UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Msc Thesis

Reverse mode automatic differentiation of histograms in Futhark

Søren Brix
fvs630@ku.dk

Supervisor: Cosmin Eugen Oancea

Department of Computer Science
May 31, 2022

Abstract

This thesis presents a method of differentiating the parallel `Futhark`-construct `reduce_by_index` by integrating a new rewrite rule into the pipeline of the compiler, transforming the original code into a program that computes the adjoint of its input using reverse mode automatic differentiation. This thesis provides the relevant background of automatic differentiation and its forward- and reverse mode, along with the `Futhark` language itself. It will cover the main rewrite rule for reverse mode, and how it is used to derive the reverse mode rewrite rule for `reduce_by_index`. Lastly we cover an implementation of the derived rule, and evaluate its correctness and performance.

Contents

1 Introduction

Machine Learning (ML) is a constantly evolving field of research striving to create faster and more accurate models. A driving force for smarter ML algorithms is automatic differentiation (AD), which is used to compute the gradient of a function [BPR15]. In the field of deep learning, AD is used to compute the weight each input to the network had on the output. These kinds of networks often have many inputs, and few outputs, in which case reverse mode AD is preferable.

Neural networks implement a set of layers, each of which transforms a vector of inputs and passes the resulting vector to the next layer. These transformations are often parallel in nature, and are as such preferably run on a GPU.
Second-order array combinators (SOACs) are powerful parallel constructs that transform arrays of data using functions given as input. Vectors are implemented as arrays, making SOACs desireable for ML algorithms.
Mainstream ools such as PyTorch[1] and Jax[2] have been developed to increase accessibility of ML models, all supporting various degrees of reverse mode AD. A shared shortcoming among these tools is the lack of GPU support for running reverse mode AD on many SOACs.

`Futhark` is a data-parallel programming language that is centered around the usage of SOACs for bulk parallel operations, compiled to run on GPU's.
To support expression of ML algorithms in `Futhark`, progress has been made by [Sch+22] to develop techniques for application of reverse mode AD on SOACs such as `map`, `scan`, `reduce`, `reduce_by_index` and `scatter` as compiler transformations. [Sch+22] implemented the rewrite rules for most of these SOACs, but for the case of `reduce_by_index` only the high-level reasoning was developed.

`Reduce_by_index` is a SOAC that groups elements using a key, and then reduces each of those groups using an associative and commutative operator such as $(+)$. Say we have a group of people and we want to know the sum of the age of the men and women in that group, we first group them according to their gender, and then sum the ages of each respective group.

`Reduce_by_index` is a generalisation of this problem, and has a variable amount of groups, or rather *bins*, to collect values to. Each element to be grouped has a corresponding key, implemented as an integer, that dictates which bin a given element belongs to. The elements of each group are then reduced with an associative and commutative operator, which could be $(+)$ to get the sum or $(\times)$ for the product. The resulting list of elements is referred to as a histogram.
The sequential implementation of `reduce_by_index` in `Futhark` is:

```
for i < n-1 do
    (value, bin) = elements[i]
    dest[bin] ⊙= value
```

Where `dest` is an array of the bins we collect values to, `elements` is an array of values with a corresponding key, and $\odot$ is the operator to perform reduction with.

Prior to this project I had very limited knowledge about ML in general, as well as AD. In isolation AD has nothing to do with ML, but its is one of the driving forces for its development. One of the main challenges was therefore to not only research and understand AD as a concept, but also to understand its motivation through its connection to ML.

[1]https://pytorch.org/docs/stable/
[2]https://jax.readthedocs.io/en/latest/index.html

Chapter 2 covers the most relevant background of AD, along with that of the source language `Futhark`.

The contributions of this thesis is a fully developed rewrite rule for the context of reverse mode AD for `reduce_by_index`, inspired by the high-level reasoning presented in [Sch+22]. It also contributes with an extension of the `Futhark` compiler, implementing the rewrite rule as a compiler transformation. Chapter 3 presents the application of a general rewrite rule to `reduce`, and generalises it to `reduce_by_index`.

The implementation consists of three special cases that capture the more popular operators $(+, *, \text{min/max})$, and a general case for an arbitrary associative and commutative operator.
The implementation also proved to be very challenging. The `Futhark` compiler uses its own intermediate representation (IR) and spans several thousand lines of code. For the purposes of this project only a subset of the IR was required to be understood.
The general case was written from scratch, but the three special cases were made from code supplied by my supervisor, which came from an outdated branch. Fixing this outdated code served as a warm-up to get an understanding of the intricacies of the compiler before implementing the general approach. The supervisor of this project endorsed discussion of the general approach with another student.
Chapter 4 covers the relevant constructs of the `Futhark` IR, and how the four different cases were implemented.

The implemented methods were validated by comparing the results of reverse mode with that of forward mode. Forward mode is a separate method that tackles the issue of computing derivatives differently, so the odds of the same bug appearing in both methods is minimal.
All four cases are implemented by rigorous use of SOACs as to maximise parallelism. All SOACs used have the same work-asymptotic of $O(n)$, which `reduce_by_index` shares if the histogram is large. This makes the overhead of applying reverse mode AD on the `reduce_by_index`-construct a constant multiple.
The multiples measured for each of the four cases were:

- Addition: 2.5 times slower.
- Min/max: 13.6 times slower.
- Multiplication: 14.1 times slower.
- General approach: > 500 times slower.

The methods used for validation and benchmarks are discussed in chapter 5.

Finally chapter 6 discusses future work and holds a conclusion to the project.

Most of the implemented code has been added to the appendix, but the complete code can be found on my GitHub[3]. The relevant files are in "`futhark/src/Futhark/AD/Rev`" and "`futhark/tests/ad`".

[3]https://github.com/Cherosev/futhark/tree/Søren-AD

2 Preliminaries

In this chapter i will highlight some of the most important pieces of background which lay foundation to this thesis.
It will cover the parallel constructs of the `Futhark` language that are used and the key points of AD that will be used to create rewrite rules for `reduce_by_index` in section 3.

2.1 The Futhark language

`Futhark` has been a continuous project at DIKU[4] with many contributing parties such as [Hen17] and [Sch+22]. A more complete list can be found at `Futharks` own publications list[5].

While `Futhark` has sequential constructs such as loops, the general goal of programming in `Futhark` is to program in terms of SOACs.
SOACs are powerful constructs that allow for bulk parallel operations that transform large collections of data. Each SOAC has its own semantics, some of which we will soon cover, but they are all parameterized by a function which is used to perform this transformation. The possibility of passing user-defined functions to SOACs gives the programmer the freedom to apply any transformation desired.

2.1.1 Parallel constructs of the source language

This section presents the parallel constructs of `Futhark` used for the rewrite rule of `reduce_by_index`, along with their semantics.

First off we have the basic SOACs `map`, `reduce` and `scan` with the following type signatures[Hen17]:

$$\begin{aligned}
&\text{map:} && (\text{f: } \alpha \to \beta) \to (\text{as: } [n]\alpha) \to [n]\beta \\
&\text{map2:} && (\text{f: } \alpha \to \alpha \to \beta) \to (\text{as}_1 : [n]\alpha) \to (\text{as}_2 : [n]\alpha) \to [n]\beta \\
&\text{reduce:} && (\odot : \alpha \to \alpha \to \alpha) \to (e_\odot : \alpha) \to (\text{as: } [n]\alpha) \to \alpha \\
&\text{scan:} && (\odot : \alpha \to \alpha \to \alpha) \to (e_\odot : \alpha) \to (\text{as: } [n]\alpha) \to [n]\alpha \\
&\text{replicate:} && (\text{n: } i64) \to (\text{x: } \alpha) \to [n]\alpha \\
&\text{scatter:} && (\text{dest: } [m]\alpha) \to (\text{is: } [n]\text{i64}) \to (\text{as: } [n]\alpha) \to [m]\alpha
\end{aligned}$$

Where α and β are types, and $[n]\alpha$ denotes an array with n elements of type α. Type names in `Futhark` follow a syntax of a character defining the type followed by an integer describing the number of bits. In the case of the argument `is` of `scatter` i64 is a signed integer of 64 bits. Similarly, u64 is a 64-bit unsigned integer and f64 being a 64-bit floating-point number.

First we have `map` which is a function that simply takes some function f as an argument and applies f to each element of its array argument `as`:

$$map\ f\ [a_0, a_1, ..., a_{n-1}] = [f\ a_0, f\ a_1, ..., f\ a_{n-1}]$$

[4]Datalogisk Institut Københavns Universitet
[5]https://futhark-lang.org/publications.html

Map2 is much like map, but the function given now takes two parameters instead. Each value in the two lists are paired, and the function is applied:

$$map2\ f\ \left[a_{1_0}, a_{1_1}, ..., a_{1_{n-1}}\right] \left[a_{2_0}, a_{2_1}, ..., a_{2_{n-1}}\right] = \left[f\ a_{1_0} a_{2_0}, f\ a_{1_1} a_{2_1}, ..., f\ a_{1_{n-1}} a_{2_{n-1}}\right]$$

Reduce takes an associative operator $\odot$, the neutral element of that operator $e_\odot$, and an array of values as to reduce. All elements of its array argument as are then accumulated to one value:

$$\text{reduce}\ \odot\ e_\odot\ [a_0, a_1, ..., a_{n-1}] = e_\odot \odot a_0 \odot a_2 \odot ... \odot a_{n-1}$$

Scan is somewhat like reduce but instead returns an array of the same length as the input. Each element of the resulting array is the sum of all elements up until that point.
scan can either be inclusive if the current element is included in the sum, or exclusive if not:

$$\begin{aligned} \text{scan}_{\text{inc}}\ \odot\ e_\odot\ [a_0, a_1, ..., a_{n-1}] &= [e_\odot \odot a_0, e_\odot \odot a_0 \odot a_1, ..., e_\odot \odot a_0 \odot a_1 \odot ... \odot a_{n-1}] \\ \text{scan}_{\text{exc}}\ \odot\ e_\odot\ [a_0, a_1, ..., a_{n-1}] &= [e_\odot, e_\odot \odot a_0, ..., e_\odot \odot a_0 \odot a_1 \odot ... \odot a_{n-2}] \end{aligned}$$

A scan can also be a segmented scan. A segmented scan also takes a flag array as input, with non-zero elements indicating the start of a new segment, and zero indicating a continuation of a segment. A normal scan is then performed, but at the start of a new segment the accumulation is reset. The following is an example using (+) as the operator of an inclusive segmented scan:

```
Values = [1,1,1,1,1,1,1,1,1]
Flags  = [1,0,0,1,0,0,0,0,1]
Result = [1,2,3,1,2,3,4,5,1]
```

Replicate is used to initialize arrays. It returns an array of length n where each element is x:

```
replicate 4 3 = [3,3,3,3]
```

Scatter returns the destination array dest where for each index in is the value of dest at that index, is overwritten to be the corresponding value in as. Indices in is that are out of bounds of dest are ignored. Duplicate indices in is are allowed, but the result is undefined.
We present the semantics of scatter based on simple imperative code, using k as the size of dest:

```
for i in 0 ... n-1:
    index = is[i]
    value = as[i]
    if (0 <= index < k)
      dest[bin] = value
```

As the name implies, reduce_by_index is a lifted form of the reduce-operator that instead of reducing a list of values to a single value, reduces them to a number of bins. The resulting set is what we call a histogram. The specific details of this function are described in [Hen+20].

Type signature of reduce_by_index:

$$\text{reduce\_by\_index: (dest: } [k]\alpha) \rightarrow (\odot :\ \alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow (e_\odot :\ \alpha) \rightarrow (\text{inds: } [n]\text{i64}) \rightarrow (\text{as: } [n]\alpha) \rightarrow [k]\alpha$$

Reduce_by_index returns dest where for each index in inds the value in dest is updated to the application of $\odot$ to dest and the corresponding value in as. Indices in inds that are out of bounds of dest are ignored. Using k as the size of dest, its simple imperative code is:

```
for i in 0 ... n-1:
    bin = inds[i]
    value = as[i]
    if (0 <= index < k)
      dest[bin] ⊙= value
```

Since **inds** can contain duplicates, $\odot$ can be applied multiple times to the same value in `dest`. $\odot$ must therefore be both associative and commutative.

The final construct i want to mention is `loop` and is the only sequential construct in this list. `Futharks loop` works by looping over a set of parameters, with the loop body returning the same type as its parameter. If we were to implement the pseudo-code above for `reduce_by_index`:

```
let histo =
    loop dest' = copy dest for i < n do
        let bin = inds[i]
        let value = as[i]
        let result = dest'[bin] ⊙ value
        in dest'[bin] = result
```

At the loop entrance we set `dest'` to be the loop parameter, initially set to be a copy of `dest`. The body of the loop is then computed n times with i set to $0, ...n-1$. For each iteration of the loop, the result of the body is bound to be the loop parameter of the next iteration. After the final iteration the result is then bound to `histo`, which is our result.

2.2 Automatic differentiation

One of the goals of this thesis was to try and understand automatic differentiation (AD), so my first task was to research the topic. This segment introduces the key parts of AD used in this project.

AD is the concept of differentiation functions implemented as code in a systematic order. Composite functions are decomposed to intermediate variables, using the `chain rule` to compute the derivative of the composite function.
The `chain rule` states:

$$\text{if } f(x) = (g \circ h)(x) = g(h(x)) \text{ then } \frac{\partial f(x)}{\partial x} = \frac{\partial g(z)|_{z=h(x)}}{\partial z} \cdot \frac{\partial h(x)}{\partial x} \tag{1}$$

The `chain rule` says that in order to compute the derivative of function f composed by g and h, we first compute the derivative of the inner function. The result of the inner function is then used to compute the the derivative of the outer function, and the two derivatives are then multiplied. For the purposes of AD we introduce intermediate variables to store these inner results. Naming of intermediate variables for a function $f : \mathbb{R}^n \to \mathbb{R}^m$ follows the naming convention that is also used in [BPR15]:

- Variables $w_{i-n} = x_i, i = 1, ..., n$ are input variables.
- Variables $w_i, i = 1, ..., l$ are intermediate values.
- Variables $y_{m-i} = w_{l-1}, i = m-1, ..., m$ are output variables.

As a simple example we can decompose the function $y = f(g(h(x)))$. Since this function has the form $f : \mathbb{R} \to \mathbb{R}$ we simplify it slightly and focus on the naming of intermediate values:

$$\begin{aligned}
w_0 &= x \\
w_1 &= h(w_0) \\
w_2 &= g(w_1) \\
w_3 &= f(w_2) \\
y &= w_3
\end{aligned}$$

Using these intermediate values with the `chain rule`, we compute $\frac{\partial y}{\partial x}$ as:

$$\frac{\partial y}{\partial x} = \frac{\partial w_3}{\partial x} = \frac{\partial w_3}{\partial w_2} \cdot \left(\frac{\partial w_2}{\partial w_1} \cdot \left(\frac{\partial w_1}{\partial w_0} \cdot \left(\frac{\partial w_0}{\partial x} \right) \right) \right)$$

Now suppose we have a function $y = f(x_1, ..., x_n)$ of form $f : \mathbb{R}^n \to \mathbb{R}$ and want the tangent of y for some $x \in \mathbb{R}^n$, we will need to compute:

$$\nabla f(x) = \frac{\partial f(x)}{\partial x_1} \cdot (x_1) + \frac{\partial f(x)}{\partial x_2} \cdot (x_2) + ... + \frac{\partial f(x)}{\partial x_n} \cdot (x_n)$$

In order to compute this, we will need the Jacobian vector of $(\frac{\partial f(x)}{\partial x_1}, \ldots, \frac{\partial f(x)}{\partial x_n})$

If the function f has several outputs this problem escalates to be a Jacobian matrix. Suppose we have $f : \mathbb{R}^n \to \mathbb{R}^m$ with inputs $(x_1, ..., x_n)$ and outputs $(y_1, ..., y_m)$ we will need:

$$\begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

AD has two modes that handles the issue of computing the Jacobian matrix in different ways, namely `forward` and `reverse` mode. The following two segments will further explain how each mode works.

2.2.1 Forward mode

Forward mode computes the Jacobian matrix column-wise by forward-propagating derivatives. For each intermediate value we associate a derivative [BPR15]:

$$\dot{w}_i = \frac{\partial w_i}{\partial x_i}$$

For each statement on the left-hand side of Figure 1 we apply the chain rule to compute the derivative.

Forward mode computes the derivatives using two traces, both shown in Figure 1. The first trace computes the values of intermediate values (Primal Trace), and the second trace computes the derivative of each intermediate value step-by-step using whatever rule of differentiation that applies to that value (Tangent trace). Some of these rules needs the actual value, hence why we perform the Forward Primal Trace.

For the example of $f(x_1, x_2) = (x_1 + x_2) \cdot (ln\, x_1)$ we initially assign the input variable we want to differentiate with respect to a derivative of 1. All other inputs are assigned a derivative of 0. For $\frac{\partial f(x_1,x_2)}{\partial x_1}$ we set $\dot{x}_1 = 1$ and $\dot{x}_0 = 0$. A running example can be seen in Figure 1.

Forward Primal Trace

$$
\begin{array}{lll}
w_{-1} & = x_1 & = 4 \\
w_0 & = x_2 & = 3 \\
\hline
w_1 & = w_{-1} + w_0 & = 4 + 3 \\
w_2 & = \ln w_{-1} & = \ln 4 \\
w_3 & = w_1 \cdot w_2 & = 7 \cdot 1.38 \\
\hline
y & = w_3 & = 9.704
\end{array}
$$

Forward Tangent (Derivative) Trace

$$
\begin{array}{lll}
\dot{w}_{-1} & = \dot{x}_1 & = 1 \\
\dot{w}_0 & = \dot{x}_2 & = 0 \\
\hline
\dot{w}_1 & = \dot{w}_{-1} + \dot{w}_0 & = 1 + 0 \\
\dot{w}_2 & = \dot{w}_{-1} / w_{-1} & = 1/4 \\
\dot{w}_3 & = \dot{w}_1 \cdot w_2 + \dot{w}_2 \cdot w_1 & = 1 \cdot 1.38 + 0.25 \cdot 7 \\
\hline
\dot{y} & = w_3 & = 3.13
\end{array}
$$

Figure 1: Forward mode for $f(x_1, x_2) = (x_1 + x_2) \cdot (ln\, x_1)$.

Since the tangent trace computes derivatives by forward propagating those of intermediate values, the primal and tangent trace can be performed in step with one another. This can be implemented by operator overloading

Forward mode is efficient for functions $f : \mathbb{R} \rightarrow \mathbb{R}^m$ since we only need a single pass of the Forward Tangent to compute the derivative of each output variable. For functions $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ a single forward pass will yield a single column of the Jacobian matrix, requiring n executions of the forward pass instead.

2.2.2 Reverse mode

In reverse mode we compute the Jacobian matrix by back-propagating derivatives, starting with our output. Each intermediate values is associated with an adjoint that measures the changes in y_j with respect to w_i:

$$\overline{w}_i = \frac{\partial y_j}{\partial w_i}$$

Much like forward mode, reverse mode consists of two traces of the code. Exactly like forward mode we first perform a Forward Primal Trace to bring all variables into scope.

In the example shown in Figure 2 the forward trace shows that w_{-1} affects y through both its contributions to w_1 and w_2. The total contribution therefore becomes:

$$\frac{\partial y}{\partial w_{-1}} = \frac{y}{\partial w_1} \frac{\partial w_1}{w_{-1}} + \frac{\partial y}{\partial w_2} \frac{w_2}{\partial w_{-1}} = \overline{w}_1 \frac{\partial w_1}{w_{-1}} + \overline{w}_2 \frac{w_2}{\partial w_{-1}} \tag{2}$$

The different rules of differentiating each intermediate value depends on the function used, but the rewrite rule used for the reverse sweep can be generalized.
For the assignment $y = a \odot b$, the reverse sweep will compute:

$$\begin{array}{ll} \overline{a} & \mathrel{+}= \frac{\partial a \odot b}{\partial a} \cdot \overline{y} \\ \overline{b} & \mathrel{+}= \frac{\partial a \odot b}{\partial b} \cdot \overline{y} \end{array} \tag{3}$$

This is the main rewrite rule of reverse mode AD, and can be used to reason how more complex operations such as SOACs can be differentiated.
We accumulate to $\overline{a}$ and $\overline{b}$ to facilitate the possibility that a or b are used multiple times as in equation 2.

The reverse trace can be generated by applying this generalized rule to the assignment of each intermediate variable.

The reverse trace is initialized by assigning the output variable we want to differentiate with respect to an adjoint of 1, and all others to 0. The computations are then performed bottom-up. The reverse trace in Figure 2 is structured to fit with the forward trace, but should be read from bottom to top.

Forward Primal Trace

$$
\begin{array}{lll}
w_{-1} & = x_1 & = 4 \\
w_0 & = x_2 & = 3 \\
\hline
w_1 & = w_{-1} + w_0 & = 4 + 3 \\
 & & \\
w_2 & = \ln w_{-1} & = \ln 4 \\
w_3 & = w_1 \cdot w_2 & = 7 \cdot 1.38 \\
 & & \\
\hline
y & = w_3 & = 9.704
\end{array}
$$

Reverse Adjoint (Derivative) Trace

$$
\begin{array}{llll}
\overline{x}_1 & = \overline{w}_{-1} & & = 3.13 \\
\overline{x}_2 & = \overline{w}_0 & & = 1.38 \\
\hline
\overline{w}_0 & \mathrel{+}= \overline{w}_1 \frac{\partial w_1}{\partial w_0} & = \overline{w}_1 \cdot 1 & = 1.38 \\
\overline{w}_{-1} & \mathrel{+}= \overline{w}_1 \frac{\partial w_1}{\partial w_{-1}} & = \overline{w}_{-1} + \overline{w}_1 \cdot 1 & = 3.13 \\
\overline{w}_{-1} & \mathrel{+}= \overline{w}_2 \frac{\partial w_2}{w_{-1}} & = \overline{w}_2 / w_{-1} & = 1.75 \\
\overline{w}_1 & \mathrel{+}= \overline{w}_3 \frac{\partial w_3}{w_1} & = \overline{w}_3 \cdot w_2 & = 1.38 \\
\overline{w}_2 & \mathrel{+}= \overline{w}_3 \frac{\partial w_3}{\partial w_2} & = \overline{w}_3 \cdot w_1 & = 7 \\
\hline
\overline{w}_3 & = \overline{y} & = 1 &
\end{array}
$$

Figure 2: Reverse mode for $f(x_1, x_2) = (x_1 + x_2) \cdot (ln\, x_1)$.

Unlike the forward mode, the primal- and reverse trace cannot be performed in step with each other. The first step of the reverse trace is to compute the adjoints of intermediate values that contributed to the result, and they will often be the last statements of the primal trace. This is not necessarily the case, but there is no assurance. As a result the primal trace must be performed in its entirety, before performing the reverse trace.

A single pass of reverse mode computes the adjoint of all input variables w.r.t. one output variable, effectively computing the Jacobian matrix row-wise. In my running examples of Figure 1 and 2, forward mode needed to be run twice, while reverse mode only needed to be run once. This should demonstrate how reverse mode is more efficient for a function $f : \mathbb{R}^n \to \mathbb{R}^m$ when $n >> m$.
This is often the case for machine learning algorithms, taking a large amount of input variables, and returning a single value. This heavily favors the reverse mode as the go-to method of computing derivatives for these types of machine learning algorithms.

3 Rationale of the reverse-mode rewrite rules

This chapter presents the rewrite rules derived by applying a generalized rewrite rule to the constructs `reduce` and `reduce_by_index`.

The core rule for rewriting statements in reverse mode is:

$$\text{let } x = f(a,b) \Longrightarrow \begin{array}{l} \text{let } x = f(a,b) \\ \vdots \\ \text{let } \overline{a} \mathrel{+}= \frac{\partial f(a,b)}{\partial a} \cdot \overline{x} \\ \text{let } \overline{b} \mathrel{+}= \frac{\partial f(a,b)}{\partial b} \cdot \overline{x} \end{array} \tag{4}$$

This rule was touched upon in the previous section in equation 3 but is entirely given by [Sch+22] which describes how many of the other constructs in `Futhark` have been implemented for reverse mode.

When a statement $Stmt_i$ is rewritten, the original statement is first added as part of the forward sweep. The reverse sweep is only added after all following statements have been rewritten, indicated in the rule by vertical dots.

$$\begin{array}{l} \text{let } x_1 = f(a,b) \\ \text{let } x_2 = f(c,d) \end{array} \Longrightarrow \begin{array}{l} \text{let } x = f(a,b) \\ \vdots \\ \text{let } \overline{a} \mathrel{+}= \frac{\partial f(a,b)}{\partial a} \cdot \overline{x} \\ \text{let } \overline{b} \mathrel{+}= \frac{\partial f(a,b)}{\partial b} \cdot \overline{x} \end{array} \Longrightarrow \begin{array}{l} \text{let } x_1 = f(a,b) \\ \text{let } x_2 = f(c,d) \end{array} \Longrightarrow \begin{array}{l} \text{let } x = f(a,b) \\ \text{let } y = f(c,d) \\ \text{let } \overline{c} \mathrel{+}= \frac{\partial f(c,d)}{\partial c} \cdot \overline{y} \\ \text{let } \overline{d} \mathrel{+}= \frac{\partial f(c,d)}{\partial d} \cdot \overline{y} \\ \text{let } \overline{a} \mathrel{+}= \frac{\partial f(a,b)}{\partial a} \cdot \overline{x} \\ \text{let } \overline{b} \mathrel{+}= \frac{\partial f(a,b)}{\partial b} \cdot \overline{x} \end{array} \tag{5}$$

Reverse sweep is organized in reverse. Forward sweeps are chronological but reverse is not. For x_2 its reverse sweep is exectued before that of x_1.

Since the rewritten code is to be computed from the top down, the first statement in a body to be rewritten must add its reverse sweep last, and vice versa for the last statement in the body.

3.1 How to differentiate Reduce

Differentiation of `reduce_by_index` is generalization of `reduce`, hence why this segment to my thesis.

We recall the semantics of `reduce` for an arbitrary associative operator $\odot$ with corresponding neutral element $e_\odot$:

$$\text{reduce } \odot \ e_\odot \, [a_0, a_1, ..., a_{n-1}] = a_0 \odot a_1 \odot ... \odot a_{n-1}$$

In order to compute the adjoint of each input variable a_i, we bind the result to y, which will make the reasoning more simple. Using the associativity of the operator we group the elements into three groups, isolating a_i:

$$\text{let } y = \ (a_0 \odot ... \odot a_{i-1}) \odot a_i \odot (a_{i+1} \odot ... \odot a_{n-1})$$

We further simplify by reducing the elements preceding and following a_i. All elements up until a_i are named l_i, and all following elements are named r_i:

$$\text{let } l_i = a_0 \odot ... \odot a_{i-1}$$
$$\text{let } r_i = a_{i+1} \odot ... \odot a_{n-1}$$

Using these two reductions, we simplify the computation of y to be:

$$\text{let } y = \; l_i \odot a_i \odot r_i$$

Using this simplified computation we apply the general rewrite rule from equation 4 to compute the adjoint of a_i:

$$\overline{a}_i \mathrel{+}= \frac{\partial(l_i \odot a_i \odot r_i)}{\partial a_i} \cdot \overline{y} \tag{6}$$

Now, all we need is to apply this to each a_i.
Since we need to compute l_i and r_i for each a_i, the most simple approach is to perform a `scan`. l_i can easily be computed with a segmented exclusive scan. For r_i we first need to reverse the list of values, perform the segmented exclusive scan, and then reverse once more. We do an exclusive scan as to exclude a_i from the partial reduction in l_i and r_i.
Now all we need is to implement equation 6 as a function that can be `mapped` across values of l_i, a_i and r_i:

$$f : \left(\lambda \, l_i, a_i, r_i \rightarrow \frac{\partial(l_i \odot a_i \odot r_i)}{\partial a_i} \cdot \overline{y}\right)$$

Since this is reverse mode, we are ensured that $\overline{y}$ is computed beforehand, effectively making it a constant to us.

Now lets put it all together.
For the forward sweep we just need to perform the normal reduction to bring y into scope. For the reverse sweep we need to compute l_i and r_i, and then `map` f onto each l_i, a_i and r_i. The rewrite-rule for `reduce` therefore becomes:

$$\text{let } y = \text{reduce } \odot \; e_\odot \text{ as} \Longrightarrow \begin{array}{l} \text{let } y = \text{reduce } \odot \; e_\odot \text{ as} \\ \vdots \\ \text{let lis} = \text{scan}_{\text{exc}} \; \odot \; e_\odot \; as \\ \text{let ris} = \text{reverse } as \\ \qquad\qquad \triangleright \; \text{scan}_{\text{exc}}(\lambda \, x \odot y \rightarrow y \odot x) \; e_\odot \; \triangleright \; \text{reverse} \\ \text{let } \overline{as} \mathrel{+}= \text{map3 } f \text{ lis as ris} \end{array} \tag{7}$$

Where += is scalar addition.

3.2 How to differentiate Reduce_by_index

We recall the semantics and pseudocode of `reduce_by_index`:

reduce_by_index: (dest: $[k]\alpha$) $\rightarrow$ ($\odot$: $\alpha \rightarrow \alpha \rightarrow \alpha$) $\rightarrow$ ($e_\odot$: α) $\rightarrow$ (inds: $[n]$i64) $\rightarrow$ (as: $[n]\alpha$) $\rightarrow$ $[k]\alpha$

```
for i in 0 ... n-1:
    bin = inds[i]
    value = as[i]
    dest[bin] ⊙= value
```

The basic reasoning for the rewrite rule is much like `reduce`, but we need to accommodate the bins. l_i and r_i must be computed with an irregular segmented scan, and to do that we will need elements of the same bin to lie consecutively. The requires a sort to be performed.

The work-depth asymptotic of `reduce_by_index` is $\mathcal{O}(n)$, eg. linear in the input. To preserve the asymptotics of the program we sort using `Radix` sort.
The work-depth asymptotic of `Radix` sort is $\mathcal{O}(k \cdot n)$ where k is the size of the key and n is the amount of values to sort.
We implement `Radix` sort to use the bits as the key, and because the indices given to `reduce_by_index` are 64-bit integers, the key size can be considered a constant for us, giving us the asymptotic of $\mathcal{O}(n)$.

We only want to update the values in `as` contribution to the result, but because the parameter `dest` might contain values, we also need to consider its contribution to the result. If `dest` does contain values, we also need to bring its adjoint into scope, since `dest` is consumed by the operation. To solve these issues we first reduce elements into a fresh destination array with neutral elements. We denote k as the length of the histogram and `hist` as the result:

```
let temp_dest = replicate k e⊙
let hist_temp = for i in 0 ... n-1:
                  let bin = inds[i]
                  let value = as[i]
                  temp_dest[bin] ⊙= value
let hist = map2 ⊙ dest hist_temp
```

We now need to compute the adjoint of both the histogram created from only the input values (`hist_temp`), and the values of the original histogram (`dest`). We once again use the core rule of equation 4 to compute the contribution each partial histogram has on the result.
We map the following function across the bins of both partial histograms, with `orig` denoting the value of a bin in `dest`, and `temp_val` denoting the value of a bin in `hist_temp`:

$$f_1 \ : \ \left(\lambda \text{ orig},\ \text{temp\_val},\ \text{bin} \rightarrow \frac{\partial\,(\text{orig} \odot \text{temp\_val})}{\partial \text{ orig}} \cdot \overline{hist}[\text{bin}]\right)$$

$$f_2 \ : \ \left(\lambda \text{ orig},\ \text{temp\_val},\ \text{bin} \rightarrow \frac{\partial\,(\text{orig} \odot \text{temp\_val})}{\partial \text{ temp\_val}} \cdot \overline{hist}[\text{bin}]\right)$$

In two separate statements we map f_1 and f_2 onto $dest$, $hist_{temp}$ and (`iota k`) (where k is the length of the histogram) to obtain $\overline{dest}$ and $\overline{hist}_{temp}$:

```
let dest‾ = map f1 dest hist_temp (iota k)
let hist‾_temp = map f2 dest hist_temp (iota k)
```

The values in `as` only contributed to the partial result of `hist_temp`, and their adjoints must therefore be computed from $\overline{\text{hist\_temp}}$.
We now define a function to compute $\overline{as}$ with the adjoint of its corresponding bin:

$$f_3 \ : \ \left(\lambda\, l_i, a_i, r_i, bin \rightarrow \frac{\partial(l_i \odot a_i \odot r_i)}{\partial a_i} \cdot \overline{hist\_temp}[bin]\right)$$

Computing the adjoints requires us to compute $\texttt{hist}_{\text{temp}}$ which is the partial result of `hist`. We could have a forward sweep where we perform the normal `reduce_by_index` and compute $\texttt{hist}_{\text{temp}}$ in the reverse sweep, but then we would be reducing `as` twice. We therefore move the computation of $\texttt{hist}_{\text{temp}}$ to the forward sweep instead. Similarly, $\texttt{hist}_{\text{temp}}$ can be computed from `lis` and `as` by $l_i \odot a_i$ for each i that is the last value of a segment. Assuming we have a function `get_segment_sum`, we get the rewrite rule:

$$\text{let hist} = \text{reduce\_by\_index dest } \odot e_{\odot} \text{as} \Longrightarrow \begin{array}{l} \text{let } as_{\text{sorted}} = \text{radix\_sort } \ as \\ \text{let lis} = \text{segScan}_{\text{exc}} \ \odot \ e_{\odot} \ as_{\text{sorted}} \\ \text{let ris} = \text{reverse } as_{\text{sorted}} \\ \qquad\qquad \rhd \ \text{segScan}_{\text{exc}}(\lambda\, x \odot y \rightarrow y \odot x) \ e_{\odot} \ \rhd \ \text{reverse} \\ \text{let hist}_{\text{temp}} = \text{get\_segment\_sum lis } \ as_{\text{sorted}} \\ \text{let hist} = \text{map2 } \odot \text{ dest hist}_{\text{temp}} \\ \vdots \\ \text{let } \overline{\text{dest}} = \text{map } f_1 \text{ dest hist}_{\text{temp}} \\ \text{let } \overline{\text{hist}}_{\text{temp}} = \text{map } f_2 \text{ dest hist}_{\text{temp}} \\ \text{let } \overline{as} \mathrel{+}= \text{map4 } f_3 \text{ lis as ris inds} \end{array} \tag{8}$$

How `get_segment_sum` and other functions are implemented, will be elaborated on in the following chapter.

4 Implementation

This chapter will present the intermediate language (IR) of the compiler, and the implementation of the rewrite rule.

The objective of this thesis was to implement reverse-mode AD as a compiler-transformation, so my implementation is an extension of the existing compiler.

The compiler IR is driven by expressions where each type of statement has its own rewrite-rule, eg. `scan`, `reduce`, `map` have individual cases. The SOAC-type has a central function `vjpSOAC` that generates the code necessary for reverse-AD of (ideally) any instance of the SOAC-type. Its simplified function signature is:

$$\text{vjpSOAC} :: \text{SOAC} \rightarrow \text{ADM}() \rightarrow \text{ADM}()$$

The `vjpSOAC` function matches on the type of the input SOAC, and separate functions generate the code for reverse-AD of `map` and `reduce` respectively.

This projects contribution to the compiler is a new set of functions that produce the reverse-AD code for SOACs of type `Hist`, which is the intermediate representation of `reduce_by_index` inside the compiler, and the central function `vjpSOAC` is extended with a new case that enters this new set of functions. The entry-point of the added set of functions is `diffHist`:

$$\text{diffHist} :: \text{Hist} \rightarrow \text{ADM}() \rightarrow \text{ADM}()$$

Before presenting the implementation, we present the most important parts of the intermediate language (IR) used in the `Futhark` compiler. The implementation does have some limitations, the most important of which will be covered at the end of this chapter.

4.1 Intermediate language of Futhark

One of the biggest challenges of this project was to understand and work with this IR. The compiler code contains a lot of different data types and helper functions, spanning several thousand lines of code.
In the end, only a subset of these types and functions were needed to be understood and used. Using a hand-full of features also helped make the code more readable, by having a repeated pattern for generation of statements.

The IR presented in this paper has for simplification been slightly rewritten, and should therefore not be interpreted as complete. Some constructors or arguments not used have been omitted, and some have been simplified with new names.

4.1.1 ADM-monad

First we have the `ADM`-monad, which is a wrapper for reverse mode code generation functions. The monad works as an environment that gives us three key features:

1. Holds information about existing variables and their types.
2. Captures generated statements. These statements are later used to produce the code itself.

3. Allows us to generate the code of following statements between the forward of reverse sweep. This makes it very easy to implement the behaviour of the rewrite rule presented in equation 5 from section 3.1.

On a high level, the function `diffHist` this paper contributes with takes this monad as input (along with the `reduce_by_index` statements), and returns a new monad with the added statements from the rewrite rule (equation 8).

The monad also holds information about adjoints of variables, and exposes some helper functions to interact with them. `lookupAdjVal` returns the adjoint of a variable when given its name, and `updateAdj` accumulates to existing adjoints.

4.1.2 Data types and helper functions

This segment presents the most commonly used constructs and helper functions in the implementation.
To start out softly we introduce the most basic data types used:

data **VName** : String	data **PrimValue** : **IntValue** IntValue **FloatValue** FloatValue **BoolValue** Bool	data **SubExp** : **Constant** PrimValue **Var** VName

`VName` is used for variables names and is a unique string, such that no two variables have the same name. Uniqueness of `VName`s is ensured by the `ADM`-monad. `PrimValue` is simply used for constant values.
`VNames` and `PrimValue` have a union type in `SubExp` which is either a constant value, or a variable name.

Parameters also have their own type `Param` which is a `VName` combined with a `Type`. `Param`s are constructed using the helper function `newParam`:

```
newParam :: String → Type → Param
i_param ← newParam "i" $ Int64
```

The arrow (←) indicates that we performing a monadic operation using the `ADM`-monad, which in this instance means we get a `Param` with a unique `VName`. `Param` is primarily used by lambda functions to specify their parameters.

We now move on to expressions:

data **Exp** : **BasicOp** BacicOp **If** SubExp Body Body **DoLoop** [SubExp] LoopForm Body **Op** SOAC	data **SOAC** : **Scatter** SubExp [VName] Lambda [(Shape, Int, VName)] **Hist** SubExp [VName] [HistOp] Lambda **Screma** SubExp [VName] ScremaForm

`Exp` is used to generate all of our needed statements using the helper function `letExp`:

```
letExp :: String → Exp → VName
```

In the ADM monad we can use `letExp` to bind the result of an `Exp` to a `VName`, which will be captured by the monad and produced as code.

Similarly, `runBodyBuilder` is a helper function that lets us build a `Body`. A `Body` is a list of statements resulting in a list of results of type `[SubExp]`:

$$\text{Body} = \begin{cases} \text{Stm}_1 : \text{let VName}_1 \ : \ \text{Type}_1 = \text{Exp}_1 \\ \text{Stm}_2 : \text{let VName}_2 \ : \ \text{Type}_2 = \text{Exp}_2 \\ \vdots \\ \text{Stm}_n : \text{let VName}_n \ : \ \text{Type}_n = \text{Exp}_n \\ [\text{Var VName}_n, \ \text{Constant } 5] \end{cases}$$

This helper function is very complex and has a lot of monadic features that we will not go into. For our purposes we are going to be using this in conjunction with `localScope` which lets us define parameters available inside the `Body`. This is used to build the bodies of the lambda-functions we are going to need throughout the implementation.

We will now take a look at the different constructors of `Exp`:

BasicOp BasicOp:
`BasicOp` has over 20 constructors, but to keep it simple will not list them. As the name implies this is all the basic operators such as binary operations, logical operations, array indexing, iota, replicate and `SubExp`. If we want to write let $x = 5 \cdot 3$, the IR representation would be:

```
var_x <- letExp "x" $ BasicOp $ BinOp Mul 5 3
```

This line would generate the let $x = 5 * 3$ statement in the produced code with a unique variable name with prefix "x". The unique name generated is then bound to `var_x` so we can reference it later inside the compiler.

If `SubExp Body Body`:
This is pretty straightforward and follows the semantics of most other programming languages. The `SubExp` should evaluate to a Bool and the corresponding `Body` is then executed.
Example:

```
true_body <- runBodyBuilder $ do
               x <- letExp "x" $ BasicOp $ SubExp $ Constant 5
               eBody[Var x]

false_body <- runBodyBuilder $ do
                y <- letExp "y" $ BasicOp $ SubExp $ Constant 10
                eBody[Var x]

check <- letExp "check" $ BasicOp $ CmpOp CmpEq (Constant 1) (Constant 0)

result <- letExp "result" $ If
                            (Var check)
                            true_body
                            false_body
```

Here we define two bodies, `true_body` and `false_body`, returning 5 or 10 respectively. A trivial check of equality for 1 and 0 is made, and the result of that comparison is bound to a

VName which check refers to. result is then bound to 10 by evaluating the value of check to false, and subsequently evaluating false_body to be 10.

DoLoop [SubExp] LoopForm Body:
For loops we recall that Futhark loops over an initial variable that is updated at the end of each iteration along with a counter i:

```
loop acc = 0 for i < n
     acc += i
```

This example computes the sum of all numbers $0...(n-1)$.
The first argument [SubExp] are the variables to loop over, in the example above it would be the initial value of acc. LoopForm holds some information about the bounds of i, and the Body is the body of the loop to be executed each iteration.

We now move on to the **SOAC**s:

Scatter SubExp [VName] Lambda [(Shape, Int, VName)]:

The first argument holds a SubExp that defines the length of the input arrays, and the second arguement is the VNames of the input arrays.
The Lambda is for fusion with a map, and is applied to the variables of the input arrays before scattering. The final argument is the destination-array defined by the shape (dimensions) of the destination-array, its rank, and its corresponding VName. The following is an example of translation from Futhark code to its corresponding representation in the IR:

```
let plusOne = map (+1) vs
in scatter dest is plusOne
```

(a) Futhark code

$\boxed{\textbf{Scatter}\ \text{n [is, vs]}\ (\lambda v \rightarrow v+1)\ \text{[(shape(dest), rank(dest), dest)]}}$

(b) IR representation

Figure 3: IR representation of scatter

In this example we are adding 1 to every element in vs and scattering the result into dest using the indices in is.

Hist SubExp [VName] [HistOp] Lambda:

Hist is the intermediate representation of reduce_by_index inside the compiler.
We first introduce the HistOp type, which specifies the operation to perform. A HistOp has the form:

HistOp Shape [VName] [SubExp] Lambda

The first argument Shape holds the dimensions of the destination array. The second argument [VName] is a list of names of destination arrays. The third argument of [SubExp] is the neutral element of the fourth argument, which is an operator implemented as a Lambda.

For the case of Hist it holds a SubExp defining the length of the input arrays, a list of VNames of the input arrays and a list of HistOps to perform.
The list of HistOps are for fusion of histograms. If we want to get both the sum (+) and product

$(*)$ of the input, we can define two different `HistOp`s with those operators and destination arrays.

The final argument of a `Lambda` is applied to the input arrays before reduction. As with the case of `Scatter`, the `Lambda` is for fusion with a map. Example of translation from `Futhark` code to its representation in the IR:

```
let plusOne = map (+1) vs
in reduce_by_index
   dest (*) (1) is plusOne
```

(a) Pseudo code

op: HistOp shape(dest) [dest] [1] $(\lambda x, y \rightarrow x \cdot y)$
Hist len(vs) [is, vs] [op] $(\lambda x \rightarrow x + 1)$

(b) IR representation of reduce_by_index

Screma `SubExp [VName] SchremaForm`:

`Screma` is a combination of `scan`, `reduce` and `map`. It has a `SubExp` defining the length of the input arrays, a list of `VNames` for the input arrays, and finally a `ScremaForm`.
`ScremaForm` is used to fuse `Scan`, `Reduce` and maps. The given `ScremaForm` is then applied to the input. Its type is:

`ScremaForm [Scan] [Reduce] Lambda`

In the simple case where we only want to map:

```
let plusOne = map (+1) vs
```

(a) Pseudo code

Screma len(vs) [vs] (**ScremaForm** [] [] $(\lambda x \rightarrow x + 1)$)

(b) IR representation of a single map

If, however, we want to reduce the result of a map:

```
let plusOne = map (+1) vs
in reduce (+) (0) plusOne
```

(a) Pseudo code

Screma len(vs) [vs]
(**ScremaForm** [] $[(\lambda x, y \rightarrow x + y))]$ $(\lambda x \rightarrow x + 1)$)

(b) IR representation of a single map

`Lambda`, `Scan` and `Reduce` are very much alike:

Lambda : `[Param] Body ReturnType`

Scan : Lambda [SubExp]   **Reduce** : Commutativity Lambda [SubExp]

`Lambda`-functions takes a list of parameters, a `Body` to be evaluated, and a return-type of the of the `Lambda`. The parameters given should be used inside the given `Body` and the return-type of the `Body` should match the return-type of the `Lambda`.

`Scan` and `Reduce` are lifted types of `Lambda`. `Scan` consists of an operator `Lambda` and the neural element of the `Lambda` given as a `SubExp`. `Reduce` is like `Scan`, but also holds some information about the commutativity of the `Lambda`.

For completeness we also introduce patterns of type `Pat`. When reverse mode AD is applied to a statement, that statement is broken into two pieces: the pattern and the expression:

```
let hist = reduce_by_index dest (+) (0) is vs
```

In the above example the left-hand side of "=" is the pattern, and the right-hand side is the expression that is to be bound to the pattern. If we look at the type of `diffHist`:

```
diffHist :: VjpOps -> Pat Type -> StmAux() -> SOAC SOACS -> ADM () -> ADM ()
```

Here the pattern is the second parameter of the function, and is the element that the resulting histogram should be bound to.

4.2 Code

With the IR presented we move on to the implementation itself.

Because `reduce_by_index` is a SOAC, its operator is given to it as an argument. The rewrite rule presented back in section 3.2 assumes no prior knowledge of the operator used, and will work for any associative and commutative operator given to it. If, however, we did know what operator was used, we could optimize the rule for that given operator.

The implementation is therefore split into four cases, using prior helper functions of the compiler to identify the operator of `reduce_by_index` statement to generate code for:

1. Addition
2. Multiplication
3. Min/max - Strictly speaking this is two different operators, but they can be handled in the same way.
4. General case - Works for any operator.

Using `Haskells` guards, all four cases are implemented as the same function named `diffHist`, and all follow the same style of unpacking the histogram given to it:

```
diffHist _vjops pat aux soac m
  | (Hist n [inds, vs] hist_fun bucket_fun) <- soac,
    [HistOp shape rf [orig_dst] [ne] f] <- hist_fun,
    .
    .
    .
```

Where `inds` are the bins, `vs` are the values, n is the length of `inds` and `vs`, `bucket_fun` is a transformation to apply to `inds` and `vs` before performing `reduce_by_index`. `orig_dst` is the destination-array, `shape` is the length of the destination array, `f` is the operator and finally `ne` is the neutral element of `f`.

As explained in section 4.1 the histogram may potentially be fused with a map, which in this case is indicated by the variable `bucket_fun`. If no fusion is made, the `bucket_fun` will be the identity-function $\lambda x \rightarrow x$, otherwise it is some unknown lambda.
If the histogram is fused with a map, the problem of differentiating the histogram becomes more complex, since we need to consider the contributions of both the histogram and the map.
To keep things more simple we instead handle them individually, so that neither have to consider the other.

The case of the identity function is trivial since we don't do anything, but otherwise we will need to apply a rewrite rule on the `bucket_fun` as well. Luckily the compiler already has a function for differentiating lambdas.

This is implemented in the central `vjpSOAC` function in three steps:

1. Generate statement `mapStmt` which maps `bucket_fun` onto the input.
2. Generate statement `newHist` identical to the original histogram, but where the input is the output of `mapStmt`, and the `bucket_fun` is the identity function.
3. Apply rewrite rules to `newHist`, then `mapStmt`.

The extended cases of `vjpSOAC` becomes:

```
1  vjpSOAC :: VjpOps -> Pat Type -> StmAux () -> SOAC SOACS -> ADM () -> ADM ()
2  -- Histogram Case
3  vjpSOAC ops pat aux (Hist len args hist_op bucket_fun) m
4    | not $ isIdentityLambda bucket_fun = do
5      f' <- mkIdentityLambda $ lambdaReturnType bucket_fun
6      (args', stmts) <- runBuilderT' . localScope (scopeOfLParams []) $ do
7        letTupExp "input" $ Op $ Screma len args (mapSOAC bucket_fun)
8      let mapStmt = head $ stmsToList stmts
9      let newHist = Let pat aux $ Op $ Hist len args' hist_op f'
10     vjpStm ops stmt $ vjpStm ops newHist m
11
12 vjpSOAC ops pat aux hist@(Hist _ _ _ bucket_fun) m
13  | isIdentityLambda bucket_fun = do
14     diffHist ops pat aux hist m
```

`vjpStm` used in line 10 works like `vjpSOAC`, but on any statement. When `vjpStm` is called on `newHist`, it will case the statement until it reaches `vjpSOAC` again. Because the `bucket_fun` was substituted with the identity-function, the guard on line 4 will no longer match. It will instead match with the case beneath it at line 12, which subsequently applies the rewrite rule to the statement.

Now that the `bucket_fun` is dealt with, we can ignore it inside of `diffHist`.

The following sections will explain how the different cases of operators were implemented. Most intermediate steps of the pseudo code will be presented, along with its corresponding compiler code.
The code produced by the compiler can be found in the appendix, but please do keep in mind that part of the compiler pipeline optimizes the generated code. Code generating unused variables are removed, and `maps`, `reduce` and `lambdas` are fused whenever possible.

4.2.1 Special case: Addition

In order to reason about the optimization for `reduce_by_index` with $(+)$ as its operator, we first reason about the more simple case of `reduce`. We recall that the semantics for `reduce` are as follows:

$$y = \text{reduce} \odot e_{\odot} [v_0, v_1, ..., v_{n-1}] = e_{\odot} \odot v_0 \odot v_2 \odot ... \odot v_{n-1}$$

For an unknown operator $\odot$ we would compute the tangent of v_i as $\overline{v}_i = \frac{\partial(l_i \odot v_i \odot r_i)}{\partial v_i} \cdot \overline{y}$, but when we insert addition as the operator, we can simplify as such:

$$\overline{v}_i \mathrel{+}= \frac{\partial(l_i + v_i + r_i)}{\partial v_i} \cdot \overline{y} = \overline{y}$$

Now all $\overline{v}_i$ in **vs** are update with the same adjoint, being that of $\overline{y}$.
This simplifies the rewrite rule by not requiring us to perform the scan to compute l_i and r_i, and generalises to `reduce_by_index` by updating $\overline{v}_i$ with the adjoint of its corresponding bin:

$$\overline{vs}[i] \mathrel{+}= \overline{\text{histo}}[\text{inds}[i]]$$

The same logic applies to computing the adjoint of the original array, which is exactly identical to the adjoint of the result:

$$\overline{\text{dest}}[i] \mathrel{+}= \overline{\text{histo}}[i]$$

Because `dest` is consumed by `reduce_by_index`, and its value might be needed later in the reverse sweep, we start by copying it. Putting everything together we get the following psedocode:

```
- Original statement
let hist = reduce_by_index orig_dst (+) 0 inds vs

-- Forward sweep
let orig_dst_copy = copy orig_dst
let hist = reduce_by index orig_dst_copy (+) 0 inds vs
-- Reverse sweep
let hist_bar = lookupAdj hist
let dest_bar = hist_bar
let vs_bar += map (\bin -> if bin < len(dest) && bin > -1
                             then hist_bar[bin]
                             else 0) inds
```

Using the IR presented in section 4.1 we first construct a copy of `orig_dst` by a monadic operation such that the statement is captured and produced as output:

```
let orig_dst_copy = copy orig_dst
```

(a) Pseudo code

```
orig_dst_cpy <-
    letExp (baseString orig_dst ++ "_copy") $
            BasicOp $ Copy orig_dst
```

(b) Compiler code

We now reassemble the same `Hist` that we got as input, but using `orig_dst_copy` instead of `orig_dst` as the destination array. We use the monadic function `addStm` to bind the result to the original pattern specified:

```
let hist = reduce_by index
     orig_dst_copy (+) 0 inds vs
```

(a) Pseudo code

```
let histo' = Hist n [inds, vs]
             [HistOp shape rf [orig_dst_cpy] [ne] add_lam]
             bucket_fun
addStm $ Let pat aux $ Op histo'
```

(b) Compiler code

Computation of `vs_bar` requires us to build a map that applies a lambda on `inds`. The `Lambda` is constructed separately, and subsequently used to build the map.
To construct the `Lambda` needed we use `newParm` create the parameter for the bins, and use `runBodyBuilder` to build the body of the `Lambda` with those parameters:

```
(\bin -> if bin < len(dest) && bin >
      -1
              then hist_bar[bin]
              else 0)
```

(a) Pseudo code

```
1  bin_param <- newParam "bin" $ Prim int64
2  vs_adj_lam_body <- runBodyBuilder . localScope (scopeOfLParams (bin_param)) $
3      eBody
4        [ eIf -- if ind > 0 then 0 else ...
5          (eCmpOp (CmpSlt Int64) (eParam ind_param) (eSubExp int64Zero) )
6          (eBody [eSubExp $ int64Zero])
7          (eBody
8            [
9              eIf -- if histDim > ind then 0 else hist_bar[i]
10             (eCmpOp (CmpSlt Int64) (eSubExp histDim) (eParam ind_param) )
11             (eBody [eSubExp $ int64Zero])
12             ( do
13              r <- letSubExp "res" $ BasicOp $ Index hist_bar $ Slice $ [DimFix
          bin_param]
14              resultBodyM [r]
15           )
16     ]
17 let vs_adj_lam = Lambda [bin_param] vs_adj_lam_body [eltp]
18
```

(b) Compiler code

The constructs such as `eIf` `eCmpOp`, `eParam` ect. are monadic helper functions, of which there are too many to explain. A helper function with the name `e<Type>` produces a `<Type>` from its arguments, eg. `eIf` produces an If-statement.
The result of each path of the branch are defined on lines 6, 11 and 14 in the compiler code. For the result of the final branch on lines 12-14 we need to define an expression that computes `hist_bar[bin]`, and `resultBodyM` is a monadic function that returns the result as a body.
The variable `int64Zero` used in the compiler code is a `SubExp` with a 64-bit integer type and the value 0.

The needed lambda is now bound to `vs_adj_lam` which we can reference. Using `vs_adj_lam` we construct the required map, and bind the result to a `VName` that we can insert as the adjoint of `vs`:

```
let hist_bar = lookupAdj hist
let k = len(orig_dst)
let vs_bar +=
   map (\bin -> if bin < k && bin > -1
                   then hist_bar[bin]
                   else 0) inds
```

(a) Pseudo code

```
pe_bar <- lookupAdjVal $ pe
vs_bar <- letExp (baseString vs ++ "_bar") $
                Op $ Screma n [inds] $ ScremaForm [][] vs_adj_lam
void $ updateAdj vs vs_bar
```

(b) Compiler code

This concludes that case of (+) as the operator given to `reduce_by_index`. The complete compiler- and produced code can be found in the appendix at section 7.2.1.

4.2.2 Special case: Min/max

`Min` and `max` behave similarly by taking two values as input and outputting the smaller (`min`) or larger (`max`) value. We first reason for `Reduce`.

When reducing with either of these operators, you are essentially asking for the largest/smallest element of a given list i.e. only a single element from the list is returned. Whatever element was picked must therefore have contributed fully to the adjoint of the result, and should therefore be assigned the adjoint of the result.

As a simple example we use the following reduction:

$$\text{let } y = \text{reduce min } \infty \ vs$$

If the element at index i_k was the smallest element, then $\overline{v}_{i_k} \mathrel{+}= \overline{y}$, and for all other v_i we do nothing. The same logic applies to `max`.

The most straightforward way to facilitate this, is to lift the min/max operator to take two (value, index) tuples as input and pipe the index of the min/max element along. If the two elements are the same we simply pick the one with the lowest index. The lifted operator is designed as:

```
1 (\acc_v, acc_i, arg_v, arg_i -> if acc_v == arg_v
2                                   then (acc_v, min(acc_ind, arg_ind))
3                                   else let minmax = min/max(acc_v, arg_v)
4                                        if minmax == acc_v
5                                          then (acc_v, acc_ind)
6                                          else (arg_v, arg_ind))
```

The `min/max` operator on line 3 indicates that whichever of the the two is being used should be inserted here.

The generalisation to `reduce_by_index` is to implement the same approach bin-wise, which is easily achieved by using `reduce_by_index` with the lifted operator.

In order to do that we extend the neutral element to be a tuple with a negative index $(\text{ne}_{\text{min/max}}, -1)$. We also extend the destination array to be a tuple of the original values, and an index of (-1).
Since indices are always positive $(0 ... n-1)$, any resulting histogram with a negative index will indicate that the value from the destination array was picked. This is needed for the reverse sweep where we need to compute the adjoint of both the input values and the destination array.

Putting everything together we get the following pseudo code to implement using k as the size of the histogram:

```
-- Original statement
let hist = reduce_by_index orig_dst min/max ne_{min/max} vs

-- Forward sweep.
let orig_dst_copy = copy orig_dst
let minus_ones = replicate k -1
let iota_n = iota n

let maxind_lam =
    (\acc_v, acc_i, arg_v, arg_i -> if acc_v == arg_v
                                      then (acc_v, min(acc_ind, arg_ind) )
                                      else let minmax = min/max(acc_v, arg_v)
                                           if minmax == acc_v
                                              then (acc_v, acc_ind)
                                              else (arg_v, arg_ind))
let (hist, hist_inds) =
      reduce_by_index (orig_dst_copy, minus_ones) maxind_lam (ne_{min/max}, -1) (is, vs, iota_n)

-- Reverse sweep
let hist_bar = lookupAdj hist
let dest_bar = map2 (\ind, adj -> if ind < 0 then adj else 0) hist_inds hist_bar

let vs_bar_temp = lookupadj vs
let vs_bar_p = map2 (\ind, adj -> if ind < 0 then 0 else (vs_bar_temp[ind] + adj)) hist_inds hist_bar
let vs_bar = scatter vs_bar_temp hist_inds vs_bar_p
```

Now for the implementation. The header of the function is:

```
1 diffHist _vjops (Pat [pe]) aux soac m
2   | (Hist n [inds, vs] hist_max bucket_fun) <- soac,
3     True <- isIdentityLambda bucket_fun,
4     [HistOp shape rf [orig_dst] [ne] max_lam] <- hist_max,
5     Just bop <- isMinMaxLam max_lam,
```

This is much like the format presented in section 4.2, but where the operator given is named `max_lam`. Even though it is named `max_lam` inside the compiler, the guard on line 5 allows `min` to use this same rule for code generation.

For the forward sweep we start by initializing the three helper arrays:

```
let orig_dst_copy = copy orig_dst
let minus_ones = replicate k -1
let iota_n = iota n
```

(a) Pseudo code

```
orig_dst_copy <- letExp (baseString orig_dst ++ "_cpy") $
                    BasicOp $ Copy orig_dst
minus_ones <- letExp "minus_ones" $ BasicOp $
                    Replicate shape (intConst Int64 (-1))
iota_n <- letExp "iota_n" $
            BasicOp $ Iota n (intConst Int64 0) (intConst Int64 1) Int64
```

(b) Compiler code

Now we need to construct the `Hist` needed to compute the histogram. We first define the operator `HistOp` by giving it the destination of the operation, the neutral element, and the lifter operator `maxind_lam`. Once we have the `HistOp` we can construct the histogram we want to compute and bind it to the original pattern.

```
let (hist, hist_inds) =
    reduce_by_index (orig_dst_copy,
      minus_ones)
                    maxind_lam
                    (ne_min/max, -1)
                    (is, vs, iota_n)
```

(a) Pseudo code

```
let hist_op = HistOp shape rf [orig_dst_copy, minus_ones]
                              [ne, intConst Int64 (-1)]
                              maxind_lam

f' <- mkIdentityLambda [Prim int64, eltp, Prim int64]
auxing aux $ letBind hist_pat $ Op $
                Hist n [inds, vs, iota_n] [hist_op] f'
```

(b) Compiler code

Now that the forward sweep is complete, we move on to the reverse sweep.

We construct the `Lambda` to compute `dest_bar` (`lam_orig_bar`), and map it across `hist_inds` and `hist_bar`:

```
let hist_bar = lookupAdj hist
let dest_bar =
    map (\ind, adj ->
         if ind < 0
            then adj
            else 0
         ) hist_inds hist_bar
```

(a) Pseudo code

```
hist_bar <- lookupAdjVal $ hist
dest_bar <-
      letExp (baseString orig_dst ++ "_bar") $
        Op $
          Screma shapedim [inds, hist_bar]
            (ScremaForm [] [] lam_orig_bar)
insAdj orig_dst hist_bar
```

(b) Compiler code

Now we want to increase the adjoint of the values in `vs` that made it into the resulting histogram. We construct the needed lambda and name it `lam_vs_bar`, and apply it to `hist_inds` and `hist_bar`. `hist_inds` still points to the min/max element of a given bin in the original array `vs`, so we use that to scatter the values to the current adjoint of `vs`. If no element in `vs` was the min/max element it must have come from `orig_dst`. In that case the corresponding index in `hist_inds` will be -1, which the scatter will ignore.

```
let vs_bar_temp = lookupadj vs
let vs_bar_p =
    map (\ind, adj ->
        if ind < 0
           then 0
           else (vs_bar_temp[ind] + adj)
    ) hist_inds hist_bar
let vs_bar = scatter vs_bar_temp
                     hist_inds
                     vs_bar_p
```

(a) Pseudo code

```
vs_bar <- lookupAdjVal vs
vs_bar_p <-
       letExp (baseString vs_bar ++ "_partial") $
         Op $
           Screma shapedim [hist_inds, hist_bar]
                  (ScremaForm [] [] lam_vs_bar)
f'' <- mkIdentityLambda [Prim int64, eltp]
let scatter_soac = Scatter shapedim [hist_inds, vs_bar_p]
                                    f''
                                    [(Shape [n], 1, vs_bar)]
vs_bar' <- letExp (baseString vs ++ "_bar") $ Op scatter_soac
insAdj vs vs_bar'
```

(b) Compiler code

This completes the compiler transformation for the special case of min/max as the operator to `reduce_by_index`. Section 7.2.2 in the appendix contains more complete segments of compiler-code along with the produced code.

4.2.3 Special case: Multiplication

As with the other special case we first reason about the case of `reduce`, and then generalise the reasoning to `reduce_by_index`.
When $\odot$ is multiplication, computation of $\overline{v}_i$ becomes:

$$\overline{v}_i \mathrel{+}= \frac{\partial(l_i \cdot v_i \cdot r_i)}{\partial v_i} * \overline{y} = l_i \cdot r_i \cdot \overline{y}$$

Assuming the product of all vs in the list of elements is y, and that all vs are non-zero, basic math gives us $l_i \cdot r_i = y/v_i$. The issue of zeroes gives us three cases:

1. All elements a non-zero. In this case we can update the tangent of each v as $\overline{v}_i \mathrel{+}= \frac{y}{v_i} \cdot \overline{y}$.
2. Only a single element is zero, we will call this v_x. For all other vs $l_i \cdot r_i = 0$, so we update v_x with $l_i \cdot r_i \cdot \overline{y}$ and all other vs with 0.
3. More than one element is zero. $l_i \cdot r_i$ is zero for all v_i so we update them with 0, effectively doing nothing.

To facilitate case 1 and 2 we will need to compute the product of all the elements where 0's are replaced with the neutral element. For case 1 this will have no effect, but for case 2 this will allow us to use the result directly because $l_i \cdot 1 \cdot r_i = l_i \cdot r_i$.
In practice this means we will have to count the amount of zeroes encountered, so we can distinguish between which of the three cases we are in.

The generalisation to `reduce_by_index` is to use the same approach, but now on the level of the bins. For each bin we count the amount of zeroes that went into it, and replace them with the neutral element instead to compute the non-zero product.

This gives us the following pseudo code to implement:

```
-- Original statement
let hist = reduce_by_index orig_dst (*) 1 inds vs

-- Forward sweep
let nz_prd   = replicate len(orig_dst) 1
let zr_cts   = replicate len(orig_dst) 0


let nzel_zrct, nzel_zrct_flag = map (\v -> if v == 0 then (1, 1) else (v, 0)) vs

let non_zero_prod = reduce_by_index (nz_prod) (*) (1) (inds, nzel_zrct)
let zero_count    = reduce_by_index (zr_cts)  (+) (0) (inds, nzel_zrct_flag)

let hist_temp = map2 (\nz_prod, zeros -> if zeroes > 0 then 0 else nz_prod) non_zero_prod zero_count
let hist      = map2 (*) orig_dst hist_temp

-- reverse
let hist_bar = lookupAdj hist
let hist_orig_bar = map2 (*) hist_temp hist_bar
let hist_temp_bar = map2 (*) hist_orig hist_bar


let as_bar = map2  (\i, v -> let zr_cts = zero_count[i]
                             let pr_bar = hist_temp_bar[i]
                             let nz_prd = non_zero_prod[i]
                             if zr_cts == 0
                                then (nz_prd / v) * pr_bar
                                else if zr_cts == 1 && v == 0
                                     then nz_prd * pr_bar
                                     else 0
                )  inds vs
```

The following code is the functions header:

```
1 --special case *
2 diffHist _vjops (Pat [pe]) aux soac m
3   | (Hist n [inds, vs] hist_mul bucket_fun) <- soac,
4     True <- isIdentityLambda bucket_fun,
5     [HistOp shape rf [orig_dst] [ne] mul_lam] <- hist_mul,
6     Just mulop <- isMulLam mul_lam,
```

The values in `Hist` and `HistOp` are almost on the same format as the one explained in section 4.2, except the operator is called `mul_lam`. The guards on line 4 and 6 check that the `bucket_fun` is the identity function, and that the operator to use is multiplication.

We now move on to the forward sweep.
We start by initializing our three helper arrays:

```
let nz_prd   = replicate len(orig_dst) 1
let zr_cts   = replicate len(orig_dst) 0
```

(a) Pseudo code

```
nz_prods0 <- letExp "nz_prd" $ BasicOp $
                Replicate shape ne
zr_counts0 <- letExp "zr_cts" $ BasicOp $
                Replicate shape (intConst Int64 0)
```

(b) Compiler code

Next we need to transform our input values into a tuple of (0,1) when a zero is encountered, and (value, 1) otherwise. We start by building the map:

```
(\v -> if v == 0 then (1, 1) else (v, 0))
```

(a) Pseudo code

```
v_param <- newParam "v" $ eltp
map_lam_bdy <-
    runBodyBuilder . localScope (scopeOfLParams [v_param]) $
      eBody
        [ eIf
            (toExp $ CmpOpExp
                     (CmpEq eltp)
                     (LeafExp (paramName v_param) eltp)
                     (ValueExp (blankPrimValue eltp)))
            (resultBodyM [Constant $ onePrimValue eltp, intConst Int64 1])
            (resultBodyM [Var (paramName v_param), intConst Int64 0])
        ]
let map_lam = Lambda [v_param] map_lam_bdy [eltp, Prim int64]
```

(b) Compiler code

In the IR a list of tuples is actually a tuple of lists. We map over `vs` and unpack the results:

```
let nzel_zrct, nzel_zrct_flag =
    map (\v -> if v == 0
                 then (1, 1)
                 else (v, 0)
        ) vs

```

(a) Pseudo code

```
vs_lift <- letTupExp "nzel_zrct" $
             Op $ Screma n [vs] (ScremaForm [] [] map_lam)
let [nzel_zrct, nzel_zrct_flag] = vs_lift

```

(b) Compiler code

Now we want to compute the two histograms and while the pseudo code states shows them being computed separately, we can actually fuse them together. We define two `HistOps`, one where we are multiplying the non-zero values, and one where we add the flags. The destination of multiplication is the helper array `nz_prd`, and the destination of addition is `zr_cts`:

```
let hist_nzp = HistOp shape rf [nz_prods0] [ne] mul_lam
let hist_zrn = HistOp shape rf [zr_counts0] [intConst Int64 0] lam_add

```

(a) Compiler code

Now we can construct the fused histogram. The input was not defined for each of the two `HistOps`, and must be defined on `Hist`. For a fused `HistOp` of `[HistOp_1, HistOp_2]` we must align the input defined in `Hist` as `[inds_1, inds_2, vs_1, vs_2]`:

```
let non_zero_prod =
    reduce_by_index (nz_prod)
                    (*)
                    (1)
                    (inds, nzel_zrct)

let zero_count =
    reduce_by_index (zr_cts)
                    (+)
                    (0)
                    (inds, nzel_zrct_flag)

```

(a) Pseudo code

```
f' <- mkIdentityLambda [Prim int64, Prim int64, eltp, Prim int64]
let soac_exp = Op $ Hist n
                          [inds, inds, nzel_zrct, nzel_zrct_flag]
                          [hist_nzp, hist_zrn]
                          f'
auxing aux $ letBind soac_pat soac_exp

```

(b) Compiler code

The remaining part of the forward sweep is to compute `hist_temp` which is the histogram purely based on our input, and the resulting histogram `hist` obtained by mapping multiplication over `hist_temp` and `orig_dst`.
`hist_temp` is computed from `non_zero_prod` by checking if the corresponding bin has a zero in it. If there is no zeroes in that bin we can return the non-zero product. Otherwise, the non-zero product is invalid, and we return 0 as the product of that bin. We construct the lambdas and bind the results:

```
let hist_temp =
    map2 (\nz_prod, zeros ->
        if zeroes > 0
           then 0
           else nz_prod
        ) non_zero_prod zero_count

let hist = map2 (*) orig_dst hist_temp

```

(a) Pseudo code

```
hist_temp <-
      letExp "hist_temp" $
        Op $ Screma shapedim
                [nz_prods, zr_counts]
                (ScremaForm [] [] (Lambda ps2 lam_bdy_2 [eltp]))
auxing aux $
      letBind (Pat [pe]) $
        Op $ Screma shapedim
                 [orig_dst, hist_temp]
                 (ScremaForm [] [] (Lambda ps3 lam_pe_bdy [eltp]))

```

(b) Compiler code

For the reverse sweep we start by performing a lookup on the adjoint of the result, and using that compute `hist_orig_bar` and `hist_temp_bar`:

```
let hist_bar = lookupAdj hist
let hist_orig_bar = map2 (*) hist_temp
      hist_bar
let hist_temp_bar = map2 (*) hist_orig
      hist_bar

```

(a) Pseudo code

```
hist_bar <- lookupAdjVal $ patElemName pe
orig_bar <-
      letExp (baseString orig_dst ++ "_bar") $
        Op $
          Screma
            shapedim
            [hist_temp, hist_bar]
            (ScremaForm [] [] mul_lam')

updateAdj orig_dst orig_bar

hist_temp_bar <-
      letExp (baseString hist_temp ++ "_bar") $
        Op $
          Screma
            shapedim
            [orig_dst, hist_bar]
            (ScremaForm [] [] mul_lam'')

```

(b) Compiler code

Now, all we need to do is to compute the adjoint of `vs`. We construct the lambda needed as `vs_bar_lam`, and map over `inds` and `vs`:

```
let as_bar =
    map2  (\i, v ->
            let zr_cts = zero_count[i]
            let pr_bar = hist_temp_bar[i]
            let nz_prd = non_zero_prod[i]
            if zr_cts == 0
              then (nz_prd / v) * pr_bar
              else if zr_cts == 1 && v == 0
                     then nz_prd * pr_bar
                     else 0
          )  inds vs

```

(a) Pseudo code

```
vs_bar <-
      letTupExp (baseString vs ++ "_bar") $
        Op $ Screma
                n
                [inds, vs]
                (ScremaForm [] [] (vs_bar_lam))
updateAdj vs vs_bar

```

(b) Compiler code

This concludes the implementation of multiplication as the operator to `reduce_by_index`. The full implementation along with the produced code can be found in the appendix at section 7.2.3.

4.2.4 General approach

In this section the general approach will be presented. The general approach encompasses any operator not caught by the special cases of `addition`, `multiplication`, or `min/max`. We recall the rewrite rule from section 3.2:

$$
\text{let histo} = \text{reduce\_by\_index dest} \odot e_{\odot} \text{as} \Longrightarrow
\begin{array}{l}
\text{let } as_{\text{sorted}} = \text{radix\_sort } as \\
\text{let lis} = \text{segScan}_{\text{exc}} \odot e_{\odot}\, as_{\text{sorted}} \\
\text{let ris} = \text{reverse } as_{\text{sorted}} \\
\qquad\qquad \rhd \text{segScan}_{\text{exc}}(\lambda\, x \odot y \to y \odot x)\, e_{\odot} \;\rhd\; \text{reverse} \\
\text{let hist}_{\text{temp}} = \text{get\_segment\_sum lis } as_{\text{sorted}} \\
\text{let histo} = \text{map2} \odot \text{ dest hist}_{\text{temp}} \\
\vdots \\
\text{let } \overline{\text{dest}} = \text{map } f_1 \text{ dest hist}_{\text{temp}} \\
\text{let } \overline{\text{hist}}_{\text{temp}} = \text{map } f_2 \text{ dest hist}_{\text{temp}} \\
\text{let } \overline{as} \mathrel{+}= \text{map4 } f_3 \text{ lis as ris inds}
\end{array}
\tag{9}
$$

The pseudo code for the complete implementation can be found in the appendix at section 7.1. It has been omitted here since it is mostly covered by the rewrite rule in equation 9 with the filter code prepended, but it does include some steps that will not be highlighted in this section. Some of the functions in the rewrite rule, such as segmented scan, requires us to build flag arrays. In this segment i will mainly present the steps of the rewrite rule.

Forward sweep When beginning implementation, another step was introduced due to practical reasons. The list of bins accepted by `reduce_by_index` are of type `int64`, giving the possibility of passing negative values. Radix sort can be implemented to handle signed values, but the most straightforward implementation does not. The `Futhark`-compiler does not have a built-in method of sorting values, so `radix` sort needed to be implemented by hand. When you consider that any negative value would be an invalid bin, and should be ignored anyway, the solution to this problem was to initially filter our invalid bins.
This includes bins that are greater than the length of the histogram. I therefore filter out any bin not in the range $[0, ..., (k-1)]$ where k is the length of the histogram.

In order to get an understanding of how flag arrays are used throughout the rest of the code, we take a look at the implementation of the filter, which uses them a lot. The following is the pseudo code:

```
1 let flags = map (\bin -> if 0 <= ind <= m then 1 else 0) inds
2 let flag_scanned = scan (+) 0 flags
3 let n' = last flags_scanned
4 let new_inds = map (\(flag, flag_scan) -> if flag == 1 then flag_scan - 1 else -1) flags
    flag_scanned
5 let new_indexes = scatter (Scratch int n') new_inds (iota n)
6 let new_bins = map (\i -> bins[i]) new_indexes
```

The lines of code do the following:

1. A simple map of our predicate onto `inds`. Return 1 if we want to select the element, and 0 otherwise.

2. An inclusive scan of `flags` to get accumulative sum.

3. The last element of scan must be the number of elements we picked.

4. This list is needed for the following `scatter`. We map over `flags` and `flag_scanned` and check if the flag is set. If it is, we can subtract 1 from the sum up to this point, to find the position of that element in the filtered array. Otherwise we return -1 which will make the following `scatter` ignore this element.

5. `(iota n)` gives us the original index of each element in the original list, and we `scatter` those values into an empty array of size n' to the positions defined in `new_inds`.

6. Now that we have the indices of all values that met the predicate, we collect the respective bins of each of those elements.

Some of the code for filtering can be found in the appendix at section 7.2.4 figure 45.

After filtering we can go ahead with `radix` sorting.
`Radix sort` is implemented by looping over each bit from least significant to most significant. Each iteration we partition the current state of values such that those with 0's are followed by values with a 1. This is achieved by first computing the flag-array `bits` by simply checking

the current bit, and then using `partition2` to compute the new indices used to collect the new arrangement.

We need elements of the same bin to lie consecutively in a list so we sort with respect to the bins, but we also need to keep track of their original positions in order to update the adjoints later.

By sorting with respect to the bins we get a new arrangement of values, which can be applied to both the bins and the original indices after each iteration. The sorted values can be generated from the sorted indices afterwards, so we do not have to move them around multiple times as well:

```
-- Radix sort (new_bins, new_indexes) w.r.t. new_bins.
let (sorted_is, sorted_bins) =
   loop (new_indexes, new_bins) for i < 63 do
     let bits = map (\ind_x -> (ind_x >> i) & 1) new_bins
     let newidx = partition2 bits (iota n')
     in (map(\i -> new_indexes[i]) newidx, map(\i -> new_bins[i]) newidx)
let sorted_vals = map(\i -> vs[i]) sorted_issorted_vals = map(\i -> vs[i]) sorted_is
```

We start by defining the function computing `bits`. We shift i amount of times to the right where i is the loop counter.

```
let bits =
    map (\ind_x -> (ind_x >> i) & 1) new_bins
```

(a) Pseudo code

```
ind_x <- newParam "ind_x" $ Prim int64
        bits_map_bdy <- runBodyBuilder . localScope (scopeOfLParams [ind_x]) $
          eBody
          [
            eBinOp (And Int64)
              (eBinOp (LShr Int64) (eParam ind_x) (eSubExp $ Var i2))
              (eSubExp $ int64One)
          ]
        let bits_map_lam = Lambda [ind_x] bits_map_bdy [Prim int64]
        bits <- letExp "bits" $ Op $
                     Screma n' [binsForLoop] (ScremaForm [] [] bits_map_lam)
```

(b) Compiler code

We then construct the `iota` and use `partition2`-function to get the new arrangement of elements.

```
let newidx = partition2 bits (iota n')
```

(a) Pseudo code

```
-- partion2Maker - Takes flag array and values and creates a scatter SOAC
--                   which corresponds to the partition2 of the inputs
-- partition2Maker size flags values =
partition2Maker :: SubExp -> VName -> VName -> BuilderT SOACS ADM (SOAC SOACS)

temp_iota <- letExp "temp_iota" $ BasicOp $ Iota n' int64Zero int64One Int64
scatter_soac <- partition2Maker n' bits temp_iota
newidx <- letExp (baseString inds ++ "_scattered") $ Op $ scatter_soac
```

(b) Compiler code

I have omitted the code for `partition2Maker` which can be found in the appendix at section 7.3.1.

At the end of each loop we collect the bins and original indices before ending the iteration:

```
let new_indexes =
    map(\i -> filtered_indexes[i]) newidx)
let new_bins =
    map(\i -> filtered_bins[i]) newidx)
```

(a) Pseudo code

```
inner_indx_idx <- newParam "inner_indexes_idx" $ Prim int64
inner_indx_bdy <- runBodyBuilder . localScope (scopeOfLParams [inner_indx_idx
      ]) $ do
  tmp <- letSubExp "indexes_body" $
            BasicOp $ Index (paramName paramIndexes)
                            (fullSlice (Prim int64)
                            [DimFix (Var (paramName inner_indx_idx))])
  resultBodyM [tmp]
let inner_indx_lambda = Lambda [inner_indx_idx] inner_indx_bdy [Prim int64]
new_indexes <-
    letSubExp "new_indexes" $
    Op $ Screma n' [newidx] $ ScremaForm [][] inner_indx_lambda
...
--- Equivalent code for new_bins
```

(b) Compiler code

The upper-bound of i (number of loop iterations) is set to be 63 since any signed integer with a 1 as it's most significant bit is a negative number, and they have all been filtered out at this point. A more elegant solution would be to set the upper-bound it be $\lceil \log_2(\text{histDim}) \rceil$ where `histDim` is the length of the histogram. Any bin larger that `histDim` has also been filtered out, so sorting values beyond that is not optimal.

The complete code for radix sort can be found in the appendix at section 7.2.4 figure 46.

Now that our values are sorted, we can compute l_i and r_i by performing both a forward and reverse segmented exclusive scan on `sorted_vals`. Doing this requires us to compute the required flag array, which can be computed from the sorted bins:

```
let final_flags =
    map (\index ->
        let curr = sorted_bins[index]
        let prev = sorted_bins[index-1]
        if curr == prev
          then 0
          else 1) iota n'
```

(a) Pseudo code

```
iota_n' <- letExp "iota_n'" $ BasicOp $
                Iota n' (intConst Int64 0) (intConst Int64 1) Int64
mk_flag_body <- runBodyBuilder . localScope (scopeOfLParams [bin, index]) $ do
  .
  -- Lambda code
  .

let mk_flag_lambda = Lambda [bin, index] mk_flag_body [Prim $ IntType Int8]
final_flags <- letExp "final_flags" $ Op $
                Screma n' [sorted_bins, iota_n'] $
                    ScremaForm [][] mk_flag_lambda
```

(b) Compiler code

Where n' is the length of `sorted_bins`.

Furthermore we need to lift the operator to take a (value, flag) tuple and reset accumulation when the start of a new segment is encountered. We first lift the operator to be used for an inclusive segmented scan:

```
let lifted_op = (\(f1,v1) (f2,v2) ->
                   let f = f1 || f2
                   let v = if f2 then v2 else v1 ⊙ v2
                   in (f, v))
```

The operators now functions as a segmented inclusive scan, but we needed an exclusive segmented scan. To fix this we shift each element to the right, padding with the neutral element:

```
let tmp = map (\(f,i) -> if f
                          then (f, ne)
                          else (f, sorted_vals[i-1])
              ) final_flags (iota n)
```

We can now compute l_i by scanning with our lifted operator. The neutral element is set to $(false, e_\odot)$ but in practise it does not matter what it is. The first element encountered is the start of a segment, so (v2, f2) is always picked here.

The code for lifting the operator is implemented as the helper function `mkSegScanExc` and can be found in the appendix at section 7.3.2.

The lifted operator returns a tuple of flags and values, but we only unpack the values:

```
let lis = scan lifted_op (e⊙, false)
                         tmp
                         final_flags
```

(a) Pseudo code

```
-- Lift a lambda to produce an exlusive segmented scan operator.
-- mkSegScan operator neutral_elem size values flags
mkSegScanExc :: Lambda SOACS -> [SubExp] -> SubExp -> VName -> VName
                -> ADM (SOAC SOACS)

-- mkSegScanExc also computes the map for tmp
seg_scan_exc <- mkSegScanExc f nes n' sorted_vals final_flags
fwd_scan <- letTupExp "fwd_scan" $ Op seg_scan_exc
let [_, lis] = fwd_scan
```

(b) Compiler code

The full code for computation of `lis` can be found in the appendix figure 47.

In order to compute r_i we need to reverse the list of elements, perform the scan, and then reverse back again. The `lifted_op` can be reused for this purpose, but the `final_flags` need to be fixed. As an example we reverse a set of flags:

```
Flags    = [1,0,0,0,1,0.1,0,0]
Reversed = [0,0,1,0,1,0,0,0,1]
Correct  = [1,0,0,1,0,1,0,0,0]
```

Where `Flags` is the array being reversed, `Reversed` is the result of directly reversing `Flags`, and `Correct` being the correct state of the reversed flags that we wish.
The flags in `Reversed` are slightly off, and can be fixed by shifting each element to the right. We pad with 1 since the first element should always be the start of a new segment:

```
let final_flags_rev = reverse final_flags
let rev_flags =
    map (\i -> if i == 0
                then 1
                else final_flags_rev[i-1]
        ) (iota n')
```

(a) Pseudo code

```
-- eReverse: Reverses the order of the input-array
-- eReverse: VName
eReverse :: VName -> VName

final_flags_rev <- eReverse final_flags
i' <- newParam "i" $ Prim int64
rev_flags_body <- runBodyBuilder . localScope (scopeOfLParams [i']) $ do
    -- Lambda body
let rev_flags_lambda = Lambda [i'] rev_flags_body [Prim int8]
rev_flags <- letExp "rev_flags" $ Op $
                Screma n' [iota_n'] $
                    ScremaForm [][] rev_flags_lambda
```

(b) Compiler code

Where n' is the length of `final_flags_rev`.

We can now compute r_i by reversing the shifted values `tmp`, scanning them with the reversed flags, and then reversing back:

```
let rev_vals = reverse tmp
let ris_rev =
    scan lifted_op (e⊙, false)
                   rev_vals
                   rev_flags
let ris = reverse ris_rev
```

(a) Pseudo code

```
-- eReverse: Reverses the order of the input-array
-- eReverse: VName
eReverse :: VName -> VName

-- Lift a lambda to produce an exlusive segmented scan operator.
-- mkSegScan operator neutral_elem size values flags
mkSegScanExc :: Lambda SOACS -> [SubExp] -> SubExp -> VName -> VName
             -> ADM (SOAC SOACS)

-- Run segmented scan on reversed arrays.
rev_vals <- eReverse sorted_vals
rev_seg_scan_exc <- mkSegScanExc f nes n’ rev_vals rev_flags
rev_scan <- letTupExp "rev_scan" $ Op rev_seg_scan_exc
let [_, ris_rev] = rev_scan
ris <- eReverse ris_rev
```

(b) Compiler code

Once again the complete code for this step can be found in the appendix at section 7.2.4 figure 48.

Now that we have performed both the forward and reverse scan, the last part of the forward sweep is to compute the resulting histogram. The rewrite rule states that the reverse sweep needs to compute the adjoint of the original array `orig_dst`, and the histogram generated by reducing our elements `vs`. We call our partial result `hist_temp`. Since we need both partial results for the reverse sweep, we bring them into scope here and combine them to create the resulting histogram:

```
let hist_temp = reduce_by_index (replicate k e⊙) ⊙ e⊙ inds vs
let hist = map ⊙ orig_dst hist_temp
```

Where k is the length of the histogram, and `hist` is the name of the result.

The computation of `hist_temp` in the code above states that we should use `reduce_by_index`, but this is only meant to give an intuitive understanding of what `hist_temp` is. In reality we are going to compute it from `lis`, which was the whole reason that computation was moved to the forward sweep.

Since `lis` contains a segmented exclusive scan of all our elements, the last element of each segment in `lis` will be the sum of that segment, only missing the last element (due to exclusive scan). A short example to clarify using (+) as the operator:

```
Flags    = [1,0, 0,1,0]
Values   = [4,3, 7,2,4]
ExcScan  = [0,4, 7,0,2]

LastElem = [0,0, 1,0,1]
Sum      = [0,0,14,0,6]
```

In this example `Values` is the values we want to reduce, `Flags` are the flags used to compute `ExcScan`, and `LastElem` is a flag array indicating the end of a segment. By taking the last element of each segment from `Values` and adding them with the last element of each segment in `LastElem`, we get a list containing the total sum of each segment in those positions.

We implement this by using a `scatter` to get the values from `Values` and `LastElem`. Since there is no assurance that all bins in the histogram will be populated, we start by allocating two arrays with neutral elements for the last elements of both `lis` and our values:

```
let bin_last_lis_dst = replicate k e⊙
let bin_last_v_dst   = replicate k e⊙
```

(a) Pseudo code

```
bin_last_lis_dst <- letExp "bin_last_lis_dst" $
                        BasicOp $ Replicate shape (head nes)
bin_last_v_dst <- letExp "bin_last_v_dst" $
                      BasicOp $ Replicate shape (head nes)
```

(b) Compiler code

Where k once again is the length of the histogram.

These arrays will be the destination of our scatter. Their sizes ensures that once we add them together, the dimensions will fit that of the histogram.

Now we need the array of indices to scatter with. Both `lis` and `sorted_vals` have length n', and by mapping each index using (iota n') we can check if the flag of the following element is the start of a new segment. If it is, we want to take the value, and otherwise leave it. Since `scatter` ignores values scattered to index -1, we can return that index for ignoring values. For the ones that we want, we want to scatter the value to its corresponding bin. We therefore also map over `sorted_bins`:

```
let scatter_arr =
    map (\i, bin ->
        if i == n'-1
            then bin
            else if final_flags[i+1] == 1
                 then bin
                 else -1
        ) (iota n') sorted_bins
```

(a) Pseudo code

```
i'' <- newParam "i''" $ Prim int64
current_bin <- newParam "current_bin" $ Prim int64
scatter_arr_body <-
    runBodyBuilder . localScope (scopeOfLParams [i'', current_bin]) $ do
    -- Lambda code
let scatter_arr_lam = Lambda [i'', current_bin] scatter_arr_body [Prim int64]
scatter_arr <- letExp "scatter_arr" $
                    Op $ Screma n' [iota_n', sorted_bins] $
                    ScremaForm [][] scatter_arr_lam
```

(b) Compiler code

Using `scatter_arr` we can `scatter` the last elements of each segment from `sorted_vals` and `lis` to their destination arrays:

```
let bin_last_lis   =
    scatter bin_last_lis_dst
            scatter_arr
            lis
let bin_last_v     =
    scatter bin_last_v_dst
            scatter_arr
            sorted_vals
```

(a) Pseudo code

```
f''' <- mkIdentityLambda [Prim int64 , t]
bin_last_lis <- letExp "bin_last_lis" $ Op $
                    Scatter n' [scatter_arr, lis]
                                f'''
                                [(shape, 1, bin_last_lis_dst)]

f'''' <- mkIdentityLambda [Prim int64 , t]
bin_last_v <- letExp "bin_last_v" $ Op $
                Scatter n' [scatter_arr, sorted_vals]
                            f''''
                            [(shape, 1, bin_last_v_dst)]
```

(b) Compiler code

The full code for these computations can be found in section 7.2.4 figure 49.

The last step of the forward sweep is to combine `bin_last_lis` and `bin_last_v` to create `hist_temp`, and combine `hist_temp` with `orig_dst` to create `hist_res`:

```
let hist_temp = map2 ⊙ bin_last_lis
        bin_last_v
let hist_res  = map2 ⊙ hist_temp orig_dst
```

(a) Pseudo code

```
lis_param <- mapM (newParam "lis_param") [t]
v_param   <- mapM (newParam "v_param") [t]

-- Update operator to use lis_param and v_param instead
op1_lam <- renameLambda f
op1 <- mkLambda (lis_param ++ v_param) $ do
    eLambda op1_lam (map (eSubExp . Var . paramName) (lis_param ++ v_param))

hist_temp <- letExp "hist_temp" $
    Op $ Screma histDim [bin_last_lis, bin_last_v] $ ScremaForm [][] op1

-- Equivalent code for hist_res

hist_res <- letExp "hist_res" $
    Op $ Screma histDim [orig_dst, hist_temp] $ ScremaForm [][] op2

letBind pat $ BasicOp $ SubExp $ Var hist_res
```

(b) Compiler code

This last step bound the result to the original pattern and marks the end of the forward sweep.

Reverse sweep For the reverse sweep we need to compute the adjoint of the input values `vs` and the original histogram `orig_dst`. As stated in section 3.2, where the rewrite rule for `reduce_by_index` was presented, the adjoint of `hist_temp` is needed to compute the adjoint of `vs`. The adjoint of a single bin i in `orig_dst` and `hist_temp` can be computed as:

$$\overline{\text{hist}}_{\text{temp}}[i] = \frac{\partial(\text{hist}_{\text{temp}}[i] \odot \text{orig}_{\text{dst}}[i])}{\partial \text{hist}_{\text{temp}}[i]} \cdot \overline{\text{hist}}_{\text{res}}[i]$$
$$\overline{\text{orig}}_{\text{dst}}[i] = \frac{\partial(\text{hist}_{\text{temp}}[i] \odot \text{orig}_{\text{dst}}[i])}{\partial \text{orig}_{\text{dst}}[i]} \cdot \overline{\text{hist}}_{\text{res}}[i]$$

All the arrays used are of the same size, so we can compute all the adjoints by means of a `map`. To keep it simple we do this in two steps, first computing $\frac{\partial(\text{hist}_{\text{temp}}[i] \odot \text{orig}_{\text{dst}}[i])}{\partial \text{hist}_{\text{temp}}[i]}$, and then multiplying with $\overline{\text{hist}}_{\text{res}}[i]$:

```
let hist_temp_op = (λx, y → ∂(x⊙y)/∂x)
let orig_dst_op = (λx, y → ∂(x⊙y)/∂y)

let hist_temp_bar_temp = map2 hist_temp_op hist_temp          orig_dst
let hist_temp_bar      = map2 *            hist_temp_bar_temp hist_bar

let orig_dst_bar_temp  = map2 orig_dst_op  hist_temp          orig_dst
let orig_dst_bar       = map2 *            orig_dst_bar_temp  hist_bar
```

To create the lambdas `hist_temp_op` and `orig_dst_op`, an existing helper function `mkScanAdjointLam` was used. It takes a lambda that is assumed to take two parameters, and returns a lambda that differentiates with respect to one or the other. We use this to compute the temporary results, before multiplying the adjoint of the result onto them:

```
let hist_temp_bar_temp =
    map2 hist_temp_op
         hist_temp
         orig_dst

let orig_dst_bar_temp =
    map2 orig_dst_op
         hist_temp
         orig_dst
```

(a) Pseudo code

```
data FirstOrSecond = WrtFirst | WrtSecond

-- computes 'd(x op y)/dx' or d(x op y)/dy depending on FirstorSecond
-- mkScanAdjing: Operator FirstOrSecond
mkScanAdjointLam :: Lambda SOACS -> FirstOrSecond -> ADM (Lambda SOACS)

hist_orig_bar_temp_lambda <-
    mkScanAdjointLam vjops hist_orig_lam WrtFirst
hist_temp_bar_temp_lambda <-
    mkScanAdjointLam vjops hist_temp_lam WrtSecond

hist_orig_bar_temp <- letExp "hist_orig_bar_temp" $
    Op $ Screma histDim [orig_dst, hist_temp] $
            ScremaForm [][] hist_orig_bar_temp_lambda

hist_temp_bar_temp <- letExp "hist_temp_bar_temp" $
    Op $ Screma histDim [orig_dst, hist_temp] $
            ScremaForm [][] hist_temp_bar_temp_lambda
```

(b) Compiler code

We multiply the adjoint of the result onto each of them to obtain `hist_temp_bar` and `hist_orig_bar`:

```
let hist_temp_bar =
    map2 *
         hist_temp_bar_temp
         hist_bar

let orig_dst_bar =
    map2 *
         orig_dst_bar_temp
         hist_bar
```

(a) Pseudo code

```
-- getMulOp returns an instance of the multiplication operator that works for
     a given type.
getMulOp:: Type -> BinOp

let mulOp = getMulOp t

-- Takes name of two params, a binOp and the type of params and gives a lambda
      of that application
-- mkSimleLambda: first_param, second_param, operator, type
mkSimpleLambda :: String -> String -> BinOp -> Type -> ADM (Lambda SOACS)

mul_hist_orig_res_adj <- mkSimpleLambda "orig_adj" "res_adj" mulOp t
mul_hist_temp_res_adj <- mkSimpleLambda "temp_adj" "res_adj" mulOp t

hist_orig_bar <- letExp "hist_orig_bar" $ Op $
    Screma histDim [hist_orig_bar_temp, hist_res_bar] $
        ScremaForm [][] mul_hist_orig_res_adj

hist_temp_bar <- letExp "hist_temp_bar" $ Op $
    Screma histDim [hist_temp_bar_temp, hist_res_bar] $
        ScremaForm [][] mul_hist_temp_res_adj
```

(b) Compiler code

The produced code for computation of `hist_temp_bar` and `hist_orig_bar` can be found in the appendix at section 7.2.4 figure 50.

The final step is to compute the adjoint of `vs` by means of a map. From section 3.2 we recall the function for computing the adjoint of `vs`:

$$f_3 \ : \ \left(\lambda \, l_i, a_i, r_i, bin \rightarrow \frac{\partial(l_i \odot a_i \odot r_i)}{\partial a_i} \cdot \overline{histo}[bin]\right)$$

The most easy way to get this behaviour is to use one of the other rewrite rules already implemented, namely `vjpMap`:

```
vjpMap :: [Adj] -> SubExp -> Lambda -> [VName] -> ADM ()
```

`[Adj]` is a list of adjoints of the result of the map, `SubExp` is the length of the input, `Lambda` is the lambda to be used on the input to obtain the result and `[VName]` is a list the names of input arrays.

`vjpMap` will use the input and lambda given, and compute the adjoint of each input with respect to the output. If we use the input of `lis`, `vs` and `ris`, then `vjpMap` will compute the contribution from each of the three lists, although we will only be using that of `vs`.

To use vjpMap we need to fix two problems:

1. The operator $\odot$ needs to be lifted to take three arguments instead of two.
2. The adjoint of the result given to vjpMap also needs to be of the same dimensions of vs, which it currently is not.

We first lift the operator $\odot$ to a lambda that has three parameters: $f_2 : (\lambda l_i, a_i, r_i \rightarrow l_i \odot a_i \odot r_i)$. Lifting of the operator is done by the helper function mkF.

To get the array of adjoints we simply map over the sorted_bins and extract the adjoint of that bin:

```
let hist_temp_bar_repl =
    map (\bin -> hist_temp_bar[bin])
        sorted_bins
```

(a) Pseudo code

```
sorted_bin_param <- newParam "sorted_bin_p" $ Prim int64
hist_temp_bar_repl_body <-
    runBodyBuilder . localScope (scopeOfLParams [sorted_bin_param]) $ do
        hist_temp_adj <- letSubExp "hist_temp_adj" $
            BasicOp $ Index hist_temp_bar (fullSlice (Prim int64)
                          [DimFix (Var (paramName sorted_bin_param))])
        resultBodyM [hist_temp_adj]

let hist_temp_bar_repl_lambda =
    Lambda [sorted_bin_param] hist_temp_bar_repl_body [t]
hist_temp_bar_repl <- letExp "hist_temp_bar_repl" $
    Op $ Screma n' [sorted_bins] $
        ScremaForm [][] hist_temp_bar_repl_lambda
```

(b) Compiler code

Now we can use hist_temp_bar_repl as the adjoint of the result for vjpMap, binding the adjoint of vs into the ADM-monad, which we can then extract. The following is the compiler code which computes the adjoint of vs.

```
vjpMap [AdjVal $ Var hist_temp_bar_repl]
       n'
       lam_adj
       [lis, sorted_vals, ris]
vs_bar_contribs_reordered <- lookupAdjVal sorted_vals
```

Where lam_adj is the lifted operator.

The last step is to back-permute the adjoints to the correct pre-sorted version of vs. Some values might have been filtered out at the start of the forward sweep so their adjoint should be 0. We initialize an array 0's to be the destination of a scatter and then scatter the adjoints using the sorted indexes sorted_is:

```
let vs_bar_dst = replicate n 0
let vs_bar =
    scatter vs_bar_dst
            sorted_is
            vs_bar_contribs_reordered
```

(a) Pseudo code

```
vs_bar_contrib_dst <- letExp "vs_bar_contrib_dst" $
    BasicOp $ Replicate (Shape [n]) (getBaseAdj t)

f'''''' <- mkIdentityLambda [Prim int64, t]
vs_bar_contrib <- letExp "vs_bar_contrib" $
    Op $ Scatter n'
            [sorted_is, vs_bar_contrib_reordered]
            f''''''
            [(Shape [n], 1, vs_bar_contrib_dst)]

void $ updateAdj vs vs_bar_contrib
```

(b) Compiler code

This marks the end of the reverse sweep, and the rewrite rule itself. The complete code for the computation of vs_bar and the corresponding produced code can be found in the appendix at section 7.2.4 figure 51.

4.2.5 Limitations

The implementation comes with some limitations that restricts its usage, of which i have identified two. The general idea is that any legal instance of the `Hist` type which `reduce_by_index` accepts should be able to have reverse mode AD applied to it, but that is not the case currently.

The histogram type `Hist` takes a list of `HistOps`, and a list of destination arrays, such that you can compute multiple histograms with different operators on the same input. My implementation would not accept this, as it has a guard that defines both of these lists to only contain one element each.
The most straightforward way to handle this would be to handle them individually by reconstructing each of the statements that were fused, and mapping `diffHist` across them.

Another issue is that the type of input-values accepted by the implementations is limited to singletons. It could be a tuple, triplet etc. but that is not yet implemented. If each input variable is a triple (x^1, x^2, x^3), then for each bin in `hist_temp` we would have to compute the adjoint of each element of the tuple.
For some x_i where $is[i] = j$ we can compute the result of bin j as:

$$y[j] = l_i \odot (x_i^1, x_i^2, x_i^3) \odot r_i$$

In order to compute the adjoint of each value of the tuple, we would have to compute with respect to each one of them. For the adjoint of x_i^1 we would have to compute:

$$\overline{x}_i^1 \mathrel{+}= \frac{\partial(l_i \odot (x_i^1, x_i^2, x_i^3) \odot r_i)}{\partial x_i^1} \cdot \overline{y}[j] \tag{10}$$

The implemented method of computing derivatives of singleton values is mapped across each element. To support tuples this map would need another map nested inside, which would apply equation 10 to each element of the tuple instead.

5 Evaluation

This section present the measures taken to verify that the implementations are working as intended.
The first part concerns correctness of the output of the produced code by writing and generating tests in the source language `Futhark`.
The second part presents the performance of the four different cases and shows that the work-depth asymptotic of `reduce_by_index` is preserved when reverse mode AD is applied to it.

In order to validate and benchmark the four different cases, four different operators were needed. The special cases should be self-explanatory, but given the implementations limitation on usage of tuples, no associative and commutative operator not covered by the special cases came to mind.
As a result, validation and benchmarks for the general case was performed by simply commenting out the special case for multiplication, forcing the usage of the general case.

5.1 Validation

The source language of `Futhark` provides certain tools to help testing. The most simple method of testing is by writing a program that performs the action you wish to test. `Futhark` then lets you define tests as comments, which will be run when the program is given to its testing tool at the command line.
To define tests you can specify the input, and the corresponding output:

```
-- Simple histogram with multiplication
-- ==
-- compiled input { [1i64, 3i64, 2i64,-1i64, 2i64, 1i64, 1i64, 2i64, 3i64, 2i64,-1i64, 2i64, 2i64]
--                  [1f32, 1f32, 1f32, 1f32, 1f32, 1f32, 1f32, 1f32, 1f32, 1f32, 1f32, 1f32, 1f32]
--                  [4f32, 3f32, 2f32, 1f32]
--                  [9f32, 8f32, 7f32, 5f32]
--          }
-- output { [8f32, 5f32, 7f32, 0f32, 7f32, 8f32, 8f32, 7f32, 5f32, 7f32, 0f32, 7f32, 7f32]
--          [9f32, 8f32, 7f32, 5f32]
--        }

let histo_plus [w][n] (is: [n]i64) (vs: [n]f32, hist: [w]f32) : [w]f32 =
  reduce_by_index (copy hist) (+) 0.0f32 is vs

entry main [n][w] (is: [n]i64) (vs: [n]f32) (hist: *[w]f32) (hist_bar: [w]f32) =
  vjp (histo_plus is) (vs,hist) hist_bar
```

The code above is a simple tests of `reduce_by_index` using addition.
With filename `histo-plus.fut` it is run from the command line as "`futhark test histo-plus.fut`".
The tool reads the input, passes it to the function, and compares the output with the one defined in the test. All four cases were tested in this manner.

While one can calculate the expected result of a given input with pen and paper, expected results can also be generated using the forward mode.
This does however make these tests dependant on the validity of forward mode. Considering that forward mode has a much different approach to computing derivatives, and has been implemented by someone else, the odds of the same bug existing in both modes is very low. Forward mode is also documented with tests of its own.

A method of more thorough testing was to generate random data-sets, using forward mode to compute their expected outcomes. `Futhark` provides a tool to easily generate random data-sets, and even allows for bounds of values to be specified. When generating the indices, we would ideally like most of them to be valid, with only some invalid. This way we can test a lot of values having their adjoints being computed, with some invalid ones in-between to be ignored.

The bounds of all indices generated were therefore set to be in the range $[-1, k]$ where k is the length of the histogram, which differs between data-sets.
The resulting data-sets contain randomly generated lists of indices, values, values of the destination array and the adjoint of the resulting histogram.

Through these approaches each of the cases has been tested in the following manner:

1. **Addition:**
   Addition was tested using three data sets. The first one has 100 random values going into 5 bins, with the two latter having 1000 random values going into 15 bins.
   In order not to overflow in the case where many values needed to be added together, the values tested were bound to be floats in the range of $[-10000, 10000]$.

2. **Min/max:**
   Min and max implements the same solution, but for completeness they were tested individually. Both were tested with two separate data sets of 10000 values going into 50 bins. Because min/max does not include any values increasing/decreasing though additions or multiplication (initial adjoint of all values is set to zero for these tests), in any part of the code, no overflow should be possible. A bound on the values generated should therefore not be required, however due to a bug in the forward mode it was. For very large values (over 10 digits) forward mode would select the wrong value, but the reverse mode would actually select the correct one. The values were bound to be integers in the space of $[-10000, 10000]$.

3. **Multiplication:**
   Multiplication was tested by three sets of data. The first consists of 100 values going into 10 bins and the latter two having 1000 values going into 15 bins each.
   This was the case that originally showed that bounds on the generated values were necessary. When you multiply a lot of numbers, you tend to go towards zero or infinity. When the product goes towards zero, so does the adjoints, in which case the test serves little proof. In the case of infinity, the adjoints overflow and become useless.
   The output of the forward mode was manually checked for different bounds of values until a suitable one was found. The resulting bound was floats in the range $[-5, 5]$.

4. **General case:**
   The general case was simply tested by commenting out the special cases in the code, forcing any operator to enter the general case. All the previous tests were then run once more.

These tests show that for each operator its corresponding special case, the forward mode and the general approach all compute the same result on randomly generated inputs. All three methods have different ways of computing the adjoint, further decreasing the odds of the same bug being present in all three methods. The same can be said for the other operators, proving a strong case for the validity of the implementations.

The tests performed can be found in the appendix at section 7.4.

5.2 GPU benchmarks

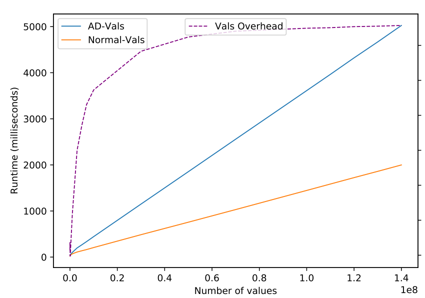
`Reduce_by_index` is parameterized by two factors: the amount of values to insert, and the size of the histogram we are inserting the values in.
In order to get a better sense of how each methods run-time scales with regards to each of these

parameters, we benchmark the impact each of them have individually.

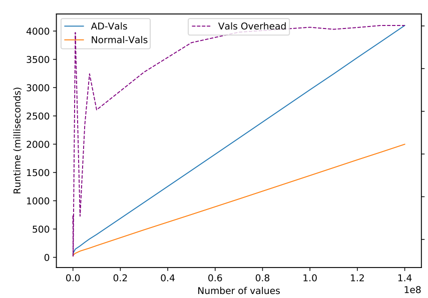
Furthermore, in order to find the overhead generated by application of reverse mode, each test is performed for both the reverse mode application and the equivalent `reduce_by_index`-statement without application of reverse mode. The overhead of reverse mode can then be computed as the factor of difference in run-time by computing AD overhead $= \frac{\text{AD run-time}}{\text{Normal run-time}}$.

We start by testing for a variable amount of input values. Testing of the values impact on run-time was performed with a static histogram-size of 10. The amount of values were in the range of $[1 \cdot 10^2, 1.4 \cdot 10^8]$ for the special cases, but smaller for the general case.
Due to the filtering performed by the general case at the start of its forward sweep, testing with random variables proved to be an issue. When using `Futharks` benchmark-tool, one cannot specify the bounds of the randomly generated values used as input. This caused the tool to generate out-of-bounds indices for almost all values. The result was that only the overhead of the filtering was measured.To combat this, randomly generated data sets were needed since they allow bounds. The data sets quickly grow in size, so only input-sizes in the interval of $[1.0 \cdot 10^2, 3.0 \cdot 10^6]$ were tested for the general case.

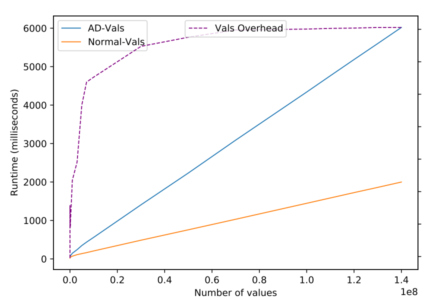
The graphs displaying the benchmarks use two y-scales. The left-hand side are used by the solid lines to measure run-time, and the right-hand side is used by the dotted line to measure the development of the AD overhead.



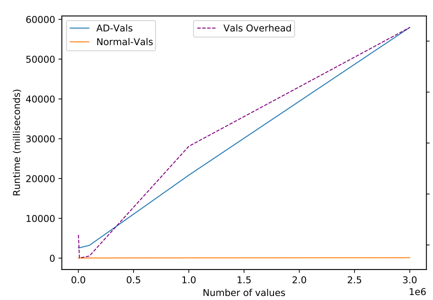
(a) Special case: Addition



(b) Special case: Min/max



(c) Special case: Multiplication



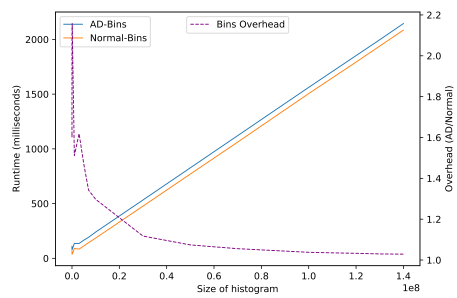
(d) General approach

Figure 38: Benchmarks using variable amount of input values

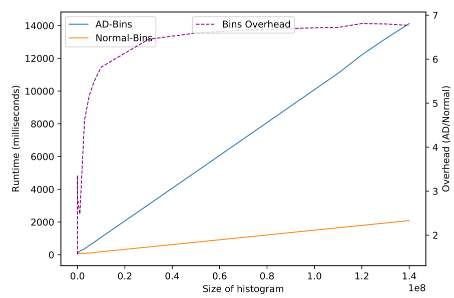
The results show that all cases scale linearly with respect to the amount of inputs, but also that the overhead caused by the general case scales linearly as well.
By looking at the profiling of the runtime, the large overhead of the general case can be attributed to the required sorting. The usage of Radix sort limited this overhead to be linear, but is still very significant.

For benchmarks of a variable histogram size, a static amount of $1.0 \cdot 10^6$ values were inserted. The variable size of histogram was in the range $[1.0 \cdot 10^2, 1.4 \cdot 10^8]$ for the special cases and $[1 \cdot 10^2, 1 \cdot 10^6]$ for the general case.

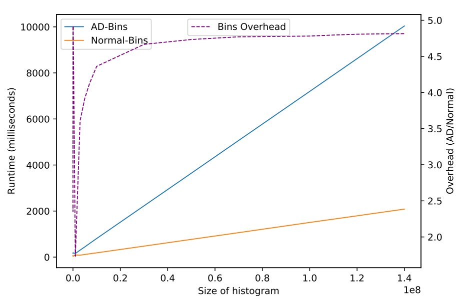
The layout used is the same as figure 38 above.



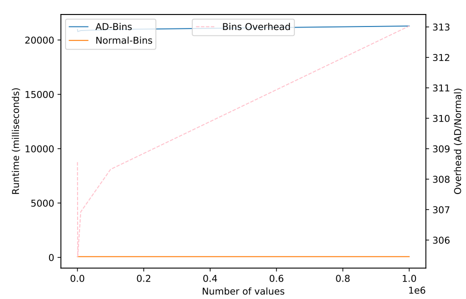
(a) Special case: Addition



(b) Special case: Min/max



(c) Special case: Multiplication



(d) General approach

Figure 39: Benchmarks using variable histogram size

The results once again show that the application of reverse mode AD scales linearly with the size of the histogram. Given that they also scale linearly in the amount of values, we can deduct that reverse mode AD scales linearly for all sizes of both inputs.

Once again we see a big overhead by using the general case, but does not vary as much as it did for the size of the input.
This is another symptom of sorting being the biggest factor of run-time. For this test we used a static amount of values and the sorting of those are such a large factor, that the runtime becomes completely dominated by it.

We now move on to measuring the overheads of applying reverse mode AD to each of the different operators.

While the AD overheads are variable, their slopes flatten at large input sizes. Since the run-time a low input sizes are short, the AD overhead at large sizes are the most interesting. The flattening of their curves indicate that they become constants at these points. This is obviously not the case for the general case, but might flatten if larger data-sets could be handled.

For the special case of addition we see that the size of the histogram has little effect on the overhead, and should be expected. The forward sweep performs a regular `reduce_by_index` and the reverse sweep performs only a map on the values. Given that the overhead caused by histogram size trends towards 1, we pick the one measured by input size.

The special cases of both min/max and multiplication are affected by both the size of the histogram and the input. For these we therefore multiply the overheads measured in both figures together.

For the general case we do not see a flattening. Given that both benchmarks proved the sorting of values to be the dominant factor in its run-time, we pick the highest overhead measured for variable input-size and define that as a lower bound.

By these methods the AD overhead of each case measures to be:

- Addition: 2.5 times slower.
- Min/max: 13.6 times slower.
- Multiplication: 14.1 times slower.
- General approach: > 500 times slower.

6 Conclusion and future work

This project presented a fully developed rewrite rule for reverse AD of the `reduce_by_index` operator of `Futhark`. The rule was then implemented by four different cases, one of them working for any operator using singleton values as input and validated by comparing results with that of the forward mode.
The methods implemented failed to accommodate all legal instances of histogram-computation, but did prove that the rewrite rule works in practice.

The special cases did require more work than if only the general case was implemented, but the benchmarks presented in section 5.2 illustrated that the optimized special cases are warranted by being much more efficient.

The most pressing future work on this topic would be to implement the missing features outlined in section 4.2.5 and accommodate all instances of histograms.

There are also unexplored options regarding optimization of the general approach.
The filtering performed at the start of the forward sweep is not strictly necessary and was done to make the following steps more easy to implement. Filtering could be removed by implementing Radix sort to handle signed integers, but the results may vary.
This all boils down to a design decision. If many bins are invalid the filter can potentially save a lot of time by reducing the number of elements. However, if no bins are invalid, the filtering step effectively does nothing else by waste time.

If filtering is kept, then for a histogram of size k, any key h for which $k \leq h \leq 0$ does not hold will be filtered out. This allows for an easy optimization of radix sort by bounding the amount of loops to $\lceil log_2(k) \rceil$.

7 Appendix

7.1 General case pseudo code

```
-- hist = reduce_by_index hist_orig op ne is vs
-- input:
--    hist_orig   :[w]t
--    inds        :[n]i64
--    vs          :[n]t
--    ne          : t
--    op          : t -> t -> t
--    w           is size of output
--    n           is size of input

flags = map (\ind -> if 0 <= ind <= histDim then 1 else 0) inds
flag_scanned = scan (+) 0 flags
n' = last flags_scanned
new_inds = map (\(flag, flag_scan) -> if flag == 1 then flag_scan - 1 else -1) flags flag_scanned
new_indexes = scatter (Scratch int n') new_inds (iota n)
new_bins = map (\i -> inds[i]) new_indexes

-- 63 should be replaced with log2ceiling(hist_dim) (number of bins)
[sorted_is, sorted_bins] =
  loop over [new_indexes, new_bins] for i < 63 do
    bits = map (\ind_x -> (ind_x >> i) & 1) new_bins
    newidx = partition2 bits (iota n')
    [map(\i -> new_indexes[i]) newidx, map(\i -> new_bins[i]) newidx]

sorted_vals = map(\i -> vs[i]) sorted_is

final_flags =
  map (\(index) ->
      if index == 0 then 1
      else
        if sorted_bins[index] == sorted_bins[index-1]
        then 0
        else 1
    ) (iota n')

fwd_scan = sgmScanExc op sorted_vals final_flags
[_, lis] = fwd_scan

rev_vals = reverse sorted_vals
rev_final_flags = reverse final_flags

rev_flags = map (\ind -> if ind == 0 then 1 else rev_final_flags[ind-1]) (iota n')


rev_scan = sgmScanExc op rev_vals rev_flags
[_, ris] = rev_scan

seg_end_idx = map (\i -> if i == n'-1 then i
                         else if final_flags[i+1] == 1
                               then i
                               else -1
                  ) (iota n')

bin_last_lis   = scatter (replicate ne (len hist_orig)) seg_end_idx lis
bin_last_v     = scatter (replicate ne (len hist_orig)) seg_end_idx sorted_vals

hist_temp = map2 op bin_lst_lis bin_lst_v_idx
hist = map2 op hist_orig hist_temp

m
-- Reverse sweep
hist_temp_bar: [w]t = lookupAdjVal y

hist_temp_bar_repl : [n]t = scatter (replicate ne (len hist_orig)) is hist_temp_bar
hist_temp_bar_repl = map (\ ind -> hist_tmp_bar[ind]) sorted_bins

lam_adj = (\li vi ri -> li op vi op ri) -- make this with mkF

vs_bar_reordered <- vjpMap ops [adjVal hist_temp_bar_repl] w lam_adj [lis, sorted_vals, ris]

vs_contrib = back_permute vs_bar_reordered
```

7.2 Compiler- and produced code

7.2.1 Special case: Addition

```haskell
diffHist :: VjpOps -> Pat Type -> StmAux() -> SOAC SOACS -> ADM () ->
    ADM ()
diffHist _vjops pat aux soac m
  | (Hist n [inds, vs] hist_add bucket_fun) <- soac,
    [HistOp shape rf [orig_dst] [ne] add_lam] <- hist_add,
    Just _ <- isAddTowLam add_lam,
    [pe] <- patNames pat = do
    -- need to create a copy of the orig histo, because the reverse
      trace might need
    -- the values of the original histogram input!
    dst_cpy <- letExp (baseString orig_dst ++ "_copy") $ BasicOp $ Copy
      orig_dst
    let histo' = Hist n [inds, vs] [HistOp shape rf [dst_cpy] [ne]
      add_lam] bucket_fun
    addStm $ Let pat aux $ Op histo'
    m
    -- Reverse trace
    let eltp = head $ lambdaReturnType add_lam
    pe_bar <- lookupAdjVal $ pe
    -- already update orig_dst bar
    void $ updateAdj orig_dst pe_bar
    -- update the vs bar; create a map nest with the branch innermost so
       all
    -- parallelism can be exploited.
    pind <- newParam "index" $ Prim int64
    map_bar_lam_bdy <- genIdxLamBdy pe_bar [(n, pind)] eltp
    let map_bar_lam = Lambda [pind] map_bar_lam_bdy [eltp]
    vs_bar <- letExp (baseString vs ++ "_bar") $ Op $ Screma n [inds] (
      mapSOAC map_bar_lam)
    void $ updateAdj vs vs_bar
```

(a) Compiler implementation

```
entry("main",
      {is: direct, vs: direct, hist: *direct,
      hist_bar: direct},
      {direct, direct})
  entry_main (n_5094 : i64, w_5095 : i64, is_5096 :
      [n_5094]i64,
              vs_5097 : [n_5094]f32, hist_5098 : *[
      w_5095]f32,
              hist_bar_5099 : [w_5095]f32)
  : {[n_5094]f32, [w_5095]f32} = {
  let {x_bar_5100 : [n_5094]f32} =
    map(n_5094,
        {is_5096},
        \ {index_5101 : i64}
          : {f32} ->
          let {binop_x_5102 : bool} = slt64(
      index_5101, n_5094)
          let {binop_y_5103 : bool} = slt64(-1i64,
      index_5101)
          let {cond_5104 : bool} = logand(
      binop_x_5102, binop_y_5103)
          let {x_5105 : f32} =
            if cond_5104
            then {
              let {res_5106 : f32} =
                hist_bar_5099[index_5101]
              in {res_5106}
            } else {0.0f32}
            : {f32}
          in {x_5105})
  in {x_bar_5100, hist_bar_5099}
}
```

(b) Generated code

Figure 40: Compiler implementation and produced code for special case (+)

7.2.2 Special case: Min/max

```
[eltp] <- lambdaReturnType max_lam,
p <- patElemName pe,
Prim ptp <- eltp,
[shapedim] <- shapeDims shape = do

orig_dst_cpy <- letExp (baseString orig_dst ++ "_cpy") $ BasicOp
       $ Copy orig_dst
f' <- mkIdentityLambda [Prim int64, eltp, Prim int64]
repl <- letExp "minus_ones" $ BasicOp $ Replicate shape (intConst
        Int64 (-1))
iota_n <- letExp "iota_n" $ BasicOp $ Iota n (intConst Int64 0) (
       intConst Int64 1) Int64
maxind_lam <- mkMinMaxIndLam ptp bop

let hist_op = HistOp shape rf [orig_dst_cpy, repl] [ne, intConst
       Int64 (-1)] maxind_lam

hist_inds <- newVName "hist_inds"
let histo_pat = Pat [pe, PatElem hist_inds (mkI64ArrType shape)]
auxing aux $ letBind histo_pat $ Op $ Hist n [inds, vs, iota_n] [
       hist_op] f'
m
```

(a) Compiler implementation

```
let {minus_ones_5917 : [w_5899]i64} =
    replicate([w_5899], -1i64)
let {iota_n_5918 : [n_5898]i64} =
    iota64(n_5898, 0i64, 1i64)
let {defunc_4_reduce_by_index_res_5919 : [w_5899]i32,
   hist_inds_5920 : [w_5899]i64} =
    hist(n_5898,
         {is_5900, vs_5901, iota_n_5918},
         {[w_5899], 1i64, {defunc_1_map_res_5913,
      minus_ones_5917},
          {0i32, -1i64},
          \ {acc_v_5921 : i32, acc_ind_5922 : i64,
      arg_v_5923 : i32,
             arg_ind_5924 : i64}
            : {i32,
               i64} ->
            let {cond_5925 : bool} = eq_i32(acc_v_5921,
      arg_v_5923)
            let {x_5926 : i32,
                 x_5927 : i64} =
              if cond_5925
              then {
                let {minmax_5928 : i64} = smin64(
      acc_ind_5922, arg_ind_5924)
                in {acc_v_5921, minmax_5928}
              } else {
                let {cmpop_y_5929 : i32} = umax32(
      acc_v_5921, arg_v_5923)
                let {cond_5930 : bool} = eq_i32(acc_v_5921,
       cmpop_y_5929)
                let {x_5931 : i32} =
                  if cond_5930
                  then {acc_v_5921} else {arg_v_5923}
                  : {i32}
                let {x_5932 : i64} =
                  if cond_5930
                  then {acc_ind_5922} else {arg_ind_5924}
                  : {i64}
                in {x_5931, x_5932}
              }
              : {i32, i64}
            in {x_5926, x_5927}},
         \ {x_5933 : i64, x_5934 : i32, x_5935 : i64}
           : {i64,
              i32,
              i64} ->
           {x_5933, x_5934, x_5935})
```

(b) Generated code

Figure 41: Compiler implementation and produced code for the forward sweep of special case (min/max)

```
pe_bar <- lookupAdjVal p
-- create the bar of ‘orig_dst‘ by means of a map:
pis_h <- zipWithM newParam ["min_ind", "h_elem"] [Prim int64,
      eltp]
let [min_ind_h, h_elem_h] = map paramName pis_h
lam_bdy_hist_bar <-
  runBodyBuilder . localScope (scopeOfLParams pis_h) $
    eBody
      [ eIf
          (toExp $ mind_eq_min1 min_ind_h)
          (resultBodyM [Var h_elem_h])
          (resultBodyM [Constant $ blankPrimValue ptp])
      ]
let lam_hist_bar = Lambda pis_h lam_bdy_hist_bar [eltp]
hist_bar <-
  letExp (baseString orig_dst ++ "_bar") $
    Op $
      Screma shapedim [hist_inds, pe_bar] (ScremaForm [] []
      lam_hist_bar)
insAdj orig_dst hist_bar
-- update vs_bar with a map and a scatter
vs_bar <- lookupAdjVal vs
pis_v <- zipWithM newParam ["min_ind", "h_elem"] [Prim int64,
      eltp]
let [min_ind_v, h_elem_v] = map paramName pis_v
lam_bdy_vs_bar <-
  runBodyBuilder . localScope (scopeOfLParams pis_v) $
    eBody
      [ eIf
          (toExp $ mind_eq_min1 min_ind_v)
          (resultBodyM [Constant $ blankPrimValue ptp])
          ( do
              vs_bar_i <-
                letSubExp (baseString vs_bar ++ "_el") $
                  BasicOp $
                    Index vs_bar $ Slice $ [DimFix $ Var
      min_ind_v]
              let plus_op = getBinOpPlus ptp
              r <- letSubExp "r" $ BasicOp $ BinOp plus_op
      vs_bar_i $ Var h_elem_v
              resultBodyM [r]
          )
      ]
let lam_vs_bar = Lambda pis_v lam_bdy_vs_bar [eltp]
vs_bar_p <-
  letExp (baseString vs_bar ++ "_partial") $
    Op $
      Screma shapedim [hist_inds, pe_bar] (ScremaForm [] []
      lam_vs_bar)
f’’ <- mkIdentityLambda [Prim int64, eltp]
let scatter_soac = Scatter shapedim [hist_inds, vs_bar_p] f’’ [(
      Shape [n], 1, vs_bar)]
vs_bar’ <- letExp (baseString vs ++ "_bar") $ Op scatter_soac
insAdj vs vs_bar’
```

(a) Compiler implementation

```
let {updated_adj_partial_5946 : [w_5899]i32} =
map(w_5899,
    {hist_inds_5920, map_adjs_5937},
    \ {min_ind_5947 : i64, h_elem_5948 : i32}
      : {i32} ->
      let {cond_5949 : bool} = eq_i64(min_ind_5947, -1i64)
      let {x_5950 : i32} =
        if cond_5949
        then {0i32} else {
          let {updated_adj_el_5951 : i32} =
            updated_adj_5945[min_ind_5947]
          let {r_5952 : i32} = add_nw32(h_elem_5948,
      updated_adj_el_5951)
          in {r_5952}
        }
        : {i32}
      in {x_5950})
let {x_bar_5953 : [n_5898]i32} =
scatter(w_5899,
        {hist_inds_5920, updated_adj_partial_5946},
        \ {x_5954 : i64, x_5955 : i32}
          : {i64,
             i32} ->
          {x_5954, x_5955},
        ([n_5898], 1, updated_adj_5945))
let {x_adj_5956 : [w_5899]i32} =
map(w_5899,
    {hist_inds_5920, map_adjs_5937},
    \ {min_ind_5957 : i64, h_elem_5958 : i32}
      : {i32} ->
      let {cond_5960 : bool} = eq_i64(min_ind_5957, -1i64)
      let {x_5961 : i32} =
        if cond_5960
        then {h_elem_5958} else {0i32}
        : {i32}
      in {x_5961})
in {x_bar_5953, x_adj_5956}
}
```

(b) Generated code

Figure 42: Compiler implementation and produced code for the reverse sweep of special case (min/max)

7.2.3 Special case: Multiplication

```
[eltp] <- lambdaReturnType mul_lam,
Prim ptp <- eltp,
[shapedim] <- shapeDims shape = do
-- Forward sweep
let pe_tp = patElemDec pe
(map_lam, _) <- helperMulOp1 ptp mulop
vs_lift <- letTupExp "nzel_zrct" $ Op $ Screma n [vs] (ScremaForm
        [] [] map_lam)
let [nz_vs, one_zrs] = vs_lift
zr_counts0 <- letExp "zr_cts" $ BasicOp $ Replicate shape (
      intConst Int64 0)
nz_prods0 <- letExp "nz_prd" $ BasicOp $ Replicate shape ne
nz_prods <- newVName "non_zero_prod"
zr_counts <- newVName "zero_count"
lam_add <- mkLamAddI64
let hist_zrn = HistOp shape rf [zr_counts0] [intConst Int64 0]
      lam_add
let hist_nzp = HistOp shape rf [nz_prods0] [ne] mul_lam
f' <- mkIdentityLambda [Prim int64, Prim int64, eltp, Prim int64]
let soac_pat =
      Pat
        [ PatElem nz_prods pe_tp,
          PatElem zr_counts $
          arrayOf (Prim int64) shape NoUniqueness
        ]
let soac_exp = Op $ Hist n [inds, inds, nz_vs, one_zrs] [hist_nzp
      , hist_zrn] f'
auxing aux $ letBind soac_pat soac_exp
-- construct the histo result:
res_part <- newVName "res_part"
ps2 <- zipWithM newParam ["nz_pr", "zr_ct"] [eltp, Prim int64]
let [nz_prod, zr_count] = map paramName ps2
if_stms <- helperMulOp2 ptp nz_prod zr_count res_part
lam_bdy_2 <- runBodyBuilder . localScope (scopeOfLParams ps2) $
      do
  addStms if_stms
  resultBodyM [Var res_part]
h_part <-
  letExp "hist_part" $
    Op $
      Screma
        shapedim
        [nz_prods, zr_counts]
        (ScremaForm [] [] (Lambda ps2 lam_bdy_2 [eltp]))
ps3 <- zipWithM newParam ["h_orig", "h_part"] [eltp, eltp]
let [ph_orig, ph_part] = map paramName ps3
lam_pe_bdy <- runBodyBuilder . localScope (scopeOfLParams ps3) $
      do
  r <- letSubExp "res" $ BasicOp $ BinOp mulop (Var ph_orig) (Var
        ph_part)
  resultBodyM [r]
auxing aux $
  letBind (Pat [pe]) $
    Op $
      Screma
        shapedim
        [orig_dst, h_part]
        (ScremaForm [] [] (Lambda ps3 lam_pe_bdy [eltp]))
m
```

(a) Compiler implementation

```
entry("main",
  {is: direct, vs: direct, hist: *direct, hist_bar: direct
      },
  {direct, direct})
entry_main (n_5491 : i64, w_5492 : i64, is_5493 : [n_5491]
      i64,
          vs_5494 : [n_5491]f32, hist_5495 : *[w_5492]f32,
          hist_bar_5496 : [w_5492]f32)
: {[n_5491]f32, [w_5492]f32} = {
let {zr_cts_5497 : [w_5492]i64} =
    replicate([w_5492], 0i64)
let {nz_prd_5498 : [w_5492]f32} =
    replicate([w_5492], 1.0f32)
let {non_zero_prod_5499 : [w_5492]f32,
   zero_count_5500 : [w_5492]i64} =
    hist(n_5491,
         {vs_5494, is_5493},
         {[w_5492], 1i64, {nz_prd_5498},
          {1.0f32},
          \ {x_5501 : f32, x_5502 : f32}
            : {f32} ->
            let {defunc_1_f_res_5503 : f32} = fmul32(x_5501
      , x_5502)
            in {defunc_1_f_res_5503},
          [w_5492], 1i64, {zr_cts_5497},
          {0i64},
          \ {a_5504 : i64, b_5505 : i64}
            : {i64} ->
            let {r_5506 : i64} = add_nw64(a_5504, b_5505)
            in {r_5506}},
         \ {arg_5507 : f32, x_5508 : i64}
           : {i64,
              i64,
              f32,
              i64} ->
           let {cond_5509 : bool} = eq_f32(arg_5507, 0.0f32
      )
           let {x_5510 : f32} =
             if cond_5509
             then {1.0f32} else {arg_5507}
             : {f32}
           let {x_5511 : i64} = btoi bool cond_5509 to i64
           in {x_5508, x_5508, x_5510, x_5511})
```

(b) Generated code

Figure 43: Compiler implementation and produced code for the forward sweep of special case (*)

```
-- reverse trace
pe_bar <- lookupAdjVal $ patElemName pe
-- updates the orig_dst with its proper bar
mul_lam' <- renameLambda mul_lam
orig_bar <-
  letTupExp (baseString orig_dst ++ "_bar") $
    Op $
      Screma
        shapedim
        [h_part, pe_bar]
        (ScremaForm [] [] mul_lam')
zipWithM_ updateAdj [orig_dst] orig_bar
-- updates the partial histo result with its proper bar
mul_lam'' <- renameLambda mul_lam
part_bars <-
  letTupExp (baseString h_part ++ "_bar") $
    Op $
      Screma
        shapedim
        [orig_dst, pe_bar]
        (ScremaForm [] [] mul_lam'')
let [part_bar] = part_bars
-- add the contributions to each array element
pj <- newParam "j" (Prim int64)
pv <- newParam "v" eltp
let j = paramName pj
((zr_cts, pr_bar, nz_prd), tmp_stms) <- runBuilderT' . localScope
       (scopeOfLParams [pj, pv]) $ do
  zr_cts <- letExp "zr_cts" $ BasicOp $ Index zr_counts $
      fullSlice eltp [DimFix (Var j)]
  pr_bar <- letExp "pr_bar" $ BasicOp $ Index part_bar $
      fullSlice eltp [DimFix (Var j)]
  nz_prd <- letExp "nz_prd" $ BasicOp $ Index nz_prods $ Slice [
      DimFix (Var j)]
  return (zr_cts, pr_bar, nz_prd)
bdy_tmp <- helperMulOp3 ptp mulop nz_prd zr_cts pv pr_bar
lam_bar <-
  runBodyBuilder . localScope (scopeOfLParams [pj, pv]) $
    eBody
      [ eIf
          (toExp $ withinBounds [(shapedim, j)])
          ( do
              addStms (tmp_stms <> bodyStms bdy_tmp)
              resultBodyM (map resSubExp $ bodyResult bdy_tmp)
          )
          (resultBodyM [Constant $ blankPrimValue ptp])
      ]
vs_bar <-
  letTupExp (baseString vs ++ "_bar") $
    Op $
      Screma
        n
        [inds, vs]
        (ScremaForm [] [] (Lambda [pj, pv] lam_bar [eltp]))
zipWithM_ updateAdj [vs] vs_bar
```

(a) Compiler implementation

```
let {hist_part_bar_5512 : [w_5492]f32} =
map(w_5492,
    {hist_5495, hist_bar_5496},
    \ {x_5513 : f32, x_5514 : f32}
      : {f32} ->
      let {defunc_1_f_res_5515 : f32} = fmul32(x_5513,
      x_5513)
      let {defunc_1_f_res_5516 : f32} = fmul32(x_5514,
      defunc_1_f_res_5515)
      in {defunc_1_f_res_5516})
let {x_bar_5517 : [n_5491]f32} =
map(n_5491,
    {is_5493, vs_5494},
    \ {j_5518 : i64, v_5519 : f32}
      : {f32} ->
      let {binop_x_5520 : bool} = slt64(j_5518, w_5492)
      let {binop_y_5521 : bool} = slt64(-1i64, j_5518)
      let {cond_5522 : bool} = logand(binop_x_5520,
      binop_y_5521)
      let {x_5523 : f32} =
        if cond_5522
        then {
          let {zr_cts_5524 : i64} =
            zero_count_5500[j_5518]
          let {pr_bar_5525 : f32} =
            hist_part_bar_5512[j_5518]
          let {nz_prd_5526 : f32} =
            non_zero_prod_5499[j_5518]
          let {cond_5527 : bool} = slt64(zr_cts_5524, 1i64)
          let {x_5528 : f32} =
            if cond_5527
            then {
              let {div_res_5529 : f32} = fdiv32(nz_prd_5526
      , v_5519)
              let {res_ctrb_5530 : f32} = fmul32(
      pr_bar_5525, div_res_5529)
              in {res_ctrb_5530}
            } else {
              let {binop_x_5531 : bool} = eq_f32(v_5519,
      0.0f32)
              let {binop_y_5532 : bool} = eq_i64(
      zr_cts_5524, 1i64)
              let {cond_5533 : bool} = logand(binop_x_5531,
       binop_y_5532)
              let {x_5534 : f32} =
                if cond_5533
                then {
                  let {res_ctrb_5535 : f32} =
                    fmul32(pr_bar_5525, nz_prd_5526)
                  in {res_ctrb_5535}
                } else {0.0f32}
                : {f32}
              in {x_5534}
            }
            : {f32}
          in {x_5528}
        } else {0.0f32}
        : {f32}
      in {x_5523})
let {x_adj_5536 : [w_5492]f32} =
map(w_5492,
    {non_zero_prod_5499, zero_count_5500, hist_bar_5496,
      hist_5495},
    \ {nz_pr_5537 : f32, zr_ct_5538 : i64, x_5539 : f32,
      x_5540 : f32}
      : {f32} ->
      let {cond_5541 : bool} = slt64(0i64, zr_ct_5538)
      let {tmp_if_res_5542 : f32} =
        if cond_5541
        then {0.0f32} else {nz_pr_5537}
        : {f32}
      let {defunc_1_f_res_5543 : f32} = fmul32(x_5539,
      tmp_if_res_5542)
      let {binop_x_adj_5544 : f32} = fmul32(x_5540,
      defunc_1_f_res_5543)
      let {binlam_res_5545 : f32} =
        fadd32(binop_x_adj_5544, binop_x_adj_5544)
      in {binlam_res_5545})
in {x_bar_5517, x_adj_5536}
}
```

(b) Generated code

Figure 44: Compiler implementation and produced code for the reverse sweep of special case (*)

7.2.4 General approach

```
-- flags = map (\ind -> if 0 <= ind <= histDim then 1 else 0 inds
ind_param <- newParam "ind" $ Prim int64
pred_body <- runBodyBuilder . localScope (scopeOfLParams [ind_param]) $
    eBody
      [ eIf -- if ind > 0 then 0 else ...
        (eCmpOp (CmpSlt Int64) (eParam ind_param) (eSubExp int64Zero) )
        (eBody [eSubExp $ int64Zero])
        (eBody
          [
            eIf -- if histDim > ind then 0 else 1
            (eCmpOp (CmpSlt Int64) (eSubExp histDim) (eParam ind_param)
      )
            (eBody [eSubExp $ int64Zero])
            (eBody [eSubExp $ int64One])
          ])

      ]
let pred_lambda = Lambda [ind_param] pred_body [Prim int64]
flags <- letExp "flags" $ Op $ Screma n [inds] $ ScremaForm [][]
      pred_lambda
-- flag_scanned = scan (+) 0 flags
add_lambda_i64 <- addLambda (Prim int64)
scan_soac <- scanSOAC [Scan add_lambda_i64 [intConst Int64 0]]
flags_scanned <- letExp "flag_scanned" $ Op $ Screma n [flags] scan_soac
```

(a) Compiler code

```
let {flag_scanned_6399 : [n_6392]i64,
   flags_6400 : [n_6392]i64} =
scanomap(n_6392,
         {is_6394},
         {\ {x_6401 : i64, y_6402 : i64}
            : {i64} ->
            let {binlam_res_6403 : i64} = add64(
      x_6401, y_6402)
            in {binlam_res_6403},
          {0i64}},
         \ {ind_6404 : i64}
           : {i64,
              i64} ->
           let {cond_6405 : bool} = slt64(ind_6404,
      0i64)
           let {x_6406 : i64} =
             if cond_6405
             then {0i64} else {
               let {cond_6407 : bool} = slt64(w_6393
      , ind_6404)
               let {cond_neg_6408 : bool} = not
      cond_6407
               let {x_6409 : i64} = btoi bool
      cond_neg_6408 to i64
               in {x_6409}
             }
             : {i64}
           in {x_6406, x_6406})
```

(b) Produced code

Figure 45: Code implementing the first two statements of the filter

```
i2 <- newVName "i2"
indexesForLoop <- newVName "new_indexes_rebound"
new_indexes_cpy <- letExp (baseString new_indexes ++ "_copyLoop") $
      BasicOp $ Copy new_indexes
new_indexes_type <- lookupType new_indexes
let isDeclTypeInds = toDecl new_indexes_type Unique
let paramIndexes = Param mempty indexesForLoop isDeclTypeInds

binsForLoop <- newVName "new_bins_rebound"
new_bins_cpy <- letExp (baseString new_bins ++ "_copyLoop") $ BasicOp $
      Copy new_bins
new_bins_type <- lookupType new_bins
let isDeclTypeBins = toDecl new_bins_type Unique
let paramBins = Param mempty binsForLoop isDeclTypeBins

let loop_vars = [(paramIndexes, Var new_indexes_cpy),(paramBins, Var
      new_bins_cpy)]

-- bound = log2ceiling(w) (inner hist size aka number of bins)
let bound = Constant $ IntValue $ intValue Int64 (64::Integer)

((idxres, binsres), stms) <- runBuilderT' . localScope (scopeOfFParams [
      paramIndexes, paramBins]) $ do
-- bits = map (\ind_x -> (ind_x >> digit_n) & 1) ind
ind_x <- newParam "ind_x" $ Prim int64
bits_map_bdy <- runBodyBuilder . localScope (scopeOfLParams [ind_x]) $
  eBody
  [
    eBinOp (And Int64)
      (eBinOp (LShr Int64) (eParam ind_x) (eSubExp $ Var i2))
      (eSubExp $ int64One)
  ]
let bits_map_lam = Lambda [ind_x] bits_map_bdy [Prim int64]
bits <- letExp "bits" $ Op $ Screma n' [binsForLoop] (ScremaForm [] []
      bits_map_lam)

-- Partition iota to get the new indices to scatter bins and inds by
temp_iota <- letExp "temp_iota" $ BasicOp $ Iota n' int64Zero int64One
      Int64
scatter_soac <- partition2Maker n' bits temp_iota
partitionedidx <- letExp (baseString inds ++ "_scattered") $ Op $
      scatter_soac

inner_indx_idx <- newParam "inner_indexes_idx" $ Prim int64
inner_indx_bdy <- runBodyBuilder . localScope (scopeOfLParams [
      inner_indx_idx]) $ do
  tmp <- letSubExp "indexes_body" $ BasicOp $ Index (paramName
      paramIndexes) (fullSlice (Prim int64) [DimFix (Var (paramName
      inner_indx_idx))])
  resultBodyM [tmp]
let inner_indx_lambda = Lambda [inner_indx_idx] inner_indx_bdy [Prim
      int64]
inner_new_indexes <- letSubExp "new_indexes" $ Op $ Screma n' [
      partitionedidx] $ ScremaForm [][] inner_indx_lambda

inner_bins_idx <- newParam "inner_indexes_idx" $ Prim int64
inner_bins_bdy <- runBodyBuilder . localScope (scopeOfLParams [
      inner_bins_idx]) $ do
  tmp <- letSubExp "indexes_body" $ BasicOp $ Index (paramName paramBins
      ) (fullSlice (Prim int64) [DimFix (Var (paramName inner_bins_idx)
      )])
  resultBodyM [tmp]
let inner_bins_lambda = Lambda [inner_bins_idx] inner_bins_bdy [Prim
      int64]
inner_new_bins <- letSubExp "new_bins" $ Op $ Screma n' [partitionedidx]
       $ ScremaForm [][] inner_bins_lambda

return (inner_new_indexes, inner_new_bins)

loop_bdy <- mkBodyM stms [subExpRes idxres,subExpRes binsres]
loop_res <- letTupExp "sorted_is_bins" $ DoLoop loop_vars (ForLoop i2
      Int64 bound []) loop_bdy
let [sorted_is, sorted_bins] = loop_res
```

(a) Compiler code

```
let {sorted_is_bins_6435 : [new_length_6411]i64,
   sorted_is_bins_6436 : [new_length_6411]i64} =
loop {new_indexes_rebound_6438 : *[new_length_6411]
       i64,
      new_bins_rebound_6439 : *[new_length_6411]i64}
       = {new_indexes_6429,

          new_bins_6432}
for i2_6437:i32 < 64i32 do {
  let {i2_6440 : i64} = sext i32 i2_6437 to i64
  let {bits_6441 : [new_length_6411]i64} =
    map(new_length_6411,
        {new_bins_rebound_6439},
        \ {ind_x_6442 : i64}
          : {i64} ->
          let {x_6443 : i64} = lshr64(ind_x_6442,
      i2_6440)
          let {x_6444 : i64} = and64(1i64, x_6443)
          in {x_6444})
  let {ps0_6452 : [new_length_6411]i64,
       ps1_6453 : [new_length_6411]i64,
       ps0_offset_6454 : i64,
       flags_inv_6455 : [new_length_6411]i64} =
    screma(new_length_6411,
           {bits_6441},
           {\ {x_6456 : i64, y_6457 : i64}
              : {i64} ->
              let {binlam_res_6458 : i64} = add_nw64
      (x_6456, y_6457)
              in {binlam_res_6458},
            {0i64},
            \ {x_6459 : i64, y_6460 : i64}
              : {i64} ->
              let {binlam_res_6461 : i64} = add_nw64
      (x_6459, y_6460)
              in {binlam_res_6461},
            {0i64}},
           {commutative \ {x_6462 : i64, y_6463 :
      i64}
              : {i64} ->
              let {binlam_res_6464 : i64} = add_nw64
      (x_6462, y_6463)
              in {binlam_res_6464},
            {0i64}},
           \ {flag_6465 : i64}
             : {i64,
                i64,
                i64,
                i64} ->
             let {x_6466 : i64} = sub_nw64(1i64,
      flag_6465)
             in {x_6466, flag_6465, x_6466, x_6466})
  let {temp_iota_copy_6471 : [new_length_6411]i64} =
    iota64(new_length_6411, 0i64, 1i64)
  let {is_scattered_6472 : [new_length_6411]i64} =
    scatter(new_length_6411,
            {flags_inv_6455, ps0_6452, ps1_6453,
      bits_6441,
             temp_iota_6419},
            \ {x_6473 : i64, y_6474 : i64,
      ps1_val_6475 : i64,
               x_6476 : i64, x_6477 : i64}
              : {i64,
                 i64} ->
              let {binlam_res_6478 : i64} = mul_nw64
      (x_6473, y_6474)
              let {x_6479 : i64} =
                add_nw64(ps0_offset_6454,
      ps1_val_6475)
              let {binlam_res_6480 : i64} = mul_nw64
      (x_6476, x_6479)
              let {binlam_res_6481 : i64} =
                add_nw64(binlam_res_6478,
      binlam_res_6480)
              let {x_6482 : i64} = sub_nw64(
      binlam_res_6481, 1i64)
              in {x_6482, x_6477},
            ([new_length_6411], 1,
      temp_iota_copy_6471))
  let {new_bins_6483 : [new_length_6411]i64,
       new_indexes_6484 : [new_length_6411]i64} =
    map(new_length_6411,
        {is_scattered_6472},
        \ {inner_indexes_idx_6485 : i64}
          : {i64,
             i64} ->
          let {indexes_body_6486 : i64} =
            new_indexes_rebound_6438[
      inner_indexes_idx_6485]
          let {indexes_body_6487 : i64} =
            new_bins_rebound_6439[
      inner_indexes_idx_6485]
          in {indexes_body_6487, indexes_body_6486})
  in {new_indexes_6484, new_bins_6483}
}
```

(b) Produced code

Figure 46: Code implementing radix sort

```
seg_scan_exc <- mkSegScanExc f nes n' sorted_vals final_flags
fwd_scan <- letTupExp "fwd_scan" $ Op seg_scan_exc
let [_, lis] = fwd_scan

```

(a) Compiler code

```
let {fwd_scan_6491 : [new_length_6411]i8,
   fwd_scan_6492 : [new_length_6411]f32,
   final_flags_6493 : [new_length_6411]i8} =
scanomap(new_length_6411,
         {sorted_is_bins_6436, temp_iota_6419},
         {\ {f1_6494 : i8, v1_6495 : f32, f2_6496 : i8,
             v2_6497 : f32}
            : {i8,
               f32} ->
            let {f'_6498 : i8} = or8(f1_6494, f2_6496)
            let {f_check_6499 : bool} = eq_i8(f2_6496, 1i8)
            let {v_6500 : f32} =
              if f_check_6499
              then {v2_6497} else {
                let {defunc_1_f_res_6501 : f32} =
                  fmul32(v1_6495, v2_6497)
                in {defunc_1_f_res_6501}
              }
              : {f32}
            in {f'_6498, v_6500},
          {0i8, 1.0f32}},
         \ {bin_6502 : i64, iot_n'_6503 : i64}
           : {i8,
              f32,
              i8} ->
           let {idx_minus_one_6504 : i64} =
             sub_nw64(iot_n'_6503, 1i64)
           let {prev_elem_6505 : i64} =
             sorted_is_bins_6436[idx_minus_one_6504]
           let {cond_6506 : bool} = eq_i64(iot_n'_6503, 0i64)
           let {x_6507 : i8} =
             if cond_6506
             then {1i8} else {
               let {cond_6508 : bool} =
                 eq_i64(prev_elem_6505, bin_6502)
               let {cond_neg_6509 : bool} = not cond_6508
               let {x_6510 : i8} = btoi bool cond_neg_6509 to i8
               in {x_6510}
             }
             : {i8}
           let {prev_elem_6511 : f32} =
             sorted_vals_6488[idx_minus_one_6504]
           let {cond_6512 : bool} = eq_i8(x_6507, 1i8)
           let {x_6513 : f32} =
             if cond_6512
             then {1.0f32} else {prev_elem_6511}
             : {f32}
           in {x_6507, x_6513, x_6507})

```

(b) Produced code

Figure 47: Code implementing computation of forward scan

```
---- Reverse segmented exculsive scan. Reverse flags and vals.
-- rev_vals = reverse sorted_vals
rev_vals <- eReverse sorted_vals
-- final_flags_rev = reverse final_flags
final_flags_rev <- eReverse final_flags

-- Need to fix flags after reversing
-- rev_flags = map (\ind -> if ind == 0 then 1 else rev[ind-1])
i' <- newParam "i" $ Prim int64
rev_flags_body <- runBodyBuilder . localScope (scopeOfLParams [i']) $ do
idx_minus_one <- letSubExp "idx_minus_one" $ BasicOp $ BinOp (Sub Int64
      OverflowUndef) (Var $ paramName i') (intConst Int64 1)
prev_elem <- letSubExp "prev_elem" $ BasicOp $ Index final_flags_rev (
      fullSlice (Prim int64) [DimFix idx_minus_one])

let firstElem =
      eCmpOp
        (CmpEq $ IntType Int64)
        (eSubExp $ Var $ paramName i')
        (eSubExp $ intConst Int64 0)

eBody
  [
    eIf
      firstElem
      (resultBodyM [trueSE])
      (resultBodyM [prev_elem])
  ]
let rev_flags_lambda = Lambda [i'] rev_flags_body [Prim int8]
rev_flags <- letExp "rev_flags" $ Op $ Screma n' [iota_n'] $ ScremaForm
      [][] rev_flags_lambda

-- Run segmented scan on reversed arrays.
rev_seg_scan_exc <- mkSegScanExc f nes n' rev_vals rev_flags
rev_scan <- letTupExp "rev_scan" $ Op rev_seg_scan_exc
let [_, ris_rev] = rev_scan
ris <- eReverse ris_rev
```

(a) Compiler code

```
let {rev_scan_6514 : [new_length_6411]i8,
   rev_scan_6515 : [new_length_6411]f32} =
scanomap(new_length_6411,
         {temp_iota_6419},
         {\ {f1_6516 : i8, v1_6517 : f32, f2_6518 :
      i8,
             v2_6519 : f32}
           : {i8,
              f32} ->
           let {f'_6520 : i8} = or8(f1_6516,
      f2_6518)
           let {f_check_6521 : bool} = eq_i8(
      f2_6518, 1i8)
           let {v_6522 : f32} =
             if f_check_6521
             then {v2_6519} else {
               let {defunc_1_f_res_6523 : f32} =
                 fmul32(v1_6517, v2_6519)
               in {defunc_1_f_res_6523}
             }
             : {f32}
           in {f'_6520, v_6522},
          {0i8, 1.0f32}},
         \ {i_6524 : i64}
           : {i8,
              f32} ->
          let {idx_minus_one_6525 : i64} = sub_nw64
      (i_6524, 1i64)
          let {binop_y_6526 : i64} =
            mul_nw64(-1i64, idx_minus_one_6525)
          let {slice_6527 : i64} =
            add_nw64(rev_start_6420, binop_y_6526)
          let {prev_elem_6528 : i8} =
            final_flags_6493[slice_6527]
          let {cond_6529 : bool} = eq_i64(i_6524, 0
      i64)
          let {x_6530 : i8} =
            if cond_6529
            then {1i8} else {prev_elem_6528}
            : {i8}
          let {prev_elem_6531 : f32} =
            sorted_vals_6488[slice_6527]
          let {eq_x_z_6532 : bool} = eq_i8(1i8,
      prev_elem_6528)
          let {not_p_6533 : bool} = not cond_6529
          let {p_and_eq_x_y_6534 : bool} =
            logand(eq_x_z_6532, not_p_6533)
          let {cond_6535 : bool} =
            logor(cond_6529, p_and_eq_x_y_6534)
          let {x_6536 : f32} =
            if cond_6535
            then {1.0f32} else {prev_elem_6531}
            : {f32}
          in {x_6530, x_6536})
```

(b) Produced code

Figure 48: Code implementing computation of reverse scan

```
i'' <- newParam "i''" $ Prim int64
current_bin <- newParam "current_bin" $ Prim int64
scatter_arr_body <- runBodyBuilder . localScope (scopeOfLParams [i'',
      current_bin]) $ do
idx_plus_one <- letSubExp "idx_plus_one" $ BasicOp $ BinOp (Add Int64
      OverflowUndef) (Var $ paramName i'') (intConst Int64 1)
lastElemIdx <- letExp "lastElemIdx" $ BasicOp $ BinOp (Sub Int64
      OverflowUndef) (n') (intConst Int64 1)

let isLastElem =
      eCmpOp
        (CmpEq $ IntType Int64)
        (eSubExp $ Var $ paramName i'')
        (eSubExp $ Var $ lastElemIdx)

eBody
  [
    eIf
    isLastElem
    (resultBodyM $ [Var $ paramName current_bin])
    (eBody
      [
        eIf
          ( do
          next_elem <- letExp "next_elem" $ BasicOp $ Index final_flags
      (fullSlice (Prim int64) [DimFix idx_plus_one])
              (eCmpOp
                (CmpEq $ IntType Int8)
                (eSubExp $ Var next_elem)
                (eSubExp $ Constant $ IntValue $ Int8Value 1))
          )
        (resultBodyM [Var $ paramName current_bin])
        (resultBodyM [Constant $ IntValue $ Int64Value (-1)])
      ]
    )
  ]

let scatter_arr_lam = Lambda [i'', current_bin] scatter_arr_body [Prim
      int64]
scatter_arr <- letExp "scatter_arr" $ Op $ Screma n' [iota_n',
      sorted_bins] $ ScremaForm [][] scatter_arr_lam
--bin_lst_lis   = scatter (replicate (len hist_orig) ne) scatter_arr lis
bin_last_lis_dst <- letExp "bin_last_lis_dst" $ BasicOp $ Replicate
      shape (head nes)
f''' <- mkIdentityLambda [Prim int64 , t]
bin_last_lis <- letExp "bin_last_lis" $ Op $ Scatter n' [scatter_arr,
      lis] f''' [(shape, 1, bin_last_lis_dst)]

-- lis was exc-scan, so we need the last element aswell.
-- bin_last_v_dst = scatter (replicate (len hist_orig) ne) scatter_arr
      sorted_vals
bin_last_v_dst <- letExp "bin_last_v_dst" $ BasicOp $ Replicate shape (
      head nes)
f'''' <- mkIdentityLambda [Prim int64 , t]
bin_last_v <- letExp "bin_last_v" $ Op $ Scatter n' [scatter_arr,
      sorted_vals] f'''' [(shape, 1, bin_last_v_dst)]

```

(a) Compiler code

```
let {bin_last_v_6287 : [w_6130]f32,
     bin_last_lis_6288 : [w_6130]f32} =
scatter(new_length_6147,
        {sorted_is_bins_6162, fwd_scan_6229,
      rev_scan_rev_6265,
         sorted_is_bins_6161, temp_iota_6160,
      sorted_is_bins_6162,
         sorted_vals_6215, fwd_scan_6229},
        \ {sorted_bin_p_6289 : i64, x_6290 : f32,
      x_6291 : f32,
           x_6292 : i64, i''_6293 : i64,
      current_bin_6294 : i64,
           x_6295 : f32, x_6296 : f32}
          : {i64,
             i64,
             i64,
             f32,
             f32,
             f32} ->
          let {idx_plus_one_6297 : i64} = add_nw64(1
      i64, i''_6293)
          let {cond_6298 : bool} = eq_i64(i''_6293,
      rev_start_6227)
          let {x_6299 : i64} =
            if cond_6298
            then {current_bin_6294} else {
              let {next_elem_6300 : i8} =
                final_flags_6214[idx_plus_one_6297]
              let {cond_6301 : bool} = eq_i8(
      next_elem_6300, 1i8)
              let {x_6302 : i64} =
                if cond_6301
                then {current_bin_6294} else {-1i64}
                : {i64}
              in {x_6302}
            }
            : {i64}
```

(b) Produced code

Figure 49: Code for generation of bin_last_lis and bin_last_v

```
-- Lookup adjoint of histogram
hist_res_bar <- lookupAdjVal $ patElemName pe

-- Rename original function
hist_temp_lam <- renameLambda f
hist_orig_lam <- renameLambda f

-- Lift lambda to compute differential wrt. first or second element.
hist_orig_bar_temp_lambda <- mkScanAdjointLam vjops hist_orig_lam
      WrtFirst
hist_temp_bar_temp_lambda <- mkScanAdjointLam vjops hist_temp_lam
      WrtSecond

-- Lambda for multiplying hist_res_bar onto result of differentiation
let mulOp = getMulOp t
mul_hist_orig_res_adj <- mkSimpleLambda "orig_adj" "res_adj" mulOp t
mul_hist_temp_res_adj <- mkSimpleLambda "temp_adj" "res_adj" mulOp t

-- Compute adjoint of each bin of each histogram (original and added
      values).
hist_orig_bar_temp <- letExp "hist_orig_bar_temp" $ Op $ Screma histDim
      [orig_dst, hist_temp] $ ScremaForm [][] hist_orig_bar_temp_lambda
hist_orig_bar      <- letExp "hist_orig_bar" $ Op $ Screma histDim [
      hist_orig_bar_temp, hist_res_bar] $ ScremaForm [][]
      mul_hist_orig_res_adj

hist_temp_bar_temp <- letExp "hist_temp_bar_temp" $ Op $ Screma histDim
      [orig_dst, hist_temp] $ ScremaForm [][] hist_temp_bar_temp_lambda
hist_temp_bar      <- letExp "hist_temp_bar" $ Op $ Screma histDim [
      hist_temp_bar_temp, hist_res_bar] $ ScremaForm [][]
      mul_hist_temp_res_adj

-- Set adjoint of orig_dst for future use
void $ insAdj orig_dst hist_orig_bar
```

(a) Compiler code

```
let {hist_temp_bar_6280 : [w_6130]f32} =
map(w_6130,
    {hist_6133, hist_bar_6134},
    \ {x_6281 : f32, res_adj_6282 : f32}
      : {f32} ->
      let {defunc_1_f_res_6283 : f32} = fmul32(
      x_6281, x_6281)
      let {x_6284 : f32} = fmul32(res_adj_6282,
      defunc_1_f_res_6283)
      in {x_6284})
let {x_adj_6309 : [w_6130]f32} =
    map(w_6130,
        {bin_last_lis_6288, bin_last_v_6287,
      hist_bar_6134, hist_6133},
        \ {lis_param_6310 : f32, v_param_6311 : f32,
       res_adj_6312 : f32,
           x_6313 : f32}
          : {f32} ->
          let {defunc_1_f_res_6314 : f32} = fmul32(
      lis_param_6310, v_param_6311)
          let {x_6315 : f32} = fmul32(res_adj_6312,
      defunc_1_f_res_6314)
          let {binop_x_adj_6316 : f32} = fmul32(
      x_6313, x_6315)
          let {binlam_res_6317 : f32} =
            fadd32(binop_x_adj_6316,
      binop_x_adj_6316)
          in {binlam_res_6317})
```

(b) Produced code

Figure 50: Code for computing the adjoint of hist_temp and orig_dst

```
-- For each bin in sorted_bins we fetch the adjoint of the corresponding
       bucket in hist_temp_bar
-- hist_temp_bar_repl = map (\bin -> hist_temp_bar[bin]) sorted_bins
sorted_bin_param <- newParam "sorted_bin_p" $ Prim int64
hist_temp_bar_repl_body <- runBodyBuilder . localScope (scopeOfLParams [
      sorted_bin_param]) $ do
hist_temp_adj <- letSubExp "hist_temp_adj" $ BasicOp $ Index
      hist_temp_bar (fullSlice (Prim int64) [DimFix (Var (paramName
      sorted_bin_param))])
resultBodyM [hist_temp_adj]
let hist_temp_bar_repl_lambda = Lambda [sorted_bin_param]
      hist_temp_bar_repl_body [t]
hist_temp_bar_repl <- letExp "hist_temp_bar_repl" $ Op $ Screma n' [
      sorted_bins] $ ScremaForm [][] hist_temp_bar_repl_lambda

-- We now use vjpMap to compute vs_bar
(_, lam_adj) <- mkF f
vjpMap vjops [AdjVal $ Var hist_temp_bar_repl] n' lam_adj [lis,
      sorted_vals, ris] -- Doesn't support lists

-- We are only using values not sorted out. Need to add 0 for each value
       that was sorted out.
-- Get adjoints of values with valid bins (computed by running vjpMap
      before)
vs_bar_contrib_reordered <- lookupAdjVal sorted_vals
-- Replicate array of 0's
vs_bar_contrib_dst <- letExp "vs_bar_contrib_dst" $ BasicOp $ Replicate
      (Shape [n]) (getBaseAdj t)
-- Scatter adjoints to 0-array.
f'''''' <- mkIdentityLambda [Prim int64, t]
vs_bar_contrib <- letExp "vs_bar_contrib" $ Op $ Scatter n' [sorted_is,
      vs_bar_contrib_reordered] f'''''' [(Shape [n], 1,
      vs_bar_contrib_dst)]
-- Update the adjoint of vs to be vs_bar_contrib
void $ updateAdj vs vs_bar_contrib
```

(a) Compiler code

```
let {vs_bar_contrib_6286 : [n_6129]f32,
   bin_last_v_6287 : [w_6130]f32,
   bin_last_lis_6288 : [w_6130]f32} =
scatter(new_length_6147,
        {sorted_is_bins_6162, fwd_scan_6229,
      rev_scan_rev_6265,
         sorted_is_bins_6161, temp_iota_6160,
      sorted_is_bins_6162,
         sorted_vals_6215, fwd_scan_6229},
        \ {sorted_bin_p_6289 : i64, x_6290 : f32,
      x_6291 : f32,
           x_6292 : i64, i''_6293 : i64,
      current_bin_6294 : i64,
           x_6295 : f32, x_6296 : f32}
          : {i64,
             i64,
             i64,
             f32,
             f32,
             f32} ->
          let {idx_plus_one_6297 : i64} = add_nw64(1
      i64, i''_6293)
          let {cond_6298 : bool} = eq_i64(i''_6293,
      rev_start_6227)
          let {x_6299 : i64} =
            if cond_6298
            then {current_bin_6294} else {
              let {next_elem_6300 : i8} =
                final_flags_6214[idx_plus_one_6297]
              let {cond_6301 : bool} = eq_i8(
      next_elem_6300, 1i8)
              let {x_6302 : i64} =
                if cond_6301
                then {current_bin_6294} else {-1i64}
                : {i64}
              in {x_6302}
            }
            : {i64}
          let {hist_temp_adj_6303 : f32} =
            hist_temp_bar_6280[sorted_bin_p_6289]
          let {binop_x_adj_6304 : f32} = fmul32(
      x_6291, hist_temp_adj_6303)
          let {binop_y_adj_6305 : f32} = fmul32(
      x_6290, binop_x_adj_6304)
          in {x_6292, x_6299, x_6299,
      binop_y_adj_6305, x_6295, x_6296},
        ([n_6129], 1, vs_bar_contrib_dst_6285), ([
      w_6130], 1,

      bin_last_v_dst_6267),
        ([w_6130], 1, bin_last_lis_dst_6266))
```

(b) Produced code

Figure 51: Code for computing the adjoint of vs

7.3 Helper-functions

7.3.1 Partition2

```
-- Reorders a list of values according to a list of flags. Resulting list has 0's at the head and 1's as its tail.
def partition2 flags values =
    -- Inverse of flags.
    let flags_inv  = map (\f -> 1 - f) flags
    -- Scan flags_inv.
    let ps0        = scan (+) 0 (flags_inv)
    -- Multiply ps0 with flags_inv to remove elements where flag was 1.
    let ps0_clean  = map2 (*) flags_inv ps0
    -- Reduce inverted flags.
    let ps0_offset = reduce (+) 0 flags_inv
    -- Scan flags.
    let ps1        = scan (+) 0 flags
    -- Map offset of all the values where flag was 0.
    let ps1'  = map (+ ps0_offset) ps1
    -- Multiply ps1_clean with flags to remove elementes where flag was 0.
    let ps1_clean = map2 (*) flags ps1'
    -- Add the two list together. Because of cleaning we know for each index that one of them will be 0, hence why we can just add.
    let ps = map2 (+) ps0_clean ps1_clean
    -- Accumulation started at 1. Subtract 1 from all to get valid indexes
    let ps_actual = map (-1) ps
    -- Scatter values to new indices.
    in scatter values ps_actual values
```

```haskell
-- partion2Maker - Takes flag array and values and creates a scatter SOAC
--                  which corresponds to the partition2 of the inputs
-- partition2Maker size flags values =
partition2Maker :: SubExp -> VName -> VName -> BuilderT SOACS ADM (SOAC SOACS)
partition2Maker n flags xs = do

let bitType = int64
let zeroSubExp = Constant $ IntValue $ intValue Int64 (0 :: Integer)
let oneSubExp = Constant $ IntValue $ intValue Int64 (1 :: Integer)

-- let bits_inv = map (\b -> 1 - b) bits
flag <- newParam "flag" $ Prim bitType
bits_inv_map_bdy <- runBodyBuilder . localScope (scopeOfLParams [flag]) $ do
eBody
  [
    eBinOp (Sub Int64 OverflowUndef)
    (eSubExp $ oneSubExp)
    (eParam flag)
  ]
let bits_inv_map_lam = Lambda [flag] bits_inv_map_bdy [Prim bitType]
flags_inv <- letExp "flags_inv" $ Op $ Screma n [flags] (ScremaForm [] [] bits_inv_map_lam)

-- let ps0 = scan (+) 0 (flags_inv)
ps0_add_lam <- binOpLambda (Add Int64 OverflowUndef) bitType
let ps0_add_scan = Scan ps0_add_lam [zeroSubExp]
f' <- mkIdentityLambda [Prim bitType]
ps0 <- letExp "ps0" $ Op $ Screma n [flags_inv] (ScremaForm [ps0_add_scan] [] f')

-- let ps0_clean = map2 (*) flags_inv ps0
ps0clean_mul_lam <- binOpLambda (Mul Int64 OverflowUndef) bitType
ps0clean <- letExp "ps0_clean" $ Op $ Screma n [flags_inv, ps0] (ScremaForm [] [] ps0clean_mul_lam)

-- let ps0_offset = reduce (+) 0 flags_inv
ps0off_add_lam <- binOpLambda (Add Int64 OverflowUndef) bitType
ps0off_red <- reduceSOAC [Reduce Commutative ps0off_add_lam [intConst Int64 0]]
ps0off <- letExp "ps0_offset" $ Op $ Screma n [flags_inv] ps0off_red

-- let ps1 = scan (+) 0 flags
ps1_scanlam <- binOpLambda (Add Int64 OverflowUndef) bitType
let ps1_scan = Scan ps1_scanlam [zeroSubExp]
f'' <- mkIdentityLambda [Prim bitType]
ps1 <- letExp "ps1" $ Op $ Screma n [flags] (ScremaForm [ps1_scan] [] f'')

-- let ps1' = map (+ps0_offset) ps1
ps1_val <- newParam "ps1_val" $ Prim bitType
ps1clean_lam_bdy <- runBodyBuilder . localScope (scopeOfLParams [ps1_val]) $ do
eBody
  [
    eBinOp (Add Int64 OverflowUndef)
    (eParam ps1_val)
    (eSubExp $ Var ps0off)
  ]
let ps1clean_lam = Lambda [ps1_val] ps1clean_lam_bdy [Prim bitType]
ps1' <- letExp "ps1'" $ Op $ Screma n [ps1] (ScremaForm [] [] ps1clean_lam)

-- let ps1_clean = map2 (*) flags ps1'
ps1cleanprim_mul_lam <- binOpLambda (Mul Int64 OverflowUndef) bitType
ps1_clean <- letExp "ps1_clean" $ Op $ Screma n [flags, ps1'] (ScremaForm [] [] ps1cleanprim_mul_lam)

-- let ps = map2 (+) ps0_clean ps1_clean
ps_add_lam <- binOpLambda (Add Int64 OverflowUndef) bitType
ps <- letExp "ps" $ Op $ Screma n [ps0clean, ps1_clean] (ScremaForm [] [] ps_add_lam)


-- let ps_actual = map (-1) ps
psactual_x <- newParam "psactual_x" $ Prim bitType
psactual_lam_bdy <- runBodyBuilder . localScope (scopeOfLParams [psactual_x]) $ do
eBody
  [
    eBinOp (Sub Int64 OverflowUndef)
    (eParam psactual_x)
    (eSubExp $ oneSubExp)
  ]
let psactual_lam = Lambda [psactual_x] psactual_lam_bdy [Prim bitType]
psactual <- letExp "psactual" $ Op $ Screma n [ps] (ScremaForm [] [] psactual_lam)

-- let scatter_inds = scatter inds ps_actual inds
-- return scatter_inds
f''' <- mkIdentityLambda [Prim int64, Prim int64]
xs_cpy <- letExp (baseString xs ++ "_copy") $ BasicOp $ Copy xs
return $ Scatter n [psactual, xs] f''' [(Shape [n], 1, xs_cpy)]
```

Figure 52: Code implementing Partition2

7.3.2 mkSegScanExc

```haskell
-- tmp = map (\(i,f) -> if f then (ne, f) else (vals[i-1], f)) (iota n) (flags)
-- scan (\(v1,f1) (v2,f2) ->
--            let f = f1 || f2
--            let v = if f2 then v2 else op v1 v2
--            in (v, f)) tmp flags

-- Lift a lambda to produce an exlusive segmented scan operator.
mkSegScanExc :: Lambda SOACS -> [SubExp] -> SubExp -> VName -> VName -> ADM (SOAC SOACS)
mkSegScanExc lam ne n vals flags = do
  -- Get lambda return type
  let rt = lambdaReturnType lam
  -- v <- mapM (newParam "v") rt
  v1 <- mapM (newParam "v1") rt
  v2 <- mapM (newParam "v2") rt
  f <- newParam "f" $ Prim int8
  f1 <- newParam "f1" $ Prim int8
  f2 <- newParam "f2" $ Prim int8
  let params = (f1 : v1) ++ (f2 : v2)

  iota_n <- letExp "iota_n" $ BasicOp $ Iota n (intConst Int64 0) (intConst Int64 1) Int64
  i <- newParam "i" $ Prim int64

  -- (\(flag, i) -> if f then (f, ne) else (f, vals[i-1]))

  tmp_lam_body <- runBodyBuilder . localScope (scopeOfLParams [f, i]) $ do
    idx_minus_one <- letSubExp "idx_minus_one" $ BasicOp $ BinOp (Sub Int64 OverflowUndef) (Var $ paramName i) (intConst Int64 1)
    prev_elem <- letTupExp "prev_elem" $ BasicOp $ Index vals (fullSlice (Prim int64) [DimFix idx_minus_one])

    let f_check =
          eCmpOp
            (CmpEq $ IntType Int8)
            (eSubExp $ Var $ paramName f)
            (eSubExp $ intConst Int8 1)

    eBody
      [
        eIf
          f_check
          (resultBodyM $ (Var $ paramName f) : ne )
          (resultBodyM $ ((Var . paramName) f) : (map Var prev_elem))
      ]

  let tmp_lam = Lambda [f, i] tmp_lam_body (Prim int8 : rt)
  lam' <- renameLambda lam

  scan_body <- runBodyBuilder . localScope (scopeOfLParams params) $ do
    -- f = f1 || f2
    f' <- letSubExp "f'" $ BasicOp $ BinOp (Or Int8 ) (Var $ paramName f1) (Var $ paramName f2)
    -- v = if f2 then v2 else (lam v1 v2)
    op_body <- mkLambda (v1++v2) $ do
      eLambda lam' (map (eSubExp . Var . paramName) (v1++v2))

    v2_body <- eBody $ map (eSubExp . Var . paramName) v2

    f_check <- letExp "f_check" $ BasicOp $ CmpOp (CmpEq $ IntType Int8) (Var $ paramName f2) (intConst Int8 1)

    v <- letSubExp "v" $
          If (Var f_check)
          v2_body
          (lambdaBody op_body)
          (IfDec (staticShapes rt) IfNormal)

    -- Put together
    eBody $ map eSubExp ([f', v])

  let scan_lambda = Lambda params scan_body (Prim int8 : rt)

  return $ Screma n [flags, iota_n] $ ScremaForm [Scan scan_lambda ((intConst Int8 0) : ne)] [] tmp_lam
```

Figure 53: Implementation of mkSegScanExc

7.4 Validation tests

```
-- Validation of histogram with addition
-- ==
-- entry: rev fwd
-- compiled input @ histo-plus-data1.txt
-- output @ histo-plus-data1Res.txt
-- compiled input @ histo-plus-data2.txt
-- output @ histo-plus-data2Res.txt
-- compiled input @ histo-plus-data3.txt
-- output @ histo-plus-data3Res.txt

def singleadj (n: i64) (i: i64) (adj: f32) : [n]f32 =
  map (\j -> if (i==j) then adj else 0.0f32) (iota n)

let histo_plus [n][w](is: [n]i64) (dest: [w]f32) (vs: [n]f32) : [w]f32 =
  reduce_by_index (copy dest) (+) 0f32 is vs

entry rev [n][w](is: [n]i64) (vs: [n]f32) (hist_orig: [w]f32) (hist_bar': [w]f32) =
  map (\i -> vjp (histo_plus is hist_orig) vs (singleadj w i hist_bar'[i])) (iota w)


entry fwd [n][w](is: [n]i64) (vs: [n]f32) (hist_orig: [w]f32) (hist_bar': [w]f32) =
  map (jvp (histo_plus is (hist_orig: [w]f32)) vs)
    (map (\ i -> let adj = if is[i] < 0i64 then 0f32 else hist_bar'[is[i]]
                 in singleadj n i adj) (iota n))
  |> transpose
```

Figure 54: Testing reduce_by_index with operator (+) running on three data sets

```
-- Simple histogram with i32.min operator
-- ==
-- entry: rev fwd
-- compiled input @ histo-min-data1.txt
-- output @ histo-min-data1Res.txt
-- compiled input @ histo-min-data2.txt
-- output @ histo-min-data2Res.txt

def singleadj (n: i64) (i: i64) (adj: i32) : [n]i32 =
  map (\j -> if (i==j) then adj else 0i32) (iota n)

let histo_max [n][w](is: [n]i64) (dest: [w]i32) (vs: [n]i32) : [w]i32 =
  reduce_by_index (copy dest) (i32.min) i32.highest is vs

entry rev [n][w](is: [n]i64) (vs: [n]i32) (hist_orig: [w]i32) (hist_bar': [w]i32) =
  map (\i -> vjp (histo_max is hist_orig) vs (singleadj w i hist_bar'[i])) (iota w)


entry fwd [n][w](is: [n]i64) (vs: [n]i32) (hist_orig: [w]i32) (hist_bar': [w]i32) =
  map (jvp (histo_max is (hist_orig: [w]i32)) vs)
    (map (\ i -> let adj = if is[i] < 0i64 then 0i32 else hist_bar'[is[i]]
                 in singleadj n i adj) (iota n))
  |> transpose
```

Figure 55: Testing reduce_by_index with operator "min" running on three data sets

```
-- Simple histogram with u32.max operator
-- ==
-- entry: rev fwd
-- compiled input @ histo-max-data1.txt
-- output @ histo-max-data1Res.txt
-- compiled input @ histo-max-data2.txt
-- output @ histo-max-data2Res.txt

def singleadj (n: i64) (i: i64) (adj: u32) : [n]u32 =
  map (\j -> if (i==j) then adj else 0u32) (iota n)

let histo_max [n][w](is: [n]i64) (dest: [w]u32) (vs: [n]u32) : [w]u32 =
  reduce_by_index (copy dest) (u32.max) u32.lowest is vs

entry rev [n][w](is: [n]i64) (vs: [n]u32) (hist_orig: [w]u32) (hist_bar': [w]u32) =
  map (\i -> vjp (histo_max is hist_orig) vs (singleadj w i hist_bar'[i])) (iota w)


entry fwd [n][w](is: [n]i64) (vs: [n]u32) (hist_orig: [w]u32) (hist_bar': [w]u32) =
  map (jvp (histo_max is (hist_orig: [w]u32)) vs)
    (map (\ i -> let adj = if is[i] < 0i64 then 0u32 else hist_bar'[is[i]]
                 in singleadj n i adj) (iota n))
  |> transpose
```

Figure 56: Testing reduce_by_index with operator "max" running on three data sets

```
-- Validation of histogram with multiplication
-- ==
-- entry: rev fwd
-- compiled input @ histo-mul-data1.txt
-- output @ histo-mul-data1Res.txt
-- compiled input @ histo-mul-data2.txt
-- output @ histo-mul-data2Res.txt
-- compiled input @ histo-mul-data3.txt
-- output @ histo-mul-data3Res.txt

def singleadj (n: i64) (i: i64) (adj: f32) : [n]f32 =
  map (\j -> if (i==j) then adj else 0.0f32) (iota n)

let histo_mul [n][w](is: [n]i64) (dest: [w]f32) (vs: [n]f32) : [w]f32 =
  reduce_by_index (copy dest) (*) 1f32 is vs

entry rev [n][w](is: [n]i64) (vs: [n]f32) (hist_orig: [w]f32) (hist_bar': [w]f32) =
  map (\i -> vjp (histo_mul is hist_orig) vs (singleadj w i hist_bar'[i])) (iota w)


entry fwd [n][w](is: [n]i64) (vs: [n]f32) (hist_orig: [w]f32) (hist_bar': [w]f32) =
  map (jvp (histo_mul is (hist_orig: [w]f32)) vs)
    (map (\ i -> let adj = if is[i] < 0i64 then 0f32 else hist_bar'[is[i]]
                 in singleadj n i adj) (iota n))
  |> transpose

```

Figure 57: Testing reduce_by_index with operator (*) running on three data sets

References

[BPR15] Atilim Gunes Baydin, Barak A. Pearlmutter, and Alexey Andreyevich Radul. "Automatic differentiation in machine learning: a survey". In: *CoRR* abs/1502.05767 (2015). arXiv: `1502.05767`. URL: `http://arxiv.org/abs/1502.05767`.

[Hen17] Troels Henriksen. "Design and Implementation of the Futhark Programming Language". PhD thesis. Universitetsparken 5, 2100 København: University of Copenhagen, Nov. 2017.

[Hen+20] Troels Henriksen et al. "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, 2020. ISBN: 9781728199986.

[Sch+22] Robert Schenck et al. *AD for an Array Language with Nested Parallelism*. 2022. DOI: `10.48550/ARXIV.2202.10297`. URL: `https://arxiv.org/abs/2202.10297`.