Futhark: Purely Functional GPU-Programming with Nested Parallelism and In-Place Array Updates

Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea

Department of Computer Science, University of Copenhagen (DIKU), Denmark

athas@sigkill.dk, ngws@metanohi.name, mael@di.ku.dk, henglein@diku.dk, cosmin.oancea@diku.dk

Abstract

Futhark is a purely functional data-parallel array language that offers a machine-neutral programming model and an optimising compiler that generates OpenCL code for GPUs. This paper presents the design and implementation of three key features of Futhark that seek a suitable middle ground with imperative approaches. First, in order to express efficient code inside the parallel constructs, we introduce a simple type system for in-place updates that ensures referential transparency and supports equational reasoning. Second, we furnish Futhark with parallel operators capable of expressing efficient strength-reduced code, along with their fusion rules. Third, we present a flattening transformation aimed at enhancing the degree of parallelism that (i) builds on loop interchange and distribution but uses higher-order reasoning rather than array-dependence analysis, and (ii) still allows further locality-of-reference optimisations. Finally, an evaluation on 16 benchmarks demonstrates the impact of the language and compiler features and shows application-level performance competitive with hand-written GPU code.

CCS Concepts • **Computing methodologies** → **Parallel programming languages**; • **Software and its engineering** → **Source code generation**; *Software performance*

Keywords functional language, parallel, compilers, GPGPU.

1. Introduction

Massively parallel graphics processing units (GPUs) are today commonplace. While GPUs provide high peak performance, they are difficult to program, and require understanding of hardware details to obtain good performance. A rich body of language and compiler solutions aim at enhancing programmer productivity by transparently mapping hardware-independent programs to GPUs.

On one side, we find sophisticated analyses [16, 27, 41, 42, 55, 56], centered on dependency analysis of affine loop nests in low-level imperative languages such as C and Fortran. These analyses often find themselves "fighting the language", not only due to the lack of high-level invariants (e.g., **filter** and **scan** patterns are difficult to recognize and optimize), but also because aliasing, non-affine indexing, and complex control flow may restrict applicability.

On the other side, there are a number of embedded (morally functional) data-parallel languages [2, 15, 35, 48, 51, 52] that express parallelism explicitly by means of bulk operators, which allow the compiler to reason at a higher level about restructuring the program. The downside is that some of these languages do not support in-place updates or explicit indexing inside parallel operators [15, 48, 52], and none of them systematically utilizes (imperfectly) nested parallelism. A notable exception to the latter is NESL [7, 11], which flattens all nested parallelism under asymptotic work-depth guarantees. However, such an aggressive flattening approach is often inefficient in practice, because the transformation prevents further locality-of-reference optimizations and efficient sequentialization of excess parallelism.

This paper presents Futhark [28–32], a simple but expressive purely-functional parallel array language with guaranteed race-free semantics.The language and compiler design seeks a common ground that combines the advantages of functional and imperative features.

First, we present a type system extension that supports in-place modification of arrays, without compromising language purity (i.e., the parallel semantics of operators is still guaranteed). The extension builds on uniqueness types [4, 5] and is formalized, including alias analysis, by inference rules in Section 3. The contribution is not in linear-type theory (more powerful systems [24, 54] exist), but rather in designing a simple system that does not overcomplicate analysis. To our knowledge, it is the first of its kind used in a high-performance purely functional language. The motivation for in-place updates is twofold, namely (i) to express dependent

code inside parallel constructs, and (ii) to sequentialize efficiently the excess parallelism (see example in Section 2.4).

Second, we furnish Futhark with a set of *streaming* operators that allow the user to express strength-reduction invariants generically (i.e., without compromising the available parallelism), and also ease code generation. These constructs and their fusion rules are presented in Section 4.

Third, Section 5 presents a transformation that rearranges the available parallelism into perfect nests of parallel operators that optimise the degree of parallelism that can be statically (and efficiently) exploited. In comparison to imperative approaches, we also build on loop distribution and interchange [36], but we lift the reasoning to rely on the higher-order operator semantics rather than on low-level index analysis. This approach enables (more) aggressive rewrite rules that transform not only the schedule (i.e., the iteration space) but also the storage. In comparison to NESL [11], we support in-place updates, more operators, and an algorithm that can flatten only some of the top-level parallelism, i.e., it stops before introducing irregular arrays and before destroying the program structure needed for spatial and temporal locality optimizations. We demonstrate the latter by implementing memory coalescing and simple block tiling.

Finally, we present an evaluation of Futhark-generated OpenCL code on 16 benchmarks ported from Rodinia [17], Accelerate [39], Parboil [50], and FinPar [1], that demonstrates performance competitive to reference implementations on AMD and NVIDIA GPUs: speedup ranges from about $0.6\times$ (slower) on FinPar's LocVolCalib benchmark to $16\times$ (faster) on Rodinia's NN benchmark. On the 12 benchmarks that come with a low-level CUDA/OpenCL implementation, the geometric mean of Futhark's speedup is $1.81\times$. We attribute the positive speedup to the tediousness of writing low-level GPU code, for example, the reference implementation leaving unoptimised (i) the spatial/temporal locality of reference (Myocyte/MRI-Q), or (ii) some (nested) reduce operators (e.g., NN, Backprop, K-means, SRAD). Futhark is slower on 4 out of 12 benchmarks with a geometric mean of $0.79\times$, which we believe is a better estimation of where Futhark stands in relation to hand-optimised GPU code. In summary, the main contributions of this paper are:

- The introduction of a simple type system (based on uniqueness types [4, 5]) supporting race-free in-place updates in a purely-functional, data-parallel language. An intuitive demonstration on a code example was given in [28] but the typing rules were not presented there.
- The fusion rewrite rules for streaming SOACs, which capture and express efficient strength-reduced code. A simplified version of the streaming SOACs was introduced as a functional notation in [1], and non-overlapping aspects of the fusion engine were presented in [28, 32].
- The flattening algorithm that exploits partial top-level parallelism, and allows further locality optimisations.
- An experimental validation on 16 benchmarks that demonstrates (i) application-level performance competitive with hand-written OpenCL code, and (ii) significant impact of in-place updates, fusion, coalescing and block tiling. The research artifacts are publicly available at `https://github.com/HIPERFIT/futhark-pldi17`

2. Preliminaries: Calculus and Language

The theoretical foundation of Futhark are the list homomorphisms of Bird and Meertens [9], realised in the form of purely functional parallel second-order array combinators (SOACs). Their rich equational theory for semantics-preserving transformations are employed in Futhark for fusion, streaming, and flattening of parallelism. We first discuss this theory in Section 2.1, then we introduce the concrete syntax of Futhark's core language in Section 2.2, then we show the compiler pipeline in Section 2.3, and we demonstrate the use of in-place updates and streaming operators on a code example in Section 2.4.

2.1 Array Combinator Calculus

We first describe the basic SOACs and later introduce *streaming combinators*. The basic SOACs include (i) **map**, which constructs an array by applying its function argument to each element of the input array, (ii) **reduce**, which applies a binary-associative operator $\oplus$ to all elements of the input, and (iii) **scan**, which computes all prefix sums of the input array elements. Their types and semantics are shown below:

$$
\begin{aligned}
&\mathrm{map} : (\alpha \to \beta) \to \Pi n.[n]\alpha \to [n]\beta \\
&\mathrm{map}\ f\ [a_1, \ldots, a_n] = [f\ a_1, \ldots, f\ a_n] \\
&\mathrm{reduce} : (\alpha \to \alpha \to \alpha) \to \alpha \to \Pi n.[n]\alpha \to \alpha \\
&\mathrm{reduce}\ \oplus\ 0_\oplus\ [a_1, \ldots, a_n] = 0_\oplus \oplus a_1 \oplus \ldots \oplus a_n \\
&\mathrm{scan} : (\alpha \to \alpha \to \alpha) \to \alpha \to \Pi n.[n]\alpha \to [n]\alpha \\
&\mathrm{scan}\ \oplus\ 0_\oplus\ [a_1, \ldots, a_n] = [a_1, \ldots, a_1 \oplus \ldots \oplus a_n]
\end{aligned}
$$

Here $[a_1, \ldots, a_n]$ denotes an array literal, $[n]\tau$ denotes the type of arrays with n elements of type τ, and $0_\oplus$ denotes the neutral element of the binary associative operator $\oplus$. The Πn notation indicates where the size n becomes fixed; it indicates, for instance, that we can partially apply map to a function and apply the resulting function to arrays of different sizes.

Tuples and tuple types are denoted by comma-separated values or types, enclosed in parentheses. We treat the zip/unzip-isomorphic types $[n](\tau_1, \ldots, \tau_k) \cong [n]\tau_1, \ldots, [n]\tau_k$ as interchangeable in any context. Similarly, we treat the curry/uncurry-isomorphic types $[m]([n]\tau) \cong [m \times n]\tau$ as interchangeable. This isomorphic treatment is justified because both streaming and indexed access to either type can be efficiently implemented without explicitly applying the isomorphism and materializing (storing) the result first.

The SOAC semantics enables powerful rewrite rules. For example, mapping an array by a function f followed by

mapping the result with a function g gives the same result as mapping the original array with the composition of f and g:

$$(\text{map } f) \circ (\text{map } g) \equiv \text{map } (f \circ g)$$

Applied from left-to-right and from right-to-left this rule corresponds to producer-consumer (*vertical*) fusion and fission, respectively. *Horizontal* fusion/fission refers to the case when the two **map**s are independent (i.e., not in any producer-consumer relation), as in the equation below:

$$(\text{map } f\ x, \text{map } g\ y) \equiv \text{map } (\lambda(a,b).(f\ a, g\ b))\ (x,y)$$

The rest of this section shows how **map** and **reduce** are special cases of a more general bulk-parallel operator named fold, which (i) can represent (fused) compositions of **map** and **reduce** (and **filter**) operators and, as such, (ii) can itself be decomposed into a **map**-**reduce** composition. Similarly, we introduce the parallel operator sFold, which generalizes Futhark's streaming operators.

Notation. We denote array concatenation by $\#$ and the empty array by ϵ; $\text{inj}(a)$ is the single-element array containing a. A *partitioning* of an array v is a sequence of arrays $v_1, \ldots, v_k$ such that $v_1 \# \ldots \# v_k = v$. Given binary operations f and g, their *product* $f * g$ is defined by componentwise application, i.e., $(f,g)(x) = (f\ x, g\ x)$.

Parallel Operator fold. Many arrays operations are *monoid homomorphisms*, which conceptually allows for splitting an array into two parts, applying the operation recursively, and combining the results using an associative operation $\oplus$. Every monoid homomorphism is uniquely determined by $(\oplus, 0_\oplus)$ and a function g for mapping singleton arrays. The combinator fold thus expresses all such homomorphisms:

$$\begin{aligned}&\text{fold} : (\alpha \to \alpha, \alpha) \to (\beta \to \alpha) \to \Pi n.([n]\beta \to \alpha)\\ &\text{fold}\ (\oplus, 0_\oplus)\ g\ [b_1, \ldots, b_n] = 0_\oplus \oplus (g\ b_1) \oplus \ldots \oplus (g\ b_n)\end{aligned}$$

The fold combinator can decompose previously seen SOACs:

$$\begin{aligned}\text{map } g &= \text{fold}\ (\#, \epsilon)\ (\text{inj} \circ g)\\ \text{reduce}\ (\oplus, 0_\oplus) &= \text{fold}\ (\oplus, 0_\oplus)\ \text{id}\end{aligned}$$

and fold can itself be decomposed by the equation

$$\text{fold}\ (\oplus, 0_\oplus)\ g \ =\ \text{reduce}\ (\oplus, 0_\oplus) \circ \text{map } g.$$

Parallel Operator sFold. A key aspect of Futhark is to *partition* implementations of fold, which partitions a vector into *chunks* before applying the operation on the chunks individually and eventually combining them:

$$\begin{aligned}&\text{sFold} : (\alpha \to \alpha) \to (\Pi m.([m]\beta \to \alpha)) \to \Pi n.[n]\beta \to \alpha\\ &\text{sFold}\ (\oplus)\ f\ (v_1 \# \ldots \# v_k) = (f\ \epsilon) \oplus (f\ v_1) \oplus \ldots \oplus (f\ v_k)\end{aligned}$$

Because a vector can have multiple partitions, sFold is well-defined—it gives the same result for all partitions—if and only if f is itself a fold with $\oplus$ as combining operator. Futhark assumes such properties to hold; they are not checked at run-time, but a programmer responsibility. The streaming combinators permit Futhark to choose freely *any* suitable partition of the input vector. Futhark uses specialized versions of sFold:

$$\begin{aligned}\textbf{stream\_map}\ f &= \text{sFold}\ (\#)\ f\\ \textbf{stream\_red}\ (\oplus)\ f &= \text{sFold}\ ((\oplus) * (\#))\ f\end{aligned}$$

Fusion and fission transformations are based on the universal properties of fold; for example, horizontal (parallel) fusion is expressed by the "banana split theorem" [40], read as a transformation from right to left:

$$\begin{aligned}&\text{fold}\ ((\oplus, 0_\oplus) * (\otimes, 0_\otimes))\ (f,g)\\ &= (\text{fold}\ (\oplus, 0_\oplus)\ f, \text{fold}\ (\otimes, 0_\otimes)\ g)\end{aligned}$$

The map-map rule $\text{map}\ (g \circ f) = \text{map } g \circ \text{map } f$ is the functorial property of arrays; it is used for fusion from right to left and eventually, as a fission rule, from left to right as part of flattening nested parallelism (see Section 5). The flattening rule $\text{map}(\text{map} f) \cong \text{map } f$ eliminates nested parallel maps by mapping the argument function over the product index space (an isomorphism modulo the curry/uncurry isomorphism). Finally, sequential (de)composition of the discussed SOACs can be similarly reasoned in terms of more general iterative operators (known as foldl and sfoldl).

2.2 Futhark Core Language

Futhark is a monomorphic, statically typed, strictly evaluated, purely functional language. The paper uses a subset of Futhark's *core language* (i.e., compiler IR), whose abstract syntax is shown in Figure 1. This is a simplified subset of the full Futhark *source language*. Throughout the paper we write $\overline{z}^{(n)} = z_0, \cdots, z_{(n-1)}$ to range over sequences of n objects of some kind (and write $\overline{z}$ when the size does not matter).[1]

In contrast to the source language, the SOACs of the core language may take as input and result in several arrays.[2] Consider the following example containing nested parallelism:

```
fun main (matrix : [n][m]f32): ([n][m]f32, [n]f32) =
  map (λrow : ([m]f32, f32) →
        let row' = map (λx : f32 → x+1.0) row
        let s = reduce (+) 0 row
        in (row',s))
      matrix
```

[1] This notation is used to shorten bits of program syntax, for instance for the parameters of a function or the array indices. We may also treat such sequences as sets for the purpose of subsetting and set inclusion.

[2] The compiler transforms arrays-of-tuples to tuples-of-arrays [12] at an early compilation stage. It follows that the compiler IR supports tuples only as fully expanded patterns; for example **map** can be seen as implicitly zipping/unzipping its input/result.

$$
\begin{array}{lcll}
t_0 & ::= & \mathbf{t} \mid [\,v\,]\,t_0 & \text{(Constant/array type)} \\
t & ::= & t_0 \mid (t, t) & \text{(Tuple type)} \\
dt & ::= & t \mid \star t & \text{(Nonunique/Unique type)} \\
k & ::= & \mathbf{x} \mid [\,v_1, \ldots, v_n\,] & \text{(Value)} \\
p & ::= & v \mid (v : t) & \text{(Binding name, type optional)} \\
l & ::= & \lambda \overline{p}^{(n)} : dt \ \texttt{->}\ e & \text{(Anonymous Function)} \\
\mathit{fun} & ::= & \textbf{fun}\ v\ \overline{p}{:}dt = e & \text{(Named function)} \\
\mathit{prog} & ::= & \epsilon \mid \mathit{fun}\ \ \mathit{prog} & \text{(Program)}
\end{array}
$$

$$
\begin{array}{lcll}
e & ::= & k \mid v & \text{(Constant/Variable)} \\
 & \mid & (v_1, \ldots, v_n) & \text{(Tuple)} \\
 & \mid & v_1 \odot v_2 & \text{(Apply binary operator)} \\
 & \mid & \textbf{if}\ v_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 & \\
 & \mid & v\,[\,v_1, \ldots, v_n\,] & \text{(Array indexing)} \\
 & \mid & v\ \ v_1\ \ \ldots\ \ v_n & \text{(Function call)} \\
 & \mid & \textbf{let}\ (p_1, \ldots, p_n) = e_1\ \textbf{in}\ e_2 & \text{(Let binding)} \\
 & \mid & v\ \textbf{with}\ [v_1, \ldots, v_n] \leftarrow v & \text{(In-place update)} \\
 & \mid & \textbf{loop}\ (pv)\ \textbf{for}\ v < v\ \textbf{do}\ e & \text{(Loop)} \\
 & \mid & \textbf{iota}\ v & ([0,\ldots,v-1]) \\
 & \mid & \textbf{replicate}\ n\ v & ([v, \ldots, v] \text{ of size } n) \\
 & \mid & \textbf{rearrange}\ (\overline{k})\ v & \text{(Rearrange dimensions)} \\
 & \mid & \textbf{map}\ l\ v_1\ \ldots\ v_n & \\
 & \mid & \textbf{reduce}\ l\ (v_1, \ldots, v_n)\ v'_1\ \ldots\ v'_n & \\
 & \mid & \textbf{scan}\ l\ (v_1, \ldots, v_n)\ v'_1\ \ldots\ v'_n & \\
 & \mid & \textbf{stream\_seq}\ l\ (v_1, \ldots, v_n)\ v'_1\ \ldots\ v'_m & \\
 & \mid & \textbf{stream\_seq}\ l\ v'_1\ \ldots\ v'_m & \\
 & \mid & \textbf{stream\_red}\ l_1\ l_2\ (v_1, \ldots, v_n)\ v'_1\ \ldots\ v'_m &
\end{array}
$$

Figure 1: Core Futhark syntax.

The main function receives a matrix of 32-bit floating point numbers and results in a tuple of (i) a matrix obtained by adding 1.0 to each element of the input matrix, and (ii) a vector obtained by summing up each row of the input matrix.

Every written array type is parametrised with exact shape information, such as in [n][m]**f32**, which denotes a two-dimensional n × m array of 32-bit floats. When used as a parameter type, n and m are bound to the size of the array value. In a return type, the shape information serves as a dynamically checked postcondition. Precise shape information is computed for each bound variable and for each function by a slicing technique based on [30], which also supports existential types for the values whose shapes cannot be computed in advance. In this paper, all sizes are expliclt, and if a function returns an array, then its dimensions must be expressible in terms of the formal parameters. All arrays must be regular, meaning that, all rows of an array must have the same shape. For example, the array [[4],[1,0]] is illegal; when static verification fails—for example because we cannot determine in general whether all iterations of a **map** produce a value of the same shape—dynamic checks (much like bounds checks) are automatically inserted inside the **map**, but they can often be sliced/hoisted out of the **map** itself and checked in advance [29].

Futhark supports **for** (and **while**) loops, which have sequential semantics and are morally equivalent to a simple

```
-- y is free in loop expression
loop (x = a) for i < n do
        g i y x
-- Loop above is equivalent to:
f y 0 n a
```

```
-- Assuming x and y have types
-- t and ty, then f is defined as:
fun f (y:ty) (i:int) (n:int) (x:t) :t
  = if i >= n then x
    else f y (i+1) n (g i y x)
```

Figure 2: Loop as recursive function.

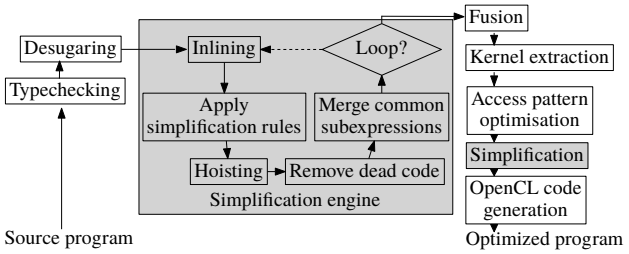

Figure 3: Compiler architecture.

form of tail-recursive function, as illustrated in Figure 2. An array update x **with** i ← v, produces an array identical to x, except that position i contains v.[3] One may use **let** x[i] = v as syntactic sugar for **let** x = x **with** i ← v.

In function declarations, the return and parameter types may have an optional *uniqueness attribute*, which is used for typing in-place updates as detailed in Section 3.

We will occasionally add type annotations to the parameters of anonymous functions to aid readability, and use a more compact notation for anonymous functions; writing e.g. **map** (+1) (**iota** 9) instead of **let** n = 9 **in let** a = **iota** n **in let** one = 1 **in map** (λi → i + one) a

2.3 The Compiler Pipeline

Figure 3 shows the compiler pipeline. On the desugared program, we apply common optimizations to remove simple inefficiencies. Fusion, kernel extraction (flattening), and access pattern optimisations (for obtaining good locality of reference) are discussed in the remainder of the paper. Memory allocations are aggressively hoisted/expanded out of loops and parallel operators prior to code generation.

2.4 Example: K-means Clustering

This section demonstrates several Futhark features, including in-place updates and the **stream_red** SOAC, and shows how they are used to implement efficiently Rodinia's [17] K-means clustering of n d-dimensional points, for arbitrary $k, n, d > 0$. This example refers to the computation of cluster sizes. Assuming $n \gg k$, the sequential implementation shown in Figure 4a does $O(n)$ work: the loop-variant array counts is initialized to a k-element array of zeros, and each iteration of the loop increments the element at position membership[i] by one.

[3] While we restrict ourselves to single-element updates here, our approach readily generalises to bulk updates where an entire range of an array is updated simultaneously.

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

(a) Sequential calculation of counts.

```
let increments =
  map(λ(cluster: int): [k]int →
        let incr = replicate k 0 in let incr[cluster] = 1 in incr)
      membership
let counts =
  reduce (λ(x: [k]int) (y: [k]int): [k]int → map (+) x y)
         (replicate k 0) increments
```

(b) Parallel calculation of counts.

```
let counts = stream_red (map(+))
  (λ(acc: *[k]int) (chunk: [chunksize]int): [k]int →
     let res = loop (acc) for i < chunksize do
                 let cluster = chunk[i]
                 in acc with [cluster] ← acc[cluster]+1
     in res)
  (replicate k 0) membership
```

(c) Efficiently-sequentialized parallel calculation of counts.

Figure 4: Counting cluster sizes in Rodinia's K-means.

A possible parallel implementation is shown in Figure 4b. Each element of membership, denoted cluster is mapped with a function that produces a k-size array, which has a one at position cluster and zeros elsewhere. The result of the mapping is an array of "increments" of type [n][k]**int**, which is reduced with the vectorized addition operator **map**(+), and a k-size zero vector as the neutral element. This solution is fully parallel, but *not* work efficient, as it does $O(n \cdot k)$ work. Unless the hardware can exploit all $n \cdot k$ degrees of parallelism, the work overhead is prohibitive.

We need a language construct that can exposes enough parallelism to take full advantage of the machine, but that will run efficient sequential code within each thread. The **stream_red** SOAC provides just such functionality. As shown in Figure 4c, **stream_red** is given an associative reduction function (**map**(+)), together with a function for processing a chunk of the array. Intuitively, **stream_red**'s partitions the input array membership into an arbitrary number of chunks, for example equal to the degree of hardware parallelism. Chunks are processed in parallel with each other, and the per-chunk results are reduced again with vectorized addition (**map**(+)). Semantically, acc is initialized to a new k-size array of zeros for each chunk, and, as such, its in-place update is guaranteed not to generate data races.

3. In-Place Updates

In a pure language, an array update takes time proportional to the array size. However, if the original array is known not to be used after the update point, an implementation can avoid the array copying and perform the update in-place. Effectively, the update will then only take time proportional to the size of the updated element.

In this section, we present a type system extension that (i) *guarantees* that the cost of an in-place update is proportional to the element size and (ii) preserves referential transparency. For example, the loop in Figure 4a guarantees $O(n)$ rather than $O(nk)$ work. Moreover, in Figure 4c, the array chunk could be (declared unique and) updated in place if desired, because the streaming of membership guarantees disjoint chunks, which ensures that no data races are possible. For simplicity of exposition, we limit loops to have only one variant variable. Similarly, functions return a single value.

3.1 Uniqueness Type Semantics

In-place updates are supported in Futhark through a type-system feature called *uniqueness types*, which is similar to, but simpler than the one of Clean [4, 5], where the primary motivation is modeling IO. Our use is reminiscent of the ownership types of Rust [33]. Alongside a relatively simple (conservative) aliasing analysis in the type checker, this approach is sufficient to determine at compile time whether an in-place modification is safe, and signal an error otherwise. We introduce uniqueness types through the example below, which shows a function declaration:

```
fun modify (n: int) (a: *[n]int) (i: int) (x: [n]int): *[n]int =
  a with [i] ← (a[i] + x[i])
```

A call modify n a i x returns a, but where a[i] has been increased by x[i]. In the parameter declaration a: $*$[n]**int**, the asterisk ($*$) means that modify has been given "ownership" of the array a. The caller of modify will never reference array a after the call. As a consequence, modify can change the element at index i in place, without first copying the array. Further, the result of modify is also unique—the $*$ in the return type declares that the function will not share elements with any of the non-unique parameters (it might share elements with a but not with x). Finally, the call modify n a i x is valid if neither a nor any variable that *aliases* a is used on any execution path following the call to modify.

We say that an array is *consumed* when it is the source of an in-place update or is passed as a unique parameter to a function call; for instance, a is consumed in the expression a **with** [i] ← x. Past the consumption point, neither a nor its aliases may be used again. From an implementation perspective, this contract allows type checking to rely on simple intra-procedural analysis, both in the callee and in the caller, as described in the following sections.

3.2 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 in which the central judgment takes the form $\Sigma \vdash e \Rightarrow \langle \sigma_1, \ldots, \sigma_n \rangle$, which 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. The context Σ maps variables in scope to their aliasing sets.

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

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

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

$$\frac{}{\Sigma \vdash \mathbf{map}(l, \overline{v}^{(n)}) \Rightarrow \langle \overline{\emptyset}^{(n)} \rangle} \quad \text{(ALIAS-MAP)}$$

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

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

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

$$\frac{v \text{ is of rank} > n}{\Sigma \vdash v[\overline{v}^{(n)}] \Rightarrow \langle \{v\} \cup \Sigma(v) \rangle} \quad \text{(ALIAS-SLICEARRAY)}$$

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

$$\frac{}{\Sigma \vdash v_a\ \mathbf{with}\ [\overline{v}^{(n)}] \leftarrow v_v \Rightarrow \langle \Sigma(v_a) \rangle} \quad \text{(ALIAS-UPDATE)}$$

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

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

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

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

Figure 5: Aliasing rules.

The aliasing rules are listed in Figure 5, 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. Alias sets for values produced by SOACs such as **map** are empty. We can imagine the arrays produced as *fresh*, although the compiler is of course free to reuse 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, which fits the implementation intuition.

The most interesting aliasing rules are the ones for function calls (ALIAS-APPLY-*). Since our alias analysis is intra-procedural, we are forced to be conservative. There are two rules, corresponding to functions returning unique and non-unique arrays, respectively. When the result is unique the alias set is empty, otherwise the result conservatively aliases all non-unique parameters.

3.3 In-Place Update Checking

In our implementation, alias computation and in-place update checking is performed at the same time, but is split here for expository purposes. Let aliases(v) the alias set of the variable v. We denote by $\mathcal{O}$ the set of the variables *observed* (used) in expression e, and by $\mathcal{C}$ 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*.

Figure 5 defines a *sequencing* judgment between two occurrence traces, which takes the form $\langle \mathcal{C}_1, \mathcal{O}_1 \rangle \gg \langle \mathcal{C}_2, \mathcal{O}_2 \rangle : \langle \mathcal{C}_3, \mathcal{O}_3 \rangle$ and which 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 because sequencing is sometimes not derivable—for example in 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 sequentialized if and only if no array consumed in the left-hand trace is used in the right-hand trace.

Some of the inference rules for checking if an expression e is functionally safe with respect to in-place updates are shown in Figure 6, where the central judgment is $e \rhd \langle \mathcal{C}, \mathcal{O} \rangle$.

The rule for in-place update v_a **with** $[\overline{v}^{(n)}] \leftarrow v_v$ gives rise to an occurrence trace indicating that we have *observed* v_v and *consumed* v_a. Indices $\overline{v}^{(n)}$ are ignored as they are necessarily scalar variables and cannot be consumed.

Another case is checking the safety of a **map** expression. We do not wish to permit the function 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**. However, the function may consume its parameters, which should be seen as the **map** expression as a whole consuming the corresponding input array. This restriction also preserves the parallel semantics of **map**, because different rows of a matrix can be safely updated in parallel. An example can be seen on Figure 7, which shows an in-place update nested inside an array. To express this restriction, we define an auxiliary judgment $\mathcal{P} \vdash \langle \mathcal{C}_1, \mathcal{O}_1 \rangle \triangle \langle \mathcal{C}_2, \mathcal{O}_2 \rangle$. 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 the variables in the set $\mathcal{P}[v]$ (taking the union of all such replacements), producing $\mathcal{C}_2$. However, if v does not have such a mapping, the judgment is not derivable. The precise inference rules are shown at the bottom of 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

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

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

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

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

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

$$\frac{}{v_a \textbf{ with } [\overline{v}^{(n)}] \leftarrow v_v \rhd \langle \text{aliases}(v_a), \text{aliases}(v_n) \rangle} \quad \text{(SAFE-UPDATE)}$$

$$\frac{e_b \rhd \langle \mathcal{C}, \mathcal{O} \rangle \qquad \overline{p_i \mapsto \text{aliases}(v_i)}^{(n)} \vdash \langle \mathcal{C}, \mathcal{O} \rangle \triangle \langle \mathcal{C}', \mathcal{O}' \rangle}{\textbf{map } (\lambda \overline{p}^{(n)} . \overline{t}^{(m)} \rightarrow e_b) \overline{v}^{(n)} \rhd \langle \mathcal{C}', \mathcal{O}' \rangle} \quad \text{(SAFE-MAP)}$$

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

$$\frac{}{\mathcal{P} \vdash \langle \emptyset, \emptyset \rangle \triangle \langle \emptyset, \emptyset \rangle} \quad \text{(OBSERVE-BASECASE)}$$

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

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

$$\frac{v \in \mathcal{P} \qquad \mathcal{P} \vdash \langle \mathcal{C}, \mathcal{O} \rangle \triangle \langle \mathcal{C}', \mathcal{O}' \rangle}{\mathcal{P} \vdash \langle \{v\} \cup \mathcal{C}, \mathcal{O} \rangle \triangle \langle \mathcal{P}[v] \cup \mathcal{C}', \mathcal{O}' \rangle} \quad \text{(OBSERVE-NONPARAM)}$$

Figure 6: Checking uniqueness and parameter consumption.

```
-- This one is OK and considered to consume 'as'.
let bs = map (λ(a) → a with [0] ← 2) as
let d  = iota m
-- This one is NOT safe, since d is not a formal parameter.
let cs = map (λ(i) → d with [i] ← 2) (iota n)
```

Figure 7: Examples of **map**s with in-place updates.

body consumes only those of the function's parameters that are unique.

4. Streaming SOACs and Their Fusion Rules

Futhark's fusion engine builds previously published work [28, 32]. Semantically, producer-consumer fusion is realized greedily, at all nesting levels, during a bottom-up traversal of the dependency graph, in which SOACs are fused by T2 graph reductions (i.e., a SOAC is fused if it is the source of only one dependency edge and the target is a compatible SOAC). If producer-consumer fusion is not possible, then horizontal fusion is attempted within the same block of **let** expressions. The technique centers on the **redomap** SOAC (generalised to**stream_red** in this paper), which allows repeated composition of **map**, **reduce**, and in some cases **filter** operators.

Notation: n, m, p, q, r integers, a, b, c arrays, f function, α, β, γ types, $\oplus$ associative binop, $\bar{\#}$ vectorized array concatenation, $\overline{[n]\alpha}^{(q)}$ expands to $[n]\alpha_1, \ldots, [n]\alpha_q$, and we informally use $\overline{a[i]}^{(q)} \equiv a_1\,[i], \ldots, a_q\,[i]$ as tuple value.

$$\begin{aligned}
\textbf{map}: \;& \Pi n.(\overline{\alpha}^{(p)} \rightarrow \overline{\beta}^{(q)}) \rightarrow \overline{[n]\alpha}^{(p)} \rightarrow \overline{[n]\beta}^{(q)} \\
\textbf{reduce}: \;& \Pi n.(\overline{\alpha} \rightarrow \overline{\alpha} \rightarrow \overline{\alpha}) \rightarrow \overline{\alpha} \rightarrow \overline{[n]\alpha} \rightarrow \overline{\alpha} \\
\textbf{scan}: \;& \Pi n.(\overline{\alpha} \rightarrow \overline{\alpha} \rightarrow \overline{\alpha}) \rightarrow \overline{\alpha} \rightarrow \overline{[n]\alpha} \rightarrow \overline{[n]\alpha} \\
\textbf{stream_map}: \;& \Pi n.(\Pi m.\overline{[m]\beta}^{(q)} \rightarrow \overline{[m]\gamma}^{(r)}) \rightarrow \overline{[n]\beta}^{(q)} \rightarrow \overline{[n]\gamma}^{(r)} \\
\textbf{stream_map}\; & f\,\overline{b}^{(q)} \equiv \overline{c_1}^{(r)} \bar{\#} \ldots \bar{\#} \overline{c_s}^{(r)} \\
& \text{where } \overline{c_i}^{(r)} = f\,(\overline{b_i}^{(q)}) \\
& \text{for any } s\text{-partitioning of } \overline{b}^{(q)} = \overline{b_1}^{(q)} \bar{\#} \ldots \bar{\#} \overline{b_s}^{(q)} \\
\textbf{stream_red}: \;& \Pi n.(\overline{\alpha}^{(p)} \rightarrow \overline{\alpha}^{(p)} \rightarrow \overline{\alpha}^{(p)}) \rightarrow \\
& (\Pi m.\overline{\alpha}^{(p)} \rightarrow \overline{[m]\beta}^{(q)} \rightarrow (\overline{\alpha}^{(p)}, \overline{[m]\gamma}^{(r)})) \rightarrow \\
& \overline{\alpha}^{(p)} \rightarrow \overline{[n]\beta}^{(q)} \rightarrow (\overline{\alpha}^{(p)}, \overline{[n]\gamma}^{(r)}) \\
\textbf{stream_red}\; & \oplus\; f\,(\overline{e}^{(p)})\,\overline{b}^{(q)} \equiv (\overline{e}^{(p)} \oplus \overline{a_1}^{(p)} \oplus \ldots \oplus \overline{a_s}^{(p)}, \overline{c}^{(r)}) \\
& \text{where } (\overline{a_i}, \overline{c_i}) = f\,\overline{e}\,\overline{b_i} \text{ and } \overline{c} = \overline{c_1} \bar{\#} \ldots \bar{\#} \overline{c_s}, \\
& \text{for any } s\text{-partitioning of } \overline{b}^{(q)} = \overline{b_1}^{(q)} \bar{\#} \ldots \bar{\#} \overline{b_s}^{(q)} \\
\textbf{stream_seq}: \;& \Pi n.(\Pi m.\overline{\alpha}^{(p)} \rightarrow \overline{[m]\beta}^{(q)} \rightarrow (\overline{\alpha}^{(p)}, \overline{[m]\gamma}^{(r)}) \rightarrow \\
& \overline{\alpha}^{(p)} \rightarrow \overline{[n]\beta}^{(q)} \rightarrow (\overline{\alpha}^{(p)}, \overline{[n]\gamma}^{(r)}) \\
\textbf{stream_seq}\; & f\,(\overline{a_0}^{(p)})\,\overline{b}^{(q)} \equiv (\overline{a_s}^{(p)}, \overline{c_1}^{(r)} \bar{\#} \ldots \bar{\#} \overline{c_s}^{(r)}) \\
& \text{where } (\overline{a_i}^{(p)}, \overline{c_i}^{(r)}) = f\,\overline{a_{i-1}}^{(p)}\,\overline{b_i}^{(q)} \\
& \text{for any } s\text{-partitioning of } \overline{b}^{(q)} = \overline{b_1}^{(q)} \bar{\#} \ldots \bar{\#} \overline{b_s}^{(q)}
\end{aligned}$$

Figure 8: Types and semantics of several SOACs.

4.1 Types and Rationale of Streaming SOACs

Figure 8 presents the types and semantics of the streaming SOACs. The **stream_map** SOAC receives an arbitrary number q of input arrays $\overline{b}^{(q)}$ of the same outermost size n and produces an arbitrary number of arrays necessarily of outer size n, by (i) applying its function argument to an arbitrary partitioning of the input (but all b_i are partitioned in the same way) and (ii) concatenating the results. The semantics is that chunks can be processed in parallel, but it is the user who ensures the strong invariant that any partitioning leads to the same result. The **stream_red** SOAC extends **stream_map** by allowing each chunk to produce an additional result, which is then reduced in parallel across the chunks by an associative operator.[4]

[4] The operators passed into **reduce**, **scan**, and **stream_red** are user defined and are assumed to be associative, but the user may specify commu-

$\mathcal{F}1$: **map** f $\overline{b}$ $\implies$ **stream_map** ($\lambda(\overline{b_c}) \to$ **map** f $\overline{b_c}$) $\overline{b}$

$\mathcal{F}2$: **map** f $\overline{b}$ $\implies$ **stream_seq** (λ(a,$\overline{b_c}$) $\to$ (0, **map** f $\overline{b_c}$)) (0) $\overline{b}$

$\mathcal{F}3$: **reduce** $\oplus$ $\overline{e}$ $\overline{b}$ $\implies$
stream_red $\oplus$ ($\lambda(\overline{a},\overline{b_c}) \to \overline{a} \oplus$ **reduce** $\oplus$ $\overline{e}$ $\overline{b_c}$) ($\overline{e}$) $\overline{b}$

$\mathcal{F}4$: **reduce** $\oplus$ $\overline{e}$ $\overline{b}$ $\implies$
stream_seq ($\lambda(\overline{a},\overline{b_c}) \to (\overline{a}) \oplus$ **reduce** $\oplus$ $\overline{e}$ $\overline{b_c}$) ($\overline{e}$) $\overline{b}$

$\mathcal{F}5$: **scan** $\oplus$ $\overline{e}$ $\overline{b}$ $\implies$ **stream_seq**($\lambda(\overline{a},\overline{b_c}) \to$
let $\overline{x_c}$ = **scan** $\oplus$ $\overline{e}$ $\overline{b_c}$
let $\overline{y_c}$ = **map** ($\overline{a} \oplus$) $\overline{x_c}$
in ($\overline{\text{last}(y_c)}$, $\overline{y_c}$)) ($\overline{e}$) $\overline{b}$

$\mathcal{F}6$:

let ($\overline{r_1}$,$\overline{x}$,$\overline{y}$) =
stream_red $\oplus$ f ($\overline{e_1}$) $\overline{a}$
let ($\overline{r_2}$,$\overline{z}$) =
stream_red $\odot$ g ($\overline{e_2}$) $\overline{x}$ $\overline{b}$

$\implies$

let ($\overline{r_1}$, $\overline{r_2}$, $\overline{x}$, $\overline{y}$, $\overline{z}$) = **stream_red**
($\lambda(\overline{c_1}, \overline{d_1}, \overline{c_2}, \overline{d_2}) \to$
($\overline{c_1} \oplus \overline{c_2}$, $\overline{d_1} \odot \overline{d_2}$))
($\lambda(\overline{e_1}, \overline{e_2}, \overline{a_c}, \overline{b_c}) \to$
let ($\overline{r_1}$,$\overline{x_c}$,$\overline{y_c}$) = f $\overline{e_1}$ $\overline{a_c}$
let ($\overline{r_2}$,$\overline{z_c}$) = g $\overline{e_2}$ $\overline{x_c}$ $\overline{b_c}$
in ($\overline{r_1}$, $\overline{r_2}$, $\overline{x_c}$, $\overline{y_c}$, $\overline{z_c}$))
($\overline{e_1}$, $\overline{e_2}$) $\overline{a}$ $\overline{b}$

$\mathcal{F}7$:

let ($\overline{r_1}$,$\overline{x}$,$\overline{y}$) =
stream_seq f ($\overline{e_1}$) $\overline{b}$
let ($\overline{r_2}$,$\overline{z}$) =
stream_seq g ($\overline{e_2}$) $\overline{x}$ $\overline{d}$

$\implies$

let ($\overline{r_1}$, $\overline{r_2}$, $\overline{x}$, $\overline{y}$, $\overline{z}$) = **stream_seq**
($\lambda(\overline{a_1}, \overline{a_2}, \overline{b_c}, \overline{d_c}) \to$
let ($\overline{r_1}$,$\overline{x_c}$,$\overline{y_c}$) = f $\overline{a_1}$ $\overline{b_c}$
let ($\overline{r_2}$,$\overline{z_c}$) = g $\overline{a_2}$ $\overline{x_c}$ $\overline{d_c}$
in ($\overline{r_1}$, $\overline{r_2}$, $\overline{x_c}$, $\overline{y_c}$, $\overline{z_c}$))
($\overline{e_1}$, $\overline{e_2}$) $\overline{b}$ $\overline{d}$

Figure 9: Fusion rules for streaming SOACs.

The rationale for supporting parallel streams in Futhark is to allow a generic encoding of strength-reduction invariants. In essence, the streaming SOACs express all available parallelism, together with an alternative for efficient sequentialization of the excess parallelism. The optimal chunk size is thus the maximal one that still fully occupies hardware. For example, Sobol pseudo-random numbers can be computed by a slower but **map**-parallel formula, or by a cheaper (recurrence) one, but which requires **scan** parallelism [1]. This property can be expressed elegantly with **stream_map**. Each chunk starts by applying the independent formula once, then sequentially applying the cheaper formula. K-means from Section 2.4 is another example.

The **stream_seq** SOAC processes chunks sequentially: the result of processing chunk i becomes the accumulator for processing chunk $i+1$. Note that **stream_seq** permits recovery of all inner parallelism by maximizing chunk size such that **stream_seq** f a $\equiv$ f a, while chunk size equal to 1 typically leads to efficient sequentialization and asymptotically reduced per-thread memory footprint.

4.2 Fusion Rules of Streaming Operators

Figure 9 presents several of the rewrite rules, which assume that fusion is legal between the two SOACs. Rules $\mathcal{F}1$–$\mathcal{F}5$ show that **map**, **reduce**, and **scan** can be straightforwardly converted to parallel and sequential streams. In particular, the **scan**'s translation (only) to **stream_seq** states that if the reduction of all elements up until the current chunk is known to be the accumulator $\overline{a}$, then the current-chunk result can be computed by independently scanning the input chunk and adding to each of its elements the contribution of the previous chunks $\overline{a}$. The last element of the result becomes the accumulator for the next iteration.

Rules $\mathcal{F}6$ and $\mathcal{F}7$ show how to compose two uses of **stream_red** and **stream_seq** under producer-consumer fusion ($\overline{x} \neq \emptyset$) or horizontal fusion ($\overline{x} = \emptyset$), respectively. The calls of function arguments are serialized inside the new function and the input and result accumulator and arrays are merged. In the case of **stream_red**, the associative operators are also tupled to reduce component-wise each of the corresponding results. We remark that, in practice, the input arrays/arguments and results of the fused operator that are not used are removed.

Composition between streama_red and **stream_map** is not shown since it is a subcase of the $\mathcal{F}6$ composition. Finally, to fuse a **map** or **reduce** (or **scan**) with a parallel/sequential stream, one transforms it first to that type of stream and then fuses it by the rules $\mathcal{F}6$ and $\mathcal{F}7$. Notice that fusion between parallel and sequential streams is disallowed because it would result in a sequential stream, which may lose the strength-reduction invariant encoded by the user.

We conclude with two remarks. First, in-place updates are not a burden on the fusion engine; the only significant restriction is not to move a source SOAC past a consumption point of one of its input array, for example, in cases such as **let** x=**map**(f,a) **in let** a[0]=0 **in map**(g,x).

Second, the current fusion engine is based on rewrite rules and not on array index analysis. If an array is indexed explicitly in a target SOAC, then its producer SOAC will not be fused with the target. This is not a severe limitation because good programming style in Futhark is to use parallel SOACs with implicit indexing whenever possible. This is helped by the delayed-semantics of reshaping transformations such as transposition. If implicit indexing is not possible then fusion without duplicating computation is unlikely. In such cases the user can use explicit indexing by applying SOACs over **iota**. Loops should be used as last resort since they have sequential semantics.

4.3 Example of Fusion

Figure 10a shows a simplified example from the Option-Pricing benchmark [1]. The **stream_map** SOAC is used to apply an independent but computationally expensive formula on the first element of the chunk f^{ind}(iss[0]), and to process the rest of the chunk sequentially with a **scan**-based cheaper formula. Figure 10b shows the partial result after fusing the **stream_red** and **reduce** on the outermost level. One can observe that there seems to be a tension be-

tativity as well. Figure 8 also shows the types of **map**, **reduce**, and **scan**; their semantics was introduced in Section 2. Other SOACs are supported (**filter**, scatter), but they are not in the scope of this paper.

```
fun main(n: int): int =
  let Y = stream_map (λ(iss: [m]int): [m]int →
                        let a = f^ind (iss[0])
                        let t = map (g a) iss
                        let y = scan ⊙ 0 t
                        in y)
                     (iota n)
  let b = reduce (+) 0 Y
  in b
```

(a) Program before fusion.

```
fun main(n: int): (int,int) =
  stream_red (+) (λ(e1: int) (iss: [m]int): int →
                    let a = f^ind(iss[0])
                    let t = map (g a) iss
                    let y = scan (⊙) 0 t
                    let b = reduce (+) e1 y
                    in b)
             (0) (iota n)
```

(b) Program after fusion at outer level.

```
fun main(n: int): (int,int) =
  stream_red (+) (λ(e1: int) (iss: [m]int): int →
                    let a = f^ind(iss[0]) in
                    let (tmp, b) =
                      stream_seq (λay (is: [q]int): (int,int) →
                                    let t = map (g a) is
                                    let y'= scan ⊙ 0 t
                                    let y = map (ay⊙) y'
                                    let b = reduce (+) e1 y
                                    in (y[q-1], ab+b))
                                 e1 iss
                    in b)
             (0) (iota n)
```

(c) Program after all-level fusion.

Figure 10: Demonstrating streaming-operator fusion.

tween efficient sequentialization that would require chunk-size maximization and the per-thread memory footprint. The latter refers to the use of **scan**, which cannot be fused in a parallel construct with the **reduce** following it, hence each thread would use memory proportional to the chunk size m. Figure 10c shows the result after the **map**, **scan**, and **reduce** inside **stream_red**'s function have been fused in a **stream_seq**, by the application of rules $\mathcal{F}2$, $\mathcal{F}4$, $\mathcal{F}5$, and $\mathcal{F}7$. The tension has been solved. Indifferent to the outer chunk m, if **stream_seq** is efficiently sequentialized by choosing its chunk size $q = 1$ (and replacing it with a loop) then the thread footprint is $O(1)$. That is, all arrays used in **stream_seq** have size one and can be replaced by scalars held in registers.

5. Flattening and Locality Of Reference

This section presents a transformation that aims to enhance the degree of statically-exploitable parallelism by reorganizing the (imperfectly) nested parallelism into perfect-SOAC nests, in which the outer levels correspond to **map** operators (which are trivial to map to GPUs), and the innermost one is an arbitrary SOAC or scalar code.[5] The resulting perfect nests are then translated to a different IR, resembling GPU kernels; this last step is outside the scope of the paper.

In a purely functional setting, Blelloch's transformation [11] flattens (bottom-up) all available parallelism, while asymptotically preserving the depth and work of the original nested-parallel program. The approach is however arguably inefficient in some cases [7], for example because it does not account for locality of reference.

Our algorithm, presented in Section 5.1, builds on **map**-**loop** interchange and **map** distribution,[6] and attempts to exploit some of the efficient *top-level parallelism*, for example by (i) not seeking the parallelism inside **if** branches, which would require expensive **filter** operations, and by (ii) terminating distribution when it would introduce irregular arrays, which would obscure access patterns and prevent further spatial- and temporal-locality optimizations.

To demonstrate the viability of performing further optimisation on the output of our flattening algorithm, Section 5.2 reports (i) optimization of non-coalesced accesses by transposing the inner, non-parallel array dimensions outwards, and (ii) simple block tiling in fast on-chip memory[7], which is driven by recognizing streamed arrays that are invariant to one of the parallel dimensions of the SOAC nest.

5.1 Flattening Example and Rules

Figures 11a and 11b demonstrate the application of our algorithm on a contrived but illustrative example that demonstrates many of the flattening rules exploited in the generation of efficient code for the various benchmark programs. The original program consists of an outer **map** that encloses (i) another **map** operator implemented as a sequence of **map**s, **reduce**s, and **scan**s and (ii) a loop containing a **map** whose implementation is given by a **reduce** and some scalar computation. As written, only one level of parallelism (e.g., the outermost) can be statically mapped on GPGPU hardware. Our algorithm distributes the outer **map** across the enclosed **map** and **loop** bindings, performs a **map**-**loop** interchange, and continues distribution. The result consists of four perfect nests: a **map**-**map** and **map**-**map**-**map** nest at the outer level, and a **map**-**map**-**reduce** (segmented reduction) and **map**-**map** nest contained inside the loop. In the first **map**-**map** nest, the **scan** and **reduce** are sequentialized because further distribution would generate an irregular array, as the size p of cs is variant to the second **map**.

Figure 12 lists the rules that form the basis of the flattening algorithm. We shall use Σ to denote *map nest contexts*, which are sequences of *map contexts*, written $\mathsf{M}\ \overline{x}\ \overline{y}$,

[5] For example, a **map**-nest ending in a **scan** corresponds to a "segmented" **scan** [10], which is implemented as a **scan** with a modified (associative) operator. Likewise with a nested reduction. Such cases are handled by the compiler, but the details related to these are outside the scope of this paper.

[6] It is always safe to interchange inwards or to distribute a parallel loop [36].

[7] Called *shared memory* in OpenCL and *local memory* in CUDA.

```
let (asss, bss) =
map
  (λps: ([m][m]int,[m]int) →
   let ass =
   map (λp: [m]int →
      let cs =
        scan (+) 0 (iota p)
      let r = reduce (+) 0 cs
      let as = map (+r) ps
      in as)
     ps
   let bs =
   loop (ws=ps) for i < n do
      let ws' =
      map (λas w: int →
         let d = reduce (+) 0 as
         let e = d + w
         let w' = 2 * e
          in w')
        ass ws
      in ws'
   in (ass, bs))
 pss -- pss : [m][m]int
```

(a) Program before distribution.

```
let rss =
map (λps: [m]int →
    map (λp: int →
      let cs = scan (+) 0 (iota p)
      let r = reduce (+) 0 cs in r)
    ps) pss
let asss =
map (λps rs: [m]int →
     map (λint (r) →
       map (+r) ps)
      fs) pss rss
let bss =
loop (wss=pss) for i < n do
    let dss =
    map(λass: [m]int →
       map(λas: int →
          reduce (+) 0 as
       , ass), asss) in
    map (λws, ds: [m]int →
       map (λw d: int →
          let e = d + w in
          let w' = 2 * e in w')
       ws ds) wss dss
```

(b) Program after distribution.

Figure 11: Extracting kernels from a complicated nesting.

where $\overline{x}$ denotes the bound variables of the map operator over the arrays held in $\overline{y}$. The flattening rules, which take the form $\Sigma \vdash e \Rightarrow e'$, specify how a source expression e may be translated into an equivalent target expression e' in the given map nest context Σ. Several rules may be applied in each situation. The particular algorithm used by Futhark bases its decisions on crude heuristics related to the structure of the map nest context and the inner expression. Presently, nested **stream_red**s are sequentialised, while nested **map**s, **scan**s, and **reduce**s are parallelised. These rules were mainly chosen to exercise the code generator, but sequentialising **stream_red** is the right thing to do for most of the data sets we use in Section 6.

For transforming the program, the flattening algorithm is applied (in the empty map nest context) on each map nest in the program. Rule G1 (together with rule G3) allows for manifestation of the map nest context Σ over e. Whereas rule G1 can be applied for any e, the algorithm makes use of this rule only when no other rules apply. Given a map nest context Σ and an instance of a `map` SOAC, rule G2 captures the `map` SOAC in the map nest context. This rule is the only rule that extends the map nest context.

Rule G4 allows for map fission ($\text{map}\ (f \circ g) \Rightarrow \text{map}\ f \circ \text{map}\ g$), in the sense that the map nest context can be materialized first over e_1 and then over e_2 with appropriate additional context to allow for access to the now array-materialized values that were previously referenced through the let-bound variables $\overline{a_0}$. The rule can be applied only if the intermediate arrays formed by the transformation are ensured to be regular, which is enforced by a side condition in the rule. To avoid unnecessary excessive flattening on

$$\frac{\Sigma \vdash \textbf{map}\ (\lambda\, \overline{x}\ \rightarrow\ e)\ \overline{y} \Rightarrow e'}{\Sigma, \textbf{M}\ \overline{x}\ \overline{y} \vdash e \Rightarrow e'} \tag{G1}$$

$$\frac{\Sigma, \textbf{M}\ \overline{x}\ \overline{y} \vdash e \Rightarrow e'}{\Sigma \vdash \textbf{map}\ (\lambda\, \overline{x}\ \rightarrow\ e)\ \overline{y} \Rightarrow e'} \tag{G2}$$

$$\frac{}{\emptyset \vdash e \Rightarrow e} \tag{G3}$$

$$\frac{\begin{array}{c}\Sigma = \textbf{M}\ \overline{x_p}\ \overline{y_p}, \ldots, \textbf{M}\ \overline{x_1}\ \overline{y_1} \\ \Sigma' = \textbf{M}\ (\overline{x_p}, \overline{a_{p-1}})\ (\overline{y_p}, \overline{a_p}), \ldots, \textbf{M}\ (\overline{x_1}, \overline{a_0})\ (\overline{y_1}, \overline{a_1}^{q_1}) \\ \overline{a_p}, \ldots, \overline{a_1} \text{ fresh names} \\ \text{size of each array in } \overline{a_0} \text{ invariant to } \Sigma \\ \Sigma \vdash e_1 \Rightarrow e_1' \quad \Sigma' \vdash e_2 \Rightarrow e_2'\end{array}}{\Sigma \vdash \textbf{let}\ \overline{a_0} = e_1\ \textbf{in}\ e_2 \Rightarrow \textbf{let}\ \overline{a_p} = e_1'\ \textbf{in}\ e_2'} \tag{G4}$$

$$\frac{\begin{array}{c}g = \textbf{reduce}\ (\lambda\, \overline{y}^{2*p}\ \rightarrow\ e)\ \overline{n}^p \\ \Sigma \vdash \textbf{map}\ (g)\ (\textbf{transpose}\ z_0) \ldots (\textbf{transpose}\ z_{p-1}) \Rightarrow e' \\ f = \textbf{map}\ (\lambda\, \overline{y}^{2*p}\ \rightarrow e)\end{array}}{\Sigma \vdash \textbf{reduce}\ (f)\ (\overline{\textbf{replicate}\ k\ n}^p)\ \overline{z}^p \Rightarrow e'} \tag{G5}$$

$$\frac{\Sigma \vdash \textbf{rearrange}\ (0, 1+k_0, \ldots, 1+k_{n-1})\ y \Rightarrow e}{\Sigma, \textbf{M}\ x\ y \vdash \textbf{rearrange}\ \overline{k}^n\ x \Rightarrow e} \tag{G6}$$

$$\frac{\begin{array}{c}\Sigma' = \Sigma, \textbf{M}\ (\overline{x}, \overline{y})\ (\overline{xs}, \overline{ys}) \qquad (\{n\} \cup \overline{q}) \cap (\overline{x}, \overline{y}) = \emptyset \\ m = \text{outer size of each of } \overline{xs} \text{ and } \overline{ys} \\ \text{f contains exploitable (regular) inner parallelism} \\ \Sigma \vdash \textbf{loop}\ (\overline{zs'} = \overline{\textbf{replicate}\ m\ z_i}, \overline{ys'} = \overline{ys}) \\ \qquad \textbf{for}\ i < n\ \textbf{do}\ \textbf{map}(f\ i\ \overline{q})\ \overline{xs}\ \overline{ys}\ \overline{ys'}\ \overline{zs'} \Rightarrow e\end{array}}{\Sigma' \vdash \ \textbf{loop}\ (\overline{z'} = \overline{z},\ \overline{y'} = \overline{y}) = \\ \qquad \textbf{for}\ i < n\ \textbf{do}\ f\ i\ \overline{q}\ \overline{x}\ \overline{y}\ \overline{y'}\ \overline{z} \Rightarrow e} \tag{G7}$$

Figure 12: Flattening rules.

scalar computations, the **let**-expressions are rearranged using a combination of **let**-floating [43] and tupling for grouping together scalar code in a single **let**-construct. In essence, inner SOACs are natural splitting points for fission. For example, `let b = x+1 in let a = b+2 in replicate n a` is split as `let a=e1 in e2`, where `e2` is `replicate n a` and `e1` is `let b = x+1 in b+2`.

Rule G5 allows for **reduce-map** interchange where it is assumed that the source neutral reduction element is a replicated value. The original pattern appears in K-means (see Figure 4c) as a reduction with a vectorized operator, which is inefficient if executed as such. The interchange results in a segmented-reduce operator (applied on equally-sized segments), at the expense of transposing the input array(s). This rule demonstrates a transformation of both schedule and (if the transposition is manifested) data of the program being optimized.

Rule G6 allows for distributing a **rearrange** construct by rearranging the outer array (input to the map nest) with

an expanded permutation. The semantics of **rearrange** p a is that it returns a with its dimensions reordered by a statically-given permutation p. For instance, the expression **rearrange** (2,1,0) a reverses the dimensions of the three-dimensional array a. For convenience, **transpose** a is syntactic sugar for **rearrange**(1,0,...) a, which swaps the two outermost dimensions. Similar rules can be added to handle other expressions that have a particularly efficient formulation when distributed on their own, such as concatenation (not covered in this paper).

Finally, rule G7 implements a **map**-**loop** interchange. The simple intuition is that

map (λx $\rightarrow$ **loop** (x'=x) **for** i < n **do** (f x')) xs

is equivalent to

loop (xs'=xs) **for** i < n **do** (**map** f xs')

because they both produce [f^n(xs[0]), ... ,f^n(xs[m-1])]. The rule is sufficiently general to deal with all variations of variant and invariant variables in the **loop** body. The side condition in the rule ensures that $z \subseteq q$ are free variables and thus invariant to Σ. The rule is applied only if the body of the loop contains inner parallelism, such as maps, otherwise its application is not beneficial (e.g., it would change the Mandelbrot benchmark from Section 6 to have a memory- rather than a compute-bound behavior). However, rule G7 is essential for efficient execution of the LocVolCalib benchmark, because a loop separates the outer map from four inner maps.

We conclude by remarking that some of the choices made in the flattening rewrite rules about how much parallelism to exploit and how much to sequentialize efficiently are arbitrary, because there is no size that fits all. For example, we currently sequentialize a **stream_red** if it is inside a **map** nest, but the algorithm can easily be made more aggressive. A more general solution would be to generate all possible code versions, and to discriminate between them at runtime based on static predicates that test whether the exploited parallelism is enough to fully utilize hardware. Work is in progress in this direction.

5.2 Optimizing Locality of Reference

Ensuring coalesced accesses to global memory is critical for GPU performance. Several of the benchmarks discussed in Section 6, such as FinPar's LocVolCalib, Accelerate's N-body, and Rodinia's CFD, K-means, Myocyte, and LavaMD, exhibit kernels in which one or several innermost dimensions of the mapped arrays are processed sequentially inside the kernel. In the context of our flattening algorithm, this typically corresponds to the case where rule G1 has been applied with e being a SOAC. This nested SOAC is transformed to a **stream_seq** using the rules of Figure 9; removing parallelism, but retaining access pattern information.

A naive translation of a nested **stream_seq** would lead to consecutive threads accessing global memory with a stride equal to the size of the inner (non-parallel) array dimensions, which may generate one-order-of-magnitude slowdowns. The Futhark compiler solves this by, intuitively, transposing the non-parallel dimensions of the array innermost, and the same for the result and all temporary arrays created inside the kernel.[8] This approach is guaranteed to resolve coalescing if the sequential-dimension indices are invariant to the parallel array dimensions. For example, consider this expression: **map** (λxs $\rightarrow$ **reduce** (+) 0 xs) xss.
Assuming the inner reduction is implemented sequentially, the expression is optimized by changing the representation of xss to be column major (the default is row major), via transposition in memory, as follows:

```
let xss' = as_column_major xss
in map (λxs → reduce (+) 0 xs) xss'
```

The type of xss′ is the same as that of xss. The approach generalizes to higher rank arrays by using a variant of **rearrange** that only changes the representation, not the type. The representation of an array is recorded as a symbolic composition of affine transformations, which needs to be examined to determine in what way the source array should be transposed to achieve coalesced accesses.

The compiler also performs simple block *tiling* of parallel dimensions, which is driven by recognizing arrays that are used as input to **stream_seq** constructs *and* are invariant to one of the parallel dimensions. For example, the code

```
map (λp → stream_seq (λa (ps': [q]int) →
                        g a ps') ps) ps
```

exhibits such an optimization opportunity for the streamed array ps, and is transformed into:

```
map (λp → stream_seq (λa (ps': [q]int) →
                        let ps'' = local ps'
                        in g a ps'') (0) ps) ps
```

where ps'' is a fast memory[9] array created by collective copying (**local**), and used instead of ps' in g. The size q is determined (at runtime) such that the array ps'' will fit in the fast memory. This example essentially illustrates the structure of the N-body benchmark discussed in Section 6. Futhark also supports two-dimensional tiling where two streamed arrays are invariant to different parallel dimensions (e.g., as in matrix-matrix multiplication). LavaMD exhibits an interesting tiling pattern that we support, in which the to-be-tiled array is the result of an indirect index, such as

```
map(λi → map (λj →
  j + stream_seq g (0) (xs[f i])) js) is
```

but two levels deep in the kernel code.

[8] The common case corresponds to directly mapped arrays, but we also perform simple index analysis to support common explicitly-indexed accesses.

[9] Called *local memory* in OpenCL, and *shared memory* in CUDA.

6. Evaluation on Sixteen Benchmarks

We have manually translated programs from the Rodinia [17], FinPar [1], Parboil [50], and Accelerate [15] benchmark suites to Futhark.[10] The three former consist of hand-written OpenCL programs, while Accelerate is an established Haskell DSL for GPU computation. The translation of Accelerate programs is straightforward; for Rodinia and FinPar, parallel loops were translated to bulk-parallel operators such as **map** and **reduce**, while preserving the original code structure as much as possible. We have focused on the shorter Rodinia benchmarks, while FinPar contains more challenging programs. We invoke the Futhark compiler with no benchmark-specific flags — no tuning is done, and the exact same code is executed on both of our test systems. The compiler inserts instrumentation that records total runtime minus the time taken for (i) loading program input onto the GPU, (ii) reading *final* results back from the GPU, and (iii) OpenCL context creation and kernel build time. (Excluding these overheads emphasizes the performance differences.) Any other host-device communication/copying is measured. Rodinia benchmarks have been modified to time similarly, while Accelerate and FinPar already did so. When in doubt, we have erred in favor of non-Futhark implementations. We use Accelerate version 0.15.1, with the CUDA backend. Runtimes were measured through the built-in `--benchmark` command line option.

Figure 13 and Table 1 show the speedups and the runtimes averaged on ten runs. Two systems were used, namely an NVIDIA GeForce GTX 780 Ti with CUDA 8.0 (blue) and an AMD FirePro W8100 (orange). Only benchmarks with OpenCL implementations were executed on the latter.

6.1 Discussion of Benchmark Results

As the Futhark compiler generates OpenCL, all Rodinia benchmarks are the OpenCL versions, except where noted. The used datasets are presented in Table 2.

Most of our slowdown is related to generic issues of unnecessary copying and missing micro-optimization that are common to compilers for high-level languages; exceptions noted below. The speedup on Backprop seems related to a reduction that Rodinia has left sequential. Running time of the training phase is roughly equal in Rodinia and Futhark ($\sim 10\ ms$). Rodinia's implementation of HotSpot uses time tiling [26], which seems to pay off on the NVIDIA GPU, but not on AMD. Futhark's slowdown is due to double buffering via copy rather than pointer switching, accounting for 30% of runtime. Our speedup on K-means is due to Rodinia not parallelizing computation of the new cluster centers, which is a segmented reduction. Myocyte's dataset has been expanded because its degree of parallelism was one (`workload`=1), but we have done so in the CUDA version. We attribute our speedup to automatic coalescing optimizations, which is tedious to do by hand on such large programs.

NN speedup is due to Rodinia leaving 100 **reduce** operations for finding the nearest neighbors sequential on the CPU. Possibly because the reduce operator is atypical; it computes both the minimal value and the corresponding index. Speedup is less impressive on the AMD GPU, due to higher kernel launch overhead—this benchmark is dominated by frequent launches of short kernels.

For Pathfinder, Rodinia uses time tiling, which, unlike HotSpot, does not seem to pay off on the tested hardware. The four benchmarks Crystal, Fluid, Mandelbrot, and N-body are from Accelerate. The N-body simulation comprises

	NVIDIA GTX780		AMD W8100	
Benchmark	**Ref.**	**Futhark**	**Ref.**	**Futhark**
Backprop	46.9	20.7	41.5	12.9
CFD	1878.2	2235.9	3610.0	4177.5
HotSpot	35.9	45.3	260.4	72.6
K-means	1597.7	572.2	1216.1	1534.9
LavaMD	5.1	6.7	9.0	7.1
Myocyte	2733.6	555.4	—	2979.8
NN	178.9	11.0	193.2	37.6
Pathfinder	18.4	7.4	18.2	6.5
SRAD	19.9	16.1	195.1	34.8
LocVolCalib	1211.1	1293.2	3117.0	5015.8
OptionPricing	136.0	106.8	429.5	360.8
MRI-Q	20.2	15.5	17.9	14.3
Crystal	41.0	8.4	—	8.4
Fluid	268.7	100.4	—	221.8
Mandelbrot	30.8	8.1	—	14.8
N-body	613.2	89.5	—	269.8

Table 1: Average benchmark runtimes in milliseconds.

Benchmark	**Dataset**
Backprop	Input layer size equal to 2^{20}
CFD	`fvcorr.domn.193K`
HotSpot	1024×1024; 360 iterations
K-means	`kdd_cup`
LavaMD	`boxes1d=10`
Myocyte	`workload=65536`, `xmax=3`
NN	Default Rodinia dataset duplicated 20 times
Pathfinder	Array of size 10^5
SRAD	502×458; 100 iterations
LocVolCalib	large dataset
OptionPricing	large dataset
MRI-Q	large dataset
Crystal	Size 2000, degree 50
Fluid	3000×3000; 20 iterations
Mandelbrot	4000×4000; 255 limit
N-body	$N = 10^5$

Table 2: Benchmark dataset configurations.

[10] We have selected the MRI-Q benchmark from Parboil mainly to demonstrate tiling. We have selected the four (out of 15 available) Accelerate examples that were best suited as self-contained benchmarks. We have selected two out of three FinPar benchmarks, because the third one contains limited (irregular) parallelism. Finally, we have selected the 9 (out of the 21 available) Rodinia benchmarks that (i) looked most friendly from a data-parallel perspective, and (ii) were expressible in terms of a nested composition of **map**, **reduce**, **scan**, **stream_seq**, **stream_red**, **stream_map**.

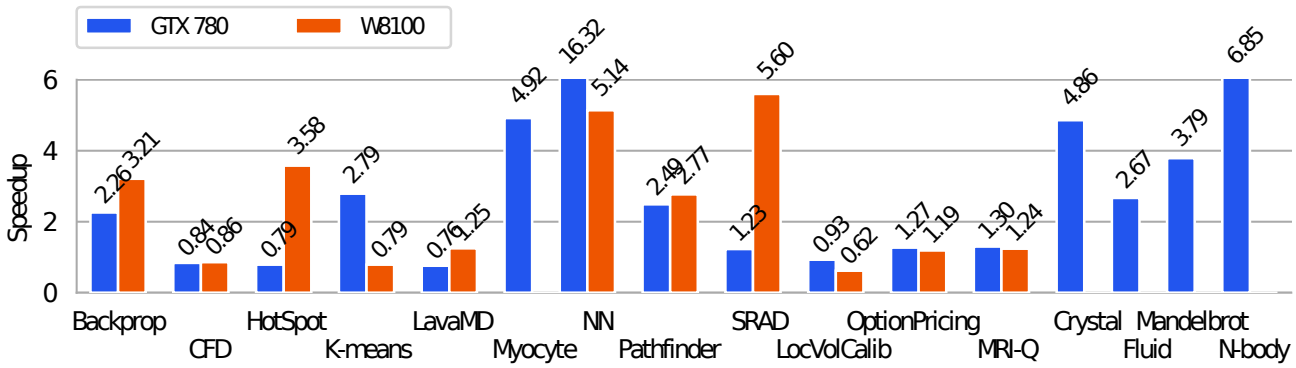

Figure 13: Relative speedup compared to reference implementations (some bars are truncated for space reasons).

a width-N map where each element performs a fold over each of the N bodies.

OptionPricing from FinPar is essentially a **map**-**reduce**-composition. The benchmark primarily measures how well the Futhark compiler sequentialises excess parallelism inside the complex **map** function. LocVolCalib from FinPar is 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. The slowdown on the AMD GPU is due to transpositions, inserted to fix coalescing, being relatively slower than on the NVIDIA GPU.

6.1.1 Impact of Optimisations

Impact was measured by turning individual optimisations off and re-running benchmarks on the NVIDIA GPU. We report only where the impact is non-neglible.

Fusion has an impact on K-means ($\times 1.42$), LavaMD ($\times 4.55$), Myocyte ($\times 1.66$), SRAD ($\times 1.21$), Crystal ($\times 10.1$), and LocVolCalib ($\times 9.4$). Without fusion, OptionPricing, N-body, and MRI-Q fail due to increased storage requirements.

In the absence of in-place updates, we would have to implement K-means as on Figure 4b—the resulting program is slower by $\times 8.3$. Likewise, LocVolCalib would have to implement its central tridag procedure via a less efficient **scan**-**map** composition, causing a $\times 1.7$ slowdown. OptionPricing uses an inherently sequential Brownian Bridge computation that is not expressible without in-place updates.

The coalescing transformation has an impact on K-means ($\times 9.26$), Myocyte ($\times 4.2$), OptionPricing ($\times 8.79$), and LocVolCalib ($\times 8.4$). Loop tiling has an impact on LavaMD ($\times 1.35$), MRI-Q ($\times 1.33$), and N-body ($\times 2.29$).

7. Related Work

Futhark builds on previous work in type systems and parallel compilation techniques. More elaborate and powerful uniqueness [4, 5], linear [24] and affine [54] type systems than Futhark's exist. To our knowledge, these techniques have not previously been used for performance-oriented cost guarantees in a parallel language. Delite [51] and Lime [2] are impure languages that use effect systems to ensure that side effects cannot inhibit safe parallelisation, although Delite does permit a potentially unsafe parallel foreach. Previous work [37] presents a data structure for functional arrays with asymptotically-efficient in-place updates and parallel semantics, but without requiring a type system extension. The cost guarantees assume a task-parallel (fork/join) rather than a data-parallel setting. The implementation requires underlying mutable arrays, and is based on run-time structures to handle the change-logs of interior arrays (potentially involving allocation), which is not straightforward to compile to efficient GPU code. In comparison, uniqueness types are checked statically, and the code generated for an in-place update is a simple memory write.

There is a rich body of literature on embedded array languages and libraries targetting GPUs. Imperative solutions include Copperhead [14], Accelerator [53], and deep-learning DSLs, such as Theano [6] and Torch [20].

Purely functional languages include Accelerate [39], Obsidian [18], and NOVA [19]. More recent work [48] shows that stochastic combinations of rewrite rules opens the door to autotuning. These languages support neither arbitrary nested parallelism, nor explicit indexing and efficient sequential code inside their parallel constructs. Work has been done on designing purely functional representations for OpenCL kernels [49], which could in principle be targeted by our flattening algorithm. The Futhark compiler presently uses a similar (but simpler) representation, the details of which are outside the scope of this paper.

A number of dataflow languages aim at efficient GPU compilation. StreamIt supports a number of static optimizations on various hardware, for example, GPU optimizations [34] include memory-layout selection (shared/global memory), resolving shared-memory bank conflicts, increasing the granularity of parallelism by vertical fusion, and untilizing unused registers by software prefetching and loop unrolling, while multicore optimizations [25] are aimed at finding the right mix of task, data and pipeline parallelism.

Halide [46] uses a stochastic approach for finding optimal schedules for fusing stencils by a combination of tiling, sliding window and work replication. This is complementary to Futhark, which does not optimise stencils, nor uses autotunning techniques, but could benefit from both. In comparison, Futhark supports arbitrary nested parallelism and flattening transformation, together with streaming SOACs that generically encode strength-reduction invariants.

Delite uses rewrite rules to optimize locality in NUMA settings [13] and proposes techniques [38] for handling simple cases of nested parallelism on GPUs by mapping inner parallelism to CUDA block and warp level. We use transposition to handle coalescing, and, to our knowledge, no other compiler matches our AST-structural approach to kernel extraction, except for those that employ full flattening, such as NESL [7, 12], which often introduces inefficiencies and does not support in-place updates. Data-only flattening [8] shows how to convert from nested to flat representation of *data*, without affecting program structure. This would be a required step in extending Futhark to exploit irregular parallelism. In comparison, Futhark flattens some of the top-level parallelism *control*, while preserving the inner structure and opportunities for locality-of-reference optimizations.

Imperative GPU compilation techniques rely on low-level index analysis ranging from pattern-matching heuristics [22, 56] to general modeling of affine transformations by polyhedral analysis [44, 55]. Since such analyses often fight the language, solutions rely on user annotations to improve accuracy. For example, OpenMP annotations can be used to enable transformations of otherwise unanalyzable patterns [16], while PENCIL [3] provides a restricted C99-like low-level language that allows the (expert) user to express the parallelism of loops and provide additional information about memory access patterns and dependencies. X10 demonstrates an elegant integration of GPUs into a PGAS language [21], but does not by itself provide an abstraction over hardware-specific limitations and performance characteristics. The programmer is responsible for ensuring coalesced memory access, taking advantage of the memory hierarchy, and so forth.

In comparison, Futhark relies on higher-order reasoning and also transforms data (transposition,AoS-SoA), not only (affine) schedules as most imperative work does. For example, in an imperative setting, it is difficult to recognize **scan** and **filter** implemented as loops [42], or to encode strength-reduction invariants and achieve the fusion result of Figure 10c.

Our flattening transformation resembles the tree-of-bands construction [55] in that it semantically builds on interchange and distribution, but we use higher-order rules. For example, imperative approaches would implement a reduction with a vectorized operator via a histogram-like computation [47], which is efficient only when the histogram size is small. In comparison, rule G5 in Figure 12 transforms a reduction with a vectorized operator to a (regular) segmented reduction, which always has an efficient implementation.

In comparison to Futhark, imperative analyses [55, 56] are superior at performing all kinds of tiling, for example hexagonal time tilling [26] and achieving memory coalescing by semantically transposing arrays on the fly (via tiling). However, non affine accesses may still restrict applicability: for example indirect-array accesses would prevent them from optimising memory coalescing for the Option-Pricing benchmark, where Futhark's simpler, transposition-based approach succeeds.

Finally, we note that Futhark is not intended as a general-purpose language, but rather it is aimed at (i) expressing computational kernels, which can then be linked with applications written in mainstream languages, and, at (ii) being used as a code-generation target for high-level DSLs. Such a usage was demonstrated by an experiment [23, 31] in which a subset of APL was compiled to Futhark, executed on GPU, and used from Python for visualization purposes.

8. Conclusions

We have presented a fully automatic optimizing compiler for Futhark, a pure functional array language. We have demonstrated (i) how to support in-place updates in Futhark's type system, (ii) how second-order array combinators can express symbolically both all available parallelism and efficient sequentialization alternatives, and (iii) how to fuse the program aggressively and then how to decompose it yet again into kernels in a manner that can improve the amount of efficient (common-case) parallelism.

We have validated our approach on 16 benchmark programs, with performance compared to reference implementations ranging from $\times 0.6$ slowdown to $\times 16$ speedup, with competitive performance on average. Our results show that while the ease of high-level structural transformation permitted by a functional language is powerful, attention must still be paid to low-level issues such as memory access patterns.

Acknowledgments

The work on Oclgrind [45] has proven invaluable in the development of the Futhark compiler.

This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center 'HIPERFIT: Functional High Performance Computing for Financial Information Technology' (`http://hiperfit.dk`) under contract number 10-092299.

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. *ACM Trans. Archit. Code Optim. (TACO)*, 13(2):18:1–18:27, June 2016. ISSN 1544-3566.

[2] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In *Procs. of ACM Int. Conf. on Object Oriented Prog. Systems Languages and Applications*, OOPSLA '10, 2010. ACM.

[3] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *Procs of Int. Conf. on Parallel Architecture and Compilation (PACT)*, PACT '15, 2015. IEEE Computer Society.

[4] 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*, 1993.

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

[6] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: A CPU and GPU Math Compiler in Python. In S. van der Walt and J. Millman, editors, *Procs. of the 9th Python in Science Conference*, 2010.

[7] L. Bergstrom and J. Reppy. Nested Data-parallelism on the GPU. In *Procs of 17th ACM SIGPLAN Int. Conf. on Functional Prog.*, ICFP'12, 2012. ACM.

[8] L. Bergstrom, M. Fluet, M. Rainey, J. Reppy, S. Rosen, and A. Shaw. Data-only Flattening for Nested Data Parallelism. In *Procs. of the 18th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP '13, 2013. ACM.

[9] R. S. Bird. Algebraic Identities for Program Calculation. *Computer Journal*, 32(2):122–126, 1989.

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

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

[12] 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.

[13] K. J. Brown, H. Lee, T. Rompf, A. K. Sujeeth, C. De Sa, C. Aberger, and K. Olukotun. Have Abstraction and Eat Performance, Too: Optimized Heterogeneous Computing with Parallel Patterns. In *Procs. of Int. Symp. on Code Generation and Optimization*, CGO 2016, 2016. ACM.

[14] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an Embedded Data Parallel Language. In *Procs. of ACM Symp. on Principles and Practice of Parallel Programming*, PPoPP '11, 2011. ACM.

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

[16] P. Chatarasi, J. Shirako, and V. Sarkar. Polyhedral Optimizations of Explicitly Parallel Programs. In *Procs. of Int. Conf. on Parallel Architecture and Compilation (PACT)*. IEEE, 2015.

[17] 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 *Procs. of IEEE Int. Symp. on Workload Characterization (IISWC)*, Oct 2009.

[18] K. Claessen, M. Sheeran, and B. J. Svensson. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Procs. of Workshop on Declarative Aspects of Multicore Programming (DAMP)*. ACM, 2012.

[19] A. Collins, D. Grewe, V. Grover, S. Lee, and A. Susnea. NOVA: A Functional Language for Data Parallelism. In *Procs. of Int. Workshop on Libraries, Languages, and Compilers for Array Prog.*, ARRAY'14, 2014. ACM.

[20] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, Neural Information Processing Systems*, 2011.

[21] D. Cunningham, R. Bordawekar, and V. Saraswat. GPU Programming in a High Level Language: Compiling X10 to CUDA. In *Procs. of the ACM SIGPLAN X10 Workshop*, X10 '11, 2011. ACM.

[22] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a High-level Language for GPUs: (via Language Support for Architectures and Compilers). In *Procs. of ACM SIGPLAN Int. Conf. on Programming Language Design and Implementation*, PLDI'12, 2012. ACM.

[23] 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.

[24] M. Fahndrich and R. DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Procs. of Int. Conf. on Programming Language Design and Implementation*, PLDI '02, 2002. ACM.

[25] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Procs. of Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, 2006. ACM.

[26] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid Hexagonal/Classical Tiling for GPUs. In *Procs. Int. Symp. on Code Generation and Optimization*, CGO '14. ACM, 2014.

[27] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Interprocedural Parallelization Analysis in SUIF. *Trans. on Prog. Lang. and Sys. (TOPLAS)*, 27(4):662–731, 2005.

[28] T. Henriksen and C. E. Oancea. A T2 Graph-reduction Approach to Fusion. In *Procs. of the 2nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, 2013. ACM.

[29] T. Henriksen and C. E. Oancea. Bounds Checking: An Instance of Hybrid Analysis. In *Procs. of ACM SIGPLAN Int. Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, 2014. ACM.

[30] T. Henriksen, M. Elsman, and C. E. Oancea. Size Slicing: A Hybrid Approach to Size Inference in Futhark. In *Procs. of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC'14, 2014. ACM.

[31] T. Henriksen, M. Dybdal, H. Urms, A. S. Kiehn, D. Gavin, H. Abelskov, M. Elsman, and C. Oancea. APL on GPUs: A TAIL from the Past, Scribbled in Futhark. In *Procs. of the 5th Int. Workshop on Functional High-Performance Computing*, FHPC'16, 2016. ACM.

[32] T. Henriksen, K. F. Larsen, and C. E. Oancea. Design and GPGPU Performance of Futhark's Redomap Construct. In *Procs. of the 3rd ACM SIGPLAN Int. Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'16, 2016. ACM.

[33] G. Hoare. The Rust Programming Language, June 2013.

[34] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: Portable Stream Programming on Graphics Engines. In *Procs. of Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, 2011. ACM.

[35] K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar. Compiling and Optimizing Java 8 Programs for GPU Execution. In *Procs. of Int. Conf. on Parallel Architecture and Compilation*, PACT '15, 2015. IEEE Computer Society.

[36] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[37] A. Kumar, G. E. Blelloch, and R. Harper. Parallel Functional Arrays. In *Procs. of the 44th ACM SIGPLAN Symp. on Principles of Programming Languages*, POPL'17, 2017. ACM.

[38] H. Lee, K. J. Brown, A. K. Sujeeth, T. Rompf, and K. Olukotun. Locality-Aware Mapping of Nested Parallel Patterns on GPUs. In *Procs. of the 47th Annual IEEE/ACM Int. Symp. on Microarchitecture*, MICRO-47, 2014. IEEE Computer Society.

[39] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising Purely Functional GPU Programs. In *Procs. of the ACM SIGPLAN Int. Conf. on Functional Programming*, ICFP '13, 2013. ACM.

[40] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proc. 5th ACM Conf. on Functional Programming Languages and Computer Architecture (FPCA)*, 1991.

[41] C. E. Oancea and L. Rauchwerger. Logical Inference Techniques for Loop Parallelization. In *Procs. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI'12, 2012. ACM.

[42] C. E. Oancea and L. Rauchwerger. Scalable Conditional Induction Variables (CIV) Analysis. In *Procs. of the 13th IEEE/ACM Int. Symp. on Code Generation and Optimization*, CGO'15, 2015. IEEE Computer Society.

[43] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: Moving Bindings to Give Faster Programs. In *Procs. of the First ACM SIGPLAN Int. Conf. on Functional Programming*, ICFP'96, 1996. ACM.

[44] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop Transformations: Convexity, Pruning and Optimization. In *Procs. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL'11, 2011. ACM.

[45] J. Price and S. McIntosh-Smith. Oclgrind: An extensible OpenCL device simulator. In *Procs. of the 3rd Int. Workshop on OpenCL*. ACM, 2015.

[46] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Procs. of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI'13, 2013. ACM.

[47] C. Reddy, M. Kruse, and A. Cohen. Reduction Drawing: Language Constructs and Polyhedral Compilation for Reductions on GPU. In *Procs. of Int. Conf. on Parallel Architectures and Compilation*, PACT'16, 2016. ACM.

[48] 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. In *Procs. of the ACM SIGPLAN Int. Conf. on Functional Programming*, ICFP'15, 2015.

[49] M. Steuwer, T. Remmelg, and C. Dubach. Lift: A Functional Data-parallel IR for High-performance GPU Code Generation. In *Procs. of Int. Symp. on Code Generation and Optimization*, CGO'17, 2017. IEEE Press.

[50] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[51] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, Apr. 2014. ISSN 1539-9087.

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

[53] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. Technical report, October 2006.

[54] J. A. Tov and R. Pucella. Practical Affine Types. In *Procs. of the ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL'11, 2011. ACM.

[55] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim. (TACO)*, 9(4):54:1–54:23, Jan. 2013. ISSN 1544-3566.

[56] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *Procs. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI'10, 2010. ACM.