the first block, and (ii) with status flag `A`, i.e., partial prefix, for the other blocks,

(6) An inter-block communication scheme is employed to compute the prefix of the blocks preceeding the current one (in program order).

(7) The prefix of the previous blocks and of the previous threads in the current block are added to the per-thread scanned elements;

(8) The fully-scanned elements are copied in coalesced fashion to global memory, using as before shared memory as a transposition buffer.

Our compiler implementation extends Merrill and Garland's algorithm in several ways: First, we of course support scan operators $\alpha \to \alpha \to \alpha$ for arbitrary type α. Second, and more importantly, our compiler implementation allows for the scan bulk operation to be fused with a map operation that produces some of the values that are scanned. Throughout this section, and the implementation section, we will use the variable M to denote the number of elements (from to-be-scanned array) that will be sequentially processed by each thread[3]. The value we have used is $M = 9$, but in a more complete implementation this should be adjusted based on the number and type of parameters. This is discussed in the Resource usage and the Possible improvements & limitations sections. The next section present the algorithm in more detail, and flowchart 2 graphically depicts steps (2)-(4) in the itemized list above.

[2] For example, let us assume that a maximal number q of blocks can be concurrently executed, but that block 0 has not been scheduled for execution yet. Then we are in a deadlock situation because the currently executed blocks need the value of the prefix of block 0, but block 0, who is the one computing it, cannot be scheduled before one of the other blocks finishes.

[3] This conforms with Brent's Lemma related to efficiently sequentializing the parallelism in excess of what the hardware can support.

3 Algorithm - Detailed explanation

In this section, we will go into details of how each part of the algorithm works.

3.1 Dynamic block numbering

Since this algorithm is dependent upon each block receiving the prefix corresponding to the computation of previous blocks, we need to make sure that if block i has been scheduled for execution, then all blocks less than i have finished or are also under execution. (Otherwise we may possibly have a deadlock, as explained in section 2.2) Normally, there is no such guarantee, but by dynamically allocating block numbers, after a block has been spawned, we can achieve the same effect. This is accomplished by having each block perform an atomic read, which increases a counter maintained in global memory. The atomic read returns the previous (non-increased) value, which is then used as the block number, throughout the whole implementation.

```
1   // DynamicIdGenesis is a global memory location, which is initialized to 0
2   // before anything is ran.
3
4   shared dynamicIdShared = null
5
6   if threadId == 0:
7     dynamicIdShared = atomicInc(DynamicIdGenesis) // Reads and increases by 1
8
9   localBarrier()
10  dynamicId = dynamicIdShared
11
```

Figure 1: dynamic id pseudocode

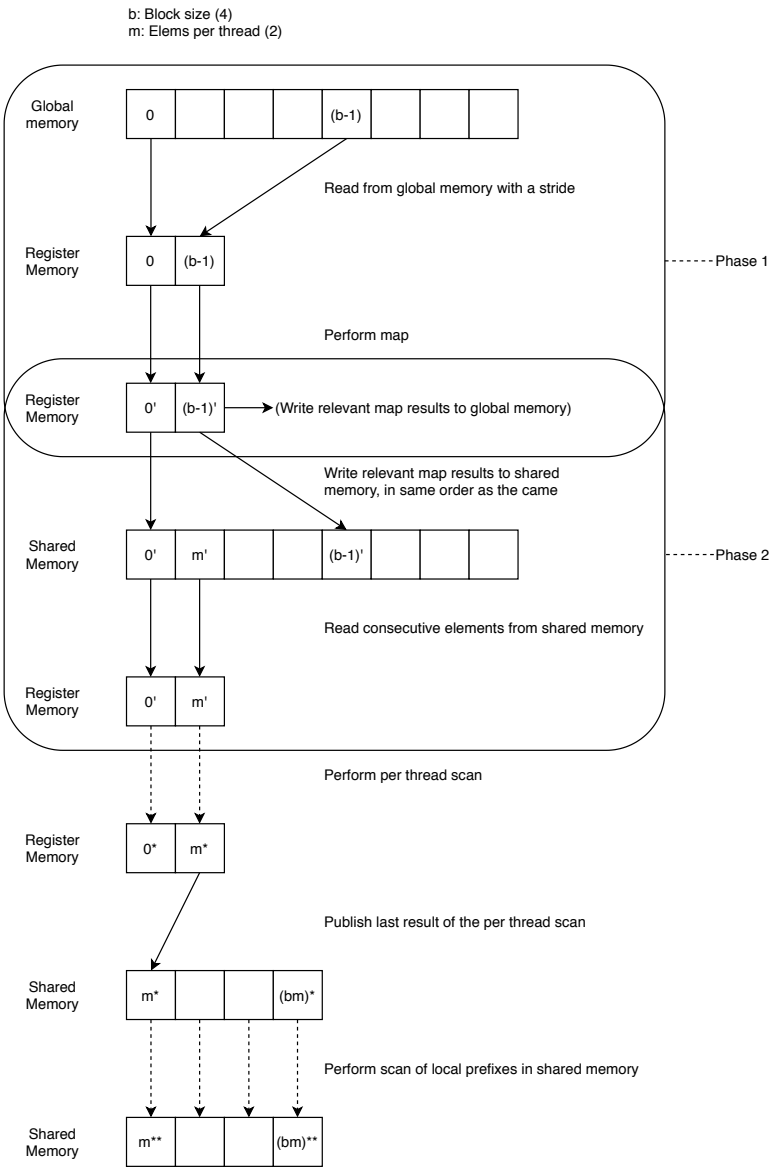

Figure 2: Flow chart of the algorithm, up to before lookback (not including dynamic id

3.2 Global memory read and map - Phase 1

This phase is illustrated in Figure 2, in the Phase 1 part.

In order to achieve a high throughput of the algorithm, it is especially important to read from global memory as fast as possible. In order to do this, we need to read in a coalesced fashion. This is done by having each thread read from addresses adjacent to each other, requiring each thread to read with a stride, as show in Figure 2 (see Figure 3 for how the indexes are calculated). The value read is then mapped, and any part of the result that needs to be returned directly, is written to global memory, and any part that is input to the scan function is kept in register memory. This process is repeated for each element each thread has to process (ie. M times).

```
1   blockOffset = dynamicId * M * groupSize
2   private[M] privateArray
3
4   for i in 0 to M:
5     globalIndex = blockOffset + threadId + i * groupSize
6     value       = input[globalIndex]
7     mapResult   = mapFunction(value)
8
9     (mapResult_global, mapResult_private) = split(mapResult)
10    mapOutput[globalIndex] = mapResult_global
11    privateArray[i]        = mapResult_private
12
```

Figure 3: Load and map pseudocode

3.3 Transposition - Phase 2

This phase is illustrated in Figure 2, in the Phase 2 part.

We now have the input to the scan function in our private arrays, but they are in the wrong order. Fx. the first thread has the scan inputs to index 0, $0 + \texttt{blockSize}$, $0 + 2 \cdot \texttt{blockSize}$, and so on. In order for the algorithm to progress, we need each thread to have consecutive elements. To achieve this, all threads needs to put their mapped values into shared memory, and read back elements in consecutive order. This can be done in 2 ways. One way is to have each thread write it's values into shared memory, in the order that is has them, and then read back with a stride of `blockSize`. The other other way (the one we have chosen to use), is to have each thread write with a stride of `blockSize` to shared memory, and read back consecutive elements (as illustrated in Figure 2 and Figure 4). The performance differences between the two methods should be negligible.

```
1   // privateArray contains values set in the previous section,
2   // in the same order as they were set.
3   local[m * groupSize] transpositionArray
4
5   for i in 0 to M:
6     sharedIdx = threadId + i * groupSize
7     transpositionArray[sharedIdx] = privateArray[i]
8
9   for i in 0 to M:
10    sharedIdx = threadId * M + i
11    privateArray[i] = transpositionArray[sharedIdx]
12
```

Figure 4: "Transposition" of mapped values pseudocode

3.4 Per thread scan

In order to maximize throughput, we need to minimize the amount of communication needed between threads and between blocks. To do this, we allow each thread to process as many elements as possible by itself, since this requires no inter-thread communication. This number (M) is ultimately limited by the amount of shared memory and register memory that can be used, without significantly affecting occupancy. This is simply achieved by having each thread sequentially scan its elements, which are kept in register memory. Once each thread has completed this step, all the threads in a block "publish" their last element into an array in shared memory.

3.5 Block level scan

The results published in the previous stage, is then scanned, in order to produce an array of prefixes. Each thread then gets the index of its own `threadId - 1`, in this array of prefixes. This value is the accumulated values of all of the elements preceding each thread, inside the block. If each thread were to map this value onto its elements, using the scan operator, all of the elements in the block would be scanned, as if they were the only elements. Since we would still need to map the accumulated value of all of the elements preceding this block, we don't do this yet. Further, since there are no elements in the block, before the elements of the first thread, the first thread gets the last result of the scanned local prefixes, for reasons that will become apparent in the next section. The process in illustated in Figure 5.

```
1   // Contains the value of each threads last element:
2   local[groupSize] individualThreadResults
3
4   localPrefixes = scan operator individualThreadResults
5
6   accumulator = null
7   if threadId == 0:
8     // First thread get the last element
9     accumulator = localPrefixes[groupSize - 1]
10  else:
11    accumulator = localPrefixes[threadId]
12
```

Figure 5: How each thread finds it's accumulator

3.6 Synchronizing with other blocks

Before we detail how we synchronize, we need to establish a couple of things. To communicate with the other blocks, we maintain three arrays in global memory, all of them having a length equal to the total number of blocks. One array records the status flags (see Table 1 for the flags and their meaning), one array records the *aggregates* values, and the last array records the *inclusive prefixes*. A block's *aggregate* is the reduction across all the elements processed by the current block. A block's *inclusive prefix* is the accumulated value of all of the elements

Flag	Meaning
`X`	The block hasn't published anything
`Aggregate`	The aggregate of the block can be found in the global aggregate array
`Prefix`	The inclusive prefix of the block can be found in the global prefix array

Table 1: Flags and what they mean

processed by the blocks up until and including the that block, i.e., is the sum of the aggregates up to and including the that block.

So far, we only have that each thread in our block has scanned its elements (and has a local prefix ready from the previous threads). So to continue, we need to find the inclusive prefix of the previous block, i.e., the accumulated value from all of the aggregates of all of the previous blocks. To do this, we need each block to publish (share) its aggregated value. This is achieved by the first thread in each block, which updates the corresponding entry in the array of aggregates (how the first thread got this value is detailed in the section 3.5). When it is certain that the value has been written to global memory (accomplished by using a global fence), the status flag array is also updated at the index of its `blockId` to the value `Aggregate`. Since the very first block has no elements before it, it publishes its aggregate as a `prefix` instead.

When we have published our own aggregate (which was as soon as possible, in order to help other blocks), we can start looking back at what other blocks have published. We start by looking back at the block immediately before us, to see if it has published its inclusive prefix. If it has, we can use it directly and don't need to look further back. This first part is not strictly necessary, as the algorithm would still get the correct result without it, but it improves performance performance.

If the previous block has not published its inclusive prefix yet, the current block needs to compute it itself.[4] To do this, we start by getting the flags and values of the previous 32 (the size of a warp) blocks. We then reduce these flags and values down to a single flag and value. This reduction is done from lower numbered blocks to higher numbered, by combining their flags and values according to table 2.

Left hand flag	Right hand flag	Result
Anything	`X`	Resulting flag is X, and there is no value
`X`	Anything	The right hand flag and value is taken
`Aggregate`	`Aggregate`	Resulting flag is aggregate, and the 2 values combined using the scan operator
`Prefix`	`Aggregate`	Resulting flag is prefix, and the 2 values are combined using the scan operator

Table 2: How flags and values are combined during reduction

[4]The alternative is for the current block to wait until the previous block computes its inclusive prefix. This approach underutilizes the system's bandwidth, and induces a serial dependency, so instead we use the approach in which each block may perform some redundant computation.

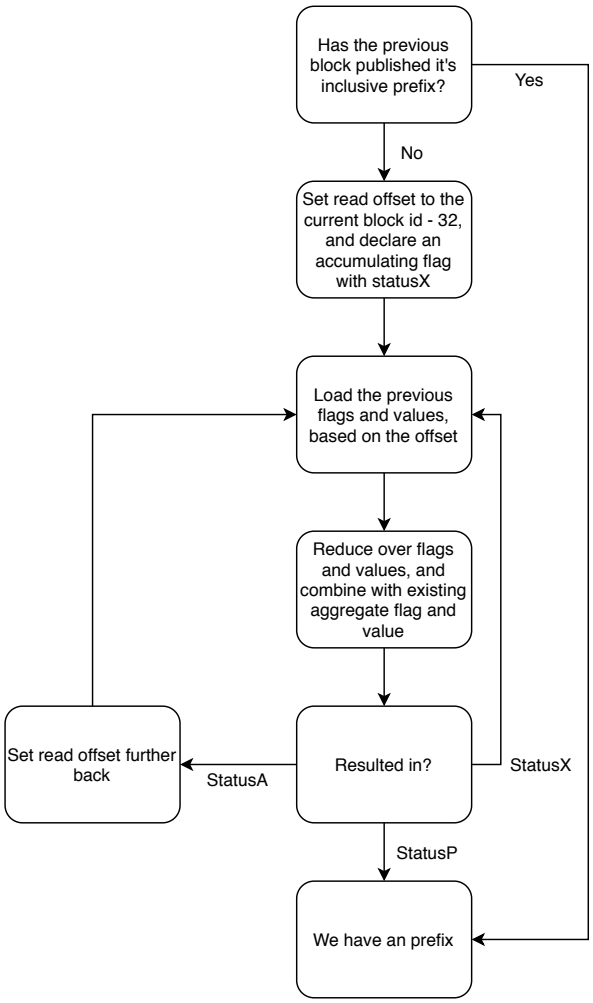

Figure 6: Decision diagram for the lookback phase

As detailed in Figure 6, we now do one of three things.

(1) If we ended up with a flag of `X`, we simply repeat the process, reading from the same indexes (hoping that the previous blocks has published their results).

(2) If we ended up with a flag of `aggregate`, we adjust the offset we're looking back with, and continue looking further back. As long as any of these two flag are the result of the reduce, we repeat the process.

(3) When we end up with a result of `Prefix`, we have computed the full prefix of all the elements processed by previous blocks, and we can stop the lookback.

Before we start distributing this prefix to the elements of the current block, we combine it with the aggregate of the current block, to compute the *inclusive prefix* of the current block. We then write this value into the global array of inclusive prefixes, and update the flag of our block from `aggregate` to `prefix`. (A global-memory fence is required between these two updates, so that the other blocks observe the updates in the correct order, just like in the case when we published the aggregate).

The reason we need each block to spend time publishing its inclusive prefix, is to allow the following blocks to stop their lookback early, without having to go all the way back to the first block. Without this part, the depth of the algorithm, would be $O(n)$, and thus not parallel at all.

3.7 Distribution of lookback result

We now have the accumulated value of all of the elements preceding the current block, ie. the value found during the lookback. Further, each thread has the accumulated value of all of the elements preceding it, inside the block. Now each thread need to map these 2 values onto its elements. To save time, we first combine these 2 values, and then simply map the result onto our elements in a for loop, see Figure 7 for pseudocode.

```
1    // exclusivePrefix contains the result found in the previous section
2    // accumulator contains the value it got in block level scan section
3    // privateArray contains the values set in the Per thread scan section
4
5    if threadId == 0:
6      accumulator = neutralElement
7
8    accumulatorAndPrefix = operator accumulator exclusivePrefix
9
10   for i in 0..M:
11     privateArray[i] = operator accumulatorAndPrefix privateArray[i]
12
```

Figure 7: How each thread distributes the result of the lookback and the accumulator

3.8 Writing the private data to global memory

We now have a final result in a private-memory array, and we need to write it to global memory. Just like when we read the input in the first place, we have to write back to global memory in a coalesced fashion. In order to to this, we again need to "transpose" them the same way we did earlier, by using shared memory as a staging buffer, but the steps are reversed, i.e., first we write from private to shared memory, and then from shared to global memory. Figure 8 shows the pseudocode that achieves this coalesced update of global memory.

```
1   // privateArray contains the values set in the previous section
2   local[m * groupSize] transpositionArray
3
4   for i in 0 to M:
5     sharedIdx = threadId * M + i
6     transpositionArray[sharedIdx] = privateArray[i]
7
8   for i in 0 to M:
9     sharedIdx = threadId + groupSize * i
10    privateArray[i] = transpositionArray[sharedIdx]
11
12  blockOffset = dynamicId * M * groupSize
13
14  for i in 0..M:
15    globalIdx = blockOffset + (groupSize * i) + threadId
16    resultArray[globalIdx] = privateArray[i]
17
```

Figure 8: How each thread distributes the result of the lookback and the accumulator

4 Resource usage

The limiting factor for how many elements each thread can process—i.e., what should the value of M be?—is a combination of the amount of shared memory available per thread, and the amount of register memory available per thread. Before we can discuss this, we first need to explain what an element can be. In section 2.2 and 3, elements were discussed as though they were of basic types, but this is not necessarily so. For example, the type of an element can be a q-tuple tuple, where q is statically known, e.g., `(int, int, int, int)`. In the following computation we assume that the element type of the input array is some tuple type $\bar{\alpha} = (\alpha_1, \ldots, \alpha_q)$. In the places were we use shared memory, we use the following amounts:

1. During the transposition phase, we transpose one component of all of the elements, at a time. This uses a maximum of $M \cdot \texttt{groupSize} \cdot max_{i=1}^{q}(sizeof(\alpha_i))$ bytes of shared memory.

2. During the block level scan, we need to hold an entire element for each thread, in shared memory. This uses $\texttt{groupSize} \cdot sizeof(\alpha) = \texttt{groupSize} \cdot sum_{i=1}^{q}(sizeof(\alpha_i))$ bytes of shared memory.

3. During the lookback phase, we use $32\text{bytes} + 32 \cdot sizeof(\alpha)$ bytes of shared memory.

4. To share the result of the lookback, shared memory is used to store a single element. This however will always be less than any of the other phases.

These 4 instances of shared memory use, are never required to exist at the same time, and therefore we can allocate the maximal amount required by any stage, and reuse the available shared memory for each stage. The resulting allocation is a formula in unknown M, which can be computed by searching for the maximal value of M that does not result in a significant

decrease of occupancy—i.e., the number of blocks scheduled concurrently on the same SM is limited by the amount of shared memory available on the SM (for Nvidia, this is 48Kb).

We use the following amounts of register memory:

1. When loading the input to the map function, we have to hold a single element of it in register memory, before we can do the map.
2. After performing the map, we have to hold the result, before we can write any relevant part(s) of it to global memory. Again, we only need to hold one instance.
3. Throughout the algorithm, each thread holds its elements in its registers, using $M \cdot sizeof(\alpha)$ bytes of register memory.
4. After the block level scan phase, each thread additionally holds $sizeof(\alpha)$ bytes of register memory.
5. After the lookback phase, each thread additionally holds $sizeof(\alpha)$ bytes of register memory.

The first 2 needs to be able to exist at the same time as the third, but once the transposition starts, they no longer needs to exist. The last 3 of these need to exist at the same time. Therefore, the constraint they impose on M, is that the sum of them must be lesser or equal to the amount of register available to each thread. Further, there is a general overhead, since any variables used by the program also takes up space in register memory.

5 Implementation

In order to implement this single pass scan in the Futhark compiler, we've changed the implementation of the `scanomap` primitive, which is a higher order primitive function that consists of a segmented scan fused with a map. Another thing to note is that at this intermediate representation we're working at, arrays of composite types are split up into an array for each component [5].

Our implementation can be found in `src/Futhark/CodeGen/ImpGen/Kernel/SegScan.hs` in our clone of the Futhark repository. Which can be found at `https://github.com/AndreasNicolaisen/futhark/tree/wip`.

5.1 Global load and mapping phase implementation

The first stage of the scan code generation first assigns each block a dynamic id using atomic incrementation of a global variable. Then a for-loop repeating M times is generated, where for each iteration the map function is applied to the input loaded from global memory (read in a coalesced fashion). Each component of the map result is either written directly back to global memory if it's to be used elsewhere, and any parts of the components used by the scan are written to private memory.

```
1  sFor "i" tM $ \i -> do
2    -- The map's input index
3    dPrimV_ gtid $ (tvExp blockOff) + localThreadId + i * groupSize
4    -- Perform the map
5    sWhen ((Imp.vi32 gtid) .<. n) $ do
6      compileStms mempty (kernelBodyStms kbody) $
7        do let (scan_res, map_res) =
8                 splitAt (segBinOpResults scans) $ kernelBodyResult kbody
9           forM_ (zip (takeLast (length map_res) all_pes) map_res) $
10              \(dest, src) -> do
11                -- Write map results to their global memory destinations
12                copyDWIMFix (patElemName dest) [Imp.vi32 gtid]
13                            (kernelResultSubExp src) []
14
15          forM_ (zip privateArrays $ map kernelResultSubExp scan_res) $
16            \(dest, (Var src)) -> do
17              copyDWIMFix dest [i] (Var src) []
```

(a) Compiler Implementation

```
1  for (int32_t i = 0; i < 9; i++) {
2      int32_t gtid =
3        blockOff + local_tid + i * segscan_group_size;
4      if (slt32(gtid, n)) {
5          float x =
6            ((__global float *) xs_mem)[sext_i32_i64(gtid)];
7
8          private_mem[sext_i32_i64(i)] = x;
9      }
10 }
```

(b) Generated OpenCL

Figure 9: Global-load and mapping phase of the program: `scan (+) 0 xs`

In Figure 9a the Futhark compiler implementation of this phase is shown besides the generated OpenCL in Figure 9b. Lines 1-5 in the compiler code directly corresponds to line 1-4 in the generated code. Line 6 then compiles the map lambda body (which is the identity function in this case), which in effect simply loads the input into `x` as seen in the output code line 5-6. Line 7-8 then divides the results of the mapping into the components that are used by the scan (`scan_res`) and the ones to be written to global memory (`map_res`). Then line 9-13 generate the code for writing the results to global memory, and line 15-17 generate the code for writing to private memory, corresponding to line 8 in the generated code.

Note that the `privateArrays` referred to the implementation code is an array that contains a private array of M items for each component in the scan type.

5.2 Transposition phase implementation

This phase then writes the values held in the thread's private arrays into local memory such that each thread can read sequential values back into their private memory, so the data is ready to be scanned. This is done for each component, one component at a time, so that local memory only has to hold the data of one component at a time, reusing the local memory.

```
1  forM_ (zip transposedArrays privateArrays) $ \(trans, priv) -> do
2    sOp localBarrier
3    sFor "i" tM $ \i -> do
4      sharedIdx <- dPrimV "sharedIdx" $ localThreadId + i * groupSize
5      copyDWIMFix trans [tvExp sharedIdx] (Var priv) [i]
6    sOp localBarrier
7    sFor "i" tM $ \i -> do
8      sharedIdx <- dPrimV "sharedIdx" $ localThreadId * tM + i
9      copyDWIMFix priv [i] (Var trans) [tvExp sharedIdx]
10 sOp localBarrier
```

(a) Compiler Implementation

```
1  barrier(CLK_LOCAL_MEM_FENCE);
2  for (int32_t i = 0; i < 9; i++) {
3      int32_t sharedIdx = local_tid + i * segscan_group_size;
4
5      ((__local float *) local_mem)[sext_i32_i64(sharedIdx)] =
6          private_mem[sext_i32_i64(i)];
7  }
8  barrier(CLK_LOCAL_MEM_FENCE);
9  for (int32_t i = 0; i < 9; i++) {
10     int32_t sharedIdx = local_tid * 9 + i;
11
12     private_mem[sext_i32_i64(i)] =
13       ((__local float *) local_mem)[sext_i32_i64(sharedIdx)];
14 }
15 barrier(CLK_LOCAL_MEM_FENCE);
```

(b) Generated OpenCL

Figure 10: Transposition phase of the program: `scan (+) 0 xs`

We can see in Figure 10 that line 1 in the implementation starts looping over the private and local array for each component. At each iteration line 3-5 then generates the code that transposes the private values into shared memory, corresponding to line 2-7 in the generated code. And line 7-9 generate the code load the sequential elements back into private memory, corresponding to line 9-14 in the generated code.

5.3 Per-thread scan phase implementation

In this phase each thread then scans its private elements in place, and then publishes the last value in local memory.

```
1  sFor "i" tM $ \i -> do
2    let xs  = map paramName $ xParams scanOp
3        ys  = map paramName $ yParams scanOp
4        nes = segBinOpNeutral scanOp
5
6    mapM_
7      (\(src, (x, y, ne, ty)) ->
8         do dPrim_ x ty
9            dPrim_ y ty
10           sIf (i .==. 0)
11             (copyDWIMFix x [] ne [])
12             (copyDWIMFix x [] (Var src) [i - 1])
13           copyDWIMFix y [] (Var src) [i])
14     $ zip privateArrays $ zip4 xs ys nes tys
15
16   compileStms mempty (bodyStms $ lambdaBody $ segBinOpLambda scanOp) $
17     mapM_
18       (\(dest, res) ->
19          copyDWIMFix dest [i] res [])
20       $ zip privateArrays $ bodyResult $ lambdaBody $ segBinOpLambda scanOp
21
22 mapM_ (\(dest, src) ->
23         copyDWIMFix dest [kernelLocalThreadId constants]
24                     (Var src) [tM - 1])
25       $ zip prefixArrays privateArrays
26 sOp localBarrier
```

(a) Compiler Implementation

```
1  for (int32_t i = 0; i < 9; i++) {
2      float x;
3      float y;
4
5      if (i == 0) {
6          x = 0.0F;
7      } else {
8          x = private_mem[sext_i32_i64(i - 1)];
9      }
10     y = private_mem[sext_i32_i64(i)];
11
12     float res;
13
14     res = x + y;
15     private_mem[sext_i32_i64(i)] = res;
16 }
17
18 ((__local float *) local_mem)[sext_i32_i64(local_tid)] =
19   private_mem[8];
20 barrier(CLK_LOCAL_MEM_FENCE);
```

(b) Generated OpenCL

Figure 11: Per-thread scan phase of the program: `scan (+) 0 xs`

We can see in Figure 11 that line 1 in the implementation that the outer loop is generated, corresponding to line 1 in the generated code. Line 2-4 then extracts which parts of the scan operator's lambda's parameters are the first and second arguments (the xs and ys) and the neutral element for each component. Then line 6-14 then either sets `x` to be neutral element or the previous scan results, and `y` to be the current element, corresponding to line 2-10 in the generated code. Line 16 then compiles the lambda body corresponding to line 12-14 in the generated code. Line 17-20 then stores each component of the result into its respective private array (in-place), corresponding to line 15 in the generated program. Finally line 22-25 then writes the last scanned element to shared memory for each component, corresponding to line 18-19 in the generated program.

5.4 Group scan phase implementation

In the group scan phase all the publishes values from the per-thread scan phase is then scanned over in local memory. Each thread then sets their accumulator to be the previous thread's scan result, which will be used later to prefix all it's private elements. The first thread in each block sets its accumulator the very last resulting value, which will be used by the lookback stage.

```
1  accs <- forM tys (\ty -> dPrim "acc" ty)
2  groupScan
3    Nothing
4    (tvExp numThreads)
5    (kernelGroupSize constants)
6    scanOp'
7    prefixArrays
8
9  forM_ (zip accs prefixArrays)
10   (\(acc, prefixes) ->
11      sIf (localThreadId .==. 0)
12          (copyDWIMFix (tvVar acc) [] (Var prefixes) [groupSize - 1])
13          (copyDWIMFix (tvVar acc) [] (Var prefixes) [localThreadId - 1]))
14
15 sOp localBarrier
```

(a) Compiler Implementation

```
1  float acc;
2
3  // ...
4  //   Group scan implementation omitted
5  // ...
6
7  barrier(CLK_LOCAL_MEM_FENCE);
8  if (local_tid == 0) {
9    acc = ((__local float *) local_mem)
10           [sext_i32_i64(segscan_group_size - 1)];
11  } else {
12   acc = ((__local float *) local_mem)
13           [sext_i32_i64(local_tid - 1)];
14  }
15 barrier(CLK_LOCAL_MEM_FENCE);
```

(b) Generated OpenCL

Figure 12: Group scan phase of the program: `scan (+) 0 xs`

As seen in Figure 12 on line 1, an accumulator variable for each component is declared, corresponding to line 1 in the generated program. Then on line 2-7 the group scan is generated using a helper function already present in the compiler implementation, the resulting (long) code has been omitted from the generated program shown. Then on line 9-13 the accumulator is loaded with the resulting values for each component, corresponding to line 8-14 in the generated program.

5.5 Lookback phase implementation

The lookback phase implementation is more or less a direct implementation of the algorithm described earlier, handling multiple components the same way as seen in earlier phases. The global inclusive-prefix and aggregate arrays are split up into multiple global arrays for each component, just like local and private arrays. The algorithm's implementation (and generated code) is rather large, so for the sake of brevity it has been omitted.

5.6 Distribution and global write phase implementation

Here the prefix gotten from the lookback is combined with each thread's aggregate in order to arrive at the value each element in the thread's private memory should be prefixed with. Then the final results are written to global memory.

We can see in Figure 13a that line 1-4 extracts the x and y parameters for two instances of the scan operator lambda (one for combing the aggregate and the prefix, and one for prefixing the private elements). Then on lien 6-12 the parameters are declared, and the first lambda's parameters are initialized, corresponding to line 1-5 in the generated program. On line 14-16 the first lambda body is then compiled and the result is assign the x parameter of the second lambda (for each component), corresponding to line 7-8 in the generated code.

```
1  let xs  = map paramName $ xParams scanOp'
2      ys  = map paramName $ yParams scanOp'
3      xs' = map paramName $ xParams scanOp''
4      ys' = map paramName $ yParams scanOp''
5
6  mapM_
7    (\((prefix, acc), (x, x'), (y, y'), ty) ->
8      do dPrim_ x ty
9         dPrim_ y ty
10        dPrimV_ x' (tvExp acc)
11        dPrimV_ y' (tvExp prefix))
12   $ zip4 (zip prefixes accs) (zip xs xs') (zip ys ys') tys
13
14 compileStms mempty (bodyStms $ lambdaBody scanOp'') $ do
15   forM_ (zip3 xs tys $ bodyResult $ lambdaBody scanOp'')
16     (\(x, ty, res) -> x <~~ toExp' ty res)
17
18 sFor "i" tM $ \i -> do
19   mapM_
20     (\(src, y) ->
21        copyDWIMFix y [] (Var src) [i])
22     $ zip privateArrays ys
23
24   compileStms mempty (bodyStms $ lambdaBody scanOp') $
25     mapM_
26       (\(dest, res) ->
27          copyDWIMFix dest [i] res [])
28       $ zip privateArrays $ bodyResult $ lambdaBody scanOp'
29
30 forM_ (zip (map patElemName all_pes) privateArrays) $ \(dest, src) -> do
31   sFor "i" tM $ \i -> do
32     dPrimV_ gtid $ (tvExp blockOff) + localThreadId * (tM) + i
33     sWhen ((Imp.vi32 gtid) .<. n) $ do
34       copyDWIMFix dest [Imp.vi32 gtid] (Var src) [i]
```

(a) Compiler Implementation

```
1  float x1;
2  float y1;
3  float x0 = acc;
4
5  float y0 = prefix;
6
7  float res0 = x0 + y0;
8  x1 = res0;
9  for (int32_t i = 0; i < 9; i++) {
10     y1 = private_mem[sext_i32_i64(i)];
11
12     float res1 = x1 + y1;
13     private_mem[sext_i32_i64(i)] = res1;
14 }
15
16 for (int32_t i = 0; i < 9; i++) {
17     int32_t gtid = blockOff + local_tid * 9 + i;
18
19     if (slt32(gtid, n)) {
20         ((__global float *) mem)[sext_i32_i64(gtid)] =
21             private_mem[sext_i32_i64(i)];
22     }
23 }
```

(b) Generated OpenCL

Figure 13: Distribution and global write back for program: `scan (+) 0 xs`

On line 18-28 a loop over all private items is generated where each component of the private items are loaded into the y parameter of the second lambda, the body compiled and the result stored back into the private array, corresponding line 9-14 in the generated code. Finally on line 30-34 all the private items then then written into global memory, corresponding to line 16-23 in the generated code.

6 Benchmarks

Our benchmarks can be seen in Table 3 and 4. In Table 3, we compare our implementation to the original Futhark compiler. We are running 3 different programs, "Simple-1GiB" is, as the name would suggest, a simple scan on 32 bit integers, with the plus operator. The "1GiB" being the size of the input. "Advanced-100MiB" is a fused map and scan, that takes a single array of 32 bit integers as input. The map then produces 2 new elements, which the scan operators on. The scan then produces 2 elements as well. Finally, all 4, as well as the input, are returned. The final program "Radix-100MiB"[5] run radix sort on 32 bit integers. All 3 programs can be found in the Appendix.

Times (μ seconds)	Our		Futhark	
Program:	GTX 780Ti	RTX 780Ti	GTX 780Ti	RTX 780Ti
Simple-1GiB	15980	4750	50150	11572
Advanced-100MiB	3109	1097	7593	2213
Radix-100Mib	136568	50178	230707	82663

Table 3: Programs and their run times.

Simple-1GiB	Our		Reference		MemCpy	
Comparison:	GTX 780Ti	RTX 780Ti	GTX 780Ti	RTX 780Ti	GTX 780Ti	RTX 780Ti
Time (micro s.)	15980	4750	12773	4377	8251	4027
Throughput (GiB/s)	125.16	421.05	156.58	456.93	242.39	496.65

Table 4: Comparison to reference solution.

As we can see in Table 3, we significantly outperform the original implementation in the Futhark compiler. For the first 2 simpler programs, the speedup is between 2 to 3 times. This is as expected, since we have around half of the amount of accesses to global memory, compared to what the original implementation has. For the last one, the speedup is not quite as good, but still significant.

In Table 4 we compare our implementation to a reference solution, written in OpenCL. We are nearly as fast, but are probaly missing some performance, since our reduction during the lookback is done with a single thread, whereas the reference implementation does it with an entire warp.

[5]adapted from: `https://github.com/diku-dk/sorts/blob/60879a9d0758d61c855540c6accf8b627e7376d3/lib/github.com/diku-dk/sorts/radix_sort.fut`

7 Possible improvements & limitations

First of all, let's list some of the limitations of the algorithm and implementation:

- The part of the compiler that our implementation replaces, also covers the case of *regular-segmented* scan, which we do not support (yet) in our implementation. However, we hypothesize that this can be added in a manner similar to how the regular-segmented reduce is implemented [7].
- We don't handle multiple scan operators being fused, albeit this is a relatively straightforward extension.

Then we have some possible improvements:

- The reduction of flags and values during the lookback, is run using a single thread, but should probably be done using an entire warp.
- The reduction function used does a lot of bitpacking, to keep the flag, and the number of used values in the same byte. This allows us the adjust the lookback offset in small steps, but might be more expensive, than just stepping with 32 at a time. Whether this is actually faster is unknown to us, the question lies in how expensive the bitpacking is, compared to reading flags and values unnecessarily.
- Remove hardcoding of number of elements per thread (M), and use constraints mentioned in section 4 to calculate the maximum possible value.

8 Conclusion

In this report, we have detailed an algorithm for single pass parallel scan. We have explained the details of the algorithm, and how it works. Further, we have explained how we implemented the algorithm in the Futhark compiler, and related how different parts of the compiler corresponds to different parts of the OpenCL code it generates. We have compared its performance to that of the original implementation in the Futhark compiler, as well as a reference implementation written directly in OpenCL. Finally, we have explained the limitations of the algorithm and implementation, and given some possible improvements that could be made.

9 References

[1] Christian Andreetta, Vivien Bégot, Jost Berthold, Martin Elsman, Fritz Henglein, Troels Henriksen, Maj-Britt Nordfang, and Cosmin E. Oancea. Finpar: A parallel financial benchmark. *ACM Trans. Archit. Code Optim.*, 13(2):18:1–18:27, June 2016.

[2] Guy E. Blelloch. Scans as Primitive Parallel Operations. *Computers, IEEE Transactions*, 38(11):1526–1538, 1989.

[3] Guy E Blelloch, Jonathan C Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a Portable Nested Data-Parallel Language. *Journal of parallel and distributed computing*, 21(1):4–14, 1994.

[4] Martin Elsman, Troels Henriksen, and Niels Gustav Westphal Serup. Data-parallel flattening by expansion. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2019, page 14–24, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Troels Henriksen, Ken Friis Larsen, and Cosmin E. Oancea. Design and gpgpu performance of futhark's redomap construct. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2016, page 17–24, New York, NY, USA, 2016. Association for Computing Machinery.

[6] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. Incremental flattening for nested data parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 53–67, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Rasmus Wriedt Larsen and Troels Henriksen. Strategies for regular segmented reductions on gpu. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, FHPC 2017, page 42–52, New York, NY, USA, 2017. Association for Computing Machinery.

[8] Duane Merrill and Michael Garland. Single-pass parallel prefix scan with decoupled look-back, 2016.

[9] Cosmin E. Oancea, Christian Andreetta, Jost Berthold, Alain Frisch, and Fritz Henglein. Financial software on gpus: Between haskell and fortran. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '12, pages 61–72, New York, NY, USA, 2012. ACM.

[10] Cosmin E. Oancea and Alan Mycroft. Set-congruence dynamic analysis for thread-level speculation (tls). In *Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008, Revised Selected Papers*, pages 156–171, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] Cosmin E. Oancea and Lawrence Rauchwerger. A hybrid approach to proving memory reference monotonicity. In Sanjay Rajopadhye and Michelle Mills Strout, editors, *Languages and Compilers for Parallel Computing*, pages 61–75. Springer Berlin Heidelberg, 2013.

[12] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop transformations: Convexity, pruning and optimization. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 549–562, New York, NY, USA, 2011. ACM.

10 Appendix

10.1 Appendix A - simple.fut

```
-- ==
-- entry: main
--
-- compiled random input { [10000000]i32 } auto output

let main [n] (xs:[n]i32): [n]i32 =
  scan (+) 0 xs
```

10.2 Appendix B - advanced.fut

```
-- ==
-- entry: main
--
-- compiled random input { [10000000]f32 } auto output

let main [n] (as: [n]i32): ([n]i32, [n]i32, [n]i32, [n]i32, [n]i32) =
  let (bs, cs) = unzip <| map (\a -> (a - 1, a + 1)) as
  let (ds, es) = unzip <| scan (\(xb, xc) (yb, yc) -> (xb + yb, xc + yc)) (0, 0)
                               <| zip bs cs
  in (as, bs, cs, ds, es)
```

10.3 Appendix C - radix_sort.fut

```
-- ==
-- entry: main
--
-- compiled random input { [26214400]i32 } auto output

local let radix_sort_step [n] 't (xs: [n]t) (get_bit: i32 -> t -> i32)
                                 (digit_n: i32): [n]t =
  let num x = get_bit (digit_n+1) x * 2 + get_bit digit_n x
  let pairwise op (a1,b1,c1,d1) (a2,b2,c2,d2) =
    (a1 `op` a2, b1 `op` b2, c1 `op` c2, d1 `op` d2)
  let bins = xs |> map num
  let flags = bins |> map (\x -> if x == 0 then (1,0,0,0)
                                 else if x == 1 then (0,1,0,0)
                                 else if x == 2 then (0,0,1,0)
                                 else (0,0,0,1))
  let offsets = scan (pairwise (+)) (0,0,0,0) flags
  let (na,nb,nc,_nd) = last offsets
  let f bin (a,b,c,d) = match bin
                        case 0 -> a-1
                        case 1 -> na+b-1
```

```
                        case 2 -> na+nb+c-1
                        case _ -> na+nb+nc+d-1
  let is = map2 f bins offsets
  in scatter (copy xs) is xs

-- | The ‘num_bits‘ and ‘get_bit‘ arguments can be taken from one of
-- the numeric modules of module type ‘integral‘@mtype@"/futlib/math"
-- or ‘float‘@mtype@"/futlib/math", such as ‘i32‘@term@"/futlib/math"
-- or ‘f64‘@term@"/futlib/math".  However, if you know that
-- the input array only uses lower-order bits (say, if all integers
-- are less than 100), then you can profitably pass a smaller
-- ‘num_bits‘ value to reduce the number of sequential iterations.
--
-- **Warning:** while radix sort can be used with numbers, the bitwise
-- representation of of both integers and floats means that negative
-- numbers are sorted as *greater* than non-negative.  Negative floats
-- are further sorted according to their absolute value.  For example,
-- radix-sorting ‘[-2.0, -1.0, 0.0, 1.0, 2.0]‘ will produce ‘[0.0,
-- 1.0, 2.0, -1.0, -2.0]‘.  Use ‘radix_sort_int‘@term and
-- ‘radix_sort_float‘@term in the (likely) cases that this is not what
-- you want.
let radix_sort [n] 't (num_bits: i32) (get_bit: i32 -> t -> i32)
                      (xs: [n]t): [n]t =
  let iters = if n == 0 then 0 else (num_bits+2-1)/2
  in loop xs for i < iters do radix_sort_step xs get_bit (i*2)

-- | A thin wrapper around ‘radix_sort‘@term that ensures negative
-- integers are sorted as expected.  Simply pass the usual ‘num_bits‘
-- and ‘get_bit‘ definitions from e.g. ‘i32‘@term@"/futlib/math".
let radix_sort_int [n] 't (num_bits: i32) (get_bit: i32 -> t -> i32)
                          (xs: [n]t): [n]t =
  let get_bit' i x =
    -- Flip the most significant bit.
    let b = get_bit i x
    in if i == num_bits-1 then b ^ 1 else b
  in radix_sort num_bits get_bit' xs

let main [n] (xs: [n]i32): [n]i32 =
  radix_sort_int i32.num_bits i32.get_bit xs
```