Universiteit Utrecht

Graduate School of Natural Sciences

Computing Science

Optimal Fusion in Data-Parallel Languages

From Diagonal Fusion to Code Generation

Author
D.P. van Balen

July 19, 2020

First supervisor
Dr. T.L. McDonell

Second supervisor
Prof. dr. G.K. Keller

Abstract

This thesis investigates the necessary steps to introduce optimal fusion into a compiler for a data-parallel, array based language. Existing literature includes various practical fusion methods, which are based on local rewrites and do not consider the global optimal solution, and methods that are able to identify the optimal fusion clustering. There are no methods that use this optimal clustering to achieve optimally fused code, whilst preserving parallelism and type-safety. In this thesis, we expand upon the latter methods and ensure that the resulting fusion system preserves parallelism and type-safety. The proposed method consists of an *integer linear programming* formulation to identify the optimal clustering, an *explicit fusion AST* to encode this clustering in the *abstract syntax tree*, and an *explicitly loopy intermediate representation* to generate fused code.

In the process of developing this system, we formalise the concepts of *vertical* and *horizontal* fusion, and introduce *diagonal* fusion. These concepts are used in the explicit fusion AST, and allow us to preserve type-safety throughout the compiler, amounting to a partial proof of correctness.

We demonstrate that this approach is viable through a partial implementation in the compiler for Accelerate, an embedded language in Haskell.

Contents

List of Figures

Acknowledgements

I would like to thank:

My supervisors Trevor McDonell and Gabriele Keller, for their guidance and countless reviews.

My mother Judit Hódos, for her support and help with the title page.

Maartje, for your patience and advice.

1 Introduction

Consider the following Haskell function:

```
h :: [a] -> ([a], [a])
h xs = (map g ys, ys)
  where ys = map f xs
```

A compiler could generate code for this function that first fully computes `ys`, and then computes `map g ys`. The next iteration is identical in meaning:

```
h1 :: [a] -> ([a], [a])
h1 xs = (map (g . f) xs, map f xs)
```

This version is obtained by inlining `ys`, and fusing the two `map`s. It is possible to keep optimising along this path, which will eventually compute `h` in one pass over `xs`, but there is a big downside: We duplicate a lot of work, because the function `f` gets evaluated twice for each element in `xs`. A different approach changes the type signature slightly[1], but is able to fuse this function optimally:

```
h2 :: [a] -> [(a, a)]
h2 = map (\x -> let y = f x in (g y, y))
```

In this context, *optimal* means that this is the quickest, most efficient formulation of `h`. We would prefer a compiler or optimisation framework where we can write the first version of `h`, and get the performance of the optimally fused version. This would free programmers from having to consider the implementation details of fusing loops, and allow them to write more modular and maintainable code. There has been a lot of research into fusion systems with excellent results for simple cases, such as `map f . map g`, however many of these systems still fail to fuse examples such as `h`.

Performance of data-parallel computing, such as on a GPU, is often highly memory-bound. This makes fusion even more relevant in this field, but none of the fusion techniques for parallel architectures are capable of fusing the function `h`. Moreover, with respect to the examples that these techniques can fuse, whenever there are several conflicting options, they often do not make the optimal choice that would lead to the quickest code.

There is previous research in finding the optimal fusion clustering. This is a computationally expensive task, which makes it unsuitable for most compilers. Interestingly, such work has not focused on data-parallel array languages. However, these could be a great fit for relatively expensive optimisation techniques and analyses, because they are focused on high performance computing where compilation time is regarded to be less important than execution time.

In a relatively separate field, much work has gone into local fusion techniques: These do not consider the global optimisation problem of clustering all functions into fusible clusters, but greedily fuse adjacent functions. Such techniques are much faster, making them more suitable for mainstream compilers, but they do not achieve optimal results.

Our research question is *How can we optimally fuse all programs in a data-parallel language?* In the process of answering this question, we make the following contributions:

- We introduce *diagonal fusion* as a parallel to *horizontal* and *vertical* fusion in Section 4.1,
- We adapt an existing technique for finding *optimal fusion clusters* to support parallel operations in Section 4.3,
- We introduce the idea of annotating the *Abstract Syntax Tree* (AST) with fusion information in Section 4.4,

[1]Fusing `h` optimally without changing its type signature is of course also possible, but it might not be expressible in Haskell. In the case of linked lists, we can achieve this with explicit recursion instead of `map`.

- And we devise a technique and representation for using such a clustering to fuse Accelerate programs in Section 4.5.

We will design an approach to find the optimal fusion clustering for a data-parallel array language, by adapting an existing solution. We also present a novel approach to representing this clustering in the AST, which consists of explicitly annotating pairs of functions with *vertical, horizontal*, and *diagonal* fusion. Finally, we attempt to generate fused code from this AST.

Chapter 2 will expand upon the relevant background, including a more detailed introduction on *fusion.* Chapter 3 will introduce Accelerate, an open-source data-parallel array language embedded in Haskell, which we use as a case study. Chapter 4 covers the theoretical contribution of this thesis, and in Chapter 5 we detail a partial implementation of the proposed solution in the compiler of Accelerate. Finally, Chapters 6 and 7 cover our results and discuss their implications.

2 Background

2.1 Fusion

Throughout this thesis, we will use imperative loops as the key intuition for thinking about fusion. At the same time, we are mainly interested in fusion on functional languages, where these loops are not explicitly present. Because of this, it is useful to develop a simple intuition for how functional idioms like `map, fold, filter, scan, zip` relate to these imperative loops.

Functions like `map` abstract over the underlying data structure, but their implementation may vary. In the following section, we will assume immutable arrays (such as provided by `Data.Array` and `Data.Vector` in Haskell). A simple comparison of these functions and a possible implementation is shown below. In each example, `n` is the length of the array `xs`, and for the `zip` it is assumed that both input arrays have the same size. Initialising statements are omitted, because the focus lies on the loopy structure.

Haskell	Imperative
`ys = map f xs`	`for (int i = 0; i<n; i++)`
	`    ys[i] = f(xs[i]);`
`y  = foldl f e xs`	`for (int i = 0; i<n; i++)`
	`    y = f(y, xs[i]);`
`ys = filter f xs`	`for (int i = 0; i<n; i++)`
	`    if (f(xs[i])){`
	`        ys[j] = xs[i];`
	`        j++;`
	`    }`
`ys = scanl f e xs`	`for (int i = 0; i<n; i++)`
	`    ys[i+1] = f(ys[i], xs[i]);`
`zs = zip xs ys`	`for (int i = 0; i<n; i++)`
	`    zs[i] = (xs[i], ys[i]);`

Composition of such functions and loops is done in Haskell by applying one of these functions to the result of another, such as `map f (map g xs)`, or explicitly: `map f . map g`. In the explicit loopy imperative world, this is represented by simply pasting the second loop after the first loop.

Haskell	Imperative
`zs = map f (map g xs)`	`for (int i = 0; i<n; i++)`
	`    ys[i] = g(xs[i]);`
	`for (int j = 0; j<n; j++)`
	`    zs[j] = f(ys[j]);`

Often, when we encounter two separate loops, it is possible to rewrite the loopy imperative code such that only one loop remains. This process is called fusion, and automatic fusion is an important optimisation step for various languages and compilers. There are various benefits, depending on the language and context in which fusion is performed. The big factors are memory traffic (reducing the number of both reads and writes), memory use, and eliminating duplicate loop counters and redundant allocations. We will formalise the rules for when fusion is valid in the coming sections.

	Haskell	Imperative

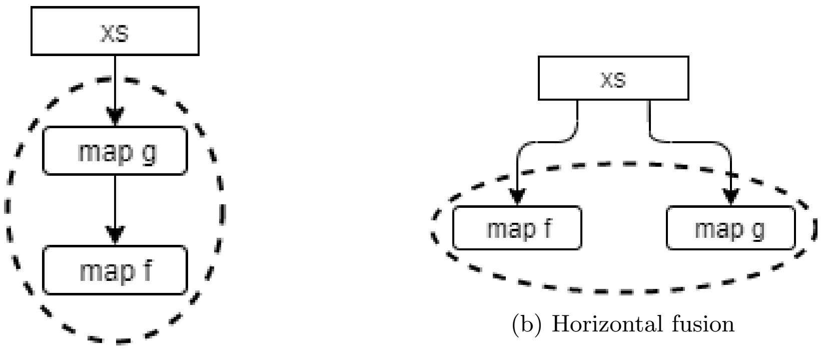
Unfused	`zs = map f (map g xs)`	`for (int i = 0; i<n; i++)`
		`    ys[i] = g(xs[i]);`
		`for (int j = 0; j<n; j++)`
		`    zs[j] = f(ys[j]);`

Fused	`zs = map (f . g) xs`	`for (int i = 0; i<n; i++){`
		`    y = g(xs[i]));`
		`    zs[i] = f(y);`
		`}`

If we look at the simple example above where two maps are fused, we can see all these factors come into play: Fusing these loops reduced memory use, by eliminating the need for the array `ys`.[2] Secondly, for each element of `xs`, we previously had two array reads and two array writes. Because we removed the array `ys`, the fused version only has one of each, which allows it to wait for memory access only half as often. Finally, fusion eliminates one set of loop counters: Instead of counting from `0` to `n` with both `i` and `j`, we eliminate `j` completely.

2.1.1 Classifying Fusion Categories

While all forms of fusion essentially combine two loops into one loop, we would like to categorise specific fusion instances. The terms *vertical fusion* and *horizontal fusion* describe the relationship of the two loops, and they are widely accepted in the existing literature [1]. Their intuïtive meaning is best illustrated using a control flow graph:



(a) Vertical fusion

(b) Horizontal fusion

Figure 1: Vertical and horizontal fusion, as seen in a control flow graph

Vertical Fusion The canonical example of vertical fusion is the `map` *fusion rule*, which can be seen in Figure 1a:

```
map f . map g = map (f . g)
```

Here, we see that any occurrence of this pattern of two sequential maps can be rewritten into one map.

Vertical fusion involves eliminating an intermediate result. By writing these two steps in a single loop, we can reduce memory usage and traffic: The intermediate result does not need to be written to and read from main memory. Moreover, if the intermediate result is an array, it need

[2]Of course, both of these maps could also be done in place, if we use mutable arrays. That would eliminate this one upside of fusion, but the other points still stand.

never exist fully. We can simply generate and consume this intermediate result, one element at a time. In the control flow graph of Figure 1a, we see that the second step is directly beneath the first one.

Horizontal Fusion An example of horizontal fusion is shown in Figure 1b, where two independent functions are mapped over the same array. This corresponds to the following fusion rule[3]:

```
\xs -> (map f xs, map g xs) ≈ map (\x -> (f x, g x))
```

Horizontal fusion is performed by fusing the independent loops into a single loop, producing two results. Horizontal fusion could be classified into more detailed categories: The two loops might iterate over the same source array, which would mean that we save some memory access time, because we can read each element only once and use it for both of the independent loops. It is also possible that the two loops are truly independent, as long as they have the same iteration size. As in vertical fusion, we also eliminate the duplicate loop counters. In the control flow graph of Figure 1b, we see that neither of these loops in horizontal fusion is a descendant of the other: they have a horizontal relationship.

2.1.2 Limitation

Consider the following function, which appeared in the introduction:

```
-- Unfused
f xs = let ys = map g xs
       in (ys, map h ys)
```

```
// Unfused
for (int i = 0; i < n; i++)
    ys[i] = g(xs[i]);
for (int j = 0; j < n; j++)
    zs[j] = h(ys[j]);
return (ys, zs);
```

We cannot horizontally fuse these loops, because the second loop (`map h ys`) depends on the result of the first loop (`map g xs`). We could vertically fuse these loops, but then we lose access to `ys`: it would get fused away. In the introduction, we saw that by inlining `ys`, we can fully fuse this function, at the cost of computing `g` twice for each element of `xs`. Needless to say, this is not satisfactory.

We see that the combination of vertical and horizontal fusion is not always enough to directly encode all fusion possibilities. The above function serves as a minimal example: It should be apparent that this function can be rewritten into a single loop over xs without duplicating work, as below. We currently have no terminology for talking about the fusion of such functions. Section 4.1 will introduce *diagonal fusion*, which covers this case. The combination of *horizontal, vertical* and *diagonal fusion* will allows us to classify every fusion opportunity.

```
// Fused
for (int i = 0; i < n; i++){
    y = g(xs[i]);
```

[3]Note that in this case, there is the subtle difference between a list of pairs and a pair of lists. In an imperative language, we can fuse this function without making this change: the fused loop can still write to two arrays. In purely functional languages, it is often difficult to express this fused version that writes to two separate arrays. Some compilers transform the first representation, called an *Array of Structs*, into the second one, a *Struct of Arrays*. Performing this conversion at *runtime* is called *zipping* and *unzipping*.

```
        ys[i] = y;
        zs[i] = h(y);
    }
```

2.2 Shortcut fusion systems

Shortcut fusion systems represent a subset of fusion systems which can be implemented relatively easily, through a combination of general-purpose optimisations (such as inlining and specialisation) and targeted rewrite rules. During the optimisation passes of the compiler, where these rules and other optimisations are repeatedly applied, these rules replace every occurrence of the left-hand side of the rule with the corresponding right-hand side.

```
{-# RULES
"map/map" ∀f g xs. map f (map g xs) = map (f.g) xs
    #-}
```

For example, if we have a rewrite rule for the map/map fusion rule like above, the two maps are fused any time the compiler encounters the given pattern.

A naïve way to use these rewrite rules for large-scale fusion this would be to write a rule for all combinations of the list primitives: One rule for map/map, one for foldr/map, one for foldr/filter, one for map/filter, etc. This is not a very desirable solution: It leads to a quadratic number of rewrite rules, which in turn leads to unmaintainable libraries that cannot efficiently be combined, and longer compile times. Instead, *shortcut fusion systems*[2] have been designed. The idea behind such systems is to write all list primitives in terms of a very small set of combinators, and use rewrite rules and inlining to fuse instances of these combinators. When designing such a system, like any other optimisation, two properties are important: The transformation must be *correct*, preserving semantics, and it must be an improvement with respect to the optimisation goal. For fusion, although the end goal is the speed of generated code, we usually count the number of *main memory* reads and writes, or the number of allocations.

In the case of rewrite rules, we add another requirement: the optimisation process must always terminate after a finite number of rewrites. This last requirement is usually trivial to check, since each application of the rewrite rules strictly reduces the number of combinators in each of the systems we will look at. It is important to keep this in mind, as GHC[4] itself does not check whether the rewrite rules terminate. There is also no fine-grained mechanism to control the order in which these rules fire.

All fusion systems based on these rewrite rules, and in particular all shortcut fusion systems, are inherently local optimisations. This means that they fuse programs step by step, and never consider the globally optimal fusion. Still, they illustrate some core ideas: Each shortcut fusion system uses just one or two combinators, and one or two rewrite rules, to dramatically simplify their implementation. Secondly, we will see that each of these techniques rewrites list functions to a completely different representation. Because each of these representations seems logical, one can only appreciate the choice when we look at a couple of shortcut fusion systems in more detail.

2.2.1 Stream fusion

The popular Haskell library `Vector` uses Stream fusion to fuse `Arrays`.

Note that sometimes, the word *stream* is used to refer to infinite lists. For stream fusion, we use the following definition of a stream:

```
data Stream a = ∃s. Stream (s -> Step (a, s)) s
data Step a s = Done
```

[4]The Glasgow Haskell Compiler is the de facto compiler for Haskell.

```
                | Skip    s
                | Yield a s
```

In this context, a stream is a reformulation of a list, with a seemingly redundant `Skip` constructor. This isomorphism follows from the following transformation functions:

```
stream :: [a] -> Stream a
stream xs = Stream uncons xs
  where
    uncons []     = Done
    uncons (x:xs) = Yield x xs

unstream :: Stream a -> [a]
unstream (Stream next s) = unfold next s
  where
    unfold next s = case next s of
      Done       -> []
      Skip s'    -> unfold next s'
      Yield x s' -> x : unfold next s'
```

One might notice that the phantom type variable `s`, though existentially quantified, is always instantiated to `[a]` in these functions. That is not a coincidence: This is always the case. The reason `Stream` is defined in this way, is that it increases typesafety: This formulation, while not valid in normal Haskell[5], abstracts over `s` which makes it much easier to prove its correctness [3].

In a stream fusion system, all list functions are written to work on streams. For example, map is written as:

```
map :: (a -> b) -> [a] -> [b]
map f = unstream . map_s f . stream

map_s :: (a -> b) -> Stream a -> Stream b
map_s f (Stream next s) = Stream next' s
  where
    next' str = case next str of
      Done      -> Done
      Skip s'   -> Skip s'
      Just x s' -> Yield (f x) s'
```

Surprisingly, the only fusion rule we need for stream fusion is the following rule, which removes redundant conversions:

```
{-# RULES
"stream/unstream" ∀s. stream (unstream s) = s
  #-}
```

This rule (combined with the other optimisation techniques like inlining and case-of-case transformations) is enough to generate optimally fused code for many functions. Specifically, if `map` cannot be fused with anything, these other optimisations are sufficient to eliminate all the Step and Stream wrappers and generate a normal definition of `map`.

[5]Haskell, as defined in the language reports, does not support this. GHC, the de facto Haskell compiler, supports such quantification through language extensions.

2.2.2 Unbuild/unfoldr

Unbuild/unfoldr fusion is based on the functions `unfoldr` and `unbuild` (sometimes referred to as `destroy`) [4]:

```
unfoldr :: (s -> Maybe (a,s)) -> s -> [a]

unbuild :: (∀s. (s -> Maybe (a,s)) -> s -> b) -> [a] -> b
unbuild g xs = g uncons xs
uncons [] = Nothing
uncons (x:xs) = Just (x, xs)
```

Here `unfoldr` is a familiar list construction function, and `unbuild` is a list consumer that is tailored to fuse with `unfoldr`. Like in Stream fusion, the nested *forall* in the type signature reflects existential quantification, even though `s` will always be instantiated to `[a]`. In this fusion system, all fusible functions are written in terms of these two building blocks. The fusion rule for unbuild/unfoldr fusion is the following:

```
{-# RULES
"unbuild/unfoldr" ∀k g s. unbuild g (unfoldr k s) = g k s
  #-}
```

This fusion system allows us to express `zip`-like functions in such a way that they can fuse both input lists. On the flip side, some other functions (most notably `filter`) cannot be fused in this fusion system at all. This is because, while `filter` can be written in terms of `unfoldr`, it requires a recursive 'stepping function' which cannot be fused. In fact, unbuild/unfoldr fusion is functionally identical to Stream fusion, aside from the `Skip` option. If we replace the `Maybe` in unbuild/unfoldr fusion with a `Step`, or the `Step` in Stream fusion with a `Maybe`, these two fusion systems can fuse exactly the same functions.

2.2.3 Foldr/build

The final shortcut fusion system we discuss, is the first one that was developed: The *foldr/build* fusion system [5]. This is the system which GHC still uses (with minor changes) to fuse functions on linked lists in the `base` package. *Foldr/build* fusion is based on the functions `foldr` and `build`:

```
foldr :: (a -> b -> b) -> b -> [a] -> b

build :: (∀b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

Foldr is a familiar list consuming operation, which functional programmers come across regularly. It is preferred over foldl since it can deconstruct a cons list one element at a time. A useful intuition for foldr, especially in this context, is that it replaces the (:) and [] in a list:

```
foldr c n [1,   2,   3]         ==
foldr c n (1 : (2 : (3 : []))) ==
        (c 1 (c 2 (c 3 n)))
-- In particular:
foldr (:) [] == id
```

Once again, we refer to language extensions and proofs on free theorems for the nested *forall* in the type of `build` [3], instead of instantiating `b` with `[a]`.

Armed with the `foldr`-intuition, we can see how `build` is a list producer that should fuse with a `foldr`-like consumer. This is exploited using the only rewrite rule of this fusion scheme:

```
{-# RULES
"foldr/build" ∀f g z. foldr f z (build g) = g f z
  -#}
```

Foldr/build fusion is capable of fusing filters, but many other common list-based functions cannot be expressed in terms of `foldr` and `build`. For example, `zip`-like functions can only be fused with one of their arguments. Left folds have also been problematic for a long time, because although they can be expressed in terms of `foldr`, the resulting code was far from optimal [2]. In recent versions of GHC, `foldl` can now be fused efficiently with *foldr/build* fusion [6].

2.3 Flow fusion

In Lippmeier et al. [7], the authors advocate for a new fusion system called *flow fusion*. It is compared with stream fusion, as one of the main goals is to fuse zip-like functions (which the *foldr/build* fusion system cannot handle) more efficiently. They critiques stream fusion on two aspects: the reliance on optimisation passes like inlining, and the duplication of loop counters. We have mentioned the first of these aspects before, as a positive aspect: Shortcut fusion methods are remarkably easy to implement if these other optimisations are already present. Lippmeier et al. [7] points to a concrete downside of this reliance: The system is not just fragile, but there is a whole family of fusible constructs that cannot be inlined. For example, whenever a list producer has multiple consumers, it cannot be inlined without duplicating work, and thus it cannot be fused at all in a shortcut fusion system.

The second critique this paper has on stream fusion is the duplication of loop counters. This is a situation which happens when a `zip`-like function is fused: As there is no guarantee that the input lists are of equal length, code generated by stream fusion will introduce a loop counter for each of the inputs, as well as for the output list. These variables are always identical (they are incremented simultaneously) and could be optimised away, but the bounds checks cannot.

Flow fusion first identifies, on a global level, which functions can be fused. It does this using *rate inference*. The idea behind rates is that they represent an approximation to array length equality: Two lists with the same rate are guaranteed to have the same length, but not the other way around. After rate inference, the program is divided into processes. Each process is guaranteed to completely fuse.

Each of these processes is then scheduled into a procedure, which is an abstract, imperative loop nest. Procedures are defined such that they can never contain unfused loops: They consist of a 'start' and an 'end' statement, which are both executed exactly once for each evaluation of the loop, a 'body', which is executed as many times as the loop loops, and one 'inner nest', which is a recursive definition. This is the phase where fusion happens. Afterwards, these abstract loop nests are concretized and inserted back into the main program.

The idea of these *procedures* is very similar to a part of my contribution. Section 4.5 goes into more detail and compares the approaches. All in all, Flow fusion is capable of perfectly fusing sequential programs, if they are properly clustered. Section 2.5 goes into more detail on what this means.

2.4 Fusion in purely functional data-parallel languages

There are a number of purely functional data-parallel languages, which serve to make high-performance computing declarative. This Section explores the approaches these languages use for fusion.

Accelerate is one of these languages. It is deeply embedded in Haskell, and compiles to parallel code for both CPU and GPU. Accelerate is discussed in more detail in Section 3. Accelerate uses *Delayed Arrays* for fusing its operations[8, 9]. This approach fuses all cheap primitives vertically into their consuming counterparts. It does not support horizontal fusion in general, though use of functions like `zipWith` could result in horizontal fusion. I will discuss the current state of fusion in Accelerate in more detail in Section 3.2.

The language Futhark is similar in purpose to Accelerate, but it is not embedded in a host language. Futhark's backend is significantly different to Accelerate, and it uses streams extensively. The fusion is based on rewrite rules that greedily attempt to fuse these streams [10]. Since these rewrite rules and their ordering are static, there is not much control over the fusion process.

The language Lift also falls under the category of data parallel array languages. Lift aims to be an intermediate representation that high-level languages can compile into. In turn, Lift then performs low-level optimisations and emits OpenCL code. Lift performs some limited vertical fusion [11].

Halide is a domain specific language for image processing pipelines embedded in C++, whose backend can perform a wide range of optimisation techniques including fusion. It is most suited for automatic tuning, where a large number of optimisation techniques are attempted in a stochastic search in order to experimentally find an optimal code generation scheme[12].

Single assignment C (SaC) is a purely functional array language, that compiles to many parallel backends. Its syntax is aimed at providing imperative programmers a familiar, C-like interface. This makes it feel like a subset of C with some added features, like the with-loop which defines parallel generation, modification and folding of arrays [13]. SaC implements vertical fusion as *with-loop folding* [14] and horizontal fusion as *with-loop fusion* [15]. As in many other languages, these fusion opportunities are considered separately, and no attempt is made to find an optimal fusion clustering.

Other language projects that aim to provide a functional programming interface to parallel architectures include SPOC [16], Obsidian [17] and Nikola [18]. None of these seem to perform a significant amount of array fusion.

2.5 Finding optimal fusion clusterings

All shortcut fusion methods are inherently local transformations. If we consider fusion on a whole-program basis instead, it turns out that there can be multiple ways to partition the graph into fused clusters. Some of these are simply less fused than others, but there can also be multiple distinct local optima. In the general case, this problem is NP-complete. Troels Henriksen and Cosmin Eugen Oancea [19] define this as a graph partitioning problem, and provide an exponential exact algorithm, a greedy approximation algorithm, and a safe preprocessing step that, in practice, speeds up the exact algorithm significantly.

A different approach uses Integer Linear Programming. An *Integer Linear Program* (ILP) is a set of integer (or real) variables, with linear constraints and a linear objective function. Solving ILPs is NP-complete, but there are solvers (both commercial and free) that use heuristics and various algorithms which are capable of solving most ILPs quite quickly. Because of this, they are a popular tool for solving NP-complete problems. In the rest of this section, we look at two papers that use ILPs to find optimal fusion clusterings.

2.5.1 Optimal weighted loop fusion for parallel programs

In Megiddo and Sarkar [20], the authors describe an Integer Linear Programming formulation for finding the optimal clustering of loops to fuse. They do this in two steps: First a naïve formulation, and then they refine it into a more efficient ILP. They also describe a custom branch-and-bound algorithm which they suspect to be superior in performance to a general purpose ILP solver.

First, they introduce some terminology: A loop dependency graph is a directed acyclic graph, where each node represents a loop. There are two types of edges in this graph: Fusible edges, which represent the idea that although one of the loops depends on the other, they can be fused, and infusible edges, which represent the idea that one of the loops depends on the other, and they cannot be fused. In either case, if the edge (i, j) exists, the cluster containing j cannot be computed earlier than the cluster containing i.

The first proposed ILP, which has $\mathcal{O}(n^2)$ variables and $\mathcal{O}(n^3)$ constraints where n is the size of the graph that represents the program, assigns a π variable to each node on the loop dependency graph. These variables will tell us which loops are fused: if $\pi_i = \pi_j$, we fuse loops i and j. Furthermore, they introduce a constraint for each fusible edge (i, j) that $\pi_j - \pi_i \geq 0$ and for each infusible dependency edge (i, j) that $\pi_j - \pi_i \geq 1$. They also have a variable $x_{i,j}$ for each i, j, which signifies whether i and j are fused. Finally, they introduce some constraints that relate these variables to each other to be consistent, including a transitivity constraint $x_{i,k} \leq x_{i,j} + xj, k$.

A simple proof shows that these transitivity constraints are not required, and indeed follow from the other rules. This leads to a more efficient ILP formulation, that is linear in size with respect to the graph.

2.5.2 Fusing Filters with Integer Linear Programming

The authors of Robinson et al. [21] iterate on Megiddo and Sarkar [20]. Where the original ILP formulation does not support `filter`-like operations, as they alter the size of arrays, this paper describes a way to embed them into the existing framework. Arrays that can be deduced to have the same size are handled just like in the original formulation. When two arrays do not always have the same size, but they have a pair of ancestors that do, the arrays can only be fused if they are both fused with their respective ancestors, and the ancestors are fused with each other.

2.6 Summary

In Figure 2, we compare the existing fusion methods that are discussed in this thesis, on five aspects. Note that *delayed arrays* are also in this comparison, they will be covered in more detail in Section 3.2. The columns `zip` and `filter` show which fusion methods are capable of fusing both inputs to a `zip`, or handle `filter`-like operations. The other three columns compare different aspects of these fusion systems: Can they preserve parallelism, do they consider global implications or are they based on local rewrites, and is the fusion method type-preserving?

	parallel	global	`zip`	`filter`	typed
stream			✓	✓	✓
unfoldr/unbuild			✓		✓
foldr/build				✓	✓
flow		✓	✓	✓	
delayed arrays	✓		✓		✓

Figure 2: Comparing existing fusion methods

This table shows that there is currently no fusion system that answers all our needs. We identified the following gaps:

1. The existing concepts *horizontal* and *vertical* fusion are not sufficient to describe all legal fusion opportunities.

2. There is no complete, type-safe compiler that uses a method such as the ILPs from Section 2.5 to optimally fuse programs.

3 Accelerate

In this thesis, we use the language Accelerate as a case study and we partially implement the proposed fusion process in its compiler. The strengths and weaknesses of the proposed approach to fusion align well with the language Accelerate: Longer compile times are acceptable, achieving optimal performance from declarative code is already the goal, and the language only exposes higher-order looping structures (no explicit recursion or loops). For this reason, we provide an overview of the relevant details of Accelerate in this section.

3.1 Introduction to Accelerate

Accelerate is an open source language deeply embedded in Haskell aimed at high-performance parallel computing [22, 23, 8, 24]. Accelerate code embedded into Haskell is not complied by the Haskell compiler: The Accelerate library includes a runtime compiler which generates and compiles parallel SIMD or GPU code at application runtime. Accelerate code often looks very similar to Haskell code. For example, this is an implementation of the *dot product* function:

```haskell
dotp :: Num a => Acc (Vector a) -> Acc (Vector a) -> Acc (Scalar a)
dotp xs ys = fold (+) 0 $ zipWith (*) xs ys
```

In this example, `fold` and `zipWith` are not the familiar Haskell functions that work on lists, but Accelerate functions operating on array computations embedded in the Accelerate world. This is reflected in the types:

```haskell
fold :: (Shape sh, Elt a)
     => (Exp a -> Exp a -> Exp a)
     -> Exp a
     -> Acc (Array (sh :. Int) a)
     -> Acc (Array sh a)

zipWith :: (Shape sh, Elt a, Elt b, Elt c)
        => (Exp a -> Exp b -> Exp c)
        -> Acc (Array sh a)
        -> Acc (Array sh b)
        -> Acc (Array sh c)
```

The Accelerate language consists of two levels, `Acc` and `Exp`. `Exp` is the world of scalar values and functions, which can be executed on a single thread of, for example, a GPU. `Acc` describes arrays and collective array operations, such as `map` and `fold`. The typeclass `Elt` describes the types of values that can be contained in arrays, and the typeclass `Shape` describes the possible shapes of these arrays. Two instances of `Shape` are `Z` (a *scalar* array, an array that contains only one element, an array with dimension 0) and `(Shape sh) => Shape (sh :. Int)` (an array with one extra dimension). `Z` and `:.` are used to denote both type-level shapes, and value-level shapes, which double as (zero-based) indexes into arrays.

One of the restrictions of Accelerate is that it does not support nested parallelism. `Exp` computations can index into arrays, but only if each `Exp` computation in one `Acc` computation indexes into the same set of arrays: Scalar expressions cannot initiate parallel computations. For example, the following function is not allowed:

```
matvecmul :: Num a => Acc (Matrix a) -> Acc (Vector a) -> Acc (Vector a)
matvecmul mat vec =
  let Z :. rows :. cols  = shape mat
  in  generate (Z :. rows)
               (\(Z :. row) ->
                  (dotp vec
                     (backpermute (Z :. cols)
                                  (\(Z :. col) -> Z :. col :. row)))
                  ! Z)
```

Here, we try to define matrix-vector multiplication using the `dotp` function we saw earlier, by using `backpermute` to extract the relevant row from the matrix. The reason this will not work, is that each scalar expression would have to compute the parallel `dotp` and `backpermute` functions. However, this does not mean we can never write functions that have a collective operation inside of a `generate`: It is just that the collective operation may not depend on the index. In this example, the index `row` is used inside of the `backpermute`. This breaks the two-level system where individual `Exp` functions are not allowed to create their own collective `Acc` computations.

3.1.1 Shapes

Shapes are *snoc lists*, which means that the *outermost* `Int` corresponds to the *innermost* dimension of the array. Functions like `fold` reduce along this *innermost* dimension. For example, in the following code snippet, `xs` is a two-dimensional array with twelve elements: four rows with three elements each. The row ranging from `(Z :. 0 :. 0)` to `(Z :. 0 :. 2)` contains the numbers `{0, 1, 2}` in order, and the suggestively named function `seven` indexes into the 8th location to find the number 7.

```
seven :: Exp Int
seven = xs ! (Z :. 2 :. 1)
  where
    xs :: Acc (Array (Z :. Int :. Int) Int)
    xs = use $ fromList (Z :. 4 :. 3) [0..11]
```

3.1.2 Primitive Array operations

Accelerate has a variety of primitive array operations. Since these are the subjects of fusion, it is important to be familiar with their declarative meaning and to gain an intuition for how these operations might be evaluated on parallel hardware.

Map The function `map` applies an `Exp` function to each element in an `Acc` array. Depending on the amount of parallelism available, any number of these computations might be done synchronously.

```
map :: (Shape sh, Elt a, Elt b)
    => (Exp a -> Exp b)
    -> Acc (Array sh a)
    -> Acc (Array sh b)
```

Fold The function `fold` reduces an array along its innermost dimension. Unlike `foldl` or `foldr`, this function assumes associativity[6] of the combination function and does not specify the order of operations: Depending on the backend, the size of the array, and even the amount of parallelism available different methods might be used.

```
fold :: (Shape sh, Elt a)
     => (Exp a -> Exp a -> Exp a)
     -> Exp a
     -> Acc (Array (sh :. Int) a)
     -> Acc (Array sh          a)
```

Scans The functions `scanl, scanr` have the same declarative meaning as their Haskell counterparts. For example, `scanl (+) 0` computes the prefix sum of a vector. On higher-dimensional data, scans (like folds) work along the innermost dimension. Where folds reduce each innermost row to a single element, scans preserve the dimensionality. Like folds, scans require the function to be associative.

```
scanl, scanr :: (Shape sh, Elt a)
             => (Exp a -> Exp a -> Exp a)
             -> Exp a
             -> Acc (Array (sh :. Int) a)
             -> Acc (Array (sh :. Int) a)
```

Stencils A `stencil` is like a `map`, except the input for the `Exp` function is not a single element, but the surrounding region in the input array. For example, a $3*3$ `stencil` looks at all nine inputs in the range `{i-1, i, i+1}` × `{j-1, j, j+1}` to compute the new value at position `(i, j)`. The typeclass `Stencil` captures the possible input ranges, and the data `Boundary` describes how the edges of the stencil should be computed, when some of the input coordinates do not exist. Stencils provide an interesting tradeoff for fusion: Whenever a stencil fuses (vertically) with its input, work is duplicated. Sometimes, the benefits of fusion may outweigh the recomputation of a cheap function, but at other times fusing may have a net negative effect.

```
stencil :: (Stencil sh a stencil, Elt b)
        => (stencil -> Exp b)
        -> Boundary (Array sh a)
        -> Acc (Array sh a)
        -> Acc (Array sh b)
```

Generate For the cases where all the specialised primitives do not have the desired expressivity, there is also the `generate` function. It generalises all array computations: Given a shape and a generating function, this constructs an array that has the desired element at each index. `generate` has no explicit array input, which means that it does not fuse with anything that happens *before* the generate.

```
generate :: (Shape sh, Elt a)
         => Exp sh
         -> (Exp sh -> Exp a)
         -> Acc (Array sh a)
```

Indexing Arrays can be indexed either with their multidimensional coordinates, or with a linear index. Indexing is a fusion preventing operation: The indexed array has to be fully computed before it can be accessed safely.

[6]Reminder: an operator $\oplus$ is *associative* if $(a \oplus b) \oplus c = a \oplus (b \oplus c)$ for all a, b, c. The related concept of *commutativity* holds if $a \oplus b = b \oplus a$ for all a, b, but Accelerate does not assume this property.

3.2 Fusion through Delayed Arrays

The current approach to fusion in Haskell is through the use of *Delayed Arrays*[25, 8, 9]. The idea is simple: Wherever possible, arrays are not represented explicitly, but as a shape and an indexing function. This means that, for example, composing index space transformations is achieved by composing the functions instead of creating the intermediate arrays.

The primitive collective operations in Accelerate are categorised as either *element-wise* or *collective*. In this distinction, element-wise operations are the primitives where each element of the output only depends on a single element of the input array. Conversely, collective operations are the primitives where a single element of the output array can depend on multiple elements of the input array. Element-wise operations can be written as delayed arrays, but consumers cannot because they would introduce nested parallelism. Primitives that do not belong to either of these categories are never fused. This includes constructs like control flow and foreign function interfaces. In Figure 3, we show how most primitives are classified into these categories.

Element-wise	Collective	Infusible
Map	Folds	Variable
Generate	Scans	Let-binding
Backpermute	Permute	Array introduction
ZipWith	Stencils	Control flow
Slice		Foreign functions
Replicate		Function application

Figure 3: The element-wise, collective, and infusible operations in Accelerate

Array fusion in Accelerate now amounts to a two-step process:

1. In the *producer-producer* step, successive element-wise operations in the AST are vertically fused into a single delayed array. This is done in a straightforward manner, ignoring any work duplication that might arise: Functions like Backpermute can access a single value in the source array arbitrarily often. For example, a chain of maps will be transformed according the classic map-fusion rule mentioned earlier. Each fusion step here is a source-to-source rewrite.

2. In the *producer-consumer* step, the AST is annotated to signify which of the above delayed arrays should be computed directly, and which should be embedded into the consuming collective operation. During code generation, the embedding of delayed arrays into collective operations will happen. The output of a collective operation never gets fused.

It is possible that a chain of maps and backpermutes, which are simple content and index space transformation functions, is not allowed to fuse with their input and output. For example, there is an explicit fusion preventing function to give the programmer more control. In this case, we would still like to fuse these maps and backpermutes with each other, as these functions fuse very well. To facilitate this, a hybrid Map/Backpermute primitive is added to the AST.

While this current approach solves many issues that previous methods would have had, the most important of which is the preservation of parallelism, it is still not the ideal solution. Only a limited subclass of fusion operations can be expressed in this framework (notably, only vertical fusion is possible), and one can even construct examples where the current fusion causes arbitrarily high slowdowns.[7]

[7]This occurs when an expensive `map` on a small array is followed by a `backpermute` that transforms it into a big array.

3.3 Internal Representation

Accelerate's internal representation is strongly typed. This makes it a lot more difficult to make incorrect transformations, as they would often be caught by the typechecker. This gives us some correctness guarantees, but it also makes large-scale transformations on the AST more difficult. In sections 4.4 and 5 we will work with this representation and expand it.

3.3.1 Typed de Bruijn indices

The internal representation is based on type-safe De Bruijn indices[26]. This means that the environment is modelled as a list of values, and variables are represented by indices in this list. Concretely, each let-binding introduces new variables on top of this list. The indices and environment are strongly typed. This gives some guarantees: An index is a *proof* that the environment contains an variable of a certain type, and we cannot change the environment or the indices without properly updating the other.

The datatypes for indices `Idx` and environments `Val` are shown below, together with the essential projection `prj` which uses an index to get a value from an environment.

```
data Idx env t where
  ZeroIdx ::                Idx (env, t) t
  SuccIdx :: Idx env t -> Idx (env, s) t

data Val env where
  Empty :: Val ()
  Push  :: Val env -> t -> Val (env, t)

prj :: Idx env t -> Val env -> t
prj ZeroIdx       (Push _   v) = v
prj (SuccIdx idx) (Push val _) = prj idx val
```

To be able to do transformations on the AST, there are some utilities for working with the environment. For example, the following newtypes represent *Weakening* and *Strengthening*, the process of making an environment bigger or smaller. Weakening is always valid, but strengthening returns a `Maybe` to represent the idea that an `Idx` might not be present in the smaller environment.

```
newtype env :>  env' = Weaken     (∀t. Idx env t ->        Idx env' t)
newtype env :?> env' = Strengthen (∀t. Idx env t -> Maybe (Idx env' t))
```

Accompanying these newtypes are functions that apply a weakening or strengthening to (a branch of) an AST, by applying the index transformation inside to each `Idx`.

A `LeftHandSide arrs env env'` describes how the environment `env` can be extended with the array or arrays `arrs` to get `env'`. In other words, it is a generalisation of `env' = (env, arrs)`, which can deal with pairs of arrays and with wildcards, that do not expand the environment.

```
data LeftHandSide arrs env env' where
  LeftHandSideArray
    :: (Shape sh, Elt e)
    => LeftHandSide (Array sh e) env (env, Array sh e)

  LeftHandSideWildcard
    :: ArraysR arrs
    -> LeftHandSide arrs env env

  LeftHandSidePair
    :: LeftHandSide arrs1          env  env'
    -> LeftHandSide arrs2          env' env''
    -> LeftHandSide (arrs1, arrs2) env  env''
```

3.3.2 Abstract Syntax Tree

The abstract syntax tree is the internal representation of an Accelerate program. It uses the De Bruijn indices both on the `Acc` level, for array-level variables, and on the `Exp` level, for element-level variables. In this thesis, we will only focus on the array-level.

In the listing below, the type of the environments is represented by `aenv, aenv'`. We see that, in the case of `Alet` (the array-level let-binding constructor), the `LeftHandSide` representation from Section 3.3.1 is used.

```
data PreOpenAcc acc aenv a where
  Alet        :: LeftHandSide bndArrs aenv aenv'
              -> acc            aenv  bndArrs
              -> acc            aenv' bodyArrs
              -> PreOpenAcc acc aenv  bodyArrs

  Map         :: (Shape sh, Elt e, Elt e')
              => PreFun     acc aenv (e -> e')
              -> acc            aenv (Array sh e)
              -> PreOpenAcc acc aenv (Array sh e')

  ZipWith     :: (Shape sh, Elt e1, Elt e2, Elt e3)
              => PreFun     acc aenv (e1 -> e2 -> e3)
              -> acc            aenv (Array sh e1)
              -> acc            aenv (Array sh e2)
              -> PreOpenAcc acc aenv (Array sh e3)
  (..)
```

The AST is not defined recursively, but parametrised over a recursive closure. This allows us to define arbitrary recursive closures in the backend, for example to annotate each node with some information.

```
newtype OpenAcc aenv t = OpenAcc (PreOpenAcc OpenAcc aenv t)
```

The recursive closure `OpenAcc` is the most neutral one: It is just a wrapper around `PreOpenAcc`. This means that an `AST` of type `OpenAcc` can be interpreted as a `PreOpenAcc` where all instances of `acc` are simply recursive.

The closure that currently represents a fused Accelerate program is shown below. It is a *generalized algebraic datatype* (GADT) with two constructors: `Delayed`, which represents a fused chain of element-wise operations as an array that can be generated from its shape and a function from each index, and `Manifest`, which represents a real array that will be computed in memory.

```
data DelayedOpenAcc aenv a where
  Manifest              :: PreOpenAcc DelayedOpenAcc aenv a
                        -> DelayedOpenAcc aenv a

  Delayed               :: (Shape sh, Elt e) =>
    { extentD           :: PreExp DelayedOpenAcc aenv sh
    , indexD            :: PreFun DelayedOpenAcc aenv (sh  -> e)
    , linearIndexD      :: PreFun DelayedOpenAcc aenv (Int -> e)
    }                   -> DelayedOpenAcc aenv (Array sh e)
```

4 Theoretical contribution

The main question we try to answer with this thesis is: *How can we optimally fuse all programs in a data-parallel language?* To this end, we formulate the following subquestions:

1. What kind of data structure can express a fusion clustering in the AST? Sections 4.1 and 4.4 cover this question.

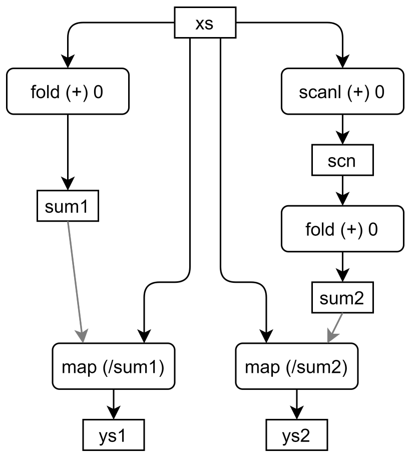
2. Can we use an ILP solver to find the optimal fusion clustering for a data-parallel array program? This question will be answered in Section 4.3.

3. How do we represent and evaluate these fused programs? We answer this question in Section 4.5.

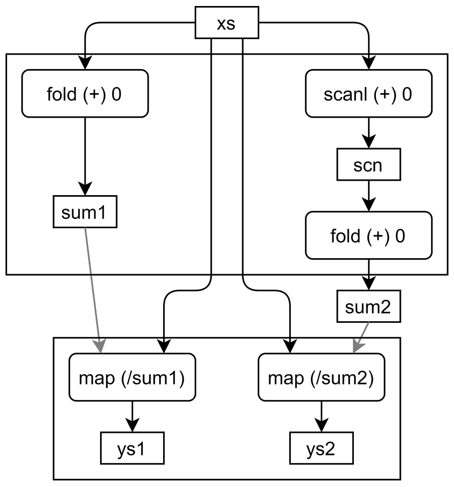
Section 4.2 gives an overview of the sections that follow it, and details how the ideas that are presented interact. Section 5 will discuss how these ideas are leveraged in the Accelerate compiler.

Throughout this chapter, we will look at the following Accelerate program as an example. The goal is to optimally fuse it. This example is a slightly altered version of `normalise2` from Robinson et al. [21]. Compared to the original version, we replace the filter with a scan.[8]

```
normalise2 :: Acc (Array DIM1 Int) -> Acc (Array DIM1 Int, Array DIM1 Int)
normalise2 xs = (ys1, ys2) where
  sum1 = the $ fold  (+) 0      xs
  scn  =       scanl (+) 0      xs
  sum2 = the $ fold  (+) 0      scn
  ys1  =       map (`div` sum1) xs
  ys2  =       map (`div` sum2) xs
```



(a) Unfused



(b) The optimal clustering

Figure 4: The control flow graph of `normalise2`

[8]Accelerate (due to its parallel nature) does not have filter as a primitive operation. It is defined in Accelerate's standard library, but an efficient filter on highly parallel backends consists of multiple passes. In other words, the data parallel filter implementation can never fully fuse with itself. Fusion techniques presented here and elsewhere could fuse the first and last pass of this filter with its input and output, if the filter is just one function in a larger program.

As noted in Robinson et al. [21], and demonstrated in Figure 4, the optimal fusion grouping for this program is {sum1, scn, sum2}, {ys1, ys2}. We will explore how we can get to this conclusion in Section 4.3, and how we aim to generate fused code in Section 4.5. First, we explore the idea of 'diagonal fusion' in the next section.

4.1 The gap between vertical and horizontal fusion

The concepts of vertical and horizontal fusion have been introduced in Section 2.1.1. Figure 5 shows how we can apply these concepts to fuse the first half of `normalise2`: First we vertically fuse the scan with its consumer, the fold that produces `sum2`. After that, the `sum1` fold and the scan/fold combination both consume xs, and they could do so in lockstep. This means that we can horizontally fuse them.

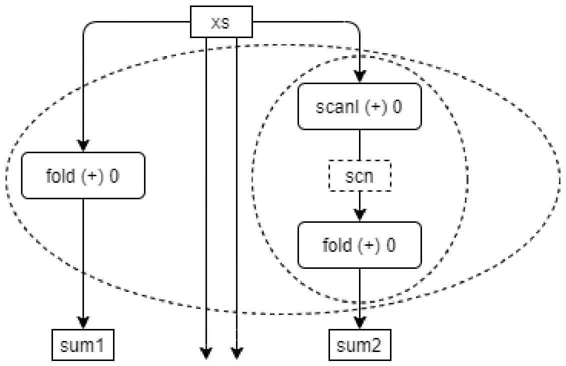


Figure 5: Applying vertical and horizontal fusion to the first half of `normalise2`

We have seen in Section 2.2 that many prior fusion systems are not capable of fusing both inputs to a `zip`, but this concept is easily subsumed by horizontal and vertical fusion: We can horizontally fuse the inputs, and then vertically fuse that result into the `zip`.

We were able to fuse `sum1, scn` and `sum2` with just vertical and horizontal fusion, because `normalise2` is not a very complex example. To make it more interesting, let us expand it slightly: Consider that xs is a result of a `map`, as in Figure 6.

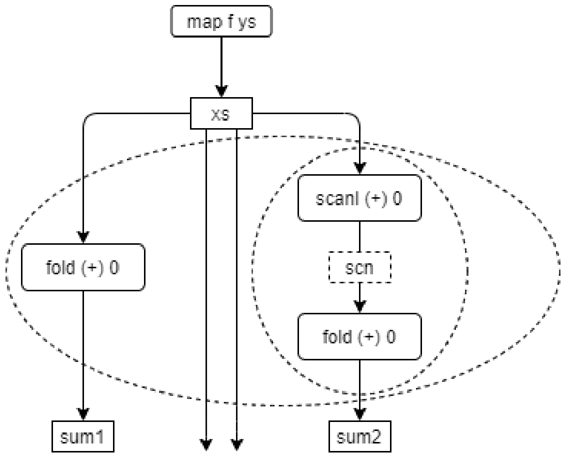


Figure 6: How can we fuse xs?

Here, the familiar constructs of vertical and horizontal fusion cannot help us: We want to somehow perform the map in lockstep with the first half of `normalise2`, but also store xs as a manifest array for the second half of `normalise2`. A horizontal fusion is not legal, as the scan and folds depend on xs, but a vertical fusion would eliminate xs. Introducing the concept of diagonal fusion will give us a general way to talk about this situation and describe more complex fusion opportunities.

4.1.1 Diagonal fusion

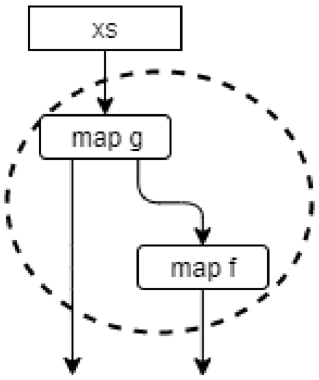


Figure 7: Diagonal fusion

Diagonal fusion is a term and concept which has not been expressed before. As the name suggests, it is a combination of vertical and horizontal fusion. In diagonal fusion, the two steps which we fuse into one loop have a vertical relationship (meaning that the result of one step is used to compute the other step), but the intermediate result is also stored for future computations or even as part of the result of the program. This allows us to still reap half[9] of the benefits of vertical fusion, in situations where vertical fusion would not be possible. Note that it is possible to emulate diagonal fusion by first horizontally fusing the identity function and some other function, and then vertically fusing the result (see Figure 8). However, this added `map id` could have performance implications. The strength of adding diagonal fusion explicitly is that we can guarantee that this identity function is free, and it allows us to construct and decompose the fusion AST more easily, while keeping it decoupled from language constructs.

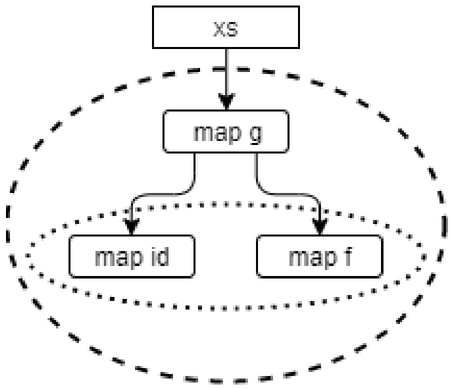


Figure 8: Emulating diagonal fusion using horizontal and vertical fusion

[9]Diagonal fusion eliminates the read of the intermediate result, but not its write.

Diagonal fusion is very similar to a standard let binding, with respect to its expressivity. This means that diagonal fusion, in combination with some form of environment weakening, subsumes both vertical and horizontal fusion: One could emulate vertical fusion by discarding a result after diagonal fusion, and one could emulate horizontal fusion using diagonal fusion, by weakening one of the functions to work in an environment that includes the result of the other function. This guarantees that it is expressive enough to capture any possible fusion instance, as they could also be expressed using let-bindings.

As is a common theme in this thesis, we note that expressivity is not always a good thing: The concepts of vertical and horizontal fusion are useful, precisely because they are more restrictive. This benefit extends beyond human conversation; the restrictions on these fusion concepts allow more targeted optimisations. As a result, we aim to use all three fusion directions in our approach. Note that this choice is somewhat contrary to the philosophy of shortcut fusion schemes: They are focused on rewriting every fusible function into one or two combinators. That said, using three fusion combinators is still very doable, even if we encounter quadratic or even exponential blow-up in the number of functions we have to write later on.

Together, the three concepts of *Vertical, Horizontal* and *Diagonal* fusion cover all local data dependency relationships that two fusible loops can have, except for complete independence.[10] It could be possible that two loops are guaranteed to have the same size, without having a common ancestor. In particular, this occurs when the result arrays of these two loops are `zip`ped: The function `zip` ignores the excess input of the largest array, so we are allowed to simply not compute that part in the first place. For our purposes, it is sufficient to treat this case like ordinary *horizontal fusion*. In other words, we define *horizontal fusion* as any fusion where the two loops can be computed independently.

Coming back to our example, it is now obvious that we want to *diagonally fuse* `xs = map f ys` with the cluster containing the scan and folds, as we can see in Figure 9.

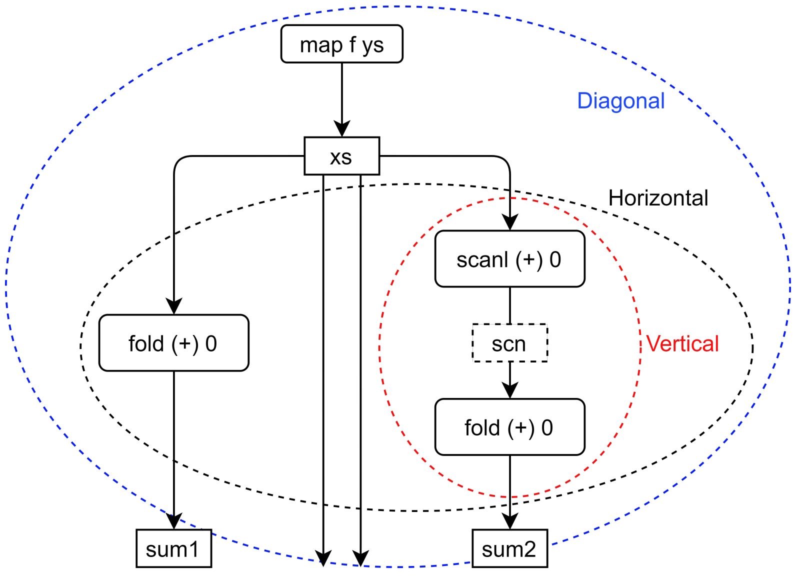


Figure 9: Fusing the first half of the extended version of `normalise2`

[10]This follows from a simple argument: Either the two nodes are completely unrelated (independence) or not. If not, then either one is the ancestor of the other (vertical or diagonal relationship), or not (horizontal relationship).

4.2 Compiler pipeline

In this section, we give a quick overview of the parts of the compiler pipeline that are affected by the new fusion system. The changes can roughly be split in two phases: *Fusion* and *Code Generation*. The fusion phase is mostly backend-agnostic, whilst the code generation phase has to be implemented for each supported backend. Accelerate supports parallel backends for the CPU and the GPU, and an interpreter in Haskell.

The fusion phase is discussed in sections 4.4 and 4.3, and the code generation phase is discussed in Section 4.5. In Section 5, the implementation in Accelerate is discussed in more detail.

1. Fusion phase:
   (a) Convert the AST to a labelled and let-bound version
   (b) Generate the ILP constraints
   (c) Solve the ILP
   (d) Reorder the AST to group the clusters
   (e) Explicitly annotate the AST with *vertical, horizontal, diagonal* fusion primitives
2. Simplification passes
3. Code generation phase:
   (a) Convert each AST node to the Intermediate Representation
   (b) Fuse these representations using the fusion annotations
   (c) Generate code for each fully fused loop nest

4.3 ILP formulation

As noted in Section 2.5, the problem of finding the optimal fusion clustering is, in general, NP-hard. Figure 10 shows three locally optimal fusion clusterings for `normalise2`. In none of these clusterings can two clusters be fused, because that would violate a non-fusible edge.

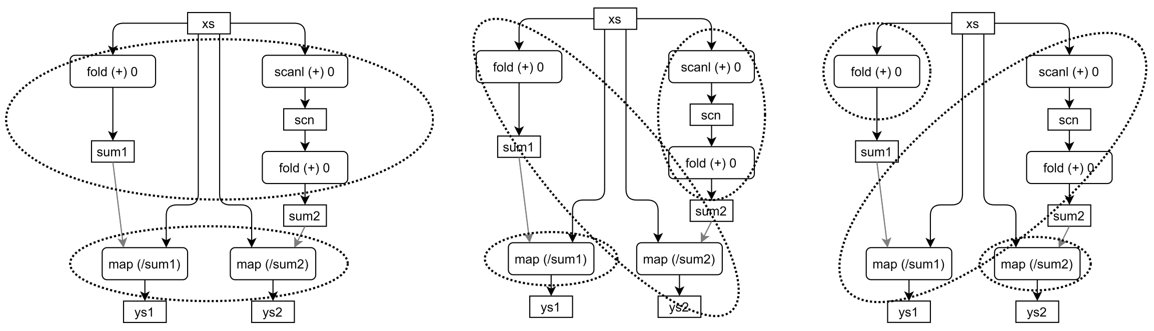


Figure 10: Three locally optimal fusion clusterings.

The core idea behind the ILP has been discussed in Section 2.5. To summarise:

- Numeric π variables are assigned to each node, these label the clusters. All nodes with the same π value are in the same cluster, and the clusters can be computed in order from 1 to n. Each cluster can potentially have a data dependency on any of the previous clusters, but not on any of the later ones: We require an acyclic clustering.

- Binary fusion variables are assigned either to each pair of nodes (first, inefficient formulation and the formulation that supports filters), or to each edge (second, more efficient formulation).

- Linear constraints relate these variables to each other.

The choice to solve an NP-hard problem at compile time, in order to optimise the runtime of a program, means that we have to accept that compilation might take longer than usual: Programs that would otherwise compile in seconds, could potentially take minutes in the ILP solver. Because of this, we take care to keep the ILP formulation as small as possible, eliminating many redundant variables and constraints.

As in Megiddo [20], our formulation only has variables and constraints for 'relevant' edges. Megiddo's ILP formulation uses a fusion variable only for each edge in the graph. We use one for each edge and one for each pair of *sibling*[11] nodes in the graph. This means that the number of constraints is quadratic in the outgoing degree of the nodes, but in practice this degree tends to be small. The reason that the formulation by Robinson et al. [21] is bigger, is that their approach uses a constraint, and thus a variable, relating each pair of nodes of different *iteration size* to their parent nodes. Because solving ILPs is computationally expensive, and worst-case exponential, we avoid having the ILP be quadratic in the size of the AST. In Accelerate, we can get away with an ILP that is roughly linear in the size of the AST, due to the following line of reasoning:

When we only have variables and constraints for local relations, as in Megiddo [20], global fusion relations can be extracted in two ways: Through the π variables (when two nodes belong to the same cluster, they should be fused), and through transitive fusion relations (when nodes x and y are both fused with node z, then x and y are also fused with each other). We note that, necessarily, the transitive relationship implies the π relationship but the reverse does not always hold. In other words, the π relationship describes a fusion clustering that is at least as fused as the transitive relationship. If we instead have fusion variables for each pair of nodes, these two relationships always lead to the same clustering.

In Megiddo [20], the version with π variables is used. We cannot guarantee that this π property is strong enough, so we must choose the other, more restrictive, option. This means that, sometimes, we might not fuse two clusters that could have fused. In return, we know that the subgraph of the program that belongs to each cluster is now always connected.

In the context of a language like Accelerate, not fusing these independent clusters turns out to be a good thing, even when fusing them would be possible: The only cost is the, often negligible, overhead of loop counters, and in return we can schedule these independent clusters concurrently.

To deal with Accelerate's particular needs, for each node we define *input direction, shape* and *output direction, shape* variables. These variables relate to the fusion variables according to two simple rules:

- If two connected nodes are fused, the output shape and output direction of the first node must be identical to the input shape and input direction of the second node;

- If two sibling nodes are fused, the input shape and input direction of both nodes must be identical.

The direction variables describe the order in which, respectively, the input or output arrays are read or written. For example, scans have a particular direction in which they have to be evaluated, and (back)permutes have an unpredictable order. This means that `scanl . scanl` and `permute . backpermute` are both vertically fusible, whilst `scanr . scanl` and `backpermute . permute` are not.

[11]Two nodes are siblings in the graph if they have the same ancestor.

In the case of unpredictable memory access patterns, it might be tempting to simply never fuse. However, this is too restrictive: For example, it is perfectly feasible to fuse `backpermute . map`, it simply means we evaluate the `map` out of order. Note that it depends on the size of the input and output arrays whether this fusion makes the map cheaper or more expensive, but even if it duplicates work, fusion can be beneficial. In the future, array size inference and function cost analysis could provide the ILP with information to make this decision.

The rest of the ILP consists of node specific constraints: they describe which nodes are allowed to be fused under which circumstances, and which invariants must hold. Specifically, they all put restrictions on the shape and direction variables. For example, in any clustering, the input direction and output direction must match for each `map` node, and a `scanl` node has to be evaluated left-to-right. All of these constraints follow trivially from the imperative-loop intuition for the corresponding node.

Finally, we experiment with some GPU-specific *shape* variables. Where the direction variables specify the *order*, these specify the *location* of the array is in memory. Concretely, this is a number representing how many dimensions each thread or threadblock contains.

4.4 Explicit Fusion AST

The previous section described a method to identify the optimal fusion clustering for a given problem, but there is still some work to do before we reach our end goal, which is code where those loops have been fused. In this section we show that we cannot perform this fusion as a source-to-source rewrite, and propose a representation that allows us to delay this until a later stage, where it is possible to perform fusion.

We cannot fuse on a normal Accelerate AST, as it simply is not expressive enough. For example, we could fuse a `map` and a `fold` over the same array horizontally, but we cannot write this fused function in the normal Accelerate AST: There is no way to construct something that produces two arrays of different dimensionality, and automatically gets compiled into a single loop. Instead, we will use the information from the graph and the ILP solution to explicitly annotate the AST with labels that describe which nodes should be fused, and in what fashion. We will then utilise this new AST to perform the actual fusion at code generation time.

We want to annotate the AST with detailed instructions on how to fuse each of the operators. Each node of the AST should consist of a tree of fusions, and only the leaves of these trees contain the PreOpenAcc datatype. We could describe our example, the fused first half of `normalise2` in Figure 4b, as the fusion tree in Figure 11.

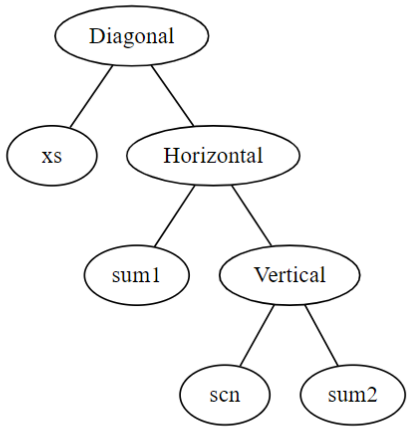


Figure 11: Fusion tree of the first half of `normalise2`

```haskell
data Plurality = Single | Multiple
data PreFusedOpenAcc (n :: Plurality) aenv a where
  RootOfCluster :: PreFusedOpenAcc Single   aenv a
                -> PreFusedOpenAcc Multiple aenv a

  LeafOfCluster :: AST.OpenAcc            aenv a
                -> PreFusedOpenAcc Single aenv a

  Unfused    :: AST.PreOpenAcc (PreFusedOpenAcc Multiple) aenv a
             -> PreFusedOpenAcc Multiple                  aenv a

  Vertical   :: LeftHandSide a aenv benv
             -> PreFusedOpenAcc Single aenv a
             -> PreFusedOpenAcc Single benv b
             -> PreFusedOpenAcc Single aenv b

  Horizontal :: PreFusedOpenAcc Single aenv  a
             -> PreFusedOpenAcc Single aenv    b
             -> PreFusedOpenAcc Single aenv (a,b)

  Diagonal   :: LeftHandSide a aenv benv
             -> PreFusedOpenAcc Single aenv  a
             -> PreFusedOpenAcc Single benv    b
             -> PreFusedOpenAcc Single aenv (a,b)
```

Listing 1: The explicit fusion closure

We want to embed this information in the AST. We do this by defining a new recursive knot, as seen in Section 3.3.2. In Listing 1, the definition of this explicit fusion AST is shown, and Listing 2 shows how this encapsulates the fusion of `normalise2`.[12] We use the GHC extension `DataKinds` to make the phantom types Single and Multiple of kind Plurality. The AST of an Accelerate program will, in general, consist of multiple clusters. The phantom type of kind Plurality disambiguates whether an AST node is inside one of the clusters (Single), or a high-level node that contains multiple clusters (Multiple), such as a let-binding or array-level control flow. The nodes that contain multiple clusters are all under the constructor `Unfused`, which has a recursive closure back into the explicit fusion tree. Each cluster has a single `RootOfCluster` node, underneath which zero or more of the `Vertical`, `Horizontal` and `Diagonal` nodes form a tree. In the leaves of this tree, `LeafOfCluster` nodes contain regular AST nodes. Note that these final nodes do not have a `PreFusedOpenAcc` closure, as they are supposed to only contain very minimal nodes, like a single map or fold.[13]

The three fusion constructors reflect the patterns that we encountered before. They use the `LeftHandSide` constructor we encountered previously to encode the relationship between the environments: For both vertical and diagonal fusion, the `LeftHandSide` tells us that `benv` is `aenv`, extended with `a`. As always, the final type constructor (`b` or `(a, b)`) in this GADT represents the type of the value that the corresponding AST evaluates to. Note that these reflect the idea that horizontal and diagonal fusion return both `a` and `b`, whilst vertical fusion fuses `a` away.

[12]The function definitions (map, fold, scanl) have been simplified in this snippet, instead of expanding the entire Accelerate AST terms.

[13]In fact, we could give these nodes a closure that only allows indexing to statically guarantee this. This is omitted for the sake of clarity.

```
RootOfCluster $ Diagonal
  LeftHandSideArray
  LeafOfCluster $ map f
  Horizontal
    LeafOfCluster $ fold (+) 0
    Vertical
      LeftHandSideArray
      LeafOfCluster $ scanl (+) 0
      LeafOfCluster $ fold  (+) 0
```

Listing 2: Fused `normalise2` in the fusion AST

An important note at this point, is that there is no restriction on the shape of the results for any of the fusion constructs: It is for example, by design, possible to horizontally fuse a fold and a map over the same array, even though the result types have different dimensionality. This parallel concept cannot be expressed in the current delayed/manifest representation, regardless of how it is constructed. Of course, not every fusion that can be expressed in this AST is actually viable, as we could not make all impossible states unrepresentable.

Adding more type-level safety, ensuring that all fusions that can be encoded are valid in some broader sense, is left for future work. Possible avenues include statically ensuring that both steps in horizontal fusion loop over the same source, and that the second step in both vertical and diagonal fusion can use the first result as a streaming parameter.

4.5 Generating fused code

In this section, we will look at how to transform the annotated AST we introduced in the previous section into fused, low-level code. We cannot perform this step before code generation in a normal Accelerate AST, as it simply is not expressive enough. For example, we could fuse a `map` and a `fold` over the same array horizontally, but we cannot write this fused function in the normal Accelerate AST. This is why we introduce the explicit fusion AST in Section 4.4, which manages to delay this problem until code generation. Fusing these loops once they have been completely generated into target code, such as LLVM IR, is also not an attractive option, as these low-level loops are too expressive: Blindly fusing them is unsafe as they may contain various side-effects, and it might not be trivial to figure out how to fuse them exactly.

In fact, that is very close to the approach that *flow fusion* takes [7]. In flow fusion, fusible functions are identified and then converted into explicit loops, which are then fused with no regard for safety: If the first phase incorrectly classifies functions as fusible, the fusion process will proceed and generate incorrect loops.

The approach we detail in this thesis differs from flow fusion on two main fronts: We add type safety, which is especially useful in the Accelerate compiler where the typechecker is leveraged across the entire pipeline, and we make sure that all the explicit loops can be executed in parallel, which allows us to generate code for the parallel hardware that Accelerate targets.

We accomplish this by defining a strongly typed, restrictive representation for these abstract loop nests. We will refer to this as the *intermediate representation* (IR) from now on. This intermediate representation should be expressive enough to allow transpiling Accelerate into it, while being restrictive enough to allow safe fusion of, arbitrary but fusible, loop nests. The code generation of a fusion-annotated AST will then consist of the following steps:

1. Transpile each Accelerate instruction into the IR.
2. Directly fuse the IR of parts that were annotated to be fused.
3. Generate code for the resulting IR.

The high-level idea of this representation is to expose the array-level control flow, using one looping constructor and a couple of strongly typed composition constructors. The loop is the `For` constructor of the `IntermediateRep` GADT. Contrary to all user-facing Accelerate functions, this `For` loop is closer to the imperative code that gets generated: It iterates over the outermost dimension of the data, whereas all functions on the user-facing end work on the innermost dimensions. The intermediate representation is designed in such a way that while these `For` loops may be nested and composed with statements at each level, there are never two loops at the same *depth*. This gives a static guarantee that every instance of this IR represents a fully fused program: It can easily be evaluated or compiled in such a way that memory reads and writes are minimised. Moreover, since the `For` loop is by design a parallel construct in this language, we can also generate fused code for parallel backends from such a program. Section 5 details how this representation is implemented in Accelerate.

As a motivating example, let us look at the first half of `normalise2` again. Since it can be fully fused, our goal is to generate low-level code for it containing only one loop. First, we translate each of the functions into an intermediate representation. The resulting pieces of explicitly loopy ASTs are shown below. Higher-level Accelerate code is shape-polymorphic, but in the process of code generation we have to instantiate it with a code that works on specific arrays of given dimensionalities. Explicit loops means that this instantiation has already happened.

The implementation of `IntermediateRep` belongs in Section 5, this example serves as an informal introduction to it and the underlying ideas, which we will discuss here. We do not advise the reader to dive into the example in-depth here, the takeaway is that all four of these have their own `For` loop, inside of which is some computation.

```haskell
xs :: IntermediateRep ((), Vector Int) () ((), Vector Int)
xs = For (Simple $ Take' (OneToOne (\(PushEnv _ ys) _ ->
  get >>= \i -> modify (+1) >> return (ys ! (Z :. i)))) Base) Id t1

sum1 :: IntermediateRep ((), Vector Int) ((), Vector Int) ((), Scalar Int)
sum1 = For (Simple $ Take (ManyToOne (const $ modify . (+)) get) Base) t1 t2

scn :: IntermediateRep ((), Vector Int) ((), Vector Int) ((), Vector Int)
scn = For (Simple $ Take (OneToOne (\_ a -> modify (+a) >> get)) Base) t1 t1

sum2 :: IntermediateRep ((), Vector Int) ((), Vector Int) ((), Scalar Int)
sum2 = sum1

-- Some index space transfomations, to embed computation inside the loops
t1 :: Transform ((), Scalar Int) ((), Vector Int)
t1 = Fn 1 Id
t2 :: Transform ((), Array Neg1 Int) ((), Scalar Int)
t2 = Fn 0 Id
```

If we can write the entire first half of `normalise2` as one such `IntermediateRep`, we have successfully fused their imperative versions. All that would then remain is to translate the resulting `IntermediateRep` into a real imperative language.

Because of the way we designed this intermediate representation, it is relatively easy to write this fused version by hand. Once again, the example is too convoluted to examine in-depth, but the main takeaway is that we now have one `For` loop, inside of which `inner` contains all of the loop bodies:

```
let xs          = Take' (OneToOne (\(PushEnv _ ys) _ ->
      get >>= \i -> modify (+1) >> return (ys ! (Z :. i)))) Base
    sum1        = Take (ManyToOne 30 (const $ modify . (+)) get) Base
    sum2        = Weaken (Fn 0 (Skip Id)) $ Simple $
      Take (ManyToOne 30 (const $ modify . (+)) get) Base
    scn         = Take (OneToOne ref (\_ a -> modify (+a) >> get)) Base
    temp        = PushEnv EmptyEnv . Array (fromElt Z) <$> newArrayData 1
    sum2scn     = Before (Partition (Skip Id) (Fn 0 (Skip Id))) temp scn sum2
    sum1sum2scn = Besides (Partition (Skip Id) (Fn 0 (Skip Id))) sum1 sum2scn
    inner       = Before' (Partition (Skip Id) Id) (Partition
      (Skip (Skip Id)) (Fn 0 (Fn 0 (Skip Id)))) xs sum1sum2scn
in For inner Id (Fn 0 (Fn 0 (Fn 1 Id)))
```

Here, we fused the loops and manually fix the environments (using transformations like `Skip`). Sadly, this is not quite sufficient: We cannot expect a programmer to manually fuse `IntermediateRep` structures halfway in the compilation phase, this is a process which must happen automatically. Given the first definitions of `xs, sum1, scn, sum2` (the ones that each have their own loop), we want to have functions that can fuse them automatically, like this:

```
program = fuseDiagonal xs $ fuseHorizontal sum1 $ fuseVertical scn sum2
```

Once we manage this, we could use the explicit fusion AST to generate fused (intermediate representation) code.

5 Implementation in Accelerate

In this chapter, we tie all the pieces from chapter 4 together and show how they are, partially, implemented in the Accelerate compiler.

5.1 Fusion phase

In order to generate an ILP formulation corresponding to a program, we add some intermediate steps. First, the AST is rewritten into one where each relevant node has a unique label, which will serve to name the ILP variables, and where all combinators work only on variables. As a concrete example, below are excerpts from the normal Accelerate AST, and the AST used for fusion analysis:

```
-- normal version
data PreOpenAcc acc aenv a where
  Map         :: (Shape sh, Elt e, Elt e')
              => PreFun     acc aenv (e -> e')
              -> acc            aenv (Array sh e)
              -> PreOpenAcc acc aenv (Array sh e')

-- fusion version
data PreLabelledAcc acc aenv a where
  Map         :: (Shape sh, Elt e, Elt e')
              => NodeId
              -> PreFun         acc aenv (e -> e')
              -> ArrayVars          aenv (Array sh e)
              -> PreLabelledAcc acc aenv (Array sh e')
```

Note that in the `PreLabelledAcc` version, the `Map` constructor has an extra field for its `NodeId`, and the array that it maps over is always a variable.

A different option would have been to use the existing AST, and encode the `NodeId` in the *recursive knot*. There are two reasons why we did not go for this option:

1. Not every AST node should have a `NodeId`. In particular, we do not want to assign a name to let-binding nodes, because they are not relevant for the ILP, and the global transformations that will happen in the fusion process will create new binding nodes.

2. Statically ensuring that each combinator works on a variable helps to simplify the whole process. This translation needs to happen at some point, because it is needed for the explicit fusion AST, so adding it while labelling all nodes is a sensible option.

The corresponding excerpt of the translation can be found below. The State monad is used to label each node individually. Here, `letBind` abstracts a very common pattern in the `letBindAcc` function: it binds `acc` to a variable in an `Alet` node, and it embeds the continuation inside it. `w $:> letBindFun f` is shorthand for applying the weakening `w` to the AST `letBindFun f`. This is needed, because let-binding the array `acc` means that `f` now operates in a larger environment, as it has access to `acc`.

```
letBindAcc :: AST.OpenAcc aenv a
           -> State Int (LabelledOpenAcc aenv a)
letBindAcc (AST.OpenAcc pacc) = LabelledOpenAcc <$> case pacc of
AST.Map f acc -> letBind acc $ \w var ->
    Map <$> getInc <*> w $:> letBindFun f <*> var
```

Now we have this new, labelled and let-bound, AST, we can use it to generate the ILP. First, a single pass over the AST abstracts away all the irrelevant details to make a Directed Acyclic Graph (DAG), represented as a list of nodes. These nodes contain only the important details: their label, the label of their input variables, and the type of node (map, fold, etc). We also generate a list of fusion preventing edges. Most of these occur as a result of indexing, as we can only safely index into an array if it is already fully computed in an earlier cluster. An exception to this rule occurs when we index into an element-wise array: If each element can be computed without nested parallelism, we could inline the *delayed array* representation instead. Identifying these opportunities, and weighing them against possible work duplication, is left for future work.

```
data DirectedAcyclicGraph = DAG {
  nodes :: IM.IntMap NodeType,
  fpes :: [(NodeId, NodeId)]
}
```

Subsequently, this DAG is used to create the ILP, consisting of the variables and constraints mentioned previously. The ILP determines a clustering of the nodes. The entire pipeline up to this point has been succesfully implemented. What is left, is to use the weakening and strengthening operations to rewrite the program into a tree of these clusters, and finally, rewrite each of these clusters into the explicit fusion AST we encountered in Section 4.4

5.2 Code generation phase

In this section, we look at the code generation phase and how it changes due to this new approach to fusion. As mentioned before, we translate the explicitly fused Accelerate AST into some intermediate representation, that we can directly perform fusion on. Afterwards, we generate code for the fused version of this representation. Most of the contents in this Section will have to be re-implemented for each backend, as it is intertwined with code generation itself.

In the code in this section, the AST will include regular Haskell-level functions to denote element-level expressions. This came about as a result of the language being written for a Haskell-level interpreter, and is kept that way here for the sake of simplicity. For code-generating backends, some slight alterations need to be made to only one GADT in this language. Parts of the language are purposefully, but only slightly, altered here for the sake of simplicity. For example, typeclass constraints are omitted from nearly all functions and data types: they are usually not instructive in this context, as much as they are just boilerplate or leaking implementation details. The uncensored versions can be found on GitHub.

5.2.1 Intermediate Representation

The first step in the code generation phase is to translate the individual Accelerate AST nodes into the IR. This is not implemented yet, the main challenge here is to translate the contained `Exp` expressions. Luckily, for parallel backends, where the IR will contain the `Exp` expressions instead of haskell-level functions, this part is irrelevant.

Like the Accelerate AST, the GADTs for this intermediate representation are strongly typed. The types encode the environment that these programs work on, which are tuple-based lists of arrays. We need to be able to do index transformations on these arrays and environments. Since these appear everywhere in the design, it is best we discuss them up front:

```
data Transform from to where
  Id :: Transform a a
  Fn :: Int
     -> Transform from to
     -> Transform (from, Array sh e) (to, Array sh' e)
  Skip :: Transform from to
       -> Transform from (to, Array sh e)
```

The intuition behind a `Transform`, is that it is a constructive proof that `from` is a subset of `to`. The constructors `Id` and `Skip` very clearly resemble this relationship: Together, they allow us to *weaken* an environment, such that we can embed a program into one where the environment is bigger. `Fn` is less obvious, partially because there is no type-level guarantee that this subset relationship holds. In practice, the way `Fn` is used is inside of loops: You can embed a program that consumes a 1-dimensional array inside of a loop, to make a program that consumes a 2-dimensional array. The Int represents the offset that should be added to the virtual index, to get the physical index in the manifest array.

The astute reader might have noticed that using only `Skip` and `Id` we can only skip arrays from the top of the environment, as there is no `Take` constructor. We get around this by using `Fn 0` as a synonym for `Take`, when `sh` and `sh'` are the same shape.

Currently, there is no type-level guarantee that relates the shapes `sh` and `sh'` in `Fn`. Doing so is left for future work, one of the possibilities is to add a GADT `GreaterShape` as such:

```
data Transform from to where
  Fn :: GreaterShape sh sh'
     -> Int
     -> Transform from to
     -> Transform (from, Array sh e) (to, Array sh' e)

data GreaterShape sh sh' where
  Equal   :: GreaterShape sh sh'
  Greater :: GreaterShape sh sh'
          -> GreaterShape sh (sh :. Int)
```

As with other parts, like the explicit fusion AST, we attempted to make this functional before adding more typesafety, making less illegal states representable.

Transforms compose naturally, due to the transitivity of the subset relationship. We can even define an instance of `Category`:

```
compose :: Transform b c -> Transform a b -> Transform a c
compose Id x = x
compose x Id = x
compose (Skip x) y = Skip (compose x y)
compose (Fn _ x) (Skip y) = Skip (compose x y)
compose (Fn i ab) (Fn j bc) = Fn (i+j) (compose ab bc)

instance Category Transform where
  id = Id
  (.) = compose
```

For brevity, we also introduce a `Partition`, which is nothing more than two `Transforms` that point to the same manifest environment and a promise that the resulting environments are disjoint. `Partitions` inside of the IR should generally represent a true partition, that is, they should also cover the entire source environment. In the interpreter, some notation is abused and the second condition does not necessarily hold, as the partition is then fused with weakening `Transforms`.

```haskell
data Partition a b ab = Partition (Transform a ab) (Transform b ab)
```

The main meat of this language is the GADT `IntermediateRep`:

```haskell
data IntermediateRep permanentIn tempIn out where
  For  :: IntermediateRep pin tin out
       -> Int
       -> Transform tin tin'
       -> Transform out out'
       -> IntermediateRep pin tin' out'
  Weaken :: Transform a a'
         -> IntermediateRep p a  b
         -> IntermediateRep p a' b
  Simple :: LoopBody pin a b
         -> IntermediateRep pin a b

  -- Vertical fusion
  Before  :: Partition a b ab
          -> ValArr b
          -> LoopBody        pin a  b
          -> IntermediateRep pin ab c
          -> IntermediateRep pin a  c
  After   :: Partition a b ab
          -> ValArr b
          -> IntermediateRep pin a  b
          -> LoopBody        pin ab c
          -> IntermediateRep pin a  c

  -- Diagonal fusion
  Before' :: Partition a b ab
          -> Partition b c hor
          -> LoopBody        pin a  b
          -> IntermediateRep pin ab c
          -> IntermediateRep pin a  hor
  After'  :: Partition a b ab
          -> Partition b c hor
          -> IntermediateRep pin a  b
          -> LoopBody        pin ab c
          -> IntermediateRep pin a  hor

  -- Horizontal fusion
  Besides :: Partition b c hor
          -> LoopBody        pin a b
          -> IntermediateRep pin a c
          -> IntermediateRep pin a hor
```

As mentioned before, this language is mainly focussed on the control flow and the types of the in- and output environments. In this GADT, the outside environment is called `permanentIn`, abbreviated to `pin`. This consists of arrays that are not fused into this IR. The type variable `tempIn`, abbreviated to `tin`, represents input variables that this IR fuses away. The type variable `out` contains the type of the output environment that this IR produces.

We need to separate these environments, to be able to recursively embed instances of this IR inside of the loops. The temporary input and output variables grow with the loops, enforcing locality, whilst the permanent inputs do not, allowing us to randomly index into manifest arrays.

The most important building block in this IR is the `For` loop. Though it is not statically enforced in the types, this loop *should* reduce the dimensionality of every array in both the temporary inputs and the outputs by one. More specifically, it represents iterating over the outermost dimension, contrary to how the source language Accelerate functions are defined. This means that the existence of a `For` implies that all the arrays in both the temporary input and the output environments should have identical outermost dimensions. This is indeed one of the necessary conditions that allow us to completely fuse arrays in parallel computing contexts. All the other shown constructors represent the various ways to compose a `LoopBody` with an `IntermediateRep` that may contain a `For` loop. Note that it is impossible to sequentially compose two loops: this language cannot express unfused structures. The differences between the vertical, diagonal, and horizontal constructors lie in the way they handle the type variables.

One of the concepts that seems inherent to this design is the notion of an `Array` of dimension *negative one*. Such an array cannot exist, but being able to construct and reason about an `IntermediateRep` that consumes or produces an array of this dimension starts to make sense once we embed it in a `For` loop.

Consider a normal Accelerate fold, over a two-dimensional array: The innermost loop traverses the input array, combining its intermediate result with each element, one at a time. Once this loop has *finished*, it can write the result into the output array. The shapes in the type of the complete program should be `DIM2 -> DIM1`, and the shapes in the type of the program inside of the outermost loop should be `DIM1 -> DIM0`. But what kind of type can we assign to the innermost loop body? Here, we choose to say that its type should be `DIM0 -> Neg1`, which keeps true to the invariants of the `For` loop, that the input and output shapes should increase by one.

```
-- Accelerate:
example1 :: Acc (Array DIM2 Int) -> Acc (Array DIM1 Int)
example1 = fold (+) 0

// Imperative version:
for xs in xss
  t := 0       // initialise
  for x in xs
    t += x     // combine
  y := t       // assign
```

In the imperative version, we say that inside of the innermost for-loop, `t` has dimension *negative one*: It is a partial result for what will be a scalar (zero-dimensional array) outside of the loop.

Note that we do not need more negative dimensions: Even if we vertically fuse two folds, we can stack the current implementation. Naively interpreting or compiling this might produce unnecessary temporary values like `t3` below, but removing those later should not be a difficult task (one would expect the low level optimisations to take care of that). Regardless, it should not be counted as a memory read or write as it is a local scalar value.

```
-- Accelerate:
example2 :: Acc (Array DIM3 Int) -> Acc (Array DIM1 Int)
example2 = fold (*) 1 . fold (+) 0
```

```
// Imperative version:
for xss in xsss
  t1 := 1        // initialise
  for xs in xss
    t2 := 0      // initialise
    for x in xs
      t2 += x    // combine
    t3 := t2     // assign
    t1 *= t3     // combine
  y := t1        // assign
```

Since `LoopBody` appears so frequently in the `IntermediateRep` above, we will discuss it next. Important to note is that while `IntermediateRep` is quite universal, and essential to the design of this fusion, `LoopBody` is much more backend-specific. The version that we will see now is only suitable for the sequential interpreter, but by replacing the Haskell-level functions inside with `Exp`-like Accelerate ASTs, the same ideas should be applicable to code generating backends. Some implementation details like `IORef`s will also show up here, as they make it much simpler to write an interpreter. Again, these are really artifacts that belong to the interpreter, and not inherent to the idea of a `LoopBody`. `LoopBody` consists of two GADTs: `LoopBody'`, which contains the statements to convert one array into another one, and `LoopBody`, which may consist of multiple such `LoopBody'`s and ties them into the environment.

```
data LoopBody permIn tempIn out where
  Base :: LoopBody  pin () ()
  Weak :: Partition a rest a'
       -> Partition rest b b'
       -> LoopBody p a b
       -> LoopBody p a' b'
  Use  :: Array sh e
       -> IORef Int
       -> LoopBody pin a b
       -> LoopBody pin a (b, Array DIM0 e)
  Take :: LoopBody' pin (Array sh1 a) (Array sh2 b)
       -> LoopBody  pin c d
       -> LoopBody  pin (c, Array sh1 a)  (d, Array sh2 b)

data LoopBody' permIn tempIn out where
  OneToOne :: IORef s
           -> (ValArr pin -> a -> State s b)
           -> LoopBody' pin (Array DIM0 a) (Array DIM0 b)
  ManyToOne :: IORef (s, Int)
            -> Int
            -> (ValArr pin -> a -> State s ())
            -> State s b
            -> LoopBody' pin (Array DIM0 a) (Array Neg1 b)
  OneToMany :: IORef (s, [b], Int)
            -> Int
            -> (ValArr pin -> a -> State s [b])
            -> LoopBody' pin (Array Neg1 a) (Array DIM0 b)
```

A `LoopBody'` describes how to turn a single array into another one. Maps, but also scans, can be implemented by the `OneToOne` constructor. The `ManyToOne` constructor allows us to implement folds, and the `OneToMany` constructor is useful for unfolds. While unfolds in general are not a part of Accelerate, they include replicates and allow arbitrary generates. An implementation aimed at generating code for parallel hardware probably needs to reconsider this constructor.

An important detail in `LoopBody'` is the dimensionality of the input and output arrays: The `OneToOne` constructor has scalar in- and outputs, but in the other two constructors we see the negative dimensions that were discussed earlier resurface: Each iteration of a `ManyToOne LoopBody'` updates the partial result, but only when we step out of the loop will we have a complete scalar result of type `Array DIM0 b`.

`LoopBody` describes how we can turn one environment into another environment, by applying `LoopBody'`s, weakening and introducing new arrays.

5.2.2 Fusing the Intermediate Representation

Fusing arbitrary `IntermediateRep` programs is now a matter of writing the functions that belong to the following type signatures. Since each IR has up to eight possible constructors, these functions tend to get quite lengthy. However, thanks to the strongly typed representation, it is fairly doable to let the types guide us in this development and most cases turn out to be easy. Nevertheless, these functions are not yet fully implemented.

```
weakenIR :: Transform a b
         -> IntermediateRep p a c
         -> IntermediateRep p b c

fuseHorizontal :: IntermediateRep p a' b
               -> IntermediateRep p a'' c
               -> Partition a' a'' a
               -> Partition b c d
               -> IntermediateRep p a d

fuseVertical :: IntermediateRep p a  b
             -> IntermediateRep p ab c
             -> Partition a b ab
             -> IntermediateRep p a  c

fuseDiagonal :: IntermediateRep p a  b
             -> IntermediateRep p ab c
             -> Partition a b ab
             -> Partition b c d
             -> IntermediateRep p a  d
```

Though the strongly typed environments and `Transforms` guide the writing of these functions, they do not make them more readable. For example, below are some exerpts from the implementation of `fuseHorizontal`, showing three key cases: Fusing two loops recursively, and the two variations of fusing a single loopbody with a recursive loop nest. These implementations use the omitted functions `mkSmth` and `trivialPartition` to generate the appropriate `Transforms`. `Something` is used to generate the `Transforms` inside of the loops, there are actually multiple viable options here.

```
fuseHorizontal (For bIR n'  ti'  to')
               (For cIR n'' ti'' to'')
               (Partition aa' aa'') (Partition bd cd)
                | n' == n''
                = case   mkSmth aa' aa'' ti' ti'' of -- input
                  Exists'   (Something aa1 a1a1' a1a1'') ->
                    case mkSmth bd cd to' to'' of -- output
                    Exists' (Something dd1 b1d1 c1d1) ->
                      For (fuseHorizontal bIR cIR a1a1' a1a1'' b1d1 c1d1) n' aa1 dd1

fuseHorizontal (Simple lb) ir (Partition aa' aa'') (Partition bd cd) =
  Besides (Partition bd cd)
          (Weak (Partition aa'  mkSkips) trivialPartition lb)
          (Weaken aa'' ir)

fuseHorizontal ir (Simple lb) (Partition aa' aa'') (Partition bd cd) =
  Besides (Partition cd bd)
          (Weak (Partition aa'' mkSkips) trivialPartition lb)
          (Weaken aa'  ir)

trivialPartition :: Partition () a a
```

5.2.3 Evaluating the Intermediate Representation

The only backend that is currently implemented for this fusion process is the interpreter. Given the inputs and an environment consisting of allocated mutable arrays to hold the output, the interpreter is able to successfully compute the output arrays and store them. It does so sequentially, as the interpreter runs inside Haskell, but on a completely fused IR.

When evaluating a complete Accelerate program that is fused into multiple clusters, we evaluate each cluster in turn and use the resulting environment as the *permanent input* of the subsequent clusters.

6 Results

In Figure 12, some analytical results are presented on three versions of `normalise2`. The first version corresponds with the declarative, unfused code. The second version shows what happens to the presented metrics when subjected to stream fusion, or fusion through delayed arrays.[14] Finally, we also show the optimally fused version. We also present the same metrics, applied to the expanded version of `normalise2`: Here, the input array `xs` is replaced with a `map`, which allowed us to demonstrate diagonal fusion in Section 4.1.

	`normalise2`			expanded `normalise2`		
	Unfused	Stream	Optimal	Unfused	Stream	Optimal
Loops	5	4	2	6	5	2
Array reads	$5n+2$	$4n+2$	$2n+2$	$6n+2$	$5n+2$	$2n+2$
Array writes	$3n+2$	$2n+2$	$2n+2$	$4n+2$	$3n+2$	$3n+2$

Figure 12: The effects of different fusion techniques on `normalise2`

In these results, n refers to the length of the input list, which corresponds with the length of the output lists. These results are based on a counting argument from reading the code, no experimental tests were performed. However, as GPU programs are often memory-bound, these numbers give a rough indication of the relative runtime performance of the variations of `normalise2`: The optimally fused version has about half the number of array reads and writes, which would imply a significant speedup.

In these tables, we notice that in both cases, stream fusion eliminates one read and one write per element. Optimal fusion is able to improve upon that by eliminating another two or three reads per element, but no more writes. This reflects the idea that stream fusion and delayed arrays are good at vertical fusion, eliminating the intermediate result, but do not perform horizontal or diagonal fusion. Horizontal and diagonal fusion, by themselves, only eliminate reads, not writes. Of course, there are cases where horizontal or diagonal fusion is necessary to enable vertical fusion, in which case they can eliminate writes.

To assure that the *intermediate representation* based on explicit loops can be evaluated, we wrote an interpreter and tested it on `normalise2`. This interpreter works in Haskell, which makes it unsuitable for performance comparisons, but it demonstrates that this explicitly loopy representation can be evaluated.

[14] Though different, these two fusion mechanisms amount to the same fusion clustering in this case.

7 Discussion and conclusion

The main question we set out to answer with this thesis is: *How can we optimally fuse all programs in a data-parallel language?* This question reflects the gap in the existing literature between algorithms that find optimal fusion clusterings for sequential programs, and non-optimal fusion systems that are applicable to data-parallel languages. None of these existing solutions find the optimal fusion clustering in a data-parallel context, and generate low-level code that accomplishes this fusion clustering.

In order to answer the main question, we identified several subquestions. These were applied to the case study of Accelerate. Because not all fusion opportunities can be expressed in the domain specific language, the first subquestion is concered with expressing a fusion clustering in the AST. This revealed that the existing terminology of *vertical* and *horizontal* fusion was not sufficient, so the term *diagonal fusion* was coined and formalised. Diagonal fusion could be described as a variant on vertical fusion, whilst also storing the intermediate result. Fusion clusters can be expressed in the AST as a tree of these three concepts.

The second subquestion this thesis answered concerns the algorithm that identifies the optimal fusion clustering. An existing *Integer Linear Programming* solution for sequential programs was adapted to a data-parallel context. This new formulation only has variables and constraints that are local in the program graph, which keeps the size of the ILP roughly linear with respect to the size of the program. We also added *direction* and *shape* variables, and proposed a method to encode in the ILP when random access functions like `permute` can safely be fused.

Finally, we also investigated the impact on code generation. An *intermediate representation* was designed to be both expressive and restrictive enough to allow translating Accelerate into it, to allow fusion of instances of it, and to allow code generation from it. This intermediate representation is strongly typed, and utilises the formalisation of *vertical, horizontal* and *diagonal* fusion to leverage the type system for static correctness guarantees.

In conclusion, we argue that whilst ambitious, it is feasible to perform optimal fusion in a compiler for a purely functional data-parallel language, by separating fusion into distinct phases and delaying actual fusion until the code generation phase.

The underlying principle in this thesis, which is to explicitly annotate the constructs that should be fused and delay fusing them until a later stage, could be applied to a much broader range of applications. In this case, we were interested in performing fusion while retaining the ability for parallel execution. One could possibly adopt the entire approach for a programming language different context, with other limitations or concerns. For example, this technique might be applicable in a language where we are not interested in data parallelism, but in concurrency. Or alternatively, a compiler that generates formal proofs of correctness alongside programs or performs automatic differentiation might also have slightly different fusion constraints. Amongst smaller changes, the ILP formulation would need to be redesigned to reflect the fusion constraints for the domain at hand.

7.1 Future work

There are some limitations of the proposed method which should be addressed by future research: First, the current implementation has not been finished, and some of the gaps might pose challenges. In particular, the large-scale rewriting to group nodes that should be fused has not been attempted, and there is only an interpreter for the intermediate representation.

Secondly, it would probably be more accurate to dissect high-level nodes into individual kernels before generating the ILP: Highly parallel implementations of functions like a fold consist of multiple kernels that cannot be fused with each other. As such, it makes little practical sense to fuse this whole fold with something else: Instead, we could argue to fuse 'the last kernel of a fold with the first kernel of a scan', for example. This way, we could find even more fusion opportunities and have a representation that does not contradict itself.

Finally, it would be interesting to explore the idea mentioned near the end of Section 5.1: Indexing into an array might not always be fusion preventing, if the indexed array is element-wise (in other words, if it can be represented as a delayed array). We are interested in figuring out how to incorporate this idea into the proposed solution, because there are scenarios where this leads to a more efficient fusion clustering than the proposed 'optimal' one.

References

[1] Allen Goldberg and Robert Paige. Stream processing. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 53–62, New York, NY, USA, 1984. ACM.

[2] Duncan Coutts. Stream fusion: Practical shortcut fusion for coinductive sequence types. 2011.

[3] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, 1989.

[4] Josef Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. *ACM SIGPLAN Notices*, 37(9):124–132, 2002.

[5] Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232, 1993.

[6] Joachim Breitner. Call arity. In *International Symposium on Trends in Functional Programming*, pages 34–50. Springer, 2014.

[7] Ben Lippmeier, Manuel MT Chakravarty, Gabriele Keller, and Amos Robinson. Data flow fusion with series expressions in haskell. *ACM SIGPLAN Notices*, 48(12):93–104, 2013.

[8] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional gpu programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 49–60, New York, NY, USA, 2013. ACM.

[9] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. *Optimising purely functional GPU programs.* PhD thesis, University of New South Wales, Sydney, Australia, 2015.

[10] Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. In *ACM SIGPLAN Notices*, volume 52, pages 556–571. ACM, 2017.

[11] Michel Steuwer. Improving programmability and performance portability on many-core processors. 2015.

[12] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 519–530, New York, NY, USA, 2013. Association for Computing Machinery.

[13] Clemens Grelck. *Single Assignment C (SAC) High Productivity Meets High Performance*, pages 207–278. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[14] Sven-Bodo Scholz. With-loop-folding in sac-condensing consecutive array operations. In Chris Clack, Kevin Hammond, and Tony Davie, editors, *Implementation of Functional Languages*, pages 72–91, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[15] Clemens Grelck, Karsten Hinckfuß, and Sven-Bodo Scholz. With-loop fusion for data locality and parallelism. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *Implementation and Application of Functional Languages*, pages 178–195, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[16] Mathias Bourgoin, Emmanuel Chailloux, and Jean-Luc Lamotte. Spoc: Gpgpu programming through stream processing with ocaml. *Parallel Processing Letters*, 22(02):1240007, 2012.

[17] Joel Svensson. *Obsidian: GPU Kernel Programming in Haskell.* PhD thesis, Chalmers University of Technology, 2011.

[18] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled gpu functions in haskell. In *ACM Sigplan Notices*, volume 45, pages 67–78. ACM, 2010.

[19] Mads RB Kristensen, Simon AF Lund, Troels Blum, and James Avery. Fusion of parallel array operations. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 71–85. IEEE, 2016.

[20] Nimrod Megiddo and Vivek Sarkar. Optimal weighted loop fusion for parallel programs. In *SPAA*, volume 97, pages 282–291. Citeseer, 1997.

[21] Amos Robinson, Ben Lippmeier, and Gabriele Keller. Fusing filters with integer linear programming. In *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*, pages 53–62. ACM, 2014.

[22] Robert Clifton-Everest, Trevor L. McDonell, Manuel M. T. Chakravarty, and Gabriele Keller. Streaming Irregular Arrays. In *Haskell '17: The 10th ACM SIGPLAN Symposium on Haskell*, pages 174–185. ACM, September 2017.

[23] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *DAMP '11: The 6th workshop on Declarative Aspects of Multicore Programming.* ACM, January 2011.

[24] Trevor L. McDonell, Manuel M T Chakravarty, Vinod Grover, and Ryan R Newton. Type-safe Runtime Code Generation: Accelerate to LLVM. In *Haskell '15: The 8th ACM SIGPLAN Symposium on Haskell*, pages 201–212. ACM, September 2015.

[25] Gabriele Keller, Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. *ACM Sigplan Notices*, 45(9):261–272, 2010.

[26] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. North-Holland, 1972.