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`

May 24, 2016

Contents

1 Abstract

2 Introduction

NOTE: the contents of this section is lifted from our synopsis and is probably placeholder
The Futhark language is a functional programming with which the main idea is to allow for the expression of sufficiently complex programs while keeping complexity to a level where programs can be aggressively optimised and have their parallelism exploited [1].

The Futhark compiler already supports a range of fusion optimisations [2], but does not currently support fusion between `Map` and `Scan` statements.

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

Fusion has the "[..] potential to optimize both the memory hierarchy time overhead and, sometimes asymptotically, the space requirement" [2]. Hence the main motivation for adding Map-Scan fusion capabilities to the optimiser of the Futhark compiler, is the potential for enabling performance increases for some 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.

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

There are problems/calculations that gets to a size where normal sequential programming involving consecutive execution of processes, will reach a computation time unsatisfying for the intended users. In some cases these calculations can be done much faster with parallel programming. Parallel programming is where many calculations are carried out simultaneously, with the idea of dividing a problem into smaller sub problems solved at the same time. Parallel and sequential programming are not mutually exclusive, in the sense that if you use parallel programming, you cant use sequential programming, in many cases they are used together. Parallel programming can be done on the CPU with its multiple cores, but when possible and advantages it is much better to harness the thousands of cores in the GPU - graphics processing units. The GPU is no longer only used to do graphical calculations, but also General-purpose computing, GPGPU (General-purpose computing on graphics processing units).

To do GPGPU, Hyperfit a joint research center addressing the simultaneous challenges of high transparency, high computational performance and high productivity in finance, employing an integrated approach of financial mathematics, domain specific languages, parallel functional programming, and high-performance systems [?] created Futhark.

3.1 Futhark

As described in Troels Henriksens master thesis [3, The $\mathcal{L}_0$ language, p. 8] the language $\mathcal{L}_0$ later renamed Futhark is in a sense "sufficient", in that it is Turing-complete, and can express imperative style loops with do-loops. However Futhark is ment to use second-order array combinations (SOACs) to do bulk operations on arrays instead of using the do-loops. In this sections the reasoning behind using SOACs will be explained by showing the difference in their computation when done sequentially vs. parallelly.

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. Furthermore, in Futhark, these array inputs and outputs are tuples of arrays, and not arrays of tuples. 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 loop-fusion. [3, chap. 7]

3.2.1 Map

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

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

bAnd 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.

3.2.2 Scan

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

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

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.

4 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`:

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

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

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

Recalling that element of the scanned array c depend only on the previous prefixes, we have that c_i will only depend on b_j for $j \leq i$ being calculated. This means that, ideally, we can interleave this access pattern without breaking any dependencies – giving us a pattern more like this:

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

In this case, all accesses to b become wholly uneccessary and can be removed, giving us:

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

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.

4.1 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,

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

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

$$\odot_f\ :\ \alpha \to \beta \to \alpha$$

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.

4.2 Necessary Conditions

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 1. 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 $f(x)$).

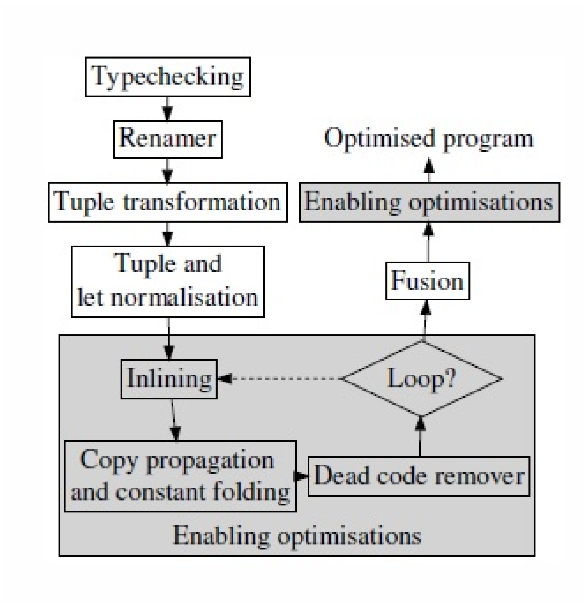


Figure 1: Compile pipeline [2, page 4]

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. Reaching the fusion stage does not guarantee that a fusion transformation can be done, as shown in figure 2 there are 6 cases where a fusion transformation should not happen.

Case 1: Fusion across a in-place update is not possible. 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.

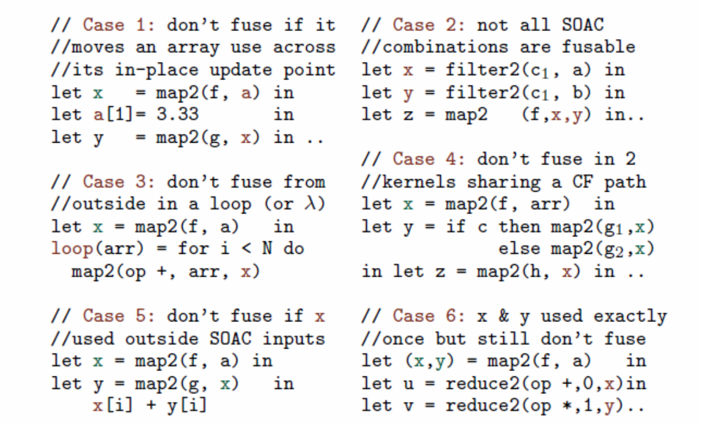
4.3 Fusing Scanomap

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,

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

4.4 Fusion Strategy/Example

Walk through the entire fusion process concisely - or similarly summarize the relevant T2 Graph reduction fusion paper rooted in scan. Show the fusion of an example instance of maps and scans through dependency graphs etc.



```
// 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 2: Don't Fuse Cases [2, page 6]

5 Implementation

What is the state of the Futhark codebase at project start. How is Scan currently handled. What is already there, and what do we need to implement.
Our solution is closely related to how Redomap fusion is handled. How does it differ.
What parts of the code have we touched and why.
Describe how different parts of the Map-Scan fusion is done - e.g. describe how function composition is implemented and so on.

6 Benchmarking and Testing

How have we tested our implementation. Does it work? Why?
How does the performance of a fused program compare with a non-fused program both sequentially and parallelly. Why?

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