Gotta Go Fast

An Optimising GPGPU Compiler for a Data-Parallel Purely Functional Language

Abstract

We give a performance analysis of the purely functional language Futhark and its optimising compiler on fourteen benchmarks, and give a description of three features that contribute to achieving performance *competitive with hand-written low-level GPGPU code*.

1. A type system for in-place updates of arrays that ensures referential transparency and supports equational reasoning.
2. Two new bulk parallel operators intended to support efficient sequentialisation, along with their fusion rules.
3. Extraction of flat parallel GPU kernels through an implementation of functional *loop distribution*, which aims to exploit only as much parallelism as is cost-effective on concrete hardware.
4. Tested on fourteen Rodinia, Accelerate and FinPar benchmarks, Furthark's OpenCL code is faster than competitors by a harmonic-mean factor of $1.75\times$.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords GPU, functional programming, compilers

1. Introduction

Parallel programming has undergone an increase in interest in recent years, which can be likely attributed to the slowdown in sequential performance improvements over the last decade, and a corresponding increase in available hardware parallelism. Today, some of the most computationally powerful hardware available are highly parallel Graphics Processing Units (GPUs), which are being increasingly used for general non-graphics computation under the term GPGPU. Unfortunately, while GPUs provide very high potential performance, they are still difficult to program, and performance easily suffers if the programmer does not make proper use of the memory hierarchy, or accesses memory in suboptimal patterns, or provides insufficient parallelism (or too much), etc.

We believe that an automatic solution requires a clever optimising compiler capable of aggressively restructuring the code written by the programmer. The effectiveness of such a compiler can be significantly aided by compiling a language built to facilitate the necessary restructuring, such as a simple high-level data-parallel functional language. In contrast, parallelising compilers in the imperative context often find themselves fighting the language and resorting to complex low-level analysis to derive the necessary invariants, such as loop parallelism.

There has been much work on embedded high-performance data-parallel programming in functional languages [9, 10, 27], but such solutions often suffer severe limitations imposed by the host language. Therefore, we have chosen to develop a standalone, non-embedded language, as this gives us the freedom to design it without much in the way of external constraints.

Our language, Futhark, and its compiler contains a mixture of features inspired by imperative languages and their optimising compilers, such as `for`-loops, hoisting, loop distribution, in-place updates, and indirect array accesses; but in contrast to imperative languages, we use functional data-parallel control structures instead of index analysis to express parallelism. Thus, our work seeks a common ground between functional and imperative approaches.

Whilst Futhark supports the writing of efficient sequential code through the use of sequential loops and in-place updates, this is *not* to facilitate completely sequential programs. Rather, these features are intended to (i) express complicated dependent code (i.e. not simple folds) that appears *inside* parallel constructs, and (ii) to express efficient sequentialisation of excess parallelism (see example in section 6). The support for in-place updates in Futhark's type-system is formalized in Section 3.

In order to accomplish goal (ii), we also furnish Futhark with two new bulk array operators–`redomap` and `streamRed`–and extend the fusion system of [19] to incorporate these new constructs. In particular, we fuse aggressively at all nesting levels of parallelism, and support horizontal fusion, as presented in Section 4.

Since we do not assume that the code is already in flat-parallel structure, and, in particular, the aggressive multi-level fusion may invalidate this structure, the next step is to perform *kernel extraction*, as described in Section 5. The proposed technique is inspired from the loop-distribution transformation supported by some imperative compilers, and attempts to unveil as much as possible of the statically exploitable and efficient parallelism. For example, we interchange parallel bulk-operators with sequential loops when further distribution increases the parallelism degree (see sections 6 and 7.4), but we do not exploit inner parallelism hidden in branches because `scatter/gather` operations are expensive.

Our final contribution is an evaluation of the Futhark compiler on fourteen benchmark programs - nine ported from the Rodinia benchmark suite [11], four from Accelerate [22], and one from the FinPar suite of financial benchmarks [1]. Several of these benchmarks (at least `LocVolCalib`, `cfd`, `lavaMD`, and `Myocyte`) require features (indirect accesses, nested parallelism) that are to the best of our knowledge not available in restricted languages such as Accelerate. On average, Futhark is a (harmonic-mean) factor of $1.75\times$ faster than the evaluated competitors, with values ranging from being $1.52\times$ slower – on FinPar's `LocVolCalib`, large dataset, – to $20.67\times$ faster on Rodinia's `NN` benchmarks, respectively. This is without any per-benchmark tuning and with default compiler options.

```
$t$      ::=  t                       (Primitive type)
           |  [$t$,$v$]               (Arrays)

$dt$     ::=  $t$                     (Nonunique type)
           |  *$t$                    (Unique type)

$k$      ::=  x                       (Primitive value)
           |  [$v_1, \ldots, v_n$]    (Array value)

$p$      ::=  v                       (Name in binding position )

$l$      ::=  fn $\bar{t}^{(n)}$ ($p_1$, ..., $p_n$) => $e$
              (Anonymous function)

$e$      ::=  $k$  (Constant)
           |  $v$  (Variable)
           |  {$v_1$,...,$v_n$}
           |  $v_1 \odot v_2$
           |  if $v_1$ then $e_2$ else $e_3$
           |  $v$[$v_1$, ..., $v_n$]
           |  $v$($v_1$, ..., $v_n$)
           |  let {$p_1$, ..., $p_n$} = $e_1$ in $e_2$
           |  iota($v$)
           |  map($l$, $v_1$, ..., $v_n$)
           |  reduce($l$, {$v_1$, ..., $v_n$}, $v_{n+1}$, ..., $v_{2n}$)
           |  scan($l$, {$v_1$, ..., $v_n$}, $v_{n+1}$, ..., $v_{2n}$)
           |  redomap($l_1$, $l_2$,
                      {$v_1$, ..., $v_n$}
                      $v_{n+1}$, ..., $v_{n+m}$)
           |  streamRed($l_1$, $l_2$,
                        {$v_1$, ..., $v_n$}
                        $v_{n+1}$, ..., $v_{n+m}$)
           |  $v$ with [$v_1$,...,$v_n$] <- $v_v$
           |  loop ($p_1$ = $v_1$) for $p_2$ < $v_2$ do $e_3$

$fun$    ::=  fun $dt_r$ $v$($dt_1$ $p_1$,...$dt_n$ $p_n$) = $e$

$prog$   ::=  $\epsilon$
           |  $fun\ prog$
```
Figure 1: Core Futhark Syntax

1.1 Notation

Whenever z is an object of some kind, we write $\bar{z}$ to range over sequences of objects of this kind. When we want to be explicit about the size of a sequence $\bar{z} = z_0, \cdots, z_{(n-1)}$, we often write it on the form $\bar{z}^{(n)}$ and we write $z, \bar{z}$ to denote the sequence $z, z_0, \cdots, z_{(n-1)}$. We will also use this notation to shorten bits of program syntax, e.g. for the parameters of a function or the indices used to index a multi-dimensional array. We may also treat such sequences as sets for the purpose of subsetting and set inclusion.

2. Futhark

Futhark is a monomorphic, statically typed, strictly evaluated, purely functional language, intended to support parallel programs to run on massively parallel hardware, such as GPGPUs. The language, named after the first six letters of the runic alphabet (i.e., "Fuþark") enables a regular notion of nested parallelism via a set of four second-order array combinators (SOACs): `map`, `reduce`, `filter`, and `scan`. Throughout the paper we use a subset of the Futhark *core language*, which is a restricted language used internally by the Futhark compiler. The abstract syntax is shown in Figure 1. Important details include:

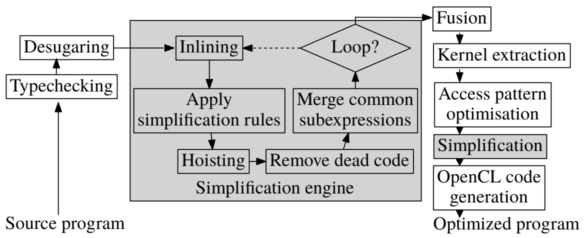


Figure 2: Compiler pipeline.

- The operands of compound expressions must be variables, which results in a representation similar to A-normal form [25].
- The keyword `in` is optional before `let` (this is solely for aesthetic reasons).
- Every written array type is parametrised with exact shape information, e.g. `[[int,m],n]` means a two-dimensional $n \times m$ array of integers, where n and m must be variables in scope.
- Tuples are not supported. As a result, SOACs such as `map` take as input *several* arrays, and similarly results in several arrays. Our `map` can be seen as implicitly `unzipping` its output.
- All arrays must be *regular*, that is, all rows of an array must have the same shape. For example array `[[4],[1.0]]` is illegal; this is mostly verified through run-time checks.
- In function declarations, the return and parameter types may have an optional *uniqueness attribute*. The purpose of these is covered in depth in Section 3.
- Functions return only a single value. This is a simplification for exposition purposes; our actual core language supports any number of return values. If an array is returned, its dimensions must be expressible in terms of the formal parameters of the function. Again, this is a simplification: our actual core language uses existential types in the style of [20] to type return values whose size cannot be known in advance.
- We support sequential `for`-loops. These are morally equivalent to a simple form of tail-recursive functions, as illustrated in Figure 3. Apart from syntactic convenience, we also take advantage of the information provided by these loops to carry out certain compiler transformations, as we shall see in Section 5. As with functions, we limit `for`-loops to just one variant variable (which must have a loop-invariant shape) for simplicity of exposition.

For conciseness, we will occasionally stray from the strict syntax in some examples, and write e.g. **map**(+1, **iota**(10)) instead of

```
let n = 10 in let a = iota(n) in let one = 1
in map(fn int (int i) => i + one, a)
```

We may also add type annotations to lambda parameters to aid readability (as above).

2.1 Overall Compiler Design

The compiler architecture is depicted in figure 2. Type checking is performed on the original program to enable meaningful error messages, but desugaring renames all bindings to unique names and brings the program to an IR that resembles A-normal form [25], that is, three-address code of statement-like `let`-bindings. Additionally, tuples are flattened, in the sense that arrays-of-tuples are transformed to tuples-of-arrays [8] and tuples are expanded such that no variable is bound to a tuple value. The result is a program in the Futhark core language shown in figure 1. On the desugared program, we apply a mixture of rewrite rules, CSE, hoisting, and dead code removal to clean up the code and remove simple inefficiencies. The next step is producer-consumer and horizontal fusion

```
loop (x = a)                -- Equivalent function
  for i < n do              fun t f(int i,
    g(i, x)                         int n,
                    ⇒               t x) =
-- Equivalent to:             if i >= n then x
f(0, n, a)                    else f(i+1, n, g(i, x))
```

Figure 3: Loop to recursive function

(section 4), followed by extraction of flat parallel kernels suitable for translation to GPU code (section 5). Notably, we perform fusion at a stage where the program still contains nested parallelism, in contrast to the approach taken when porting NESL (another language supported nested parallelism) to GPU, where fusion is performed on flattened VCODE instructions [5].

The flat-parallelism program is then passed through the simplifier for cleanup, followed by a compiler pass that rearranges the eventual in-memory representation of arrays to ensure *coalesced access* by GPU threads, which involves ensuring that neighbouring threads access neigh-boring memory addresses in the same cycle. This optimisation is outside the scope of the present paper, but is briefly discussed in section 7.

3. In-Place Updates

While Futhark is a pure functional language, it is occasionally useful to express certain algorithms in an imperative style. Consider a function for computing the n first Fibonacci numbers[1]:

```
fun [int,n] fib(int n) =
  let res = loop (arr = iota(n))
    for i < n-2 do
      let arr' =
        arr with [i+2] <- arr[i] + arr[i+1]
      in arr'
  in res
```

If the array `arr` is copied for each iteration of the loop, we are going to spend a lot of time moving data around, even though it is clear in this case that the "old" value of `arr` will never be used again. Specifically, what should be an algorithm with complexity $O(n)$ becomes $O(n^2)$, due to copying the size n array (an $O(n)$ operation) for each of the $n-2$ iterations of the loop.

To prevent this, we will update the array *in-place*, that is, with a static guarantee that the operation will not require any additional memory allocation, such as copying the array. With an in-place modification, we modify the array in time proportional to the slice being updated ($O(1)$ in the case of the Fibonacci function), rather than time proportional to the size of the final array, as would the case if we perform a copy. In order to perform the update without violating referential transparency, we need to know that no other references to the array exists, or at least that such references will not be used on any execution path following the in-place update.

In Futhark, this is done through a type system feature called *uniqueness types*, similar to, although simpler, than the uniqueness types of Clean [3, 4]. Alongside a (relatively) simple aliasing analysis in the type checker, this is sufficient to determine at compile time whether an in-place modification is safe, and signal an error if the in-place update safety cannot be guaranteed. The simplest way to introduce uniqueness types is through an example. To that end, let us consider the following function definition.

```
fun *[int,n] modify(int n,
                    *[int,n] a, int i,
                    [int,n] x) =
  let b = a with [i] <- a[i] + x[i]
  in b
```

[1] This example is used only for demonstration purposes - there are better ways to compute Fibonacci numbers in Futhark.

The function call `modify`(a,i,x) returns a, but where the element at index `i` has been increased by x. Note the *asterisks*: in the parameter declaration `*[int] a`, this means that the function `modify` has been given "ownership" of the array a, meaning that any caller of `modify` will never reference array a after the call again. In particular, `modify` can change the element at index `i` without first copying the array, i.e. `modify` is free to do an in-place modification. Furthermore, the return value of `modify` is also unique - this means that the result of the call to `modify` does not share elements with any other visible variables, i.e., it might share elements with `a` but not with `x`.

Let us consider a call to `modify`, which might look as follows.

```
let b = modify(a, i, x)
```

In which circumstances is this call valid? Two things must hold:

1. The type of a must be `[int]`.
2. Neither a or any variable that *aliases* a may be used on any execution path following the call to `modify`.

When a value is passed for a unique parameter in a function call, we consider that value to be *consumed*, and neither it nor any of its aliases can be used again. Otherwise, we would break the contract that allows the function to manipulate the argument freely. In general, we say that an array is *consumed* when it must never again be accessed on any following possible execution. For example, in the expression 'a **with** [i] <- x', the array `a` is being consumed. From an implementation perspective, this contract allows type checking to rely on simpler, intra-procedural (alias) analysis, both in the callee and in the caller, as detailed in the next section.

3.1 Alias Analysis

We perform alias analysis on a program that we assume to be otherwise type-correct. Our presentation uses an inference rule-based approach similar to the one usually used for type systems. The central judgment is as follows:

$$\boxed{\Sigma \vdash e \sqsubset \langle \sigma_1, \ldots, \sigma_n \rangle}$$

This judgment asserts that, within the context Σ, the expression e produces n values, where value number i has the *alias set* σ_i. An alias set is a subset of the variable names in scope, and indicates which variables an array value (or variable) may share elements with. Alias sets are also computed for non-array variables, but they are meaningless and not used for anything. The context Σ is a mapping from variables in scope to their aliasing sets.

The aliasing rules are listed in figure 4, although for space reasons, some are left out. The ALIAS-VAR-rule defines the aliases of a variable expression to be the alias set of the variable joined by the name of the variable itself - this is because $v \notin \Sigma(v)$, as can be seen by ALIAS-LETPAT. Notably, the alias sets for the values produced by SOACs such as `map` are empty. Operationally, we can imagine the arrays produced as *fresh*, although the compiler is of course free to re-use existing memory if it can do so safely. The ALIAS-INDEXARRAY rule tells us that a scalar read from an array does not alias its origin array, but ALIAS-SLICEARRAY dictates that an array slice does. This fits with our intuition about how such operations might be implemented.

The most interesting aliasing rules are the ones for function calls. Since we would like our alias analysis to be intraprocedural, we are forced to be conservative. There are two rules: one for functions returning unique arrays, and one for functions returning nonunique arrays. In the former case, since the returned array is unique, the alias set is empty. In the latter case, the returned array

$$\boxed{\Sigma \vdash e \sqsubset \langle \sigma_1, \ldots, \sigma_n \rangle} :$$

$$\frac{}{\Sigma \vdash v \sqsubset \langle \{v\} \cup \Sigma(v) \rangle} \quad \text{(ALIAS-VAR)}$$

$$\frac{}{\Sigma \vdash k \sqsubset \langle \emptyset \rangle} \quad \text{(ALIAS-CONST)}$$

$$\frac{}{\Sigma \vdash \texttt{map(}l\texttt{, } \bar{v}^{(n)}\texttt{)} \sqsubset \langle \bar{\emptyset}^{(n)} \rangle} \quad \text{(ALIAS-MAP)}$$

$$\frac{\Sigma \vdash e_2 \sqsubset \langle s_1^2, \ldots, s_n^2 \rangle \qquad \Sigma \vdash e_3 \sqsubset \langle s_1^3, \ldots, s_n^3 \rangle}{\Sigma \vdash \texttt{if } v_1 \texttt{ then } e_2 \texttt{ else } e_3 \sqsubset \langle s_1^2 \cup s_1^3, \ldots, s_n^2 \cup s_n^3 \rangle} \quad \text{(ALIAS-IF)}$$

$$\frac{\begin{array}{c}\Sigma \vdash e_1 \sqsubset \langle \bar{\sigma}^{(n)} \rangle \\ \Sigma, p_i \mapsto \sigma_i \vdash e_2 \sqsubset \langle \bar{\sigma'}^{(n)} \rangle\end{array}}{\Sigma \vdash \texttt{let \{}\bar{p}^{(n)}\texttt{\} = } e_1 \texttt{ in } e_2 \sqsubset \langle \bar{\sigma'}^{(n)} \rangle \setminus \{\bar{p}^{(n)}\}} \quad \text{(ALIAS-LETPAT)}$$

$$\frac{v \text{ is of rank } n}{\Sigma \vdash v\texttt{[}\bar{v}^{(n)}\texttt{]} \sqsubset \langle \emptyset \rangle} \quad \text{(ALIAS-INDEXARRAY)}$$

$$\frac{v \text{ is of rank} < n}{\Sigma \vdash v\texttt{[}\bar{v}^{(n)}\texttt{]} \sqsubset \langle \Sigma(v) \rangle} \quad \text{(ALIAS-SLICEARRAY)}$$

$$\frac{\begin{array}{c}\Sigma \vdash v_1 \sqsubset \langle \sigma \rangle \\ \Sigma, v_1 \mapsto \sigma \vdash e_3 \sqsubset \langle \sigma' \rangle\end{array}}{\Sigma \vdash \begin{array}{c}\texttt{loop (}p_1\texttt{ = }v_1\texttt{)} \\ \texttt{for } p_2 \texttt{ < } v_2 \texttt{ do } e_3\end{array} \sqsubset \langle \sigma' \setminus \{p_1\rangle\}} \quad \text{(ALIAS-DOLOOP)}$$

$$\frac{}{\Sigma \vdash v_a \texttt{ with [}\bar{v}^{(n)}\texttt{] <- } v_v \sqsubset \langle \Sigma(v_a) \rangle} \quad \text{(ALIAS-UPDATE)}$$

$$\frac{\begin{array}{c}\text{lookup}_{\text{fun}}(v_f) = \langle t_r, dt_1, \ldots, dt_n \rangle \\ \Sigma \vdash v_i \sqsubset \langle \sigma_i \rangle \\ \sigma = \bigcup_{dt_i \text{ is not of form } *t} \sigma_i\end{array}}{\Sigma \vdash v_f\texttt{(}v_1\texttt{, ..., }v_n\texttt{)} \sqsubset \langle \sigma \rangle} \quad \text{(ALIAS-APPLY-NONUNIQUE)}$$

$$\frac{\text{lookup}_{\text{fun}}(v_f) = \langle *t_r, dt_1, \ldots, dt_n \rangle}{\Sigma \vdash v_f\texttt{(}v_1\texttt{, ..., }v_n\texttt{)} \sqsubset \langle \emptyset \rangle} \quad \text{(ALIAS-APPLY-UNIQUE)}$$

$$\boxed{\langle \mathcal{C}_1, \mathcal{O}_1 \rangle \gg \langle \mathcal{C}_2, \mathcal{O}_2 \rangle : \langle \mathcal{C}_3, \mathcal{O}_3 \rangle} :$$

$$\frac{(\mathcal{O}_2 \cup \mathcal{C}_2) \cap \mathcal{C}_1 = \emptyset}{\langle \mathcal{C}_1, \mathcal{O}_1 \rangle \gg \langle \mathcal{C}_2, \mathcal{O}_2 \rangle : \langle \mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{O}_1 \cup \mathcal{O}_2 \rangle} \quad \text{(OCCURENCE-SEQ)}$$

Figure 4: Aliasing rules

is conservatively to assumed to alias all parameters, except those that were passed to satisfy a unique parameter.

In our actual implementation, type checking, alias computation, and in-place update checking is all performed at the same time, but we have split it up for expository purposes. In order to simplify the presentation, the following section will use aliases(v) to refer to the alias set of the variable v.

3.2 In-Place Update Checking

Now we can talk about whether an expression is functionally safe with respect to in-place updates. The central judgment is:

$$\boxed{e \ \triangleright \langle \mathcal{C}, \mathcal{O} \rangle}$$

Here $\mathcal{O}$ is a set of the variables *observed* (used) in e, and $\mathcal{C}$ is the set of variables *consumed* through function calls and in-place updates. Together, the pair $\langle \mathcal{C}, \mathcal{O} \rangle$ is called an *occurrence trace*.

We also define a judgment for *sequencing* two occurrence traces on figure 4. The sequencing judgment

$$\boxed{\langle \mathcal{C}_1, \mathcal{O}_1 \rangle \gg \langle \mathcal{C}_2, \mathcal{O}_2 \rangle : \langle \mathcal{C}_3, \mathcal{O}_3 \rangle}$$

can be derived if and only if it is acceptable for $\langle \mathcal{C}_1, \mathcal{O}_1 \rangle$ to happen first, then $\langle \mathcal{C}_2, \mathcal{O}_2 \rangle$, giving the combined occurrence trace $\langle \mathcal{C}_3, \mathcal{O}_3 \rangle$. The formulation as a judgment is to underline the fact that sequentialisation is sometimes not derivable - this corresponds to the case where an array is used after it has been consumed. The judgment is defined by a single inference rule, which states that two occurrence traces can be sequentialised if and only if no array consumed in the left-hand trace is used in the right-hand trace.

$$\boxed{e \ \triangleright \langle \mathcal{C}, \mathcal{O} \rangle}$$

$$\frac{}{v \ \triangleright \langle \emptyset, \text{aliases}(v) \rangle} \quad \text{(SAFE-VAR)}$$

$$\frac{}{k \ \triangleright \langle \emptyset, \emptyset \rangle} \quad \text{(SAFE-CONST)}$$

$$\frac{\begin{array}{c}e_1 \ \triangleright \langle \mathcal{C}_1, \mathcal{O}_1 \rangle \qquad e_2 \ \triangleright \langle \mathcal{C}_2, \mathcal{O}_2 \rangle \\ \langle \mathcal{C}_1, \mathcal{O}_1 \rangle \gg \langle \mathcal{C}_2, \mathcal{O}_2 \rangle : \langle \mathcal{C}_3, \mathcal{O}_3 \rangle\end{array}}{\texttt{let \{}v_1\texttt{, ..., }v_n\texttt{\} = } e_1 \texttt{ in } e_2 \ \triangleright \langle \mathcal{C}_3, \mathcal{O}_3 \rangle\rangle} \quad \text{(SAFE-LETPAT)}$$

$$\frac{\begin{array}{c}v_1 \ \triangleright \langle \mathcal{C}_1, \mathcal{O}_1 \rangle \qquad e_2 \ \triangleright \langle \mathcal{C}_2, \mathcal{O}_2 \rangle \qquad e_3 \ \triangleright \langle \mathcal{C}_3, \mathcal{O}_3 \rangle \\ \langle \mathcal{C}_1, \mathcal{O}_1 \rangle \gg \langle \mathcal{C}_2, \mathcal{O}_2 \rangle : \langle \mathcal{C}_2', \mathcal{O}_2' \rangle \\ \langle \mathcal{C}_1, \mathcal{O}_1 \rangle \gg \langle \mathcal{C}_3, \mathcal{O}_3 \rangle : \langle \mathcal{C}_3', \mathcal{O}_3' \rangle\end{array}}{\texttt{if } v_1 \texttt{ then } e_2 \texttt{ else } e_3 \ \triangleright \langle \mathcal{C}_2' \cup \mathcal{C}_3', \mathcal{O}_2' \cup \mathcal{O}_3' \rangle} \quad \text{(SAFE-IF)}$$

$$\frac{}{v_a \texttt{ with [}nseqvn\texttt{] <- } v_v \ \triangleright \langle \text{aliases}(v_a), \text{aliases}(v_n) \rangle} \quad \text{(SAFE-UPDATE)}$$

$$\frac{\begin{array}{c}e_b \ \triangleright \langle \mathcal{C}, \mathcal{O} \rangle \\ p_i \mapsto \text{aliases}(v_i) \vdash \langle \mathcal{C}, \mathcal{O} \rangle \triangle \langle \mathcal{C}', \mathcal{O}' \rangle\end{array}}{\texttt{map(fn } \bar{t}^{(m)} \texttt{ (}\bar{p}^{(n)}\texttt{) => } e_b\texttt{, } \bar{v}^{(n)}\texttt{)} \ \triangleright \langle \mathcal{C}', \mathcal{O}' \rangle} \quad \text{(SAFE-MAP)}$$

Figure 5: Uniqueness checking

The inference rules for safety checking are given in figure 5. One very important rule is of course the one for in-place update v_a `with` `[`$\bar{v}^{(n)}$`]` `<-` v_v, which gives rise to an occurrence trace indicating that we have *observed* v_v and *consumed* v_a. We ignore the indices $\bar{v}^{(n)}$ as these are necessarily scalar variables, and thus cannot be consumed.

Another interesting case is checking the safety of a `map` expression. Intuitively, this means we do not wish to permit the lambda of a `map` to consume any array bound outside of it, as that would imply the array is consumed once for every iteration of the `map`. It is, however, acceptable for the lambda to perform in-place updates on its parameters, which should be seen as the `map` expression as a whole consuming the corresponding input array. An example can be seen on figure 7 To express this, we define an auxiliary judgment:

$$\boxed{\mathcal{P} \vdash \langle \mathcal{C}_1, \mathcal{O}_1 \rangle \triangle \langle \mathcal{C}_2, \mathcal{O}_2 \rangle}$$

Here, $\mathcal{P}$ is a mapping from parameter names to alias sets. Any variable v in $\mathcal{O}_1$ that has a mapping in $\mathcal{P}$ is replaced with $\mathcal{P}[v]$ to produce $\mathcal{O}_2$. If no such mapping exists, v is simply included in $\mathcal{O}_2$. Similarly, any variable v in $\mathcal{C}_1$ that has a mapping in $\mathcal{P}$ is replaced with $\mathcal{P}[v]$ to produce $\mathcal{C}_2$. However, if v does not have such a mapping, the judgment is not derivable. This corresponds to the body of the lambda attempting to consume something that is not a formal parameter. The precise inference rules are shown in Figure 6. Do-loops and function declarations can be checked for safety in a similar way: a function is safe with respect to in-place updates if its body consumes only those of the functions parameters that are unique (i.e. have an * annotation).

4. New SOACs and Their Fusion Rules

Futhark's fusion is inspired from Henriksen and Oancea's work [19], who propose an aggressive technique for fusing producer-consumer SOACs at all nesting levels, without duplicating computation, even when the produced array is used in several places. Their fusion engine is centered around the `redomap` SOAC, which has the type: `redomap`:: $(\alpha \to \alpha \to \alpha) \to (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \ \to \ \alpha$ and is generated by a `map-reduce` composition rewritten as:

$$\boxed{\mathcal{P} \vdash \langle \mathcal{C}_1, \mathcal{O}_1 \rangle \triangle \langle \mathcal{C}_2, \mathcal{O}_2 \rangle}$$

$$\frac{\mathcal{P}[v] = \sigma \quad \mathcal{O}_v = \{v_i \mid v_i \in \sigma\} \quad \mathcal{P} \vdash \langle \emptyset, \mathcal{O} \rangle \triangle \langle \emptyset, \mathcal{O}' \rangle}{\mathcal{P} \vdash \langle \emptyset, \{v\} \cup \mathcal{O} \rangle \triangle \langle \emptyset, \mathcal{O}_v \cup \mathcal{O}' \rangle} \quad (\text{OBSERVE-PARAM})$$

$$\frac{v \notin \mathcal{P} \quad \mathcal{P} \vdash \langle \emptyset, \mathcal{O} \rangle \triangle \langle \emptyset, \mathcal{O}' \rangle}{\mathcal{P} \vdash \langle \emptyset, \{v\} \cup \mathcal{O} \rangle \triangle \langle \emptyset, \{v\} \cup \mathcal{O}' \rangle} \quad (\text{OBSERVE-NONPARAM})$$

$$\frac{\mathcal{P}[v] = \sigma \quad \mathcal{C}_v = \{v_i \mid v_i \in \sigma\} \quad \mathcal{P} \vdash \langle \mathcal{C}, \mathcal{O} \rangle \triangle \langle \mathcal{C}', \mathcal{O}' \rangle}{\mathcal{P} \vdash \langle \{v\} \cup \mathcal{C}, \mathcal{O} \rangle \triangle \langle \mathcal{C}_v \cup \mathcal{C}', \mathcal{O}' \rangle} \quad (\text{CONSUME-PARAM})$$

Figure 6: Parameter consumption

```
-- This one is OK and considered
-- to consume 'as'.
let bs = map(fn [int,m] (a) =>
                  let b = a with [i] = 2
                  in b,
                as)
let d = iota(m)
-- This one is not, because we consume
-- something that is not a formal parameter.
let cs = map(fn [int,m] (i) =>
                  let c = d with [i] = 2
                  in c,
               iota(n))
```

Figure 7: Examples of `map`s doing in-place updates

`red` $\odot$ `e . map f` $\equiv$ `red` $\odot$ `e . map (red` $\odot$ `e . map f) .split`$_p$ $\equiv$ `red` $\odot$ `e . map (foldl g e) . split`$_p$ $\equiv$ `redomap` $\odot$ `g e`, i.e., the input array is split into number-of-processors chunks, each processor efficiently sequentializes the computation on its chunk via `foldl`, and the partial (one-per-processor) results are reduced. In essence `redomap` is a symbolic notation that permits efficient sequentialization of the excess parallelism in later compiler stages, without losing any available parallelism on the way: `redomap` $\odot$ `g e` $\equiv$ `red` $\odot$ `e . map (g e)`.

This section presents an extension of the set of SOACs and the fusion approach of [19] that solves three significant limitations:

1 With the current semantics `redomap` cannot produce both an array and its reduction, because its result type does not allow it.

2 "Horizontal" fusion is not supported, e.g., two `redomap`s cannot be fused and neither can `redomap` $\circ$ `map` even when the result of the `map` is used after the `redomap`. This is significant because such fusion may enable producer-consumer fusion.

3 `scan` does not fuses with anything and in particular poor-man streaming by fusion is not supported.

We present the new SOACs and their rationale in Section 4.1, and demonstrate the fusion engine on a telling example in Section 4.2.

4.1 Types, Semantics and Rationale of New SOACs

Figure 8 presents the types and semantics of the new SOACs.
First, `redomap` may return an arbitrary number r of arrays, necessarily of the same outermost-dimension size as the input arrays, in addition to the reduced result (of type $\bar{\alpha}^{(p)}$). Its semantics is equivalent to mapping the input arrays by its second (function) parameter, and returning the last r arrays of the result, together with the result of reducing the first p arrays with the first-parameter operator. With this extension, it is possible to fuse `redomap` with other `map`s, `reduce`s and `redomap`s which appear before or after the current `redomap`. The result is a `redomap`.

Notation: n, m, p, q, r `integers,` a, b, c `arrays,` f `function,` α, β, γ `types,` $\oplus$ `binary associative operator,` $\bar{\#}$ `vectorized array concatenation,` $(v_1, \ldots, v_n)$ `tuple (type),` $\bar{v}^{(p)}$ `expands to` $v_1, \ldots, v_p$.

I. ***New SOACs Types and Semantics***

$$\textbf{redomap} :: (\ (\bar{\alpha}^{(p)}, \bar{\alpha}^{(p)}) \rightarrow \bar{\alpha}^{(p)},\ (\bar{\alpha}^{(p)}, \bar{\beta}^{(q)}) \rightarrow (\bar{\alpha}^{(p)}, \bar{\gamma}^{(r)})$$
$$,\ \bar{\alpha}^{(p)},\ [\beta_1, n], \ldots, [\beta_q, n]\ )\ \rightarrow\ (\bar{\alpha}^{(p)},\ [\gamma_1, n], \ldots, [\gamma_r, n])$$
$$\textbf{redomap}(\oplus, f, \bar{e}^{(p)}, \bar{b}^{(q)})\ \equiv\ \texttt{let}\ (\bar{a}^{(p)},\ \bar{c}^{(r)})\ =\ \texttt{map}(f,\ \bar{b}^{(q)})$$
$$\texttt{in}\ (\texttt{reduce}(\oplus, \bar{e}^{(p)}, \bar{a}^{(p)}),\ \bar{c}^{(q)})$$
$$\textbf{streamRed} :: (\ (\bar{\alpha}^{(p)}, \bar{\alpha}^{(p)}) \rightarrow \bar{\alpha}^{(p)}$$
$$,\ (\bar{\alpha}^{(p)}, [\beta_1, m], \ldots, [\beta_q, m]) \rightarrow (\bar{\alpha}^{(p)}, [\gamma_1, m], \ldots, [\gamma_r, m])$$
$$,\ \bar{\alpha}^{(p)},\ [\beta_1, n], \ldots, [\beta_q, n]\ )\ \rightarrow\ (\bar{\alpha}^{(p)},\ [\gamma_1, n], \ldots, [\gamma_r, n])$$
$$\textbf{streamRed}(\oplus, f, \bar{e}^{(p)}, \bar{b}^{(q)})\ \equiv\ (\bar{e}^{(p)}\ \oplus\ \bar{a_1}^{(p)}\ \oplus \ldots \oplus\ \bar{a_s}^{(p)},\ \bar{c}^{(r)})$$
$$\texttt{where}\ (\bar{a_i}^{(p)}, \bar{c_i}^{(r)}) = f(\bar{e}^{(p)}, \bar{b_i}^{(q)})\ \texttt{and}$$
$$\bar{c}^{(r)} = \bar{c_1}^{(r)} \bar{\#} \ldots \bar{\#} \bar{c_s}^{(r)},\ \forall \bar{b}^{(q)} \equiv \bar{b_1}^{(q)} \bar{\#} \ldots \bar{\#} \bar{b_s}^{(q)}$$
$$\textbf{streamSeq} :: (\ (\bar{\alpha}^{(p)}, [\beta_1, m], \ldots, [\beta_q, m]) \rightarrow (\bar{\alpha}^{(p)},\ [\gamma_1, m], \ldots, [\gamma_r, m])$$
$$,\ \bar{\alpha}^{(p)},\ [\beta_1, n], \ldots, [\beta_q, n]\ )\ \rightarrow\ (\bar{\alpha}^{(p)},\ [\gamma_1, n], \ldots, [\gamma_r, n])$$
$$\textbf{streamSeq}(f,\ \bar{a_0}^{(p)},\ \bar{b}^{(q)})\ \equiv\ (\bar{a_s}^{(p)},\ \bar{c_1}^{(r)} \bar{\#} \ldots \bar{\#} \bar{c_s}^{(r)}),\ \texttt{where}$$
$$(\bar{e_i}^{(p)}, \bar{c_i}^{(r)}) = f(\bar{e_{i-1}}^{(p)}, \bar{b_i}^{(r)}),\ \forall \bar{b}^{(q)} \equiv \bar{b_1}^{(q)} \bar{\#} \ldots \bar{\#} \bar{b_s}^{(q)}$$

Figure 8: Types & Semantics of `redomap` and `streamRed/Seq`.

Second, `streamRed` is designed in much of the same way as `redomap`, except that it applies its second-parameter function f to each element of an arbitrary partitioning of the input arrays $\bar{b}^{(q)}$ (as long as all $\bar{b}^{(q)}$ are partitioned in the same way), and then it reduces and concatenates across chunks the first p and last r results, respectively. While the semantics is that chunks can be processed in parallel, it is left to the user to ensure the strong invariant that any partitioning of the input arrays leads to the same result[2].

Futhark also supports a `streamMap` SOAC (not shown), which is a simplified `streamRed` in the same way that `map` is a simplified `redomap`, i.e., it lacks the reduce component. The main reason for supporting `streamMap/Red` in Futhark is allowing the user to express both all available parallelism (chunk size 1) and an efficient sequentialization alternative. The latter has many guises: it may correspond to writing a loop with in-place updates, such as in the `k-means` example presented in Section 6 or to specifying an algorithmic invariant, such as strength reduction. For example, Sobol pseudo-random numbers can be computed by a slower but `map`-parallel formula, or by a cheaper (recurrence) formula, but which requires `scan` parallelism [23]. This can be expressed elegantly with `streamMap`: the computation of each chunk starts with applying the independent formula, while the remaining elements are processed with the cheaper recurrence formula. With respect to fusion rules, `map`, `reduce`, `redomap`, `streamMap` and `streamRed` fuse together resulting into either `streamMap` or `streamRed`, depending on whether a reduction is needed.

Finally, `streamSeq` requires sequential execution of chunks, because the result $\bar{e_i}^{(p)}$ of processing chunk $\bar{b_i}^{(q)}$ becomes the (accumulator) input for processing the next chunk `i+1`. With respect to fusion rules, anything fused with a `scan` or `streamSeq` results in a `streamSeq`. For example, assuming `f::int`$\rightarrow$`int` and `A` and `B` integer arrays of size `n`, then `map(f,scan(+,0,A))` is fused into

```
streamSeq( fn {int, [int,r]} (int acc, [int,r] a) =>
             let b = scan(+, 0, a)
```

[2] This is compatible with Futhark supporting arbitrary reduce operators, and delegating to the user the responsibility of ensuring their associativity.

```
        let c = map (+acc, b)
        let d = map(f, c)      in {c[r-1], d}
    , 0, A)
```

Note that streamSeq offers poor-man streaming by fusion, and also, allows to recover all parallelism if needed by maximizing the chunk size. It is important to remark however that fusion between streamSeq or scan and streamMap/Red is disallowed, because it would result in a streamSeq, and thus may potentially destroy the parallelism expressed by the user via streamMap/Red. (This is because the body of streamMap/Red may correspond to the efficient sequentialization alternative and may offer no parallelism.)

4.2 Futhark Fusion By Example

The top part of Figure 9 shows a contrived Futhark program that, at its outer level, is a composition of four SOACs: streamMap computes [0,q,2q,...,(n-1)q] by performing the multiplication for the first element of the chunk, while the rest are computed by the recurrence formula x_{i+1}=x_i+q with a scan (under the misguided assumption that the parallel computation will benefit from it). The remaining SOACs multiplies by 2 the array produced by streamMap, sums it up, and also increases by one each element of input array A.

Fusion proceeds bottom-up and maintains a set of SOACs that are possible candidates for fusion. Whenever a SOAC is encountered, the candidate set is examined: preference is given to SOACs that are in a producer-consumer relation and if no such case then horizontal fusion is tried. In our case, when fusion reaches the map producing Z, there is exactly one candidate, i.e., the reduce producing r, and horizontal fusion succeeds and generates the SOAC numbered (7) in Figure 9. We remark that this fusion would not be possible under the technique of Henriksen and Oancea [19] because: (i) of the restricted result type of redomap, (ii) horizontal fusion is not supported and (iii) because array Z is used outside a SOAC, i.e., it is returned by the main function.

Our implementation alleviates these restrictions: horizontal fusion may proceed as long as (i) the two SOACs are located in the same block of let statements (named body), (ii) the outermost sizes of the input arrays match, and (iii) there is no (unfused) use of the result array in between the two SOACs[3]. The latter can be further relaxed by using a dependency-graph representation of body.

Continuing with our example, *Step II* fuses the map that produces Z with the redomap SOAC obtained in the previous step, which consumes Z. This results in a new redomap kernel (8).

Step III is to fuse the streamMap that produces array S with its redomap consumer, obtained in the previous step. The result of fusion is the streamRed SOAC numbered (9) in Figure 9, which borrows the redomap's associative operator and neutral element, and "inlines" the redomap in the body of streamMap's second-function argument. The final step is to fuse the redomap ∘ scan ∘ map composition inside the streamRed's function argument. This generates a streamSeq (inner) SOAC that has two (sequential) accumulators: acc0 and acc1 for the scan and redomap, respectively.

In conclusion, the streamRed-streamSeq nest provides the means (i) to preserve/recover all available parallelism, e.g., by setting streamRed's chunk size to 1, and (ii) to provide efficient sequentialization of the parallelism in excess. For example, setting the chunk size of streamSeq to 1 minimizes the memory footprint of the parallel computation. Furthermore, if we disregard arrays A and Z, the memory footprint of one iteration of streamRed would be $O(1)$, regardless of its chunk size.

5. Kernel Extraction through Loop Distribution

Futhark supports nested regular parallelism, but our target platform, GPUs, requires that we extract flat parallel *kernels*. This is typically

[3] We have similarly relaxed this restriction for consumer-producer fusion.

```
-- Original Program:
fun {int,[int]} main([int,n] A, int q) =
  let S = streamMap(                            -- (4)
          fn [int,c] ([int,c] B) =>
              let M = map (fn int (int k) => -- (6)
                             if k!=0 then q
                             else B[0]*q
                         , iota(c) )
              in  scan(+, 0, M)                 -- (5)
        , iota(n) )
  let Y = map   (*2,    S)                      -- (3)
  let Z = map   (+1,    A)                      -- (2)
  let r = reduce(+, 0, Y)                       -- (1)
  {r,Z}

-- I. horizontal fusion (1) and (2) results in (7):
let {r, Z} = redomap(                         -- (7)
    fn int (int x, int y) => x + y
  , fn {int, int} (int x, int a, int y) => {x+y, a+1}
  , {0}, A, Y )

-- II. producer-consumer fusion (7) and (3) results in (8):
let {r, Z} = redomap(                           -- (8)
    fn int (int x, int y) => x + y
  , fn {int,int} (int x, int s, int a) => {x+s*2, a+1}
  , 0, S, A )

-- III. producer-consumer fusion (8) and (4) results in (9):
let {r,Z} = streamRed(                          -- (9)
    fn int (int x, int y) => x + y
  , fn {int, [int]} (int ne, [int,c1] B, [int,c1] A0) =>
        let M = map ( fn int (int k) =>         -- (6)
                        if k!=0 then q else B[0]*q
                    , iota(c1) )
        let SC= scan(+, 0, M)                   -- (5)
        let {r, Z} = redomap(                   -- (10)
            fn int (int x, int y) => x + y
          , fn {int,int} (int x,int s,int a) =>
                {x+s*2, a+1}
          , 0, SC, A )
        in  {ne+r, Z}
  , 0, iota(n), A )

-- IV. Producer-consumer fusion between (5), (6) and (10)
--      results in streamSeq (due to the scan in (5)).
fun {int,[int]} main([int,n] A, int q) =
  streamRed(
    fn int (int x, int y) => x + y
  , fn {int, [int,c1]}
        (int ne, [int,c1] B, [int,c1] A0) =>
          let {_, acc, Z} = streamSeq(
            fn {int, int, [int,c2]}
               (int acc0,int acc1,[int,c2] ks,[int,c2] a)=>
                  let M = map ( fn int (int k) =>
                                  if k!=0 then q
                                  else B[0]*q
                              , ks )
                  let SC= scan(+, 0, M)
                  let nxt_acc0 = acc0 + SC[c2-1] in
                  let {acc11, z} = redomap(
                        fn int (int x, int y) => x + y
                      , fn {int, int}
                           (int x, int y, int a_el) =>
                              {x + (acc0+y)*2, a_el+1}
                      , 0, M, a )
                  in  {nxt_acc0, acc11+acc1, z}
            , 0, 0, iota(c1), A0 )
          in {ne + acc, Z}
  , 0, iota(n), A )
```

Figure 9: Futhark's Fusion Demonstrated on a Contrived Example: *Step I* horizontally fuses the last map and reduce into a redomap. *Step II* fuses the map producing Y into the redomap of *Step I*. *Step III* fuses streamMap into its consumming redomap obtained in *Step II*, which results in a (outer) streamRed. *Step IV* fuses the inner map ∘ scan ∘ redomap into streamSeq. The outer streamRed provides parallelism, while allowing efficient sequentialization: regardless of streamRed's chunk size, setting streamSeq's chunk size to 1 effectively minimizes the memory footprint of the computation.

```
let as = map(fn int (bs) =>
     let c = reduce(+, 0, bs)
     let a = c + 2
     in a,
  bss)
```

(a) Before distribution

```
let cs = map(fn int (bs) =>
     let c = reduce(+, 0, bs) in c,
  bss)
let as = map(fn int (c) =>
     let a = c + 2 in a,
  cs)
```

(b) After the binding of a has been distributed

Figure 10: An example of loop distribution

done through the use of the *flattening transformation* [6], which attempts to maximise the amount of parallelism exposed. However, existing implementations of flattening suffer from two major drawbacks: (i) The flattened code tends to exhibit high space usage; and (ii) the communication and access patterns of the original code are obscured, which makes following optimisations harder to perform.

Additionally, flattening will often result in code using irregular arrays, which is more difficult to compile and optimise than code using just regular arrays. Hence, we have developed an novel algorithm for extracting flat data-parallel loop kernels from a nesting of various parallel loops (mainly `map`). One important property of our approach is that it does not change the representation of arrays, which helps preserve the space usage asymptotics of the original program. For simplicity, this section will cover only parallel loop nests that have a `map` as their outermost level of parallelism.

Our algorithm is based on the idea of *loop distribution*, where component expressions of a loop are extracted as separate loops. A simple example can be seen in figure 10 – here, a `map` expression with nested parallelism (the `reduce`) is turned into two flat parallel kernels through the distribution of a scalar computation. The first kernel is parallel by virtue of the fact that a `map` containing just a `reduce` is compilable as a segmented reduction over the *entire* two-dimensional array given as input to the `map`. Suppose that the input array `bss` is of size $n \times m$. The original loop would provide work for n parallel threads, but after distribution we obtain one nest with up to $n \times m$ parallel threads, and another with n. If m is much greater than n, this will have a dramatic impact on total runtime on manycore hardware.

The algorithm is given a function in a normalised form where the result of every expression is bound to a variable. We when traverse the function body looking for outermost `map` expressions, and when one is found, we apply loop distribution in order to turn the expression (which may contain nested parallelism) into a sequence of flat-parallelism kernels.

In this section, we will transform nests of `map`s into different nests of `map`s. This is for simplicity of exposition only, in order to reduce the number of distinct syntactic constructs we need to describe. For example, in our actual compiler, we do not create a `map` containing a `reduce` as in figure 10, but rather use a distinct AST construct for representing segmented reductions.

5.1 Loop Distribution by Example

We will show the workings of the loop distribution algorithm by demonstrating its application on a complicated `map` nest containing three levels of parallelism, some of which we can exploit directly, and others which are not exploitable, as distribution would induce irregular arrays.

The core idea is a backwards traversal of the statements within the nest. Whenever we reach a statement `let` p `=` e, we can distribute that statement *and every following statement* as a new kernel. This is what is done in figure 10, where the statement `let a = c + 2` is distributed as its own kernel. The *residual* loop nest is then modified to return the values needed by the statements that we have distributed, and traversal continues.

Apart from distributing, we can also perform other operations. Consider the original program in in figure 11a - it consists of a `map` nest containing a sequential `loop`, which again contains another `map`. That is, we have two parallel loops separated by a sequential loop, which prevents us from fully exploiting the parallelism - parallel dimensions must be adjacent. Fortunately, it is well known that it is always valid to interchange a parallel loop *inwards* (more details given in section 5.2). Thus, if we distribute the `loop` by itself, we obtain a parallel `map` containing just the sequential loop, after which we can swap the outer `map` and the middle `loop`, and obtain the nesting shown in on figure 11b. Kernel extraction is then applied to the body of the new top-level sequential loop. Here we find a `map`, and begin traversing its statements in a backwards fashion, as we did for the original map nest. Neither of the statements for `b'` or `e` motivate us to distribute, but when we reach the reduction, we wish to distribute it by itself. Distributing a statement by itself is carried out simply by distributing everything following the statement as its own kernel, then the statement itself. The result is seen on figure 11d.

This finishes off the body of the sequential loop, and we continue with the `asss` statement. Here we find a `map` expression. There is no further parallelism within its function, but we do want to distribute the `map` by itself in order to exploit all levels of parallelism, the result of which can be seen on figure 11f, leaving a residual of two statements: a `scan` and a `reduce`.

We first come upon the reduction, which we would normally wish to distribute by itself. However, the reduction is on the array `cs`, whose size is `p`, which is variant to the loop nest. Distributing the reduction would give rise to an irregular array, which is not supported in our framework[4]. Hence, we move on to the preceding `scan`, which we also cannot distribute by itself. The result becomes a kernel with two levels of `map` containing two (sequential) loops.

5.2 Choices when Distributing

The core of our kernel extraction algorithm is thus walking backwards through the statements of a `map` nest, and at every statement, possibly performing an action. The possible decisions are based both on the statement we are inspecting, as well as the context (such as whether we can distribute without using irregular arrays). Most of the cases presently considered in the Futhark compiler, based on the statement `let` p `=` e being inspected, is as follows.

Case `let` p `=` e**:**
We always have the fallback case of sequentialising e if nothing else works. In the worst case, we will end up sequentialising every statement in the `map` and only exploit the parallelism of the outermost nesting.

Case `let` p `= map(fn` $\bar{t}^{(m)}$ `(...) =>` e`, ...):`
The simplest case of nested parallelism: we recursively descend into the statements of e, and if any are left un-distributed, we distribute the `map` containing the residual statements (if possible).

Case `let` p `= loop (`p_V `=` v_1`) for` p_i `<` v_2 `do` e**:**
If e does not contain any `map` statements, we do nothing special with the `for`-loop. If it does contain `map`-statements however, we would like to bring the outer parallel dimension(s)s (remember, this `for`-loop is itself inside of at least one `map`) next to the parallel dimensions of the inner `map`(s). This can be done by

[4] Consider what would be the declared shapes of the array returned by the residual loop nest. It should intuitively be `[[int,n],p]` for "many values of `p`", but this is not expressible.

```
let {asss, bss} = map(fn {[[int,m],m], [int,m]} (ps) =>
    let ass = map(fn [int,m] (p) =>
                  let cs = scan(+, 0, iota(p))
                  let f = reduce(+, 0, cs)
                  let as = map(+f, ps)
                  in as,
               ps)
    let bs = loop (ws=ps) for i < n do
      let ws' = map(fn int (as, w) =>
                    let d = reduce(+, 0, as)
                    let e = d + w
                    let w' = 2 * e
                    in w',
                  ass, ws)
      in ws'
    in {ass, bs},
  pss)
```

(a) Initial program, we inspect the loop.

```
let asss = map(...)
let bss = loop (wss=pss) for i < n do
  let wss' =
    map(fn [int,m] (ass, ws) =>
      let ws' = map(fn int (as, w) =>
                    let d  = reduce(+, 0, as)
                    let e  = d + w
                    let w'  = 2 * e
                    in w',
                  ass, ws)
      in ws',
    asss, wss)
  in wss'
```

(b) After the loop has been distributed and interchanged.

```
let asss = map(...)
let bss = loop (wss=pss) for i < n do
  let wss' =
    map(fn [int,m] (ass, ws) =>
      let ws' = map(fn int (as, w) =>
                    let d  = reduce(+, 0, as)
                    let e  = d + w
                    let w' = 2 * e
                    in w',
                  ass, ws)
      in ws',
    asss, wss)
  in wss'
```

(c) None of these scalar statement give rise to new kernels by themselves...

```
let asss = map(...)
let bss = loop (wss=pss) for i < n do
  let dss =
    map(fn [int,m] (ass) =>
      let ds = map(fn int (as) =>
                   let d = reduce(+, 0, as)
                   in d,
                 ass)
      in ds,
    asss)
  let wss' =
    map(fn [int,m] (ws, ds) =>
      let ws' = map(fn int (w, d) =>
                    let e = d + w
                    let w' = 2 * e
                    in w',
                  ws, ds)
      in ws',
    wss, dss)
  in wss'
```

(d) ...but we wish to create a kernel containing the reduction by itself (segmented reduction), and hence distribute every statement after it as a new kernel.

```
let asss = map(fn {[int,l], [int,m]} (ps) =>
    let ass = map(fn int (p) =>
                  let cs = scan(+, 0, iota(p))
                  let f = reduce(+, 0, cs)
                  let as = map(+f, ps)
                  in as,
               ps)
    in ass,
  pss)
let bss = loop ...
```

(e) This `map` can be distributed by itself to exploit the parallelism of all three `map` nesting levels.

```
let fss = map(fn {[int,l], [int,m]} (ps) =>
    let fs = map(fn int (p) =>
                 let cs = scan(+, 0, iota(p))
                 let f = reduce(+, 0, cs)
                 in f,
              ps)
    in fs,
  pss)
let asss = map(fn {[int,l], [int,m]} (ps, fs) =>
    let ass = map(fn int (f) =>
                  let as = map(+f, ps)
                  in as,
               fs)
    in ass,
  pss, fss)
let bss = loop ...
```

(f) Neither the remaining `reduce` or `scan` can be distributed, due to a loop-variant size (`iota(p)`), so they are sequentialised and only the two outer map-nestings are kept parallel.

Figure 11: Extracting kernels from a complicated nesting

interchanging the outer parallel dimensions inwards, but *only* if we could distribute the loop by itself.

To get an intuition for the validity of the interchange, suppose that we have a `map` containing a `for`-loop, where each iteration of the `for`-loop applies the function f to the loop-variant variable (the following is simplified syntax for exposition purposes:

```
map(fn (x) =>
      loop (x'=x) for i < n do f(x),
    xs)
```

Suppose $\texttt{xs} = [x_0, x_1, \ldots, x_m]$, then the result of the map is

$$[f^n(x_0),\ f^n(x_1),\ \ldots,\ f^{n-1}(x_m)].$$

If we interchange the map, then we get the following:

```
loop (xs'=xs) for i < n
  map(f, xs')
```

At the conclusion of iteration i, we have

$$xs' = [f^{i+1}(x_0),\ f^{i+1}(x_1),\ \ldots,\ f^{i+1}(x_m)].$$

Thus, at the conclusion of the last iteration $i = n - 1$, we have obtained the same result as the non-interchanged `map`. Note that the validity of the interchange does not depend on whether the `for`-loop contains a `map` itself.

Case `let` p `= reduce(fn` $\bar{t}^{(m)}$ `(...) =>` e`, ...)`:
A `reduce` statement is parallel by itself, and if we encounter it inside of a `map`, it is the equivalent of a segmented reduction, which we can turn into a kernel by itself. However, there is one interesting special case, namely if operator of the reduction is a `map` on its operands. This occurs, for example, if we are using vector addition to reduce the rows of a matrix. Cheating a bit with the syntax, we can write this as **reduce**(**map**(+), a, b). We can interchange the `reduce` and `map` if we also transpose the input and output, yielding **transpose**(**map**(**reduce**(+), **transpose**(a), **transpose**(b)). The advantages of this transformation are twofold: (1) we bring two levels of `map` parallelism together, and (2) we avoid a potentially large array-typed accumulator in the reduction. When generating OpenCL code, it is beneficial to store accumulators in so-called *local memory*, but this is sharply limited in size, and we do not know in advance the sizes of arrays.

Case `let` p `= scan(...)`:
Same considerations as for `reduce`, including the possibility of interchange with inner `map`.

Case `let` p `= redomap(...)`:
Same considerations as for `reduce` and `scan`, although the possibility of interchange with inner `map` is only available if we first decompose the `redomap` into a separate `map` and `reduce`.

One very important detail is that we do not distribute inside `if`-expressions contained in a parallel nest. This implies that a conditional expression is always sequentialised.

5.3 Choice Means Being Able to Make the Wrong One

Several of the possible choices in section 5.2 overlap in various cases, and all of them overlap with the fallback choice (sequentialisation). Hence, the Futhark compiler needs to make decisions about how to distribute when there are several options. Consider a `reduce` inside of a `map`, such as the one on figure 10. While we can extract a parallel segmented reduction from the distributed loop nest, we could also simply sequentialise the reduction, resulting in a number of parallel threads each running a sequential loop. If the degree of parallelism in the outer `map` is sufficient to saturate the hardware, this would result in much faster code than the comparatively heavyweight communication required by the threads involved in a segmented reduction.

Unfortunately, there is no way for a compiler to make the right choice, as it depends on available hardware parallelism and the dimensions of the input data. With a large outer dimension and a small inner dimension, it is not worth parallelising the reduction, but if the outer dimension is small (or in an extreme case: a single row), parallelising the reduction is crucial to performance. In fact, if the outer dimension is particularly small, it may be worthwhile to sequentialise the *outer* map, and simply launch several (non-segmented) reduction kernels concurrently.

The right solution, which we have planned for future work, is to generate *multi-versioned-code*: whenever we have a choice in how to perform kernel extraction, we should generate *all possible variants* (possibly with some limit to prevent combinatinatorial explosion), and construct an `if` expression that selects the optimal version for the given input size.

For now, our compiler follows these rules of thumb: nested `reduce` or `scan` statements are always turned into a segmented operation, while `redomap` and `streamRed` statements are sequentialised. Note that this applies only to statements found within a map nesting: `redomap` and `streamRed` are turned into reduction kernels when encountered as the outermost level of parallelism.

6. Example: k-means Clustering

So far, we have presented several unusual language features, including in-place updates and the `streamRed` construct. In this section, we will demonstrate how they are used to efficiently implement a nontrivial algorithm: k-means clustering of n d-dimensional points, for arbitrary $k, n, d > 0$ (although k tends to be small in practice, e.g. 5). This problem is from the Rodinia benchmark suite [11], which we also discuss in section 7.2. The algorithm we wish to implement is as follows:

1. k initial d-dimensional "means" are randomly generated within the data domain.
2. k clusters are created by associating every point with the nearest mean.
3. The centroid of each of the k clusters becomes the new mean.
4. Steps (2) and (3) are repeated until convergence has been reached. In our case, the convergence criteria is that no point has changed cluster membership.

The most complicated part of the algorithm is step (3), where we have to compute cluster centres. A subproblem is to compute the number of points in each of the k cluster, which is what we will focus on here[5]. We have the arrays `points : [[f32,d],n]` and `membership : [int,n]`, and we must produce an array `counts : [int,k]`. One possible implementation is to use a parallel `map` to compute “increments”, and then perform a reduction of these increments. This solution, shown in figure 12a, is is parallel, but not work-efficient. Intuitively, the problem is that each of the `n` points give rise to `k` units of work, as we produce `n` arrays of `k` elements each, only to sum them all together. Unless we are executing on a computer that can exploit all $n \times k$ degrees of parallelism, we are paying an overhead for more parallelism that we can use. In contrast, the sequential implementation shown in figure 12b does less work, but is–alas–sequential.

What we need is a combination: a language construct that can be “dialed” up and down to exploit at much parallelism as is needed to take full advantage of the machine, but runs efficient sequential code within each thread. This is where `streamRed` enters the picture. Using `streamRed`, seen on figure 12c, we specify a parallel

[5] Computing the cluster centres is done in approximately the same way, but with another dimension on top.

```
let increments = map(fn [int,k] (int cluster) =>
                        let incr = replicate(k, 0)
                        let incr[cluster] = 1
                        in incr,
                       membership)
let counts = reduce(fn [int,k] ([int,k] x, [int,y]) =>
                      map(+, x, y),
                    {replicate(k, 0)}, increments)
```

(a) Parallel calculation of counts

```
let counts = loop (counts = replicate(k,0))
  for i < n do
    let cluster = membership[i]
    let new_counts =
      counts with [cluster] <- counts[cluster] + 1
    in new_counts
```

(b) Sequential calculation of counts

```
let counts = streamRed(map(+),
       fn [int,k] (chunksize, [int,k] acc,
                   [int,chunksize] chunk) =>
          let res =
            loop (acc) for i < chunksize do
              let cluster = chunk[i]
              let acc[cluster] = acc[cluster]+1
              in acc
          in res,
       {replicate(k,0)}, membership)
```

(c) Streamed calculation of counts

Figure 12: Counting cluster sizes

```
let num_threads = get_num_threads()
-- produces an array of type [[int,k],num_threads]
let per_thread_results =
  mapPerThread(num_threads, sequential code..., membership)
-- combine the per-thread results
let cluster_sizes =
  reduce(map(+), replicate(k, 0), per_thread_results)
```

(a) Stream kernels before interchange

```
...
-- combine the per-thread results
let cluster_sizes =
  map(reduce(+, 0), transpose(per_thread_results))
```

(b) After interchanging the reduction

Figure 13: Kernel extraction for k-means `streamRed`

reduction function and a sequential fold function to use for reducing the array. The compiler is able to exploit as much parallelism as is optimal on the hardware, and can use our sequential code inside each thread. Note how we make use of in-place updates to provide an efficient sequential fold - this is safe, because we only do in-place updates on our formal parameters, which we are given exclusive use of. Note also that this implementation performs only a single pass of the `points` array.

There is one more trick to this use of `streamRed`, which relates to kernel extraction. To start with (and here we are moving a little outside our core language), the stream will be broken up by the kernel extractor as shown on figure 13a - this makes use of the "pseudo-SOAC" `mapPerThread`, which divides an array among some number of threads, performs a sequential fold in every thread, and returns one result per thread. A reduction using the parallel operator from the `streamRed` is used to combine the per-thread results. However, the kernel extractor recognises the pattern of a `reduce` with an inner `map` and interchanges the two, leading to the code on figure 13a, which uses a segmented scan (with regular segments) to combine the per-thread results.

The performance of this implementation does rely on one assumption - that k is small. This is due to the in-place update at a data-dependent index in the sequential fold, which causes uncoalesced access to memory if different GPU threads within the same 32-thread *warp* access different indices in the same cycle. With a small k, this effect is lessened, but for large k, a different algorithm with more passes over the input may be preferable.

7. Evaluation

We open this section with four performance related observations, then describe the experimental methodology in Section 7.1 and discuss results from nine Rodinia benchmarks and five Accelerate and FinPar [1] benchmarks in Sections 7.2, 7.3, and 7.4, respectively:

1 In several cases, such as Rodinia's `Backprop`, `Hotspot`, `kmeans`, and FinPar's `LocVolCalib`, Futhark's sequential code is up to a factor of two slower than the hand-written `C` code. This is due to inefficiencies (unnecessary copying) in the sequentialization phase that transforms bulk-parallel operators to loops with in-place updates, and are subject to future work. In particular, kernel code would equally benefit from such improvements.

2 Our kernel-extraction algorithm does not necessarily takes advantage of all application parallelism, e.g., it would sequentialize a `map` that appears inside an `if` branch of an outer map, and a `redomap` that is nested inside a `map` nest. Such examples of efficient sequentialization occur in Rodinia's `LavaMD` and `Myocyte` benchmarks and allow Futhark's code to be competitive.

3 However a `reduce` that is perfectly nested in a `map` nest is executed in parallel by Futhark as a segmented reduce, such as in Rodinia's `Backprop` and `Kmeans`. This is currently implemented as a general segmented scan followed by a pack, which is rather inefficient and can be improved by a significant factor, which is to say there is still significant room for improvement.

4 Finally, the most impactful optimization for the GPGPU code is ensuring coalesced accesses to global memory. In essence, several benchmarks, such as FinPar's `LocVolCalib` and Rodinia's `CFD`, `Kmeans`, `Myocyte`, and `LavaMD`, exhibit kernels in which one or several innermost dimensions of the mapped arrays are processed sequentially inside the kernel, either because they correspond to dependent computation, i.e., loops with in-place updates, or because they have been sequentialized by the kernel extraction. A naive translation of this code would lead to consecutive threads accessing global memory with a stride equal to the size of the inner (non-parallel) array dimensions, a.k.a. non-coalesced, which may generate one-order-of-magnitude slowdowns. While not in the scope of this paper, the Futhark compiler solves this by, intuitively, transposing the parallel dimensions of the array innermost, and the same for the result array.

7.1 Experimental Methodology and Hardware

We have translated a number of programs from Rodinia and Accelerate repositories to Futhark. The translation of Accelerate programs is relatively straightforward; for the Rodinia ones, we have translated parallel loops to bulk-parallel operators such as `map`, `reduce`, while preserving at our best the original code structure. Both the Futhark compiler and translated benchmarks are publicly available and can be straightforwardly tested, but anonymous-review concerns prevent us to disclose the location yet.

We use the default Futhark-compiler flags (i.e., no tuning/flags) to compile the translated code to a binary (based on sequential-`C` and parallel-`OpenCL` code), and run it with `binary -t time_file < inp_file`, where `inp_file` contains the dataset and `time_file` records the total runtime minus the time taken for (i) reading the input data, validating and writing the result to file, and (ii) GPU context creation and build time (but memory-transfer time between host and device is accounted for). We use the time instrumentation of our competitors to measure the same runtime (as defined above), and where not straightforward, e.g., several Rodinia cases, we conservatively err in favor of our competitors.

Reported running times are averaged across twenty runs, and are measured on an Intel(R) system running OPEN-SUSE 12.3, and consisting of 16 Xeon(R) cores, model `E5-2650 v2`, running at 2.60 GHz, and 128`GB` memory. The GPGPU is a GeForce GTX 780 Ti NVIDIA, having 3 Gbytes of global memory, 2880 cores running at 1.08 GHz, and 1.5 Mbytes of L2 cache. We use GCC 4.8.4 and CUDA 6.0 to compile `C++/C` and `OpenCL/CUDA` code.

7.2 Evaluation of Nine Rodinia Benchmarks

Figure 14 shows the sequential-CPU and parallel-GPGPU *speedups*, respectively, computed as the ratio between Rodinia's and Futhark's running times, hence the higher the bars the better.

Backprop is tested on a dataset that has the size of the input layer equal to 2^{20}. We implement the randomized initialization phase by a `map` with the Sobol independent formula [1], while Rodinia uses a sequential loop that calls `C`'s `rand()`. This explains in part the $3\times$ slowdown of our sequential code (`rand()` is faster). The parallel computation of Sobol numbers is also the main reason for the $5\times$ speedup of the GPGPU code: Rodinia runtime does not include the initialization, but takes into account the host-to-device memory transfer time, which is expensive. The Futhark runtime includes the initialization, but this takes place directly on the GPGPU, hence the host-to-device transfer time is negligible. However, the kernel running time of the training phase (excluding initialization) are roughly equal in Rodinia and Futhark ($\sim 10\ ms$).

CFD is tested on the `fvcorr.domn.193K` dataset, with 1300 iterations of the convergence loop.

Hotspot is tested on an image of size 1024×1024, with a convergence loop count of 360. Rodinia's `OpenCL` version uses time tiling [17], which seems to pay off on the tested hardware.

Kmeans is tested on the `kdd_cup` dataset. The main reason for our speedup is that Rodinia does not parallelize the computation of the new cluster centers, which is a segmented reduce; see Section 6.

LavaMD is tested on the dataset with `boxes1d=10`. Our speedup is a bit surprising since the Rodinia implementation uses tile-blocking in fast (local) memory to optimize an global-memory access that is invariant to the innermost-parallelized `map`, while Futhark does not, i.e., uses only global memory. Tested on an older GPGPU[6], Rodinia gains the upper hand by a factor of $1.33\times$, i.e., $15\ ms$ to $20\ ms$, but Rodinia's version in which the use of fast memory is deselected is $3.5\times$ slower than Futhark, i.e., $65\ ms$.

Possible explanations are that (i) in Futhark's case, the L1 caching of the invariant access alleviates much of the global-memory overhead, and (ii) that the Rodinia version might exhibit fast-memory bank conflicts. Also, the Rodinia version exhibits non-coalesced accesses to global memory on the `d_box_gpu` array, whose element type is a structure named `box_str` that contains six integers and a 26 element array, each element being a five-int tuple. In contrast, Futhark automatically transforms arrays of tuples to tuples of arrays and then achieves coalesced access by transposition.

Myocyte is tested on a dataset with `workload=65536` and `xmax=3`. Since the only dataset available in Rodinia is for `workload=1` and the amount of data parallelism is equal to `workload` we have extended that dataset by replicating and perturbing the original input directly on GPGPU. For this reason the Rodinia's reported CUDA runtime refers to kernel-time only, i.e., the host-device transfer time is not accounted for. We attribute our speedup to automatic memory-coalescing optimization, which is tedious to do by hand on such large programs.

NN is tested on an enlarged dataset that duplicates the default Rodinia's dataset 20 times, and also the number of nearest neighbors is raised accordingly from 5 to 100. The reasons for the sequential slowdown of Rodinia is that its `OpenMP` and GPGPU versions differ significantly; our implementation follows Rodinia's GPGPU version. The reason for the Futhark GPGPU version being more than twenty times faster is that in Rodinia, the 100 `reduce` operations for finding the nearest neighbors are sequentialized on the CPU. This is probably because the reduce operator is atypical: it computes both the minimal value and its corresponding index.

Pathfinder is tested on an array of size 100000 with a convergence loop of count 100. Both Futhark's sequential and `OpenCL` versions are significantly faster than Rodinia's, mainly because Rodinia uses time tiling in both cases, which, unlike `Hotspot`, does not seem to pay off on the tested hardware.

SRAD is tested on the default Rodinia's dataset, i.e., an image of size 502×458 with a convergence loop of count 100. Futhark sequential version is faster because the Rodinia's code in `srad_v1` exhibits bad locality. We suspect that Futhark's `OpecCL` code is $1.2\times$ faster because the `reduce` implementation is more efficient.

7.3 Evaluation of Four Accelerate Benchmarks

The benchmarks in this section are from the `accelerate-examples` package version 0.15.1.0, and compiled with version 0.15.1.0 of `accelerate` and version 0.15.1.1 of `accelerate-cuda`, these being the most recently released versions at the time of writing. Tests with unreleased versions from the Accelerate Git repository showed similar results. Runtime of Accelerate implementations were measured through their built-in `--benchmark` command line option, and Futhark versions were measured as we did with Rodinia, and again without any benchmark-specific tuning. We only report the runtime for one dataset per benchmark, typically the largest one available, which tends to favor Accelerate.

Figure 15 show the parallel *speedups*, respectively, computed as the ratio between Accelerate's and Futhark's running times, hence the higher the bars the better.

Crystal consists of two nested maps that for the chosen dataset have size 4000×4000, whose body contains some scalar computation and a sequential fold.

Fluid is a rank-1 stencil computation on a 3000×3000 grid, where each grid point executes a 20-iteration sequential fold. The Accelerate version uses a stencil primitive, but the Futhark version is merely a `map` over `iota` followed by explicit array indexing.

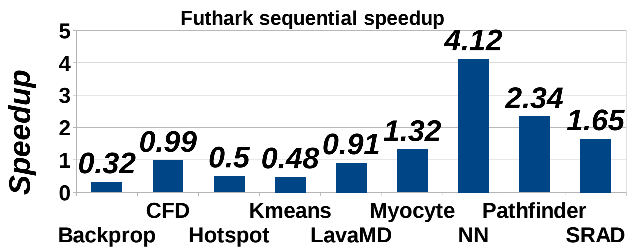
Mandelbrot is a naively written Mandelbrot implementation (no clever load balancing), which we evaluate for a 800×600 grid. Unusually among all ported Accelerate benchmarks, Futharks relative performance improves as we increase the grid size, and reaches $5.87\times$ for a grid of size 8000×8000.

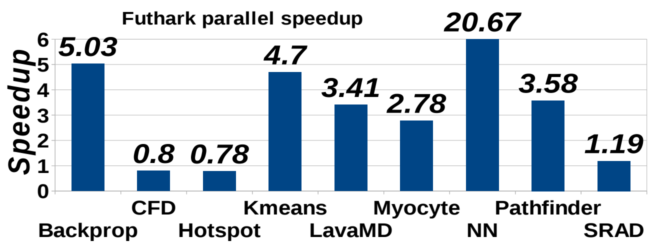
N-body is a naive n-body simulation, consisting of a width-n map where each element performs a fold over each of the n bodies. Accelerate provides two possible implementations; we have chosen the fastest one (`Naive2`). We are reporting runtime for $n = 10^5$.

7.4 Evaluation of Local Volatility Calibration

`LocVolCalib` belongs to FinPar suite [1], and we have implemented the version corresponding to a sequential TRIDAG computation. The result can be seen on the "VC" bars of Figure 15. We beat the hand-written OpenCL implementation for the small dataset, as it is optimised for larger datasets - the small dataset requires a parallel TRIDAG implementation to fully saturate the GPU.

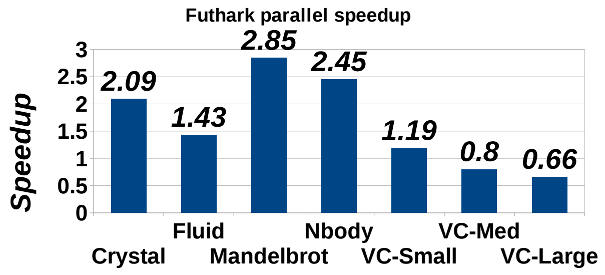
[6] An NVIDIA GeForce GTX680, which has 1536 cores, where `LavaMD` is faster, but Rodinia's `Hotspot` and `SRAD` are much slower than Futhark. In Futhark's case, the kernel time of all benchmarks are about a factor of $2\times$ faster on the current GPGPU in comparison with the old GPGPU, which is what one would expect since the number-of-cores ratio is also about 2.





Rodinia Runtimes	Backprop	CFD	Hotspot	Kmeans	LavaMD	Myocyte	NN	Pathfinder	SRAD
`Sequential-C` (sec)	.5214	191.1	1.507	3.882	7.795	47.65	.4845	1.684	.9330
`GPGPU-OpenCL` (sec)	.1014	1.293	.0419	2.112	.0406	2.552	.2404	.0266	.0186

Figure 14: Sequential and OpenCL speedup (their runtime/our runtime) on 9 Rodinia benchmarks. Higher is better.



Runtimes in miliseconds

Impl.	Crystal	Fluid	N-body	Mandelbrot
Accelerate	126.7	154.5	622.3	1.29
Futhark	60.6	107.87	253.6	0.45

LocVolCalib	Small	Medium	Large
FinPar	257.9	106.8	859.9
Futhark	217.1	133.6	1302.9

Figure 15: GPU Speedup on Accelerate and FinPar Benchmarks

This benchmark is by far the most challenging one we present due to its structure, which contains an outer `map`, containing a sequential `for`-loop, which itself contains several more `map`s. Exploiting all parallelism requires the compiler to interchange the outer `map` and the sequential loop, as described in section 5.2. Further, the inner sequential TRIDAG computation consists of a complicated dependent loop that requires the use of in-place updates.

8. Related Work

Most related work to Futhark is SAC [15, 16], which seeks to provide a common ground between functional and imperative domains for targeting parallel architectures, including both multi-processor architectures [13] and massively data-parallel architectures [18]. SAC uses `with` and `for` loops to express map-reduce style parallelism and sequential computation, respectively. More complex array constructs can be compiled into `with` and `for` loops, as demonstrated, for instance, by the compilation of the APL programming language [21] into SAC [14]; a prototype APL-to-Futhark compiler based on TAIL [12] is also being developed. In contrast to SAC, Futhark holds on to the SOAC combinators also in the intermediate representations in order to perform critical optimizations, such as fusion, even in cases involving filtering and scans, which are not straightforward constructs for SAC to cope with.

Much related work has been carried out in the area of supporting nested parallelism, including the seminal work on flattening of nested parallelism in NESL [6, 7] and in the more recent work on flattening [5, 28]. Investigating the use of a moderate/multi-version flattening transform for supporting parallel implementations of arbitrarily nested parallelism in Futhark is future work.

Recent work [26] has shown how rewrite rules can be used to obtain some of the transformations performed by our compiler. The use of rewrite rules which can be applied in many different ways opens the door to autotuning, but it may be more difficult to express more complicated single transformations.

Finally, compiler work in imperative setting reveals very useful complementary with our work on Futhark. Such solutions typically use complex polyhedral analysis [24] and excel at speeding up loops of relatively small sizes, for example by time tiling [17], but such analysis might not scale well to detect parallelism or to perform distribution of larger loops with complex control flow. For example, Pencil [2] reports speedups comparable to ours on Rodinia, but these speedups result from index-based analysis, such as tiling or skewing guided by extensive, per-dataset auto tuning. In comparison, our speedups are rooted in being able to express all parallelism, and in analysis that scales to program level – fusion, memory coalescing, kernel extraction – precisely because it relies on higher-level language invariants (no tuning was performed).

9. Conclusion

This paper has presented an optimizing compiler for Futhark, a purely functional language tailored to aid the compiler in generating efficient GPGPU code. The perspective has been to seek a common ground between imperative and functional approaches that combines advantages. For example, we support loops and in-place updates and rely heavily on loop-like hoisting to hoist memory allocations outside GPGPU kernels, but we also rely on scalable analysis that exploits the higher-order semantics of functional combinators.

We have presented (i) how to support in-place updates in Futhark's type system, (ii) how second-order array combinators (SOACs) can express symbolically both all available parallelism and efficient sequentialization alternatives, and (iii) how to aggressively compose (fusion) the program and then to decompose it yet again into kernels in a manner that maximizes the amount of efficient (common-case) parallelism.

Finally, we have validated our fully-automatic compiler, which supports straightforward, flag-and-human-free GPGPU compilation on a number of 14 Rodinia, Accelerate and FinPar benchmarks and we have reported a harmonic-mean speedup of 1.75× against competitors, with the highest observed slowdown being 1.52×.

While it has been a forgone conclusion for a while now that the advance of massively parallel hardware such as GPGPUs should favor functional languages, we believe that this works brings a significant contribution into demonstrating this potential.

References

[1] C. Andreetta, V. Bégot, J. Berthold, M. Elsman, F. Henglein, T. Henriksen, M.-B. Nordfang, and C. E. Oancea. Finpar: A parallel financial benchmark. In *ACM TACO*, 2016.

[2] R. Baghdadi and et al. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *Proc. Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2015.

[3] E. Barendsen and S. Smetsers. Conventional and Uniqueness Typing in Graph Rewrite Systems. In *Found. of Soft. Tech. and Theoretical Comp. Sci. (FSTTCS)*, volume 761 of *LNCS*, pages 41–51, 1993.

[4] E. Barendsen and S. Smetsers. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.

[5] L. Bergstrom and J. Reppy. Nested data-parallelism on the GPU. *SIGPLAN Not.*, 47(9):247–258, Sept. 2012. ISSN 0362-1340. doi: 10.1145/2398856.2364563. URL http://doi.acm.org/10.1145/2398856.2364563.

[6] G. E. Blelloch. *Vector models for data-parallel computing*, volume 75. MIT press Cambridge, 1990.

[7] G. E. Blelloch. Programming Parallel Algorithms. *Communications of the ACM (CACM)*, 39(3):85–97, 1996.

[8] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a Portable Nested Data-Parallel Language. *Journal of parallel and distributed computing*, 21(1):4–14, 1994.

[9] M. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A Status Report. In *Int. Work. on Decl. Aspects of Multicore Prog. (DAMP)*, pages 10–18, 2007.

[10] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009. doi: 10.1109/IISWC.2009.5306797.

[12] M. Elsman and M. Dybdal. Compiling a Subset of APL Into a Typed Intermediate Language. In *Procs. Int. Workshop on Lib. Lang. and Compilers for Array Prog. (ARRAY)*. ACM, 2014.

[13] C. Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming (JFP)*, 15(3): 353–401, 2005.

[14] C. Grelck and S.-B. Scholz. Accelerating APL programs with SAC. In *Proceedings of the Conference on APL '99: On Track to the 21st Century*, APL'99, pages 50–57. ACM, 1999.

[15] C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multithreaded execution. *Int. Journal of Parallel Programming*, 34(4):383–427, 2006.

[16] C. Grelck and F. Tang. Towards Hybrid Array Types in SAC. In *7th Workshop on Prg. Lang., (Soft. Eng. Conf.)*, pages 129–145, 2014.

[17] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid hexagonal/classical tiling for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2670-4. doi: 10.1145/2544137.2544160. URL http://doi.acm.org/10.1145/2544137.2544160.

[18] J. Guo, J. Thiyagalingam, and S.-B. Scholz. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *Procs. Workshop Decl. Aspects of Multicore Prog. (DAMP)*, pages 15–24. ACM, 2011.

[19] T. Henriksen and C. E. Oancea. A t2 graph-reduction approach to fusion. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 47–58. ACM, 2013.

[20] T. Henriksen, M. Elsman, and C. E. Oancea. Size slicing: a hybrid approach to size inference in futhark. In *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*, pages 31–42. ACM, 2014.

[21] K. E. Iverson. *A Programming Language*. John Wiley and Sons, Inc, May 1962.

[22] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising Purely Functional GPU Programs. In *Procs. of Int. Conf. Funct. Prog. (ICFP)*, 2013.

[23] C. Oancea, C. Andreetta, J. Berthold, A. Frisch, and F. Henglein. Financial Software on GPUs: between Haskell and Fortran. In *Funct. High-Perf. Comp. (FHPC'12)*, 2012.

[24] L. Pouchet and et al. Loop Transformations: Convexity, Pruning and Optimization. In *Int. Conf. Princ. of Prog. Lang. (POPL)*, 2012.

[25] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *SIGPLAN Lisp Pointers*, V(1):288–298, Jan. 1992. ISSN 1045-3563.

[26] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. *SIGPLAN Not.*, 50(9):205–217, Aug. 2015. ISSN 0362-1340. doi: 10.1145/2858949.2784754. URL http://doi.acm.org/10.1145/2858949.2784754.

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

[28] Y. Zhang and F. Mueller. CuNesl: Compiling nested data-parallel languages for SIMT architectures. In *Proceedings of the 2012 41st International Conference on Parallel Processing*, ICPP'12, pages 340–349, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4796-1.