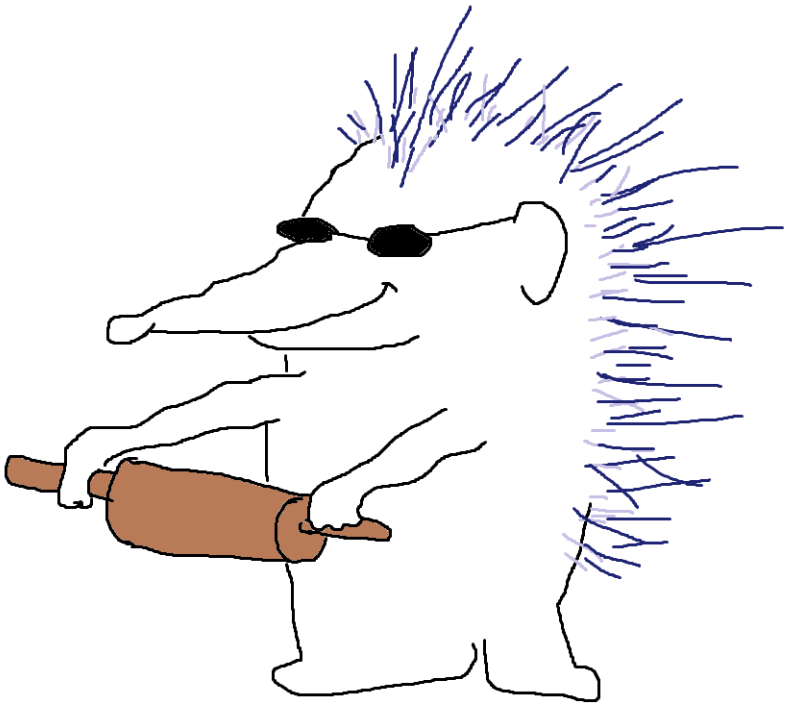
UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE



BSc Thesis

Flattening Irregular Nested Parallelism in Futhark

Cornelius Sevald-Krause
<lgx292@alumni.ku.dk>

Supervised by Troels Henrikesen
Department of Computer Science

June 12, 2023

Abstract

In this report we describe our work on flattening irregular nested parallelism in the data parallel language Futhark. We present a new flattening rule for flattening `match`-expressions as a generalization of flattening `if`-expressions and implement the flattening transformation in the Futhark compiler. We benchmark this solution against one using nested `if`-expressions and find that the dedicated `match`-transformation runs modestly faster than the nested `if` version.

We also detail function lifting, a technique for flattening function calls, along with an implementation in the Futhark compiler as a part of a continued effort of achieving full irregular flattening in the Futhark compiler.

Contents

1. Introduction **5**

2. Background **6**
- 2.1. Nested Data-Parallelism . . . 6
- 2.2. Futhark . . . 6
  - 2.2.1. Syntax . . . 6
  - 2.2.2. Types . . . 7
  - 2.2.3. Second-Order Array Combinators . . . 8

3. Irregular Flattening **11**
- 3.1. Representing Irregular Arrays . . . 11
- 3.2. Segmented Operations . . . 13
  - 3.2.1. Segmented Scan . . . 13
  - 3.2.2. Segmented Reduce . . . 14
  - 3.2.3. Segmented Iota . . . 15
  - 3.2.4. Constructing the Auxiliary Arrays . . . 15
- 3.3. Flattening . . . 15
  - 3.3.1. Flattening scan inside map . . . 17
  - 3.3.2. Flattening reduce inside map . . . 17
  - 3.3.3. Flattening iota inside map . . . 17
  - 3.3.4. Flattening map inside map . . . 18
  - 3.3.5. Flattening if-then-else inside map . . . 18
  - 3.3.6. Flattening match-expressions inside map . . . 19
- 3.4. Function Lifting . . . 22

4. Implementation **26**

5. Evaluation **31**
- 5.1. Tests . . . 31
  - 5.1.1. Testing Function Lifting . . . 31
  - 5.1.2. Testing Flattening Match-Expressions . . . 32
- 5.2. Benchmarks . . . 33

6. Conclusion **36**

7. Future Work **37**
- 7.1. Improving PartitionK . . . 37
- 7.2. Incremental Flattening . . . 37

7.3. The Replication Problem 38

A. Code Listings **39**

Bibliography **41**

1. Introduction

Since the dawn of computing, the computing power of processors have increased exponentially. As Moore's law predicted in the 1960's [Moo06], the number of transistors have roughly doubled every other year, but as transistors are reaching their physical limitations with regard to size, there is strong evidence that Moore's law (in its current form) is nearing the end of the line. This, among other factors such as the power- and memory wall, have lead computer architects to reach for parallel processors as a means of increasing performance of computer systems [DAS12]. This shift takes several forms: The number of cores in microprocessors are ever-increasing, SIMD instructions such as Advanced Vector Extensions (AVX) extend existing instruction sets and General-Purpose Graphics Processing Units (GPGPUs) with tens of thousands of threads are commercially available.

To effectively exploit the parallelism of today's hardware, software needs to be written with parallelism in mind, but current methods such as OpenCL and CUDA are laborious and error-prone due to their low-level nature, and in the case of CUDA is non-portable. High-level programming models for parallel programming are therefore needed for software developers to take advantage of massively parallel hardware. One such model is *data-parallelism* in which the same operation is applied to different data (e.g. SIMD instructions). A limitation of many data-parallel programming languages is that they are not able to express *nested data-parallelism* [PP93]. In these languages, the programmer can't freely express nested parallel operations, and instead have to preform the tedious task of flattening by hand.

Futhark is a high-level data-parallel array language that targets, among other things, GPUs [Hen+17]. Futhark already supports *regular nested flattening* in which the arrays have uniform shape, as opposed to *irregular nested flattening* in which the nested arrays can have varying sizes. In this paper, we implement *function lifting* and full flattening of match-expressions in Futhark, as part of achieving full irregular flattening.

2. Background

2.1. Nested Data-Parallelism

The term “Data Parallelism” was introduced by Daniel Hillis and Guy L. Steele [HS86] and refers to a programming style in which the same operation is applies to different data à la SIMD. It was not until Blelloch’s data parallel language NESL [Ble95] that the notion of nested data parallelism, in which parallel constructs could themselves contain further parallel constructs, became possible.

In NESL, this nesting took the form of the *apply-to-each* construct, which somewhat resembles the list-comprehension syntax of Haskell:

$$\left\{\texttt{f}(\texttt{x}) : \texttt{x in } [\texttt{x}_0, \ldots, \texttt{x}_n]\right\};$$

but with the caveat that `f` applies to each $\texttt{x}_0$, ..., $\texttt{x}_n$ *in parallel.* As NESL supports *nested* data parallelism, the function `f` is allowed to also contain parallel constructs e.g. it might itself contain an apply-to-each. Nested data parallelism is especially useful when implementing divide-and-conquer algorithms such as quicksort, as it automatically results in proper load balancing [Ble+93].

Since then, there have been several other data parallel languages. Examples other than NESL include Proteus [PP93], Data Parallel Haskell [Pey+08] and of course also Futhark.

A major problem of nested data parallel languages is how to translate an arbitrarily deep nestings of parallel constructs onto the flat-parallel model of parallel hardware.

2.2. Futhark

Futhark [Hen17] is, as the Futhark website claims,

> [...] a **statically typed, data-parallel, and purely functional array language** in the ML family, and comes with a heavily optimising ahead-of-time compiler that presently generates either GPU code via CUDA and OpenCL, or multi-threaded CPU code. ([1])

2.2.1. Syntax

Futharks syntax is similar to Haskell and ML-style languages. The `let` keyword is used for binding variables and declaring functions (which can also be declared with the `def`

[1] `https://futhark-lang.org/`, as of May 27, 2023.

keyword). Anonymous functions (lambdas) are declared like in Haskell, an example is shown in listing 2.1. Futhark also has limited support for higher-order functions. Unlike most functional languages, Futhark does not allow recursive functions.

Listing 2.1: Contrived Futhark function for adding together two 64-bit numbers

```
def f (x : i64) (y : i64) : i64 =
  let g = \z -> x + z
   in g y
```

Beside `if`-expressions, Futhark also has a `match`-expression with pattern matching for writing conditional code. The cases of a `match`-expression *must* exhaust all of the possible values, but overlap is allowed i.e. patterns from two branches are allowed to match the same value. In such a case the first (closest to the top) branch is chosen. The '`_`' pattern matches any value and discards the value, whereas a variable name pattern matches any value and binds that value to the name. An example of a `match`-expression is shown in listing 2.2.

Listing 2.2: Futhark `match` expression with four branches.

```
def f (x : i64) (y : i64) : i64 =
  match (x, y) case (0, 0) -> -1
               case (a, 0) ->  a
               case (0, b) ->  b
               case _      ->  x * y
```

Futhark has zero-based arrays that are indexed with C-like syntax i.e. `arr[i]` is the `i`th element of array `arr`. For multi-dimensional arrays, multiple indices can be given each separated by a comma. Futhark also supports in-place updates with the syntax `arr with [i] = x`, which sets the `i`th element of `arr` equal to `x`. An example is shown in listing 2.3. To make sure that the side-effect of updating `arr` is not observable, Futhark leverages a system known as *uniqueness types*, which are denoted by asterisks.

Listing 2.3: In-place updates in Futhark. Note the asterisk prefix of the type of `a` and the return type. Example taken from [DIK23, p. 36].

```
def modify (a : *[]i32) (i : i32) (x : i32) : *[]i32 =
  a with [i] = a[i] + x
```

2.2.2. Types

Futhark has both the signed and unsigned 8, 16, 32 and 64 bit integers, 16, 32 and 64 bit floating point numbers as well as booleans as primitive types. The type of a Futhark array includes both the element type and a 64-bit integer for its length (for multi-dimensional arrays, it includes one integer per dimension), meaning that `arr : [n]f32` is an array of `n` 32-bit floats. If we omit the size in the type of an array, it is an *anonymous size* but during compilation it will be inferred.

In the declaration of a function, an asterisk prefixed to the type of a parameter means that that argument is *consumed* and it (or any aliases of it) may not be used after calling the function. A parameter whose argument is not consumed (i.e. no asterisk prefix) is *observed*. Wether or not an argument is consumed or observed is called the arguments *diet*. If a function return type is prefixed with an asterisk, it means that the result is *alias-free* i.e. it produces a new value that can't alias any of the parameters. Returning to the example in listing 2.3, we see that `a` is consumed and that the result is alias-free. While it might seem like the result aliases `a`, remember that the argument given as `a` has been consumed and can't be used anymore after `modify` has been called, meaning that it is impossible for something to alias the result.

As already shown, Futhark has tuples as a means of constructing compound types. In addition, there are also sum types and records.

2.2.3. Second-Order Array Combinators

To allow the programmer to easily write parallel code, Futhark comes with a set of parallel Second-Order Array Combinators (SOACs). A list of relevant SOACs in Futhark are briefly described in table 2.1. Here we explain each of the SOACs in more detail:

`map` takes a function f and applies it to each element of an array, in parallel, resulting in a new array of the results of f.

$$\texttt{map}\ f\ [a_1, a_2, \ldots, a_n] = [f\ a_1, f\ a_2, \ldots, f\ a_n]$$

`iota` takes a non-negative 64-bit integer n and produces an array of length n with the first element (if it exists) being 0, the next 1 and so on, until the last element which is equal to $n-1$.

$$\texttt{iota}\ n = [0, 1, \ldots, n-1]$$

`replicate` takes a non-negative 64-bit integer n and a value x, and returns an array filled with n copies of x.

$$\texttt{replicate}\ n\ x = \overbrace{[x, \ldots, x]}^{n}$$

`reduce` takes an *associative* binary operator $\oplus$ and a neutral element to that operator $0_\oplus$ and applies a fold (in parallel) over an array as.

$$\texttt{reduce}\ \oplus\ 0_\oplus\ [a_1, a_2, \ldots, a_n] = a_1 \oplus a_2 \oplus \ldots \oplus a_n$$

`scan` is like `reduce`, it takes an associative binary operator $\oplus$ and a neutral element $0_\oplus$, but returns an array of intermediate results from left to right such that the last result is equal to an equivalent `reduce`.

The default Futhark `scan` is *inclusive*, meaning that the first element of the scan is equal to the first element of the input array. There is also a version called the *exclusive* `scan` for which the first element of the resulting array is the neutral element $0_\oplus$ i.e. all of the elements are "shifted" one to the left from an inclusive scan. Unless otherwise indicated, a `scan` is assumed to be inclusive.

$$\begin{aligned}\texttt{scan}^{inc}\ \oplus\ 0_\oplus\ [a_1, a_2, \ldots, a_n] &= [a_1, a_1 \oplus a_2, \ldots, a_1 \oplus a_2 \oplus \ldots \oplus a_n]\\ \texttt{scan}^{exc}\ \oplus\ 0_\oplus\ [a_1, a_2, \ldots, a_n] &= [0_\oplus, 0_\oplus \oplus a_1, \ldots, 0_\oplus \oplus a_1 \oplus \ldots \oplus a_{n-1}]\end{aligned}$$

`scatter` takes three arrays: `dest`, `is` and `vs` and, for each pair `(i, v)` in the zipped `is` and `vs`, writes `v` to `dest` at index `i`. If an index is out-of-bounds, the write is ignored. If multiple different values are written at the same index, the result is undefined. `dest` is consumed, meaning the first argument passed to a `scatter` can't be used after.

Function signature			Description
`map`	:	$(\texttt{f} : \alpha \to \beta) \to (\texttt{as} : [n]\alpha) \to {}^*[n]\beta$	Applies the function `f` to each element of `as`.
`iota`	:	$(\texttt{n} : \texttt{i64}) \to {}^*[n]\texttt{i64}$	Create an array counting from 0 up to (but not including) `n`.
`replicate`	:	$(\texttt{n} : \texttt{i64}) \to (\texttt{x} : \tau) \to {}^*[n]\tau$	Create an array of `n` copies of `x`.
`reduce`	:	$(\oplus : \alpha \to \alpha \to \alpha) \to (0_\oplus : \alpha)$	A parallel fold.
	$\to$	$(\texttt{as} : [n]\alpha) \to \alpha$	
`scan`	:	$(\oplus : \alpha \to \alpha \to \alpha) \to (0_\oplus : \alpha)$	Like `reduce` but keeps intermediate results.
	$\to$	$(\texttt{as} : [n]\alpha) \to {}^*[n]\alpha$	
`scatter`	:	$(\texttt{dest} : {}^*[k]\tau) \to (\texttt{is} : [n]\texttt{i64})$	Writes the values of `vs` to `dest` at the indices `is`. This consumes `dest`.
	$\to$	$(\texttt{vs} : [n]\tau) \to {}^*[k]\tau$	

Table 2.1.: A list of SOACs in Futhark along with their descriptions.

3. Irregular Flattening

3.1. Representing Irregular Arrays

In most programming languages, arrays are represented as a pointer to a contiguous section of memory. The array might also include an integer either along with the pointer or before the array elements that denotes the length of the array such that bounds-checking can be automatically done at runtime. In such a language — with the array length before the first element — an *irregular array* (array of arrays, each with potentially different lengths) might be represented as an array of a pointers to other arrays, as shown in fig. 3.1.

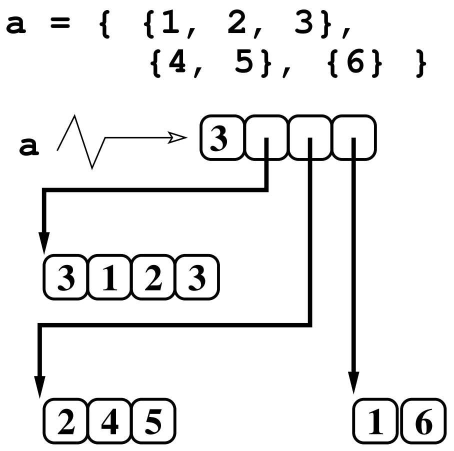


Figure 3.1.: Irregular array represented as pointers to arrays. From [Oan18b, slide 25].

This representation of using pointers is not suitable for parallelizing operations over irregular arrays as GPUs typically require flat arrays. We instead represent irregular arrays with two arrays: A *data array* and a *segment array* (also: shape array or length array) which respectively describe the content and shape of the irregular array. The data array is a flat array of all of the elements of the irregular array i.e. it is just the concatenation of the sub-arrays of the irregular array. The segment array contains the lengths of each sub-array of the irregular array [Ble90, p. 6]. Henceforth, the sub-arrays of an irregular array will be called its *segments*. For example, the array from fig. 3.1 would be represented by the data array $\mathtt{D}_a$ and segment array $\mathtt{S}_a$:

$$\begin{aligned}\texttt{D}_a &= \begin{bmatrix}1 & 2 & 3 & 4 & 5 & 6\end{bmatrix}\\ \texttt{S}_a &= \begin{bmatrix}3 & 2 & 1\end{bmatrix}\end{aligned}$$

We can generalize this to nested irregular arrays of any depth by using one segment array for each nesting level. The first segment array will simply hold the length of the outer array, the next holds the lengths of the segments at nesting level one and so on. As an example with an array with nesting depth three:

$$\begin{aligned}arr &= \begin{bmatrix}\begin{bmatrix}\begin{bmatrix}1 & 2\end{bmatrix} & \begin{bmatrix}3\end{bmatrix}\end{bmatrix} & \begin{bmatrix}\end{bmatrix} & \begin{bmatrix}\begin{bmatrix}\end{bmatrix} & \begin{bmatrix}4 & 5\end{bmatrix}\end{bmatrix} & \begin{bmatrix}\begin{bmatrix}6\end{bmatrix}\end{bmatrix}\end{bmatrix}\\ \texttt{D}_{arr} &= \begin{bmatrix}1 & 2 & 3 & 4 & 5 & 6\end{bmatrix}\\ \texttt{S}^0_{arr} &= \begin{bmatrix}4\end{bmatrix}\\ \texttt{S}^1_{arr} &= \begin{bmatrix}2 & 0 & 2 & 1\end{bmatrix}\\ \texttt{S}^2_{arr} &= \begin{bmatrix}2 & 1 & 0 & 2 & 1\end{bmatrix}\end{aligned}$$

A property of the segment arrays is that the sum of $\texttt{S}^i_{arr}$ is equal to the length of $\texttt{S}^{i+1}_{arr}$, and the sum of the final segment array is equal to the length of the data array. When dealing with two dimensional irregular arrays, we will forgo this notation and simply use a single segment array $\texttt{S}_{arr}$ with no superscript, which denotes the lengths of the segments of arr. Unless otherwise noted, the following part assumes we are only dealing with two dimensional arrays.

While the data- and segment array are enough to represent any irregular nested array, there are other auxiliary structures useful for representing irregular nested arrays [Oan22].

Offset array (`B`) The offset array has the same length as the segment array, and denotes where the segments start in the flat data.

Flag array (`F`) The flag array has the same length as the flat data, and has a ‘`T`’ at index i if the element at index i starts a new segment, and otherwise has a ‘`F`’.

Segment- and Inner indices ($\texttt{II}^1$ & $\texttt{II}^2$) Both have the same length as the data array. $\texttt{II}^1[i]$ is the index of the segment that the ith element of the flat data belongs to. $\texttt{II}^2[i]$ is the index within a segment that the ith element of the flat data belongs to.

These can also be generalized to arrays of deeper nesting levels, but we will not show that here.

We use the array from fig. 3.1 to demonstrate these auxiliary arrays in fig. 3.2

$$
\begin{aligned}
a &= \begin{bmatrix}\begin{bmatrix}1 & 2 & 3\end{bmatrix} & \begin{bmatrix}4 & 5\end{bmatrix} & \begin{bmatrix}6\end{bmatrix}\end{bmatrix}\\
\mathtt{D}_a &= \begin{bmatrix}1 & 2 & 3 & 4 & 5 & 6\end{bmatrix}\\
\mathtt{F}_a &= \begin{bmatrix}\mathtt{T} & \mathtt{F} & \mathtt{F} & \mathtt{T} & \mathtt{F} & \mathtt{T}\end{bmatrix}\\
\mathtt{II}_a^1 &= \begin{bmatrix}0 & 0 & 0 & 1 & 1 & 2\end{bmatrix}\\
\mathtt{II}_a^2 &= \begin{bmatrix}0 & 1 & 2 & 0 & 1 & 0\end{bmatrix}\\
\mathtt{S}_a &= \begin{bmatrix}3 & 2 & 1\end{bmatrix}\\
\mathtt{B}_a &= \begin{bmatrix}0 & 3 & 5\end{bmatrix}
\end{aligned}
$$

Figure 3.2.: Example showing the full representation of the array in fig. 3.1.

3.2. Segmented Operations

We've described a way of representing nested irregular arrays, but we do not have any operations to work on these arrays. In this section, we introduce segmented versions of common Futhark SOACs that, as the name implies, work independently over segments of arrays. All of the segmented operations that are shown are adapted from [Oan18a].

3.2.1. Segmented Scan

The segmented scan does a scan over each segment of an array. The intuition behind the segmented scan is that we can take any associative binary operator $\oplus$, and we *augment* it with a flag value to turn it into a new operator associative $\oplus'$ which we can then use in a scan to get a segmented scan [Sch80, p. 491]. The definition of the segmented operator is shown in listing 3.1. The neutral element $0_{\oplus'}$ of our segmented operator is simply $(\mathtt{F}, 0_{\oplus})$.

Listing 3.1: Function for making a binary operator segmented.

```
def segop 'τ (op      : τ -> τ -> τ)
             (flag_x  : bool, x : τ)
             (flag_y  : bool, y : τ): (bool, τ) =
  let flag  = flag_x || flag_y
  let value = if flag_y
              then y
              else op x y
   in (flag, value)
```

Listing 3.2: Segmented scan in Futhark.

```
def segscan [n] 'τ (op    : τ -> τ -> τ)
                   (ne    : τ)
                   (flags : [n]bool)
                   (arr   : [n]τ): [n]τ =
  let op'       = segop op         -- segmented operator
  let ne'       = (false, ne)      -- neutral element of seg op
  let fvs       = zip flags arr    -- flag/value pairs
  let (_, res)  = unzip <| scan op' ne' fvs
   in res
```

Now, a segmented scan is simply a regular scan with our segmented operator. A segmented scan implemented in Futhark is shown in listing 3.2

Note that this segmented scan is inclusive. While we won't show the implementation, it's quite straight forward to make a segmented exclusive scan.

3.2.2. Segmented Reduce

Segmented reduce applies a reduce over each segment of an array. To implement segmented reduce, we use the fact that the last element of an inclusive scan is equal to the result of a reduce with the same operator. We therefore do a segmented scan over the data array with the corresponding flag array. To extract the final element from each segment, we do an inclusive scan over the segment array, which gives us the index of the start of each segment, starting from the second, which we'll then just subtract 1 from to get the indices of the ends of the segments. An implementation in Futhark is shown in listing 3.3.

Listing 3.3: Segmented reduce in Futhark.

```
def segreduce [w][n] 'τ (op    : τ -> τ -> τ)
                        (ne    : τ)
                        (flags : [n]bool)
                        (segs  : [w]bool)
                        (arr   : [n]τ): [w]τ =
  let sc_arr = segscan op ne flags arr
  let indsp1 = scan (+) 0 segs
  in map2 (\seg ip1 -> if seg == 0 then ne
                       else sc_arr[ip1-1]
          ) segs inds
```

3.2.3. Segmented Iota

The idea behind segmented iota is that, if we map `iota` over an array `arr`, we get a segmented array for which `arr` is the segment descriptor. We make the observation that `iota n` can be implemented as an exclusive prefix-sum over an array of `n` 1s. The segmented operation is then simply using a segmented exclusive scan instead. The implementation is shown in listing 3.4.

Listing 3.4: Segmented iota in Futhark.

```
def segiota [n] (flags : [n]bool): [n]i64 =
  let ones = replicate n 1
  in segscan^exc (+) 0 flags ones
```

3.2.4. Constructing the Auxiliary Arrays

Here we briefly show how to construct the auxiliary arrays from the segment array using the segmented operations we have described. The code is shown in listing 3.5. The `mkFlags` function allow for any values for the flags (and lack thereof), whereas `mkFlagsTF` constructs a boolean flag array.

3.3. Flattening

Segmented operations give us a way to operate on irregular nested arrays, but they require the programmer to manually keep track of the irregular representation of the irregular nested arrays. Here we introduce the *flattening transformation* denoted $\mathcal{F}(\cdot)$, which takes as input a program of the form "`map (`λ`xs ... ->` e`) xss ...`" where e is an expression that may contain further nested parallelism and "`xss`..." are (potentially) irregular arrays, and transforms it to a semantically equivalent program that only uses one level of parallelism. It does so by (a) translating e into a series of simple statements, (b) distributing the map over these statements and flatting them and (c) applying a set of *flattening rules*, which we will cover shortly. We also make the assumption that e does not contain any free variables i.e. all variables in e are bound in the lambda of the `map`. To deal with this in practice, we use *replication* wich we'll describe later.

When flattening, we are always dealing with one nesting level (i.e. one `map`) at a time. This means that, for deeper nested irregular arrays, we can flatten their inner nesting levels to one irregular level e.g. a `map` over an array `arr[w][n][m]` where `n` and `m` can vary over the outer dimension `w` can be flattened to a two-dimensional array `arr[w][n*m]` by collapsing the inner segments. Using the example from section 3.1, we can collapse the inner dimension of *arr* which is shown in fig. 3.3. This can be done for any nesting level by iteratively flattening the inner segments. It is therefore enough to only have a single segment array for the representation of an irregular array for each step in flattening, even if the irregular array has more than two dimensions.

Listing 3.5: Futhark functions for constructing auxiliary arrays.

```
def mkOffsets [n] (segs : [n]i64): [n]i64 =
  scan^exc (+) 0 segs

def mkFlags 'τ [n] (segs  : [n]i64)
                   (zero  : τ)
                   (flags : [n]τ): []τ =
  let m = reduce (+) 0 segs
  let offs = mkOffsets segs
  let inds = map2 (\seg ind ->
                    if seg == 0 then -1
                                else ind
                  ) segs offs
   in scatter (replicate m zero) inds flags

def mkFlagsTF [n] (segs : [n]i64): []bool =
  mkFlags segs false (replicate n true)

def mkSegmentInds [n] (segs : [n]i64): []i64 =
  let heads = mkFlags segs 0 (iota n)
  let flags = mkFlagsTF segs
  in segscan (+) 0 flags heads

def mkInnerInds [n] (segs : [n]i64): []i64 =
  let flags = mkFlagsTF segs
  in segiota flags
```

$$
\begin{aligned}
arr' &= \begin{bmatrix} \begin{bmatrix}1 & 2 & 3\end{bmatrix} & \begin{bmatrix}\,\end{bmatrix} & \begin{bmatrix}4 & 5\end{bmatrix} & \begin{bmatrix}6\end{bmatrix} \end{bmatrix} \\
\mathtt{D}_{arr'} &= \begin{bmatrix}1 & 2 & 3 & 4 & 5 & 6\end{bmatrix} \\
\mathtt{S}_{arr'} &= \begin{bmatrix}3 & 0 & 2 & 1\end{bmatrix} \\
\mathtt{B}_{arr'} &= \begin{bmatrix}0 & 3 & 3 & 5\end{bmatrix} \\
\mathtt{F}_{arr'} &= \begin{bmatrix}1 & 0 & 0 & 1 & 0 & 1\end{bmatrix}
\end{aligned}
$$

Figure 3.3.: Collapsing the inner segments of the three-dimensional array *arr* from section 3.1, along with its new irregular representation.

We now introduce a few flattening rules for some of the Futhark SOACs, adapted from [Oan18a; Oan22] (note that those are presented in a bottom-up manner, whereas we use a top-down manner of flattening). When presenting these flattening rules we assume that we always have the irregular representation of the input arrays (the arrays we are mapping over) available. Additionally, if we have the segment array $\mathtt{S}_{xss}$ for an array `xss` we also assume we have the full shape representation available i.e. the offset array $\mathtt{B}_{xss}$, flag array $\mathtt{F}_{xss}$ segment indices $\mathtt{II}^1_{xss}$ and inner indices $\mathtt{II}^2_{xss}$. We do this for notational convenience as we've already shown how to construct these auxiliary arrays from the segment array.

If the result from flattening is a regular array, the result of the flattening rule is simply the regular array. If the result from flattening is instead an irregular array, the result of the flattening rule is the segment array and the flat data array which describe the irregular array.

3.3.1. Flattening scan inside map

Flattening a scan preserves the shape of the outer array.

$$
\begin{aligned}
&\mathrm{res} = \mathcal{F}(\mathrm{map}\ (\backslash \mathrm{xs} \rightarrow \mathrm{scan}\ \oplus\ 0_{\oplus}\ \mathrm{xs})\ \mathrm{xss})\\
&\Rightarrow\\
&\mathtt{S}_{res} = \mathtt{S}_{xss}\\
&\mathtt{D}_{res} = \mathrm{segscan}\ \oplus\ 0_{\oplus}\ \mathtt{F}_{xss}\ \mathtt{D}_{xss}
\end{aligned}
$$

3.3.2. Flattening reduce inside map

Flattening a reduce results in a regular array.

$$
\begin{aligned}
&\mathrm{res} = \mathcal{F}(\mathrm{map}\ (\backslash \mathrm{xs} \rightarrow \mathrm{reduce}\ \oplus\ 0_{\oplus}\ \mathrm{xs})\ \mathrm{xss})\\
&\Rightarrow\\
&\mathrm{res} = \mathrm{segreduce}\ \oplus\ 0_{\oplus}\ \mathtt{F}_{xss}\ \mathtt{D}_{xss}
\end{aligned}
$$

3.3.3. Flattening iota inside map

As `iota n` gives an array of length `n`, flattening an iota gives an array whose segment array is the input array.

$$
\begin{aligned}
&\mathrm{res} = \mathcal{F}(\mathrm{map}\ (\backslash \mathrm{n} \rightarrow \mathrm{iota}\ \mathrm{n})\ \mathrm{ns})\\
&\Rightarrow\\
&\mathtt{S}_{res} = \mathrm{ns}\\
&\mathtt{D}_{res} = \mathrm{segiota}\ \mathtt{F}_{res}
\end{aligned}
$$

3.3.4. Flattening map inside map

A map inside another map preserves the shape of the outer array. We strip away the outer map and flatten the inner map (as e might contain further nested parallelism). To make sure the dimensions match, we flatten the outer dimension of `xss` with the `flatten` function from the Futhark prelude. After stripping away the outer map, e might contain free variables, specifically if e uses variables bound in the outer map lambda. To solve this we use replication, which will be explained later.

```
res = $\mathcal{F}$(map (\xs -> map (\x -> $e$) xs) xss)
$\Rightarrow$
res' = $\mathcal{F}$(map (\x -> $e$) (flatten xss))
S$_{res}$ = S$_{xss}$
D$_{res}$ = D$_{res'}$
```

As an example, we'll show the flattening transformation where $e =$ `reduce (+) 0 x` and `xss` $= [[[3]], [[2, 1], [0]]]$. Let `xs'` $=$ `flatten xss` $= [[3], [2, 1], [0]]$:

```
res = $\mathcal{F}$(map (\xs -> map (\x -> reduce (+) 0 x) xs) xss
$\Rightarrow$
res' = $\mathcal{F}$(map (\x -> reduce (+) 0 x) xs')
$\Rightarrow$
res' = segreduce (+) 0 F$_{xs'}$ D$_{xs'}$ -- F$_{xs'}$ = [T,T,F,T], D$_{xs'}$ = $[3, 2, 1, 0]$
S$_{res}$ = S$_{xss}$ -- = $[1, 2]$
D$_{res}$ = D$_{res'}$ -- = $[3, 3, 0]$
```

As can be seen from the irregular representation, the result is `res =` $[[3], [3, 0]]$ which is what we'd expect from summing the inner segments of `xss`.

3.3.5. Flattening if-then-else inside map

Flattening if-then-else expressions are tricky, as each thread of the GPU has to preform the same operation and, by nature, if-then-else expressions preform different operations. We use a technique called *branch-packing* [Ble90, p. 154], in which the inputs for the 'true' and 'false' branches are partitioned such that the two branches can be executed separately and the results can then be merged. We will first need to introduce two functions: `splitInput` and `mergeResult`, shown in listings 3.6 and 3.7 resp. `splitInput` is a segmented version of `gather`[1]: It collects the segments at indices `is` of an irregular array `xss`. `mergeResult` is a segmented version of `scatter`: It writes back segments of an irregular array `xss` to a destination array at indices `is`.

We can now write the flattening rule for `if-then-else`-expressions in a map, which is shown in listing 3.8. We first split the inputs of the 'true' and 'false' branch, and then

[1]See `https://futhark-lang.org/examples/gather-and-scatter.html` for a `gather` example in Futhark.

Listing 3.6: Take the segments of array `xss` at indices `is`.

```
def splitInput 'τ (is    : []i64)
                  (S_xss : []i64)
                  (D_xss : []τ) : ([]i64, []τ) =
  let S_res = map (\i -> S_xss[i]) is
  let D_res = map2 (\ii1 ii2 ->
                     let idx = B_xss[is[ii1]] + ii2
                     in D_xss[idx]
                   ) II¹_res II²_res
  in res -- We implicitly construct 'res' from 'S_res' and 'D_res'.
```

Listing 3.7: Write segments of array `xss` to `dest` at indices `is`.

```
def mergeResult 'τ (dest  : *[]τ)
                   (B_res :  []i64)
                   (is    :  []i64)
                   (S_xss :  []i64)
                   (D_xss :  []τ) : []τ =
  let idxs = map2 (\ii1 ii2 -> B_res[is[ii1]] + ii2) II¹_xss II²_xss
  in scatter dest idxs D_xss
```

flatten the two expressions e_{T} and e_{F} with their corresponding inputs. We then use a `scatter` to calculate the shape of the result from the shapes of the results of the two branches, and finally we merge the results of the two branches together to a single result.

3.3.6. Flattening match-expressions inside map

It is possible to translate any Futhark `match`-expression into a series of nested `if-then-else`-expressions, and we therefore already have a naive way of flattening `match`-expressions. It is not very efficient, however, as it requires $k - 1$ calls to `partition` for a `match`-expression with k cases. We therefore present a generalized version of the `if-then-else` flattening transformation that works for `match`-expressions, which is shown in listing 3.10. It works similarly to flattening `if-then-else`-expressions, but instead of partitioning the input in a 'true' and 'false' branch, we partition it into k different branches, flatten each branch, and then merge each of the results of the k branches together to a final result. `partitionK` does the heavy lifting, partitioning an array `ns` into k different equivalence classes. It returns a flat permutation of `ns` as well as a segment array describing the sizes of the partitions. We use those segment sizes and their corresponding offsets to slice $\mathtt{D}_{iss}$ for each of the k branches. The definition of `partitionK` is shown in listing 3.9.

Listing 3.8: Flattening rule for `if-then-else`-expressions in a map.

```
res = $\mathcal{F}$(map2 (\b xs -> if b then $e_\mathsf{T}$ else $e_\mathsf{F}$) bs xss)
$\Rightarrow$
(is$_\mathsf{T}$, is$_\mathsf{F}$) = partition (\i -> bs[i]) (iota (length bs))
xss$_\mathsf{T}$ = splitInput is$_\mathsf{T}$ $\mathtt{S}_{xss}$ $\mathtt{D}_{xss}$
xss$_\mathsf{F}$ = splitInput is$_\mathsf{F}$ $\mathtt{S}_{xss}$ $\mathtt{D}_{xss}$
res$_\mathsf{T}$ = $\mathcal{F}$(map (\xs -> $e_\mathsf{T}$) xss$_\mathsf{T}$)
res$_\mathsf{F}$ = $\mathcal{F}$(map (\xs -> $e_\mathsf{F}$) xss$_\mathsf{F}$)
$\mathtt{S}_{res}$ = scatter (replicate n 0) (is$_\mathsf{T}$ ++ is$_\mathsf{F}$) ($\mathtt{S}_{res_\mathsf{T}}$ ++ $\mathtt{S}_{res_\mathsf{F}}$)
blank_res   = replicate (reduce (+) 0 $\mathtt{S}_{res}$) 0
partial_res = mergeResult blank_res $\mathtt{B}_{res}$ is$_\mathsf{T}$ $\mathtt{S}_{res_\mathsf{T}}$ $\mathtt{D}_{res_\mathsf{T}}$
$\mathtt{D}_{res}$ = mergeResult parital_res $\mathtt{B}_{res}$ is$_\mathsf{F}$ $\mathtt{S}_{res_\mathsf{F}}$ $\mathtt{D}_{res_\mathsf{F}}$
```

Listing 3.9: *k*-way partition.

```
def partitionK [n] (k  : i64)
                   (p  : i64 -> i64)
                   (ns : [n]i64) : ([k]i64, [n]i64) =
  let cls_flags = tabulate_2d k n (\i j ->
                    if p ns[j] == i
                      then 1 else 0
                  )
  let loc_offs  = map (scanExc (+) 0) cls_flags
  let counts    = map (reduce (+) 0) cls_flags
  let glob_offs = scanExc (+) 0 counts
  let cls_offs  = tabulate_2d k n (\i j ->
                    glob_offs[i] + loc_offs[i,j]
                  )
  let is        = map (\i -> cls_offs[p ns[i], i]) (iota n)
  let blank     = replicate n 0
  let res       = scatter blank is ns
  in (counts, res)
```

Listing 3.10: Flattening rule for `match`-expressions in a map.

```
res = 𝓕(map2 (\c xs -> match c case p_1 -> e_1
                                ...
                                case p_k -> e_k
             ) cs xss)
⇒
cls = map (\c -> match c case p_1 -> 0
                         ...
                         case p_k -> k-1
          ) cs
(S_iss, D_iss) = partitionK k (\i -> cls[i]) (iota (length cls))
is_1 = D_iss[B_iss[0]:B_iss[0] + S_iss[0]]
xss_1 = splitInput is_1 S_xss D_xss
res_1 = 𝓕(map (\xs -> e_1) xss_1)
...
is_k = D_iss[B_iss[k-1]:B_iss[k-1] + S_iss[k-1]]
xss_k = splitInput is_k S_xss D_xss
res_k = 𝓕(map (\xs -> e_k) xss_k)
S_res = scatter (replicate n 0) iss (flatten [S_res_1,...,S_res_k])
partial_res_0 = replicate (reduce (+) 0 S_res) 0
partial_res_1 = mergeResult partial_res_0 B_res is_1 S_res_1 D_res_1
...
partial_res_k = mergeResult blank_res_{k-1} B_res is_k S_res_k D_res_k
D_res = partial_res_k
```

3.4. Function Lifting

While we have shown how to flatten a subset of the operations in Futhark, the question still remains how we flatten function calls i.e. what is the rule for flattening $\mathcal{F}\big($`map (λxs -> f xs) xss`$\big)$? The answer is that we *lift* the function `f` into a function $\texttt{f}^L$ such that $\texttt{f}^L$ `xss` is semantically equivalent to `map (λxs -> f xs) xss` [Pey+08].

For notational convenience we will restrict ourselves to only considering functions of a single argument (but the argument can be a tuple). We will also ignore some of the finer points about the Futhark type system in this section, especially w.r.t. size types.

To lift a function `f`, we add a parameter `w` which is the size of the outer map, and then we *lift its signature* and *lift its body*:

$$\texttt{f}\ (\texttt{args} : \alpha) : \beta = e$$
$$\Downarrow$$
$$\texttt{f}^L\ (\texttt{w} : \texttt{i64})\ (\texttt{args} : \mathcal{L}_t[\![\alpha]\!]) : \mathcal{L}_t[\![\beta]\!] = \mathcal{L}[\![e]\!]$$

The rules for lifting types is shown in fig. 3.4. The first rules states that lifting a scalar type becomes a (regular) array with `w` as its length. The second rule states that lifting an array type becomes an irregular array, which we represent as a segment array with the type `[w]i64` and a flat data array with the type `[]`τ. The third rule just states that to lift a tuple you just lift the inner types.

$$\begin{array}{rcll}
\mathcal{L}_t[\![\tau]\!] & = & \texttt{[w]}\tau & \text{Scalars become arrays} \\
\mathcal{L}_t[\![\texttt{[]}\tau]\!] & = & (\texttt{[w]i64}, \texttt{[]}\tau) & \text{Arrays become irregular arrays} \\
\mathcal{L}_t\big[\!\big[(\tau_1, \ldots, \tau_k)\big]\!\big] & = & (\mathcal{L}_t[\![\tau_1]\!], \ldots, \mathcal{L}_t[\![\tau_k]\!]) & \text{Lift inner types of tuple arguments}
\end{array}$$

Figure 3.4.: Rules for lifting function types.

Lifting the body e of a function amounts to performing the flattening-transformation $\mathcal{F}\big($`map (λ(xs`$_1$`,…,xs`$_k$`) -> e) (zip xss`$_1$` … xss`$_k$`)`$\big)$ where `xss`$_1$`,…,xss`$_k$ are the lifted arguments of `f`.

As an example, consider lifting the function

```
mango (n : i32, xs : []i32) : i32 = n + reduce (+) 0 xs
```

Its `n` argument would become an array, and its `xs` argument would become two arguments: a segment array $\texttt{S}_{xss}$ and a flat data array $\texttt{D}_{xss}$ and it would return a regular array of type `[w]i32`. The lifting transformation is shown in listing 3.11. In step 1 the inner body of the map being flattened is distributed. In step 2 the “reduce inside map” flattening rule is used and in step 3 the result is flattened and as it had no inner parallelism, we just simplified the result.

The function `mango`^{L} has — excluding `w` — a regular array argument its first argument, an irregular array as its second argument and its return type is a regular array. Consider instead the function:

```
banjo (n : i64) : []i64 = iota (n*2)
```

Its lifted counterpart would have — again excluding `w` — one regular array as its argument, and it would return an irregular array (in the form of a shape- and flat data array). We show the lifting of `banjo` in listing 3.12. In step 1 the inner body of the map being flattened is distributed. In step 2 the first statement is flattened. As there's no inner parallelism, we just simplify it. And in step 3 the "iota inside map" flattening rule is used to get the result.

We can now finally show the flattening rule for functions inside a map. There are four different rules, depending on the type of the function:

(R1) If both the argument and result of `f` are scalars

```
res = 𝓕(map (\x -> f x) xs)
⇒
w   = length xs
res = f^L w xs
```

(R2) If the argument of `f` is a scalar and the result is an array

```
res = 𝓕(map (\x -> f x) xs)
⇒
w   = length xs
(S_res, D_res) = f^L w xs
```

(R3) If the argument of `f` is an array and the result is a scalar

```
res = 𝓕(map (\xs -> f xs) xss)
⇒
w   = length S_xss
res = f^L w (S_xss, D_xss)
```

(R4) If both the argument and result of `f` is an array

```
res = 𝓕(map (\xs -> f xs) xss)
⇒
w   = length S_xss
(S_res, D_res) = f^L w (S_xss, D_xss)
```

There is one additional caveat when flattening function application: if the argument to the function `f` is a constant, we need to replicate it such that it becomes an array of length `w` and pass that array as the argument to `f`^{L} instead.

Listing 3.11: Lifting a simple function.

```
mango^L (w : i64)
    (ns : [w]i32
   ,(S_xss : [w]i64, D_xss : []i32)) : [w]i32
 = 𝓕(map (\(n, xs) -> n + reduce (+) 0 xs) (zip ns xss))
 ⇒1
 = let reds = 𝓕(map (\xs -> reduce (+) 0 xs) xss)
   in 𝓕(map (\(n,red) -> n + red) (zip ns reds))
 ⇒2
 = let reds = segreduce (+) 0 F_xss D_xss
   in 𝓕(map (\(n,red) -> n + red) (zip ns reds))
 ⇒3
 = let reds = segreduce (+) 0 F_xss D_xss
   in map2 (+) ns reds
```

Listing 3.12: Lifting another simple function

```
banjo^L (w : i64) (ns : [w]i64) : ([w]i64, []i64)
 = 𝓕(map \(n -> iota (n*2)) ns)
 ⇒1
 = let n2s = 𝓕(map (\n -> n * 2) ns)
   in 𝓕(map (\n2 -> iota n2) n2s)
 ⇒2
 = let n2s = map (*2) ns
   in 𝓕(map (\n2 -> iota n2) n2s)
 ⇒3
 = let n2s = map (*2) ns
   in let S_res = n2s
      let D_res = segiota F_res
      in (S_res, D_res)
```

Of course actual Futhark functions can have more than one argument, and multiple results via tuples, which means that you sometimes have to combine these four rules. To illustrate this, we'll show the flattening of the expression

```
map (λxs -> mango (5, xs)) xss
```

Note that one of the arguments is a constant and will therefore need to be replicated in order to be passed to the lifted version of `mango`.

The example is shown in listing 3.13. In step 1 The '5' is replicated into the array `fives` and given as argument to the map. In step 2 we use both flattening rule (R1) and flattening rule (R3), which means we pass the `fives` array unchanhed and the segment- and data arrays of `xss` to `mango`^{L}.

Listing 3.13: Flattening example of function inside map

```
res = 𝓕(map (\xs -> mango (5, xs)) xss)
⇒1
res = let fives = replicate (length S_xss) 5
      in 𝓕(map2 (\xs fives -> mango (five, xs)) xss fives)
⇒2
w   = length S_xss
res = let fives = replicate (length S_xss) 5
      in mango^L w (fives, (S_xss, D_xss))
```

4. Implementation

While we've shown the flattening rules and function lifting transformations as operations on (a subset of) the Futhark source language, the actual implementation of irregular flattening is implemented as a compiler pass[1] that operates on the Futhark *intermediate representation* (IR), and is written in Haskell just as the rest of the Futhark compiler. A Futhark compiler pass is simply a function from one program to another program, potentially with a different *representation*. In the case of the flattening compiler pass, it transforms a program with the "SOACS" representation, which supports nested parallelism, to a program with the "GPU" representation, which only supports flat parallelism.

The Futhark IR is a first-order, monomorphic language which greatly simplify the implementation of the flattening as we don't have to deal with lifting higher-order functions which would quickly leads to problems such as arrays of functions [Pey+08, p. 396]. The IR only has values of array and primitive types, which means that user-defined data structures and tuples from the source language are translated into multiple values, which is why we haven't mentioned records or sum types in regards to flattening. Additionally, the Futhark IR is in *Administrative Normal Form* (ANF) where all sub-expressions are either constants or variables. As an example showing the Futhark IR, the source-level statement

```
let res = (x + y + 1, 2 * x - y)
```

would be translated to something like the following Futhark IR:

```
let {a     : i32} = x + y
let {res_0 : i32} = a + 1
let {b     : i32} = x * 2
let {res_1 : i32} = b - y
in {res_0, res_1}
```

This means means that the first step of flattening — translating the mapped expression into simple statements — is already taken care of.

In the implementation we have to be more aware of which arrays are regular and which are irregular. It is of course possible to represent a regular array with an irregular representation, but in practice this is inefficient as some of the flattening rules can be optimized if we know that their inputs are regular. We therefore make the distinction: An arrays *representation* is either as a regular array, in which case it is simply represented

[1]The flattening compiler pass can be found at https://github.com/diku-dk/futhark/blob/e49203a0b13139bb983f62c39363521c2b65c8e7/src/Futhark/Pass/Flatten.hs

as-is, or an irregular array, in which case it is represented by its its segments-, flags-, offset- and data array (in that order). Using four arrays to represent an irregular array instead of just the data- and segment array adds some overhead but also saves us from having to re-calculate the offset- and flag arrays.

When introducing the flattening rules and function lifting, we took the liberty of seamlessly converting between an irregular array and its representation i.e. if an irregular array `xss` was in scope, its data array $\mathtt{D}_{xss}$ and shape descriptors $\mathtt{S}_{xss}$, $\mathtt{B}_{xss}$, etc. were also in scope and, conversely, if the data- and segment arrays of `xss` were in scope, so was `xss`. In the actual implementation we have to do the bookkeeping manually, which is done by associating a *result tag* with each result from flattening a statement, and then provide a mapping from variables used in a statements to result tags (called the *distribution inputs*, denoted Σ) and a mapping from result tags to array representations (called the *distribution environment*, denoted $\mathcal{E}$).

To make this more clear, we return to the example of lifting the `mango` function, which was shown in listing 3.11 on page 24. When we start flattening the function body, our distribution inputs are

$$\Sigma = [\mathtt{n} \mapsto \mathtt{r0}, \mathtt{xs} \mapsto \mathtt{r1}, \mathtt{red} \mapsto \mathtt{r2}, \mathtt{mango\_res} \mapsto \mathtt{r3}]$$

and our initial distribution environment is

$$\begin{aligned}\mathcal{E}_0 = [\ & \mathtt{r0} \mapsto \mathsf{Regular}\ \mathtt{ns} \\ ,\ & \mathtt{r1} \mapsto \mathsf{Irregular}\ \mathtt{S}_{xss}\ \mathtt{F}_{xss}\ \mathtt{B}_{xss}\ \mathtt{D}_{xss}]\end{aligned}$$

When flattening the first statement we lookup `xs` in Σ which gives us the tag `r1`, which we can then use to get the irregular representation of `xss` via a lookup in $\mathcal{E}_0$ which we then use in the `segreduce`. After flattening the statement, we add the result (which is a regular array) to the distribution environment, which is now

$$\begin{aligned}\mathcal{E}_1 = [\ & \mathtt{r0} \mapsto \mathsf{Regular}\ \mathtt{ns} \\ ,\ & \mathtt{r1} \mapsto \mathsf{Irregular}\ \mathtt{S}_{xss}\ \mathtt{F}_{xss}\ \mathtt{B}_{xss}\ \mathtt{D}_{xss} \\ ,\ & \mathtt{r2} \mapsto \mathsf{Regular}\ \mathtt{reds}]\end{aligned}$$

When flattening the second statement we lookup the `n` and `red` in Σ which gives the tags `r0` and `r2` resp. We lookup these in $\mathcal{E}_1$ to get the representation of `ns` and `reds` which we then map the `(+)` over. After flattening this statement, we add the result (which again is a regular array) to the distribution environment, which then becomes

$$\begin{aligned}\mathcal{E}_2 = [\ & \mathtt{r0} \mapsto \mathsf{Regular}\ \mathtt{ns} \\ ,\ & \mathtt{r1} \mapsto \mathsf{Irregular}\ \mathtt{S}_{xss}\ \mathtt{F}_{xss}\ \mathtt{B}_{xss}\ \mathtt{D}_{xss} \\ ,\ & \mathtt{r2} \mapsto \mathsf{Regular}\ \mathtt{reds} \\ ,\ & \mathtt{r3} \mapsto \mathsf{Regular}\ \mathtt{res}]\end{aligned}$$

When returning from `mango` we lookup the result `mango_res` in Σ and use the tag to get the representation of the result via a lookup to $\mathcal{E}_2$ which in this case is `res`, the result of the last statement of `mango`.

You might be wondering where `mango_res` and `res` came from as they're nowhere to be found in listing 3.11. This is due to another subtle difference between the Futhark source language and IR: in the source language, function bodies are just expressions but in the IR a function body is a series of statements (let bindings) as well as a list of sub-expressions which are the results of the body. This means that we have to bind the final statement of the $\texttt{mango}^L$ function to a variable (`res` in this case) and then return that variable as the result of the function body.

Consider now a function body in which the result is a constant c:

```
let {x : i32} = ...
let {y : i32} = ...
...
in {c}
```

If we were to lift this function body, we'd distribute a map over each of the statements and flatten them:

```
let {xs : i32} = 𝓕(map (\v -> ...) ...)
let {ys : i32} = 𝓕(map (\v -> ...) ...)
...
in {c}
```

but notice that we are not distributing the map over the result. If the result was not a constant but instead a variable, we would do a lookup in the distribution inputs and distribution environment and return the appropriate value(s), but in the case of a constant we have to manually replicate it such that it fits the dimension `w` of the outer map:

```
let {xs : i32} = 𝓕(map (\v -> ...) ...)
let {ys : i32} = 𝓕(map (\v -> ...) ...)
...
let {c_lifted : i32} = replicate w c
in {c_lifted}
```

This is related to a similar problem that we have ignored until now, which is that of free variables when flattening of expressions. Here, a free variable is one that is not bound to the lambda of the map we are currently flattening. If we're flattening an expression

$$\mathcal{F}\big(\texttt{map (}\lambda\texttt{x ... -> } e\texttt{) xs ...}\big)$$

and we encounter a free variable `v` in e, we move `v` outside the map, replicate it with the size `w` of the map and finally add it as an argument to the map:

$$\texttt{let vs = replicate w v in } \mathcal{F}\big(\texttt{map (}\lambda\texttt{x v ... -> } e\texttt{) xs vs ...}\big)$$

If we revise the example in listing 3.13 on page 25, we see that this is exactly the same we do when are flattening a function call with a constant is its argument.

Another point that we glossed over in the flattening chapter was size types. As we have already mentioned, the type of an array in Futhark includes its size. This also means that, if we want to pass an array to a function, we need to include its size as a parameter to the function. We have actually already done this in the Function Lifting section, where we add `w` — the size of the outer map — as a parameter when lifting a function. What we are missing is including the length of the data array when passing the representation of an irregular array. When returning (the representation of) an irregular array we also need to return the length of the flat data array, and include an *existential size*[2] in the type of the flat data array.

To summarize, the process of lifting the signature of a function `f` is:

- Add a parameter `w` of type `i64` which is the size of the outer map.
- For each parameter of `f`:
  - If the parameter is a scalar type τ, make it an array of type `[w]`τ.
  - If the parameter is an array type with elements of type τ, add five parameters to the lifted function signature: The number of elements, of type `i64`, and the segments-, flags-, offsets- and data array. The segments- and offsets arrays are both of type `[w]i64`, the flags arrays is of type `[num_elems]bool` and the data array is of type `[num_elems]`τ.
- For each result of `f`, lift the return type in a similar manner to lifting the parameters.

There is also the matter of the uniqueness of the parameters and return types. When lifting scalar parameters / return types we mark the lifted array as “Non-Unique” as it is impossible to consume a scalar parameter, and it is therefore also impossible for the array to be consumed. When lifting array parameters / return types, we mark the segment-, flag- and offset arrays as “Non-Unique” as we know we won’t be doing any in-place modifications on them, and we then assign the uniqueness of the original array as the uniqueness of the flat data array.

The functions for lifting the parameters and return types of a function are shown in listings A.1 and A.2.

We will now again return to the example of the `mango` function on page 22, and show the proper lifting transformation of its signature. We will not show lifting the body as it is pretty much the same as in listing 3.11. The first step is translating the Futhark source-level signature to one that resembles the IR, which means we have to add the implicit length of the array `xs` as a parameter:

```
fun mango (d : i64, n : i32, xs : [d]i32) : {i32}
```

We now apply the rules for lifting a function signature: First we add the parameter for the size of the outer map `w`. Then, we lift the `d` parameter to an array of length `w`, and we do the same for `n`. As `xs` is an array, we add the number of elements $\texttt{num\_elems}_{xs}$, segment array $\texttt{S}_{xs}$, flag array $\texttt{F}_{xs}$, offset array $\texttt{B}_{xs}$, and flat data array $\texttt{D}_{xs}$ as parameters. Finally, as the return type is a scalar, we lift it to an array of length `w`.

[2]See https://futhark.readthedocs.io/en/stable/language-reference.html#size-types

```
mango_lifted (w   :      i64,
               ds  :  [w]i64,
               ns  :  [w]i32,
               num_elems_xs  : i64,
               S_xs  :  [w]i64,
               F_xs  :  [num_elems_xs]bool,
               B_xs  :  [w]i64,
               D_xs  :  [num_elems_xs]i32
              ) : {[w]i32}
```

As the Futhark IR is a first-order language, we can't actually use the `partitionK` function shown in listing 3.9 as it has a functional parameter `p : i64 -> i64`. In the implementation, the `partitionK` function takes three parameters:

`n` The length of the `cs` array.

`k` The number of partitions to split `cs` into

`cs` An array of the equivalence classes.

The call `partitionK` *k* `(λi -> cls[i]) (iota (length cls))` from listing 3.10 would then instead simply be `partitionK (length cls)` *k* `cls` in the implementation. All of the caveats that we have mentioned when lifting function bodies also apply to lifting the bodies of the branches of a `match`-expressions e.g. we need to replicate constants if they're the results of a branch.

5. Evaluation

5.1. Tests

In this section we cover the testing we have done to verify our implementation. Futhark comes with a built-in testing tool which makes it easy to make simple unit tests. Our tests are split in two categories: Tests of function lifting and tests of flattening `match`-expressions. Of course by virtue of using functions, the `match`-expression tests' validity hinges on the correctness of function lifting.

5.1.1. Testing Function Lifting

With function lifting, we test cases such as functions that have both regular and irregular parameters and return types, functions that are called with free variables or constants as their arguments or return free variables or constants as their results, to name a few.

The test cases are not "pretty" and usually just do some bogus computation, but what is important is not the actual contents of the functions, but instead the boundary between calling / returning from a function. We also have to dance around the Futhark compiler to avoid it optimizing out the parts we are interested in. Specifically, we add a `#[noinline]` attributes to our functions, but even then the Futhark compiler will always inline the first function we are calling from `main`.

As an example, the `func_mix.fut` test program is shown in listing 5.1. When compiled, `foo` gets inlined and only `bar` is lifted.

Listing 5.1: Function with both scalar- and array parameters / return types.

```
#[noinline]
let bar (y : i64) (xs : []i64) : ([]i64, i64) =
  let z = y * reduce (+) 0 xs
  in (iota z, z)

#[noinline]
let foo (a : i64) (b : i64) =
  let xs      = iota a
  let (ys, z) = bar b xs
  in reduce (+) 0 ys - z

def main (as : []i64) (bs : []i64) = map2 foo as bs
```

We give a short description of the different tests we have made in table 5.1. The test cases are all given a few inputs that are compared against a known output. The actual tests can be found at[1].

Test filename	Description
`func_simple.fut`	Tests lifting a simple function with no inner parallelism and only scalar parameters and return types.
`func_irreg_input.fut`	Tests lifting a function with an array parameter.
`func_irreg_result.fut`	Tests lifting a function with an array return type.
`func_fully_irreg.fut`	Tests lifting a function with an array parameter and return type.
`func_mix.fut`	Tests lifting a function with both scalar and array parameters and return types.
`func_mix_nested.fut`	Like `func_mix.fut` but chains another function call such that one lifted function calls another lifted function.
`func_const.fut`	Tests lifting a function that both receives as argument and returns a constant.
`func_free.fut`	Tests lifting a function that both receives as argument and returns a free variable.
`func_irreg_update.fut`	Tests lifting a function that consumes its argument.

Table 5.1.: Summary of tests of function lifting

There are 9 test files with a total of 20 unit tests. The tests were run with the OpenCL and CUDA backend, and in both cases 20/20 test cases passed.

5.1.2. Testing Flattening Match-Expressions

When testing flattening of match-expressions we took a similar approach, testing different combination of regular and irregular inputs and results. While most of the tests use `if-then-else`-expressions, they are just syntactic sugar for a `match`-expression with two branches.

We give a short description of the different tests in table 5.2. The actual tests can be found at[2].

There are 5 test files with a total of 21 unit tests. The tests were run with the OpenCL and CUDA backend, and in both cases 21/21 test cases passed.

[1]https://github.com/diku-dk/futhark/blob/e49203a0b13139bb983f62c39363521c2b65c8e7/tests/flattening/function-lifting/

[2]https://github.com/diku-dk/futhark/blob/e49203a0b13139bb983f62c39363521c2b65c8e7/tests/flattening/match-case/

Test filename	Description
`if.fut`	Tests flattening of simple `if`-expression with scalar inputs and results.
`if_irreg_input.fut`	Tests flattening of `if`-expression that takes an array as input to one of its branches.
`if_irreg_result.fut`	Tests flattening of `if`-expression whose branches' results are arrays.
`if_fully_irreg.fut`	Tests flattening of `if`-expression whose branches' results and inputs are a mix of scalars and arrays.
`match_fully_irreg.fut`	Tests flattening of `match`-expression whose branches' results and inputs are a mix of scalars and arrays.

Table 5.2.: Summary of tests of `match`-expressions

5.2. Benchmarks

In chapter 3 we mentioned that it was possible to translate any `match`-expression into a series of nested `if-then-else`-expressions, but we claimed that it was not efficient, and we therefore introduced a flattening rule for general `match`-expressions. To verify that this claim is actually true, and to what extent, we benchmark two semantically equivalent programs with n branches, one written as a single `match`-expression and the other as nested `if-then-else`-expressions. We do this for $n = 2, 4, 8, 16, 32$ and 64 branches.

An example for $n = 4$ branches is shown in listings 5.2 and 5.3 for the nested `if` and single `match` version respectively. The other versions follow the same pattern but with a different number of branches. To make sure that the branches that match are distributed equally, we take the modulus n of the input argument `z` before comparing.

Futhark comes with a utility `futhark bench`, which we used to run the benchmarks. The benchmarks were run with the CUDA backend on a NVIDIA A100 GPU.

The benchmark results are shown in fig. 5.1. As can be seen, for all cases except $n = 2$ branches, the `match`-version runs faster than the `if`-version. It is to be expected that the two versions with $n = 2$ branches have similar performance as a single `if-then-else`-expression is simply translated to a `match`-expression with two branches.

The speedups of the `match`-versions over the `if`-versions is shown in fig. 5.2. It appears that the speedup increases as the number of branches increase, reaching up to 5x at best, but as the input gets larger the speedups seem to converge at around 1.7x.

Listing 5.2: Nested `if`-expression with four branches.

```
def f (z : u8) : u8 = let x = z % 4 in
                       if x == 0
                       then 1
                       else if x == 1
                       then 2
                       else if x == 2
                       then 3
                       else 4

def main [n] (xs : [n]u8) = map f xs
```

Listing 5.3: Single `match`-expression with four branches.

```
def f (z : u8) : u8 = let x = z % 4 in
  match x case 0 -> 1
          case 1 -> 2
          case 2 -> 3
          case _ -> 4

def main [n] (xs : [n]u8) = map f xs
```

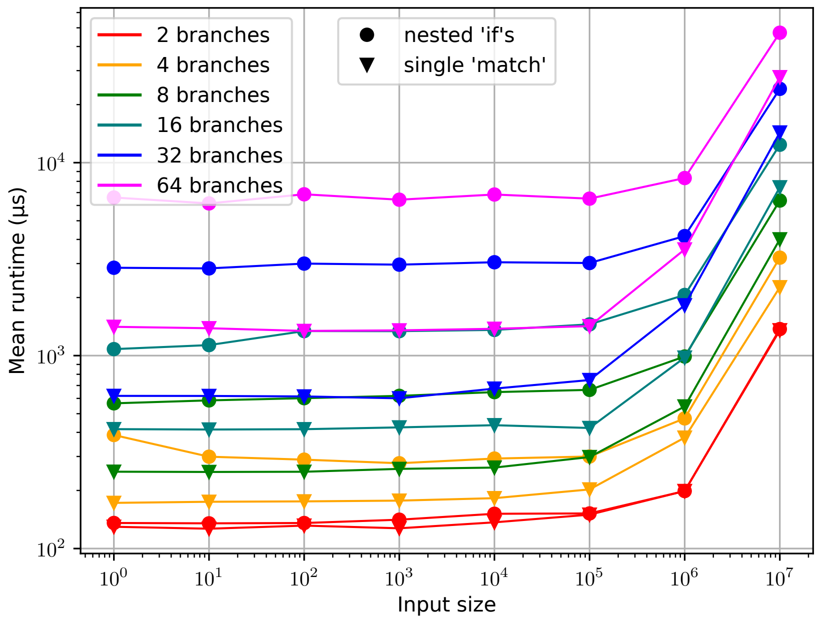


Figure 5.1.: Benchmarking nested `ifs` vs flat `match` expressions.

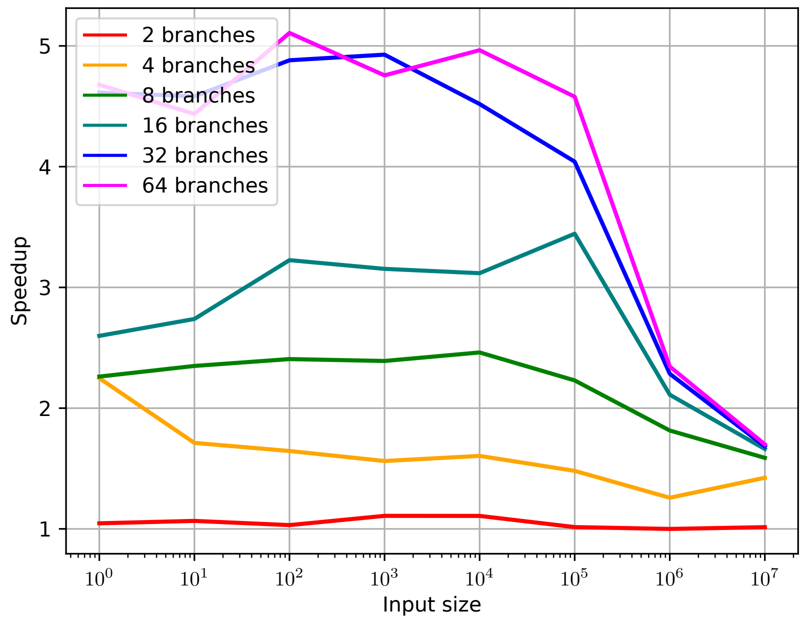


Figure 5.2.: Speedups of single `match`-expression over nested `ifs`.

6. Conclusion

This project has two goals: To implement flattening of `match`-expressions and implement function lifting in the Futhark compiler. We initially give an overview of flattening as a concept , previous work related to flattening, and show a handful of existing flattening rules. We then present a new rule for flattening `match`-expressions as a generalization of flattening `if-then-else`-expressions and theorize that it is more efficient than flattening nested `if-then-else`-expressions. We also cover function lifting and show how it can be used to flatten function calls nested inside maps.

We explain how the theoretical model of flattening translates to the actual implementation that operates on the Futhark internal representation and present an implementation of both flattening of `match`-expressions and function lifting.

Our implementation of both flattening of `match`-expressions and function lifting has been validated with a set of tests, all of which pass. We also confirm our hypothesis that flattening `match`-expressions indeed is faster by benchmarking otherwise equivalent programs using nested `if`-expressions and a single `match`-expression. Our benchmarks show speedups up to ∼5x, but for larger inputs we only gain a modest speedup of ∼1.7x.

7. Future Work

The irregular flattening part of the Futhark compiler is very much under construction, and our implementation of flattening `match`-expressions and function lifting is only a small part of it. While the tests we did write all pass, there are surely edge cases that we have not covered which would likely reveal bugs.

There are other Futhark SOACs and other constructs for which flattening has not been implemented yet. A natural next step would be to implement flattening of the `loop` control structure, which is Futharks alternative to recursive functions. In theory, function lifting should also allow for the possibility of having recursive functions in Futhark.

Here we present some possible improvements that are more directly related to this project.

7.1. Improving PartitionK

The implementation of the `partitionK` function is not very efficient. The `cls_flags` array is a $k \times n$ array of `i64`s, where n is the number of elements we are partitioning, which can consume a lot of memory when flattening a `match`-expression with many branches. A potential improvement to the memory use and likely also runtime of the `partitionK` function would be to implement it with a sorting function such as radix-sort, which has been shown to work well on parallel hardware [BS90, p. 3].

7.2. Incremental Flattening

Full flattening can oversaturate the availabel parallelism of the hardware if the operation which is preformed is so light that the overhead of spawning new threads exceeds the cost of doing the operations sequentially on fewer threads. Additionally, fully flattening a program can destroy opportunities for optimizing locality [Oan18a].

Incremental flattening [Hen+19] is a technique in which the optimal amount of flattening is chosen by compiling different versions of the same code that are at different levels of flattening, and then statistically choosing the version that best utilizes the parallelism available at runtime. Futhark already has incremental flattening for regular nested parallelism, but the current version of irregular flattening in the Futhark compiler performs full flattening.

7.3. The Replication Problem

When flattening an expression that contains free variables, those free variables must be hoisted out of the map and replicated. This is not a problem if the variables are scalars, but if they are large arrays this can incur a large memory and runtime cost. There have been numerous efforts to mitigate this [Ble95, p. 57][PPW95]. One way to handle the replication problem is to instead index indirectly trough the segment indices [Oan22, slide 59]. Another solution called *generalized segment descriptors* let the segments of the irregular representation of an array overlap [Mad12]. Further work is required to explore how the Futhark compiler could solve the replication problem.

A. Code Listings

Listing A.1: Function for lifting the return types of a function

```
liftRetType :: SubExp -> [RetType SOACS] -> [RetType GPU]
liftRetType w = concat . snd . L.mapAccumL liftType 0
  where
    liftType i rettype =
      let lifted = case rettype of
            Prim pt ->
              pure $ arrayOf (Prim pt)
                             (Shape [Free w])
                             Nonunique
            Array pt _ u ->
              let num_elems = Prim int64
                  segs = arrayOf (Prim int64)
                                 (Shape [Free w])
                                 Nonunique
                  flags = arrayOf (Prim Bool)
                                  (Shape [Ext i :: Ext SubExp])
                                  Nonunique
                  offsets = arrayOf (Prim int64)
                                    (Shape [Free w])
                                    Nonunique
                  elems = arrayOf (Prim pt)
                                  (Shape [Ext i :: Ext SubExp])
                                  u
               in [num_elems, segs, flags, offsets, elems]
       in (i + length lifted, lifted)
```

Listing A.2: Function for lifting a parameter of a function

```
liftParam :: SubExp
          -> FParam SOACS
          -> PassM ([FParam GPU], ResRep)
liftParam w fparam =
  case declTypeOf fparam of
    Prim pt -> do
      p <-
        newParam (desc <> "_lifted")
                 (arrayOf (Prim pt) (Shape [w]) Nonunique)
      pure ([p], Regular $ paramName p)
    Array pt _ u -> do
      num_elems <-
        newParam (desc <> "_num_elems") $ Prim int64
      segments <-
        newParam (desc <> "_segments") $
          arrayOf (Prim int64) (Shape [w]) Nonunique
      flags <-
        newParam (desc <> "_flags") $
          arrayOf (Prim Bool)
                  (Shape [Var (paramName num_elems)])
                  Nonunique
      offsets <-
        newParam (desc <> "_offsets") $
          arrayOf (Prim int64) (Shape [w]) Nonunique
      elems <-
        newParam (desc <> "_elems") $
          arrayOf (Prim pt)
                  (Shape [Var (paramName num_elems)])
                  u
      pure
        ( [num_elems, segments, flags, offsets, elems],
          Irregular $ IrregularRep
              { irregularSegments = paramName segments,
                irregularFlags = paramName flags,
                irregularOffsets = paramName offsets,
                irregularElems = paramName elems
              }
        )
  where
    desc = baseString (paramName fparam)
```

Bibliography

Blelloch, Guy E. *NESL: A nested data-parallel language.(version 3.1)*. Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 1995.

— *Vector models for data-parallel computing*. Vol. 2. Citeseer, 1990.

Blelloch, Guy E. and Gary W. Sabot. "Compiling collection-oriented languages onto massively parallel computers". In: *Journal of parallel and distributed computing* 8.2 (1990), pp. 119–134.

Blelloch, Guy E. et al. "Implementation of a portable nested data-parallel language". In: *ACM Sigplan Notices* 28.7 (1993), pp. 102–111.

DIKU. *Futhark User's Guide*. Version 0.24.3. Apr. 23, 2023. URL: `https://futhark.readthedocs.io/_/downloads/en/v0.24.3/pdf/`.

Dubois, Michel, Murali Annavaram, and Per Stenström. *Parallel computer organization and design*. Cambridge University Press, 2012.

Henriksen, Troels. "Design and Implementation of the Futhark Programming Language". PhD thesis. Universitetsparken 5, 2100 København: University of Copenhagen, Nov. 2017.

Henriksen, Troels et al. "Futhark: purely functional GPU-programming with nested parallelism and in-place array updates". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pp. 556–571.

Henriksen, Troels et al. "Incremental flattening for nested data parallelism". In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 2019, pp. 53–67.

Hillis, W Daniel and Guy L Steele Jr. "Data parallel algorithms". In: *Communications of the ACM* 29.12 (1986), pp. 1170–1183.

Madsen, Frederik M. *Flattening Nested Data Parallelism*. 2012. URL: `http://hjemmesider.diku.dk/~fmma/publications/nested.pdf` (visited on 05/25/2023).

Moore, Gordon E. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pp. 33–35.

Oancea, Cosmin E. *Flattening Irregular Nested Parallelism. DPP Lecture Slides*. 2022. URL: `https://raw.githubusercontent.com/diku-dk/dpp-e2022-pub/main/slides/L5-irreg-flattening.pdf` (visited on 05/29/2023).

— *Lecture Notes for the Software Track of the PMPH Course*. 2018. URL: `https://raw.githubusercontent.com/diku-dk/dpp-e2022-pub/main/material/flattening/lecture-notes-pmph.pdf` (visited on 05/25/2023).

— *Machine Code Generation. IPS Lecture Slides*. 2018. URL: `http://hjemmesider.diku.dk/~torbenm/ICD/IPS-lecture-slides.zip` (visited on 05/29/2023).

Palmer, D.W., J.F. Prins, and S. Westfold. “Work-efficient nested data-parallelism”. In: *Proceedings Frontiers ’95. The Fifth Symposium on the Frontiers of Massively Parallel Computation*. 1995, pp. 186–193. DOI: 10.1109/FMPC.1995.380449.

Peyton Jones, Simon et al. “Harnessing the multicores: Nested data parallelism in Haskell”. In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2008.

Prins, Jan F and Daniel W Palmer. “Transforming high-level data-parallel programs into vector operations”. In: *ACM SIGPLAN Notices* 28.7 (1993), pp. 119–128.

Schwartz, Jacob T. “Ultracomputers”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2.4 (1980), pp. 484–521.