Extending automatic differentiation for an array language with nested parallelism

Gilli Reynstind Fjallstein - gwn787

Master's thesis in Computer Science

Advisor: Cosmin E. Oancea

May 30, 2022

Abstract

This project explores the concept of automatic differentiation for functional languages. In particular, it gives mathematical reasoning for how a rewrite rule for the functional primitive *reduce_by_index* is derived. The project also derives even more efficient rewrite rules for the cases where this construct is given addition, multiplication, or min/max as its operator. These were implemented as compiler transformations for the data-parallel functional language Futhark. The implementations are argued to be valid, and benchmarks show that the general case is not as efficient as the forward mode equivalent. However, nearly all the special cases outperform both. This project argues that this shows how other constructs can be derived and how much is to gain from supporting even more special cases.

Contents

1 Introduction

Automatic differentiation (AD) is a common and efficient method of computing derivatives of functions expressed as programs. This type of differentiation especially useful when the programs are complicated and when the precision of finite difference methods are inadequate. (Leal 2018) Modern deep learning is based on the reverse mode AD implementation which excels when the number of inputs is vastly larger than the number of outputs of the program. Although in practice only a selected few parallel frameworks are used these include PyTorch (Paszke et al. 2019) and Tensorflow. (Developers 2022)

These common libraries do however not support AD on higher-order functions and that is where the novel approach taken in the data-parallel language Futhark come into play. (Henriksen, Serup, et al. 2017) This is a language which supports nested parallelism and has the performance of low-level languages while maintaining a high-level functional approach. Every program in Futhark is composed of a few functional constructs each of which need to have AD explicitly supported in the compiler for programs to be fully differentiated. (Schenck et al. 2022)

One of these functional construct unique to Futhark which is not supported yet by the reverse mode AD in the compiler is *reduce_by_index*. (Henriksen, Hellfritzsch, et al. 2020) This construct allows for an efficient method of histogram computations, like an N-ary *reduce* operator. The imperative equivalent of a call of *reduce_by_index* (RBI) can be seen in figure 1. It takes as input a destination histogram, operator *op*, index array *is* and value array *as*

```
1 m = length hist_orig
2 for i in 0 .. (length is) - 1:
3     ind = is[i]
4     v = as[i]
5     if ind >= 0 && ind < length as:
6         hist_orig[ind] = op hist_orig[ind] v
```

Figure 1: Imperative equivalent of RBI

This parallel construct is non-trivial to differentiate and is not supported by any other AD libraries since they do not have a representation of it. This project will derive a rewrite rule

for the construct and implement it in the Futhark compiler. Furthermore three special cases of the construct with much more efficient reverse mode AD methods will be introduced and implemented. These implementations will then be benchmarked and compared to eachother and the forward mode version.

In section 2 the preliminaries for reasoning about parallel algorithms, functional notation, and the Futhark language will be presented. Section 3 will then give a thorough introduction to automatic differentiation. This will lead into section 4 which reasons about the rewrite rule for *reduce* and how it is extended to a rewrite rule for RBI as well as present the special cases. Section 5 will present how the rewrite rule for RBI is converted to pseudocode, and introduce a simplified version of Futhark IR which will be used to show snippets of the implementation. Section 6 will give a short explanation of how the implementation was tested and verified to be correct. Section 7 will present the benchmarks reason about their results. Section 8 will give a short conclusion of the contributions of this project and what can be improved by future work.

2 Preliminaries

This section introduces some of the preliminaries needed for understanding the discussions and code presented later in the project. Specifically, the work and span model for reasoning about the run-time of parallel algorithms will be very roughly presented. After which, certain concepts from functional programming will be presented. Finally, the Futhark language, along with some of the essential constructs it contains, will be explained.

2.1 Work and span

This project involves reasoning about the performance of parallel algorithms. As such, the regular RAM model used to reason about the performance of sequential algorithms is inadequate. Therefore, this project will use the cost model based on Parallel Random Access Machines, PRAM. Specifically, the asymptotics of the work-span model will be used to analyse the parallel constructs in this project. (Blelloch 1996)

The work of T, presented as W(T), represents the total number of operations executed by the function T. This is very similar to the regular big-O notation for cost used for the RAM model. It can be generalised as the time the program would take if it ran on a single processor.

The span of T, presented as S(T), is defined as the maximum length of the series of sequential dependencies in the computation. This is what limits the computing time in an idealised machine with infinite processors.

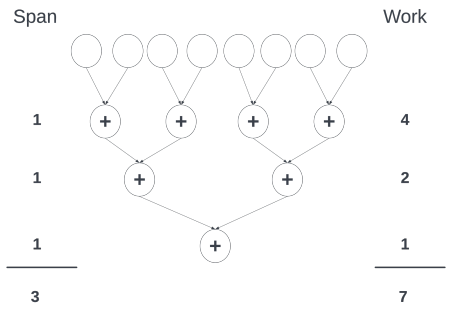

Figure 2: Rudimentary example of work and span

The example program T is shown in figure 2. This program sums eight values in a parallel fashion. The length of sequential dependencies is three, and thus we have the span S(T) = 3. Similarly, the amount of computations is seven and thus, the work W(T) = 7.

These examples merely serve to show how to reason about work and span. In practice, asymptotics is dealt with, and constants are disregarded. Thus one would say that both span and work of the example would be O(1).

2.2 Functional notation

This project involves vital concepts from Haskell-inspired functional programming languages. As such, an assortment of the most important ones will be briefly explained in this section. This is not intended to be a deep dive into functional programming but rather just an explanation of some of the essential concepts to understand the pseudocode and core of this project.

The type signatures used in this project extend the typical Haskell type signature that looks like the example in listing 1. Here the name of the function is separated from the type signature by two colons. This type signature denotes each input followed by an arrow where the last type is the output. So the function foobar takes two inputs of type a and returns an output of type b.

```
foobar :: a -> a -> b
```

Listing 1: Haskell type signature of an example function

Many of the functional primitives in the next section take anonymous functions as input. These are represented as in listing 2 where the input is separated by whitespace after the backslash on the left-hand side of the arrow, and the function output is defined on the right-hand side.

```
(\x y -> x + y)
```

Listing 2: Anonymous function that performs addition on two inputs

Associativity and commutativity are two concepts that are also important to certain func-

tions, which will be brought up in this project. If an operator with the type $\odot$ is commutative, then the two following representations are equivalent $x \odot y$ and $y \odot x$. Associativity means that the following two representations are equivalent $(a \odot b) \odot c$ and $a \odot (b \odot c)$

2.3 Futhark

Futhark is a data-parallel, purely functional language that offers a machine-neutral programming model. In recent years, the increase in CPU core speed has drastically slowed. It has ushered in a new age of programming where performance for larger programs is equivalent to how attuned they are to exploit parallelism. Hardware such as general-purpose graphics processing units (GPGPU) levy this parallelism to substantially outperform programs written in a single-core framework. (Podobas 2015) This is where the hardware-agnostic aspect of Futhark shines, as it allows a user to compile programs for both CPU and GPU backends.

A core feature of Futhark is supporting for arbitrary nesting of second-order array combinators (SOACs) such as `map`, `scan`, `reduce`, and `reduce_by_index`. (Henriksen, Hellfritzsch, et al. 2020) All of which are implemented as data-parallel constructs.

It also allows for a functional type of in-place updates in arrays. For example **let** xs[i] = x is syntactic sugar for **let** xs' = xs **with** [i] <= x which is the equivalent of returning a copy of xs with the i-th index updated in-place to be x.

The following section will briefly describe the semantics and syntax of each of the essential inbuilt functions and primitives used in this project. The Futhark notation will also be explained since the pseudo-code in later sections will be inspired heavily by it.

Futhark supports a size-dependent type system that verifies the compatibility of arrays passed to functions. In listing 3 we use a size parameter [n] to explicit quantify sizes of arrays. This parameter is not explicitly passed when calling `reverse` but is merely deduced by the input array of that size. Similarly, Futhark uses Hindley-Milner-Style type inference (Damas and Milner 1982), so most of the time, explicit type annotations are not needed. Thus we can define annotate functions when generic types that need to be compatible with all inputs that use that type. The asterisk before the return type indicates that the return value is *unique* and has no aliases. This annotation is mainly used here in the preliminaries for completeness.

`iota` is very useful Futhark function that given an integer n, returns an array of size n consisting of the set of integers $0, 1, .., n-1$. Its type can be found in listing 3. It is often used to create an array of indices. `reverse` takes an input array and reverses their order. `replicate` takes an explicit size and an element and returns an array with the element replicated to be of that size.

```
1 iota           : (n: i64) -> *[n]i64
2 reverse [n] 't : (x: [n]t) -> [n]t
3 replicate   't : (n: i64) -> (x: t) -> *[n]t
```

Listing 3: Type signature of common functions

The SOAC `map` applies a given function to every element in an input array and returns a new array with the size of the input array containing elements of the return type of the function. Its type signature can be found in listing 4. Work of map is $O(n \cdot W(f))$ and span $O(S(f))$

```
1 map 'a [n] 'x  : (f: a -> x) ->
2                  (as: [n]a) ->
3                  *[n]x
```

Listing 4: Type signature of `map`

The SOAC `scatter` seen in listing 5 calculates the equivalent of the imperative code found in figure 3.

```
1 for i in 0...length is-1:
2   ind = is[i]
3   v = as[i]
4   if ind >= 0 && ind < length as:
5     as[ind] = v
```

Figure 3: Imperative equivalent of the `Scatter`

Essentially `scatter` overwrites *dest* with the values in *as* for the corresponding index in *is*.

Notice how if the index in *is* outside the index domain of *dest* it has no effect. It has work of $O(n)$ and span $O(1)$

```
scatter 't [m] [n] :: (dest: *[m]t) ->
                      (is: [n]i64) ->
                      (as: [n]t) ->
                      *[m]t
```

Listing 5: Type signature of `scatter`

The SOAC `scan` is also called the inclusive prefix scan. It takes a binary associative operator *op*, a neutral element *ne*, and an input array *as*. The type signature can be found in 6. If $as = [a_0, a_1, a_2]$, and the operator $\circ$ then scan would return $[ne, ne \circ a_0, ne \circ a_0 \circ a_1]$. The exclusive prefix scan, which is not part of Futhark, is slightly different and will be referenced later in the pseudocode. This is not included in the language because it is easily derived from the inclusive scan by simply dropping the first element and appending the result of using the operator on the last value of the inclusive scan and the last value in the input array. Scan has work of $O(n \cdot W(op))$ and span $O(log(n) \cdot W(op))$

```
scan [n] 'a :: (op: a -> a -> a) ->
               (ne: a) ->
               (as: [n]a) ->
               *[n]a
```

Listing 6: Type signature of `scan`

The SOAC `reduce` works similarly to `scan` and has the same input. But instead of returning the entire array it simply reduces the whole array to one value. The returned value is the same as the last element in the result of the exclusive prefix scan. In the example used for `scan` the result would be $ne \circ a_0 \circ a_1 \circ a_2$. The work of this SOAC is $O(n \cdot W(op))$ and span is $O(\log(n) \cdot W(op))$.

```
reduce [n] 'a :: (op: a -> a -> a) ->
                 (ne: a) ->
                 (as: [n]a) ->
                 a
```

Listing 7: Type signature of `reduce`

The SOAC `reduce_by_index` (RBI) performs a reduction with *op* as the operator and the value in *dest* as well as all values in *as* that have the same index in *is*. The indices in *is* can be thought of which bucket the corresponding value in *as* should be reduced with. The equivalent imperative code can be expressed as shown in figure 1.

ne must be the neutral element for *op*, which may be applied multiple times and hence must be associative and commutative. Similarly to scatter, any indices outside the index domain of *hist_orig* will have their values disregarded. The work of RBI is $O(n \cdot W(op))$ and span is $O(n \cdot W(op))$ in the worst case where all updates are to the same position, but $O(W(op))$ in the best case.

```
reduce_by_index a [n] [m] :: (hist_orig: *[m]a) ->
                             (op: a -> a -> a) ->
                             (ne: a) ->
                             (is: [n]i64) ->
                             (as: [n]a) ->
                             *[m]a
```

Listing 8: Type signature of `reduce_by_index`

3 Introduction to Automatic Differentiation

Computing derivatives of functions is an integral part of essential algorithms in machine learning, scientific computing, and financial algorithms. (Baydin, Pearlmutter, and Radul 2015) A frequently used machine learning algorithm that can benefit from AD is gradient descent. It can be described as using the gradient of some loss function to optimize the parameters for the next iteration in the optimization process until a potential extremum is encountered.

Computing derivatives of functions in computer programs can generally be classified into four categories: (1) Manually deriving the function by hand and implementing the computation as code. (2) Numerical differentiation using finite difference approximations. (3) Symbolic differentiation using expression manipulation in computer algebra systems. (4) Automatic differentiation, which computes the derivative in conjunction with the result of the function. This last type is the focal point of this thesis.

Automatic Differentiation (AD) is a family of techniques used to efficiently and accurately evaluate the derivatives of numeric functions expressed as computer programs. It often involves augmenting the actual computation of the program also to populate data structures that, depending on the mode, either accumulate into the final derivative in lockstep with the regular computation or are used in an additional computation afterwards that accumulates the derivatives in reverse from the result to the input.

These two different modes will be explained more in-depth with examples in this section. They have in common the augmentation of the program that is being computed. This most often involves inserting additional intermediate variables that are derivatives with regard to the sub-computation at that point in the program.

3.1 Forward mode

AD in the forward mode associates each intermediate variable in the program with a derivative. For example if we have consider the computation of the function $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$, we can see how the tangent computation of the derivative with respect to x_1 beside it in table 1. This is done by firstly associating each intermediate variable v_i with a derivative as defined in equation 1

$$\dot{v}_i = \frac{\partial v_i}{\partial x_1} \tag{1}$$

This derivative is then used to express each sub-computation as a function with which we can reason about the right-hand side of the table mathematically. For example, if we express the result as such $y = f(g(h(x)))$ where g is a previous sub-computation that depends on the result of h and x is the input. If we want to differentiate y, we can use the chain rule to get a more digestible representation. Recall that the chain rule allows us to differentiate a composite function $P(x) = k(l(x))$ as $P'(x) = k'(l(x)) \cdot l'(x)$.

$$\begin{aligned}
\dot{y} &= \frac{\partial f(g(h(x)))}{\partial x_i} \\
&= \frac{\partial f(g(h(x)))}{\partial g(h(x))} \cdot \frac{\partial g(h(x))}{\partial x_i} \\
&= \frac{\partial f(g(h(x)))}{\partial g(h(x))} \cdot \left(\frac{\partial g(h(x))}{\partial h(x)} \cdot \frac{\partial h(x)}{\partial x_i} \right) \\
&= \frac{\partial f(g(h(x)))}{\partial g(h(x))} \cdot \left(\frac{\partial g(h(x))}{\partial h(x)} \cdot \left(\frac{\partial h(x)}{\partial x} \cdot \frac{\partial x}{\partial x_i} \right) \right)
\end{aligned}$$

Figure 4: Chain rule used to decompose an example function in forward AD fashion

Then forward mode AD can be seen as computing $\dot{y}$ in a right-to-left manner, starting from the input. As such we can augment the original program to compute these partial derivatives of the sub-expressions in lockstep with their actual computation. The AD transformation for the forward mode involves the core rewrite rule found in equation 2

$$\begin{aligned}
&\textbf{let } v = f(a, b) \\
&\qquad \downarrow \\
&\textbf{let } v = f(a, b) \\
&\textbf{let } \dot{v} = \tfrac{\partial f(a,b)}{\partial a}\dot{a} + \tfrac{\partial f(a,b)}{\partial b}\dot{b}
\end{aligned} \tag{2}$$

This is exactly what we see on the right hand side of table 1, where we differentiate with respect to x_1.

	Regular computation		Tangential forward mode AD	
	$v_{-1} = x_1$	$= 2$	$\dot{v}_{-1} = \dot{x}_1$	$= 1$
	$v_0 = x_2$	$= 5$	$\dot{v}_0 = \dot{x}_2$	$= 0$
	$v_1 = \ln v_1$	$= \ln 2$	$\dot{v}_1 = \dot{v}_{-1}/v_{-1}$	$= 1/2$
	$v_2 = v_{-1} \times v_0$	$= 2 \times 5$	$\dot{v}_2 = (\dot{v}_{-1} \times v_0) + (\dot{v}_0 \times v_{-1})$	$= 5 + 2$
	$v_3 = \sin v_0$	$= \sin 5$	$\dot{v}_3 = \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
	$v_4 = v_1 + v_2$	$= 0.693 + 10$	$\dot{v}_4 = \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5.5$
	$v_5 = v_4 - v_3$	$= 10.693 + 0.959$	$\dot{v}_5 = \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
	$y = v_5$	$= 11.652$	$\dot{y} = \dot{v}_5$	$= 5.5$

Table 1: Forward mode AD example setting $\dot{x_1} = 1$ to compute $\frac{\partial y}{\partial x_1}$

Performing forward AD on a function $f : \mathbb{R}^n \to \mathbb{R}^m$ with n input x_i and m outputs y_j and setting one variable $\dot{x}_i = 1$ and the rest to zero can be generalized as computing one column of the Jacobian. Repeating this process for every input makes it possible to populate the entire Jacobian matrix.

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

Figure 5: Jacobian matrix for function f with n inputs and m outputs

Treating the input variables in the forward tangent as a vector of size n and initializing each derivative as its counterpart in this vector allows for an efficient and matrix-free method of computing the Vector-Jacobian product.

In Futhark, the forward mode AD is called by the function `jvp` the definition of which can be found in listing 9

```
jvp :: (f: ϵ -> β) -> (x: ϵ) -> (x': ϵ) -> ϵ
```

Listing 9: Type signature of *jvp*

3.2 Reverse mode

AD in the reverse mode uses the same idea of decomposition as shown in figure 4. However, instead of starting the computation from the inputs, the reverse mode starts from the derivative of the output. It computes the derivative shown in the figure in a left-to-right fashion, as can be seen in figure 6. This means that the regular computation needs to fully compute the expected result before the AD can start computing derivatives. The regular computation is referred to as the forward pass, and the derivative computation that follows is referred to as the backwards pass.

$$
\begin{aligned}
\bar{y}_i &= \frac{\partial f(g(h(x)))}{\partial x} \\
&= \frac{\partial f(g(h(x)))}{\partial g(h(x))} \cdot \frac{\partial g(h(x))}{\partial x} \\
&= \left(\frac{\partial f(g(h(x)))}{\partial g(h(x))} \cdot \frac{\partial g(h(x))}{\partial h(x)} \right) \cdot \frac{\partial h(x)}{\partial x}
\end{aligned}
$$

Figure 6: Chain rule used to decompose an example function in reverse AD fashion

In the backwards pass each sub-expression v_i is complemented by an adjoint value $\bar{v}_i = \frac{\partial y_j}{\partial v_i}$. These represent how sensitive the derivative with regards to y_j is with respect to changes in v_i. A sub-expression can be used multiple times throughout a program, so these adjoints are accumulated accordingly throughout the backwards pass.

The reverse mode AD transformation involves the core rewrite rule in equation 3

$$
\begin{aligned}
&\textbf{let } v = f(a,b) \\
&\qquad \downarrow \\
&\textbf{let } v = f(a,b) \\
&\qquad \vdots \\
&\textbf{let } \bar{a} \mathrel{+}= \tfrac{\partial f(a,b)}{\partial a} \bar{v} \\
&\textbf{let } \bar{b} \mathrel{+}= \tfrac{\partial f(a,b)}{\partial b} \bar{v}
\end{aligned}
\tag{3}
$$

The backwards pass starts by initializing the adjoints of the output as the unit vector for the output we wish to differentiate relative to. Setting $\bar{y} = e_i$ where e_i is the i-th unit vector and propagating the adjoints in reverse from outputs to inputs will enable us to compute a row of the Jacobian at a time. This process can be seen for the same example as earlier in table 2.

This process will need to be repeated for each output to populate the Jacobian matrix fully. Notice how this differed from the forward mode, where the number of repetitions depends on the number of inputs.

Similarly to the forward mode, a Jacobian-Vector product can be computed using the reverse mode AD. This is done by initializing the $\bar{y}$ to the vector. This project will be using the Futhark notation for a `vjp` function to denote reverse mode AD transformation. This operator takes a function *f*, input *x*, as well as the adjoint of the result *y'*.

```
1 vjp :: (f: ϵ -> β) -> (x: ϵ) -> (y': β) -> ϵ
```

Listing 10: Type signature of *vjp*

	Regular computation		
	$v_{-1} = x_1$	$= 2$	
	$v_0 = x_2$	$= 5$	
	$v_1 = \ln v_1$	$= \ln 2$	
	$v_2 = v_{-1} \times v_0$	$= 2 \times 5$	
	$v_3 = \sin v_0$	$= \sin 5$	
	$v_4 = v_1 + v_2$	$= 0.693 + 10$	
	$v_5 = v_4 - v_3$	$= 10.693 + 0.959$	
	$y = v_5$	$= 11.652$	
		Reverse phase	
		$\bar{v}_5 = \bar{y}$	$= 1$
		$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4} = \bar{v}_5 \times 1$	$= 1$
		$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3} = \bar{v}_5 \times (-1)$	$= -1$
		$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1} = \bar{v}_4 \times 1$	$= 1$
		$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2} = \bar{v}_4 \times 1$	$= 1$
		$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0} = \bar{v}_3 \times \cos v_0$	$= -0.284$
		$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}} = \bar{v}_2 \times v_0$	$= 5$
		$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0} = \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
		$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}} = \bar{v}_{-1} + 1 / v_{-1}$	$= 5.5$
		$\bar{x}_2 = \bar{v}_0$	$= 1.716$
		$\bar{x}_1 = \bar{v}_{-1}$	$= 5.5$

Table 2: Reverse mode AD example computing both $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$.
Notice how the primal trace is computed before the accumulation of adjoints.

At first glance, the forward mode AD looks more efficient because it can be computed in lockstep with the results, but this is often not the case. If we have a program where the amount of inputs n differs a lot from the number of outputs m, we can confidently reason which mode is most efficient. If $m << n$, then the reverse mode AD will be the more efficient choice since the difference in how many repetitions are needed will significantly outweigh the cost of additional accumulation computations added to the program. In other cases, the forward mode is expected to be the faster choice for the same reasons.

In most machine learning algorithms, this is the exact case. The number of inputs will most often be significantly more than outputs. The goal in machine learning is often to take a massive amount of inputs and give either a classification in a single output or to output a relatively small vector which can be used as weights in a model. Thus, we expect the reverse mode AD to be the more efficient choice for traditional machine learning algorithms.

4 Code transformation for primitives

This section will explain how the rewrite rule for reverse AD, seen in equation 3, is used to reason about the code transformations for the functional primitives in Futhark. Specifically, the transformation for `reduce` will be reasoned about first. Then it will be presented how this can be extended to a valid code transformation for the `reduce_by_index` primitive.

4.1 Reduce

Recall that the semantics for `reduce`, seen in listing 7, assumes the operator is associative. If a reduce has the operator $\odot$, neutral element $e_\odot$, input $as = [a_0, a_1, .., a_{n-1}]$ and its output bound to the variable y. Then we can reason about the contributions to the adjoint if we group the terms as

$$\text{let } y = (a_0 \odot .. \odot a_{-1}) \odot a_i \odot (a_{i+1} \odot .. \odot a_{n-1}) \tag{4}$$

Then for every i we can define the variable l_i as the left group as such $l_i = (a_0 \odot .. \odot a_{-1})$ and define r_i similarly for the group right of a_i. Thus if we consider these two variables as constants we can use the rewrite role for reverse AD in equation 3 to get the following

$$\bar{a}_i \overline{+}= \frac{\partial(l_i \odot a_i \odot r_i)}{\partial a_i} \bar{y} \tag{5}$$

where $\overline{+}$ denotes a potentially vectorized addition.

The right-hand side of this equation can be generically generated in code as a function f that can be mapped over all a_i, l_i, and r_i.

$$f \leftarrow \text{vjp } ((\backslash\, l_i\ a_i\ r_i\ \rightarrow l_i \odot a_i \odot r_i), (ls, as, rs), \bar{y})$$

To compute l_i for all possible i, we perform an exclusive scan over the input with the operator $\odot$. For r_i, we do the same thing but over the reversed input, which we reverse again to have the final list. Assuming $\odot$ has no free variables, we can write the reverse AD rewrite for `reduce` as is shown in figure 7.

```
1 -- Forward sweep
2 let y = reduce ⊙ e_⊙ as
3 -- Reverse sweep
4 let ls = scan^exc ⊙ e_⊙ as
5 let rs = reverse as |> scan^exc (\a b -> b ⊙ a) e_⊙ |> reverse
6 let as' += map f ls as rs
```

Figure 7: The rewrite rule for `reduce`

Although adding overhead in the computation of *reduce_by_index*, the addition of the reverse sweep does not affect the work or span asymptotics for the program.

4.2 Reduce_by_index

Recall how the semantics of `reduce_by_index` at its core an N-ary implementation of `reduce`. This indicates that the transformation above will also be used here but extended to compute for each bucket instead. This code transformation for arbitrary operators is not implemented in Futhark and is the main extension proposed in this project.

The first part of this section covers the reasoning and explanation of the rewrite rule for RBI for arbitrary operators. The second part of this section introduces three special cases for which there is a theoretically more efficient rewrite rule.

4.2.1 Overview of how the rewrite rule is derived

The forward pass in the rewrite will also only consist of the *reduce_by_index* call. But for the reverse pass we will need to do an extension of the rewrite rule for `reduce` Consider the statement

```
let y = reduce_by_index hist_orig ⊙ e_⊙ is as
```

If the operator is lifted to work on arrays we can deconstruct the expression above to the line below where w is the of *hist_orig*.

```
let y = hist_orig ⊙^L reduce_by_index (replicate w e_⊙) ⊙ e_⊙ is as
```

The expression above shows how the RBI call computes an N-ary reduction and adds it to the values already present in *hist_orig*. This indicates that with the rewrite rule for AD

seen in equation 3 we will also have to accumulate on its adjoint. So the goal of the reverse pass will therefore be to calculate the contributions to the adjoint of *as* as well as *hist_orig*.

To derive $\overline{\text{hist\_orig}}$ we can think of the result of each bucket as $y_i = x_i \odot s_i$ where s is the result of the right-hand side of $\odot^{\text{L}}$ and x is *hist_orig*. Hence the adjoint can be computed by

$$\overline{hist\_orig}_i \mathrel{+}= \frac{\partial(hist\_orig_i \odot s_i)}{\partial hist\_orig_i} \bar{y}_i \tag{6}$$

Then similarly to `reduce` we can derive the contributions to $\overline{as}$ using variables that represent all values scanned left and right of a_i of a specific bin, see equation 4. However, for RBI we will need to extend them based on the corresponding index in *is*. This is to make the l_i and r_i variables only take values in their bucket into account. Assume this, and also the adjoint of our contributions $\bar{s}$ is already computed. Since as is of size n and $\bar{s}$ is of size w where $w <= n$, we will need to make a new variable that is size n and holds the adjoint of the bucket that the corresponding value in as belongs to. This stretched representation will be referred to as $\bar{s}^s$. With this, the adjoint of as can be computed as such

$$\bar{a}_i \mathrel{+}= \frac{\partial(l_i \odot a_i \odot r_i)}{\partial a_i} \bar{s}_i^s \tag{7}$$

4.2.2 Rewrite rule for the general case

Since we are interested in a rewrite rule for programs, we will need to iron out some of the assumptions made in the derivations above. Firstly we need to compute the l_i and r_i variables for each bucket. Secondly we need to compute our contributions, s, before we can accumulate $\overline{hist\_orig}$. After this accumulation is done we need to compute $\bar{s}$, which is defined similarly to $\overline{hist\_orig}$ just differentiated with regards to s instead of *hist_orig*. Then we have everything we need to compute $\overline{as}$.

To compute the l_i and r_i variables for each bucket, we will need to perform the same scans and reverse them as in the rewrite rule for `reduce` but also extend to take the index into account and compute them isolated from the values of other buckets.

The order of the input arrays can be shuffled, which is unsuitable for parallel operations. This is because of the memory locality between values belonging to the same bucket.

Therefore we will sort the values in both *as* and *is* according to the value in *is*, and then create a flag array to indicate the start of these bucket segments. This allows segmented operations that are much more suitable to run in parallel. Assuming a constant key size, we can do this without affecting the work-depth asymptotic by using Radix sort.

With the sorted elements, we can use segmented scans to get results for all buckets in one parallel operation. These segmented operations take arrays with binary indicators of whether the value is the segment's start and then compute the scan or reduce for all values within the same segment. An example of this can be found below, where the inner values of certain variables can be found in the comments

```
1  let as = [1, 2, 6, 3, 5, 8]
2  let is = [0, 1, 2, 1, 1, 0]
3
4  let (sorted_is, sorted_as) = radixSort (is, as)
5  let seg_flags = mkSegFlags sorted_is
6
7  -- sorted_is => [0, 0, 1, 1, 1, 2]
8  -- sorted_as => [1, 8, 2, 3, 5, 6]
9  -- seg_flags => [1, 0, 1, 0, 0, 1]
10
11 let seg_scan_res = seg_scan + 0 seg_flags sorted_as
12 -- seg_scan_res   => [0, 1, 0, 2, 5, 0]
```

Figure 8: An example of sorting wrt. *is* and a segmented scan operation

Furthermore, if we perform an `iota` to represent the order and zip it to *as* before we sort it, we can easily permute our adjoint contribution back to the original order with a `scatter` when needed.

To compute the contributions, we use our operator on the last value in each segment and the corresponding value in *lis*. This corresponds to performing a segmented scan.

All of these intermediate variables then enable the computations of the adjoint values derived above. This results in the rewrite rule shown in figure 9. It includes some auxiliary functions that are not shown since they are considered implementational details.

```
1  -- Forward pass
2  let y = reduce_by_index hist_orig ⊙ e⊙ is as
3
4  -- Backwards pass
5  let as_iota = zip as (iota n)
6  let sorted_is, (sorted_as, sorted_iota) = radixSort is as_iota
7  let segment_flags = mkSegFlags sorted_is
8  let lis = seg_scan^exc ⊙ e⊙ segment_flags sorted_as
9
10 let (rev_vals, rev_flags) = reverseCorrectly sorted_as segment_flags
11 let ris = seg_scan^exc ⊙ e⊙ rev_flags rev_vals |> reverse
12
13 let seg_last_lis, seg_last_value = extractLastInSeg lis sorted_as
14 let s = map2 op seg_last_lis seg_last_value
15
16 let hist_orig = map3 (\ x_i s_i ȳ_i -> (f_x x_i s_i) * ȳ_i) hist_orig s ȳ
17
18 let s̄ = map3 (\ x_i s_i ȳ_i -> (f_s x_i s_i) * ȳ_i) hist_orig s ȳ
19
20 let ās_contrib_sorted =
21     vjp((\ li ai ri -> li ⊙ ai ⊙ ri),
22         (lis, sorted_as, ris),
23         s̄)
24 let ās_contrib = backPermute ās_contrib_sorted sorted_is
25 let ās += ās_contrib
26
27 where
28     ȳ is available after forward pass
29     f_x is (\ x s → ∂(x⊙s)/∂x)
30     f_s is (\ x s → ∂(x⊙s)/∂s)
```

Figure 9: The reverse mode AD rewrite rule for `reduce_by_index`

Since none of these operations or functions has a larger asymptotic work or span than *reduce_by_index*, the asymptotics are unaffected. Although obviously introduces some overhead to the program that is not negligible in practice.

This rewrite rule is not the most efficient method of performing reverse AD of RBI. Although it does not affect the asymptotics of the program, many redundant computations can be avoided. The main one is how the result is computed twice because we need the intermediate variable s. This could be avoided by computing *lis* and *sorted_as* in the forward pass and using them to create the result. This, along with other details, will be explored and addressed in the implementation section of this project.

4.2.3 Special case: Multiplication

When the operator is multiplication, there is a special case when there are one or more zeroes in the input. Since the result will be a zero in these cases, the Jacobian will either be fully or almost entirely inhabited by zeroes.

This is because if any more than one a_i in a bucket is zero, the adjoint will always result in zero. Although, if there is only a single zero input value in the bucket there is a case where neither l_i nor r_i for a bucket evaluate to zero and because the adjoint of that specific bucket can be simplified to $(l_i \odot r_i) \cdot \bar{y}$ there is a single value in the adjoint of that bucket that is nonzero.

The proposed algorithm does a slightly modified forward pass to compute the nonzero products and the zero counts. This is done by mapping over the values in *as* to create tuples containing the value and an integer boolean indicating if the value was zero. If the value were zero, the first part of the tuple would have a one instead. This can then be reduced into the two variables by using a single reduce, which multiplies the first value in the tuple and sums the second. Then all that remains is to check whether there were any zeros in the input and set the output result for that bucket to either the nonzero product or zero accordingly.

In the backward pass, all values are mapped, and their adjoints are computed depending on the value of their buckets' zero counts. If there are no zeroes, the normal adjoint value is computed with the regular rewrite rule for *reduce*. However, if there is a single nonzero value and the current value is zero, the adjoint value for that index is $(l_i \odot r_i) \cdot \bar{y}$. Otherwise, zero is returned as the adjoint.

```
-- case : let y = reduce_by_index hist_orig (*) 1 is as
-- Input:
-- Assuming t is a type of real number
-- hist_orig: [w]t
-- is:        [n]i64
-- as:        [n]t

-- Forward pass

-- First compute zeroes for each bucket
let as_zeros = map (\a -> if a==0 then 1 else 0) as
let zero_counts = reduce_by_index (replicate w 0) (+) 0 is as_zeros
-- Then compute nonzero product
let as_nz = map (\a -> if a == 0 then 1 else a)
let as_nzprod = reduce_by_index (replicate w 1) (*) 1 is as_nz

let y_contrib =
    map2 (\nzp zc ->
            if (zc > 0) then 0 else nzp
         ) as_nzprod zero_counts
let y = map2 (*) y y_contrib

-- Backwards pass
let hist_orig_bar = map2 (*) y_bar y_contrib

let as_bar_contrib =
    map (\i a ->
            if zero_counts[i] == 0 then
                (as_nzprod[i] / a) * y_bar[i]
            else if zero_counts[i] == 1 && a == 0 then
                as_nzprod[i] * y_bar[i]
            else
                0
        ) as is
as_bar += as_bar_contrib
```

Figure 10: Efficient reverse mode AD special case for RBI with multiplication

4.2.4 Special case: Min/Max

The second special case is when the operator is either *max* or *min*, and the neutral element is the smallest or largest possible element, respectively. This optimisation aims to extend the forward pass with an operator that can save at which index the maximum or minimum element is found for each bucket and then use that index to compute the adjoint directly. This saves a lot of unnecessary computations otherwise done in the general case.

```
1  -- case : let y = reduce_by_index hist_orig minmax ne is as
2  -- Input:
3  -- Assuming t is a type of real number with valid min/max function
4  -- hist_orig: [w]t
5  -- is:        [n]i64
6  -- as:        [n]t
7
8  -- Forward pass:
9  let ind_op =
10     (\ acc_v acc_i v i ->
11         if (acc_v == v) then (acc_v, min acc_i i)
12         else if (acc_v == minmax acc_v v)
13              then (acc_v, acc_i)
14              else (v, i))
15 let (y, y_ind) =
16     reduce_by_index hist_orig ind_op (ne,-1) is (zip as (iota n))
17
18 -- Backwards pass:
19 let hist_orig_bar = map (\min_ind el ->
20                     if min_ind == -1 then el else 0
21                 ) hist_inds y_bar
22
23 let as_bar_contrib_reordered =
24     map2 (\ i h_v ->
25         if i == -1 then 0 else as_bar[i] + h_v
26     ) histo_inds histo_bar
27 let as_bar_contrib = scatter histo_inds as_contrib_reordered
28 let as_bar += as_bar_contrib
```

Figure 11: Efficient reverse mode AD special case for RBI with min/max

4.2.5 Special case: Addition

When the operator is addition, either vectorised or scalar, and the neutral element is zero then the regular reduce has the special adjoint case of $\frac{\partial(l_i+a_i+r_i)}{\partial(a_i)}\bar{y} = \bar{y}$ which implies that the return sweep is simply $let\ \overline{as}\ +\!= \bar{y}$.

To translate this into the `reduce_by_index` structure we simply change the adjoint contributions index into the adjoint of the result y.

```
1  -- case : let y = reduce_by_index hist_orig (+) 0 is as
2  -- Input:
3  -- Assuming t is a type of real number
4  -- hist_orig:  [w]t
5  -- is:         [n]i64
6  -- as:         [n]t
7
8  -- Forward Pass:
9  let y = reduce_by_index hist_orig (+) 0 is as
10
11 -- Backwards Pass:
12 let hist_orig_bar = y_bar
13
14 let as_bar_contrib =
15     map (\i ->
16         y_bar[is[i]]
17     ) (iota n)
18
19 let as_bar += as_bar_contrib
```

Figure 12: Efficient reverse mode AD special case for RBI with addition

5 Implementation

This section will start by giving an introduction to the intermediate representation (IR) used in the Futhark compiler. This includes an explanation of how it is used and an example. Followed by this will be a thorough look into how the rewrite ruled for RBI, seen in figure 9, is translated to pseudocode. After which selected parts of the code will be discussed and explained. Finally, a short discussion of the shortcomings of this solution will be presented.

5.1 Intermediate representation

This section will present a simplified version of the intermediate representation used in the compiler of Futhark. This IR will then be used to show how the rewrite rules from the previous chapter can be implemented as extensions to the Futhark compiler.

```
1  data SOAC rep =
2        Screma SubExp [VName] ScremaForm
3      | Hist SubExp [VName] [HistOp rep] (Lambda rep)
4      | Scatter SubExp [VName] (Lambda rep) VName
5  data ScremaForm rep =
6        ScremaForm [Scan rep] [Reduce rep] (Lambda rep)
7  data SubExp rep =
8        Constant PrimValue, Var VName
9  data Scan rep =
10       Scan (Lambda rep) [SubExp]
11 data Reduce rep =
12       Reduce (Lambda rep) [SubExp]
13 data Lambda rep =
14       Lambda [Param rep] (BodyT rep) [Type]
15 data PrimValue rep =
16       Intvalue, FloatValue
17 data BasicOp =
18       Index VName SubExp
19     | Iota, CmpOp, SubExp, BinOp
20 VName :: String
```

Figure 13: Simplified IR

The simplification is merely a tool to convey the most vital aspects of how the rewrite rules get translated to compiler code.

In Figure 13 the most important subset of the IR used in this project is shown. The *Screma* is a combination of scan, reduce, map, which takes as input; a size, a list of input arrays and this combination. Semantically it performs the map first to create the input for the scans, which is followed by the reductions. The *Hist* is the internal representation of *reduce_by_index*. The *Lambda* is an index generating function. In the code we implement, it will always just be the identity function. *BasicOp* contains many of the basic operations such as indexing, binary operations like subtraction and addition, comparing used to make predicates, and more. These are, for the most part, self-explanatory when seen in context. Certain data types are omitted because their internals are needlessly complicated for this demonstration.

The Futhark compiler also contains many functions to modularise compiler extensions. This project has also introduced a number of them. However, they will only be explained if it is necessary for the following sections.

5.2 The essence of the Futhark compiler

Futhark has many complicated features, most of which are already compiled before hitting the part of the compiler pipeline that performs AD. (Elsman et al. 2018) There are also several standard optimisations performed prior to AD. At this point, the only higher-order functions that remain are the SOACs. However, it is followed by a simplification step that can potentially fuse SOACS if it is a valid optimisation.

The compiler itself uses monads to build the contributions of each step of the compiler into the final product. The adjoints for reverse AD are similarly stored and combined with the AD monad, `ADM`. This monad allows for accumulating adjoint values and is responsible for storing and updating the intermediate adjoints. There are specialised monads that allow for building expressions. For example, the *Builder* monad will be used to build the *BodyT* expression, which can be used to express anonymous functions, among other things. Both of these monads are implementations of *MonadBuilder*, which allows for many vital utilities like binding expression, referencing them, scoping and more.

This project will extend the part of the compiler in which *vjp* has been called with the `Hist` SOAC. Here we will match the SOAC pattern to ensure it is the correct one, and for the

special cases, we will also match to see that they are valid cases.

5.3 IR example

This section will present a simple example to showcase how the IR is used for code generation. This is supposed to give a general idea of what compiler code is equivalent to a simple source code example and what the generated code from this looks like.

A simple snippet from the pseudocode of the special multiplication case will have its implementation walked through step by step. The snippet in question can be seen in figure 14a. This is the line where an indicator array for whether the values in *as* are zeros or not is created. This is written as a simple anonymous function that is mapped over *as* originally.

In figure 14c, the compiler code implementing this line can be seen. Note that the lines that create *oneSubExp*, and *zeroSubExp* are omitted. They are simple variables that represent the numerical values zero and one for *t*, which is the type of input. Although there are many compiler-specific intricacies in even a simple snippet, only the most important will be explained.

In line 1 the *MonadBuilder*s building block *letSubExp* is used to define a zero value of type *t* under the name "t_zero". In line 2 the utility function *newParam* defines a parameter variable which has the name "value". Line 3 starts the creation of the body of the anonymous function. Here the *runBodyBuilder* is called and has its scope extended with the newly created parameter. In Line 4, the body of the anonymous function is started, and the following list is its output, which is only a single value. Line 5 through 8 shows how an if statement is declared, where line 6 is the predicate, line 7 is the true branch, and line 8 is the false branch. On line 10, the list of parameters, the body of the anonymous function and the return type are used to construct a Lambda expression. On lines 11 and 12, this Lambda is used in a Screma with the size and input array to bind it to an expression called *as_zeros*.

```
let as_zeros =
map (\a ->
    if a==0 then 1 else 0
) as
```

(a) Source code

```
let {as_zeros : [n]f32} =
    map(n, {as},
        \ {value : f32}
          : {f32} ->
          let {cond : bool} =
            eq_f32(value, 0.0f32)
          let {x : f32} =
             if cond
             then {1.0f32}
             else {0.0f32}
             : {f32}
           in {x})
```

(b) Generated code

```
t_zero <- letSubExp "t_zero" $ zeroExp $ Prim t
v_f <- newParam "value" $ Prim t
zero_lam_bdy <- runBodyBuilder . localScope (scopeOfLParams [v_f]) $ do
    eBody [
      eIf
          (eCmpOp (CmpEq t) (eParam v_f) (toExp t_zero))
          ( eBody [ eSubExp oneSubExp ] )
          ( eBody [ eSubExp zeroSubExp ] )
    ]
let zero_lam = Lambda [v_f] zero_lam_bdy [Prim int64]
as_zeros_indicator <-
    letExp "as_zeros" $ Op $ Screma n [as] (ScremaForm [] [] zero_lam)
```

(c) Compiler code

Figure 14: The three stages of the compiler code

In figure 14b, a simplification of the generated code is portrayed. It shows how the *as_zeros* is defined as the result of a map function. The semantics of the generated code is slightly different from regular source code. The map, for example, has three inputs; the size, the input, and the anonymous function, which has the output in the final array.

5.4 Pseudocode

This subsection will present the pseudocode for how the reverse AD rewrite rule for *reduce_by_index* has been implemented by this project. There will be some auxiliary functions that are glossed over for their triviality, but the actual implementation is still as close to this pseudocode as possible.

Recall how the definition of *vjp*, seen in listing 10, differentiates with regards to the second input given to it. So even if it is possible to only differentiate with regards to only one of the inputs *hist_orig* or *as*, the pseudocode below is optimised for the case that the adjoint for both arrays is needed. This is the case in most programs since all intermediate adjoints need accumulation.

The example call and inputs can be seen in listing 11, here w is the size of the output, and n is the size of the input. As the algorithm involves filtering out indices, and sorting values, while also using the original array, there can easily be some confusion about which variables are actually sorted and what size they are. Therefore the pseudocode below will use these simple notations in variable names to indicate size and whether they are the sorted version of an original array. Any input variable with a suffix apostrophe, ′, is the filtered version. For example, n' is the size of the filtered arrays, and is' is the filtered indices. Similarly, any sorted variable will have the *_sorted* suffix. Furthermore, to be able to reconstruct the original order so an `iota` variable will be constructed at the start and used to represent the order of the variables.

```
1 let y = reduce_by_index hist_orig op ne is as
2
3 where
4     hist_orig    :[w]t
5     is           :[n]i64
6     as           :[n]t
7     ne           :t
8     op           :t -> t -> t
```

Listing 11: Types of the inputs

To filter, we would first like to flag any indices in *is* that are inside the index domain. These can then be scanned to create new indices that increase continuously but disregard any values outside the index domain. The filtering is done by creating a temporary n-sized index array equal to the scanned flag array except for the indices that should be disregarded, which are set to negative ones instead. This n-sized array will then be used to scatter an `iota n` to create a final index array of size n'. This new array is used to create the array that contains the buckets for all values we actually will use in our reductions.

```
let temp_flags = map (\ind -> if 0 <= ind < w then 1 else 0) is
let tmp_flags_scanned = scan (+) 0 temp_flags
let n' = last tmp_flags_scanned
let temp_inds =
    map ( \flag flag_scan ->
        if flag == 1 then flag_scan - 1 else -1
    ) temp_flags temp_flags_scanned
let iota' = scatter n' (Scratch int n') temp_inds (iota n)
let is' = map (\i -> is[i]) iota'
```

Listing 12: Pseudocode of how *is* is filtered

The sorting is performed according to the values in *is'* since we are interested in a representation where we have all values from the same buckets grouped. We also the *iota'* to enable us to reverse the sorting for *is'* later when needed. The sorting is performed with a custom radix sort, which can be seen in the appendix listing 19. We then use the sorted and filtered *iota* to index into *as*, which results directly in a filtered and sorted representation.

```
let (sorted_iota', sorted_is') = radix_sort (iota',is')
let sorted_as' = map(\i -> as[i]) sorted_iota'
```

Listing 13: Pseudocode of how *is'*, *iota'* and *as* are sorted

The segmented scans that we will perform require a flag array as input that indicates the start of each segment. This array is created by iterating through the *sorted_is'* array and indicating whether the last bucket is different from the current.

```
let seg_flags =
    map (\index ->
        if index == 0 then 1
        else
            if sorted_is'[index] == sorted_is'[index-1]
             then 0
            else 1
    ) (iota n')
```

Listing 14: Pseudocode of how the segment flag array is created

The pseudocode for the segmented scan can be found in the appendix listing 20. The listing below shows how the l_i and r_i variables for each segment are computed using segmented scans. Notice how the reverse segmented scan has the flags reversed and rotated by one place for them to indicate where each segment begins for the reversed values correctly.

```
let lis = sgmScanExc op sorted_as' seg_flags

let rev_sorted_as' = reverse sorted_as'
let rev_seg_flags_tmp = reverse seg_flags
let rev_seg_flags =
    map (\ ind ->
        if ind == 0 then 1
        else rev_seg_flags_temp[ind-1]
    ) (iota n')

let ris = sgmScanExc op rev_as' rev_sorted_as'
```

Listing 15: Pseudocode of the segmented scans

Since we have the l_i values already available, we can perform our reduction computation by using the operator on each of the last l_i values for each segment along with the last value in each segment from *sorted_as'*. This is done in listing 16 by scattering and then computing the contributions and adding them to the *hist_orig* input variable with two simple maps. This concludes the forward pass.

```
let seg_last_flag = map (\i -> if i == n'-1 then i
                        else if seg_flags[i+1] == 1
                             then i
                             else -1
                  ) (iota n')

let seg_last_lis   = scatter (replicate new) seg_last_flag lis
let seg_last_value = scatter (replicate new) seg_last_flag sorted_as'

-- s is the contributions made to y
let s = map2 op seg_last_lis seg_last_value

let y = map2 op hist_orig s
```

Listing 16: Pseudocode of how *lis* and *as* is used to compute result of RBI

Backwards pass

With the result of the forward pass available, the only thing missing is computing the contributions to the adjoint values. The sorted and filtered variables will still be used because their contributions will be scattered to an array of zeroes which we will use for the final update of adjoints. Thus the values filtered do not have any contributions to the adjoint of *hist_orig* or *as*. At this point, the adjoint of *y* will be available.

An auxiliary function will be used to differentiate the input operation $\odot$ with relation to the first or second parameter. The lambda function *derivOpByX* variable will then represent $(\backslash\ x\ y\ \rightarrow \frac{\partial(x \odot y)}{\partial x})$, and similarly the *derivOpByX* is the equivalent of $(\backslash\ x\ y\ \rightarrow \frac{\partial(x \odot y)}{\partial y})$. Giving the first adjoint lambda function *hist_orig* and *s* as its inputs and then multiplying this by $\bar{y}$ will give us the the following representation for the computation of the adjoint of each bucket x_i in *hist_orig* and s_i in *s* $\bar{x}_i = \frac{\partial(x_i \odot s_i)}{\partial x_i} \cdot \bar{y}_i$.

```
let hist_orig_bar_temp = map2 derivOpByX hist_orig s
let hist_orig_bar = map2 (\t y_b -> t*y_b) hist_orig_bar_temp y_bar
```

Listing 17: Pseudocode of how the adjoint for *hist_orig* is computed

To compute the adjoint of *sorted_as* we first need the adjoint of s, as seen in equation 7. Then since each value in the same bucket needs their derivative multiplied by the same adjoint value we simply stretch out *s_bar* to also be of size *n'*. Then we take the differential of *op* and use it in a new lambda function which computes (l_i op_adj a_i op_adj r_i) where op_adj is the derived operator. This function can then be used in a `vjp` call with *s_bar_repl* as the adjoint of the result, the result of which simply needs to ber back permuted to the original order and then we have our contributions to the adjoint of *as*.

```
1  let s_bar_temp = map2 derivOpByY hist_orig s
2  let s_bar = map2 (\ x y -> x*y) s_bar_temp y_bar
3  let s_bar_repl = map (\bin -> hist_temp_bar[bin]) sorted_as
4
5  let map_lam = (\li as vi -> op_adj (op_adj li as) vi)
6  let as_bar_contrib_reordered =
7      vjp(map_lam, (lis, sorted_as', ris), s_bar_repl)
8  let as_bar_contrib = scatter n' sorted_iota' as_bar_contrib_reordered
9
10 as_bar += as_bar_contrib
```

Listing 18: Pseudocode of how the adjoint for *as* is computed

5.5 Implementational details

This project has implemented the pseudocode above along with the special cases for multiplication, addition, and min/max found in figure 10, 12, and 11 respectively. This section aims to give some insight into a few of the non-trivial implementational decisions made. There will also be examples of how these decisions were translated to compiler code.

Important to note is that some details were discussed with my fellow student Søren Brix for the implementation of the general case. This was done with my supervisor's permission and ultimately only helped us get a working product ready in time. We did not discuss the special cases or much about the theory behind the project.

In the implementation of *vjp* it first starts by matching the input and then calling the appropriate internal function to handle it accordingly. For example, SOACs have their own file where they have matched, and the functions that compute their adjoint contributions

are called. This file has been extended by this project to match the *Hist* constructor, which, as discussed earlier, is the internal representation for the *reduce_by_index* SOAC. The assumptions of the special cases are matched first, where the operator and neutral element are the most obvious ones. This is then followed by the general case, which has some assumptions it needs to check. If any of these matches, the appropriate function is called.

These functions that compute the adjoint for `reduce_by_index` are all located in a file called *RBI.hs*, where there is a function for each of these cases. The special cases are mainly direct translations of the pseudocode in the same fashion as can be seen in figure 14. Therefore only snippets from the general case will be considered here.

The design process for all cases focused on creating modular code. This was done through a couple of different methods. The first one is that each line of the pseudocode was considered and implemented in isolation. This means that each new expression can be thought of as an individual function that assumes its input is valid and is implemented as close to the pseudocode as possible to ensure the validity of its output. This does, however, mean that the code is even more verbose than it could have been. For example the *screma* expressions could have been given fused *Lambda* functions or scans along with it. Instead, this implementation has a *screma* for each scan, map, and reduce performed. However, the simplification step after AD in the compiler takes care of this for the generated code, so it is not likely that it affects the performance of the code. Furthermore, there are utility functions for some of the bigger anonymous function bodies or lines that are used more than once.

The most complex part of the algorithm is the radix sort. This is implemented as a simple loop, bit shift and a custom written partition function. The pseudocode for this can be seen in appendix figure 19. The complexity came from how vital this part was and how many components it consisted of. Because of the modular approach discussed earlier, the implementation got a bit bloated. Thus, great care was taken to verify its output in almost every step of its development. The biggest missing feature also lies within this part of the program. The sorting algorithm has a constant loop size of 6 and therefore does not sort correctly for bucket sizes larger than 64. This also means that there is redundant sorting for any sizes less than 32. The loop size should be $\lceil \log_2(w) \rceil$ in the correct implementation.

```
let flags =
    map (\ind ->
      if 0 <= ind < w then 1
      else 0
    ) is
let flag_scanned =
    scan (+) 0 flags
let n' =
    last flags_scanned
```

(a) Source code

```
let {f_scan:[n]i64, flags:[n]i64} =
 scanomap(n, {is},
  {\ {x : i64, y : i64}: {i64} ->
   let {add_res : i64} = add64(x, y)
   in {add_res}, {0i64}},
  \{ind : i64}: {i64, i64} ->
    let {cond : bool} = ind < 0i64
    let {x : i64} =
      if cond then {0i64} else {
       let {cond : bool} = w < ind
       let {c_neg : bool} = not cond
       let {z : i64} = btoi c_neg
       in {z}}: {i64}
    in {x, x})
let {le : i64} = sub(n, 1i64)
let {new_length : i64} = f_scan[le]
```

(b) Generated code

```
i_param <- newParam "ind" $ Prim int64
pred_body <- runBodyBuilder . localScope (scopeOfLParams [i_param]) $ do
 eBody [ eIf
    (eCmpOp (CmpSlt Int64) (eParam i_param) (eSubExp zeroSubExp))
    (eBody [eSubExp zeroSubExp])
    (eBody [ eIf
        (eCmpOp (CmpSlt Int64) (eSubExp wsubexp) (eParam i_param))
        (eBody [eSubExp zeroSubExp])
        (eBody [eSubExp oneSubExp])
      ]) ]
let p_lam = Lambda [i_param] pred_body [Prim int64]
flags <- letExp "flags" $ Op $ Screma n [inds] $ ScremaForm [][] p_lam

scan_soac <- scanSOAC [Scan (addLambda (Prim int64)) [zeroSubExp]]
flags_scanned <- letExp "f_scan" $ Op $ Screma n [flags] scan_soac

lastElem <- letSubExp "le" $ BasicOp $ BinOp Sub n oneSubExp
n' <- letSubExp "new_length" $ BasicOp $ Index flags_scanned lastElem
```

(c) Compiler code

Figure 15: The three stages of the compiler code

A core idea of the pseudocode from the previous section is how the input arrays are filtered before the results and adjoints are computed. The size of the filtered arrays was computed by flagging all valid indices, scanning them with addition and extracting the last value. The compiler code for this can be seen in figure 15a. It works very similarly to the example from earlier. The anonymous function body can easily be defined by creating a parameter and using it in a simple nested if statement. This is then used in the Lambda for a Screma to create the first indicator flag array. A utility function for creating addition lambdas is used to create a scanSOAC for the scanned flags Screma easily. Finally, a subtraction and indexing operation is performed to have the final expression n'. The generated code in figure 15b shows an example of how the compiler can fuse operators after the AD step. In this case, the scan and map resulted in the scanomap fusion. This is a more efficient approach than performing them separately. In this case, the inputs to it are the size, the index, the scan lambda, and then the map lambda. Logically one can think of this as the map outputting two arrays, and then the scan is performed on only one of them afterwards. Due to this construct being implemented to run in parallel, it is not the reality.

Another example of a core part of the algorithm is how the adjoints are actually updated. The snippet of pseudocode where the adjoint for *as* is updated for the general case can be found in 16a. Here the adjoint of the operator is used to get the reverse mode AD for the lambda function $l_i \odot' a_i \odot' r_i$, the result of which is then scattered to an n-sized array constitutes the final contributions for this program. In figure 16c, the simplified compiler code equivalent to this can be found. Here an auxiliary function makes the adjoint lambda function for the operator and uses it in a *vjp* call. After this, the adjoint of the sorted values is scattered to a zero array which are the final contributions for *as*. In the generated code, the adjoint lambda creation and *vjp* call are not shown explicitly. Instead, the lambda function of the scatter computes this by simply multiplying the adjoints of the input with l_i and r_i. This is another case of fusion by the compiler.

The rest of the algorithm are snippets similar to figure 15 and figure 16. Most of the non-trivial choices were made in the development of the pseudocode discussed in the previous subsection.

```
let map_lam =
    (\li as vi ->
    op_adj (op_adj li as) vi)
let as_bar_contrib_r =
    vjp(map_lam,
    (lis, sorted_as', ris),
    s_bar_repl)
let as_bar_contrib =
    scatter
    n
    sorted_iota'
    as_bar_contrib_r

as_bar += as_bar_contrib
```

(a) Source code

```
let {as_bar_contrib_dst : [n]f32} =
  replicate([n], 0.0f32)
let {as_bar_contrib : [n]f32} =
  scatter(new_length,
    {sorted_is_bins_1, fwd_scan,
     rev_scan_rev,sorted_is_bins_2},
    \ {sorted_bin_p : i64,x_1 : f32,
       x_2 : f32, x_3 : i64}
      : {i64, f32} ->
      let {hist_temp_adj : f32} =
        hist_bar'[sorted_bin_p]
      let {binop_x_adj_1 : f32} =
        fmul32(x_1, hist_temp_adj)
      let {binop_y_adj_2 : f32} =
        fmul32(x_2, binop_x_adj_1)
      in {x_5940, binop_y_adj_2},
    (as_bar_contrib_dst)
in {as_bar_contrib}
```

(b) Generated code

```
(_, lam_adj) <- mkF op
vjp lam_adj [lis, sorted_as', ris] [hist_temp_bar_repl]

as_bar_contrib_r <- lookupAdjVal sorted_as'
as_bar_contrib_dst <-
 letExp "hist_orig_bar_contrib_dst" $ BasicOp $ Replicate n t_zero
f <- mkIdentityLambda [Prim int64, t]
as_bar_contrib <-
  letExp "as_bar_contrib" $ Op $
    Scatter n' [sorted_iota', as_bar_contrib_r] f as_bar_contrib_dst

updateAdj as as_bar_contrib
```

(c) Compiler code

Figure 16: The three stages of the compiler code

5.6 Shortcomings of the implementation

The implementation matches the ideal scenario with a single operator, neutral element and a single set of input arrays. However, since the compiler fuses throughout the pipeline, these assumptions may not be the reality. The implementation does not support input arrays that do not contain singletons—so having tuples or lists as the values in *as* is not supported in this implementation.

There are also 2d and 3d versions of the *reduce_by_index* construct, which are not supported by this implementation. Although they are represented similarly in the IR, the assumptions made in the implementation do not allow for them to be matched.

However, the biggest shortcoming is that the radix sort only sorts a constant amount of bits. For example, it only sorts the first 6 bits in this project, so any call with less than 64 unique bins will be correct. An alternative is setting this constant to 64 so that it returns the correct result no matter how many bins it has to take into account. This is much too inefficient to give a realistic code benchmark. The solution is relatively simple; one has to implement a snippet that computes how many bits need to be sorted, which is equal to $\lceil \log_2(w) \rceil$ where w is the number of bins in the output.

6 Validation

Since this is a compiler project, testing and validity are of the highest importance since subtle errors are incredibly easy to miss and can have devastating consequences at any time in the future. These extensions have their results reliant on much intermediate code, which has made it impossible to unit test through the source code. Therefore testing and verification were mainly done through testing the final result. The intermediate code was tested as thoroughly as possible using various techniques in the development phase. An example of this is how in the general algorithm, the radix sort was tested during the development by setting the result of the entire algorithm as the output of the sort. These manual tests are not robust enough to verify anything, but they exemplify how creative one must be when the code itself cannot be unit tested.

An assumption made very early on is that if nothing obvious proved the contrary, then every previously implemented part of the compiler must be correct. This allows for verification of the result of the *vjp* function by comparing it to the result of the *jvp* function which already supports *reduce_by_index*. Of course, because of the nature of these methods, the outputs will not give the same shaped output, but with some elementary maps and a transposition, they are expected to be equivalent. These cross-test cases include different types, shuffling of indices, and having indices out of bounds. Some of the smaller cases were also worked out by hand to be certain that the result of the *jvp* was correct.

Another method of verification that was used both in the development phase and after was generating the code for the algorithm using the `futhark dev -s` command. Constructing a test case that uses the algorithm we are interested in testing and giving the filename to this command gives us the generated intermediate code. Then, one can verify that this code corresponds to the actual code and the pseudocode for the algorithm. Of course, because of simplifications and fusions done by the pipeline, the result is not directly equivalent, so it takes quite a bit of time and effort to assess the validity of the code. This generated code is the same as was used in the code snippets in the previous section. For the longer algorithms, such as the general case, the generated code can be quite difficult to accurately verify because of how much the generated code deviates from the written counterpart.

The reason why the Haskell code is near impossible to unit test even though it is written in a modular method is that this code manipulates intermediate representations in monads where one would have to somehow construct the expected state of the ADM from scratch

to verify that the code up to that point is correct. This was decided to be outside the scope of this project since the result of the algorithm is what will be used in the end. Therefore, the main focus is on verifying that the algorithms' results are correct. This is mainly done with tests in which we know the outcome. For some of the tests, the outcome has been computed by hand. While for others, the already implemented forward mode is used to check against.

7 Benchmarks

This section will briefly present benchmarks of the different algorithms. The performance of the AD for *reduce_by_index* will be compared between the forward mode, reverse mode, and the special cases in the reverse mode.

The benchmarks were performed using the CUDA backend for Futhark on a home computer. This computer has 32GB of memory and an Nvidia GeForce 2080 Super. The benchmarks were performed on Windows Subsystems for Linux (WSL) because Futhark does not support native Windows. The input dataset was created using *futhark dataset*. Unless stated otherwise, the input dataset is non-zero 32-bit floats. There were no significant relative differences observed when using integers. The maximum size of the inputs is $5 \cdot 10^7$. After this size, CUDA was not able to allocate enough memory. All of the benchmarks are the meantime of 5 repetitions. There are two different types of benchmarks, one that derives the differentiated *hist_orig* and one which differentiates *as*. A baseline RBI without AD will also be included. Optimally a benchmark with the program differentiated with regard to both inputs would be included, but it was not possible to get it working for *jvp*. Since the general case for RBI is optimised to differentiate both arrays, this might be reflected in worse performance in the benchmarks.

Since *reduce_by_index* is not a construct available in other languages or libraries that support AD, the benchmarks are only within the Futhark compiler. An extension of this could be to implement the constructor just the reverse AD algorithm in a library like PyTorch, though this was not done in this project.

Important to note is that the general implementation for the reverse mode always performed six rounds of sorting since the radix sort was not implemented dynamically. For these sizes, that is much overhead if using less than 64 buckets. All of the benchmarks are performed with an output size of 64. This is not ideal since we are not able to vary the size of *hist_orig* enough to get an accurate difference in performance for the reverse-mode AD. However, even with the overhead the speedup of having one bin instead of 64 in the general case is 20%. So if the sorting were dynamic, there would be some more exciting benchmarks.

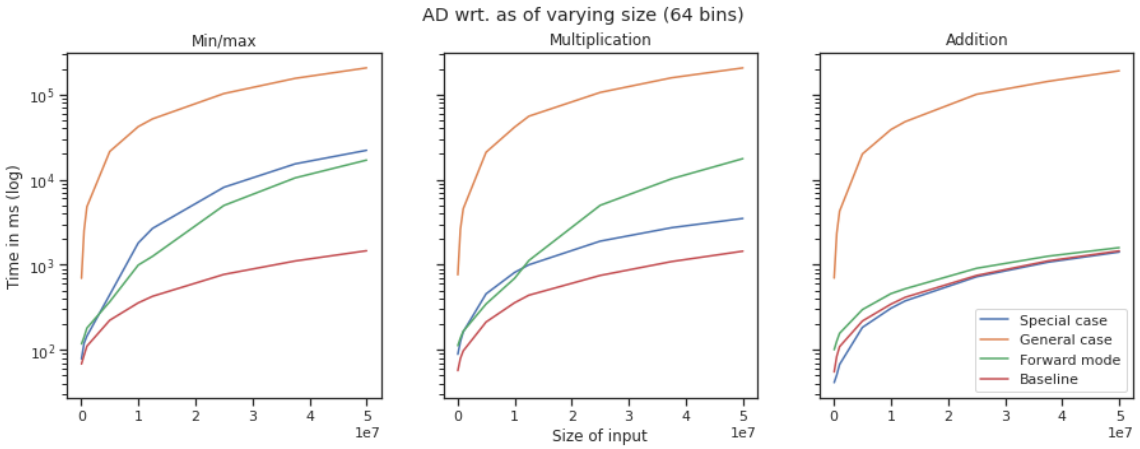

Figure 17: Benchmarks of AD wrt. *as*

In figure 17, the result of a benchmark of the different algorithms when differentiating wrt. *as* is shown. The time in milliseconds on the y-axis has been logarithmically scaled. It is very clear that the general case for the reverse mode is by far the slowest for all three operators. The forward mode also beats the special case for the min/max operator. However, the special cases outperforms the forward mode AD for multiplication and addition. For multiplication, it even nears the baseline without AD.

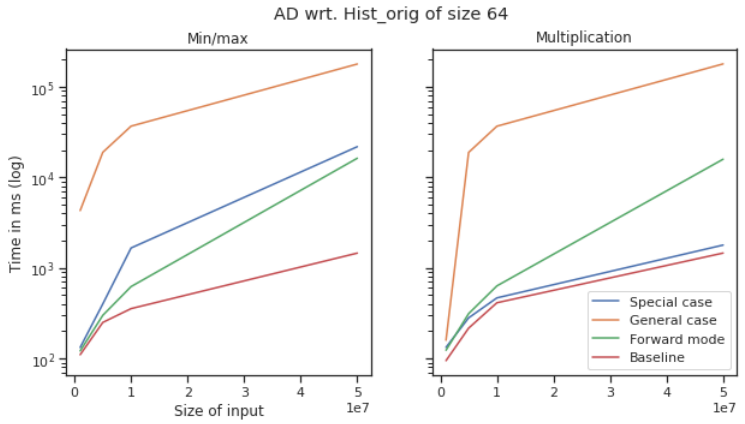

Figure 18: Benchmarks of AD wrt. *hist_orig*

In figure 18 the benchmark of AD wrt. to *hist_orig* is shown. The addition is missing

because the compiler recognizes that its result is simply the input adjoint, so it finishes computation in under a millisecond, no matter the size. The results for the other two operators are very similar to the previous benchmark. As mentioned earlier, the adjoint of *hist_orig* is tied in with the sorting and segmented scans in the general case because it is optimized to find both adjoints. This is probably where this poor performance stems from.

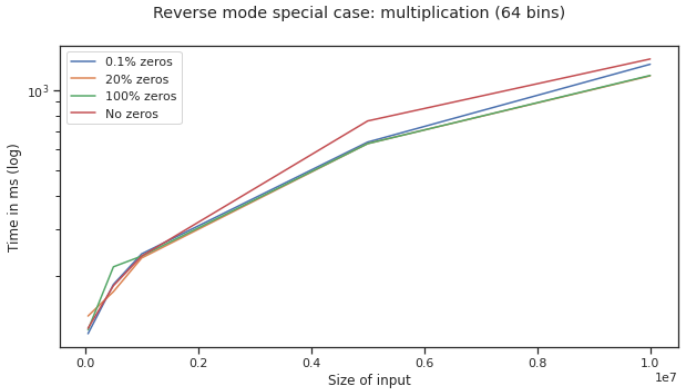

Figure 19: Benchmarks of mult. reverse mode AD wrt. *as*

In figure 19, we see the benchmark for the special case reverse mode AD of multiplication. This benchmark uses 64-bit integers as its input values. The general case was not used in this benchmark. It is only the special case algorithm. It showcases how there is up to 17% speedup when the input contains zeros.

In table 3 the AD overhead for the different cases is shown wr.t *hist_orig* and *as*.

	Hist_orig	as
Forward mode (mul)	122x	142x
General case (mul)	10.8x	12x
Addition	1x	15x
Multiplication	1.22x	2.4x
Min/Max	15x	15x

Table 3: AD overhead (AD-Time / Baseline-Time)

8 Conclusion & further work

This project has derived a rewrite rule for the reverse mode AD of the *reduce_by_index* construct. It has also attempted implementing it in the Futhark compiler, along with some special cases that were theorised to show significant speedups relative to this general case.

The implementations serve as suitable proof of concept, but the radix sort in the general case needs to be addressed to be considered finished. This is a minor addition but is missing from this project's implementation. An alternative is to set the loop number to 64, which would mean the reverse mode AD would be correct but inefficient. This would mean that the RBI construct would be fully supported and ready to use for programs needing reverse mode AD in Futhark. So even if it is the case that the forward mode is faster for the general case, it could still be an essential tool in the arsenal of programmers since no other libraries support this.

Furthermore, this project has introduced special cases for the most common operators that are incredibly efficient, and this could outweigh the ineffectiveness of the general case for many use cases. They also show how much can be won by deriving even more reverse mode special cases.

To better reason about the performance of these rewrite rules, better benchmarks are needed. The static nature of the general case stifles any accurate attempts at gauging the speedup when changing the number of bins. Furthermore, since the general case is optimised for finding the adjoints of both arrays, a better benchmarking would differentiate with regards to both arrays for both forward and reverse. Also, a baseline outside the Futhark language, for example, an implementation in PyTorch would be helpful for comparison.

9 Appendix

The code can be found at this GitHub link. It will be made private after the defence. Changes can be primarily found in src/Futhark/AD/Rev/RBI.hs

```
let partition2 =
  let bits_inv = map (\b -> 1 - b) bits
  let ps0 = scan (+) 0 (bits_inv)
  let ps0_clean = map2 (*) bits_inv ps0
  let ps1 = scan (+) 0 bits
  let ps0_offset = reduce (+) 0 bits_inv
  let ps1_clean = map (+ps0_offset) ps1
  let ps0_clean = map2 (*) bits_inv ps0
  let ps = map2 (+) ps0_clean ps1_clean'
  let ps_actual = map (-1) ps
  let scatter_inds = scatter inds ps_actual inds


let (sorted_is, sorted_bins) =
  loop over [new_indexes, new_bins] for i < Ceiling(log2(w)) 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)
```

Listing 19: Pseudo code for Radix sort

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

Listing 20: Pseudo code for segmented exclusive scan

References

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

Blelloch, Guy E. (Mar. 1996). "Programming Parallel Algorithms". In: *Commun. ACM* 39.3, pp. 85–97. ISSN: 0001-0782. DOI: 10.1145/227234.227246. URL: https://doi.org/10.1145/227234.227246.

Damas, Luis and Robin Milner (1982). "Principal Type-Schemes for Functional Programs". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: Association for Computing Machinery, pp. 207–212. ISBN: 0897910656. DOI: 10.1145/582153.582176. URL: https://doi.org/10.1145/582153.582176.

Developers, TensorFlow (May 2022). *TensorFlow*. Version v2.8.2. Specific TensorFlow versions can be found in the "Versions" list on the right side of this page.
See the full list of authors on GitHub. DOI: 10.5281/zenodo.6574269. URL: https://doi.org/10.5281/zenodo.6574269.

Elsman, Martin et al. (July 2018). "Static Interpretation of Higher-order Modules in Futhark: Functional GPU Programming in the Large". In: *Proc. ACM Program. Lang.* 2.ICFP, 97:1–97:30. ISSN: 2475-1421. DOI: 10.1145/3236792. URL: http://doi.acm.org/10.1145/3236792.

Henriksen, Troels, Sune Hellfritzsch, et al. (2020). "Compiling Generalized Histograms for GPU". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '20. Atlanta, Georgia: IEEE Press. ISBN: 9781728199986.

Henriksen, Troels, Niels G. W. Serup, et al. (June 2017). "Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates". In: *SIGPLAN Not.* 52.6, pp. 556–571. ISSN: 0362-1340. DOI: 10.1145/3140587.3062354. URL: https://doi.org/10.1145/3140587.3062354.

Leal, Allan M. M. (2018). *autodiff, a modern, fast and expressive C++ library for automatic differentiation*. https://autodiff.github.io. URL: https://autodiff.github.io.

Paszke, Adam et al. (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., pp. 8024–8035. URL: http://papers.neurips.cc/

paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Podobas, Artur (2015). "Improving Performance and Quality-of-Service through the Task-Parallel Model". In.

Schenck, Robert et al. (2022). *AD for an Array Language with Nested Parallelism*. DOI: 10.48550/ARXIV.2202.10297. URL: https://arxiv.org/abs/2202.10297.