UNIVERSITY OF COPENHAGEN

Master's Thesis in Computer Science

Lotte Bruun
Ulrik Larsen

Reverse Automatic Differentiation in Futhark

May 2022

Supervised by Cosmin Oancea

Lotte Maria Bruun
Ulrik Stuhr Larsen
Reverse Automatic Differentiation in Futhark
Master's thesis at University of Copenhagen, May 2022
Supervisor: Cosmin Eugen Oancea

University of Copenhagen
Faculty of Science
Department of Computer Science
Universitetsparken 1
2100 Copenhagen

Abstract

In this thesis, we describe our work with reverse automatic differentiation (AD) in the data-parallel programming language Futhark. The main motivation is to extend and optimise reverse AD in Futhark's compiler to permit more expressive programs to by differentiated automatically and efficiently.

Futhark's AD consists of a set of rewrite rules that are used to transform a program to its differentiated counterpart. We present reverse mode AD rewrite rules for the operations *reduce-by-index* and *scan*. Reduce-by-index, also known as multi-reduce, has a generic cases and multiple special cases, of which the latter are loosely described by Schenck et al. 2022. We formulate and present a rewrite rule for the generic case and present specific rewrite rules for the special cases as Futhark pseudo-code.

Likewise, we examine the reverse AD rewrite rules for scan presented by Schenck et al. 2022, one of which we have simplified with a performance benefit. The existing AD implementation is modified to work when the scan operates on tuples. We have extended the generic case with specialised rewrite rules for scan operators whose Jacobian matches specific patterns.

We have implemented the presented rewrite rules of both reduce-by-index and scan in Futhark's compiler. The performance of differentiated programs is evaluated experimentally and compared to its primal program performance and the program differentiated with forward AD instead. In many case, this demonstrates reasonable reverse AD overheads and competitive performance to Futhark's established forward AD implementation.

Contents

Introduction

1

Differentiation is an essential tool for countless tasks. The subject of efficient differentiation methods remains relevant, especially to computer scientists where machine learning often rely heavily on differentiation (Baydin et al. 2018). ML algorithms typically take millions of data points as parameters, making efficient and precise differentiation vital but tedious to compute by hand. One solution is to use automatic differentiation (AD) which is a method that has only caught the attention of the machine learning community a couple of years ago, and it continues to provide an opportunity for expansion (Domke 2009).

ML workloads are run on (clusters of) GPUs so it is essential that AD is implemented in high-level languages with efficient mappings to GPU hardware. *Futhark* is one such language (Henriksen, Serup, et al. 2017, Elsman et al. 2018a, Henriksen 2017). AD has already partly been implemented in the Futhark compiler but mostly the forward mode which is not optimal for most ML tasks. Reverse mode AD is certainly preferable for these tasks but the implementation has yet to be completed. This project aims to examine, extend and optimise parts of the reverse mode AD implementation of Futhark.

Multiple methods of differentiation already exist, so why the interest in automatic differentiation especially? Its advantages appear clearly, when we take a look at AD in comparison to the other methods of derivative computations. We can discriminate between the following differentiation method categories: manual, numerical, symbolic and automatic (Baydin et al. 2018). Figure 1.1 shows how a function might be differentiated by each method category. The first and simplest category is that of manual differentiation (example figure 1.1a). This method might be easy for small, simple expressions but would be infeasible for large programs. It might take a long time and is prone to errors. As programmers we would much rather take the lazy approach and let a computer do the work for us, even when possibly accepting a certain dynamic overhead in comparison with hand-optimised code.

The next two methods, numerical and symbolic, could be confused for variants of automatic differentiation but are actually separate methods. Numerical differentiation relies on the computation of finite differences in sample points from the original function and computes a numeric derivative (example figure 1.1b). Numerical methods come with the disadvantage that

Program P and the corresponding mathematical expression $f(x)$:

```
def f x =
  let a = x * x
  in a + 10
```

$$f(x) = x^2 + 10$$

$$\begin{tikzcd}
f(x) = x^2 + 10 \arrow[d, dashed, "\textit{Mental calculations}"] \\
\frac{d}{dx} f(x) = 2x
\end{tikzcd}$$

(a) Manual differentiation

$$\frac{d}{dx} f(x) \approx \frac{f(x+h) - f(x)}{h}$$

(b) Numerical differentiation with step size $h > 0$

Derivation rules:

$$\frac{d}{dx}(h(x) + g(x)) \rightsquigarrow \frac{d}{dx}h(x) + \frac{d}{dx}g(x)$$

$$\frac{d}{dx}c \rightsquigarrow 0 \qquad \frac{d}{dx}x^n \rightsquigarrow nx^{n-1}$$

Derivative of $f(x)$:

$$\frac{d}{dx} f(x) = \frac{d}{dx}(x^2 + 10) = 2x$$

(c) Symbolic differentiation

```
def f' x ẋ =
  let a = x * x
  let ȧ = 2*x * ẋ
  let res = a + 10
  let rės = ȧ
  in rės
```

(d) Automatic differentiation (here forward mode). ẋ is the tangent of variable x

Figure 1.1: Examples of the four categories of differentiation computation

they only compute an approximation and not an exact derivative.

Symbolic differentiation fixes this problem by automatically manipulating expressions to its derivatives (example figure 1.1c). It gradually transforms the input expression using derivation rules to finally reveal an exact derivative expression. However, the runtime may be bad since expressions might grow exponentially large compared to the original function, also known as expression swell (Baydin et al. 2018, p. 7). This problem is caused by unnecessary recomputation of duplicate subexpressions in the derivative. It also requires the input expression to be closed-form which limits the expressiveness of algorithms greatly.

While numeric and symbolic methods are both automatic in their computation, they each diverge from AD on crucial points. Automatic differentiation covers methods relying on the accumulation of values through code execution resulting in a numerical derivative (example figure 1.1d). Thus, AD uses parts from both these methods: It uses mathematical derivation rules like symbolic differentiation; but returns a numerical derivative like numerical differentiation (Griewank 2003). AD delivers a derivative at machine precision without expression swell as intermediate variable derivatives are accumulated and reused.

The aim of this thesis is to extend the reverse mode AD implementation in the Futhark compiler, specifically *reduce-by-index* and *scan*. These operations will be presented as they are used (chapter 4 and chapter 5). We use the rewrite rules presented by Schenck et al. 2022 as a starting point and extend and optimise them in accordance with the original semantics. During the development of the implementation, the focus has been on performance concerns, specifically analytical considerations of performance such as asymptotic work-depth analysis and examination of constant factors based on Futhark's compiler transformations, e.g. fusion of GPU parallel constructs. For evaluation, we compare the runtime observations with Futhark's forward mode AD implementation and our expected performance calculations based on AD theory and GPU behaviour. Our main contributions are implementations of reverse AD for *reduce-by-index* and *scan*. The specific contributions of this thesis are:

- A reverse mode AD rewrite rule for reduce-by-index with generic case operators which preserves the expected work-depth asymptotics of the primal program.
- A formal representation of the rewrite rules reduce-by-index with special case operators: min/max, addition and multiplication.
- A systematic derivation of the rewrite rule for scan (with an arbitrary operator), inspired from Schenck et al. 2022, that has allowed us to perform some simplifications on the initial rule.
- Analysis for optimising the re-write rules of scan based on statically-reasoned sparsity of intermediate results.
- Implementation of reverse mode AD of reduce-by-index with generic and special cases in the Futhark compiler.
- Extending the implementation of reverse mode AD for scan with tuple operators and special case operators in the Futhark compiler.
- An experimental evaluation of differentiated programs using reduce-by-index, demonstrating (i) reasonable overheads compared to the primal programs for special cases, (ii) significant speedups on computation of the full Jacobian compared to forward mode AD in Futhark, and (iii) that a significant performance bottleneck is due to Futhark not supporting a GPU-efficient Radix sort implementation.

- An experimental evaluation of differentiated programs using scan, demonstrating (i) competitive performance with forward mode AD when using special case operators, (ii) up to 3× speedup of the differentiated program when exploiting a Jacobian pattern. The exploitation of sparse Jacobian patterns enables scans with operators working on tuples containing more than 4 elements, which runs out of shared memory with the unoptimized rewrite rule; it would likely result in order-of-magnitude speedups otherwise.

This thesis consists of three main parts: *Methods and Materials* (part I), *Implementation* (part II) and *Evaluation* (part III). *Methods and Materials* presents relevant parts of the Futhark language and internal representation (chapter 2), the theoretical background of AD including forward and reverse mode (chapter 3), and the previous work with regards to reverse AD of reduce-by-index and scan (chapter 4 and chapter 5). Mind that the first part of the Futhark chapter (section 2.1) is meant as an index table for semantics of Futhark language constructs as well as an introduction to the language. *Implementation* includes the details of our contributions where for the generic and special cases of reduce-by-index and scan, we present a high-level strategy based in the theory, a rewrite rule as generated Futhark-esque pseudocode, a primitive analysis of generated internal representation, and an asymptotic work-depth analysis. In *Evaluation*, we consider validation and performance of our implementation (chapter 8 and chapter 9). The validation chapter presents our testing strategy which includes manually derived tests and tests with random input that validates our implementation against forward mode AD. In the performance chapter, we present and analyse the benchmarks of our implementation and assess it in comparison with primal programs and forward mode AD. Lastly, the conclusion and future work sections are to be found in chapter 10 and chapter 11 respectively.

Part I

Methods and Materials

Futhark

2

A goal of this thesis is to expand the AD implementation in the Futhark compiler, specifically the reverse mode. Section 2.1 explains relevant parts of Futhark syntax and semantics as well as relevant parts of the internal representation of the language.

2.1 The Futhark Language

Futhark is a pure functional programming language designed for efficient general purpose GPU (GPGPU) code (Henriksen, Serup, et al. 2017). In pure functional languages, program statements are side effect free. This is a desirable property for AD since side effects complicate the intermediate adjoint/tangent computations.

A Futhark function generally consists of a sequence of let-bindings:

```
let v_1 = ...
...
let v_k = ... in v_k
```

As Futhark is side effect free, variables cannot be modified after assignment but can be overshadowed. Notice that in-place updates are disallowed in the form they exist in imperative programs like C (e.g. `ys[i] = x`). We can access the same semantics of in-place updates by:

```
let ys = xs with [i] = x
```

Semantically, `ys` is a copy of `xs` where index i is updated to `x`. Operationally, no copying is performed and essentially the i'th element of `xs` is overwritten with the new value `x`, thus preserving the cost of the imperative update. `xs` is *consumed* by the operation which means it is illegal for subsequent statements to reference that array. Consumption of the array is required to preserve a side effect free language. This is implemented by a uniqueness type mechanism that essentially type checks that `xs` is not reachable by any program statement following the update (Henriksen, Serup, et al. 2017).
Additionally, Futhark supports a syntactic sugar resembling the imperative style notation more closely by `let xs[i] = x` but this still creates a new variable overshadowing `xs`.

Mind that if we have a statement `let x = as` where `as` is an array, `x` is an alias of `as`, i.e. they reference the same memory. Alternatively, we can use `copy` such that `let x = copy as`. In this statement, the contents of `as` are copied to a new memory space which `xs` references. This prevents `xs` and `as` from becoming aliases but introduces the overhead of reading and writing the entirety of `as`.

Futhark has loops as well even though they are inherently an imperative style construct. For this project only the Futhark for-loop is relevant:

```
loop acc = init for i in is do
  body
```

`acc` is an accumulator that is set to an initial expression `init` in the loop header and after each iteration `acc` is bound to the result of that iteration. `i` is in iterator which is taken from the array `is`. Notice that such a loop is sequentially executed.

Some common Futhark functions are explained in table 2.1. A $*$ in an input parameter type means that the function consumes that parameter, i.e. any use of that actual parameter is illegal after the program point of the call. A $*$ in the output type means that the result is `unique`, which means the result is guaranteed to not alias any non-unique parameters.

2.1.1 Second-Order Array Combinators

Futhark features second-order array combinators (SOACs). These are data-parallel operations executed by the GPU. Table 2.2 shows the semantics of the SOACs that are relevant to our project.

The figure includes the work-depth asymptotics of the SOACs in Futhark. In a classic asymptotic runtime analysis, only the number of operations is taken into account but the runtime of parallel programs depends heavily on the amount of parallelism. Work-depth analysis provides a method for assessing the efficiency of parallel programs (Shiloach et al. 1982). The analysis consists of two measures: work complexity and depth. Work complexity is the amount of executed operations and depth is the required number of sequential steps. Ideally, a parallel program should be *work efficient*, i.e. its asymptotic work is equal to that of an optimal sequential implementation. Below we give a brief explanation of the semantics of each SOAC.

Map: apply a function to every element of an input array. You can also map over multiple arrays at once, e.g. `map3` maps over 3 arrays with a three-ary function. Since the function applications are independent, a `map` can be fully parallelised with a depth of $O(d_f)$ (depth of a single function application).

Type and Semantics	Explanation

`iota : (n: i64) → *[n]i64`
`iota n ≡ [0..n-1]`
	Returns an array with the numbers from 0 to $n-1$.

`replicate : (n: i64) → α → *[n]α`
`replicate n x ≡ [x,...,x]`
	Returns an array with n copies of `x`.

`zip : [n]α → [n]β → [n](α,β)` `zip a b ≡` $[(a_0,b_0),(a_1,b_1),\ldots,(a_{n-1},b_{n-1})]$	Returns an array where the i'th element is `a[i]` tupled with `b[i]`.

`unzip : [n](α,β) → ([n]α,[n]β)`
`unzip xs ≡`
`(`$[xs_0.0, xs_1.0, \ldots, xs_{n-1}.0]$`,`
$[xs_0.1, xs_1.1, \ldots, xs_{n-1}.1]$`)`
	Takes an array of 2-ary tuples and returns a tuple with two arrays where the first one holds the initial indices of element tuples and the second array holds the second indices of the element tuples.

`reverse : [n]α → [n]α` `reverse xs ≡` $[xs_{n-1}, xs_{n-2}, \ldots, xs_0]$	Returns the input array in reverse order.

`copy : [n]α → *[n]α`
`copy xs ≡ xs`
	Returns a unique copy of the input array.

Table 2.1: Types and semantics of common Futhark functions. `zip` and `unzip` also exist for 3-ary, 4-ary and 5-ary tuples by appending the arity to the function name, e.g. `zip3`.

Type and Semantics	Work	Depth
map : $(\alpha \to \beta) \to$ [n]$\alpha \to$ [n]β map f [$x_0,\ldots,x_{n-1}$] $\equiv$ [f $x_0,\ldots,$ f x_{n-1}]	$O(w_\mathtt{f} n)$	$O(d_\mathtt{f})$

scatter : $*$[n]$\alpha \to$ [m]i64 $\to$ [m]$\alpha \to *$[n]α
scatter dst is vs $\equiv$
  for i in [0..m-1] do
    dst[is[i]] = vs[i]
	$O(m)$	$O(1)$

reduce : $(\alpha \to \alpha \to \alpha) \to \alpha \to$ [n]$\alpha \to \alpha$ reduce $\odot$ e [$x_0,\ldots,x_{n-1}$] $\equiv$ e $\odot$ x_1 $\odot$ ... $\odot$ x_{n-1}	$O(wn)$	$O(w\log(n))$
scan : $(\alpha \to \alpha \to \alpha) \to \alpha \to$ [n]$\alpha \to$ [n]α scan $\odot$ e [$x_0,\ldots,x_{n-1}$] $\equiv$ [$x_0, x_0 \odot x_1, \ldots, x_0 \odot \cdots \odot x_{n-1}$]	$O(wn)$	$O(w\log(n))$
seg_scan : $(\alpha \to \alpha \to \alpha) \to$ $\alpha \to$ [n]bool $\to$ [n]$\alpha \to$ [n]α seg_scan $\odot$ e flags vs $\equiv$ -- with k+1 segments with lengths k_i+1 [$x_{0.0}, x_{0.0} \odot x_{0.1}, \ldots, x_{0.0} \odot \cdots \odot x_{0.k_0}$, $x_{1.0}, x_{1.0} \odot x_{1.1}, \ldots, x_{1.0} \odot \cdots \odot x_{1.k_1}$, $\ldots$, $x_{k.0}, x_{k.0} \odot x_{k.1}, \ldots, x_{k.0} \odot \cdots \odot x_{k.k_k}$]	$O(wn)$	$O(w\log(n))$
reduce_by_index : $*$[m]$\alpha \to (\alpha \to \alpha \to \alpha) \to$ $\alpha \to$ [n]i64 $\to$ [n]$\alpha \to$ $*$[m]α reduce_by_index dst $\odot$ e inds vs $\equiv$ for i in [0..n-1] do dst[inds[i]] $\odot$= vs[i]	$O(wn)$	$O(wn)^\bigstar$

Table 2.2: Types and semantics of relevant Futhark SOACs as well as their asymptotic work-depths (Elsman et al. 2018b). Mapping function f has $w_\mathtt{f}$ work and $d_\mathtt{f}$ depth. Operator $\odot$ has w work. $\bigstar$: In practice, the expected depth is $O(w\log(n))$.

Scatter: make m in-place updates in destination `dst` at indices `is` with values `vs`. The first $*$ in the operation type indicates that the input destination array is consumed. Recall that consumption means the array cannot be legally referenced afterwards. As before, the $*$ in the output type means that the result is nique (has no aliases). Notice that the work do not depend on the length of the destination array only on the number of in-place updates: It simply writes to the needed indices. Illegal indices are ignored, i.e. if the index is negative or out-of-bounds of the destination array. The depth is $O(1)$ because the updates do not depend on each other so `scatter` can be fully parallelised. It requires that no index is updated more than once, as this would result in undeterministic behaviour.

Reduce: accumulate all values of an input array with binary operator $\odot$. The operator needs to be associative so multiple subarrays can be processed in parallel while maintaining deterministic behaviour.

Scan: similar to `reduce` but it returns a list with all the intermediate results of accumulating with associative operator $\odot$. There are two types of `scan`: inclusive and exclusive. Futhark's build-in scan operator is *inclusive* which means the first position is index 0 element x_0, then $x_0 \odot x_1$ and so on up to $x_0 \odot \cdots \odot x_n$ (semantics shown in table 2.2). An *exclusive* scan has the neutral element `e` in first position, then x_0 etc. up to $x_0 \odot \cdots \odot x_{n-1}$. Notice the last element of the input array is not taken into account by the exclusive scan, so the resulting array will have length n using either type of scan. An exclusive scan `scan_exc` can be implemented with little overhead by shifting the elements of the input array one index:

```
scan_exc ⊙ e as =
  scan ⊙ e (map (λi → if i == 0
                      then e
                      else as[i-1])
                (iota n))
```

Segmented scan: scans inside specified segments of an input array (Blelloch 1989). The segments are defined by an n-length flag array which marks the beginning of each new segment by `true` flag and all other flags are `false`. Like single scan, segmented scan can be inclusive or exclusive inside each segment. Mind that segmented scan `seg_scan` is not build into Futhark but it is a common SOAC nonetheless. An inclusive segmented scan can be implemented in Futhark by (Elsman et al. 2018c):

```
seg_scan ⊙ e flags vs =
  let (_, res) = unzip ◁
    scan (λ(x_flag, x) (y_flag, y) →
            let fl = x_flag || y_flag
            let vl = if y_flag then y else x ⊙ y
            in (fl, vl)
          ) (false, ne) (zip flags arr)
  in res
```

Intuitively, this segmented scan implementation might seem excessively complicated. Why not just map over the segments with a scan? The challenge is that Futhark only supports regular arrays, i.e. internal arrays of the same level must be the same length. All segments might not have the same length so an array with the segments as elements would be irregular. It is possible to work with "irregular" structures by e.g. using a single array for all segments. Then a flag array can be used to store the placements of the segments, like in segmented scan.

Reduce-by-index: we will be implementing the reverse AD derivative of `reduce_by_index`. The type and semantics of `reduce_by_index` are explained thouroughly in section 4.3.

The SOACs `reduce`, `scan` and `seg_scan` all have the same work-depth asymptotics. Mind that the depth is $O(w \log(n))$ where w is the work of ⊙ because the operator will be executed sequentially.

2.2 Internal Representation of Futhark

The Futhark compiler is known to aggressively optimise programs to the point where they can hardly be recognised. The process yields relatively efficient data-parallel programs which can be constructed from the convenient perspective of a high-level language. The compiler translates programs to an internal representation (IR) and optimises the program by e.g. removing dead code, rearranging program order and control flow, and *fusing* SOACs.

The latter, fusing, is applied to sequences of SOACs when they take similar input arrays. Here the word *similar* means that the SOACs take the same, partly the same input arrays, or possibly just arrays of the same length. The SOACs are merged to a single kernel which is executed on the GPU. Fusing saves much time since we can remove superfluous kernels call overheads and memory accesses. It is especially important to be mindful of memory accesses since global GPU memory is slow in comparison to accesses registers for example.

The Futhark syntax does not specify which SOACs are fused, so IR uses modified constructs

IR Construct	Type	SOAC
`Map`	$\overbrace{\mathbb{R}}^{\text{input length}} \rightarrow \overbrace{\{\alpha_j^n\}^{j\in 0..i}}^{\text{input arrays}} \rightarrow \overbrace{(\{\alpha_j\}^{j\in 0..i} \rightarrow \{\beta_j\}^{j\in 0..k})}^{\text{map function}} \rightarrow \{\beta_j^n\}^{j\in 0..k}$	`map`
`ScatterOMap`	$\overbrace{\mathbb{R}}^{\text{values length n}} \rightarrow \overbrace{\{\alpha_j^n\}^{j\in 0..i}}^{\text{input arrays}} \rightarrow \overbrace{(\{\alpha_j^n\}^{j\in 0..i} \rightarrow \{\mathbb{R}^n \times \beta_j^n\}^{j\in 0..k})}^{\text{map function}}$ $\rightarrow \overbrace{\{\beta_j^m\}^{j\in 0..k}}^{\text{destinations}} \rightarrow \{\beta_j^m\}^{j\in 0..k}$	`scatter` ∘ `map`
`ScanOMap`	$\overbrace{\mathbb{R}}^{\text{input length n}} \rightarrow \overbrace{\{\alpha_j^n\}^{j\in 0..i}}^{\text{input arrays}} \rightarrow \overbrace{\{(\beta_j \rightarrow \beta_j \rightarrow \beta_j) \times \beta_j\}^{j\in 0..k}}^{\text{scan ops w. neutral elems}}$ $\rightarrow \overbrace{(\{\alpha_j^n\}^{j\in 0..i} \rightarrow \{\beta_j^n\}^{j\in 0..k})}^{\text{map function}} \rightarrow \{\beta_j^n\}^{j\in 0..k}$	`scan` ∘ `map`★
`ReduceOMap`	$\overbrace{\mathbb{R}}^{\text{input length n}} \rightarrow \overbrace{\{\alpha_j^n\}^{j\in 0..i}}^{\text{input arrays}} \rightarrow \overbrace{\{(\beta_j \rightarrow \beta_j \rightarrow \beta_j) \times \beta_j\}^{j\in 0..k}}^{\text{reduce ops w. neutral elem.s}}$ $\rightarrow \overbrace{(\{\alpha_j^n\}^{j\in 0..i} \rightarrow \{\beta_j^n\}^{j\in 0..k})}^{\text{map function}} \rightarrow \{\beta_j\}^{j\in 0..k}$	`reduce` ∘ `map`★
`HistOMap`	$\overbrace{\mathbb{R}}^{\text{values length n}} \rightarrow \overbrace{\{\alpha_j^n\}^{j\in 0..i}}^{\text{input arrays}}$ $\rightarrow \overbrace{\{\overbrace{\mathbb{R}}^{\text{buckets m}} \rightarrow \overbrace{\beta_j}^{\text{neutral elem}} \rightarrow \overbrace{\beta_j^m}^{\text{dest}} \rightarrow \overbrace{(\beta_j \rightarrow \beta_j \rightarrow \beta_j)}^{\text{reduce op.}}\}^{j\in 0..k}}^{\text{histograms}}$ $\rightarrow \overbrace{(\{\alpha_j^n\}^{j\in 0..i} \rightarrow \{\mathbb{R}^n \times \beta_j^n\}^{j\in 0..k})}^{\text{map function}} \rightarrow \{\beta_j^m\}^{j\in 0..k}$	`reduce_by_index` ∘ `map`

Table 2.3: SOAC constructs in the internal representation (IR) and their types. The constructs take $i+1$ input arrays and result in $k+1$ outputs. The notation $\{\mu_j\}^{j\in 0..x}$ denotes a tuple with $x+1$ elements of types $\mu_0 \times \cdots \times \mu_x$. μ^n means an n-length array with element type μ. Type uniqueness information is omitted. ★: the `map`s in `scan` and `reduce` may return additional arrays which are not used by the scan/reduce functions (not shown in figure).

for representing computation of fused SOACS. The relevant SOAC constructs are shown in table 2.3. Consider first the construct Map whose input is a list of arrays and a mapping functions. The compiler merges fusable SOACs into a single mapping function, whose input is a tuple with elements from all input arrays. The fused function returns a tuple with an element for each result array. Let us look at a Futhark example:

```
def main [n] (as: [n]i64) (vs: [n]i64) =
  let a = map2 (*) as vs
  let v = map (*5) vs
  in (a,v)
```

The two maps can be fused since the arrays are the same length and they both map over vs. The generated IR code for this example is:

```
entry_main(n : i64, as : [n]i64, vs : [n]i64)
  : {*[n]i64, *[n]i64} = {
  let {a : [n]i64,
       v : [n]i64} =
    Map(n, {as, vs},
        λ{x1 : i64, x2 : i64}
          : {i64, i64} →
            let {res1 : i64} = mul64(x1, x2)
            let {res2 : i64} = mul64(5i64, x2)
            in {res1, res2})
  in {a,v}
}
```

Mind that for the sake of readability, the code shown above is a beautified version of the IR. Notice there is only one map construct which computes both a and v from the original Futhark program, so the compiler has indeed fused the maps. Notice the compiler will only need to read vs once - if the maps had not been fused, vs would be read twice. This fused map takes 2 input arrays and constructs 2 new arrays but there are no general requirements to the ratio between input arrays and outputs. The map function simply takes elements of the used arrays and outputs a tuple with elements for each resulting array, here a and v[1].

The other constructs shown in table 2.3 are essentially function compositions of some SOAC and a map. ScatterOMap applies a map to some input arrays which computes index and value arrays. It can fuse multiple scatters as long as the value arrays are all the same length. The

[1] The cost model is that fusion should never duplicate computation.

destination arrays need not have the same length.

Similarly, ScanOMap, ReduceOMap and HistOMap are able to fuse multiple scans, reduces and reduce_by_indexs respectively. These fused SOACs may have different operators and neutral elements which are provided to the SOAC construct in an array. As an example, consider the Futhark program:

```
def main [n] (vs: [n]i64) (as: [n]i64) =
  let a = scan (+) 0 (map (*3) as)
  let v = scan (*) 1 vs
  in (a,v)
```

The compiler translates it to:

```
entry_main (n : i64, vs : [n]i64, as : [n]i64)
  : {*[n]i64, *[n]i64} = {
  let {a : [n]i64,
       v : [n]i64} =
    ScanOMap(n,
             {as, vs},
             {λ {x : i64, y : i64}
                : {i64} →
                let {res : i64} = add64(x, y)
                in {res},
              {0i64},
              λ {x : i64, y : i64}
                : {i64} →
                let {res : i64} = mul64(x, y)
                in {res},
              {1i64}},
             λ {ai : i64, vi : i64}
               : {i64,
                  i64} →
               let {res : i64} = mul64(3i64, ai)
               in {res, vi})
  in {a, v}
}
```

This IR code fuses the scans and the map from the Futhark code into one single construct ScanOMap.

Mind, that this aggressive optimisation approach provide some limitations as well as speed. It might be difficult for a programmer to understand how their program is mapped to IR. Print statements do not exist since some optimisations require the language to be pure (Henriksen 2018). This restricts Futhark programmers from e.g. using the common "printf debugging" method. Also arrays of tuples actually do not exist in the IR and are converted to tuples of arrays instead. Thus `zip` and `unzip` are constructs that only exists in the Futhark language but not in the internal representation, i.e. they are syntactic sugar.

The IR also includes unsafe operations which do not exist in Futhark. One such is `Scratch` which allocates an array of some type without initialising the memory. Notice that reading any of these elements results in undeterministic behaviour! However, scratching can be used safely when every element is written to before reading it. In these cases, it is more efficient than writing dummy values which are are overwritten anyhow. However, it is inappropriate to provide unsafe operations to the layman Futhark programmer. Inside the compiler, the developers can guarantee their usages of `Scratch` are safe.

Theory of Automatic Differentiation 3

In this section, we will give the intuition behind automatic differentiation as well as its mathematical foundation. Automatic differentiation is the generation of a program derivative. In its essence, it relies on the rules of differentiation. E.g. a program computing the function $f : \mathbb{R} \to \mathbb{R}, f(x) = x^2$ will have the derivative $f(x) = 2x$ by the rule $\frac{dx^n}{dx} = nx^{n-1}$. The initial challenge is, how do you translate a program to a function that you can differentiate? The derivation rules are defined on mathematical expressions, not program statements. A program defines a function that can be derived according to an input or output. Mind that the program needs to be side-effect free (which is guaranteed for pure functional languages). A program consists of statements which can each be interpreted as a function. Then these functions can be combined with function composition. Thus a program is essentially a composition of functions where function $f(x) = h(g(x)) = (h \circ g)(x)$ corresponds to:

```
def f x =
  let v1 = g x in h v1
```

Notice that in the mathematical expression, the computation order is from right to left, whereas in the program the computation is from left to right.

AD takes a program and transforms it to a differentiated program by applying transformation rules to every program statement. Each statement makes a contribution to the derivative. The idea is to construct derivative statements for all intermediate variables individually. These are accumulated to a final derivative of the output or input. Mind that some statements make a zero contribution, e.g. control flow statements. Intuitively, conditions in control flow statements do not directly affect the computation of the result. Thus the control flow appears unchanged in the differentiated program and just the statement body(s) are transformed.

The derivative contribution of a program statement can be represented as a Jacobian. The Jacobian of a function $f(\mathbf{x})$ is a matrix that holds the derivative in a given input point $\mathbf{x}$. The Jacobian of a differentiable function $f : \mathbb{R}^a \to \mathbb{R}^b$ at point $\mathbf{x} \in \mathbb{R}^a$ is defined as (Baydin et al.

2018, p. 10):

$$J_f(\mathbf{x}) = \left[\frac{\partial f}{\partial x_1}(\mathbf{x}) \quad \cdots \quad \frac{\partial f}{\partial x_a}(\mathbf{x}) \right] = \begin{bmatrix} \nabla^{+T} f_1(\mathbf{x}) \\ \vdots \\ \nabla^{\mathrm{T}} f_b(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}) & \cdots & \frac{\partial f_1}{\partial x_a}(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_b}{\partial x_1}(\mathbf{x}) & \cdots & \frac{\partial f_b}{\partial x_a}(\mathbf{x}) \end{bmatrix}$$

Mind that the entries of the Jacobian are scalars. As mentioned, programs can be viewed as function compositions, where the contribution of each statement is a separate Jacobian. The contributions are combined to a derivative of a program by applying the chain rule to the Jacobians. Remember, the chain rule is used to differentiate composed functions and has the definition $\frac{df(z(x))}{dx} = \frac{df(z(x))}{dz(x)} \cdot \frac{dz(x)}{dx}$. As an example, consider a differentiable function $P : \mathbb{R}^a \to \mathbb{R}^d$ that defines program P. Assume $P(\mathbf{x}) = h(g(k(\mathbf{x})))$ where the sub-functions are $k : \mathbb{R}^a \to \mathbb{R}^b, g : \mathbb{R}^b \to \mathbb{R}^c, h : \mathbb{R}^c \to \mathbb{R}^d$. Then the Jacobian of P at point $\mathbf{x} \in \mathbb{R}^a$ is

$$J_P(\mathbf{x}) = J_h(g(k(\mathbf{x}))) \cdot J_g(k(\mathbf{x})) \cdot J_k(\mathbf{x})$$

which we get by simply applying the chain rule. Notice that matrix multiplication is associative so there are two possible computation orders. These two orders are the basis for the main types of AD: reverse mode and forward mode. The program P is derived with the two modes in the following way:

$$\begin{aligned} \text{Forward mode:} \quad & \overrightarrow{J_P}(\mathbf{x}) = J_h(g(k(\mathbf{x}))) \cdot [J_g(k(\mathbf{x})) \cdot J_k(\mathbf{x})] \\ \text{Reverse mode:} \quad & \overleftarrow{J_P}(\mathbf{x}) = [J_h(g(k(\mathbf{x}))) \cdot J_g(k(\mathbf{x}))] \cdot J_k(\mathbf{x}) \end{aligned}$$

In forward mode, the matrices are multiplied right to left, so intermediate derivatives are computed in program order. In reverse mode, the matrices are multiplied left to right, so intermediate derivatives are computed in reverse program order.

3.1 Efficiency of Differentiated Programs

The two modes are desirable in different use cases. We continue with the example in program $P : \mathbb{R}^a \to \mathbb{R}^d$. If there are considerably more inputs than outputs i.e. $d \ll a$, reverse mode will be much faster than forward mode. The cause is that in forward mode the intermediate result is a matrix of dimensions $a \times c$ which is much larger than the $b \times d$ intermediate matrix produced by reverse mode. With analogous argument, if we have considerably more outputs than inputs i.e. $a \ll d$, forward mode will be faster. Lets look at an example where $P(\mathbf{x}) = h(g(k(\mathbf{x})))$

with $k : \mathbb{R}^n \to \mathbb{R}^n, g : \mathbb{R}^n \to \mathbb{R}^n, h : \mathbb{R}^n \to \mathbb{R}$. I.e. a program that takes n inputs and returns one output, so $a = n$ and $d = 1$. Then the Jacobian is:

$$\overbrace{J_P(\mathbf{x})}^{1\times n} = \overbrace{J_h(g(k(\mathbf{x})))}^{1\times n} \cdot \overbrace{J_g(k(\mathbf{x}))}^{n\times n} \cdot \overbrace{J_k(\mathbf{x})}^{n\times n}$$

In reverse mode, we first multiply the matrices J_h of $1 \times n$ and J_g of $n \times n$ which can be done in $O(n^2)$ work. The resulting matrix has dimensions $1 \times n$ which is multiplied on J_k of $n \times n$. This is $O(n^2)$ work so reverse mode uses $O(n^2)$ work in total for this example.
In forward mode, it starts by multiplying the matrices J_g of $n \times n$ and J_k of $n \times n$, which takes $O(n^3)$ work. This results in a matrix of size $n \times n$ on which J_h of $1 \times n$ is multiplied, taking $O(n^2)$. Thus forward mode takes $O(n^3)$ work in this example where reverse mode only took $O(n^2)$.

Notice that $\overrightarrow{J_P}(\mathbf{x})$ depends on intermediate results from the original program, $k(\mathbf{x})$ and $g(k(\mathbf{x}))$. In forward mode, the derivatives are computed in program order, so the computations can simply be interwoven in the original program. In reverse mode, we have to execute the original program $P(\mathbf{x})$ first while saving intermediate results and then compute $\overleftarrow{J_P}(\mathbf{x})$, e.g. we need to compute $g(k(\mathbf{x}))$ to compute Jacobian $J_h(g(k(\mathbf{x})))$. This means that reverse mode derivatives introduce more runtime constants and uses more memory than forward mode. Therefore, forward mode is preferred when $a \approx d$.

Another important property of AD is that a row/column of the Jacobian can be constructed such that it has the same asymptotic work-depth as the original program. This is because a derivative statement only produces a constant overhead to the corresponding original statement (Baydin et al. 2018, p. 3). However in practice, it might be more efficient to modify the asymptotics slightly, which we will look more into later in the report.

3.2 Forward Mode

In forward mode, $\overrightarrow{J_P}(as)$ is computed in point $as = [a_0, \ldots, a_n]$ with an initial direction of $\dot{as} = [\dot{a}_0, \ldots, \dot{a}_n]$. Derivatives of intermediate variables are computed in program order, right after their initialisation. A program statement can be transformed with the following rule

Forward Primal Trace

```
let v1 = x1 * x2
let v2 =
  if v1 > 0 then v1
            else -1 * v1
let y = v2 + 2 * x1**2
```

Forward Tangent (Derivative) Trace

```
let v̇1 = x2 * ẋ1 + x1 * ẋ2
let v̇2 =
  if v1 > 0 then v̇1
            else -1 * v̇1
let ẏ = v̇2 + 4 * x1 * ẋ1
```

Figure 3.1: An example program derived with forward mode. The first column shows the original program $P : \mathbb{R} \to \mathbb{R} \to \mathbb{R}$ which implements function $f(x_1, x_2) = |x_1 x_2| + 2x_1^2$. The second column shows the computation of the tangents. The differentiated program $\overrightarrow{P}(x_1, x_2, \dot{x}_1, \dot{x}_2)$ is the primal trace interleaved with the tangent operations.

(Schenck et al. 2022, p. 2):

$$\textbf{let } v = f(a_0, \ldots, a_n) \Longrightarrow \begin{array}{l} \textbf{let } v = f(a_0, \ldots, a_n) \\ \textbf{let } \dot{v} = \sum_{i=0}^{n} \dfrac{\partial f(a_0, \ldots, a_n)}{\partial a_i} \dot{a}_i \end{array} \tag{3.1}$$

where the notation $\dot{v}$ means the *tangent* of intermediate variable v. A tangent in this context is the derivative of v with respect to the direction $\dot{as}$. I.e. the tangent is the derivative of an intermediate variable with respect to the input. Notice that the rule assumes access to the tangent $\dot{a}_i$. This is a safe assumption since a_i cannot be used before initialisation so its tangent must have been computed in the transformation of a previous statement.
By applying this transformation to every line of the program, a program derivative is produced. The derivative of a program $P(as)$ is $\overrightarrow{P}(as, \dot{as})$. The program transformation is defined in the function (Jacobian-vector product):

$$jvp : (P : \mathbb{R}^n \to \mathbb{R}^m) \to (\overrightarrow{P} : \mathbb{R}^n \to \mathbb{R}^n \to \mathbb{R}^m)$$

Mind that a transformation of the program according to $\dot{as}$ only produces a single column of the Jacobian. To compute some column i of the Jacobian, you call jvp with `dx = (0,...,0) with [i] = 1`, i.e. the i'th unit vector. The whole Jacobian is computed by mapping jvp over the standard basis of the input domain $\mathbb{R}^n$.

Figure 3.1 shows the forward mode derivative of a small example program.

3.3 Reverse Mode

A program derivative constructed by reverse mode, consists of a forward and a return sweep. In

Forward Primal Trace

```
let v1 = x1 * x2
let v2 =
  if v1 > 0 then v1
            else -1 * v1
let y = v2 + 2 * x1**2
```

Reverse Adjoint (Derivative) Trace

```
--from v1 assignment
let x̅1̅ += x2 * v̅1̅
let x̅2̅ += x1 * v̅1̅
--from v2 assignment
let v̅1̅ +=
  if v1 > 0 then v̅2̅
            else -1 * v̅2̅
--from y assignment
let v̅2̅ += y̅
let x̅1̅ += 4 * x1 * y̅
```

Figure 3.2: Deriving an example program $P : \mathbb{R} \rightarrow \mathbb{R}$ with reverse mode. The first column shows the original program. The second column shows the computation of the adjoints which are executed bottom up. The differentiated program $\overleftarrow{P}(x_1, x_2, \overline{y})$ is the primal trace followed by the reverse trace.

```
def P⃗ x1 x2 ẋ1 ẋ2 =
  let v1 = x1 * x2
  let v̇1 = x2 * ẋ1 + x1 * ẋ2
  let v2 =
    if v1 > 0 then v1
              else -1 * v1
  let v̇2 =
    if v1 > 0 then v̇1
              else -1 * v̇1
  let y = v2 + 2 * x1**2
  let ẏ = v̇2 + 4 * x1 * ẋ1
  in (y,ẏ)
```

(a) Forward mode derivative $\overrightarrow{P}$

```
def P⃖ x1 x2 y̅ =
  let v1 = x1 * x2
  let v2 =
    if v1 > 0 then v1
              else -1 * v1
  let y = v2 + 2 * x1**2
  let v̅2̅ += y̅
  let x̅1̅ += 4 * x1 * y̅
  let v̅1̅ +=
    if v1 > 0 then v̅2̅
              else -1 * v̅2̅
  let x̅1̅ += x2 * v̅1̅
  let x̅2̅ += x1 * v̅1̅
  in (y, x̅1̅, x̅2̅)
```

(b) Reverse mode derivative $\overleftarrow{P}$

Figure 3.3: The forward and reverse mode differentiated programs from figure 3.1 and figure 3.2 side-by-side.

its simplest form, the forward sweep is just the original code where intermediate variables are saved. As mentioned earlier, intermediate variables are saved as the derivatives are computed in reverse program order but depend on earlier intermediate results. Sometimes the forward sweep code will be modified to compute extra variables used for optimising the reverse sweep.

The return sweep computes the *adjoints* in reverse program order. In the context of AD, an adjoint $\overline{a}$ of variable a is the derivative of P with respect to a. An adjoint may be updated multiple times as every use of a contributes to $\overline{a}$. The adjoint $\overline{a}$ is updated with the derivative of the statement result with respect to a. The program transformation done by reverse mode AD is given by the following function (vector-Jacobian product):

$$vjp : (P : \mathbb{R}^n \to \mathbb{R}^m) \to (\overleftarrow{P} : \mathbb{R}^n \to \mathbb{R}^m \to \mathbb{R}^n)$$

$\overleftarrow{P}$ starts computing adjoints from the end of the program so it takes output adjoints $\overline{ys}$ as argument as well as an input point $as = [a_0, \dots, a_n]$. Mind that a single reverse AD application to a program with $\overline{ys}$, only produces a single row of the Jacobian. To compute the i'th row of the Jacobian, you call vjp with the i'th unit vector. The full Jacobian is constructed by mapping vjp over the standard basis of output domain $\mathbb{R}^m$.

The transformation rule for reverse mode AD is as follows (Schenck et al. 2022, p. 2):

$$\begin{array}{l} \textbf{let } v = f(a_0, \dots, a_n) \\ stmts \end{array} \Longrightarrow \begin{array}{l} \textbf{let } v = f(a_0, \dots, a_n) \\ vjp(stmts) \\ \textbf{let } \overline{a_0} += \dfrac{\partial f(a_0, \dots, a_n)}{\partial a_0}\overline{v} \\ \vdots \\ \textbf{let } \overline{a_n} += \dfrac{\partial f(a_0, \dots, a_n)}{\partial a_n}\overline{v} \end{array} \tag{3.2}$$

where $stmts$ is a placeholder for the program statements subsequent to the assignment of v. Notice that the transformed program uses $\overline{v}$ but does not explicitly define it. However, it is guaranteed that any adjoint $\overline{v}$ is finalised for a legal program P. Clearly, all uses of v must appear after the initialisation of v so when the adjoints are accumulated in reverse program order and we reach the assignment of v, there can be no more contributions to $\overline{v}$. Thus $\overline{v}$ is final at this point in the return sweep. The adjoint of a variable will be suitably initialised before the first accumulation, which is not shown.

Figure 3.2 shows the reverse mode derivative of a small example program, which is the same example used for forward mode in figure 3.1. Figure 3.3 shows both of the differentiated programs for comparison.

3.3.1 Nested Scopes in Reverse Automatic Differentiation

This section is rather technical and only included for completeness, so it can safely be omitted by the reader.

As mentioned, control structures are not modified when deriving a program. Only statements directly affecting the result are affected by AD. However in some cases of reverse AD, the derivation will introduce significant memory overheads when program scopes are nested. The reason for this is that intermediate variables and results of inner scopes are used by the return sweep so they need to be stored for a longer period of time than in the original program. This problem can be accommodated by instead recomputing variables of inner scopes when they are used in the return sweep (Schenck et al. 2022, p. 4). For example, a new scope will appear when introducing an anonymous function. Functions can be derived by using vjp_{λ}:

$$
\begin{aligned}
vjp_{\lambda}(\overline{res}, \lambda x_1 \ldots x_n \rightarrow body) \Rightarrow \lambda x_1 \ldots x_n \rightarrow \overleftrightarrow{body}& \\
\textbf{where}\ stms\ \textbf{in}\ res \leftarrow body& \\
\overleftrightarrow{stms}\ \textbf{in}\ (res, \overline{fvs_{body}}) \leftarrow vjp_{body}(\overline{res}, body)& \\
\overleftrightarrow{body} \leftarrow \overleftrightarrow{stms}\ \textbf{in}\ \overline{fvs_{body}}&
\end{aligned}
$$

where vjp_{body} returns the derivative of some body of statements and $\overline{fvs_{body}}$ is the adjoints of the free variables in $body$, $\overline{FV(body)}$. The double arrow in $\overleftrightarrow{stms}$ and $\overleftrightarrow{body}$ express that they contain both a forward and a return sweep. This means that a anonymous function derivative may have multiple nested forward and return sweeps. Similarly, control structure such as loops or if-statements will have introduce new scopes. These are derived using vjp_{body}. Consider the example shown in figure 3.4. The task is to derive the anonymous function in figure 3.4a which has four nested scopes. This results in 4 forward sweeps and 4 return sweeps (see figure 3.4b), meaning the forward sweep is computed 4 times in order to make the original variables available to the return sweep. However, the original variables are not used by the return sweeps in this function, so the compiler detects the re-executed forward sweeps as dead code and eliminates them (see figure 3.4c). This is a common pattern which occurs when scopes are perfectly nested, i.e. the entire content is in the innermost scope[1] (Schenck et al. 2022, p. 4). Perfect nesting ensures that the result of the forward sweep is not used in the return sweep, since the outer scopes only contains a single statement each. In the example, all of the re-computation is removed but notice this may not be the case if the innermost scope has more than one statement. A common expectation is thus to execute the forward sweep twice (Schenck et al. 2022, p. 4): once for the outermost scope, and once for the innermost

[1] This does not apply to perfectly nested loops but these are not relevant to the project.

```
-- scope 0: outer
λ ass →
  let xss = -- scope 1: first map
    map (λ c as →
           let xs = -- scope 2: if-branches
             if c then ...
                  -- scope 3: second map
                  else map (λ a → a*a) as
           in xs
        ) cs ass
  in xss
```

(a) A Futhark function

```
λ ass x̅s̅s̅ →
  let xss = -- forward scope 0
    map (λ c as →
           if c then ... else map (λ a → a*a) as
        ) cs ass
  let a̅s̅s̅ = -- return scope 0
    map (λ c as x̅s̅ →
            let xs = if c then ... -- forward scope 1
                          else map (λ a → a*a) as
            in if c then ... -- return scope 1
                    else -- forward scope 2
                      let xs' = map (λ a → a*a) as
                      let a̅s̅ = -- return scope 2
                        map (λ a x̅ →
                               -- forward scope 3
                               let x = a * a
                               -- return scope 3
                               in 2 * a * x̅
                            ) as x̅s̅
                      in a̅s̅
        ) cs ass x̅s̅s̅
  in a̅s̅s̅
```

(b) The function derived using vjp_λ

```
λ ass x̅s̅s̅ →
  let a̅s̅s̅ =
    map (λ c as x̅s̅ →
           if c then ...
           else map (λ a x̅ → 2*a*x̅ ) as x̅s̅ ) cs ass x̅s̅s̅
  in a̅s̅s̅
```

(c) The optimised function derivative (dead code removal)

Figure 3.4: Example of redundant execution in reverse AD with nested scopes (Schenck et al. 2022, p. 4).

scope. The redundant re-execution in the innermost scope will typically be cheap because it often only handles scalar operations.

Differentiation of (Multi-)Reduce in Reverse Mode 4

Reverse mode differentiation of reduce has already been implemented in Futhark – however, the more complicated version reduce-by-index has not. We will begin by examining the automatic differentiation of single-reduce since it is essentially a special case of reduce-by-index. As such we will build on the differentiation of reduce to derive the rule for differentiation of reduce-by-index.

First, the general derivative is explained which works for all valid operators ⊙. In some cases, the derivative can be computed more efficiently than the generic case so afterwards we go through a couple of special cases, which are also relevant for reduce-by-index. Recall the semantics of `reduce`:

$$\begin{aligned}\textbf{reduce} &: (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to \alpha\\ \textbf{reduce}\ \odot\ e_{\odot}\ [a_0, a_1, \ldots, a_{n-1}] &\equiv a_0 \odot a_1 \odot \cdots \odot a_{n-1}\end{aligned}$$

where ⊙ is an associative operator.

4.1 Generic Case for Reduce

Consider a program statement:

```
let y = reduce ⊙ ne as
```

so we have $y = a_0 \odot \cdots \odot a_{n-1}$. Thus the result y of a reduce is affected by each element a_i of the input array *as* so their adjoints $\overline{a_i}$ are updated. We can more easily reason about an adjoint $\overline{a_i}$ by grouping the terms of y:

$$y = \overbrace{a_0 \odot \cdots \odot a_{i-1}}^{l_i} \odot a_i \odot \overbrace{a_{i+1} \odot \cdots \odot a_{n-1}}^{r_i}$$

If we have l_i and r_i for every i then we can simply apply the reverse AD rewrite rule 3.2:

$$\overline{a_i} \mathrel{\overline{+}=} \frac{\partial l_i \odot a_i \odot r_i}{\partial a_i}\overline{y}$$

$\overline{+}$ represents an addition operator that matches the data type (notice it might be vectorised). Mind that $\odot$ is not allowed to have free variables in the Futhark AD implementation (yet). If $\odot$ had free variables, the adjoints for those would need to be computed as well. The derivative of $l_i \odot a_i \odot r_i$ is computed with the helper function generated by the application of the vjp_λ transformation (briefly mentioned in section 3.3.1) to the extended operator, which reduces three elements:

$$f \leftarrow vjp_\lambda(\overline{y}, \lambda(l_i, a_i, r_i) \rightarrow l_i \odot a_i \odot r_i)$$

This is how a single adjoint update is computed but we remain to explain the computation of l_i and r_i which should be computed for each a_i. The full return sweep of the generic case is (Schenck et al. 2022, p. 6):

```
-- Return sweep:
let ls = scan_exc ⊙ e_⊙ as
let rs = reverse as ▷
         scan_exc (flip ⊙) e_⊙ ▷ reverse
let a̅s̅ +̅= map f ls as rs
```

where ▷ pipes the result to a function and `flip` flips the argument order of a function.
To compute the updates, we need l_i and r_i for all indices i in as. The left values ls can be found by a scan with $\odot$. The scan is exclusive since l_i should not include a_i. The right values rs are more complicated to compute as the scan should be from right to left. The problem is solved by reversing as before scanning and then reversing the result. The arguments of $\odot$ are flipped in the scan since commutativity is not guaranteed. Afterwards, the adjoint updates can be computed by mapping over ls, rs and as with function f.

The differentiated program preserves work-depth asymptotics of the original program as **reduce** has $O(\log(n))$ depth and $O(n)$ work which is the same as the most expensive operator in the derivative (scan). However, the derivative requires two scans, a map and two reverses, which add some expensive constants. Thus we are interested in identifying special cases where we can lower those constants.

4.2 Special Cases of Reduce

In some cases, parts of the generic reduce derivative are excessive or can be done in more efficient ways. Some of these cases and their derivatives are presented here. The details of case identification and execution are explained more thoroughly in the implementation section.

Addition: When $\odot \equiv +$, the derivative is simply $\frac{\partial l_i + a_i + r_i}{\partial a_i}\overline{y} = \overline{y}$. Thus the adjoint contribution will be $\overline{as} + = \overline{y}$.

Multiplication: When $\odot \equiv \cdot$, the derivative is simply $\frac{\partial l_i \cdot a_i \cdot r_i}{\partial a_i}\overline{y} = l_i \cdot r_i \cdot \overline{y}$. Here, we can further identify three relevant subcases:

- If *as* contains only nonzero values, then we have $l_i \cdot r_i = y/a_i$ when y is the result of the reduce. Thus we update the adjoint with $a_i + = \frac{y}{a_i} \cdot \overline{y}$.

- The second case is that *as* contains exactly one zero value at element a_k. Then the derivative is zero for all indexes except k i.e. $i \in [0..n]/k, l_i \cdot r_i \equiv 0$ so there is no adjoint contribution to insert in the differentiated program for these indices. The zero index k will, however, have the contribution $a_k + = l_i \cdot r_i \cdot \overline{y}$.

- If *as* contains multiple zero values, then $\forall i, l_i \cdot r_i \equiv 0$, so the adjoints are not changed.

To determine which of the above cases hold, statements are added to the program that counts the number of zeros in *as* and compute the product of non-zero elements. The pseudocode below shows the modified forward sweep.

```
-- Multiplication forward sweep:
let (zs, prod_zs) = reduce (λ(c1,p1) (c2,p2) → (c1+c2, p1*p2))
                           (0,1)
                           (map (λx →
                                  if x == 0 then (1,1)
                                  else (0,x))
                              as)
let y = if zs > 0 then 0 else prod_zs
```

Min-max: When we reduce with the min or max operators, only one element of *as* will affect the result, namely the minimum or maximum element a_k. Thus the return sweep only updates the adjoint of a_k by $\texttt{let}\ \overline{\texttt{as}}\texttt{[k]}\ \overline{\texttt{+=}}\ \texttt{as[k]}$. The forward sweep is required to store the minimum/maximum element a_k and its index k (using `argmin` or `argmax`).

4.3 Reduce-by-index

Reduce-by-index is a SOAC in Futhark which executes a collection of reduces (Henriksen, Hellfritzsch, et al. 2020). The intuition of reduce-by-index is that it generalises the computation of a histogram where values are put into some buckets specified by indices. It takes three arrays: destination, indices and values. It reduces values into the corresponding indices in the destination array. Like single reduce, reduce-by-index uses an operator ⊙ to combine values. ⊙ is required to be associative (like a normal reduce) and commutative. It needs to be commutative since there are no specified computation order so a non-commutative operator would give non-deterministic results. Recall that the semantics of reduce-by-index are:

```
reduce_by_index : *[m]α → (α → α → α) → α → [n]i32 → [n]α → *[m]α
reduce_by_index dst (⊙) e inds vs =
  for i = 0..n-1 do
    dst[inds[i]] ⊙= vs[i]
```

where e is the neutral element of ⊙. Like scatter, the destination array is consumed because the implementation overwrites the original elements of used buckets. If the index array inds holds any illegal bucket numbers (out-of-bounds of destination), these indices are simply ignored. Notice that the neutral element $\mathtt{e}_{\odot}$ is not used. This is because presented semantics are sequential but it is required for efficient GPU code generation.

The derivative of reduce-by-index is a modified version of the single-reduce derivative. The intuition is that each separate bucket of reduce-by-index should be derived like a single-reduce. Thus reduce-by-index has the same generic and special cases but on every bucket instead of a single bucket. Another difference is that we need to update the adjoint of the destination array as well as the value adjoints. Adjoints of the indices should not be updated since they do not directly affect the result. Our solution is explained in chapter 6.

Differentiation of Scan in Reverse Mode

5

Recall the semantics of scan:

$$
\begin{aligned}
&\textbf{scan} : (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to [n]\alpha \\
&\textbf{scan}\ \odot\ e_{\odot}\,[x_0, \ldots, x_{n-1}] \equiv [x_0, x_0 \odot x_1, \ldots, x_0 \odot \cdots \odot x_{n-1}]
\end{aligned}
$$

where $\odot$ is an associative operator. The reverse AD transformation of scan has only been implemented partially in the compiler. Currently, it does not work with arrays containing tuples when the operator crosses the tuple entries. For example, using the scan operator

$$\lambda(a_1, a_2)\,(b_1, b_2)\ \to\ (b_1 \cdot a_2 + b_2 \cdot a_1, b_1 \cdot b_2)$$

does not work in the current implementation even though the operator is associative. Our goal is to modify and extend the implementation so scan works for arrays with tuples of scalars. Additionally, the compiler is lacking some optimisations, including some that have been identified by Schenck et al. 2022.

5.1 Generic Case of Scan

The generic case is based on the solution presented by Schenck et al. 2022. This section presents the steps and their associated arguments when constructing a return sweep for the reverse AD derivative of scan. As a starting point, we look at an imperative implementation of scan:

```
def scan [n] (as: [n]α) : [n]α =
  let rs[0] = as[0]
  let ys = loop rs = rs for i in [1..n-1] do
             let rs[i] = rs[i-1] ⊙ as[i] in rs
```

The idea is to unroll the loop, apply the reverse AD rewrite rule and then roll the statements back into a loop. The unrolled loop is:

```
let rs[0] = as[0]
let rs[1] = rs[0] ⊙ as[1]
...
let rs[n-1] = rs[n-2] ⊙ as[n-1]
```

Then we can apply the reverse AD rewrite rule given by (3.2) to each statement, resulting in the reverse sweep:

```
let r̅s̅[n-2] = ∂(rs[n-2] ⊙ as[n-1])/∂rs[n-2] * r̅s̅[n-1]
let a̅s̅[n-1] = ∂(rs[n-2] ⊙ as[n-1])/∂as[n-1] * r̅s̅[n-1]
...
let r̅s̅[0] = ∂(rs[0] ⊙ as[1])/∂rs[0] * r̅s̅[1]
let a̅s̅[1] = ∂(rs[0] ⊙ as[1])/∂as[1] * r̅s̅[1]
let a̅s̅[0] = r̅s̅[0]
```

When we roll the return sweep back into a loop, we get:

```
let (r̅s̅,a̅s̅) = loop (r̅s̅,a̅s̅) = (copy y̅s̅,a̅s̅) for i in [n-1..1] do
  let r̅s̅[i-1] += ∂(rs[i-1] ⊙ as[i])/∂rs[i-1] * r̅s̅[i]
  let a̅s̅[i] += ∂(rs[i-1] ⊙ as[i])/∂as[i] * r̅s̅[i]
  in (r̅s̅,a̅s̅)
let a̅s̅[0] += r̅s̅[0]
```

Notice that the loop is in reverse order of the original loop with the iterator going from $n-1$ to 1. When analysing the dependencies between iterations, we notice that $\overline{\texttt{rs}}$ is independent of $\overline{\texttt{as}}$, and $\overline{\texttt{as}}$ depends only on $\overline{\texttt{rs}}$ elements modified by previous iterations. Thus the updates to $\overline{\texttt{rs}}$ and $\overline{\texttt{as}}$ can be safely distributed into two different loops without breaking any dependency cycles.

```
let r̅s̅ = loop r̅s̅ = copy y̅s̅ for i in [n-1..1] do
  let r̅s̅[i-1] += ∂(rs[i-1] ⊙ as[i])/∂rs[i-1] * r̅s̅[i]
  in r̅s̅

let a̅s̅ = loop a̅s̅ = a̅s̅ for i in [n-1..0] do
  let a̅s̅[i] +=
    if i==0 then r̅s̅[i]
    else ∂(rs[i-1] ⊙ as[i])/∂as[i] * r̅s̅[i]
  in a̅s̅
```

Notice that the final statement of the combined loop was `let` $\overline{\texttt{as}}$`[0]` $\overline{+}$`=` $\overline{\texttt{rs}}$`[0]`, which is moved into the $\overline{\texttt{as}}$ loop by extending the iteration space. To make the return sweep efficient in Futhark, we need to convert the code from imperative style to SOACs, which exploit Futhark's proficiency in GPU utilisation. The loop updating $\overline{\texttt{as}}$ can easily be changed to a `map` over `as` and `iota n` for indexing into $\overline{\texttt{rs}}$.

```
let a̅s̅ +̅=
  map (λi r̅i̅ ai →
         if i==0
         then r̅i̅
         else ∂(rs[i-1] ⊙ ai)/∂ai * r̅i̅
      ) (iota n) r̅s̅ as
```

In the loop constructing $\overline{\texttt{rs}}$, there are dependencies across iterations. This means that the conversion to parallel code requires a bit more thought. We start by transforming the loop to the following code:

```
let r̅s̅ = loop r̅s̅ = replicate n 0 for i in [n-1..0] do
  let r̅s̅[i] +=
    if i==n-1 then y̅s̅[i]
    else ∂(rs[i] ⊙ as[i+1])/∂rs[i] * r̅s̅[i+1]
  in r̅s̅
```

The iteration space has been extended to cover the full length of the adjoint array and the indices are shifted $+1$. Notice that before the element $\overline{\texttt{rs}}[n-1]$ was not updated by the loop body because $\overline{\texttt{rs}}$ was initialised to $\overline{\texttt{ys}}$. This loop can be expressed by a backward linear recurrence (Schenck et al. 2022, p. 8):

$$\begin{aligned}
\overline{rs}_{n-1} &= \overline{ys}_{n-1}\\
\overline{rs}_{i} &= \overline{ys}_{i} + cs_{i} \cdot \overline{rs}_{i+1}, i \in n-2\ldots 0
\end{aligned}$$

where $\overline{ys}$ is the adjoint of the scan result ys, and cs is an array of Jacobians $\overleftarrow{J}_{\odot}(\overline{rs}[i])$, defined by $cs_{n-1} = \overline{1}$ and $cs_i = {}^{\partial(rs_i \odot as_{i+1})}\!/\!_{\partial rs_i}$. The last element cs_{n-1} is not used in the recurrence but is used by the code out of convenience so it is set to the identity matrix that matches size of the Jacobians. We can transform the backward recurrence to a forward one by instead constructing a recurrence for $\overline{rsr}$ such that $\overline{\texttt{rsr}}$ `=` `reverse` $\overline{\texttt{rs}}$:

$$\begin{aligned}
\overline{rsr}_{0} &= \overline{ys}_{n-1}\\
\overline{rsr}_{i} &= \overline{ys}_{n-i-1} + cs_{n-i-1} \cdot \overline{rsr}_{i-1}, i \in 1\ldots n-1
\end{aligned}$$

This type of recurrence can be solved by a `scan` with linear function composition (Blelloch 1990):

```
scan (λ(d1,c1) (d2,c2) →
        (d2 + c2 · d1, c2 × c1)
     ) (0̅,1̅) (reverse y̅s̅) (reverse cs)
```

where $(\overline{0}, \overline{1})$ is the neutral element. $\overline{0}$ is a structure of zeros matching the element type of $\overline{\texttt{ys}}$. The input arrays are reversed since the recurrence uses the elements in reverse order. Now every imperative operation has been converted to GPU parallel operations. The final return sweep of a statement `let ys = scan` $\odot$ $e_\odot$ `as`:

```
1  -- generating adjoint of rs
2  let cs =
3    map (λi → if i==n-1
4               then 1̅
5               else ∂(rs[i] ⊙ as[i+1])/∂rs[i]
6        ) (iota n)
7  let lin⊙ (d1,c1) (d2,c2) = (d2 + c2 · d1, c2 × c1)
8  let (r̅s̅,_) =
9    scan lin⊙ (0̅,1̅) (reverse y̅s̅) (reverse cs)
10     ▷ reverse
11
12 -- updating adjoint of as
13 let a̅s̅ += 
14   map (λi r̅i̅ ai →
15           if i==0
16           then r̅i̅
17           else ∂(rs[i-1] ⊙ ai)/∂ai * r̅i̅
18       ) (iota n) r̅s̅ as
```

where as before, $(\overline{0}, \overline{1})$ is the neutral element of `lin`$_\odot$ using the appropriate types. E.g. when `cs` is an array of $d \times d$ matrices, $\overline{1}$ is a $d \times d$ identity matrix and $\overline{0}$ is a vector of d zeros. The forward sweep is not modified so it is the primal program `let ys = scan` $\odot$ $e_\odot$ `as`.

The construction of $\overline{\texttt{rs}}$ follows straight-forward from the linear recurrence (ln. 1-10). The only difference is that we reverse the result of the `scan` to get $\overline{\texttt{rs}}$ instead of the reversed recurrence $\overline{rsr}$. Notice that the input scan operator $\odot$ does not have any impact on the scan operator $lin_\odot$ in the return sweep.

`cs` is constructed at lines 2-6, which as mentioned is an array of Jacobians each defined by a partial derivative. Naturally, the partial derivatives need to be converted to actual code. The derivative code is generated by mapping vjp_λ over the identity matrix (see section 3.3). We return to this problem later in chapter 7.

$\overline{\texttt{as}}$ is updated by a simple `map` at lines 12-18. The `map` function computes Jacobian-vector product $\overleftarrow{J}_{\odot_{right}}(as_i) \cdot \overline{rs}_i$, where $\odot_{right}$ is $\odot$ where the arguments are flipped. However, we do not need to compute the full Jacobian. We have the property that $\overleftarrow{P}(x, \overline{y}) = \overline{y}\,\overleftarrow{J}_{P(x)}$ for some program P with input x, output y and output adjoint $\overline{y}$ (Schenck et al. 2022, p. 3). This means we can execute vjp_λ just once by $vjp_\lambda(\overline{\texttt{rs}}, \odot_{right})$ and use the resulting function in the `map` operator.

The return sweep presented by Schenck et al. 2022 is very similar to the one we present but our version is simplified. In their version they have a `map` following the `scan` which is unnecessary and expensive. In our version, only the first result of the `scan` $\overline{\texttt{rs}}$ is read/used whereas in their version both results are read.

5.2 Special Cases of Scan

The paper Schenck et al. 2022 presents two special cases: vectorised operations and addition.

5.2.1 Vectorised

When the scan operator is a sequence of maps, the operations can be converted to multiple scans instead. We can apply the rewrite rule (Schenck et al. 2022, p. 8):

```
scan (map ⊙) ne xs ⇒
  transpose xs ▷ map ( scan ⊙ ne ) ▷ transpose
```

where $\overline{\texttt{ne}}$ is an array with neutral elements `ne`. Mapping with `scan` $\odot$ is faster on the GPU, provided the length of the inner arrays m is less than the length of the input array n. When $\odot$ has depth d, the transformed statement has the depth $O(d \log(m))$ as we scan over arrays of length m. The original statement has depth $O(d \log(n))$ which is asymptotically worse when $n > m$.

Another advantage is that the reverse AD implementation for `scan` becomes simpler. More importantly, note that without the rewrite the generic rule for scan is not work-efficient when the `scan` operator receives arrays as arguments because two Jacobians are multiplied. Consider the statement

```
scan (map (+)) (replicate m 0) as
```

where `as: [n][m]i32`. This scan has work $O(nm)$. The Jacobians of the `map` will have size $m \times m$, which means that the reverse sweep `scan` with lin_o will have work $O(nm^3)$. The generic rewrite rule will therefore have work $O(nm^3)$, which is worse than the work of the original scan $O(nm)$ and thus not work efficient. The vectorised rewrite rule turns `scan`'s operator into one that operates on (tuples of) scalars rather than arrays such that the differentiation of `scan` becomes work efficient.

5.2.2 Addition

Another special case is that of addition. This case can be highly optimised. `cs` is redundant as every Jacobian is an identity matrix with addition as operator:

$$\frac{\partial(\texttt{rs[i]} + \texttt{as[i+1]})}{\partial\texttt{rs[i]}} = \overline{1}$$

Similarly when updating $\overline{\texttt{as}}$, the Jacobian is an identity matrix as well:

$$\frac{\partial(\texttt{rs[i-1]} + \texttt{as[i]})}{\partial\texttt{as[i]}} = \overline{1}$$

so `let` $\overline{\texttt{as}}$ $\overline{+}$`=` $\overline{\texttt{rs}}$. $\overline{\texttt{rs}}$ is only used as an intermediate variable for the return sweep, so actually we do not need to declare it. Thus the reverse sweep of the addition case is:

```
let a̅s̅ +̅= scan (+) 0 (reverse y̅s̅) ▷ reverse
```

Part II

Implementation

Reduce-by-index Implementation 6

We have implemented reduce-by-index and the presented special cases in the Futhark compiler. Recall the semantics of reduce-by-index (by imperative code):

```
reduce_by_index : *[w]α → (α → α → α) → α → [n]i32 → [n]α → *[w]α
reduce_by_index dst ⊙ ne is vs =
  for i = 0..n-1 do
    dst[is[i]] ⊙= vs[i]
```

where `dst` is the destination, $\odot$ is the operator, `ne` is the neutral element of $\odot$, and `vs` is an array of values who belong to a bucket defined by the corresponding index in `is`. Essentially, reduce-by-index does multiple reduces where each is identified by a bucket index i inside the destination array. This means the rationale behind reverse AD for reduce-by-index and single-reduce are closely related and we will use reverse AD of single-reduce as a basis for our solution.

In this section, we present the generated Futhark (pseudo)code for each case of reduce-by-index. This includes an examination of the program design and performance considerations. Additionally, the generated code is examined with work-depth analysis. This analysis is used to confirm that the differentiated program agrees with the same work-depth asymptotic as the original program. Using a conservative approach, the asymptotic depth of reduce-by-index is linear $O(kn)$, where `vs` is n long and the operator is $O(k)$ work (see table 2.2). This considers the unlikely worst case where all indices of `is` are equal. Instead, we maintain the expected asymptotic work-depth when constructing the reverse sweep, i.e. $O(nk)$ work and $O(k \log(n))$ depth. The differentiated program should keep or improve this work-depth. Notice that the size w of the destination array does not affect the asymptotic work-depth. The reason is that reduce-by-index makes in-place updates like scatter and we can update at most n elements, if all indices are in-bounds.

In summary, this chapter presents and justifies the solution for reduce-by-index with operators for the same cases as reverse AD of single-reduce. This includes for each case an implementation strategy, the generated (high-level) Futhark code, and a work-depth analysis. Additionally, the internal representation (IR) is presented for the generic case.

6.1 Generic Case

First, we go through the overall implementation strategy, then the details as presented in listing 6.1. The work-depth asymptotics of the generated Futhark pseudocode is then analysed. Lastly, to establish an understanding of how the operations of the Futhark code are fused, we present the IR (section 6.1.5).

6.1.1 Strategy

As mentioned previously, the differentiated program builds on the reasoning of differentiating `reduce` (see section 4.1). The task is to find a reverse AD transformation rule for a statement:

```
let ys = reduce_by_index dst ⊙ ne is vs
```

where we have the dimensions `vs:[n]`α, `is:[n]i64`, and `dst:[w]`α. The idea is the same as for single-reduce but we derive each bucket separately. Specifically, the derivation of reduce-by-index corresponds to applying single-reduce derivation w times. We can reason for each individual bucket y_i, whose semantics are:

$$y_i = dst_i \odot v_{j_1} \odot v_{j_2} \odot \cdots \odot v_{j_q}$$

where $j_1, \ldots, j_q$ are the q indices of values going into bucket i, i.e. we have $is_{j_1} = i, \ldots, is_{j_q} = i$. Notice that for each bucket there is exactly one value coming from `dst`, specifically dst_i. We can rewrite the original statement to two statements:

```
let h_part = reduce_by_index (replicate w ne) ⊙ ne is vs
let ys = map2 (⊙) dst h_part
```

This transformation is safe because ⊙ is commutative and associative. As it is semantically equivalent to the original statement, we can differentiate the transformed code instead. Notice, that `vs` is used only by the first statement and `dst` only by the second statement. This means we can separate the strategies for adjoint updates of `vs` and `dst`. We will begin with the strategy for `dst`.

6.1.1.1 Updating adjoints of `dst`

To update the adjoints of `dst`, we will differentiate the statement `let ys = map2 (⊙) dst h_part`. We apply the reverse mode transformation rule 3.2:

```
-- forward sweep
let ys = map2 (⊙) dst h_part
-- reverse sweep
let d̅s̅t̅ = ∂(map2 (⊙) dst h_part)/∂dst * y̅s̅
let h̅_̅p̅a̅r̅t̅ = ∂(map2 (⊙) dst h_part)/∂h_part * y̅s̅
```

The partial derivatives can be translated to code using vjp_λ. We can generically generate the adjoint updates by:

$$
\begin{aligned}
f_{dst} &\leftarrow vjp_\lambda(\overline{ys},\ map\ (\odot)) \\
f_{hpart} &\leftarrow vjp_\lambda(\overline{ys},\ map\ (\odot_{right}))
\end{aligned}
$$

which are both applied to `dst` and `h_part`. The operator $\odot_{right}$ is $\odot$ where the argument order is flipped. However, notice that $\odot$ is commutative so $\odot_{right} \equiv \odot$, meaning we need to apply vjp_λ only once creating a single differentiated function f_{dst}. Mind that for updating $\overline{\texttt{h\_part}}$, f_{dst} is applied to `h_part dst` and for updating $\overline{\texttt{dst}}$ the order is `dst h_part`. We generate the code:

```
-- forward sweep
let ys = map2 (⊙) dst h_part
-- reverse sweep
let d̅s̅t̅ = f_dst dst h_part
let h̅_̅p̅a̅r̅t̅ = f_dst h_part dst
```

I.e. we can compute the derivative for `dst` and `h_part` using two `map`s (which are fused in IR).

6.1.1.2 Updating adjoints of vs

The adjoints of `vs` are updated according to the statement `let h_part = reduce_by_index (replicate w ne) ⊙ ne is vs`. Since the destination is a `replicate`, there is no need to compute its adjoints. Naturally, the semantics are similar to those of y_i:

$$
\begin{aligned}
h\_part_i &= ne \odot v_{j_1} \odot v_{j_2} \odot \cdots \odot v_{j_q} \\
&= v_{j_1} \odot v_{j_2} \odot \cdots \odot v_{j_q}
\end{aligned}
$$

We observe that we can safely ignore the destination when updating $\overline{\texttt{vs}}$, since the neutral elements have no impact on `h_part`. Thus if we can gather the values $v_{j_1}, \ldots, v_{j_q}$ together and use essentially the same strategy as for single-reduce. The challenge is that all values going to a specific bucket do not reside at consecutive positions in `vs`. When the values for each bucket are not grouped together in the value array, it is difficult to construct an efficient GPU implementation. Our strategy is to sort `vs` and `is` with regards to `is` such that all values for a single bucket are in a continuous segment. The specifics of the sorting algorithm are given in section 6.1.2.

When the arrays are sorted, we can use segmented scans to construct the left and right partial prefixes as in the reverse sweep of single reduce. To construct the right partial prefixes, we also need to reverse the bucket segments. There are two possibilities: reverse inside each bucket segment or reverse the whole array such that the order of segments is reversed as well. We have chosen the latter, since it offers the simplest solution and simple code is often most efficient. The only challenge is that we need to construct a flag array for the segments in reverse order. However, as it turns out the new flag array can be efficiently constructed from the original flags, essentially by reversing the original flags. This is described in more detail later. After the segmented right scan, it simply reverses again which restores the original order of segments.

In the single-reduce derivative, the next step is to derive the $\odot$ application using vjp_{λ}, i.e. the function $\lambda(l_i, v_i, r_i) \rightarrow l_i \odot v_i \odot r_i$ where $l_i = v_1 \odot \cdots \odot v_{i-1}$ and $r_i = v_{i+1} \odot \cdots \odot v_q$. Based on the semantics of `h_part`, the adjoint updates of `vs` are $\overline{v_i}\mathrel{+}= \frac{\partial l_i \odot v_i \odot r_i}{\partial v_i}\overline{h\_part_i}$. Unlike the single-reduce case, we cannot simply use:

$$f \leftarrow vjp_{\lambda}(\overline{h\_part_i}, \lambda(l_i, v_i, r_i) \rightarrow l_i \odot v_i \odot r_i)$$

The problem here is that $\overline{h\_part_i}$ is the adjoint of bucket i. The number of buckets is not known at compile time but vjp_{λ} generates the code at compile time. To solve this we include a `map`:

$$f \leftarrow vjp_{\lambda}(\overline{h\_part_{seg}}, map\ (\lambda(l_i, v_i, r_i) \rightarrow l_i \odot v_i \odot r_i))$$

The resulting anonymous function is given the arguments `ls`, `svs` and `rs` and the function which is differentiated, has the adjoint $\overline{h\_part_{seg}}$. $\overline{h\_part_{seg}}$ is $\overline{h\_part}$ where the adjoint for each bucket is repeated for each value going to that bucket. f results in the `vs` adjoint updates sorted with regards to `is`. Thus in the end, the adjoints must be distributed back into their original positions, so the correct `vs` adjoints are updated. So assuming access to the partial prefixes ls and rs, sorted values `svs`, and sorted iota `siota`, we get the transformed code:

```
-- forward sweep
let h_part = reduce_by_index (replicate w ne) ⊙ ne is vs
-- reverse sweep
let s̅v̅s̅ = f ls svs rs
let v̅s̅ += scatter (Scratch α n) siota s̅v̅s̅
```

We scatter into a `Scratch` which allocates an n-length array of values element type (see section 2.2 for explanation of scratch). It is safe to use `Scratch` here as the `scatter` only rearranges the n elements into another n length array, so no indices will be left uninitialised.

6.1.2 Sorting Strategy

Our strategy requires sorting `vs` and `is` with respect to `is`. We sort `iota n` with respect to `is` and then reposition `vs` using a gather with the sorted `iota n`. For sorting, we have chosen the algorithm radix sort which has $O(n)$ work and $O(\log(n))$ depth when the size of an element is constant. Notice that the work-depth complies with the optimal work-depth of the differentiated program. Radix sort sorts by comparing elements bitwise. Our implementation is based on Henriksen 2021 which is slightly optimised by comparing 2 bits at a time instead of just 1 bit.

We have made one optimisation on the algorithm which is specific to our AD implementation. When the indices are 64-bit integers, the sorting loop has 32 iterations. However, it is only required to consider the number of bits required to index into the last bucket, under the assumption that all indices are in-bounds. The problem is that indices are not required to be in-bounds of the destination array so the algorithm must consider all bits to assure no out-of-bounds values are placed in in-bounds segments.
Notice that only values whose indices are in-bounds of destination, will affect the result. This means only the adjoints of these values will be updated so out-of-bounds values actually do not need to be sorted. The straight-forward approach would be to zip values with their index using `iota n` and filter out the out-of-bounds values. However, `filter` is a rather expensive operation in Futhark so we would rather not use it. Instead, we can map out-of-bounds indices to the number of buckets. Then the number of significant bits is:

$$\lceil \log(hist\_size + 1) \rceil$$

Because the implementation looks at two bits in each loop iterations, the number of needed iterations is the significant bits divided by two.

It is possible to optimise the algorithm even more but that is not the focus of this project.

6.1.3 Generated High-Level Futhark Code

```
 1 -- Primal, assuming vs: [n]α, is: [n]i64, dst: [w]α:
 2 --    let ys = reduce_by_index dst ⊙ ne is vs
 3 -- Forward sweep:
 4   let h_part = reduce_by_index (replicate w ne) ⊙ ne is vs
 5   let ys = map2 ⊙ dst h_part
 6 -- Reverse sweep:
 7   h̅_̅p̅a̅r̅t̅ = $f_{dst}$ h_part dst
 8   d̅s̅t̅ += $f_{dst}$ dst h_part
 9
10   let flag = map (λi → i == 0 || sis[i] != sis[i-1]) (iota n)
11   let flag_rev = map (λi → i==0 || flag[n-i]) (iota n)
12   let ls = seg_scan_exc ⊙ ne flag svs
13   let rs = reverse svs ▷
14            seg_scan_exc ⊙ ne flag_rev ▷ reverse
15   let f_bar = map (λi → if i < w && -1 < i
16                         then h̅_̅p̅a̅r̅t̅[i]
17                         else 0s
18                   ) sis
19   let s̅v̅s̅ = f svs ls rs
20   v̅s̅ += scatter (Scratch α n) siota s̅v̅s̅
21 -- Where:
22 --  siota: 'iota n' sorted wrt 'is'
23 --  sis: 'is' sorted wrt 'is'
24 --  svs: 'vs' sorted wrt 'is'
25 --  f_dst = $vjp_\lambda$ ys_bar (map2 ⊙)
26 --  f   = $vjp_\lambda$ f_bar (map3 (λsvi li ri → li ⊙ svi ⊙ ri))
27 --  0s is a structure of type α with zero(s)
```

Listing 6.1: Pseudocode for generic case of reduce-by-index (sorting omitted)

Listing 6.1 shows the code generated by the compiler as Futhark-esque pseudo-code. The pattern `let ys = reduce_by_index dst ⊙ ne is vs` is the generic reduce-by-index case with array types $is \in [n]i64, vs \in [n]\alpha, dst \in [w]\alpha$. The forward sweep is modified as described in section 6.1.1 (ln. 4-5). It first computes the result without destination array in `h_part`, then the final result is computed in `ys` by applying `dst` with ⊙. Remember that ⊙ is required to be commutative, hence the treatment is safe.

The adjoint of `dst` is updated by a function generated with vjp_λ (ln. 8 and 25). The adjoint

sis:	[0,0,0, 1, 1, 1,1,1,2]
svs:	[0,1,2, 3, 4, 5,6,7,8]
flag:	[t,f,f, t, f, f,f,f,t]
flag_rev:	[t,t,f, f, f, f,t,f,f]
reverse svs:	[8,7,6, 5, 4, 3,2,1,0]
reverse rs:	[0,0,7,13,18,22,0,2,3]
rs:	[3,2,0,22,18,13,7,0,0]

Table 6.1: Example of constructing the right scan array rs with + as operator

of h_part is computed similarly but the argument order is flipped so it is differentiated with respect to h_part (ln. 7 and 25). On a separate note, a = instead of += would suffice since dst is consumed and thus it cannot be used in subsequent statements of the original program. This means that this is the first time the adjoint $\overline{\texttt{dst}}$ is updated (it is initialised to zeros).

For the computation of $\overline{\texttt{vs}}$, we assume access to sorted arrays sis, svs and siota which are is, vs and iota n respectively, sorted with regards to is. As mentioned earlier, this is done with radix sort. The sorting code is omitted since the specific implementation is not relevant. In principle, one can use any sorting implementation, preferably of high efficiency. It starts by constructing siota from sorting iota n with respect to is and then siota is used for computing svs.

The value array should be segmented scanned from the left and right inside each bucket, creating partial prefixes ls and rs. First, we need to construct flag arrays marking the segments. The flag array of the left scan ls is simple to create as a new segment begins whenever an element of sis is different from the previous (ln. 10). Naturally, the first element of the flag array is always true as a segment always begins there. Then ls can be created with an exclusive segmented scan using ⊙ (ln. 12). The flag array flag_rev for the right scan in rs is constructed from the left scan flag array (ln. 11). Semantically, it removes the first element of flag, reverses it and prepends a true marking the start of the first segment. This method can be implemented directly in Futhark but it is much more efficient to use a map instead. The reason is that in-place updates restrict some optimisations in the compiler, e.g. fusion. rs is then build exactly like in the reverse sweep of single reduce just with a segmented scan instead of a regular scan (ln. 13-14). Notice that the parameters of $\odot$ are not flipped for the right scan since it is guaranteed to be commutative. The construction of `rs` might be difficult to follow just with a verbal argument, so we illustrate it with an example in table 6.1 to better establish an intuition.

When `ls` and `rs` have been constructed, vjp_λ is used to derive the operator application $l_i \odot v_i \odot r_i$ according to v_i (ln. 15-19 and 26). Remember that in the case an index `sis[i]` of a value `svs [i]` is out-of-bounds of `dst`, it is ignored. Since the out-of-bounds values are not filtered out, it still needs to set dummy adjoint updates which are set to 0. The updates are sorted after `is` so the adjoint updates need to be sorted back into their original indices with `siota` and a `scatter` (ln. 20). Remember it is safe scatter into a `Scratch` here as it only permutes the elements.

6.1.4 Work-Depth Analysis

The ideal asymptotic work-depth of the differentiated program is that of the original program. The asymptotic expected work-depth of the original program statement is $O(kn)$ work and $O(k\log(n))$ depth when the operator $\odot$ has $O(k)$ work. The forward sweep is $O(k(w+n))$ work and $O(k\log(n))$ depth. Thus the program derivative is not work efficient: This is caused by the `map` over the destination. However, the number of values will seldom be smaller than the number of buckets, i.e. $w < n$ (Schenck et al. 2022, p. 7). Therefore in most use cases, the forward sweep will honor the asymptotic work-depth of the original program.

The reverse sweep should comply with the asymptotic work-depth as well. Remember that the sorting of the arrays complies with the work-depth asymptotics. The maps constructing the flag arrays use $O(n)$ work and $O(1)$ depth. The segmented scans for `ls` and `rs` are $O(kn)$ work and $O(k\log(n))$ depth. Reversing the lists is $O(n)$ work and $O(1)$ depth. The `map` constructing `f_bar` is $O(n)$ work and $O(1)$ depth. The function `f` differentiates the operator $\odot$ so the resulting anonymous function will have $O(k)$ work and depth since it is executed sequentially. The `scatter` is $O(n)$ work and $O(1)$ depth. Thus the reverse sweep complies with the ideal asymptotics of $O(kn)$ work and $O(k\log(n))$ depth.

6.1.5 Internal Representation

```
let h_part = HistOMap(n, [is, vs], [HistOp(w, ne, replicate w ne,
 (*))], id)
let (ys, dst, h_part) = Map(w, [dst, h_part, ys],
                        λx y z → (y * z, x * y, x * z))
let flag = Map(n, [iota n, sis],
           λi j → if i == 0 then true
                  else sis[i-1] != j)
let (_,ls,_,rs_rev) =
  ScanOMap(n, [iota n, flag],
           [λf1 x f2 y →
              if f2 then (f1 || f2, y)
              else (f1 || f2, x * y),
           (false, ne),
           λf1 x f2 y →
             if f2 then (f1 || f2,y)
             else (f1 || f2, x * y),
           (false, ne) ],
           λi f →
             let v_exc =
               if f then ne
               else svs[i-1]
             let (rflag, v_revexc) =
               if i == 0 then (true, ne)
               else (flag[n-i],svs[n-i])
             in (f, v_exc, rflag, v_revexc) )
let vs = ScatterOMap([sis, iota n, ls, siota],
                     λsi i li sio →
                       let hi_bar =
                         if si < w && -1 < si
                         then h_part[si] else 0
                       (sio, rs_rev[n-i-1] * hi_bar * li)),
                      ([n], ne, Scratch α n) )
```

Listing 6.2: Generic case of reduce-by-index in the internal Futhark representation (without sorting code)

The compiler will optimise programs aggressively so the compiled program looks quite different from the pseudocode shown in section 4.1. The program has the same overall structure but is obfuscated by fusing of parallel operations and changes to data representations. You can look up the internal constructs in table 2.3 and explanations in section 2.2, if you need a reminder. The changes does not affect the asymptotics of the program but optimisations such as fusion might remove big runtime constants. It is very expensive to access memory from the GPU and fusion may replace global-memory accesses with register accesses, thus having the potential of greatly improving performance.

Listing 6.2 is a beautified version of the internal representation of a differentiated reduce-by-index program. As an example, we derive a Futhark program containing just a multiplication reduce-by-index `let ys = reduce_by_index dst (*) ne is vs` (disabling special cases so generic case is applied). The forward sweep computes one histogram `HistOMap` on line 1 into a dummy destination of neutral elements with the given operator $\odot \equiv *$. There is no suitable map to fuse with so the map function is the identity function `id`. The `Map` on lines 2-4 is not suitable because (1) the input lengths are different and (2) the map is on the result of the histogram. However, the three maps computing `ys`, $\overline{\texttt{h\_part}}$, and $\overline{\texttt{dst}}$ are fused together. The flag array for the left scan is computed roughly the same as in the Futhark pseudocode.

The important differences come when `ls` and `rs` are computed (ln. 7-24). This is done by fusing a scan on a map which does the computations from lines 11-14 in listing 6.1. Recall that the construct `ScanOMap` is a function composition `scan` ∘ `map`, i.e. the map function is applied first. The map essentially rotates the segments such that for each segment, the first element is set to the neutral element and the last is removed. This is used by the left scans because the built-in scan is inclusive but these scans are exclusive. Analogously, it also rotates segments the other way where the last element is set to the neutral element and the first is removed, which is used for right scans. Additionally, the map computes the flag array for the right scans.
`ScanOMap` is given two scanning functions since it scans over two value arrays generated by the map. The scan functions are duplicates as the same operators and neutral elements are used to scan on both `ls` and `rs`. Notice that it results in a reversed `rs`. The `ScatterOMap` has a map function that effectively reverses `rs` which saves a read and a write compared to a program explicitly reversing `rs` (ln. 25-31). The map function also checks that indices are in bounds of the destination array (ln. 27-29) and computes the adjoint update of the values array (ln. 30).

Another remark is that the `iota`s do not introduce runtime overheads when they are used as input to GPU parallel constructs. The GPU can simply use the thread ID instead of the iota element.

6.2 Reduce-by-index with Addition

```
1  -- Primal:
2  --    let ys = reduce_by_index dst (+) ne is vs
3  -- Forward sweep:
4    let dst_cpy = copy dst
5    let ys = reduce_by_index dst_cpy (+) ne is vs
6  -- Reverse sweep:
7    d̅s̅t̅ += y̅s̅
8    v̅s̅ += map (λi →
9                 if i < w && -1 < i
10                then y̅s̅[i]
11                else 0
12              ) is
```

Listing 6.3: Pseudocode for reverse AD of reduce-by-index with addition

The compiler identifies the addition pattern `let ys = reduce_by_index dst (+) ne is vs` with the array types $is \in [n]i64, vs \in [n]\alpha, dst \in [w]\alpha$. In the addition case, the strategy is quite simple. In the single reduce version, all value adjoints were updated by the adjoint of the reduce result. Here each value adjoint is dependent on the result adjoint in that value's bucket:

$$\overline{vs_i} += \frac{\partial(dst_j + vs_{j_1} + \cdots + vs_i + \cdots + vs_{j_q})}{\partial vs_i}\overline{ys_j} = \overline{ys_j}$$

where $j_1, \dots, j_q$ are the q indices of values going into bucket $j = is_i$. The update for the destination is analogous, $\overline{dst_i} += \overline{ys_i}$. Thus the adjoint updates only depend on adjoints of their corresponding bucket, so the translation from the single reduce derivative is straight-forward. However in contrast to single reduce, not all value updates will be updated. The program should check that the corresponding index `is[i]` of a value is in-bounds of the destination array and if not its adjoint should not be updated as it does not affect the result.

The generated Futhark pseudocode of the addition case is presented in listing 6.3. Notice that the forward sweep uses a copy of the destination array, `dst` (ln. 4-5). This is because reduce-by-index consumes its destination which is problematic if the reverse sweep uses the original destination. Consider the program in figure 6.1a where `dst` is used in a statement before the reduce-by-index. If the destination is not copied, the reverse AD implementation constructs the program derivative in figure 6.1b. This differentiated program will give a type

```
1 let a = map (**2) dst
2 let b =
3   reduce_by_index dst (+)
      0 is vs
4 let x = map2 (+) a b
5 in x
```

(a) The original program before applying AD

```
 1 -- forward sweep:
 2 let a = map (**2) dst
 3 let b =
 4   reduce_by_index dst (+)
       0 is vs
 5 let x = map2 (+) a b
 6 -- reverse sweep:
 7 let a̅ = x̅
 8 let b̅ = x̅
 9 let d̅s̅t̅ = x̅
10 let v̅s̅ =
11   map (λi →
12           if i < w && -1 < i
13           then x̅[i]
14           else 0
15         ) is
16 let d̅s̅t̅ +=
17      map (*2*a̅) dst
```

(b) Incorrect reverse AD program derivative

Figure 6.1: An example program and its type-faulty reverse AD derivative demonstrating the necessity of copying destination.

error because dst is consumed (ln. 4) and then later accessed (ln. 17). Thus we need to copy the destination in the forward sweep before it is consumed by reduce-by-index.

We now turn our attention to the reverse sweep. The destination adjoints are updated directly by adding $\overline{ys}$ (ln. 7). Adjoints of values are updated by the $\overline{ys}$ element in its bucket index if it is in-bounds of destination (ln. 8-12). Otherwise, the adjoint is not updated (update is set to 0).

6.2.1 Work-Depth Analysis

The forward sweep does a copy on dst taking $O(w)$ work and $O(1)$ depth. The reduce-by-index takes $O(n)$ work and $O(\log(n))$ depth. Thus the total work-depth for the forward sweep is $O(w+n)$ work and $O(\log(n))$ depth. The work does not agree with the original program work which was just $O(n)$. However as mentioned in the generic case, the number of buckets w is usually less than the number of values n. Thus the forward sweep (usually) complies with work-depth asymptotic.

The reverse sweep takes $O(n)$ work and $O(1)$ depth which complies with the desired work-depth.

6.3 Reduce-by-index with Multiplication

```
-- Primal, assuming vs: [n]α, is: [n]i64, dst: [w]α:
--    let ys = reduce_by_index dst (*) ne is vs
-- Forward sweep:
  let (ps, zs) = map (λv → if v == 0 then (1,1) else (v,0)) vs
  let non_zero_prod =
    reduce_by_index (replicate w ne) (*) ne is ps
  let zero_count =
    reduce_by_index (replicate w 0) (+) 0 is zs
  let h_part = map2 (λp c → if c == 0 then p else 0)
                    non_zero_prod zero_count
  let ys = map2 (*) dst h_part
-- Reverse sweep:
  d̅s̅t̅ += map2 (*) h_part y̅s̅

  let part_bar = map2 (*) dst y̅s̅
  v̅s̅ +=
    map2 (λi v →
      if -1 < i && i < w then
        let zr_cts = zero_count[i]
        let pr_bar = part_bar[i]
        let nz_prd = non_zero_prod[i]
        in if zr_cts == 0
           then pr_bar * (nz_prd / v)
           else if zr_cts == 1 and v == 0
                then nz_prd * pr_bar
                else 0
      else 0
    ) is vs
```

Listing 6.4: Pseudocode for multiplication case of reduce-by-index

The compiler recognises the pattern `let ys = reduce_by_index dst (*) ne is vs` with array types $is \in [n]i64, vs \in [n]\alpha, dst \in [w]\alpha$. The strategy has the same overall structure as single reduce multiplication with the subcases determined by the number of zeros (see section 4.2). Again the forward sweep is modified to include the non-zero product and the number of zeros but now for each bucket instead of just once. Like in the generic case, we construct the partial histogram `h_part` (the histogram without `dst`). Then the adjoints of `dst` can be updated by $\overline{dst_i} += \frac{\partial dst_i \cdot h\_part_i}{\partial dst_i}\overline{ys_i} = h\_part_i \cdot \overline{ys_i}$. Adjoint updates of the values are found by looking up the number of zeros in the corresponding bucket to determine which subcase of multiplication should be used. The subcases for each bucket are the same as for a single reduce: no zeros, one zero or multiple zeros in the bucket. The only difference is that we need to check if the corresponding bucket of a given value actually exists. A value with an out-of-bounds index will not affect the result, so it is updated by 0.

The generated Futhark pseudocode is shown in listing 6.4. The forward sweep counts the number of zeros and product of non-zero elements in each bucket. First we map over `vs` creating two arrays: `ps` which is `vs` where zeros are set to 1, and `zs` which flags the zero elements (ln. 4). The first array is used to compute the product of non-zeros in each bucket where we simply do the multiplication reduce-by-index (ln. 5-6). As destination, we use an array of neutral elements. We use addition reduce-by-index on `zs` to count number of zeros in each bucket (ln. 7-8). Here we use an array of zeros since 0 is neutral element of $+$. Instead of a reduce-by-index, we can first compute a partial histogram `h_part` with a map over `non_zero_prod` and `zero_count`. `h_part` is the histogram without regards for destination values (ln. 9-10). Then `ys` is found by multiplying `dst` on `h_part` with a `map` (ln. 11). The two maps of constructing `h_part` and `ys` will be fused by the compiler.

Notice that unlike the addition case there is no need to copy the destination array since we do not use reduce-by-index on it and thus it is not consumed.

The reverse sweep updates the adjoints of destination `dst` and values `vs`. $\overline{\texttt{dst}}$ is updated by $\overline{dst_i} += h\_part_i \cdot \overline{ys_i}$ (ln. 13). As mentioned in the generic case, $=$ could replace $+=$ without changing the semantics since $\overline{\texttt{dst}}$ cannot have been updated in previous reverse sweep statements.

The remainder of the reverse sweep, is dedicated to updating $\overline{\texttt{vs}}$ (ln. 16-28). It is updated by a map whose function (1) checks the corresponding index is in-bounds and (2) identifies the subcase of the bucket. When there are no zero elements the update is $\overline{vs_i} += ys_i / vs_i \cdot \overline{ys_i}$, with one zero element at index i the update is $\overline{vs_i} += x_i \cdot \overline{ys_i}$ and otherwise there are no updates. Notice that `nz_prd` does not take `dst` into account so `dst` is multiplied with $\overline{\texttt{ys}}$ at line 15.

6.3.1 Work-Depth Analysis

The forward sweep includes two reduce-by-index with $O(1)$ work operators on n length arrays, so they have $O(n)$ work and $O(\log(n))$ depth. These two are fused in the internal representation. Additionally, it has three maps on arrays of length w where the last two are fused. These

take $O(w)$ work and $O(1)$ depth. So the final work-depth of the forward sweep is $O(w+n)$ work and $O(\log(n))$ depth. The work does not comply with the original program which is $O(n)$ work but as mentioned in the addition case we usually have $w < n$. Thus the forward sweep generally complies with work-depth asymptotic.
The reverse sweep has three maps where the first two are fused. All operators are $O(1)$ work-depth. The fused maps use arrays of length w and the last map is on arrays of length n. Thus the reverse sweep use $O(w+n)$ work and $O(1)$ which is in agreement with the forward sweep.

6.4 Reduce-by-index with Min-Max

```
-- Primal, assuming vs: [n]α, is: [n]i64, dst: [w]α:
--    let ys = reduce_by_index dst minmax ne is vs
-- Forward sweep:
  let dst_cpy = copy dst
  let (ys, ys_inds) = zip vs (iota n)
                    ▷ reduce_by_index dst_cpy argminmax (ne,-1) is
 -- Reverse sweep:
  $\overline{dst}$ += map2 (λx_ind b → if y_ind == -1
                                 then b
                                 else 0
                  ) ys_inds $\overline{ys}$

  vs_ctrbs = map2 (λi b → if i == -1
                               then 0
                               else $\overline{vs}$[i] + b
                        ) ys_inds $\overline{ys}$
  $\overline{vs}$ = scatter $\overline{vs}$ ys_inds vs_ctrbs
```

Listing 6.5: Pseudocode for reverse AD of reduce-by-index with min-max

This case is applied to the pattern `let ys = reduce_by_index dst ⊙ ne is vs` where ⊙ is either `max` or `min`. In the min-max case only one element in each bucket affects the outcome. Thus it is only this element for each bucket whose adjoint will be updated. The strategy is that the forward sweep computes the min/max element and its index, so the reverse sweep knows which adjoints to update. If two elements are equal, the min/max operators are defined to choose the least index. The index of the bucket element in `dst` is noted as -1. Notice that this means the element in `dst` is prioritised when choosing the least index of max/min element. For destination adjoints, we make an update if the index of the max array is -1. For value adjoints, we make an update if the bucket index is in-bounds of destination and is different from -1.

The generated Futhark pseudocode is shown in listing 6.5. The forward sweep is modified to compute both the index ys_inds and the value of the min/max element ys for each bucket (ln. 4-6).

With the same argument as in the addition case, the destination array needs to be copied in the forward sweep before it is consumed by reduce-by-index (ln. 4).
The reverse sweep updates the adjoints for destination and values. $\overline{\texttt{dst}}$ is updated if it is a min/max element by $\overline{dst_i} += \overline{ys_i}$ (ln. 8-11). This is done by mapping over the max elements and their indices, and if an index k is -1 we know dst[k] is the min/max element of the bucket. Thus the adjoint of dst[k] is updated by $\overline{\texttt{ys}}$[k]. If the index is any other than -1, we use a zero update.

Like $\overline{\texttt{dst}}$, the element of the adjoint update for vs is computed by mapping over ys_inds and $\overline{\texttt{ys}}$ (ln. 13-17). $\overline{\texttt{vs}}$ is updated when the index is different from -1. To minimise the number of reads from memory, we first make an array vs_ctrbs that holds value adjoints if a value is the max/min element and 0 otherwise (ln. 13-16). Notice that vs_ctrbs will only compute the adjoints for in-bound indices because it is created with a map over ys_inds. Any value where its bucket index is out-of-bounds, will be ignored as it simply will not exist in ys_inds. The adjoints in vs_ctrbs are scattered into $\overline{\texttt{vs}}$ using ys_inds. Mind that scatter ignores any illegal indices so -1 indices where destination is the min/max element will not be used.

6.4.1 Work-Depth Analysis

The work-depth for the forward sweep is analogous to the addition case resulting in $O(w+n)$ work and $O(\log(n))$ depth. Likewise, this complies with work-depth of the original program assuming $w < n$.
The reverse sweep uses two maps and a scatter, where the two maps are fused. The maps take $O(w)$ work and $O(1)$ depth. The scatter takes $O(n)$ work and $O(1)$ depth. In total, the reverse sweep has $O(w+n)$ work and $O(1)$ depth, which agrees with the forward sweep.

6.5 Vectorising the Special Cases

As an addition to cases relevant to single-reduce, we have vectorised the special cases as well. This simply means that the compiler detects if the reduce operator is a sequence of nested maps where the innermost operation is a special case operator. Then it applies the corresponding special case using nested maps.

Instinctively, the vectorised cases maintain the ideal work-depth of their special case, since the only difference is maps using $O(n)$ work and $O(1)$ depth.

Scan Implementation 7

This section explains our reverse AD implementation of scan. Recall the semantics of `scan`:

$$\textbf{scan} : (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to [n]\alpha$$
$$\textbf{scan}\ \odot\ e_\odot\ [x_0, \ldots, x_{n-1}] \equiv [x_0, x_0 \odot x_1, \ldots, x_0 \odot \cdots \odot x_{n-1}]$$

where $\odot$ is an associative operator. In this chapter, we explain how the strategy relates to the theory of chapter 5, go through our considerations with regards to optimisations, and argue for correctness when the solution surpass what is trivial. Our implementation includes every part of the reverse AD scan theory (see chapter 5), as well as some optimisations we have identified.

Like the reduce-by-index implementation chapter, this chapter presents each case of scan with the assistance of the Futhark pseudocode generated by the compiler for that case. Then the code is examined with work-depth analysis to assure that the asymptotics agree with those of the corresponding primal program.

7.1 Generic Case

The compiler identifies the statement `let ys = scan ⊙ ne as` where $\odot$ is associative and `as` : `[n]`α. Listing 7.1 shows the code generated by the compiler in a Futhark-esque pseudocode version. The code is mostly the same as the reverse sweep presented in chapter 5, except the partial derivatives computing Jacobians are written as code instead of mathematical notation (ln. 10 and 21). The Jacobians in `cs` are made by mapping vjp_λ over a $d \times d$ identity matrix (ln. 10). The Jacobian code is generated at compile time by:

$$f_i = vjp_\lambda(idMat[i],\ \odot), \forall i \in 0..d-1 \tag{7.1}$$

where `idMat` is a $d \times d$ identity matrix and `idMat[i]` is the unit vector with the 1 in position `i` represented as a tuple. The compiler applies vjp_λ to each row of the identity matrix. Notice that this is only possible because the size of the element tuples d is known at compile time.

```
-- Primal:
--    let ys = scan ⊙ ne as
-- Forward sweep:
let ys = scan ⊙ ne as
-- Reverse sweep:
let cs =
  map (λi → if i==n-1
             then idMat
             -- see equation (7.1)
             else (f_0 rs[i] as[i+1], ..., f_{d-1} rs[i] as[i+1])
      ) (iota n)
let lin_o (d1,c1) (d2,c2) = (d2 + c2 * d1, c2 × c1)
let (r̅s̅,_) =
  scan lin_o (replicate d 0, idMat) (reverse y̅s̅) (reverse cs)
    ▷ reverse

let a̅s̅ +̅=
  map (λi r̅i̅ ai →
          if i==0
          then r̅i̅
          else f i as[i+1]
      ) (iota n) r̅s̅ as
-- idMat is dxd identity matrix
```

Listing 7.1: Futhark pseudocode for reverse AD of scan with generic case operators. Mind that $*$ denotes matrix-vector multiplication and $\times$ denotes matrix-matrix multiplication.

When updating the adjoint $\overline{\texttt{as}}$, we use:

$$f\ i = vjp_{\lambda}(\overline{\texttt{rs}}[i],\ \odot' rs[i-1])$$

where $\odot'$ is $\odot$ where the argument order is flipped.

In the IR, the construction of cs is fused with the scan in a ScanOMap. The mapping function also reverses cs and $\overline{\texttt{ys}}$. As cs is only used for constructing $\overline{\texttt{rs}}$, it does not need to instantiate it in memory, saving nd^2 writes and reads. The reverse on the scan result is fused with the map updating $\overline{\texttt{as}}$ such that a reversed version $\overline{\texttt{rsr}}$ of $\overline{\texttt{rs}}$ is saved and the map uses iota n to read $\overline{\texttt{rsr}}$ in reverse order by $\overline{\texttt{rsr}}$[n-i-1].

7.1.1 Limitations

The implementation does have some limitations with regards to what input types and scanning operators are allowed. Firstly, if $\odot$ is an array operator, it must be a map such that the code can be transformed by the vectorised special case. An example of a currently disallowed operator would be matrix multiplication where each matrix is represented as a 2D array. Cases like this are discussed in chapter 11. As an alternative, a $d \times d$ matrix can be represented as a tuple with d^2 entries, which is a functional case of the implementation.

If the input array includes tuples, the tuple elements must all be of the same type which are either integers or floats. The reason is that linear function composition applies vector addition and matrix multiplication on the result adjoints $\overline{\texttt{ys}}$ and Jacobians `cs`, so the types need to match.

7.1.2 Work-Depth Analysis

The work-depth asymptotics of the primal program is $O(wn)$ work and $O(w \log(n))$ depth when w is the work of operator $\odot$. An ideal AD implementation should not affect the asymptotic work-depth.

The construction of `cs` is $O(wn)$ work and $O(w)$ depth, assuming $f_0, \ldots, f_{d-1}$ generated by vjp_λ does not affect the $O(w)$ work of the operator $\odot$. Tuple size d is a constant so it is not included in the asymptotic measures. Constructing $\overline{\texttt{rs}}$ costs $O(wn)$ work and $O(w \log(n))$ depth for the `scan` and each of the `reverse`s are $O(n)$ work and $O(1)$ depth.

The `map` updating $\overline{\texttt{as}}$ is $O(wn)$ work and $O(w)$ depth, assuming `f` does not change the work of $\odot$.

The forward sweep is simply the unchanged primal program so it does not affect the work-depth asymptotics. Thus the total work-depth asymptotics of the differentiated program is $O(wn)$ work and $O(w \log(n))$ depth which is in agreement with the primal program.

7.2 Scan with Addition

```
-- Primal:
--     let ys = scan (+) 0 as
-- Forward sweep:
let ys = scan (+) 0 as
-- Reverse sweep:
let a̅s̅ +̅= scan (+) 0 (reverse y̅s̅) ▷ reverse
```

Listing 7.2: Pseudocode for the addition case of scan

The compiler recognises the pattern `let ys = scan (+) 0 as` where `as:[n]`α. Listing 7.2 shows the generated code when deriving a `scan` with addition operator. The reverse sweep is unchanged from the code shown in chapter 5. In the IR, the `reverse` of $\overline{\texttt{ys}}$ is fused with the reverse sweep `scan`.

The work-depth of the primal program is $O(n)$ work and $O(\log(n))$ depth. The differentiated program consists of two addition `scan`s, which have $O(n)$ work and $O(\log(n))$ depth, and two `reverse`s using $O(n)$ work and $O(1)$ depth. Thus the differentiated program is in agreement with the work-depth of the primal program.

7.3 Vectorised Scan Operators

When the scan operator is vectorised, the compiler transforms the `scan` such that it either fits the generic or addition case. Thus the compiler only has a transformation rule for vectorised operators, whose result is then piped into vjp again to construct the derivative. Recall the transformation from the theory section:

```
scan (map ⊙) $\overline{ne}$ xs ⇒
  transpose xs ▷ map (scan ⊙ ne) ▷ transpose
```

Mind that the transformation changes the work-depth asymptotics. For a vectorised operator with input array type $[n][m]\alpha$, the transformed program will have n scans on arrays of size m. The depth of the original program is $O(mw\log(n))$ when the scan operator is w work since the `map` will be sequentialised. Thus the depth is changed from $O(mw\log(n))$ to $O(w\log(m))$ by the transformation. This is an improvement of work-depth.

7.4 Jacobian Patterns

In the generic case, we compute Jacobians for the purpose of constructing $\overline{\texttt{rs}}$. These Jacobians are scanned over with linear function composition lin_o which includes expensive matrix-matrix and matrix-vector multiplication. Luckily, there are cases in which the matrix operations can be optimised. We can identify parts of the Jacobian which will always be zero and thus not affect the result. These entries can therefore be removed such that the differentiated program only needs to handle a smaller part of the Jacobians.

We have implemented two Jacobian patterns: *ZeroQuad* and *MatrixMul*. There exist more relevant patterns, some of which are discussed in chapter 11. We begin by defining the two patterns, for each of them presenting the mathematical rationale of how lin_o can be optimised in the pattern and the updated version of the generated Futhark pseudocode (section 7.4.1, section 7.4.2). The last section 7.4.3 explains how the patterns are recognised and handled at compile time.

Dimension	Explanation
n	Length of input array `as`
d	Tuple size of `as` element type
q	Dimension of the $q \times q$ diagonal matrices in the Jacobian
k	Number of diagonal matrices

Variable	Type	Usage
`as`, `ys`, $\overline{\texttt{as}}$, $\overline{\texttt{ys}}$	$[n]\overbrace{(\alpha,\ldots,\alpha)}^{d}$	All patterns
`cs`	$[n]\overbrace{(\alpha,\ldots,\alpha)}^{d^2}$	Generic pattern
`cs_i`, $i \in [1..k]$	$[n]\overbrace{(\alpha,\ldots,\alpha)}^{q^2}$	ZeroQuad
`Ms`	$[n]\overbrace{(\alpha,\ldots,\alpha)}^{q^2}$	MatrixMul
$v_{seg.i}$, $i \in [1..k]$	$[q]\alpha$	ZeroQuad and MatrixMul

Figure 7.1: Table of definitions and types, provided as a look-up table for the section of Jacobian patterns (section 7.4).

Since this section introduces a fair amount of dimensions and variables, we have chosen to include index tables for the sake of readability. The index tables are shown in figure 7.1. The information is also given in the text continuously as it are used.

7.4.1 ZeroQuad Pattern

One pattern case, which we have observed is a pattern on a $d \times d$ matrix (represented with d^2 tuple):

$$\begin{bmatrix} \begin{bmatrix}\mathbf{M}_1\end{bmatrix} & & \ldots & 0 \\ & \begin{bmatrix}\mathbf{M}_2\end{bmatrix} & & \vdots \\ \vdots & & \ddots & \\ 0 & \ldots & & \begin{bmatrix}\mathbf{M}_k\end{bmatrix} \end{bmatrix}$$

We call this pattern *ZeroQuad* (short for zero quadrant). The pattern has multiple $q \times q$ matrices along the diagonal and zero entries everywhere else. Notice the diagonal matrices fill out the matrix exactly so $d \bmod q = 0$. There are $1 < k = d/q$ diagonal matrices $\mathbf{M}_1, \ldots, \mathbf{M}_k$ which may contain both zero and non-zero entries.

The following section explains the possible optimisations when the Jacobians of `cs` are of the ZeroQuad pattern. The idea is to inspect the matrix operations of linear function composition lin_o where the Jacobians are used, and identify inefficiencies. First, we consider matrix multiplication. When two ZeroQuad matrices are multiplied, we have:

$$\begin{aligned}
&\begin{bmatrix} \begin{bmatrix}\mathbf{M}_{1.1}\end{bmatrix} & & \cdots & 0 \\ & \begin{bmatrix}\mathbf{M}_{1.2}\end{bmatrix} & & \vdots \\ \vdots & & \ddots & \\ 0 & \cdots & & \begin{bmatrix}\mathbf{M}_{1.k}\end{bmatrix} \end{bmatrix} \times \begin{bmatrix} \begin{bmatrix}\mathbf{M}_{2.1}\end{bmatrix} & & \cdots & 0 \\ & \begin{bmatrix}\mathbf{M}_{2.2}\end{bmatrix} & & \vdots \\ \vdots & & \ddots & \\ 0 & \cdots & & \begin{bmatrix}\mathbf{M}_{2.k}\end{bmatrix} \end{bmatrix} \\
&\quad = \begin{bmatrix} \begin{bmatrix}\mathbf{M}_{1.1}\end{bmatrix} \times \begin{bmatrix}\mathbf{M}_{2.1}\end{bmatrix} & & \cdots & 0 \\ & \begin{bmatrix}\mathbf{M}_{1.2}\end{bmatrix} \times \begin{bmatrix}\mathbf{M}_{2.2}\end{bmatrix} & & \vdots \\ \vdots & & \ddots & \\ 0 & \cdots & & \begin{bmatrix}\mathbf{M}_{1.k}\end{bmatrix} \times \begin{bmatrix}\mathbf{M}_{2.k}\end{bmatrix} \end{bmatrix}
\end{aligned}$$

We observe that the result is another ZeroQuad matrix so multiplication preserves the pattern. Notice that the operation corresponds to multiplying every set of diagonal matrices $\mathbf{M}_{1.i} \times \mathbf{M}_{2.i}$. This means that multiplying two ZeroQuad matrices corresponds to multiplying k smaller matrices, i.e. k independent matrix multiplications. Linear function composition includes matrix-vector multiplication as well, which has the following form with ZeroQuad:

$$\begin{bmatrix} \begin{bmatrix}\mathbf{M}_{1}\end{bmatrix} & & \cdots & 0 \\ & \begin{bmatrix}\mathbf{M}_{2}\end{bmatrix} & & \vdots \\ \vdots & & \ddots & \\ 0 & \cdots & & \begin{bmatrix}\mathbf{M}_{k}\end{bmatrix} \end{bmatrix} \times \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_d \end{bmatrix} = \begin{bmatrix} \begin{bmatrix}\mathbf{M}_{1}\end{bmatrix} \times v_{seg.1} \\ \begin{bmatrix}\mathbf{M}_{2}\end{bmatrix} \times v_{seg.2} \\ \vdots \\ \begin{bmatrix}\mathbf{M}_{k}\end{bmatrix} \times v_{seg.k} \end{bmatrix}$$

where $v_{seg.i}$ is the i'th segment of v when splitting it into k segments of length q. We observe that this corresponds to k independent matrix-vector products, multiplying a diagonal matrix $\mathbf{M}_i$ with vector segment $v_{seg.i}$. Thus the linear function composition on ZeroQuad matrices has the form:

$$\begin{aligned}
lin_o\ (v_1, m_1)\ (v_2, m_2) &= (v_2 + m_2 \times v_1, m_2 \times m_1) \\
&= \left(\begin{bmatrix} v_{2.1} \\ \vdots \\ v_{2.d} \end{bmatrix} + \begin{bmatrix} \begin{bmatrix}\mathbf{M}_{2.1}\end{bmatrix} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \begin{bmatrix}\mathbf{M}_{2.k}\end{bmatrix} \end{bmatrix} \times \begin{bmatrix} v_{1.1} \\ \vdots \\ v_{1.d} \end{bmatrix}, \right.
\end{aligned}$$

$$\begin{bmatrix} \begin{bmatrix}\mathbf{M}_{2.1}\end{bmatrix} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \begin{bmatrix}\mathbf{M}_{2.k}\end{bmatrix} \end{bmatrix} \times \begin{bmatrix} \begin{bmatrix}\mathbf{M}_{1.1}\end{bmatrix} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \begin{bmatrix}\mathbf{M}_{1.k}\end{bmatrix} \end{bmatrix} \Bigg)\Bigg)$$

$$= \left(\left(\begin{bmatrix} v_{2seg.1} + \begin{bmatrix}\mathbf{M}_{2.1}\end{bmatrix} \times v_{1seg.1} \\ \vdots \\ v_{2seg.k} + \begin{bmatrix}\mathbf{M}_{2.k}\end{bmatrix} \times v_{1seg.k} \end{bmatrix}, \right.\right.$$

$$\left.\left. \begin{bmatrix} \begin{bmatrix}\mathbf{M}_{2.1}\end{bmatrix} \times \begin{bmatrix}\mathbf{M}_{1.1}\end{bmatrix} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \begin{bmatrix}\mathbf{M}_{2.k}\end{bmatrix} \times \begin{bmatrix}\mathbf{M}_{1.k}\end{bmatrix} \end{bmatrix} \right)\right)$$

This means that lin_o on ZeroQuad matrices corresponds to applying lin_o k times on the diagonal matrices and their corresponding q-length segments of the vectors. Therefore the `scan` with lin_o can be transformed to k scans with lin_o on arrays with smaller elements. We can optimise lin_o by rewriting it to:

$$\begin{aligned} &lin_{oZQ}\ (v_{1.seg.i}, \mathbf{M}_{1.i})\ (v_{2.seg.i}, \mathbf{M}_{2.i}) = \\ &\quad = \left(v_{2seg.i} + \begin{bmatrix}\mathbf{M}_{2.i}\end{bmatrix} \times v_{1.seg.i}, \begin{bmatrix}\mathbf{M}_{2.i}\end{bmatrix} \times \begin{bmatrix}\mathbf{M}_{1.i}\end{bmatrix}\right) \end{aligned}$$

Notice that lin_{oZQ} works on vectors and square matrices with dimension q instead of d, because lin_{oZQ} is used for the k scans. It is applied to a vector segment and the corresponding diagonal matrix. Mind that lin_o and lin_{oZQ} are actually the same operator but they are applied on vectors and matrices of different dimensions. We define lin_{oZQ} for the sake of clarity so referrals to linear function composition in the generic and ZeroQuad cases are more easily distinguishable.

7.4.1.1 Efficiency Analysis

The new linear function composition optimises the differentiated program by excluding redundant operations and storage of zero entries. We begin by examining the memory efficiency. The vectors and matrices given to lin_{oZQ} have dimensions q and $q \times q$ respectively where lin_o takes d and $d \times d$. lin_{oZQ} uses q memory for a vector and q^2 for a matrix but it is run k times so the total memory usage is $kq = d$ for a vector and $kq^2 = k(d/k)^2 = d^2/k$ for a matrix. The lin_o operator uses d for a vector, which is the same as a vector in k calls to lin_{oZQ}, but lin_o uses d^2 on a matrix. Thus the lin_{oZQ} spends a fraction of the matrix memory needed by lin_o:

$$\frac{kq^2}{d^2} = \frac{k(d/k)^2}{d^2}$$

$$= \frac{d^2/k}{d^2}$$
$$= \frac{1}{k}$$

I.e. lin_{oZQ} spends $^1/_k$ of what lin_o spends on matrices. For example, consider a differentiated program that scans with 2×2 matrix multiplication. The Jacobians are 4×4 which fit the ZeroQuad pattern with $q = 2$ and $k = 2$. Thus a differentiated program utilising the ZeroQuad pattern will spend $^1/_2$ of the memory spend on matrices in lin_o. As mentioned the amount spend on vectors is the same for the two versions since the whole vector may affect the result.

Operation-wise, lin_{oZQ} has $O(q+q^2) = O(q^2)$ work for the first tuple entry and $O(q^3)$ work for the second entry. Then the total work of scanning with lin_{oZQ} is $O(nk(q^2+q^3)) = O(nkq^3)$ and the depth is $O(q^3 \log(n))$ because the k scans are run in parallel. The work-depth asymptotics of lin_o are $O(nd^3)$ work and $O(d^3 \log(n))$ depth. This means the optimised version has better asymptotics since $nkq^3 = nk(^d/_k)^3 = n(^{d^3}/_{k^2}) < nd^3$. However, the tuple size d is considered a constant so actually this provides only a constant improvement, however large.

7.4.1.2 High-Level Futhark Code for ZeroQuad Pattern

The generated Futhark pseudocode for ZeroQuad Jacobians is shown in listing 7.3 which is a modified version of the generic case in listing 7.1. At compile time we extract the diagonal matrices from the Jacobians (ln. 7). Instead of cs, we define k Jacobian arrays of length n where $\texttt{cs}_i$ is the i'th diagonal matrices from the Jacobians. Notice that the neutral elements are set to identity matrices of dimension $q \times q$ since the diagonal matrices are of this dimension (ln. 6).

The elements of $\overline{\texttt{ys}}$ (vectors) need to be split into the k segments such that they match the dimension of the diagonal matrices. First, d arrays $\overline{\texttt{ys}}_i$ are created holding the i'th elements of all the vectors (ln. 9). Then in the k scans with lin_{oZQ} the q-length segments are fetched by zipping the relevant $\overline{\texttt{ys}}_i$ arrays. The scans are still over n-length arrays but the elements are the i'th diagonal matrices and vector segments (ln. 11-17).
For the final map updating as, the scan results $\overline{\texttt{rs}}_1, \ldots, \overline{\texttt{rs}}_k$ are combined into a single vector as if it had applied lin_o to the unmodified Jacobians.

In the IR, the following operations are fused together in a ScanOMap: the construction of $\texttt{cs}_1, \ldots, \texttt{cs}_k$, the k scans and reverses on the scan input. The reverses on the scan results are fused with the map updating $\overline{\texttt{as}}$ in a Map construct.

Recall that zip and unzip are removed at compile time with no runtime overheads so

```
-- Forward sweep:
let ys = scan ⊙ ne as
-- Reverse sweep:
let (cs$_1$,...,cs$_k$) =
  map (λi → if i==n-1
              then (idMatQ,...,idMatQ)
              else extract matrices from (f$_0$ rs[i] as[i+1], ..., f$_{d-1}$ rs[i] as[i+1])
      ) (iota n)
let ($\overline{\text{ys}}_1$,...,$\overline{\text{ys}}_d$) = unzip $\overline{\text{ys}}$
let lin$_{oZQ}$ (d1,c1) (d2,c2) = (d2 + c2 * d1, c2 × c1)
let (($\overline{\text{rs}}_1$,_),...,($\overline{\text{rs}}_k$,_)) =
  (scan lin$_{oZQ}$ (replicate q 0, idMatQ) (reverse (zip $\overline{\text{ys}}_1$ ... $\overline{\text{ys}}_q$)) (reverse cs$_1$)
      ▷ reverse,
   ...,
   scan lin$_{oZQ}$ (replicate q 0, idMatQ) (reverse (zip $\overline{\text{ys}}_{(k-1)q+1}$ ... $\overline{\text{ys}}_d$)) (reverse cs$_k$)
      ▷ reverse
  )

let $\overline{\text{as}}$ $\overline{\text{+}}$=
  map (λi $\overline{\text{ri}}$ ai →
         if i==0
         then $\overline{\text{ri}}$
         else f i as[i+1]
      ) (iota n) (zip $\overline{\text{rs}}_1$ ... $\overline{\text{rs}}_k$) as
-- idMatD is dxd identity matrix
-- idMatQ is qxq identity matrix
```

Listing 7.3: Pseudocode for the generic case of scan with ZeroQuad Jacobians. Mind that $*$ denotes matrix-vector multiplication and × denotes matrix-matrix multiplication.

splitting $\overline{\text{ys}}$ in segments is free.

7.4.2 MatrixMul Pattern

Another Jacobian pattern has a very similar shape to ZeroQuad:

$$\begin{bmatrix} \begin{bmatrix} M \end{bmatrix} & & \dots & 0 \\ & \begin{bmatrix} M \end{bmatrix} & & \vdots \\ \vdots & & \ddots & \\ 0 & \dots & & \begin{bmatrix} M \end{bmatrix} \end{bmatrix}$$

It is the same pattern as ZeroQuad except the diagonal matrices **M** are equal. We call the pattern *MatrixMul* because it occurs for matrix multiplication $A \times B$ when differentiating with respect to A. Mind that the pattern might appear with other operators as well. When two matrices of this shape are multiplied, we have:

$$\begin{aligned}
&\begin{bmatrix} \begin{bmatrix}\mathbf{M}_1\end{bmatrix} & & \dots & 0 \\ & \begin{bmatrix}\mathbf{M}_1\end{bmatrix} & & \vdots \\ \vdots & & \ddots & \\ 0 & \dots & & \begin{bmatrix}\mathbf{M}_1\end{bmatrix} \end{bmatrix} \times \begin{bmatrix} \begin{bmatrix}\mathbf{M}_2\end{bmatrix} & & \dots & 0 \\ & \begin{bmatrix}\mathbf{M}_2\end{bmatrix} & & \vdots \\ \vdots & & \ddots & \\ 0 & \dots & & \begin{bmatrix}\mathbf{M}_2\end{bmatrix} \end{bmatrix} \\
&\qquad = \begin{bmatrix} \begin{bmatrix}\mathbf{M}_1\end{bmatrix} \times \begin{bmatrix}\mathbf{M}_2\end{bmatrix} & & \dots & 0 \\ & \begin{bmatrix}\mathbf{M}_1\end{bmatrix} \times \begin{bmatrix}\mathbf{M}_2\end{bmatrix} & & \vdots \\ \vdots & & \ddots & \\ 0 & \dots & & \begin{bmatrix}\mathbf{M}_1\end{bmatrix} \times \begin{bmatrix}\mathbf{M}_2\end{bmatrix} \end{bmatrix}
\end{aligned}$$

So matrix multiplication with matrices of the MatrixMul pattern, results in k identical matrix multiplications $\mathbf{M}_1 \times \mathbf{M}_2$. Notice that the operation preserves the MatrixMul pattern. The matrix-vector multiplication is of the form:

$$\begin{bmatrix} \begin{bmatrix}\mathbf{M}\end{bmatrix} & & \dots & 0 \\ & \begin{bmatrix}\mathbf{M}\end{bmatrix} & & \vdots \\ \vdots & & \ddots & \\ 0 & \dots & & \begin{bmatrix}\mathbf{M}\end{bmatrix} \end{bmatrix} \times \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_d \end{bmatrix} = \begin{bmatrix} \begin{bmatrix}\mathbf{M}\end{bmatrix} \times v_{seg.1} \\ \begin{bmatrix}\mathbf{M}\end{bmatrix} \times v_{seg.2} \\ \vdots \\ \begin{bmatrix}\mathbf{M}\end{bmatrix} \times v_{seg.k} \end{bmatrix}$$

In the matrix-vector product, all k segments of the vector are multiplied by the diagonal matrix **M**, so it corresponds to multiplying **M** on k different vectors. These vectors are obtained by splitting vector v in k segments of length q. Now we can write linear function composition in the following form.

$$\begin{aligned}
lin_o\ (v_1, m_1)\ (v_2, m_2) &= (v_2 + m_2 \times v_1, m_2 \times m_1) \\
&= \left(\begin{bmatrix} v_{2.1} \\ \vdots \\ v_{2.d} \end{bmatrix} + \begin{bmatrix} \begin{bmatrix}\mathbf{M}_2\end{bmatrix} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \begin{bmatrix}\mathbf{M}_2\end{bmatrix} \end{bmatrix} \times \begin{bmatrix} v_{1.1} \\ \vdots \\ v_{1.d} \end{bmatrix}, \right.
\end{aligned}$$

$$\left. \begin{bmatrix} \begin{bmatrix}\mathbf{M}_2\end{bmatrix} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \begin{bmatrix}\mathbf{M}_2\end{bmatrix} \end{bmatrix} \times \begin{bmatrix} \begin{bmatrix}\mathbf{M}_1\end{bmatrix} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \begin{bmatrix}\mathbf{M}_1\end{bmatrix} \end{bmatrix} \right) \Bigg)$$

$$= \left(\begin{bmatrix} v_{2seg.1} + \begin{bmatrix}\mathbf{M}_2\end{bmatrix} \times v_{1seg.1} \\ \vdots \\ v_{2seg.k} + \begin{bmatrix}\mathbf{M}_2\end{bmatrix} \times v_{1seg.k} \end{bmatrix}, \begin{bmatrix} \begin{bmatrix}\mathbf{M}_2\end{bmatrix} \times \begin{bmatrix}\mathbf{M}_1\end{bmatrix} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \begin{bmatrix}\mathbf{M}_2\end{bmatrix} \times \begin{bmatrix}\mathbf{M}_1\end{bmatrix} \end{bmatrix} \right) \Bigg)$$

This form contains a lot of duplicate operations, so we rewrite linear function composition to:

$$lin_{oMM}\ (v_1, \mathbf{M}_1)\ (v_2, \mathbf{M}_2) = \left(v_2 + \begin{bmatrix} \begin{bmatrix}\mathbf{M}_2\end{bmatrix} \times v_{1seg.1} \\ \vdots \\ \begin{bmatrix}\mathbf{M}_2\end{bmatrix} \times v_{1seg.k} \end{bmatrix}, \mathbf{M}_2 \times \mathbf{M}_1 \right)$$

The new lin_{oMM} changes the dimensions of the second entry from lin_o so the matrices are $q \times q$ instead of $d \times d$. This is unproblematic since the second tuple entry of the `scan` $\texttt{lin}_o$ is unused except inside the scan itself. The previous investigation of matrix operations with MatrixMul have shown that lin_{oMM} will respect the semantics of the first `scan` result.

7.4.2.1 Efficiency Analysis

lin_{oMM} is expected to give a significant performance boost compared to the unoptimised version lin_o both memory-wise and with respect to number of operations.

lin_{oMM} needs q^2 memory for each matrix while lin_o needs d^2. Thus a matrix in lin_{oMM} spends only a fraction of the memory spend on a matrix in lin_o:

$$\begin{aligned} \frac{q^2}{d^2} &= \frac{(d/k)^2}{d^2} \\ &= \frac{d^2/k^2}{d^2} \\ &= \frac{1}{k^2} \end{aligned}$$

So e.g. when you differentiate `scan` with 2×2 matrix multiplication, the Jacobians are 4×4. Then lin_{oMM} will use ${}^1/_{2^2} = {}^1/_4$ of the memory on matrices compared to lin_o, i.e. the optimisation eliminates ${}^3/_4$ of the matrix memory. Mind that the two versions still need the same amount of memory to store the vectors.

As the matrices are smaller, lin_{oMM} will also need less operations. The matrix multiplication uses asymptotic work $O(q^3) = O(d^3/k^3)$ while in lin_o it is $O(d^3)$. The matrix-vector mul-

```
 1 -- Forward sweep:
 2 let ys = scan ⊙ ne as
 3 -- Reverse sweep:
 4 let Ms =
 5   map (λi → if i==n-1
 6             then idMatQ
 7             else extract matrices from (f_0 rs[i] as[i+1], ..., f_{d-1} rs[i]
     as[i+1])
 8       ) (iota n)
 9 let lin_oMM ((d_1,...,d_d),c1) (d2,c2) =
10   let ts =
11     (c2 × (d_1,...,d_q),
12      ...,
13      c2 × (d_{(k-1)q+1},...,d_d))
14   in (d2 + ts, c2 × c1)
15 let (r̅s̅,_) =
16   scan lin_oMM (replicate d 0, idMatQ) (reverse y̅s̅) (reverse Ms)
17      ▷ reverse
18
19 let a̅s̅ +̅=
20   map (λi r̅i̅ ai →
21          if i==0
22          then r̅i̅
23          else f i as[i+1]
24       ) (iota n) r̅s̅ as
25 -- idMatD is dxd identity matrix
26 -- idMatQ is qxq identity matrix
```

Listing 7.4: Pseudocode for the generic case of scan with MatrixMul Jacobians. Mind that * denotes matrix-vector multiplication and × denotes matrix-matrix multiplication.

tiplication also provide a performance boost with lin_o having $O(d^2)$ work and lin_{oMM} has $O(k \cdot q^2) = O(k \cdot (d/k)^2) = O(d^2/k)$. Recall however that d is considered a constant meaning the improvement is constant.

7.4.2.2 High-Level Futhark Code for MatrixMul Pattern

Our implementation generates the code shown in listing 7.4 which like the ZeroQuad case, is a modified version of the generic case in listing 7.1. Instead of cs, we construct an array Ms holding a single $q \times q$ diagonal matrix from each Jacobian. These are extracted at compile time such that only used entries are computed at runtime.

Lines 9-14 implements the mathematical definition of lin_{oMM} directly where it pattern matches on the first vector input to construct the q-length vector segments. Then it simply scans with lin_{oMM} (ln. 15-17). Notice the neutral elements for the scan have different dimensions with a d-length vector and a $q \times q$ matrix. The update to $\overline{\texttt{as}}$ is unaffected.

In the IR, operations on lines 4-16 are fused in a `ScanOMap` construct, i.e. the construction of `Ms`, the `reverse`s of the inputs to `scan` and the `scan` itself. The `reverse` on the `scan` result and the update to $\overline{\texttt{as}}$ are fused in a `Map`.

7.4.3 Recognising Patterns at Compile Time

As mentioned, the Jacobian patterns are identified at compile time. We use vjp_λ to generate the code computing the Jacobians and this code is given to the *simplifier*. The simplifier takes a program and optimises it by removing redundancies and applying code transformations using a static analysis. In this case, we use the simplifier to identify Jacobian entries that are always zero. Mind that if one Jacobian fits a pattern then every Jacobian has the same pattern. The reason is that the simplifier examines only the derivative code for $\odot$ without any inputs.

The simplified Jacobian is then used to identify potential Jacobian patterns. Currently, all implemented patterns consists of $q \times q$ diagonal matrices so it should find the smallest possible correct q if such a q exists. The compiler creates a list of possible q sizes $[1, 2, \ldots, {}^{d}/{}_{2}]$ where the elements must divide the Jacobian completely, i.e. $d \mod q = 0$. It uses brute-force to find q by checking if all non-zero entries fit in 1×1 diagonal matrices, then 2×2, etc. The smallest possible q is chosen as it excludes as many zero entries as possible. If no q is found, the compiler applies the generic case because the matrix does not fit a Jacobian pattern. If a q is found, the compiler checks if all diagonal matrices are equal and if so the MatrixMul pattern is applied. Otherwise, it applies ZeroQuad.
It might be possible to optimise this pattern recognition, such that we do not need to use brute-force. However, the tuple size d is usually small so the overhead is not expensive. Furthermore, this analysis is done at compile time so it will not affect the runtime of differentiated programs, only the compiling time.

Our pattern recognition implementation has its limitations. Firstly, there are more relevant Jacobian patterns which have not been implemented, as discussed in chapter 11. Secondly, we apply only a static analysis and no dynamic analysis. In some cases, a pattern is not recognisable at compile time but it is at runtime because the inputs are taken into account. However, this would probably be expensive because every Jacobian should be examined when each one has a different input.

Part III

Evaluation

Validation 8

This section describes how we have validated our implementation of reverse AD scan and reduce-by-index. We have two categories of tests: the primary tests with manually selected inputs, and the secondary tests validating reverse AD against forward AD with randomly generated inputs. All the primary tests pass with our implementation with both C and CUDA backends. All secondary tests pass with the C backend, while a few encounter runtime errors with the CUDA backend, specifically reduce-by-index with vectorised operators. However, the forward AD fails in the same cases so it would indicate that the problem was not introduced by our implementation.

For the manual tests, we have systematically chosen primal programs and inputs such that the tests reach every (edge) case that we have identified. The test function receives output adjoints and input and applies vjp to the primal program. We have reused some test programs already in the Futhark AD test suit[1], which we have supplemented with extra tests for special cases. Our full test suit is found in the AD test suit of our GitHub in `futhark/tests/ad` (Larsen et al. 2022). The test outputs are either computed by hand or generated by forward AD, depending on the size of the input and output. When the input and output are relatively small, the program is typically derived by hand. Other programs would be impractical to derive by hand so the test output is generated by forward AD.

Additionally, we have tests working on randomly generated inputs that compare the full Jacobians computed by forward and reverse mode AD. Mind that the full Jacobian is costly to compute because reverse AD is run $size(output)$ times and forward AD is run $size(input)$ times. When comparing forward and reverse AD directly, we have to compute the full Jacobian as forward AD produces a column and reverse AD produces a row. Thus destination size is set to at most 100 and the number of values to at most a 1000, such that the tests are finished within a reasonable time frame. These tests handle larger data sets than our manual tests but still a small amount compared to likely "real" use-cases.

The generated input data is integers such that rounding errors or overflows do not cause random differences in the results, which would be a problem with floats.

[1] The Futhark Github, i.e. `futhark/tests/ad` (The Futhark Hackers n.d.)

8.1 Validating Reverse AD of Reduce-by-index

Tests for reduce-by-index are named `reducebyindex*.fut` where `*` is some test name. We have identified the following test categories for reduce-by-index based on the operator:

- generic case operators,
- addition,
- multiplication,
- minimum and maximum,
- multiple levels of vectorised operators,
- and vectorised special case operators.

Table 8.1 shows a sample of the tested primal programs with reduce-by-index which are differentiated with respect to destination and values using *vjp*. Some primal programs are one-liners testing reduce-by-index in isolation, while others include multiple statements. The latter are included to test if the reduce-by-index AD transformation works in collaborations with the other AD transformations. Specifically, we test that non-zero adjoints are updated correctly and the original destination array is available if needed by reverse sweep statements.

Each of the above tests categories have multiple instances reaching the relevant subcases, e.g. if the operator is `max` and the maximum element of a bucket lies in `dst`. We also ensure each special case have tests with arrays including out-of-bounds bucket indices, whose adjoints should not be updated.

The random tests comparing forward and reverse mode use the same test categories as the manual tests. The input data for indices is generated within bounds of destination. The reason is that if the indices had been generated within the full range of 64-bit integers, the large majority would be out-of-bounds, resulting in only very few adjoints actually being updated. This would effectively only test that out-of-bounds adjoints are not updated which would not be appropriate for validation.

8.2 Validating Reverse AD of Scan

The tests for reverse AD of scan are stored in files `scan*.fut` where `*` is a test name. The tests categories for reverse AD of scan are:

<table>
<tr><th>Primal Program</th><th>Operator</th></tr>
<tr><td><pre>
def sat_add (x:f32) (y:f32) =
  let sat_val = f32.i32 ((1 << 4) - 1)
  in if sat_val - x < y
     then sat_val else x + y

def primal [n][m] (is: [n]i64)
                  (dst: [m]f32, as: [n]f32) =
  reduce_by_index (copy dst) sat_add 0 is as
</pre></td><td>satadd: saturated addition
<em>Generic Case</em></td></tr>
<tr><td><pre>
def primal [n][m] (is: [n]i64)
                  (dst: [m](f32,i64),
                   vs: [n](f32,i64)) =
  reduce_by_index (copy dst) argmax (f32.lowest
   ,i64.highest) is vs
</pre></td><td>argmax
<em>Generic Case</em></td></tr>
<tr><td><pre>
def primal [n][m][k][l] (is: [n]i64)
                        (dst: [k][m][l]f32,
                         vs: [n][m][l]f32,
                         c: [k][m][l]f32) =
  let tmp = reduce_by_index
    (copy dst) (map2 (map2 (*)))
    (replicate m (replicate l 1)) is vs
  in map2 (map2 (map2 (*))) tmp c
</pre></td><td>vecmul
<em>Vectorised</em>
<em>Multiplication Case</em></td></tr>
<tr><td><pre>
def op (a1:f32,b1:f32) (a2:f32,b2:f32) =
  (b1*a2+b2*a1, b1*b2)

def primal [n] (is: [n]i64)
               (vs: [n](f32,f32)) =
  reduce_by_index (replicate 4 (0,1)) op (0,1)
   is vs
</pre></td><td>crossing tuple operator
<em>Generic Case</em></td></tr>
<tr><td><pre>
def primal [n][m] (is: [n]i64) (vs: [n]f32)
                  (dst: [m]f32) =
  let dst2 = copy dst
  let a = map (**2) dst2
  let b = reduce_by_index dst2 (*) 1 is vs
  in map2 (+) a b
</pre></td><td>checks reverse sweep access original dst
<em>Multiplication Case</em></td></tr>
</table>

Table 8.1: Sample of the primal programs for the tests of reverse AD with reduce-by-index.

Primal Program	Operator

`def primal [n] (as: [n]f32) =`
`  scan (*) 1 as`
	mul
	Generic Case with no pattern

`def primal [n] (as: [n](f32,f32,f32)) =`
`  scan (λ(a1,b1,c1) (a2,b2,c2) →`
`         (a1+a2, b1*b2, f32.max c1 c2))`
`       (0,1,f32.lowest) as`
	tuple operator
	Generic Case with Zero-Quad pattern

`def mm2by2 (a1:f32, b1:f32, c1:f32, d1:f32)`
`           (a2:f32, b2:f32, c2:f32, d2:f32) =`
`  ( a1*a2 + b1*c2`
`  , a1*b2 + b1*d2`
`  , c1*a2 + d1*c2`
`  , c1*b2 + d1*d2 )`
 	
`def primal [n] (as: [n](f32,f32,f32,f32)) =`
`  scan mm2by2 (1, 0, 0, 1) as`
	mm2by2: 2 × 2 matrix multiplication
	Generic Case with MatrixMul pattern

`def primal [n] (as: [n]f32) =`
`  scan (+) 0 as`
	add
	Addition Case

`def primal [n][k] (as: [n][k]f32) =`
`  scan (map2 (+)) (replicate k 0) as`
	vecadd
	Vectorised Addition Case

Table 8.2: Sample of the primal programs for the tests of reverse AD with scan.

- generic case operators with either
  - no statically identifiable patterns in the Jacobians,
  - ZeroQuad pattern in the Jacobians,
  - or MatrixMul pattern in the Jacobians,
- addition operator,
- and multiple levels of vectorised operators with
  - addition
  - or a generic case operator.

The primal programs shown in table 8.2 are derived using vjp with respect to input array `as`. The tuple sizes of the arguments to the operator should be varied as memory is an important concern with reverse AD of scan, e.g. we test with multiplication of 2×2 to 4×4 matrices. Both manual and random tests use the above test categories.

Performance

9

The performance of our implementation is evaluated by benchmarking differentiated programs generated by different versions of our implementation and comparing with the primal program and forward AD derivative. Section 9.1 presents the evaluation of reduce-by-index and section 9.2 does the same for scan.

All benchmarks are executed on a machine with two Intel E5-2650 CPUs and an NVIDIA RTX 2080Ti GPU using CUDA 11.3. Futhark is implemented with multiple backends of which we use the CUDA backend. The reported results are the average runtime of a 100 runs. The runtime includes all overheads except reading the input and writing the final output.
AD derivative programs have the same asymptotic work-depth as their primal program counterparts, so we would expect the derivatives to add a constant *overhead*. Here, overhead denotes the ratio between the runtimes of differentiated and primal programs. Ideally, AD introduces only a small amount of additional operations to construct the derivative program so the AD overhead should be a small constant (Griewank and Walther 2008).

9.1 Performance of Reduce-by-index

The benchmarks for reduce-by-index are run with a values array of 50 million scalars. The scalars are distributed evenly to the appropriate element type, where vectorised cases have inner arrays of length 100. The scalars are 32-bit integers in the benchmarks with operators satadd and argmax and 32-bit floats in the rest of the benchmarks. Derivatives are made with respect to values and destination using randomly chosen initial adjoints. The benchmarks are run with three different bucket numbers (destination length): 31, 1023 and 1.5 million. It is relevant to compare performance with different numbers of buckets since it affects the number of significant bits for sorting (see section 6.1.2).
The random input data for indices is generated inside a range such that they are in-bounds of the destination array. Values whose corresponding indices are out-of-bounds, are ignored so the amount of in-bounds indices greatly affects runtime. More importantly this affects the

	Primal Runtime			**Rev AD Overhead**			**Fwd AD Overhead**		
# of Buckets:	31	1023	1.5M	31	1023	1.5M	31	1023	1.5M
satadd	1146μs	1145μs	5228μs	52.4×	96.7×	33.7×	2.4×	3.5×	4.8×
argmax	2721μs	3910μs	24048μs	30.3×	38.7×	9.8×	5.0×	4.9×	3.4×

Table 9.1: Reverse and forward AD overheads for generic case operators with reduce-by-index for 31, 1023, and 1.5 million buckets. The AD overheads are the ratios between the differentiated and primal programs. Satadd is saturated addition.

runtime differently in the primal and differentiated programs because the reverse sweep will still sort the redundant elements. It is likely that real use cases would include mostly in-bounds indices so we consider it most appropriate to benchmark with in-bounds indices.

9.1.1 Performance of Generic Case with Reduce-by-index

Table 9.1 shows the reverse and forward AD overheads for generic case operators with reduce-by-index. The first observation in table 9.1 is that the reverse AD overheads are hefty compared to the forward AD overheads. This is expected because the differentiated programs sort the values and indices arrays. As explained in chapter 6, the sorting does not affect the asymptotic runtime but it is quite a big constant. We can examine the effect of sorting using table 9.2. The table shows that the sorting dominates the runtime, taking up to 82%.

As expected the runtime of Radix sort grows when increasing the number of buckets as it has to sort after more significant bits. Our expectation is that Radix sort with 1023 buckets takes twice as long as 31 buckets, and Radix sort with 1.5 million buckets takes almost twice as long as 1023 buckets[1]. This is confirmed by table 9.2. Notice that the runtime of Radix sort is independent of the operator for a fixed number of buckets. This is expected since it does not use the operator.

A surprising observation regarding sorting is that the time to permute the values using `siota` is affected by the number of buckets (see table 9.2). We would not expect this because it is constructed by simply indexing into `vs` with `siota`, namely `map (λi → vs[i]) siota`. This means that the number of memory accesses does not depend on the number of buckets. However, the sorting of the values becomes more expensive with increasing numbers of buckets (see table 9.2). Our theory is that this is caused by a better cache performance with small

[1] Radix sort with 31 buckets has 3 iterations sorting 6 bits, 1023 buckets has 6 iterations sorting 12 bits, and 1.5 million buckets has 11 iterations sorting 22 bits.

		AD Runtime		
	# of Buckets:	31	1023	1.5M
satadd	Radix Sort	35k μs	70k μs	130k μs
	Permutation of `vs`	6k μs	14k μs	16k μs
	Other tasks	19k μs	27k μs	31k μs
	Total:	60k μs	111k μs	177k μs
	Sorting Percentage:	68%	76%	82%
argmax	Radix Sort	35k μs	70k μs	130k μs
	Permutation of `vs`	11k μs	29k μs	32k μs
	Other tasks	37k μs	52k μs	73k μs
	Total:	83k μs	151k μs	235k μs
	Sorting Percentage:	55%	66%	69%

Table 9.2: Profiling reduce-by-index with generic case operators to examine the effect sorting has on the runtime with different numbers of buckets. Radix sort runtimes include the construction of `siota` and `sis`. The permutation of `vs` is the gather operation that constructs `svs`. Sorting percentage is the fraction of runtime, we spend on Radix sort and permutation of `vs`.

	Rev AD Runtime			**Fwd AD Runtime**		
# of Buckets:	31	1023	1.5M	31	1023	1.5M
satadd	1.9 *s*	110 *s*	260k *s*	140k *s*	200k *s*	1,300k *s*
argmax	2.6 *s*	160 *s*	350k *s*	680k *s*	960k *s*	4,200k *s*

Table 9.3: Estimated runtimes of computing the full Jacobian with reverse and forward AD for reduce-by-index with generic case operators.

numbers of buckets because it gets more hits in the L2 cache when indexing into `vs`. The average distance between values going to the same bucket is smaller meaning there is a higher probability of a cache hit. The `Scatter0Map` updating the `vs` adjoints has a similar runtime behavior, which is likely due to the L2 cache hit rate as well since it uses `siota` to scatter the updates.

Notice that repositioning the values in the argmax case is around twice as expensive as in the satadd case (see table 9.2). This is expected as the argmax case reduces over 2 arrays while the satadd case reduces over a single array.

When examining the behaviour of the reverse AD overheads in table 9.1, we observe that they peak at 1023 buckets. The reason is that for small bucket numbers, the GPU is able to apply a faster version of reduce-by-index (Henriksen, Hellfritzsch, et al. 2020).[2] This means the primal runtimes are close for small numbers of buckets but the runtimes of the reverse sweep still increase. This causes the reverse AD overhead to grow when going from 31 to 1023 buckets.

The overheads of reverse and forward AD can be seen in table 9.1. Recall that reverse AD computes a row of the Jacobian while forward computes a column, which means the AD runtimes of the two modes are not directly comparable. To make them comparable, we approximate runtimes of computing the full Jacobian, meaning reverse AD runtimes are multiplied by the number of outputs and forward AD runtimes are multiplied by the number of inputs. The inputs are the values and the destination arrays and the output is the resulting destination array. As the benchmarks are made with 50 million values, forward AD has 50 million more runs

[2] Reduce-by-index is implemented in multiple versions depending on the operator and number of buckets. For small numbers of buckets, it uses a single-pass version that constructs partial histograms which can fit into shared memory. With many buckets, it uses a multi-pass version instead as the partial histograms cannot fit in shared memory (Henriksen, Hellfritzsch, et al. 2020).

	Actual AD Overheads			**Theoretical AD Overheads**		
# of Buckets:	31	1023	1.5M	31	1023	1.5M
satadd	52.4×	96.7×	33.7×	∼ 22×	∼ 37×	∼ 10×
argmax	30.3×	38.7×	9.8×	∼ 18×	∼ 21×	∼ 4×

Table 9.4: The reverse AD overheads of reduce-by-index and the potential reverse AD overheads if the programs used CUB's implementation of Radix sort.

than reverse AD. Table 9.3 shows estimated runtimes for computation of the full Jacobian. It is evident that reverse AD is preferable with this number input values. The observations demonstrate excellently that even with great runtime constants, the asymptotic work-depth of a program will ultimately dictate the runtime behaviour. Mind that the current Radix sort implementation in Futhark is around 50× slower than CUB's state-of-the-art Radix sort implementation. We can make an estimate of the AD overhead using CUB's Radix sort by using Amdahl's Law (Amdahl 1967). For example, the theoretical speed-up of reduce-by-index with satadd and 1.5 million buckets is:[3]

$$Speedup = \frac{1}{(1-F)+\frac{F}{S}} = \frac{1}{(1-0.73)+\frac{0.73}{50}} =\sim 4\times$$

where F is the fraction of the program that is enhanced by a factor of S. This means the differentiated program would be around 4× faster. Table 9.4 shows estimated reverse AD overheads if CUB's Radix sort had been used. The overheads are significantly lower since sorting constitutes a large fraction of the observed runtimes.

9.1.2 Performance of Special Cases with Reduce-by-index

Table 9.5 shows the benchmarks for reduce-by-index with special case operators. The special cases are significantly faster as expected. We will consider the operators one-by-one and examine their benchmarks.

The add operator follows the addition special case. The reverse AD overhead goes below 1× meaning the differentiated program becomes faster than the primal. The reason is that the

[3] Mind the speedup is only on the Radix sort fraction of the program, not the permutation of `vs` which is otherwise considered part of the sorting cost.

	Primal Runtime			**AD Overhead**					
				Generic			*Special case*		
# of Buckets:	31	1023	1.5M	31	1023	1.5M	31	1023	1.5M
add	1165μs	1169μs	4742μs	41.8×	77.4×	31.7×	1.0×	0.9×	0.8×
vecadd	2428μs	6906μs	38629μs	6.6×	2.3×	0.8×	1.6×	0.5×	0.3×
mul	1147μs	1149μs	5255μs	52.3×	96.2×	33.5×	2.6×	2.8×	12.9×
vecmul	2432μs	13385μs	46524μs	406.7×	61.7×	16.1×	12.9×	9.4×	3.2×
max	1346μs	1365μs	5068μs	45.0×	81.5×	36.1×	2.5×	3.1×	5.1×
vecmax	1948μs	14479μs	46522μs	503.2×	57.1×	16.1×	–	6.3×	2.4×

Table 9.5: The reverse AD overheads in the special case operators of reduce-by-index applying the rewrite rule of either the generic case or appropriate special case.

forward sweep result is not used so it is removed, leaving only a cheap `map`. Thus the AD overhead becomes lower with increasing numbers of buckets as the depth of the `map` is constant but the depth of the reduce-by-index in the primal program is growing. We observe similar results for the vectorised addition operator vecadd that matches the vectorised addition case.

The mul operator matches the multiplication special case. The runtimes and the AD overheads with 31 and 1023 buckets are almost the same. This is because the runtime is dominated by the forward sweep reduce-by-index which uses the optimised version with a small number of buckets. Additionally, multiplication is supported by the hardware giving an even more efficient reduce-by-index. With 1.5 million buckets, it switches to a slower version. However, this does not explain the increase in AD overhead from 2.8× with 1023 buckets to 12.9× with 1.5 million buckets. By profiling the program, we have identified that the overhead increase is caused by the final `map` of the reverse sweep. This `map` indexes uncoalesced into arrays whose length is the number of buckets, so our theory is that the slow-down is caused by the caching being more efficient with small numbers of buckets.

The vectorised multiplication operator vecmul has decreasing AD overheads in contrary to the mul operator. The first observation is that the primal runtime increases significantly with the number of buckets. The reverse sweep does not increase as much in comparison, so the reverse

	Rev AD Overhead			Fwd AD Overhead		
# of Buckets:	31	1023	1.5M	31	1023	1.5M
add	1.0×	0.9×	0.8×	1.3×	1.4×	2.4×
vecadd	1.6×	0.5×	0.3×	1.8×	1.7×	2.1×
mul	2.6×	2.8×	12.9×	2.3×	3.3×	4.5×
vecmul	12.9×	9.4×	3.2×	7.5×	3.2×	2.0×
max	2.5×	3.1×	5.1×	2.1×	2.9×	4.9×
vecmax	–	6.3×	2.4×	9.6×	2.8×	2.0×

Table 9.6: The reverse and forward AD overheads of reduce-by-index with special case operators. The vecmax operator with 31 buckets throws a runtime CUDA error (invalid argument).

sweep constitutes a smaller part of the runtime lowering the overheads.

The max operator matches the min/max special case. The AD overhead increases with the number of buckets. The reason is that the AD transformation converts the max operator, which is supported by hardware, to an argmax operator that is not supported by the hardware. This impacts the efficiency of reduce-by-index (Henriksen, Hellfritzsch, et al. 2020).
The vecmax overheads show the same behaviour as the vecmul overheads. Again, the reason is that the primal runtimes increase greatly, resulting in the forward sweep dominating the runtime of the differentiated program. Thus the AD overheads decrease.

Table 9.6 shows the forward and reverse AD overheads for comparison. We observe that the reverse mode overheads are only slightly slower and even faster in some cases, specifically add and vecadd operators. As with the generic case operators, forward AD needs has 50 million more runs than reverse AD to compute the full Jacobian. Evidently, reverse mode is superior to forward mode using special case operators.

In addition to the benchmarks shown here, appendix A have some benchmarks for intermediate versions of the reduce-by-index AD implementation. During the development phase, we located points in the implementation with multiple possible solutions where it were difficult to argue purely theoretically for a single solution. We have done benchmarks to check which

	Primal Runtime	**Rev AD Overhead**			**Fwd AD Overhead**
	—	*Generic*	*ZeroQuad*	*MatMul*	—
mm2by2	1574μs	31.6×	18.8×	8.9×	2.4×
mm3by3	2492μs	–	31.3×	11.3×	5.9×
mm4by4	5709μs	–	–	11.3×	6.7×
lin_o	1506μs	5.5×	4.4×	4.0×	2.0×
lin_o,2by2	1680μs	137.8×	∅	∅	4.3×

Table 9.7: Benchmarks of reverse AD scan with generic case operators, potentially applying a Jacobian pattern. Forward AD overheads are shown for comparison. The AD overheads are the ratio between the differentiated and primal programs. – means the pattern is applicable but the program is not executable; ∅ means the pattern is not applicable.

solution (if possible) performs the best for the problem.

9.2 Performance of Scan

The benchmarks are done with random input data of a 100 million 32-bit float scalars. These scalars are distributed into the appropriate tuple size for the operator so e.g. the scan with operator mm2by2 has $100M/2^2 = 25M$ 4-tuples and mm4by4 has $100M/4^2 = 6.25M$ 16-tuples. Thus the input array lengths for each operator will differ but all benchmarks will work on the same amount of data (or approximately the same). The derivatives are made with respect to the input array using randomly chosen initial adjoints.

9.2.1 Performance of Scan with Generic Case

Table 9.7 shows benchmarks for selected `scan` programs comparing the runtimes of the primal program and the differentiated programs using reverse and forward AD.

The reverse AD overheads are subdivided into each Jacobian pattern which is reported when the pattern is applicable and the program executable. Notice that if some pattern A is applicable, then that *should* imply the differentiated program using pattern A is executable. However, this is not the case. Consider the case of scan with 3×3 matrix multiplication (mm3by3 in table 9.7). The Generic pattern is applicable but the GPU throws a runtime error

		ScanOMap		Map		Total Mem	AD Overhead	
		Reads	*Writes*	*Reads*	*Writes*		*Expected*	*Actual*
mm2by2	*Primal*	4n	4n	–	–	$\sim 8n$	–	–
	Generic	12n	24n	8n	4n	$\sim 48n$	∼ 6.0×	31.6×
	ZeroQuad	12n	16n	8n	4n	$\sim 40n$	∼ 5.0×	18.8×
	MatMul	12n	12n	8n	4n	$\sim 36n$	∼ 4.5×	8.9×
lin_o	*Primal*	2n	2n	–	–	$\sim 4n$	–	–
	Generic	5n	8n	4n	2n	$\sim 19n$	∼ 4.8×	5.5×
	ZeroQuad	5n	6n	4n	2n	$\sim 17n$	∼ 4.3×	4.4×
	MatMul	5n	5n	4n	2n	$\sim 16n$	∼ 4.0×	4.0×

Table 9.8: Case study of mm2by2 and lin_o for comparison of expected and actual reverse AD overheads. Expected overheads are based on approximate numbers of global memory accesses, using the ratio between primal and differentiated program accesses.

so the program is not executable. Specifically, a kernel failed to launch with error code 1 invalid argument. Similar errors occur at all GPU backends. Most likely, the cause for this error is that the kernel requests more shared memory than available. The Generic case of mm3by3 scans over 90 arrays that with a block size of 256 requires $90 \cdot 4 \cdot 256 = 92$kb of shared memory. However, the maximum amount of shared memory per CUDA block is 64kb with the GPU used for the benchmarks.

We will now consider whether the results with the Jacobian patterns are reasonable according to the theoretical background. We expect the AD overhead to be roughly the ratio between the number of memory accesses in the primal and differentiated programs. The rationale is that memory accesses are far more expensive than scalar operations so they will dominate the runtime. By inspecting the IR of the differentiated program, we can count the approximate number of global memory accesses (see table 9.8). Consider the operator case of `scan` with lin_o. The primal program has a single scan that writes $2n$ and reads $2n$ elements. The differentiated program consists of two IR constructs, `ScanOMap` and `Map`, which have a total of $\sim 19n$ memory accesses when using the Generic Jacobian pattern. By this logic we expect a Generic case AD overhead with lin_o scan to be approximately:

$$\frac{19n}{4n} = \sim 4.8\times$$

This expected overhead matches roughly the observed overhead of 5.5×. Table 9.8 shows that the expected overheads for lin_o matches the actual overheads relatively well. The difference may lie in the amount of used shared memory and extra instructions.

With the mm2by2 operator the actual overheads are much higher than expected, when we only consider the global memory accesses (see table 9.8). By profiling, we found that the runtime of reverse AD Generic case is dominated by `ScanOMap` constituting 95% of the runtime. Part of the explanation is likely that the scan uses far more shared memory, were e.g. the Generic pattern scans over 24 arrays and the primal program scans over only 4. This will reduce the number of active blocks per streaming multiprocessor (SM), i.e. it cannot run as many threads in parallel. Additionally, it has far more instructions than the primal. This also explains why the actual overheads come closer to the expected when using less memory in the more specialised Jacobian patterns.

The number of global memory accesses cannot alone explain the observed overheads for mm2by2 because the reverse sweep uses a much larger amount of shared memory and far more instructions. Instead we will look at the runtime ratio between the Jacobian pattern cases. We might expect the runtime ratios to roughly match the ratio between the number of scan arrays. This provides a measure for the shared memory usage and the number of instructions in the `ScanOMap` which seemingly has a huge impact on the AD overheads in the mm2by2 case. The Generic pattern case of mm2by2 scans over 20 arrays. The ZeroQuad case scans over 12 arrays with lin_{oZQ} so the ratio is $24/16 =\sim 1.5\times$,[4] which roughly matches the runtime ratio of $31.6/18.8 =\sim 1.7\times$ speed-up. The MatrixMul case scans over 8 arrays with lin_{oMM} giving an scan array ratio of $24/12 =\sim 2\times$, where the runtime ratio $31.6/8.9 =\sim 3.5\times$ is actually far better. The reason might be that the MatrixMul case uses far less shared memory so the global memory accesses dominates the runtime. Also the Generic pattern case has $O(d^3)$ computations while MatrixMul has only $O(q^3) = O((d/k)^3)$.

Similarly, for the case of mm3by3 the scan array ratio is $45/27 =\sim 1.7\times$ going from ZeroQuad to MatrixMul pattern whereas the runtime ratio is better $31.3/11.3 = 2.8\times$. The same explanation as mm2by2 is applicable.

The operator mm4by4 only has one executable differentiated program which is the MatrixMul case. The overhead from the primal is 11.3×, exactly like the overhead of the MatrixMul case with mm3by3. Mind that in the differentiated programs mm2by2, mm3by3 and mm4by4 with the MatrixMul pattern, the reverse sweep scan multiplies matrices of the same size as the primal program. Thus the reverse AD overhead should increase gradually with the dimension of the matrices $d \times d$ since the matrix-vector multiplication in lin_{oMM} has $O(k^2)$ work and $k = d$. We observe that the overhead does *not* increase from mm3by3 to mm4by4 with MatrixMul which

[4] The scans of the forward and reverse sweeps are fused, so we add 4 arrays.

	Primal Runtime	Rev AD Overhead	Fwd AD Overhead
add	$1545\mu s$	1.96×	1.95×
vecadd	$260770\mu s$	0.03×	1.54×
mul	$1544\mu s$	4.35×	1.97×
vecmul	$262093\mu s$	0.05×	1.53×

Table 9.9: Benchmarks for special cases addition and vectorised with reverse AD scan. Multiplication is not a special case but is included as reference point for vectorised multiplication.

might be caused by the primal being rather slow compared to the primal of mm3by3 providing an overhead of $5709/2492 = 2.3\times$ instead of the rough overhead expectation $4^2/3^2 =\sim 1.8\times$. The primal slowdown is likely caused by an increase in the shared memory usage.

The lin_o,2by2 program is a considerable outlier as the overhead is 137×, far from expected. The Jacobian is 6×6 so it scans over 42 arrays (36 from Jacobians and 6 from vectors). Again, a likely explanation is that it uses far more instruction and a lot of shared memory, meaning it cannot run as many blocks per SM.

In addition to the reverse AD overheads, table 9.7 presents the forward AD overheads. The forward overheads are a few times better than our reverse AD overheads in all cases except in lin_o,2by2 where our overhead is over 100×. Recall that the rewrite rule of forward AD introduces only a single derivative statement for each statement in the primal program, so intuitively we see that a single run of forward AD should be faster than a run of reverse AD. This matches the observations in table 9.7 showing that forward AD is a few times faster than reverse AD. As mentioned in the theory part (see section 3.1), forward AD is preferable when the numbers of inputs and outputs are equal which is the case with scan. This also means we do not need to estimate the full Jacobian runtimes as forward and reverse AD should be run the same number of times.

9.2.2 Performance of Scan with Special Cases

Table 9.9 shows benchmarks for differentiated programs constructed with the reverse AD scan special cases as well as the primal program runtimes and the forward AD overheads.
We will examine the overheads. The reverse AD transformation of scan with addition intro-

duces an extra scan with addition and a `reverse`. However, in the benchmarked program the forward sweep is never used so it is removed by the simplifier. The reverse sweep consist of a scan with addition and a `reverse`, i.e. the differentiated program is a `ScanOMap` and a `Map`. Since it has the double amount of memory accesses, we would expect the reverse AD overhead to be approximately 2×. This is in accordance with the observations with an overhead of 1.96×. The forward AD overhead is approximately the same because it also adds a single `Map` to the primal program.

In vectorised addition, the compiler first applies the vectorise transformation to the program then the addition case. Mind that the transformation is applied first so the forward sweep is also transformed. Recall that the transformation changes the asymptotic depth of the program from $O(m\log(n))$ to $O(\log(m))$ where m is the length of the innermost arrays. In this benchmark, the inner arrays have $m = 200$ elements and the outer array has $n = 100,000,000$ elements. Since $m \ll n$, the differentiated program greatly out-performs the primal program and forward AD which does not transform the program. The same holds for the case of vectorised multiplication which is slightly slower because it applies the generic case after transformation instead of the addition case.

Conclusion 10

This thesis can naturally be split into two main goals: constructing reverse AD of reduce-by-index and extending and optimising reverse AD of scan. For each of these operations, we have four notable objectives: examining the previous work on reverse AD of the operation, construct an efficient strategy, implement the rewrite rule in the Futhark compiler, and evaluate the performance of differentiated programs using our implementation. Furthermore, we examine the theory of automatic differentiation and its performance concerns.

The first objective is to explain the theoretical background of our work. We present the Futhark language and parts of the internal representation, which has been an important part of our thought process when constructing strategies for the implementation.
Automatic differentiation has been presented, both reverse and forward mode. Reverse AD lays the foundation for our work, while forward mode is an important reference point for comparison of performance and it is essential to the discussion of when it is appropriate to apply reverse mode or forward mode AD.
Others have published work on reverse AD of reduce-by-index and scan in Futhark, i.e. Schenck et al. 2022. We present their strategy and approach as part of the first objective. We managed to simplify their rewrite rule for scan resulting in a more optimised version of the generated Futhark code.

The second objective is to analyse and construct efficient strategies for the reverse AD rewrite rules. This includes performance considerations, e.g. Futhark's IR and GPU behaviour. For reduce-by-index, we have formulated a rewrite rule with generic case operators by transformation of the primal program into a semantically equivalent program with two statements of lower complexity. We have applied AD rewrite rules to the transformed statements independently to construct the adjoint updates for the values and destination arrays. The presented rewrite rule preserves the expected work-depth asymptotics of the primal program, when assuming the number of buckets is smaller than the number of values which is a reasonable assumption (Schenck et al. 2022). Additionally, Schenck et al. 2022 loosely explains the strategy of the special cases while we present the rewrite rules for these formally using Futhark

pseudocode.
For reverse AD of scan, we have used the strategies as presented by Schenck et al. 2022, and simplified the rewrite rule for generic case in such a way that it should provide a performance benefit. By using Futhark's simplifier on Jacobians of generic case scan operators, we have identified and constructed specialised rewrite rules for two Jacobian patterns, *ZeroQuad* and *MatrixMul*.

The third objective is to make an implementation of reverse AD for reduce-by-index and scan based on the presented strategies and rewrite rules. We have succeeded and validated the implementation by testing differentiated programs against the forward AD implementation and with manually constructed tests. Our implementation is freely available in our GitHub Larsen et al. 2022.[1]

The fourth objective is experimental evaluation of the performance. We have done benchmarks of primal program and their differentiated counterparts with reverse and forward mode AD. We discuss the runtime behaviour and assess the results with respect to the expectation from AD theory, Futhark's implementation and GPU behaviour.
In reduce-by-index, the behaviour is mostly as expected, where the reverse AD overheads are relatively high because of sorting. Using Amdahl's Law, we have estimated the reverse AD overheads using a state-of-the-art radix sort, instead of Futhark's own, resulting in up to $\sim 4\times$ speedup. As expected, the reverse AD overheads are higher than forward AD overheads but reverse AD significantly outperforms forward AD on computation of the full Jacobian as we have $size(output) \ll size(input)$. Additionally, the benchmarks confirm that the specialised rewrite rules for the special cases do provide a significant performance benefit, up to $\sim 70\times$ speedup compared to the generic case.
In the performance evaluation of scan, we observe that the specialised Jacobian cases provide up to $\sim 3.5\times$ speedup in comparison with the Generic pattern case. Computation of the full Jacobian with generic case operators is faster with forward mode AD, which is expected since input and output have the same size. In the special cases however, we observe that reverse AD is able to outperform forward AD in some cases and otherwise have competitive performance.

In summary, our research presents, analyses and evaluates our solutions for reverse mode AD of reduce-by-index and scan. We have implemented our solution in Futhark's compiler and confirmed that in many cases, their performance is competitive compared to Futhark's forward AD.

[1] We have contributed mainly to the files `Hist.hs`, `SOAC.hs` and `Scan.hs` of the `src/Futhark/AD/Rev` folder

Future Work 11

This chapter contains possible areas of extension and optimisation to reverse AD of reduce-by-index and scan in Futhark.

11.1 Additional Jacobian Patterns

It is possible to include even more Jacobian patterns of scan operators. E.g. we have observed the following pattern in the Jacobian of lin_o,2by2:

$$\begin{bmatrix} \begin{bmatrix}\mathbf{M}_1\end{bmatrix} & 0 \cdots 0 \\ \begin{matrix} 0 \\ \vdots \\ 0 \end{matrix} & \begin{bmatrix} & & \\ & \mathbf{M}_2 & \\ & & \end{bmatrix} \end{bmatrix}$$

This pattern has two square diagonal matrices of different dimensions. By the same arguments used in the *ZeroQuad* pattern case, this pattern allows the reverse sweep scan to be transformed into two scans on fewer elements. This has the potential of providing a speedup and less memory usage compared to the Generic pattern case. Additionally, the pattern can be extended to include any combination of different size square diagonal matrices.

11.2 Other Suggestions

The are several limitations and inefficiencies of this implementation, including:

- As mentioned, sorting constitutes the majority of the runtime in generic case reduce-by-index. It would certainly be preferable to use a faster sorting algorithm, e.g. CUB's state-of-the-art Radix sort implementation which is roughly 50× faster than Futhark's own.

- Currently, reverse AD of scan only works with tuples of primitives, i.e. apart from vectorised operators, array operators does not work. A straight-forward reverse AD implementation will not be work-efficient since the element dimension is dependent on the input. Thus the size of the Jacobians are not constant for a given program unlike tuples of primitives.

Bibliography

Amdahl, Gene M. (1967). “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560.

Shiloach, Yossi and Uzi Vishkin (1982). “An $O(n^2 \log n)$ Parallel MAX-FLOW Algorithm”. In: *Journal of Algorithms* 3.2, pp. 128–146. ISSN: 0196-6774. DOI: https://doi.org/10.1016/0196-6774(82)90013-X.

Blelloch, Guy E. (1989). “Scans as Primitive Parallel Operations”. In: *IEEE Transactions on Computers* 38.11, pp. 1526–1538. DOI: 10.1109/12.42122.

— (1990). “Prefix Sums and Their Applications”. In: *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers Inc., pp. 35–60.

Griewank, Andreas (2003). “A mathematical view of automatic differentiation”. In: *Acta Numerica* 12, pp. 321–398.

Griewank, Andreas and Andrea Walther (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Second. USA: Society for Industrial and Applied Mathematics.

Domke, Justin (2009). *Automatic Differentiation: The most criminally underused tool in the potential machine learning toolbox?* URL: https://justindomke.wordpress.com/2009/02/17/automatic-differentiation-the-most-criminally-underused-tool-in-the-potential-machine-learning-toolbox/ (visited on 02/28/2022).

Henriksen, Troels (Dec. 2017). “Design and Implementation of the Futhark Programming Language (Revised)”. PhD thesis. Department of Computer Science, University of Copenhagen.

Henriksen, Troels, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea (2017). “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*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, pp. 556–571. ISBN: 9781450349888. DOI: 10.1145/3062341.3062354.

Baydin, Atilim Günes, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind (2018). “Automatic Differentiation in Machine Learning: a Survey”. In: *Journal of Machine Learning Research* 18, pp. 1–43.

Elsman, Martin, Troels Henriksen, and Cosmin E. Oancea (2018a). *Parallel Programming in Futhark*. URL: https://futhark-book.readthedocs.io.

— (2018b). *Parallel Programming in Futhark. 6. A Parallel Cost Model for Futhark Programs*. URL: https://futhark-book.readthedocs.io/en/latest/parallel-cost-model.html.

— (2018c). *Parallel Programming in Futhark. 8.1 Segmented Scan*. URL: https://futhark-book.readthedocs.io/en/latest/regular-flattening.html#segmented-scan.

Henriksen, Troels (2018). *The Futhark Debugger*. URL: https://futhark-lang.org/blog/2018-09-16-the-futhark-debugger.html (visited on 04/05/2022).

Henriksen, Troels, Sune Hellfritzsch, Ponnuswamy Sadayappan, and Cosmin Oancea (2020). “Compiling Generalized Histograms for GPU”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’20. Atlanta, Georgia: IEEE Press. ISBN: 9781728199986.

Henriksen, Troels (2021). *Radix Sort*. https://github.com/diku-dk/sorts/blob/master/lib/github.com/diku-dk/sorts/radix_sort.fut. GitHub repository. Commit SHA: d8925894d251aa5cc5d926c50c07d8c257f6af7d.

Larsen, Ulrik and Lotte Bruun (2022). *Our contributions to the Futhark compiler*. https://github.com/UlleLarsen/futhark/tree/ad-genred-opt.

Schenck, Robert, Ola Rønning, Troels Henriksen, and Cosmin E. Oancea (Feb. 2022). “AD for an Array Language with Nested Parallelism”. In: *arXiv e-prints*.

The Futhark Hackers (n.d.). *Futhark*. URL: https://github.com/diku-dk/futhark.

Appendices

Fine-tuning the Implementation A

As explained in section 6.1.2, the indices that are out-of-bounds of the destination array, are mapped to a single out-of-bounds bucket. This can be done either before or inside the sorting loop. Table A.1 shows that it is slightly faster to place the `map` inside the loop, so this is the chosen solution for the implementation. Recall that the out-of-bounds bucket is set to the histogram size, which means the loop only needs $\lceil \log(hist\_size + 1)/2 \rceil$ iterations (see section 6.1.2). Table A.2 shows a great performance boost, especially for small numbers of buckets as expected. In the generic case of reduce-by-index, a flag array is constructed for the segmented scans (see listing 6.2). The flag array can either be constructed in a `Map` before the `ScanOMap` or it can be in-lined in the `ScanOMap`. Table A.3 shows that the unfused programs are just slightly faster for the benchmarks with over 1 million values. Data-parallel programs are likely to take a large number of inputs to justify the overhead of transferring data between the CPU and GPU. Thus we assess that the unfused solution is most appropriate.

	Inside Sorting Loop			**Before Sorting Loop**		
Input Size:	1M	25M	50M	1M	25M	50M
satadd	1831μs	55044μs	110764μs	1894μs	55806μs	112242μs
argmax	2486μs	74336μs	151247μs	2545μs	75000μs	152442μs

Table A.1: Benchmarking to identify the ideal placement of mapping out-of-bounds indices to bucket $hist\_size$

	Rev AD Overhead					
	Significant Bits			*All Bits*		
# of Buckets:	31	1023	1.5M	31	1023	1.5M
satadd	52.4×	96.7×	33.7×	308.3×	326.3×	74.3×
argmax	30.3×	38.7×	9.8×	138.4×	106.1×	18.7×

Table A.2: The reverse AD overhead of reduce-by-index with generic case operators when sorting either after all bits or just the significant bits.

	Fused			**Not Fused**		
Input Size:	1M	25M	50M	1M	25M	50M
satadd	1826μs	55240μs	111250μs	1831μs	55062μs	110746μs
argmax	2499μs	74446μs	151278μs	2547μs	74345μs	151256μs

Table A.3: Benchmarks to identify whether it is beneficial to fuse the construction of flag array with the `ScanOMap` in reduce-by-index with generic case operators.