FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Implementing Map-Scan Fusion in the Futhark Compiler

Bachelor project

Brian Spiegelhauer
brianspieg@gmail.com

William Jack Lysgaard Sprent
bsprent@gmail.com

June 9, 2016

Contents

1 Abstract

2 Introduction

The Futhark language is a functional programming with which the main idea is to allow for the expression of sufficiently complex programs while exploiting functional invariants such that programs can be aggressively optimised and have their parallelism exploited [1]. Its main target is to support general purpose computing on GPUs (GPGPU) to exploit their high parallel computational power.

Loop fusion is a method of identifying two or more loops in an iterative program which can be combined for potential speedup as a result of altering memory access patterns or reducing loop overhead. The Futhark compiler's optimisation pipeline already contains a fusion module. The Futhark compiler already supports a range of fusion optimisations, by exploiting invariants in its Second Order Array Combinator (SOAC) constructs – which are semantically identical to higher order functions found in functional languages. [2] It already supports an range of such fusions, such as fusing two `map` expressions together, but does not currently support fusion between `map` and `scan` expressions.

For our project we will explore the possibility of implementing Map-Scan fusion into the Futhark compiler, and will examine the performance benefits (if any) of performing such optimisations.

2.1 Motivation

Performing global memory accesses when performing computations on a GPU, due to their weaker caching system, is often far more penalising than on a CPU. Hence, performing loop fusion on a program which is to be on a graphics card, has the "[..] potential to optimize both the memory hierarchy time overhead and, sometimes asymptotically, the space requirement" for the program as a result of improving memory access patterns and making obsolete temporary arrays [2]. Thus, the main motivation for adding `map-scan` fusion capabilities to the optimiser of the Futhark compiler, is the potential for enabling speedups for relevant Futhark programs.

2.2 Tasks

The project can be divided into three main tasks:

1. Gain an understanding of logical reasoning behind fusion optimisations on Second Order Array Combinators.
2. Read and understand the relevant parts of the Futhark compiler required to make the necessary changes in the compiler.
3. Modify all modules of the Futhark compiler necessary to implement the Map-Scan fusion itself.
4. Benchmark the modified compiler against the unmodified compiler.
5. Evaluate the results.

At first sight, these tasks look fairly straight forward. However, we expect that the main difficulties of this project lie within unforeseen roadblocks we will run into when modifying the codebase.

3 Background Information

In the past decade or so, the constant demand for more powerful CPUs has increasingly been satisfied through increasing the amount of cores rather than simply attempting to increase clock-speed or similar. This trend of multi-core processors seeks to take advantage of latent potential for performing computational tasks in parallel – with the dream being doubling performance with a doubling of the number of cores. At the same time graphics cards with GPUS containing up to thousands of cores have become mainstream, and while they have been designed for graphics rendering, their sheer degree of parallelism brings a great potential for general purpose computation.

Reasoning about, and writing programs which exploit parallelism on GPUs can be a challenge, and parallel libraries such as CUDA and OpenCL, while useful, are not necessarily productive to work with directly. Hence it is important to have languages which natively support parallel GPU execution, so that the compiler can handle many of the repeating challenges of writing parallel programs for the GPU, and expedite the process of exploiting the GPU for computational power. Futhark seeks to fill this role by supplying a parallel language wherein specific compute intensive task can be defined and executed, lightening the load for a larger program written in a more general purpose language. [3]

This section will elaborate and detail what Futhark seeks to accoplish and how it does so.

3.1 Futhark

Futhark is a "statically typed, data-parallel, and purely functional array language" [3] which aims to perform general purpose computing on GPUs (GPGPU). While Futhark is able to express imperative concepts such as inplace updates and loops, its main focus centered around the usage of functional language paradigm to express parallel computations through bulk operations using Second Order Array Combinators – or SOACs – which are semantically identical to higher order functions such as `map` and `reduce` commonly found in functional programming.

The Futhark compiler features an aggressive optimisation strategy which relies on strong functional invariants found in constructs such as SOACs.

3.2 SOACs

Both `map` and `scan` are defined as SOACs – or Second Order Array Combinators. Hence they have no free variables, take first-order functions as arguments, and output first-order functions whose domains are arrays of the domain of the input. While Futhark SOACs are semantically identical to their namesake higher-order functions found in other functional languages, working with SOACs allows for some assumptions to be made which turn out to be useful in regards to both parallisation and optimisation. In particular each SOAC can be considered as representing a specific shape of an imperative do-loop, which is used in Futhark to expedite the loop-fusion process. [4, chap. 7]

The intermediate representation of a Futhark program in the Futhark compiler does not contain arrays of tuples. Hence when a Futhark program is compiled any expression of type $[\alpha]$ where $\alpha = (\alpha_1, \alpha_2, ..., \alpha_n)$ is converted into an expression of type $([\alpha_1], [\alpha_2], ..., [\alpha_n])$. Hence we have that Futhark SOACs internally have an arbitrary amount of input arrays and output arrays. However, such notation can be cumbersome, so for the sake of clarity and ease we will notate tuples of arrays as single variables and/or types until the implementation section where it becomes relevant. For example, $b = \texttt{map}\, f\, a$ is short hand for $(b0, b1, ..., bn) = \texttt{map}\, f\ (a0, a1, ..., am)$ where ai and bj are each arrays, and each of the arrays in a tuple have the same amount of elements.

3.2.1 Map

The `map` $f\ a$ function, has the very simple definition of taking a function $f : \alpha \to \beta$ and returning a function `map` $f : [\alpha] \to [\beta]$ which applies f to every element of an input array, a. This gives us the type signature of `map`,

$$\texttt{map}\ f\ a\ :\ (\alpha \to \beta) \to [\alpha] \to [\beta].$$

And the semantic definition of `map`,

$$\texttt{map}\ f\ a\ =\ [f(a_0), f(a_1), ..., f(a_{n-1})].$$

Having no free variables, means that each result $f(a_i)$ *only* depends on the corresponding element a_i. This makes `map`s fantastic for parallelisation as once the degree of parallism reaches the size of a, `map` $f\ a$ can be potentially be computed in a single parrallel step, or c steps for a chunk size of c.

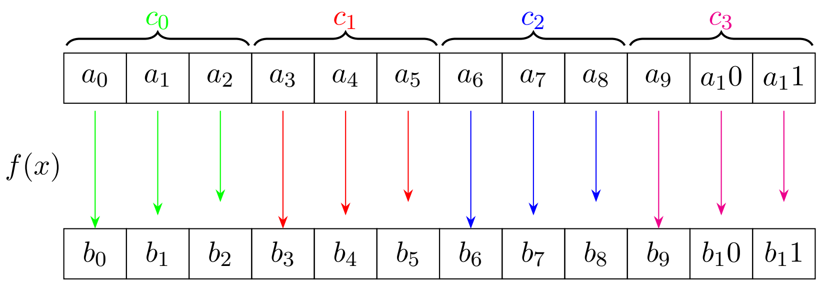

Figure 1: Parallel computation of map with chunking. Each colour represents a seperate thread.

Figure 1 displays how map can be computed in chunks. Each thread is responsible for sequentially computing `map` $f\ c_i$ across a single chunk. With no depedencies in the way, this can happen in parallel such that the 12 element list is `map`ped with just 3 sequential applications of f.

3.2.2 Scan

`scan` $\odot\ e\ a$ takes a binary, associative function $\odot : \alpha \to \alpha \to \alpha$ and returns a function `scan` $\odot : \alpha \to [\alpha] \to [\alpha]$ which computes the $\odot$ prefixes of an input array a starting with a neutral element, e. Overall, `scan` has the type signature,

$$\texttt{scan}\ \odot\ e\ a\ :\ (\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to [\alpha].$$

Computing `scan` with the function $\odot$, the array a, and neutral element e gives us,

$$\texttt{scan}\ \odot\ e\ a\ = [e \odot a_0, e \odot a_0 \odot a_1, ..., e \odot a_0 \odot ... \odot a_{n-1}].$$

However, computing such a `scan` is not as simple as with a `map` as each prefix $a_0 \odot ... \odot a_i$ obviously depends on the previous prefix $a_0 \odot ... \odot a_{i-1}$. Hence, the associativity of $\odot$ is vital as it means that this dependency does not force computation order, and partial results can be computed independently and combined.

With this, there are multiple ways in which `scan` can be computed in parallel with chunking, and Figure 2 shows a simple algorithm. It is a three phase process which step-wise involves,

1. First each thread sequentially computes the $\odot$-prefixes of their respective chunks. (c steps)

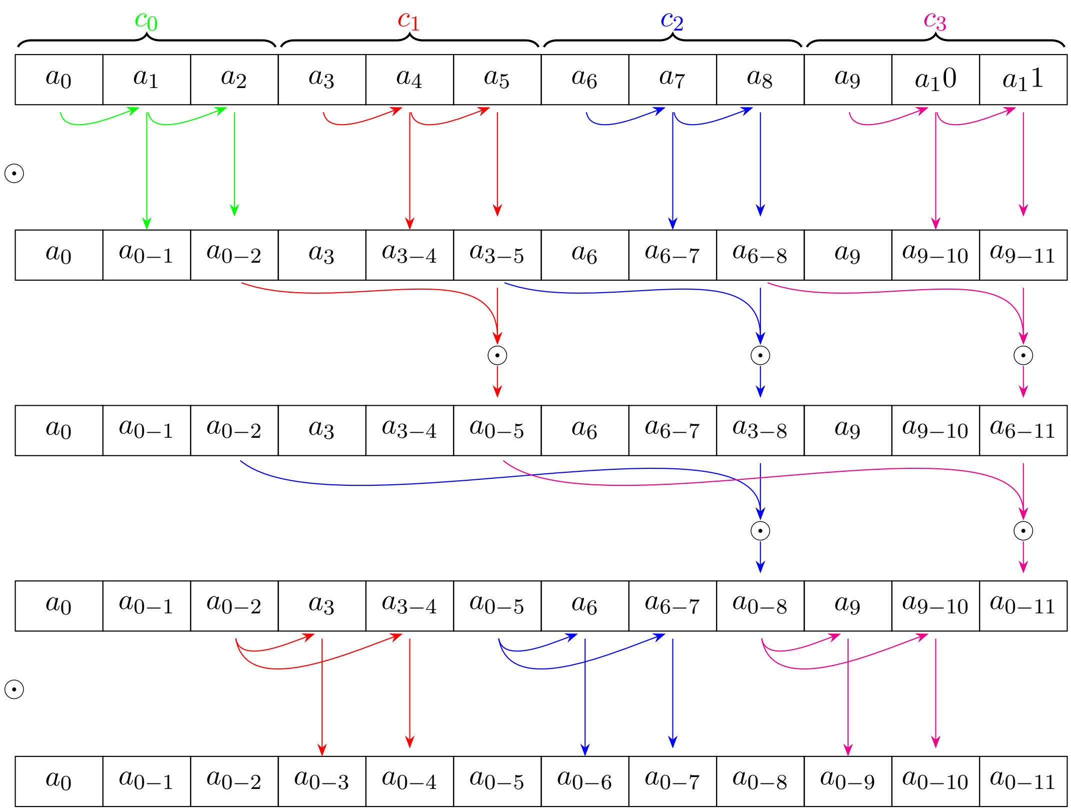

Figure 2: Parallel computation of `scan` with chunking. Each colour represents a single thread, and $a_{i-j} = a_i \odot a_{i+1} \odot ... \odot a_j$.

2. Then the final element of each chunk is recursively combined with the previous using $\odot$ with doubling stride for each recursion, such that each final element contains the a_{0-j} prefix where $j = (c * (i+1) - 1)$ for the ith chunk. ($\log(\frac{n}{c})$ steps)

3. Finally each thread adds the final element of the previous chunk to each element of the chunk which is not the tail. (c steps)

While this algorithm is not work efficient – it involves more than n applications of $\odot$ – it still has a depth of $O(2c + \log(\frac{n}{c})) < O(n)$ due to parallelism. This also shows some the idea of chunking. In the example of Figure 2, if we instead of having four threads to compute the scan, had 12 – or n threads in general – we could exploit this by simply setting the chunk-size to $c = 1$. This would forego steps 1 and 3, giving us a total depth of $O(\log(n))$.

3.2.3 Loop Conversion

Each SOAC can always be converted to an equivalent loop. For example, both $b =$ `map` $f\ a$ and and $b =$ `scan` $g\ e\ a$ can be converted into equivalent loops (see Listing 1 & 2).

Listing 1: `map` as a loop.

```
b[n];
for (i = 0; i < n; i++) {
  b[i] = f(a[i])
}
```

Listing 2: `scan` as a loop.

```
b[n];
```

```
acc = e;
for (i = 0; i < n; i++) {
  acc = g(acc, a[i])
  b[i] = acc
}
```

However, such a conversion comes with a loss of useful invariants. For example, we can always rely on a loop created from a `map` expression to have no cross-iteration dependencies. I.e. each iteration will not depend on a result computed in another iteration. This means that a `map`-loop can always be computed in parallel. On the other hand, deciding whether or not a generic loop can be computed can be computed in parallel requires sometimes complex dependency analysis.

4 Fusion

In this section the basic idea of loop fusion will be outlined, with a closer look at the two kinds of fusion implemented in this paper. The loop fusion technique gives the possibility of reducing loop overhead and increase execution speed, and the potential to optimize both the memory hierarchy time overhead and space requirements. Loop fusion is as the name implies, fusing loops together. However this requires that the loops iterate the same number of times, as that could for example make one loop try to fetch data out of bounds.

Having two loops with the same iteration range as shown in listing 5, it gives the possibility for fusion them together as one. Both loops do the work needed within the same range, and therefore it could be done in one loop as shown in listing 6. Cases where fusion will not be possible regardless of two equal length loops, will be shown later on.

By doing the fusion in this case, loop-overhead has been reduced in half, as now it is only necessary to keep track of one loop instruction.

4.1 Consumer-Producer Fusion

Producer-consumer loop fusion is slightly more complicated. In this case the two loops are connected, as in inside the producer-loop some data is being created, that is required in the consumer-loop. I.e a producer creates some data, and a consumer uses the created data, this relationship is referred to as a producer-consumer relationship.

Loop fusion of a producer-consumer relationship now gives the opportunity to fuse the functions in the 2 loops into one, i.e $f(x) \circ g(x) = f(g(x))$ as the example in listing 3 and 4 shows.

The result is reduced loop overhead, reducing in memory requirements as the output of the producer loop is no longer stored separately, but calculated within the fused result.

Listing 3: Producer-Consumer pre-fusion.

```
a[n];
b[n];
c[n];

for (int i = 0; i < n; i++) {
  b[i] = f(a[i]) // f(x) some function doing work
}

for (int i = 0; i < n; i++) {
  c[i] = g(b[i]) // g(x) some function doing work
```

```
    }
```

Listing 4: Producer-Consumer post-fusion.

```
    a[n];
    c[n];

    for (int i = 0; i < n; i++) {
        c[i] = f(g(a[i]))
    }
```

4.2 Horizontal Fusion

Horizontal fusion is a none producer-consumer fusion, i.e the two loops must have no connection with each other. Listing 5 and 6 therefore perfectly examples a horizontal fusion.

Listing 5: Pre-fusion

```

    a[n];
    b[n];

    for (int i = 0; i < n; i++) {
        a[i] = i
    }

    for (int i = 0; i < n; i++) {
        b[i] = i
    }
```

Listing 6: Post-fusion

```
    a[n];
    b[n];

    for (int i = 0; i < n; i++) {
        a[i] = i
        b[i] = i
    }
```

4.3 Fusion Hindrances

Fusing a producer loop into a consumer loop is not always valid, and deciding whether or not a producer-consumer loop pair is safely fusible requires loop dependency analysis for every statement in the consuming loop, such that dependencies can be upheld in a fused loop. For example, in Listing 7 we can see that statement `S2` depends on iteration i of `S1` and iteration $i-1$ of `S2`. Thus loop a can safely be fused together with loop b.

Listing 7: Example dependencies

```
    a[n];
loop a:
    for (int i = 0; i < n; i++) {
S1:     a[i] = f(i)
```

```
  }
  b[n];
loop b:
  for (int i = 0; i < n; i++) {
S2: b[i] = b[i - 1] + a[i]
  }
```

However, in Listing 8 we have that statement `S4` depends on iteration $i+1$ of statement `S3`. If we were to fuse loop c and loop d into a single, then we would have a situation where `S4` would depend on a future iteration of its own loop. Hence, the result of fusing loops c and d would be an invalid program, and as such it is unsafe to fuse them.

Listing 8: Example dependencies

```
  a[n];
loop c:
  for (int i = 0; i < n; i++) {
S3: c[i] = f(i)
  }
  b[n];
loop d:
  for (int i = 0; i < n; i++) {
S4: d[i] = c[i + 1]
  }
```

With larger and more complicated loops, this kind of analysis can become very sophisticated and complicated. This is why we reason about fusion through SOACs as their shapes and dependencies as loops are well known.

4.4 Fusion in Futhark

Futhark is centered around SOACs for doing operations on arrays, and you do therefore not write loops to do array functions. Loop fusion for that reason is focused on SOAC fusion. The SOACs can currently be divided up as producer consumer functions as shown below:

Table 1: Producers and Consumers in Futhark

Producers	**Consumers**
Map	Map
	Reduce
	Scan
Filter	Filter
	Redomap
	Scanomap

This brings us to the benefits of reasoning about fusion via. SOACs. Instead of being a question of “can loop a be fused with loop b?” we instead ask whether a one type of SOAC can be fused with another. For example, take `map-map` producer-consumer fusion. We know that the program,

$$\textbf{let } b = \texttt{map}\, f\, a \textbf{ in}$$
$$\textbf{let } c = \texttt{map}\, g\, b \textbf{ in}$$

can be expressed as the loops in Listing 9, Listing 9: `map` and `map` as producer and consumer

```
a[n];
b[n];
for (int i = 0; i < n; i++) {
  b[i] = f(a[i]);
}
c[n];
for (int i = 0; i < n; i++) {
  c[i] = g(b[i]);
}
```

and through dependency analysis we will find that we can fuse those loops into the loop from Listing 10.

Listing 10: Loop from Listing 9 fused as producer and consumer

```
a[n];
c[n];
for (int i = 0; i < n; i++) {
  c[i] = g(f(a[i]));
}
```

The process above is equivalent to looking at the original SOAC statements, removing the intermediate array, and performing function composition on f and g. If we do this, we get the following – which if converted to a loop gives the exact same loop as Listing 10.

$$\textbf{let } c = \texttt{map}\,(g \circ f)\; a \textbf{ in}$$

5 Map-Scan Fusion

In a typical situation involving `map` and `scan`, we will have the `map` producing some array, which is subsequently consumed by the `scan`:

$$b = \texttt{map}\; f\; a$$
$$c = \texttt{scan}\; \odot\; e\; b.$$

To demostrate the benefits of fusing these expressions, we can describe them in sequential code as two loops where the first produces the b array and the second consumes b to create c.

Listing 11: `map` and `scan` as sequential loops.

```
a[n];
...
b[n];
for (int i = 0; i < n; i++) {
  b[i] = f(a[i]);
}
c[n];
acc = e;
for (int i = 0; i < n; i++) {
  acc = g(acc, b[i]);
  c[i] = acc;
}
```

In this situation, the memory access pattern will look something like the following,

$$\texttt{load}[a_0], \texttt{store}[b_0], \texttt{load}[a_1], \texttt{store}[b_1], ..., \texttt{load}[a_{n-1}], \texttt{store}[b_{n-1}]$$
$$\texttt{load}[b_0], \texttt{store}[c_0], \texttt{load}[b_1], \texttt{store}[c_1], ..., \texttt{load}[b_{n-1}], \texttt{store}[c_{n-1}].$$

However, if the b array is not used outside the second loop, we can recalling that element of the scanned array c depend only on the previous prefixes. Thus we have that c_i will only depend on b_j for $j \leq i$ being calculated. This gives us the option of fusing the two loops into a single loop which directly calculates c from a.

Listing 12: `map` and `scan` loops fused.

```
a[n];
...
b;
c[n];
acc = e;
for (int i = 0; i < n; i++) {
  b = f(a[i]);
  acc = g(acc, b);
  c[i] = acc;
}
```

Now we have removed all accesses to b, giving us an access pattern as follows,

$$\texttt{load}[a_0], \texttt{store}[c_0], ..., \texttt{load}[a_{n-1}], \texttt{store}[c_{n-1}].$$

This is the goal of perfoming `map-scan` fusion. By fusing a `map` into a `scan` we can optimise a program by eliminating costly memory accesses. Something which is even more useful when working on a GPGPU, which generally do not have the more forgiving memory hierarchy and caching system of a CPU.

5.1 Scanomap

When looking to fuse a `map` into a `scan`, the most straight forward approach is to attempt to perform function composition on the input functions of the respective functions. Hence, turning the following

$$b = \texttt{map}\ f\ a \tag{1}$$
$$c = \texttt{scan}\ \odot\ e\, b \tag{2}$$

into,

$$c = \texttt{scan}\ \odot_f\ e\, a \tag{3}$$

where,

$$\odot_f \stackrel{\text{def}}{=} \odot \circ f$$

hence,

$$\odot \circ f = x \odot f(y)$$

would be the naive approach. However, the type signature of the resulting function

$$\odot_f\ :\ \alpha \rightarrow \beta \rightarrow \alpha$$

is not compatible with the Futhark definition of `scan`, and neither is it associative. Clearly, a different approach is needed.

The solution to this problem is to exploit how Futhark uses chunking. Since the associativity of the `scan` operator only comes into play during the parallel phase of computation, we can distiguish between how each chunk is computed sequentially and how chunks are joined.

To do so, we must expand the list of SOACs with the internal `scanomap` function. `scanomap` is semantically similar to `scan`, however it takes two function parameters – an associatve scanning function meant for parallelly scanning across chunks, and a sequential folding function meant for scanning within a chunk. Scanomap has the following type signature,

$$\texttt{scanomap}\ \odot\ \odot_f\ e\ a\ :\ (\alpha \to \alpha \to \alpha) \to (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to [\alpha].$$

and can be considered semantically similar to performing a left fold with the $\odot_f$ function

$$\texttt{scanomap}\ \odot\ \odot_f\ e\ a\ = [e \odot_f a_0, (e \odot_f a_0) \odot_f a_1, ..., ((e \odot_f a_0) \odot_f ...) \odot_f a_{n-1}]$$

which also corresponds to the chuck-wise computation of `scanomap`. The scanning operator can then be used to join two chunks,

$$\texttt{scanomap}\ \odot\ \odot_f\ e\ (a +\!\!+\ b)\ = [a'_0, a'_1, ..., a'_{n-1}] +\!\!+\ [b'_0 \odot a'_{n-1}, b'_1 \odot a'_{n-1}, ..., b'_{n-1} \odot a'_{n-1}]$$

where

$$\begin{aligned} \texttt{scanomap}\ \odot\ \odot_f\ e\ a\ &= [a'_0, a'_1, ..., a'_{n-1}] \\ \texttt{scanomap}\ \odot\ \odot_f\ e\ b\ &= [b'_0, b'_1, ..., b'_{n-1}]. \end{aligned}$$

For `scanomap` to be used in facilitating `map-scan` fusion, we first observe that we have the following equivalence,

$$\texttt{scan}\ \odot\ e\ a \equiv \texttt{scanomap}\ \odot\ \odot\ e\ a.$$

Hence we can freely turn any regular `scan` into an equivalent `scanomap`. If we once again take a look at our previous example we can see how we can now fuse a `map` into a `scan` using the `scanomap` construction:

$$\begin{aligned} b &= \texttt{map}\ f\ a \\ c &= \texttt{scan}\ \odot\ e\ b \end{aligned}$$

Firstly, we turn the `scan` into an equivalent `scanomap`,

$$c = \texttt{scanomap}\ \odot\ \odot\ e\ b$$

and then compose a folding function, $\odot \circ f = \odot_f$, from the mapping function and the scanning operator and discard the intermediate b list, to arrive at an equivalent `scanomap`,

$$c = \texttt{scanomap}\ \odot\ \odot_f\ e\ a.$$

Any producer `map` can be fused into a consuming `scan` in this manner.

5.2 Necessary Conditions

Fusion is not always viable as there are some situation where fusion cannot or should not happen. Figure 3 shows six cases where fusion transformation should not happen for some reason.

Case 1: Fusion across an in-place update is not possible – i.e. $a[i] = k$. When fusing the producer array is not created as before, but a part of the computation in the resulting fused SOAC.

Case 2: Not all combinations of SOACs are fusable.

Case 3: Fusion across a loop or a SOAC lambda would duplicate the computation and potentially change the time complexity of the program. In this case instead of calculating variable x once, a fusion would result in x being calculated in each of the loops.

Case 4: When the array x produced by a SOAC is consumed by two other SOACs located on the same execution path, the computation will be duplicated.

Case 5: If array x produced by SOAC is used other then input to another SOAC fusion is not allowed.

Case 6: If two arrays created by a SOAC is used as input in more then one SOAC the computaion is duplicated, and fusion is therefore not allowed.

However there are special cases for scanomap, as the result array x of the producer SOAC in the scanomap transformation, also will be returned by the scanomap in cases where other SOACs consumes array x.

5.3 Fusing Scanomap

Once a `map` and a `scan` has been combined into a `scanomap`, it raises the question of how can we further fuse this new construct? This section details which SOACs `scanomap` can fuse with, and how.

Map-Scanomap Fusion Fusing a `map` as a producer into a consumer `scanomap` is a simple process. Given a `map` with function g and a `scanomap` with the folding function $\odot_f$, we compose a new folding function $x \odot_{f \circ g} y = x \odot f(g(y))$. This is illustrated below,

$$
\begin{aligned}
&b = \texttt{map}\ g\ a \\
&c = \texttt{scanomap}\ \odot\ \odot_f\ e\ b \\
&\Downarrow \\
&c = \texttt{scanomap}\ \odot\ \odot_{f \circ g}\ e\ a.
\end{aligned}
$$

Scanomap-Scanomap Fusion Horizontally fusing a `scanomap` with another `scanomap`, requires that theres exists no dependency between the two SOACs – i.e. there exists no producer-consumer relationship between the two. In theses cases, it may still prove beneficial to perform fusion – even though there is no direct optimisation – to enable more fusion. This

```
// Case 1: don't fuse if it
//moves an array use across
//its in-place update point
let x    = map2(f, a) in
let a[1]= 3.33        in
let y    = map2(g, x) in ..

// Case 3: don't fuse from
//outside in a loop (or λ)
let x = map2(f, a)   in
loop(arr) = for i < N do
  map2(op +, arr, x)

// Case 5: don't fuse if x
//used outside SOAC inputs
let x = map2(f, a) in
let y = map2(g, x)   in
    x[i] + y[i]
```

```
// Case 2: not all SOAC
//combinations are fusable
let x = filter2(c1, a) in
let y = filter2(c1, b) in
let z = map2   (f,x,y) in..

// Case 4: don't fuse in 2
//kernels sharing a CF path
let x = map2(f, arr)  in
let y = if c then map2(g1,x)
             else map2(g2,x)
in let z = map2(h, x) in ..

// Case 6: x & y used exactly
//once but still don't fuse
let (x,y) = map2(f, a)   in
let u = reduce2(op +,0,x)in
let v = reduce2(op *,1,y)..
```

Figure 3: Don't Fuse Cases [2, page 6]

kind of fusion is performed by in a way concatenating input, output, and both functions for the input `scanomaps`, which gives the following,

$$\begin{aligned} c &= \texttt{scanomap}\ \odot_1\ \odot_{1f}\, e_1\, a \\ d &= \texttt{scanomap}\ \odot_2\ \odot_{2g}\, e_2\, b \\ &\Downarrow \\ (c, d) &= \texttt{scanomap}\ \odot'\ \odot'_{fg}\, (e_1, e_2)\, (a, b). \end{aligned}$$

Where $\odot'(e_1, e_2, x, y) = (e_1 \odot x, e_2 \odot y)$ and $\odot'_{fg}(e_1, e_2, x, y) = (e_1 \odot f(x), e_2 \odot f(y))$. In this way, the resulting `scanomap` now describes the computations of both input `scanomaps`.

6 Implementation

This section will detail the specifics of performing `map-scan`, and other `scanomap` fusions, as well as detailing the process of analysing a program for valid fusions.

6.1 Program State

There are a series of conditions for the fusion transformation can happen. As well as a number of cases where a fusion should not happen. The fusion transformation assumes a normalized program, with the following properties: [2, Figure 5, page 4]

- No tuple type can appear in an array or tuple type, i.e., flat tuples,
- unzip has been eliminated, zip has been replaced with assertZip, which verifies either statically or at runtime that the outer size of zip's input matches, and finally, the original SOACs (map) have been replaced with their tuple-of-array version,
- tuple expressions can appear only as the final result of a function, SOAC, or if expression, and similarly for the tuple pattern of a let binding, e.g., a formal argument cannot be a tuple,
- e_1 cannot be a let expression when used in let $p = e_1$ in e_2,
- each if is bound to a corresponding let expression, and an if's condition cannot be in itself an if expression, e.g.,
  $a +$ if(if c_1 then e_1 else e_2) then e_3 else $e_4 \rightarrow$
  let $c_2 =$ if c_1 then e_1 else e_2 in
  let $b =$ if c_2 then e_3 else e_4 in $a + b$
- function calls, including SOACs, have their own let binding, e.g.,
  reduce2(f, a) $+\ x \Rightarrow$ let $y =$ reduce2(f, e, a) in $y + x$,
- all actual arguments are vars, e.g., $f(a + b)) \Rightarrow$ let $x = a + b$ in $f(x)$.

The properties listed above is reached by going through the different stages in the compiler pipeline, as shown in figure 4. Type checking first so that any errors will refer to the actual names given by the programmer, then moving on to tuple transformation that flattens all tuples and converts arrays of tuples to tuples of arrays. Tuple and let normalisation to optimise the code based on code recognition (e.g $f(a + b)) \Rightarrow$ let $x = a + b$ in $f(x)$).

When the program has reached the properties mentioned it enters the enabling optimisations loop, with aggressive in lining (i.e., building the call-graph and moving non recursive functions into the function calling it.). Copy propagation e.g.,

$$y = x$$

$$z = 5 * y$$

Would become

$$z = 5 * x$$

and constant folding where the compiler evaluating constant expressions at compile time. After dead code removal the process repeats unless a fixed point is reached, if the fixed point is reached the program is ready for possible fusion transformations.

6.2 Bottom Up Analysis

Futhark's fusion module gathers fusible SOACs by using a bottom-up analysis pass of the program's AST. This is done by traversing the AST depth first, while using a set of environments and data structures, to among other things keep track created kernels and the production and consumption of arrays. [2]

These are structures and environments are filled out during the depth first traversal of the AST, such that when analysing a specfic SOAC in the program it is known which SOACs consume its outputs, and whether in-place update in between and so on.

For example, when analysing the program found in Figure 5 and the looking at the `map` statement, two simple look-ups will show that while b is consumed by the `scan`, it is consumed by an in-place updated previously. Hence, fusion is not possible.

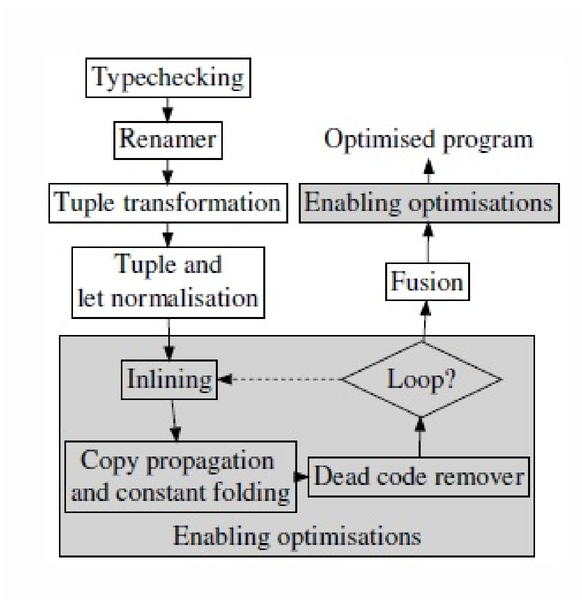

Figure 4: Compile pipeline [2, page 4]

$$\begin{array}{l}
\quad\vdots \\
\textbf{let } b = \texttt{map}\ f\ a \textbf{ in} \\
\textbf{let } b[1] = 4 \textbf{ in} \\
\textbf{let } c = \texttt{scan}\ \odot\ 0\ b \textbf{ in} \\
\quad\vdots
\end{array}$$

Figure 5: Infusible program snippet.

Fusion is then attempted along the way whenever the fusion module meets a new SOACs during its analysis. Consequently, we have that fusion is also performed bottom-up, such that the compiler will prioritize potential fusion by how "deep" the candidates occur in the program.

6.3 Map-Scanomap Fusion

Having identified a single `map` producer and a `scanomap` consumer to perform fusion on, the fusion itself may seem fairly simple. However, this is not entirely the case in most situations, as especially function composition isn't as straight forward as it can seem. There are also two factors which complicate this process:

- As previously mentioned, all SOACs in Futhark work with tuples of arrays. This means that a $b = \texttt{map}\ f\ a$ is actually short hand for $(b0, b1, .., bn) = \texttt{map}\ f\ (a0, a1, ..., an)$, and not all members of the same tuples are consumed, or produced, in the same place.
- Certain SOACs in Futhark support map-out arrays – i.e. array-outputs from a fused `map` which are still output after fusion. This is also supported by `scanomap`.

This section will go through the algorithm for fusing a single `map` into a `scanomap` – and by extension the algorithm for fusing a `map` into a `scan`.

The function inputs of SOACs in Futhark consist of a set of parameters, a set of `let` bindings, and a body which defines the function output. When looking to create a new folding function for the resulting `scanomap` construct, composed entirely from the functions found in the to-be-fused `map` and `scanomap`. Firstly, we will find the new parameter set for

$$
\begin{aligned}
&f(a0, a1, a2, a3) = && (1)\\
&\quad \textbf{let}\ (x1, x2) = g(a0, a1)\ \textbf{in} && (2)\\
&\quad \textbf{let}\ y1 = a2 + a3\ \textbf{in} && (3)\\
&(x1, x2, y1) && (4)
\end{aligned}
$$

Figure 6: Example of a SOAC lambda function.

the new folding function. Given a program, wherein a `map` is to be fused with a `scanomap` into a new `scanomap`:

$$
\begin{aligned}
b &= \texttt{map}\ f\ a\\
d &= \texttt{scanomap}\ \odot\ \odot_g\ ne\ c.
\end{aligned}
$$

Where a, b, c, d are all tuples of arrays, and ne is a tuple of neutral elements.

Fusing these two SOACs into a `scanomap` becomes a three step process of

1. determining the parameter list of the output `scanomap`,
2. determining map-out arrays, and
3. composing a new folding function.

Afterwards a new `scanomap` can then be constructed which can replace the input `map` and `scanomap`.

Parameters We have that `map` and `scanomap` are to be fused, so we must also have $(b0, b1, ..., bn) \cap (c0, c1, ..., cn) \neq \emptyset$. However we do not neccessarily have $(b0, b1, ..., bn) = (c0, c1, ..., cn)$. The parameters for our fused product must then become all of the array input parameters from the `map` as well as the array input parameters from the `scanomap` which aren't produced by the `map`. Hence, our new list of parameters become,

$$\text{New-Params} = (a \cup c)/b.$$

Map-Out Arrays Next is to find any map-out arrays which need to be returned. These can potentially consist of some subset of the output arrays of both the `map` and the `scanomap`. The map-out arrays already carried by the input `scanomap` are by convention placed last in the list of outputs, so they correspond to the $\texttt{len}(d) - \texttt{len}(ne)$ last members of d.

$$\text{map-out}_{scanomap} = \texttt{drop}(\texttt{len}(ne), d)$$

The map-out arrays from the input `map` are found by using the fusion environment – which keeps track of where arrays are consumed – through the `unfus` function which filters out any arrays which are not consumed later in the program.

$$\text{map-out}_{map} = \texttt{unfus}(b)$$

Combining these two tuples will then give us the full set of map-out arrays for the fused product,

$$\text{map-out} = m_{out} = \texttt{drop}(\texttt{len}(ne), d) \mathbin{++} \texttt{unfus}(b).$$

Function Composition Composing the folding function for the resulting `scanomap` from the input `map`'s f function, and the input `scanomap`'s $\odot_g$ function is a question of combining their bindings such that the result of the f function is computed first and bound such that it can be used to compute $\odot_g$.

We assume that input functions f and $\odot_g$ will consist of a list of let-bindings, binding names to some arbitrary expressions which can reference the function parameters as well as other visible binding. These are followed by a body which consists solely of the variables or constants to be returned.

Figure 7 shows generalized functions for both input SOACs. Here a_i, b_i, c_i, and d_i are the tuple consisting of the ith elements of the arrays contained in the a, b, c, or d tuple respecively, and x_i and y_i correspond to arbitrary names used in let bindings. It's worth noting that b_i and d_i denote the bodies of f and $\odot_g$ respectively, and that we have $b_i \subset c_i$.

$$\begin{array}{ll}
f(a_i) = & \qquad \odot_g\, (ne, c_i) = \\
\quad \textbf{let } x_1 = e^f_1 \textbf{ in} & \qquad \quad \textbf{let } y_1 = e^g_1 \textbf{ in} \\
\quad \textbf{let } x_2 = e^f_2 \textbf{ in} & \qquad \quad \textbf{let } y_2 = e^g_2 \textbf{ in} \\
\qquad \vdots & \qquad \qquad \vdots \\
\quad \textbf{let } x_n = e^f_n \textbf{ in} & \qquad \quad \textbf{let } y_n = e^g_n \textbf{ in} \\
b_i & \qquad d_i
\end{array}$$

Figure 7: The SOAC functions f and $\odot_g$ from the input `map` and `scanomap`.

Combining these two functions into a single new folding function $\odot_g \circ f$ (or $\odot_{g \circ f}$) is then a question of first applying all of the bindings from the f function, creating a new binding which binds the body of f to the corresponding parameters of $\odot_g$, and then on to that appending the bindings of $\odot_g$.

$$
\begin{array}{l}
\odot_g \circ f(ne, ((a \cup c)/b)_i) = \\
\quad \textbf{let } x_1 = e_1^f \textbf{ in} \\
\quad \textbf{let } x_2 = e_2^f \textbf{ in} \\
\qquad \vdots \\
\quad \textbf{let } x_n = e_n^f \textbf{ in} \\
\quad \textbf{let } c_i' = b_i' \textbf{ in} \\
\quad \textbf{let } y_1 = e_1^g \textbf{ in} \\
\quad \textbf{let } y_2 = e_2^g \textbf{ in} \\
\qquad \vdots \\
\quad \textbf{let } y_n = e_n^g \textbf{ in} \\
(d_i/m_{out_i}) +\!\!+\, m_{out_i}
\end{array}
$$

Figure 8: Resulting function $\odot_{f \circ g}$.

The body of $\odot_{f \circ g}$ then becomes the outputs of $\odot_g$ minus any map-out arrays appended with the new map-out arrays as found above. This is displayed in Figure 8, where $c_i' = b_i'$ is short hand for $c_i \cap b_i = b_i \cap c_i$, or assigning the elements of c_i which refer to elements of b_i to the values refered to by the respective elements of b_i.

We can then replace the original `map` and `scanomap` with a new `scanomap`,

$$(d/m_{out}) +\!\!+\, m_{out} = \texttt{scanomap} \odot \odot_{f \circ g}\, ne\ ((a \cup c)/b)\,.$$

6.4 Scanomap-Scanomap Fusion

Having two `scanomap` in a futhark program with no dependencies between the two SOACs, and equal length input, the program computation time can be optimised using horizontal fusion.

This section will go through the algorithm for fusing two `scanomap` into one `scanomap`. The fusion algorithm have two steps:

1. Composing a new assosiative operation
2. Composing a new folding function.

$$
\begin{aligned}
c &= \texttt{scanomap}\ \odot_1\ \odot_{1f}\, e_1\, a \\
d &= \texttt{scanomap}\ \odot_2\ \odot_{2g}\, e_2\, b \\
&\Downarrow \\
(c, d) &= \texttt{scanomap}\ \odot'\ \odot'_{fg}\, (e_1, e_2)\, (a, b).
\end{aligned}
$$

Parameters With two `scanomap` to be fused, the goal is to join the computations within the same loop. Hence a does not have to equal b, but the the length of the input arrays must be the same. Our new list of parameters simply become:

$$\text{New-Params} = (a, b)$$

New folding function and associative operation As the two `scanomap` are independent of each other the composition of the new fused folding function is fairly easy. Figure 9 show the pre-fusion folding functions for both scanomap, running sequentially after each other. Fusing them is simply adding the additional parameters and adding the function body of one of them into the other as shown in figure 10.

$$
\begin{aligned}
&\odot_f\,(ne_1, a_i) = \\
&\quad \textbf{let}\ x_1 = e_1\ \textbf{in} \\
&\quad \textbf{let}\ x_2 = e_2\ \textbf{in} \\
&\qquad \vdots \\
&\quad \textbf{let}\ x_n = e_n\ \textbf{in} \\
&c_i
\end{aligned}
\qquad\qquad
\begin{aligned}
&\odot_g\,(ne_2, b_i) = \\
&\quad \textbf{let}\ y_1 = e_1\ \textbf{in} \\
&\quad \textbf{let}\ y_2 = e_2\ \textbf{in} \\
&\qquad \vdots \\
&\quad \textbf{let}\ y_n = e_n\ \textbf{in} \\
&d_i
\end{aligned}
$$

Figure 9: The SOAC functions $\odot_f$ and $\odot_g$ from the two scanomap

$$
\begin{aligned}
&\odot'_{fg}\ (ne1, ne2, a_i, b_i) = \\
&\quad \textbf{let}\ \ x_1 = e_1\ \ \textbf{in} \\
&\quad \textbf{let}\ \ x_2 = e_2\ \ \textbf{in} \\
&\qquad\quad \vdots \\
&\quad \textbf{let}\ \ x_n = e_n\ \ \textbf{in} \\
&\quad \textbf{let}\ \ y_1 = e_1\ \ \textbf{in} \\
&\quad \textbf{let}\ \ y_2 = e_2\ \ \textbf{in} \\
&\qquad\quad \vdots \\
&\quad \textbf{let}\ \ y_n = e_n\ \ \textbf{in} \\
&(c_i, d_i)
\end{aligned}
$$

Figure 10: Fused function $\odot'_{fg}$

In the same manor the assosiative operations is fused into one, enabling the computation of two scanomaps to be done within one loop.

Map-Out Arrays If the map-out arrays in the `scanomaps` should be returned, it is done in the same manor as explained in the consumer-producer fusion of map and scan in section Map-Scanomap Fusion. Because the horizontally fused `scanomaps` are computed in precisely the same way, just "sequentially" in one loop it does not change anything regarding map-out arrays.

6.5 Other Fusion

7 Benchmarking

In this section we will present the programs used to benchmark the scanomap implementation, and the results retrieved.

Map-Scan Fusion benchmark Benchmarking the computation time improvement on map-scan fusion to scanomap is done with this simple Futhark program.

Listing 13: SimpleScanomap

```
fun ([int]) main([int] inp) =
  let a = map(+10, inp)
  let b = scan(+, 0, a) in
  (b)
```

The benchmark program contains a simple map producer and scan consumer, with the scanomap implementation the resulting program will contain one scanomap. By benchmarking with and without the scanomap implementation, the results will show us any increase/decrease in computation time. We run the benchmark program with increasingly large amount of data, and 50 times per data amount to try and negate any large deviations. Additionally this program will be benchmarked on both the GPU and CPU.

GPU: The result from the benchmark show that the scanomap implementation, enables the same computations to be done faster then pre-scanomap Futhark for this program. In figure 11 the red line represents the pre-scanomap Futhark, and the blue line with scanomap implemented. In the entire input size range (1000-90000000) the program runs faster with the scanomap implementation.

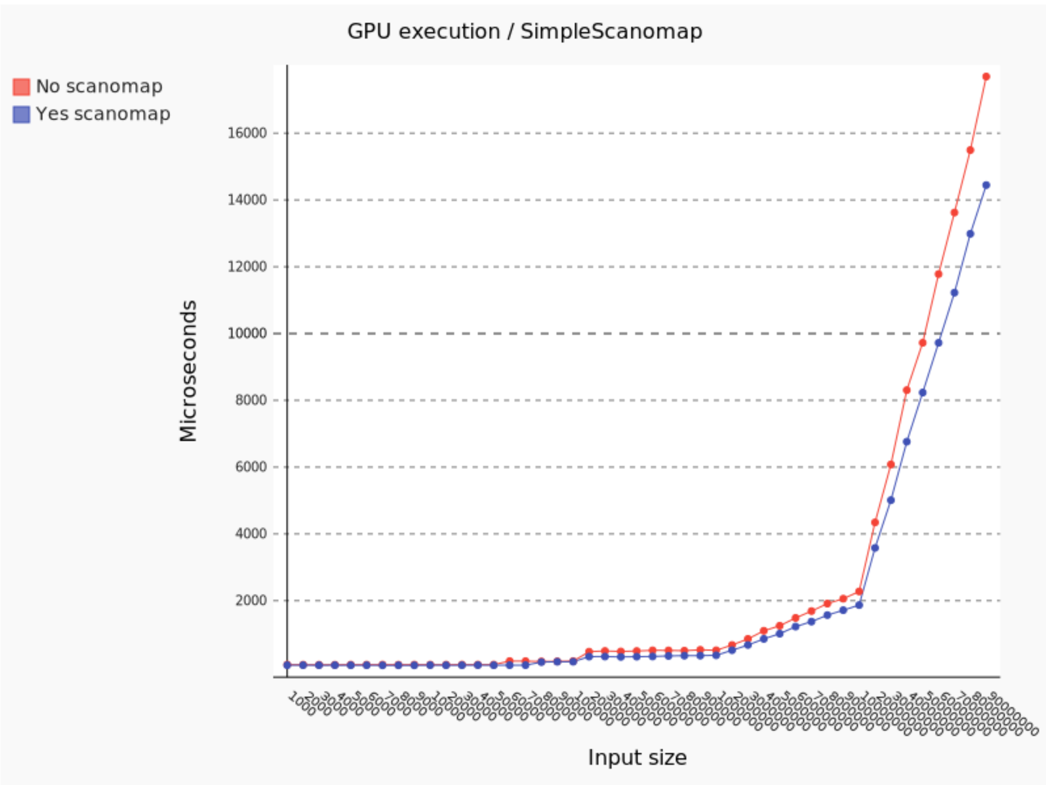

Figure 11: Computation time for Map-Scan Fusion test

Figure 12 shows the speedup with scanomap, and the general result shows that with the scanomap implementation the computation is around 1.35 times faster. For data larger then 5,000,000 the speedup seems to stay around 1.2 times faster. We do not know what caused the two high spikes at 60,000 and 70,000 data sizes, and will not investigate any cause.

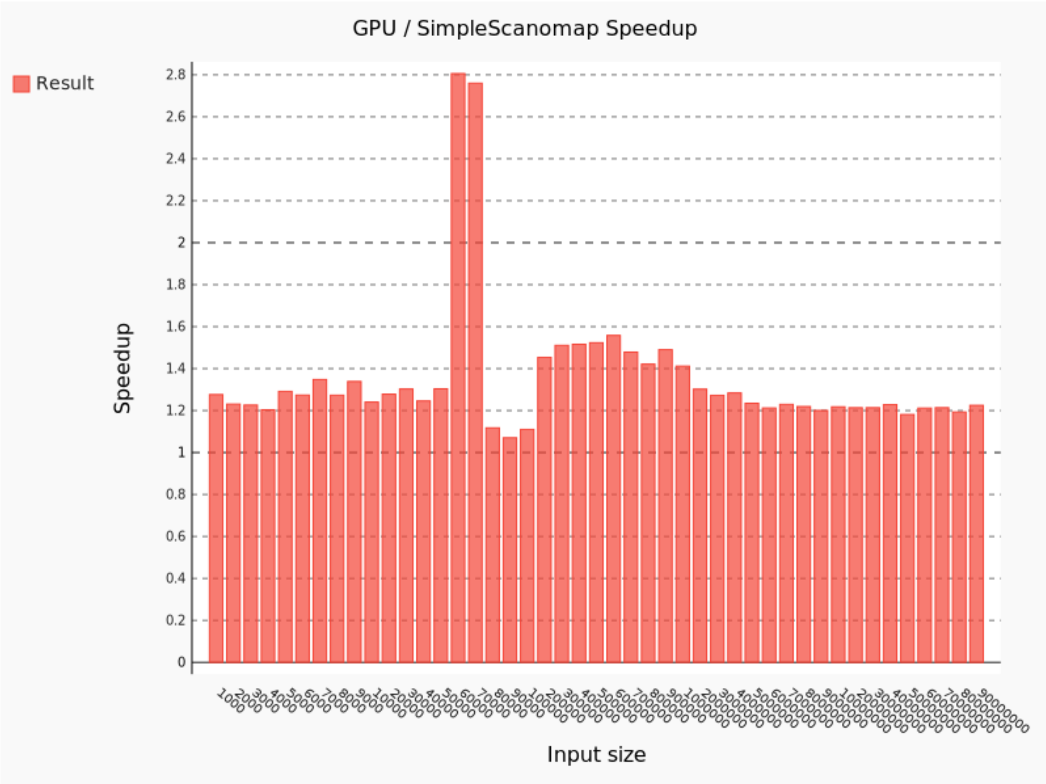

Figure 12: Speedup chart

CPU: When running the same test on the CPU, we see that the scanomap is no longer always faster. Below input size of 20000 the computation time is faster without scanomap as seen in figure 13. Higher then 20000 and especially above 100000 scanomap gives an significant increase in speedup.

Radix sort: To benchmark the scanomap implementation with a more complicated program, that does more work, we benchmarked the radix sort futhark program (Appendix A). The results displayed in figure 14 shows considerable speedup, with a average of 1.32 times faster then without scanomap.

Scanomap-Scanomap fusion: To test the effect of implementing the horizontal fusion of scanomap-scanomap, we have made a futhark program and tested it in three ways:

- Without any fusion.
- With scanomap fusion.
- With scanomap fusion and scanomap-scanomap fusion.

Listing 14: Scanomap-scanomap benchmark program

```
fun [int, n] main([int, n] inp) =
```

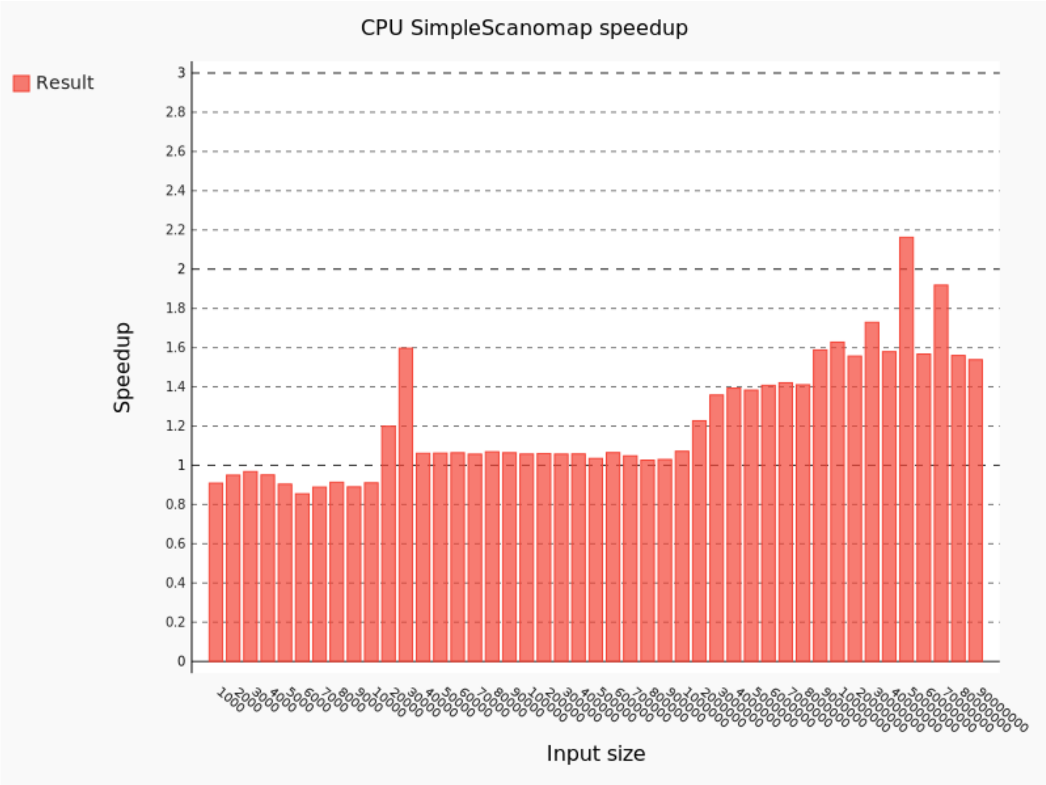

Figure 13: Speedup chart

```
    let a  = map(+10, inp) in
    let b1 = scan(+, 0, a) in
    let a2 = map(+1, a) in
    let b2 = scan(+, 0, a2) in
    map(fn int (int x,int y) => x + y, zip(b1,b2))
```

The results in figure 15 shows a speedup of average 1.19 times faster by using scanomap fusion, illustrated by the red bars. However by adding the scanomap-scanomap fusion illustrated by the blue bars, the results are an average of 1.56 times faster computation.

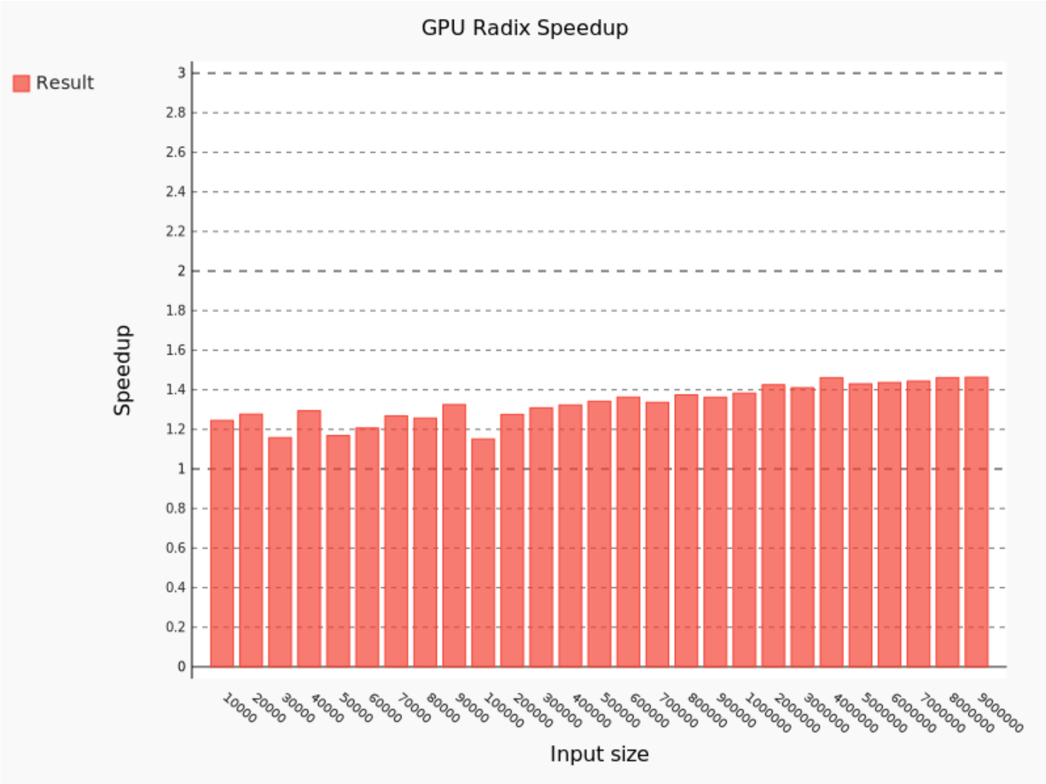

Figure 14: Radix-Speedup chart

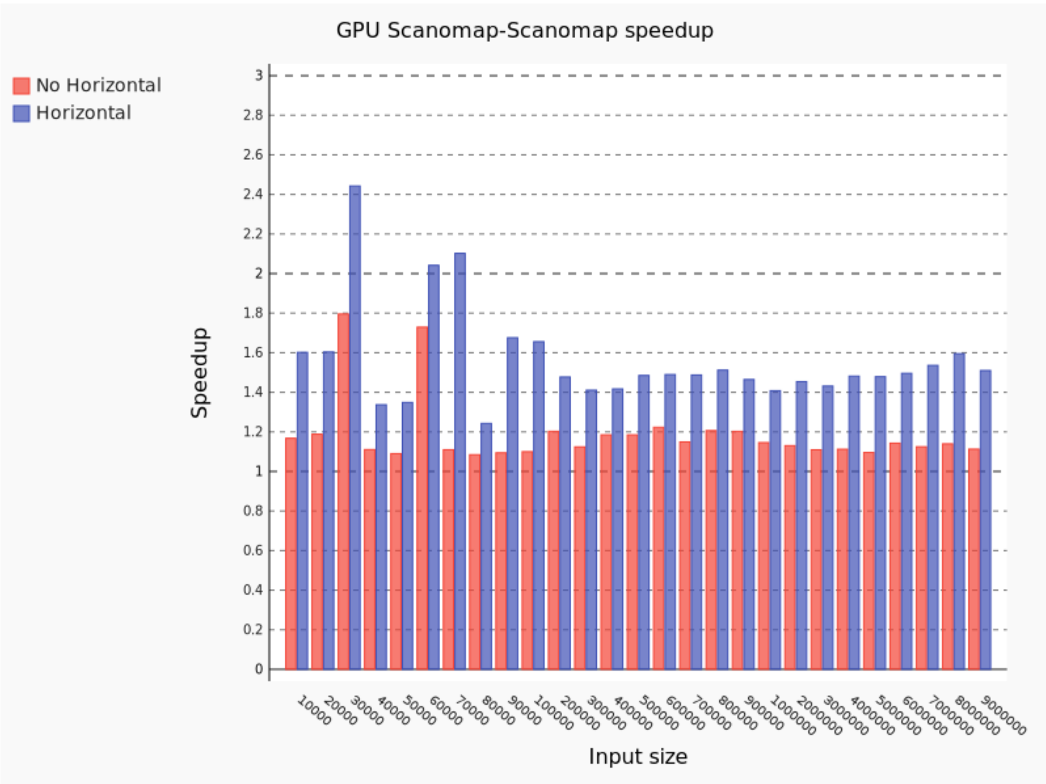

Figure 15: Speedup chart for scanomap-scanomap fusion benchmark

8 Conclusion

References

[1] Futhark language documation. `http://futhark.readthedocs.org/en/latest/index.html`. Accessed: 02-03-2016.

[2] Troels Henriksen and Cosmin Eugen Oancea. A t2 graph-reduction approach to fusion. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 47–58, New York, NY, USA, 2013. ACM.

[3] Futhark website. `http://futhark-lang.org`. Accessed: 02-03-2016.

[4] Troels Henriksen. Exploiting functional invariants to optimise parallelism: a dataflow approach. Master's thesis, DIKU, Denmark, 2014.

9 Appendix

9.1 A

Listing 15: Test program

```
fun [u32, n] main([u32, n] xs) =
  radix_sort(xs)

fun [u32, n] radix_sort([u32, n] xs) =
  loop (xs) = for i < 32 do
    radix_sort_step(xs, i)
  in xs

fun [u32, n] radix_sort_step([u32, n] xs, i32 digit_n) =
  let bits = map(fn i32 (u32 x) => i32((x >> u32(digit_n)) & 1u32),
      xs)
  let bits_inv = map(fn i32 (i32 b) => 1 - b, bits)
  let ps0 = scan(+, 0, bits_inv)
  let ps0_clean = map(*, zip(bits_inv, ps0))
  let ps1 = scan(+, 0, bits)
  let ps0_offset = reduce(+, 0, bits_inv)
  let ps1_clean = map(+ ps0_offset, ps1)
  let ps1_clean'␣=␣map(*,␣zip(bits,␣ps1_clean))
␣␣let␣ps␣=␣map(+,␣zip(ps0_clean,␣ps1_clean'))
  let ps_actual = map(fn i32 (i32 p) => p - 1, ps)
  in write(ps_actual, xs, copy(xs))
```