Reverse-Mode AD of Reduce-by-Index and Scan in Futhark

Lotte Maria Bruun, Ulrik Stuhr Larsen, Nikolaj Hey Hinnerskov, Cosmin Oancea
xts194@alumni.ku.dk,usl@di.ku.dk,nihi@di.ku.dk,cosmin.oancea@di.ku.dk
University of Copenhagen
Department of Computer Science (DIKU)
Copenhagen, Denmark

ABSTRACT

We present and evaluate the Futhark implementation of reverse-mode automatic differentiation (AD) for the basic blocks of parallel programming: reduce, prefix sum (scan), and reduce by index. We first present derivations of general-case algorithms, and then discuss several specializations that result in efficient differentiation of most cases of practical interest. We report an experiment that evaluates the performance of the differentiated code in the context of GPU execution, and highlights the impact of the proposed specializations as well as the strengths and weaknesses of differentiating at high level *vs.* low level (i.e., "differentiating the memory").

ACM Reference Format:
Lotte Maria Bruun, Ulrik Stuhr Larsen, Nikolaj Hey Hinnerskov, Cosmin Oancea. 2023. Reverse-Mode AD of Reduce-by-Index and Scan in Futhark. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Nowadays, most domains have embraced machine learning (ML) methods, which fundamentally rely on gradients to learn. But even in the absence of ML, the computation of gradients is essential in many compute-hungry applications ranging over various domains, such as risk analysis of large portfolios utilizing complex pricing methods in finance [15, 37, 53], retrieval (and tuning) of parameters and associated uncertainties from satellite products in remote sensing [12, 29, 46], solving non-linear inverse problems or for sensitivity analysis of large numerical simulations in physics [28, 49].

Automating the computation of derivatives — known as automatic differentiation (AD) — has been a central contributor to facilitating advancements in such domains, for example in designing and training new ML models [3]. As such, an argument can be made that AD should be a first-class citizen in high-level parallel languages [22, 43]. This requires algorithms that offer reliable and efficient differentiation of (unrestricted) parallel programs that scales well on modern, highly parallel hardware, such as GPUs.

A feasible way of achieving this is to differentiate at a "high level", by building the AD algorithm around higher-order array combinators — common to functional programming — whose richer semantics allows to lift the level of abstraction at which the compiler reasons. The first step in this endeavor is to develop efficient rules for differentiating such parallel combinators.

This paper presents and evaluates algorithms for reverse-mode differentiation of reduce, reduce-by-index and scan, which are implemented in (but not restricted to the context of) the Futhark language [21]. For each combinator, we present a "general-case" algorithm that typically re-writes the differentiation of the combinator in terms of a less efficient combinator. For example, reduce requires prefix sum (scan), reduce-by-index require multi-scan, and scan's differentiation is not AD efficient. (The algorithms for reduce and reduce-by-index are AD efficient, but have largish constants).

These inefficiencies motivate the development of a set of specializations that significantly reduce the AD overheads of most cases of practical interest. Specializations include:

- addition, min, max and multiplication — these are known and are not claimed as contributions, but are treated here for completeness,
- vectorized operators, which are reduced to scalar operators by re-write rules that interchange the encompassing reduce(-by-index) or scan with the vectorizing map,
- invertible commutative operators that, for example, allow the differentiation of reduce(-by-index) to be written in terms of a reduce(-by-index) with an extended operator, rather than in terms of (multi-)scan,
- simple sparsity (compiler) optimizations that exploit the block-diagonal structure of Jacobians and, for example, allow differentiating a scan with 5×5 matrix multiplication operator at a reasonable 7× AD overhead.

Note that the differentiation of reduce has been (briefly) covered in [50] and is not a contribution of this paper; we still recount it in detail for completeness and because the rationale behind it drives the treatment of the other operators.

Finally and most importantly, we report an experiment that evaluates the practical GPU performance of the reverse-mode differentiation of reduce, scan and reduce by index on various operators. Comparisons are made with algorithms of the closest-related approach [44], which we have implemented in Futhark. The evaluation demonstrates significant performance gains, and that most operators can be differentiated on GPU quite efficiently, while very few of them (e.g., reduce-by-index with saturated addition) appear better suited to lower-level approaches that "differentiate the memory".

In summary, key contributions of this paper are:

1 "general-case" reverse-mode AD algorithms for reduce-by-index and scan, the former of which is AD efficient,
2 a set of specializations that offer practical efficiency for most cases of interest,

Conference'17, July 2017, Washington, DC, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
1  P($x_0$,  $x_1$):
2    $t_0$ = sin($x_0$)
3    $t_1$ = $x_1$ · $t_0$
4    $y$ = $x_0$ + $t_1$
5    return $y$
```

```
1   P'($x_0$,  $x_1$):
2     $t_0$ = sin($x_0$)
3     $t_1$ = $x_1$ · $t_0$
4     $y$ = $x_0$ + $t_1$
5     $\overline{y}$ = 1
6     $\overline{x_0}$ = 0, $\overline{t_1}$ = 0
7     $\overline{x_0}$ += 1 · $\overline{y}$       -- $\frac{\partial(x_0+t_1)}{\partial x_0} \equiv 1$
8     $\overline{t_1}$ += 1 · $\overline{y}$       -- $\frac{\partial(x_0+t_1)}{\partial t_1} \equiv 1$
9     $\overline{x_1}$ = 0, $\overline{t_0}$ = 0
10    $\overline{x_1}$ += $t_0$ · $\overline{t_1}$      -- $\frac{\partial(x_1 \cdot t_0)}{\partial x_1} \equiv t_0$
11    $\overline{t_0}$ += $x_1$ · $\overline{t_1}$      -- $\frac{\partial(x_1 \cdot t_0)}{\partial t_0} \equiv x_1$
12    $\overline{x_0}$ += cos($x_0$) · $\overline{t_0}$
13                                -- $\frac{\partial \sin(x_0)}{\partial x_0} \equiv$ cos($x_0$)
14    return ($\overline{x_0}$, $\overline{x_1}$)
```

Figure 1: Simple example demonstrating reverse-mode differentiation of straight-line scalar code.

3 to our knowledge, the first evaluation of the GPU performance of reverse-mode AD for reduce(-by-index) and scan; the evaluation demonstrates the claims and the impact of the proposed specializations and highlights the strengths and weaknesses of the approach of differentiating at a high level.

2 PRELIMINARIES

This section provides a brief overview of the functional language used to discuss the differentiation algorithms and a brief introduction to reverse-mode automatic differentiation (AD), which are hopefully sufficient to understand the rest of the paper.

2.1 Brief Overview of the Futhark Language

Futhark is a purely-functional parallel-array language that borrows its syntax from a combination of Haskell and ML, and in which parallelism is explicitly expressed by means of a nested composition of standard second-order array combinators (SOAC), such as map, reduce, scan, and scatter (parallel write). Scatter has type:

$$\texttt{\textbf{scatter}} \;:\; \forall n, m, \alpha.\ *[n]\alpha \;\rightarrow\; [m]\texttt{\textbf{i64}} \;\rightarrow\; [m]\alpha \;\rightarrow\; *[n]\alpha$$

where **`i64`** denotes the 64-bits integral type, $[n]\alpha$ denotes a size-typed [16] array of length n, and $*[n]\alpha$ denotes a unique type, e.g., when used as an argument it means that the corresponding array is consumed by the scatter operation, and when used for the result it means that it does not alias any of the non-unique arguments.

Semantically, scatter "updates in place" the first array argument at the indices specified in the second array with the corresponding values stored in the third argument. Scatter has work $O(m)$ and depth $O(1)$. The other combinators are standard.

For brevity, the notation used in this paper is informal and omits universal quantification and types, whenever they are easily inferable by the reader.

2.2 Brief Introduction to Reverse-Mode AD

The first-order partial derivatives of a differentiable function $f : \mathbb{R}^n - > \mathbb{R}^m$ forms an $m \times n$ Jacobian matrix:

$$\mathrm{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Reverse mode AD computes a row of the Jacobian at a time. (In contrast, forward mode AD computes a column of the Jacobian at at time. Reverse mode AD is thus preferred when the size of the result is much smaller than the input size.)

For a program $P(\ldots, x_i, \ldots) = y \in \mathbb{R}$, reverse-mode AD computes the *adjoint* of each (intermediate) program variable t, denoted $\overline{t} = \frac{\partial y}{\partial t}$, that captures the sensitivity of the result to changes in t.

The *primal trace* (original program) is first executed to save intermediate program values on a *tape* (abstraction). The tape is subsequently used by the *return sweep*, which computes the *adjoint* of each variable in reverse program order.

Initially $\overline{y} = \frac{\partial y}{\partial y} = 1$, and eventually the adjoints of the input $\overline{x_i} = \frac{\partial y}{\partial x_i}$ are computed by applying the core re-write rule:

$$v \;=\; f(a, b, \ldots) \;\Longrightarrow\; \begin{array}{l} v = f(a, b, \ldots) \\ \vdots \\ \overline{a} \mathrel{+}= \dfrac{\partial f(a, b, \ldots)}{\partial a}\overline{v} \\ \overline{b} \mathrel{+}= \dfrac{\partial f(a, b, \ldots)}{\partial b}\overline{v} \\ \ldots \end{array} \tag{1}$$

where the vertical dots correspond to the statements of the primal trace and their differentiation (on the return sweep). In particular, the vertical dots compute the final value of $\overline{v}$ – because v cannot possibly be used before its definition — and partial values for $\overline{a}$ and $\overline{b}$, i.e., corresponding to their uses after the statement $v = f(a, b, \ldots)$ in the original program.

Figure 1 demonstrates how reverse-mode AD is applied to a simple example consisting of straight-line scalar code. The left-hand side shows the original program. The right-hand side shows the differentiated code:

lines 2-4: the primal trace is re-executed to bring into scope the values of t_0 and t_1 which are used in differentiation;

line 5: the adjoint of the result is initialized to $\frac{\partial y}{\partial y} = 1$;

lines 6-8 correspond to differentiating the last statement of the original program (line 4) by the application of re-write rule 1. Note that adjoints are initialized before their first use.

lines 9-11 and 12-13 similarly correspond to the differentiation of statements at lines 3 and 2 in the original program.

Reverse-mode AD is exposed to the user by means of the classical *vector-Jacobian product* (**`vjp`**) interface:

$$\texttt{\textbf{vjp}} : (f : \alpha \rightarrow \beta) \rightarrow (x : \alpha) \rightarrow (\overline{y} : \beta) \rightarrow \alpha$$

`vjp` is a second-order function that computes the derivative of f at point x given that the adjoint of the result is $\overline{y}$. For example, the Jacobian of f at point x can be computed by mapping **`vjp f x`** on the unit vectors of the result type β, i.e., each invocation of **`vjp`**

computes one row of the Jacobian. The notation in 1 is a bit confusing: since **vjp** computes the vector-Jacobian product, it should probably be written as $\overline{a}\ \ += \ \overline{v}\ \frac{\partial f(a,b,\ldots)}{\partial a}$, as it makes a difference when v is a vector. We warn the reader that the rest of the paper will use the established, albeit slightly confusing, notation.

In the paper, we will denote by $\mathbb{VJP}$ the program transformation (hinted in figure 1) that implements reverse-mode AD and we will use $\mathbb{VJP}^{\mathbb{LAM}}$ to denote the code transformation applied to the syntactic category of lambda functions. (Other syntactic categories are, for example expressions, statements and body of statements).

3 PRELIMINARIES: REVERSE-AD OF REDUCE

This section presents the algorithm (re-write rules) that implements reverse-mode differentiation of **reduce**: Section 3.1 presents the "general" case, in which the reduce operator is merely constrained to not use any free variables, Section 3.2 specializes the algorithm to commonly-used operators addition, min/max, multiplication and to vectorized operators, and Section 3.3 specializes the algorithm to a class of operators that are commutative and (left-)invertible.

The goal of this section is to make the presentation self-contained, since the rest of the paper builds on the type of reasoning used for reduce. The content of this section is not a scientific contribution of this paper because, for example, the general rule and specialized cases were briefly presented elsewhere [50] and the specialization proposed in Section 3.3 is not implemented yet.

3.1 General Algorithm

We start deriving the general rule from the definition of **reduce**. Given an associative operator $\odot$ with neutral element $e_\odot$, we have

$$y = \textbf{reduce}\ \odot\ e_\odot\ [a_0, a_1, \ldots, a_{n-1}]$$

which is equivalent to

$$y = a_0 \odot a_1 \odot \ldots \odot a_i \odot \ldots \odot a_{n-1}$$

For each a_i, we can then group the terms of the reduce as:

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

Assuming l_i and r_i are known (i.e., already computed) and that $\odot$ does not use any free variable then we can directly apply the core rule for reverse AD, given in equation 1, to compute all the contributions to the adjoints $\overline{a_i}$ in parallel:

$$\overline{a_i}\ \ \overline{+}=\ \frac{\partial(l_i \odot a_i \odot r_i)}{\partial a_i}\ \overline{y} \tag{2}$$

where $\overline{y}$ — the adjoint of the reduction result y — has aleardy been determined by execution of the return sweep until this point.

Here, l_i and r_i for all $i = 0, \ldots, n-1$ can be computed by two exclusive scans; one on the original array and one on the reversed array. Exclusive scan is defined as:

$$\textbf{scan}^{exc} \odot e_\odot\ [a_0, a_1, \ldots, a_{n-1}]\ \equiv\ [e_\odot,\ a_0,\ a_0{\odot}a_1, \ldots, a_0{\odot}\ldots a_{n-2}]$$

Figure 2 shows the general-case algorithm for reduction, where

- ▹ is the *pipe operator* which composes functions left to right,
- $\overline{+}$ is a multidimensional addition (implemented as a tower of **map**s) with rank equal to the rank of the input array, and

```
-- Assuming array as of length n
-- Primal trace is the same as original:
let y = reduce ⊙ $e_\odot$ as

-- Return sweep:
let ls = scan$^{exc}$ ⊙ $e_\odot$ as   -- forward exclusive scan
let rs = reverse ad          -- reverse exclusive scan
       ▹ scan$^{exc}$ ($\lambda x\ \ y \to y\ \ \odot\ \ x$) $e_\odot$
       ▹ reverse
    -- applying the core rule of reverse AD:
let $\overline{as}$ $\overline{+}$= map3 ($\lambda\ l_i\ a_i\ r_i \to \overline{f_{l_i,r_i}}\ a_i$) ls as rs
    denoting $\overline{f_{l_i,r_i}}\ \leftarrow\ \mathbb{VJP}^{\mathbb{LAM}}\ (\lambda a_i \to l_i \odot a_i \odot r_i)\ \overline{y}$
```

Figure 2: Reverse AD general re-write rule for reduce

- $\overline{f_{l_i,r_i}}$ is recursively obtained by applying the $\mathbb{VJP}^{\mathbb{LAM}}$ transform to generate code corresponding to $\partial(l_i \odot a_i \odot r_i)/\partial a_i \cdot \overline{y}$.

We observe that the transformation preserves the parallel asymptotic of the original program — since both **reduce** and **scan** have linear work and logarithmic depth. However, it incurs a rather large AD overhead since, even under aggressive fusion, the differentiated code still requires about 8× more global-memory accesses than the original reduce. Fortunately, standard operators, discussed next, admit more efficient differentiation.

3.2 Specialization for Common Operators

Specialized rules for addition, min, max and multiplication on numeric types are known [23] — the rule of multiplication in [23] uses two **scan**s, but a more efficient one is presented in [50]. This section recounts them in more detail for completeness.

3.2.1 Addition. The primal trace of `let y = reduce (+) 0 as` is the same as the original and its return sweep adds $\overline{y}$ to each element of $\overline{as}$ since $\partial(l_i+a_i+r_i)/\partial a_i$ simplifies to 1, hence the return sweep is:

$$\texttt{let}\ \overline{as} = \textbf{replicate}\ n\ \overline{y}\ \ \triangleright\ \ \textbf{map2}\ (+)\ \overline{as}$$

We expect the AD overhead to be as high as 3× since the original code performs n reads from global memory, and the differentiated code performs $3 \cdot n$ accesses: $2 \cdot n$ reads and n writes.[1]

3.2.2 Min/Max. A reduction with min (max) selects the minimum (maximum) element of an array. Assume that the latter is located at position k. It follows that the contribution to the adjoint of a_i is:

- 0 for any $i \neq k$ because the result y does not depend on a_i,
- $\overline{y}$ for the k^{th} element, since $\frac{\partial min(a_0,\ldots,a_k,\ldots,a_{n-1})}{\partial a_k} = \frac{\partial a_k}{\partial a_k} = 1$.

The primal trace is thus a lifted reduction whose associative and *commutative* operator, denoted $\texttt{min}^L$, keeps track of the minimum value together with its index — in case of duplicates we choose the smallest index corresponding to the minimum value:

$$\texttt{let}\ (k, y) = \textbf{zip}\ [0, \ldots, \texttt{n}-1]\ \texttt{as}\ \triangleright\ \textbf{reduce}\ \texttt{min}^L\ (\texttt{n},\ \infty)$$

The return sweep updates (only) the adjoint at position k:

$$\texttt{let}\ \overline{a_{min}} = \textbf{if}\ k < n\ \textbf{then}\ \overline{as}[k] + \overline{y}\ \textbf{else}\ 0_\alpha$$
$$\texttt{let}\ \overline{as} = \textbf{scatter}\ \overline{as}\ [\ \texttt{k}\ ]\ [\ \overline{a_{min}}\ ]$$

[1] The `reduce` of the primal performs n reads, and the `map` of the return sweep performs n reads and n writes: The `replicate` is fused with the `map` and thus not considered, but the `reduce` of the primal cannot be fused since its result `y` is used in the `map`.

```
-- Original:
let y = reduce (*) 1 as

-- Primal trace:
let ($n^{=0}$, $y^{>0}$) =
  as ▹ map (λa → if a==0 then (1i64,1) else (0,a))
     ▹ reduce (+, *) (0i64, 1)
let y = if $n^{=0}$ > 0 then 0 else $y^{>0}$

-- Return sweep:
let $\overline{as}$ = map2 (λ a $\overline{a}$ →
                    $\overline{a}$ + if $n^{=0}$ == 0
                        then (y / a) * $\overline{y}$
                        else if $n^{=0}$ == 1 && a == 0
                             then $y^{>0}$ * $\overline{y}$
                             else 0
                ) as $\overline{as}$
```

Figure 3: Reverse AD rule for reduce with multiplication

The update is implemented in terms of the parallel-write operator **scatter**, which has the semantics that it discards the updates of out of bounds indices (e.g., the case when as is empty).

We expect the AD overhead to be around 2×: because the original and primal both perform n memory reads to compute the reduction — the index space $[0, \ldots, n-1]$ is fused, hence not manifested in memory — and the return sweep might need to initialize $\overline{as}$ (with zeroes), which requires another n memory writes. GPU implementations may suffer from (expensive) host-to-device transfers if scalars and arrays are kept in the CPU and GPU memory space, respectively, i.e., the values of k and $\overline{as}[k]$ have to be brought from GPU to CPU, and the value of $\overline{a_{min}}$ has to be transferred back to GPU.

3.2.3 *Multiplication.*

The quantity of interest is

$$\frac{\partial y}{\partial a_i} = \frac{\partial (l_i \cdot a_i \cdot r_i)}{\partial a_i} = l_i \cdot r_i$$

If all elements would be known to be different than zero, then $l_i \cdot r_i$ can be computed as y/a_i, resulting in the return sweep:

$$\texttt{let}\ \overline{as} \mathrel{+}= \texttt{map}\ (\lambda\ a_i\ \rightarrow\ (y/a_i)\ *\ \overline{y})\ as$$

The case when some elements may be zero is treated by extending the primal to compute (i) the number of zero elements $n^{=0}$ and (ii) the product of the non-zero elements $y^{>0}$. Two additional cases require consideration:

- If exactly one element at index i_0 is zero, then $l_i * r_i$ is zero for all other elements and only $\overline{a_{i_0}}$ is updated: $\overline{a_{i_0}} \mathrel{+}= y^{>0} * \overline{y}$.
- If more than one zero exists, then $\overline{as}$ remains unchanged.

Figure 3 shows the code that implements this algorithm. The reason for moving the **if** inside the **map** is to permit utilization of outer(-map) parallelism, in case it exists (otherwise the introduced control flow might prevent it). Assuming that in the common case as does not contain zeroes, the AD overhead can be as high as:

3× if $\overline{as}$ is initialized at this point (i.e., lastly used in the original reduce), because its initialization will be fused with the **map**,

4× otherwise, i.e., $2 \cdot n$ reads and n writes due the **map** on the return sweep and n reads due to the **reduce** of the primal.

3.2.4 *Vectorized Operators.*

Vectorized operators are transformed with IRWIM re-write rule [19] that essentially interchanges the outer reduce inside the inner map, thus simplifying the reduce operator:

$$\begin{array}{c} \texttt{reduce}\ (\texttt{map2}\ \odot)\ (\texttt{replicate}\ \texttt{n}\ e_\odot)\ \texttt{matrix} \\ \equiv \\ \texttt{map}\ (\texttt{reduce}\ \odot\ e_\odot)\ (\texttt{transpose}\ \texttt{matrix}) \end{array} \qquad (3)$$

The intuition is that summing up the elements on each column of a matrix (**reduce (map2 (+))**) is equivalent to transposing the matrix, and summing up the elements on each row (**map (reduce (+))**). This is generalized to arrays of arbitrary rank by extending **transpose** to permute the two outermost dimensions.

The IRWIM rule is systematically applied in the $\mathbb{VJP}$ transformation whenever it matches and differentiation is performed on the *resulting code*. This is practically important because:

- it enables efficient differentiation of vectorized common operators without extra implementation effort;
- some differentiation rules [44] do not turn reductions with vectorized operators into scans or reductions with vectorized operators — such (arbitrary) operators on arrays are challenging to be mapped efficiently to the GPU hardware, e.g., the code produced by Futhark has abysmal performance.

3.3 Specialization for Invertible Operators

The treatment of multiplication hints that the underlying reasoning can be extended to encompass a larger class of operators. The key properties that we have used are

commutativity: in that $l_i \cdot a_i \cdot r_i$ can be rewritten as $l_i \cdot r_i \cdot a_i$,
invertibility: in that knowing the current element a_i, the total number of zeros $n^{=0}$ and the product of non-zero elements $y^{>0}$, one can uniquely compute $l_i \cdot r_i$.

Similar to the work on near-homomorphisms [13], we also observe that some non-invertible operations (such as multiplication) can be inverted by extending them to compute a (small) baggage of extra information. This extra information makes the operator injective and hence left invertible.

We propose to extend the language to allow the user to connect an associative and commutative operator $\odot : \alpha \rightarrow \alpha \rightarrow \alpha$ with

- its lifted associative, commutative and (left-)invertible operator $\odot^L : \beta \rightarrow \beta \rightarrow \beta$ and its (left) inverse $\odot^L_{inv} : \beta \rightarrow \beta \rightarrow \beta$.[2]
- a pair of functions that convert between α and β, denoted $f^{>}_{\odot} : \alpha \rightarrow \beta$, and $f^{<}_{\odot} : \beta \rightarrow \alpha$.

The properties that the user must ensure to hold are:

(1) $f^{<}_{\odot}\ \circ\ (\texttt{reduce}\ \odot^L\ e_{\odot^L})\ \circ\ (\texttt{map}\ f^{>}_{\odot})\ \equiv\ \texttt{reduce}\ \odot\ e_\odot$
(2) for sanity, $e_{\odot^L}$ must belong to the co-domain of $f^{>}_{\odot}$,
(3) $\forall\ a, b$ if $z = a\ \odot^L\ b$ then it holds that $a\ =\ z\ \odot^L_{inv}\ b$ and $b\ =\ z\ \odot^L_{inv}\ a$.

It can be derived from (1) and (2) that $e_{\odot^L} \equiv f^{>}_{\odot}(e_\odot)$, that $e_\odot \equiv f^{<}_{\odot}(e_{\odot^L})$ and that $f^{<}_{\odot} \circ f^{>}_{\odot} \equiv id$.

[2] We argue that this extension is reasonable since parallel languages commonly *assume* that the operators of reduce and scan are associative, and that those of reduce-by-index are also commutative. In fact, verification of such properties is undecidable in general.

The lifted operators and conversions for multiplication are:

$$\begin{aligned}(n_1^{=0},\ y_1^{>0})\ \cdot^L\ (n_2^{=0},\ y_2^{>0}) &= (n_1^{=0}+n_2^{=0},\ y_1^{>0}\cdot y_2^{>0})\\ (n_1^{=0},\ y_1^{>0})\ \cdot^L_{inv}\ (n_2^{=0},\ y_2^{>0}) &= (n_1^{=0}-n_2^{=0},\ y_1^{>0}/y_2^{>0})\end{aligned}$$

$$\begin{aligned}f_\cdot^{>}\ (a_i) &= \texttt{if}\ (a_i == 0)\ \texttt{then}\ (1,1)\ \texttt{else}\ (0,a_i)\\ f_\cdot^{<}\ (n^{=0},y^{>0}) &= \texttt{if}\ (n^{=0} == 0)\ \texttt{then}\ y^{>0}\ \texttt{else}\ 0\end{aligned}$$

Instead of differentiating **let** y = **reduce** $\odot$ $e_\odot$ as, the $\mathbb{VJP}$ transform would be applied to the semantically equivalent code:

$$\begin{aligned}&\texttt{let}\ as^L = \texttt{map}\ f_\odot^{>}\ as\\ &\texttt{let}\ y^L = \texttt{reduce}\ \odot^L\ e_{\odot^L}\ as^L\\ &\texttt{let}\ y = f_\odot^{<}\ y^L\end{aligned} \tag{4}$$

and the code generation of the return sweep for the (middle) reduce statement will exploit the commutativity and invertibility of $\odot^L$:

```
let $\overline{as^L}$ $\overline{+}$=
  map2 ($\lambda$ $a^L$ → let $b^L$ = $y^L$ $\odot^L_{inv}$ $a^L$ in $\overline{\odot_{b^L}}^L$ $a^L$) $as^L$
    denoting $\overline{\odot_{b^L}}^L$ ← $\mathbb{VJP}^{\mathrm{LAM}}$ ($\lambda$ $x$ → $b^L$ $\odot^L$ $x$) $\overline{y^L}$
```

where $\overline{\odot_{b^L}}^L$ implements the differentiation of $y^L = b^L \odot^L a^L$ with respect to a^L according to core re-write rule 1, i.e.,

$$\overline{\odot_{b^L}}^L\ (a^L)\ \equiv\ \frac{\partial(b^L \odot a^L)}{\partial a^L}\cdot \overline{y^L}$$

An interesting example of associative, commutative and invertible operator that we are unaware to have been previously reported is:

$$\begin{aligned}(p_1,\ s_1)\ \text{`sumOfProd`}\ (p_2,\ s_2) &= (p_1+p_2+s_1\cdot s_2,\ s_1+s_2)\\ (p,s)\ \text{`sumOfProd`}_{inv}\ (p_2,\ s_2) &= (p-p_2-(s-s_2)\cdot s_2,\ s-s_2)\end{aligned}$$

For example, $\Sigma_{0\le i<j<n}(a_i\cdot a_j)$ — where a_i and a_j denote different elements of an array as of length n — can be computed with:

$$\texttt{map}\ f^{>}_{\text{sumOfProd}}\ as\ \triangleright\ \texttt{reduce}\ \text{sumOfProd}\ (0,0)\ \triangleright\ f^{<}_{\text{sumOfProd}}$$

and efficiently differentiated, as shown above. The conversions of the corresponding *near-homomorphism* are:

$$\begin{aligned}f^{>}_{\text{sumOfProd}}\ (a_i) &= (0,a_i)\\ f^{<}_{\text{sumOfProd}}\ (p,\ \_) &= p\end{aligned}$$

4 REVERSE-AD FOR REDUCE-BY-INDEX

We recall that reduce-by-index, a.k.a., multi-reduce, is a second-order array combinator that generalizes a histogram computation [17]: it reduces the values falling in the same bin with an arbitrary associative and commutative operator $\odot$, having neutral element $e_\odot$. Its type and sequential/imperative semantics are:

```
def reduce_by_index ∀ w,n.
            (hist: *[w]α) (⊙:α → α → α) (e⊙ : α)
            (ks:  [n]int) (vs:  [n]α) : *[w]α =
  for i = 0..n-1 do
      key = ks[i]
      if 0 <= key && key < w
        hist[key] = hist[key] ⊙ vs[i]
  return hist
```

```
-- Assuming vs & ks of length n, and xs of length w
-- Primal trace is the same as original:
let xs = reduce_by_index (replicate w $e_\odot$)
                         ⊙ $e_\odot$ ks vs
-- Return sweep:
let (sks,svs,siota) = zip3 ks vs [0..n-1]
                    ▹ radixSortByFirst
                    ▹ unzip3
let flag_fwd = [0 .. n-1]
             ▹ map ($\lambda$i → i==0 || sks[i-1]!=sks[i])
let flag_rev = [0 .. n-1]
             ▹ map ($\lambda$i → i==0 || flag[n-i])
let ls = seg_scan$^{exc}$ ⊙ $e_\odot$ flag_fwd svs
let rs = reverse svs
       ▹ seg_scan$^{exc}$ ⊙ $e_\odot$ flag_rev
       ▹ reverse
let $\overline{\texttt{svs}}$ =
  map4 ($\lambda k_i$ $v_i$ $l_i$ $r_i$ →
         if $k_i$ < 0 || $k_i$ >= w
         then $0_\alpha$ -- the zero of the element type $\alpha$
         else $\overline{f_{l_i,r_i}}$ $v_i$
           denoting $\overline{f_{l_i,r_i}}$ ← $\mathbb{VJP}^{\mathrm{LAM}}$ ($\lambda x \rightarrow l_i \odot x \odot r_i$) $\overline{xs}[k_i]$
       ) sks svs ls rs
-- scratch creates an uninitialized array.
let $\overline{\texttt{vs}}$ = scatter (scratch $\alpha$ n) siota $\overline{\texttt{svs}}$
       ▹ map2 ($\overline{+}$) $\overline{\texttt{vs}}$
```

Figure 4: Reverse AD re-write rule for reduce_by_index

To simplify the reverse-mode AD transformation we systematically re-write **let** statements of the kind:

```
let hist = reduce_by_index hist₀ ⊙ e⊙ ks vs
```

into the semantically-equivalent code:

```
let xs = reduce_by_index (replicate w $e_\odot$)
                         ⊙ e⊙ ks vs
let hist = map2 ⊙ hist₀ xs
```

that always applies **reduce_by_index** to an initial histogram consisting only of neutral elements $e_\odot$. In this form, the adjoints of the initial histogram dst are decoupled from the **reduce_by_index**, i.e., they are updated by the differentiation of the **map2**. This re-write is reasonable because the design of **reduce_by_index** is built on the assumption that the histogram length is (significantly) smaller than the length of the input.[3]

4.1 General Case

We start by observing that reduce-by-index accepts a data-parallel work-efficient $O(n)$ implementation obtained for example by (radix) sorting the key-value pairs according to the keys (i.e., the indices in ks), and then by applying a segmented reduce to sum up (with $\odot$) each segment, where a segment corresponds to the (now consecutive) elements that share the same key value. A direct approach would be to apply the re-write above and differentiate the resulted code. The rationale for not taking this simple(r) path is because our differentiation of scan — which appears in the implementation of segmented reduce — is not work preserving (see Section 5.2).

[3] An asymptotic preserving re-write would be to replace the replicate with a scatter that writes $e_\odot$ only at the positions corresponding to the set of (unique) indices of ks. Similarly, map can be replaced with a gather-scatter that updates only those positions.

Instead, the path we take builds on the one used in the general case of reduce. Adapting equation 2 to reduce-by-index results in:

$$\overline{v_i} \;\overline{+}= \frac{\partial\,(l_i \;\odot\; v_i \;\odot\; r_i)}{\partial\, v_i} \cdot \overline{\mathtt{x}}_{k_i} \tag{5}$$

where v_i corresponds to vs[i], l_i and r_i correspond to the forward and reverse partial sums (by $\odot$) of elements up to position i that have the same key k_i as element v_i, and $\overline{\mathtt{x}}_{k_i}$ is the adjoint of element at index k_i in the resulted histogram. It follows that the implementation of 5 requires a multi-scan for computing l_i and r_i within the segment corresponding to elements sharing the same key k_i. This is commonly achieved by (radix) sorting the key-value pairs according to the keys. Figure 4 shows the asymptotic-preserving re-write rule that implements the reverse-mode differentiation:

lines 7-9 use a data-parallel implementation of radix sort ($O(n)$ work) to sort the key-value pairs (sks, svs) according to the keys; the implementation only sorts $[0, \ldots, n-1]$ according to keys, and then gathers vs according to the resulted siota.

lines 10-11 create the flag array that semantically partitions the sorted values (svs) into segments, such that all elements of a segment share the same key — a **true** value in the flag array correspond to the start of a segment.

line 14 performs a segmented scan, i.e., computes the (forward) prefix sum with operator $\odot$ for each segment.

lines 12-13 and 15-17 compute in a similar way the reverse scan for each segment.

lines 19-28 compute the adjoint contribution for each element of vs (in sorted order): if the index is out of the histogram bounds then the adjoint contribution is the zero value of the element type; otherwise the $\mathbb{VJP}$ code transformation is applied to generate the code for differentiating the function $\lambda\, x \;\rightarrow\; l_i \;\odot\; x \;\odot\; r_i$ at point v_i given the adjoint of the result $\overline{xs}[k_i]$ (where k_i denotes the key of v_i).

lines 32-33: the adjoint contributions are permuted to match the original ordering (**scatter**) and added to the currently-known adjoint of vs (which corresponds to uses of vs after the **reduce_by_index** operation).

The shown algorithm is asymptotic preserving — all operations have work $O(n)$ — but it incurs a rather large (constant) overhead due to sorting. Reasons are twofold: (1) a sorting-approach to implementing generalized histograms has been shown to be several times slower than Futhark's **reduce_by_index**, even when using the state-of-the-art implementation of CUB library, and (2) Futhark's radix sort implementation we are using is between one-to-two order of magnitude slower than CUB's.

4.2 Specialized Rules

The reasoning used for the specialized cases of reduce also extends to reduce-by-index.

4.2.1 Addition. Specializing general rule 5 to addition results in:

$$\overline{v_i} \mathrel{+}= \frac{\partial\,(l_i \;+\; v_i \;+\; r_i)}{\partial\, v_i} \cdot \overline{\mathtt{x}}_{k_i} \quad\Rightarrow\quad \overline{v_i} \mathrel{+}= \overline{\mathtt{x}}_{k_i}$$

It follows that the primal remains identical with the original:

```
let xs = reduce_by_index (replicate w 0) (+) 0 ks vs
```

```
let min$^L$ ($k_1$, $v_1$) ($k_2$, ($v_2$) =
  if      $v_1$ < $v_2$ then ($k_1$, $v_1$)
  else if $v_1$ > $v_2$ then ($k_2$, $v_2$)
  else (min $k_1$ $k_2$, v1)

-- Assuming vs & ks of length n, and xs of length w
-- Original for min operator:
let rep$_\infty$ = replicate w $\infty$
let xs = reduce_by_index rep$_\infty$ min $\infty$ ks vs
-- Primal trace for min operator:
let rep$_{(n,\infty)}$ = replicate w (n, $\infty$)
let (is$_{min}$,xs)= reduce_by_index rep$_{(n,\infty)}$ min$^L$ (n, $\infty$)
                           ks (zip [0,...n-1] vs) ▹ unzip
-- Return sweep for min operator:
let $\overline{vs}$ = map2 ($\lambda$ i$_{min}$ $\overline{x}$ $\rightarrow$ if $i_{min}$ >= n then 0$_\alpha$
                              else $\overline{vs}$[i$_{min}$] + $\overline{x}$
                 ) is$_{min}$ $\overline{xs}$
      ▹ scatter $\overline{vs}$ is$_{min}$
```

Figure 5: Reverse AD rule for reduce_by_index with min.

and the return sweep adds to the adjoint of each element of vs the adjoint of the histogram element corresponding to its key:

```
let $\overline{vs}$ = map2 ($\lambda$ k $\rightarrow$ if k >= n then 0 else $\overline{xs}$[k]) ks
        ▹ map2 (+) $\overline{vs}$
```

The AD overhead should be under a factor of 2×.

4.2.2 Min/Max. Figure 5 shows the re-write rule. The primal trace still consists of a reduce-by-index, but its operator is lifted to also compute the index of the minimal element ($\mathtt{min}^L$). The return sweep uses a scatter to update the adjoint of only the element that has produced the minimal value for that bin. In case of duplicates $\mathtt{min}^L$ selects the one at the smallest index in vs.

4.2.3 Multiplication. Specializing rule 5 to multiplication yields:

$$\overline{v_i} \mathrel{+}= \frac{\partial\,(l_i \;\cdot\; v_i \;\cdot\; r_i)}{\partial\, v_i} \cdot \overline{\mathtt{x}}_{k_i} \quad\Rightarrow\quad \overline{v_i} \mathrel{+}= (l_i \;\cdot\; r_i) \cdot \overline{\mathtt{x}}_{k_i}$$

Computing $l_i \cdot r_i$ can be achieved in a similar way as for reduction, by lifting the reduce by index (of the primal) to compute for each bin the number of zero and the product of non-zero elements falling in that bin. It follows that the lifted operator is (**i64**.+,α.*) for some numeric type α. The return sweep consists of a map that adds the contributions to the adjoint of vs.

4.2.4 Invertible Operators. Similar to reduce, if the language would allow the user to specify the (left-)inverse of an associative and commutative operator, then a re-write similar to 4 would enable a significantly more efficient differentiation rule — illustrated in figure 6 — than the general case, which is based on sorting.

4.2.5 Vectorized Operators. Assuming $\odot : \alpha \rightarrow \alpha \rightarrow \alpha$, $e_\odot : \alpha$, $\mathtt{h}^0 : \mathtt{[w][d]}\alpha$, $ks : \mathtt{[}n\mathtt{]}$**i64** and $vss : \mathtt{[n][d]}\alpha$ one can interchange the **reduce_by_index** inside the **map** of a vectorized operator with the following rule:

```
-- Assuming $\odot$ with inverse $\odot_{inv}$
-- Original is the same as primal:
let xs = reduce_by_index (replicate w $e_\odot$) $e_\odot$ ks vs

-- Return sweep:
let $\overline{vs}$ = map2($\lambda k_i\ v_i \rightarrow$ let $b_i$= xs[k] $\odot_{inv}$ $v_i$ in $\overline{\odot_{b_i}}$ $v_i$)
            ks vs ▹ map2 ($\overline{+}$) $\overline{vs}$
              denoting $\overline{\odot_{b_i}}$ $\leftarrow$ $\mathbb{VJP}^{\mathbb{LAM}}$ ($\lambda\ x \rightarrow b_i \odot x$) $\overline{xs}$[k]

-- A possible instantiation for $\odot$ and $\odot_{inv}$:
let sumOfProd (p1, s1) (p2, s2) =
      (p1 + p2 + s1*s2, s1 + s2)
let sumOfProd$_{inv}$ (p, s) (p2, s2) =
      (p - p2 - (s-s2)*s2, s - s2)
```

Figure 6: Reverse AD rule for reduce-by-index with left-invertible operator.

$$\begin{gathered}\textbf{reduce\_by\_index}\ \mathsf{h}^0\ (\textbf{map2}\ \odot)\ (\textbf{replicate}\ \mathsf{d}\ e_\odot)\ \mathsf{ks}\ \mathsf{vss}\\ \equiv\\ \textbf{map2}\ (\ \mathsf{h}^0_{col}\ \mathsf{vss}_{col} \rightarrow\\ \qquad\textbf{reduce\_by\_index}\ \mathsf{h}^0_{col}\ \odot\ e_\odot\ ks\ \mathsf{vss}_{col}\\ )\ (\textbf{transpose}\ \mathsf{h}^0)\ (\textbf{transpose}\ \mathsf{vss})\ \triangleright\ \textbf{transpose}\end{gathered} \tag{6}$$

We currently do not utilize this rule prior for differentiation. However, we do pattern match the case of vectorized common operators — i.e., a perfectly-nested sequence of maps which ultimately applies plus, min/max or multiplication — and implement it as a special case: the primal will correspond to a **reduce_by_index** with the corresponding vectorized lifted operators, and the return sweep would similarly build a map nest to perform all necessary updates.

4.2.6 Discussing Performance. The design of the reduce-by-index construct [17] navigates the time-space tradeoff by employing

- a multi-histogram technique that is aimed at reducing the conflicts in shared (or global) memory by having groups of threads cooperatively building partial histograms, and
- a multi-pass technique that processes different partitions of the histogram at a time as a way to optimize trashing in the last-level cache.

In addition, reduce-by-index attempts to maintain the histogram(s) in scratchpad memory if possible, and it implements the best form of atomic update available on the hardware for the given datatype. For example (i) addition, min, max, multiplication have efficient hardware implementation accessible through primitives such as `atomicAdd`, (ii) datatypes that fit into 64-bits use CAS instructions, while (iii) the rest use mutex-base locking, which is quite expensive.

The AD overheads of the specialized cases of reduce-by-index (other than addition) are difficult to predict (or reason at a high level) because of the lifted operators. For example, `int32.min` is efficiently supported in hardware, but its lifting operates on `(int64,int32)` tuples and thus require mutex-based locking. Furthermore, the size of the element type is tripled, which restricts the multi-histogram degree and thus impedes the reduction of conflicts. For vectorized min, the tripling of the size might make it to not fit in scratchpad memory anymore. All these factors may result in a significant overhead that is not visible in the re-write rules.

5 REVERSE-AD FOR SCAN (PREFIX SUM)

Our differentiation of scan is restricted to operators that are defined on tuples of scalars of arbitrary dimension d, or to vectorized liftings of such operators, i.e., a tower of maps applied on top of such operators. Our algorithm manifests and multiplies $d \times d$ Jacobians, which is arguably asymptotically preserving since d is a constant but it is not AD efficient. Nevertheless, section 6.3 demonstrates that it still offers competitive performance on many practical cases.

Discussion is structured as follows: section 5.1 presents the step-by-step rationale used to derive the algorithm — which we found interesting because it combines dependence analysis on arrays with functional-style re-write rules — then section 5.2 puts together the algorithm, and section 5.3 presents several specializations that enable significant performance gains — e.g., vectorized operators and sparsity patterns — and concludes with a discussion that qualitatively compare our algorithm with the one of PPAD [44].

5.1 Deriving the Differentiation of Scan

An inclusive scan [6] computes all prefixes of an array by means of an associative operator $\odot$ with neutral element $e_\odot$:

```
let rs = scan $\odot$ $e_\odot$[$a_0, \ldots, a_{n-1}$]
       ≡ [$a_0, a_0 \odot a_1, \ldots, a_0 \odot \ldots \odot a_{n-1}$]
```

While the derivation of (multi-) reduce builds on a functional-like high-level reasoning, in scan's case, we found it easier to reason in an imperative, low-level fashion. For simplicity we assume first that $\odot$ operates on real numbers, and generalize later:

```
rs[0] = as[0]
for i in 1 ... n-1 do
    rs[i] = rs[i-1] $\odot$ as[i]
```

The loop above that implements scan, writes each element of the result array `rs` exactly once. To generate its return sweep, we can reason that we can fully unroll the loop, then apply the main rewrite-rule from equation 1 to each statement and finally gather them back into the loop. The unrolled loop is:

```
rs[0] = as[0]
rs[1] = rs[0] $\odot$ as[1]
...
rs[n -1] = rs[n -2] $\odot$ as[n -1]
```

The application of $\mathbb{VJP}$ to each statement results in return sweep:

```
$\overline{rs}$[n-2] = $\partial$(rs[n-2] $\odot$ as[n-1])/$\partial$rs[n-2] * $\overline{rs}$[n-1]
$\overline{as}$[n-1] = $\partial$(rs[n-2] $\odot$ as[n-1])/$\partial$as[n-1] * $\overline{rs}$[n-1]
...
$\overline{rs}$[0] = $\partial$(rs[0] $\odot$ as[1])/$\partial$rs[0] * $\overline{rs}$[1]
$\overline{as}$[1] = $\partial$(rs[0] $\odot$ as[1])/$\partial$as[1] * $\overline{rs}$[1]
$\overline{as}$[0] = $\overline{rs}$[0]
```

The differentiated statements can be rolled back to form the loop:

```
$\overline{rs}$ = copy $\overline{ys}$
for i = n-1 ... 1 do
    $\overline{rs}$[i-1] += $\partial$(rs[i-1] $\odot$ as[i])/$\partial$rs[i-1] * $\overline{rs}$[i]
    $\overline{as}$[i]   += $\partial$(rs[i-1] $\odot$ as[i])/$\partial$as[i] * $\overline{rs}$[i]
$\overline{as}$[0] += $\overline{rs}$[0]
```

where $\overline{\mathtt{ys}}$ denotes the adjoint of `rs` corresponding to the uses of scan's result in the remaining of the program.

```
1   -- We denote with n the length of as and with
2   --    d the dimensionality of the element type α
3   -- Primal trace is the same as the original:
4   let rs = scan ⊙ e⊙ as
5
6   -- Return sweep:
7   --   (1) computes cs (Jacobians):
8   let cs = map (λi → if i == n-1
9                      then I_d
10                     else J⃖_{⊙,as} i rs[i]
11                  -- ^ i.e., ∂(rs[i] ⊙ as[i+1])/∂rs[i]
12               ) [0,...,n-1]
13    denoting J⃖_{⊙,as} i x  =  (f_{0,i,⊙,as} x, ..., f_{d−1,i,⊙,as} x)
14          f_{k,i,⊙,as}  ←  VJP^LAM (λx → x ⊙ as[i+1]) (unitVec k)
15
16  --   (2) computes the adjoint of rs by means of
17  --       parallelizing a backward linear recurrence
18  let lin_o (b1,c1) (b2,c2) = (b2 + c2 · b1, c2 × c1)
19  let (rs̅, _) = zip (reverse ys̅) (reverse cs)
20              ▹ scan lin_o (0_d, I_d)
21              ▹ reverse ▹ unzip
22  --   (3) updates the adjoint of as by a map:
23  let as̅ +̅=
24    map (λi r̅i̅ ai →
25          if i == 0 then r̅i̅
26          else g_{i,⊙} ai -- i.e., ∂(rs[i-1] ⊙ ai)/∂ai · r̅i̅
27          denoting g_{i,⊙} ← VJP^LAM (λx → rs[i-1] ⊙ x) r̅i̅
28      ) [0,...,n-1] rs̅ as
```

Figure 7: Reverse-AD Rule for Scan.

Simple dependence analysis, for example based on direction vectors, shows that the loop can be safely distributed across its two statements, since they are not in a dependency cycle:

```
rs̅ = copy ys̅
for i = n-1 ... 1 do
    rs̅[i-1] += ∂(rs[i-1] ⊙ as[i])/∂rs[i-1] * rs̅[i]

for i = n-1 ... 0 do
    as̅[i] += (i==0) ? rs̅[0] :
                    ∂(rs[i-1] ⊙ as[i])/∂as[i] * rs̅[i]
```

The second loop exhibits no cross iteration dependencies, hence the adjoints of as can be computed by a **map**. The first loop can be expressed by the backwards linear recurrence of form:

$$\overline{rs}_{n-1} = \overline{ys}_{n-1}$$
$$\overline{rs}_i = \overline{ys}_i + cs_i \cdot \overline{rs}_{i+1}, i \in n-2\ldots 0$$

where cs is defined by $cs_{n-1} = 1$ and $cs_i = \partial(rs_i \odot as_{i+1}) / \partial rs_i$. Such a recurrence is known to be solved with a scan whose operator is linear-function composition [7].

5.2 Re-Write Rule for Arbitrary-Tuple Types

The reasoning used in the previous section generalizes to d-dimensional tuples (chosen for simplicity) of the same numeric type α, essentially by lifting scalar addition and multiplication to operate on vectors and matrices.[4] Figure 7 presents the proposed re-write rule.

For example, the linear-function composition operator has type $\texttt{lin}_o : (\alpha^d, \alpha^{d\times d}) \to (\alpha^d, \alpha^{d\times d}) \to (\alpha^d, \alpha^{d\times d})$ and +, · and × denote vector addition, vector-matrix and matrix-matrix multiplication, respectively, where vectors live in α^d (its zero is $\texttt{0}_d$) and matrices in $\alpha^{d\times d}$. Similarly, $cs_{i>0}$ are the $d \times d$ Jacobians corresponding to $\partial(rs_i \odot as_{i+1}) / \partial rs_i$ and $cs_0 = \mathbf{I_d}$ is the identity matrix.

The code for computing $cs_{i>0}$ — represented in figure 7 by means of $\overleftarrow{J}_{\odot,\texttt{as}}$ `i rs[i]` — is generated by applying the $\mathbb{JVP}^{\mathrm{LAM}}$ transformation to lambda $\lambda x \to x \odot as[i+1]$ and to each of the unit vectors (`unitVec` k, $k = 0\ldots, d-1$) as the adjoint of the result.

The generated code consists of two kernels: one corresponding to the fusion of the map computing `cs` together with the reversion of `cs` and $\overline{ys}$ and the scan, and the second corresponding to the fusion of the reversion of $\overline{rs}$ with the map that updates $\overline{as}$.

5.3 Specializations

5.3.1 Addition. It is folklore knowledge that the return sweep of

let ys = **scan** (+) 0 as   is:

let $\overline{as}$ = **scan** (+) 0 (**reverse** $\overline{ys}$) ▹ **reverse** ▹ **map2** (+) 0 $\overline{as}$

This can also be derived from figure 7: $\partial(\texttt{rs[i] + as[i+1]}) / \partial \texttt{rs[i]}$ simplifies to 1, hence `cs = ` **replicate** `n 1`, which means that we are composing linear functions of the form $f_{\overline{ys}_i}\ x = \overline{ys}_i + x$, which results in $\overline{rs}$ = **scan** (+) 0 (**reverse** $\overline{ys}$) ▹ **reverse**, and so on.

5.3.2 Vectorized Operators. Scans with vectorized operators are transformed to scans with scalar operators (whenever possible) by the (recursive) application of the Iswim rule[5] [19]:

$$\begin{array}{c}\textbf{scan}\ (\textbf{map2}\ \odot)\ (\textbf{replicate}\ \texttt{n}\ e_\odot)\ \texttt{matrix}\\ \equiv \\ \textbf{map}\ (\textbf{scan}\ \odot\ e_\odot)\ (\textbf{transpose}\ \texttt{matrix})\ \triangleright\ \textbf{transpose}\end{array} \tag{7}$$

and differentiation is applied on the resulted code. This is essential because the "general-case" rule in figure 7 is *not* asymptotic preserving in the case of array datatypes due to the explicit manifestation and multiplication of Jacobians.

5.3.3 Block-Diagonal Sparsity (BDS). The expensive step in our re-write rule of figure 7 is that entire $d \times d$ Jacobians corresponding to $\partial(\texttt{rs[i] + as[i+1]}) / \partial \texttt{rs[i]}$ (computed at line 10) are stored in `cs` and later multiplied inside (the scan with operator) $\texttt{lin}_o$ (line 18/20). It is not only that $\texttt{lin}_o$ takes $O(d^3)$ time, but more importantly, the size of the elements being scanned is proportional with d^2, which quickly restricts (i) the amount of efficient sequentialization, and ultimately (ii) the storing of intermediate data in scratchpad memory that is paramount for the GPU efficiency of scan.

In this sense, we have implemented (compiler) analysis to statically detect sparse Jacobians of block-diagonal form. More precisely,

[4] The reasoning generalizes also to tuples of heterogeneous scalar types.

[5] Iswim states that summing up the elements of each column of a matrix can be achieved by transposing the matrix, summing up each row and transposing back the result.

we consider the case of k blocks, where each block has size $q \times q$, hence $d = k \cdot q$. Multiplication preserves the block-diagonal shape:

$$\begin{bmatrix} \mathbf{M_1^1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \mathbf{M_k^1} \end{bmatrix} \times \begin{bmatrix} \mathbf{M_1^2} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \mathbf{M_k^2} \end{bmatrix} = \begin{bmatrix} \mathbf{M_1^1} \times \mathbf{M_1^2} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \mathbf{M_k^1} \times \mathbf{M_k^1} \end{bmatrix}$$

We multiply such a matrix with v, a vector of length d as such:

$$\begin{bmatrix} \mathbf{v_1}, & \cdots, & \mathbf{v_k} \end{bmatrix} \times \begin{bmatrix} \mathbf{M_1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \mathbf{M_k} \end{bmatrix} = \begin{bmatrix} \mathbf{v_1} \times \mathbf{M_1} \\ \vdots \\ \mathbf{v_k} \times \mathbf{M_k} \end{bmatrix}$$

If `cs` has the BDS pattern, then, semantically, we shrink its representation down to a tuple of k arrays each of dimension $n \times (q \times q)$ and similarly, $\overline{ys}$ to a tuple of k arrays of dimension $n \times q$.[6] The computation of $\overline{\texttt{rs}}$ is performed with k different scans, each of them using a scaled-down (adjusted) operator $\texttt{lin}_o^{BDS}$ that is semantically defined on elements of type $(\alpha^q, \alpha^{q \times q})$ — the corresponding vector- and matrix-matrix multiplications inside $\texttt{lin}_o^{BDS}$ are performed as shown above. This reduces the element size of the scanned array, enabling better utilization of scratchpad memory.

5.3.4 Redundant Block-Diagonal (RBDS) Sparsity. The case when the block-diagonal sparsity has the additional property that all the blocks hold identical values, i.e., $M_1 = M_2 = \ldots = M_k$, allows an even more efficient implementation: The representation of `cs` is shrunk down to (semantically) one array of dimension $n \times (q \times q)$ and only one scan is performed. $\texttt{lin}_o^{RBDS}$ now operates on elements of type $(\alpha^d, \alpha^{q \times q})$ and it performs ***one multiplication of*** $q \times q$ ***matrices***, and k vector-matrix multiplications $\mathbf{V^q} \times M^{q \times q}$:

$$\begin{bmatrix} \mathbf{v_1}, & \cdots, & \mathbf{v_k} \end{bmatrix} \times \begin{bmatrix} \mathbf{M} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \mathbf{M} \end{bmatrix} = \begin{bmatrix} \mathbf{v_1} \times \mathbf{M} \\ \vdots \\ \mathbf{v_k} \times \mathbf{M} \end{bmatrix}$$

RBD sparsity has important applications: differentiating the multiplication of two $q \times q$ matrices $A \times B$ with respect to A (or B), results in a Jacobian that consists of q blocks of size $q \times q$, in which each block is equal to B (or A), i.e., the Jacobian of $\partial(A \times B) / \partial A$ is:

$$\begin{bmatrix} B & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & B \end{bmatrix}$$

This is important because the scan with $q \times q$ matrix multiplication is commonly used to parallelize linear recurrences of degree q [7], i.e., $x_i = a_i^0 + a_i^1 * x_{i-1} + \ldots + a_i^q * x_{i-q+1}$.

Similarly, differentiating the (classical) linear function composition with respect to the first argument also results in RBS sparsity, i.e., $\partial(b_2 + c_2 \cdot b_1, c2 \cdot c1) / \partial(b_1, c_1)$ has Jacobian $\begin{bmatrix} c_2 & 0 \\ 0 & c_2 \end{bmatrix}$

```
def linFnComp (b1:f32, c1:f32) (b2:f32, c2:f32) =
    (b2 + c2*b1,    c2 * c1)
def sumOfProd (p1:f32, s1:f32) (p2:f32, s2:f32) =
    (p1 + p2 + s1*s2,    s1 + s2)

def matMul2x2 (a1:f32, b1:f32, c1:f32, d1:f32)
              (a2:f32, b2:f32, c2:f32, d2:f32) =
    ( a1*a2 + b1*c2,    a1*b2 + b1*d2
    , c1*a2 + d1*c2,    c1*b2 + d1*d2 )
-- ^ matMul3x3, and matMul5x5 are similarly defined

def satAdd (x: f32) (y: f32) : f32 =
    if (x+y) > 1000000 then 1000000 else x+y
```

Figure 8: Non standard operators used in evaluation.

5.3.5 Discussion. Our "general-case" rule for scan computes and multiplies $d \times d$ Jacobians. While this arguably preserves the work asymptotic (d is a constant for tuples), it is not AD efficient[7] and it is theoretically inferior to the re-write rule of PPAD [44] — shown in figure 11 in Appendix — which uses only the $\mathbb{VJP}$ transformation, thus avoiding operating with (full) Jacobians.

Our "general-case" rules is however faster than PPAD when the dimensionality d is one or two, and the vectorized-operator and RBDS specializations makes it also more effective on many operators of practical interest, with speed-ups commonly ranging from $1.3\times - 2.25\times$, as reported in section 6.3. In particular the PPAD rule transforms a vectorized operator into an un-vectorized one, which is challenging to map efficiently to the GPU hardware — especially when the dimensions of the array-based element type are not statically known. In Futhark's case, such non-vectorized operators incur prohibitive AD overheads, e.g., two orders of magnitude.

Finally, for the un-vectorized operators that use array arguments, we postulate that a more suited strategy would be to differentiate the classical work-preserving (two-stage) implementation of scan [6] written in terms of loop, map and scatter operators. We plan to use PPAD's rule for high-dimensional tuples of scalars (of arity larger than 3) that do not fall under the RBDS pattern.

6 EXPERIMENTAL EVALUATION

The discussion is structured as follows: section 6.1 presents the evaluation methodology and sections 6.2 ,6.3 and 6.4 evaluate the performance of reverse-mode differentiation of reduce, scan and reduce-by-index, respectively.

6.1 Operators, Datasets, Methodology

The evaluation uses randomly generated arrays and single-precision float as the base numeric type. [8]

6.1.1 Operators. The evaluated operators are: (i) standard addition (+), multiplication (*), and min, (ii) their vectorized forms, e.g., **map2** (*), (iii) 2×2, 3×3 and 5×5 matrix multiplication, e.g., which is used in the parallel implementation of linear recurrences, (iv)

[6] Since the original scan operator is defined on tuples of scalars and Futhark compiler uses a tuple of array representation, it follows that in practice, `cs` is represented as $k \cdot q \cdot q$ arrays of length n, that we tuple differently at no runtime overhead. Similar thoughts apply to $\overline{ys}$ and $\texttt{lin}_o^{BDS}$, i.e., $\texttt{lin}_o^{BDS}$ still operates on tuples of scalars.

[7] There is no constant factor independent of the program that bounds the AD overhead.

[8] Due to negligence, we have used 32-bit integer in some cases: we will fix this in a final version, but this should not significantly influence the performance.

linear function composition, (v) sum of products, and (vi) saturated addition. For convenience, figure 8 shows the non-standard ones. Of note, linear function composition and matrix multiplication are only associative but not commutative, hence they are not valid operators for reduce-by-index, which requires commutativity.

6.1.2 Datasets. The evaluation of reduce and scan uses two datasets denoted by D_1 and D_2:

map2 (*): For vectorized multiplication, D_1 and D_2 correspond to arrays of dimensions $10^6 \times 16$ and $10^7 \times 16$, respectively, which are provided in transposed form.

matMul5x5: For 5×5 matrix multiplication, D_1 corresponds to 10 million elements – each element is a tuple of arity 25 – and D_2 corresponds to 50 million elements.

For the others D_1 and D_2 correspond to 10 and 100 million elements, e.g., the element for sum-of-products is a tuple of floats (arity 2).

The evaluation of reduce-by-index uses six datasets denoted by $D_{i,j}$ where $i = 1, 2$ refers to the length of the input arrays and $j = 1, 2, 3$ refers to the length of the histogram: 31, 401 and 50000 elements, respectively. The length of the input arrays are the same as before, except for the case of vectorized operators where $D_{1,j}$ and $D_{2,j}$ correspond to arrays of dimensions $10^6 \times 10$ and $10^7 \times 10$, respectively.

6.1.3 Hardware. The evaluation uses an Nvidia A100 40GB PCIe GPU, which has the listed peak memory bandwidth of **1555 Gb/sec**.

6.1.4 Methodology. We measure the total application running time, but *excluding* the time needed to transfer the program input and (final) result between device and host memory spaces. We report the average of at least 25 runs — or as many as are needed for a 95% confidence interval to be reached.

The performance of the primal (original program) is reported as memory throughput, measured in **Gb/sec**. Denoting with n the length of the input array and with β the size of the array element type, the total number of bytes `Nbytes` is computed as follows:

$$\texttt{Nbytes} = \begin{cases} n \cdot \texttt{sizeof}(\beta), & \text{for reduce} \\ 2 \cdot n \cdot \texttt{sizeof}(\beta), & \text{for scan} \\ 3 \cdot n \cdot \texttt{sizeof}(\beta) \; + \; n \cdot 8, & \text{for reduce-by-index} \end{cases} \tag{8}$$

For reduce and scan these are the minimal number of bytes that needs to be accessed from global memory, e.g., reduce needs to read each element once. For reduce by index we reason that:

1. reading the input array requires $n \cdot \texttt{sizeof}(\beta)$ bytes,
2. reading the key requires $n \cdot 8$ bytes, because the key is represented as a 64-bit integer,
3. updating the histogram *may* require a read and a write access, hence another $2 \cdot n \cdot \texttt{sizeof}(\beta)$ bytes.

For the primal, histograms of sizes 31 and 401 typically fit in scratchpad (shared/fast) memory, but histograms of size 50000 do not and are stored in global memory. It follows that we choose to consider the accesses that update the histogram in order to be able to meaningfully compare across different datasets and implementations — i.e., our measure of memory throughput is essentially a normalized runtime. The consequence is that on small histograms the reported **Gb/sec** may exceed the peak memory bandwidth of the hardware, because the histogram is maintained in shared memory.

The performance of the differentiated code — that computes both the primal and the adjoint results — is presented in terms of AD overhead, which is defined as the ratio between the running times of the derivative and primal (original) – the lower the better.

6.2 Reduce

The left-hand side of figure 9 presents the performance of the primal and reverse-mode differentiation of reduce on the evaluated operators and datasets. As competitor, we use the algorithm presented in PPAD [44], except for multiplication, where we use our general-case as competitor, which is also somewhat similar to [23] and is more efficient than PPAD. We make the following observations:

- In most cases, e.g., +, ·, `min`, D_1 is too small to overcome the "system" overheads, resulting in sub-optimal performance of about half the peak bandwidth, e.g., the total runtime is about 53 micro-seconds μs for min, and launching the kernel takes ten(s) μs. Matter are much improved on the larger D_2.
- The specializations for +, ·, `min` enable efficient differentiation — all AD overheads are under 3×. The case of multiplication highlights the impact of specialization: it offers 3.4−4.5× speedup in comparison with the general case (**Cmp**).
- linFnComp and sumOfProd are treated with the general-case algorithm and result in significantly larger overheads 5 − 10×.
- Applying by hand the specialization for invertible operators to sumOfProd — see section 3.3 — results in AD overheads of 2.4× and 2.8× for D_1 and D_2, which offers good efficiency.
- The impact of the Irwim rule (see re-write 3) is highlighted by the case of vectorized multiplication: the AD overhead is under 4× and reasonably close to that of multiplication. In comparison, PPAD (**Cmp**) has overhead of 797× on D_2, because it differentiate such a reduction into scans whose operators are defined on arrays but are not vectorized; such reductions/scans are ill supported by the Futhark compiler.
- Matrix multiplication triggers the general-case algorithm but results in "reasonable" AD overheads between 4.6−7.6×. This are still between 2 × −5.3× faster than the PPAD algorithm.
- in all tested cases our algorithm for reduce is faster than PPAD — this is not surprising since our algorithm is AD efficient, while PPAD's piggybacks on the algorithm for scans, which is claimed to not be AD efficient.

6.3 Scan

The right-hand side of figure 9 presents the performance of scan:

- The specialized rule for addition results in small AD overheads of under 3×, but min and multiplication — which are treated with the general-case rule (arity 1) — are not far behind, i.e., their AD overheads are under 4.1×.
- The operators defined on tuples of arity 2, namely `linFnComp` and `sumOfProd` still offer decent AD overheads of under 5.3×, where `linFnComp` is more efficient because it benefits from the optimization of RBD sparsity, discussed in section 5.3.4.
- The application of Iswim rule (see re-write 7) in the case of vectorized multiplication is very beneficial, resulting in AD overheads of 0.44×, which suggests that Futhark compiler

Op	`reduce (+)`			`reduce min`			`reduce (*)`			`scan (+)`			`scan min`			`scan (*)`		
	Prim	**Our**	**Cmp**	**Prim**	**Our**	**Cmp**	**Prim**	**Our**	**Cmp**	**Prim**	**Our**	**Cmp**	**Prim**	**Our**	**Cmp**	**Prim**	**Our**	**Cmp**
	Gb/s	$\frac{\text{AD-F}}{\text{Prim}}$	$\frac{\text{AD-C}}{\text{Prim}}$	**Gb/s**	$\frac{\text{AD-F}}{\text{Prim}}$	$\frac{\text{AD-C}}{\text{Prim}}$	**Gb/s**	$\frac{\text{AD-F}}{\text{Prim}}$	$\frac{\text{AD-C}}{\text{Prim}}$	**Gb/s**	$\frac{\text{AD-F}}{\text{Prim}}$	$\frac{\text{AD-C}}{\text{Prim}}$	**Gb/s**	$\frac{\text{AD-F}}{\text{Prim}}$	$\frac{\text{AD-C}}{\text{Prim}}$	**Gb/s**	$\frac{\text{AD-F}}{\text{Prim}}$	$\frac{\text{AD-C}}{\text{Prim}}$
D_1	755	2.4×		755	2.8×		755	2.7×	9.1×	584	1.8×		808	2.5×	5.2×	808	3.4×	4.8×
D_2	1246	2.5×		1261	2.6×		1270	2.9×	13.1×	1131	2.8×		1111	2.8×	6.3×	1131	4.1×	6.0×

Op	`reduce linFnComp`			`reduce sumOfProd`			`reduce (map2(*))`			`scan linFnComp`			`scan sumOfProd`			`scan (map2(*))`		
D_1	530	5.0×	7.3×	808	7.2×	10.9×	1016	3.4×	668×	982	3.4×	6.2×	970	4.5×	5.9×	105	0.44×	33.4×
D_2	712	5.9×	9.3×	1317	10.×	16.8×	1342	4.0×	797×	1208	3.8×	7.0×	1202	5.3×	6.8×	114	0.43×	32.8×

Op	`reduce matMul2x2`			`reduce matMul3x3`			`reduce matMul5x5`			`scan matMul2x2`			`scan matmul3x3`			`scan matMul5x5`		
D_1	611	4.6×	9.5×	684	6.4×	22.3×	252	6.3×	33.3×	1032	4.3×	6.9×	880	7.6×	14.8×	192	6.9×	12.9×
D_2	844	6.2×	12.4×	839	7.6×	27.0×	268	6.7×	35.3×	1149	4.6×	7.3×	943	8.1×	15.7×	191	6.9×	12.8×

Figure 9: Performance of the reverse-mode differentiation of `reduce` and `scan`. The performance of the original program (primal) is measured in Gb/sec and is reported in column Prim. AD performance is reported in terms of AD Overhead, which is defined as the ratio between the differentiated and primal runtimes (the lower the better). The AD overhead of our approach is reported in column Our and of the competitor technique in column Cmp. The competitor are the re-write rules of PPAD [44] except from reduce with multiplication, where we use the general case of differentiating reduction, which is similar to [23].

Op	+		`map2 (+)`		`min`		`map2 min`	
	Prim	**OV**	**Prim**	**OV**	**Prim**	**OV**	**Prim**	**OV**
	Gb/s	$\frac{\text{AD}}{\text{Prim}}$	**Gb/s**	$\frac{\text{AD}}{\text{Prim}}$	**Gb/s**	$\frac{\text{AD}}{\text{Prim}}$	**Gb/s**	$\frac{\text{AD}}{\text{Prim}}$
$D_{1,1}$	1176	1.5×	610	1.6×	1176	1.9×	621	3.0×
$D_{1,2}$	1087	1.4×	552	1.5×	1156	2.0×	557	4.1×
$D_{1,3}$	806	1.4×	135	1.1×	324	3.0×	64	3.2×
$D_{2,1}$	1947	1.9×	975	1.9×	1932	2.1×	1041	9.6×
$D_{2,2}$	1941	1.9×	950	1.8×	1932	2.1×	1009	13.7×
$D_{2,3}$	871	1.4×	164	1.1×	344	3.9×	86	4.0×

Op	*		`map2 (*)`		`sumOfProd`		`satAdd`	
$D_{1,1}$	1212	2.1×	643	4.0×	1495	29.4×	1136	29.7×
$D_{1,2}$	1111	2.0×	569	3.9×	1448	48.6×	1136	50.6×
$D_{1,3}$	303	2.6×	64	4.5×	177	9.4×	302	21.2×
$D_{2,1}$	1955	2.4×	1013	8.6×	2348	45.8×	1937	50.3×
$D_{2,2}$	1945	2.4×	977	9.0×	2353	84.0×	1923	92.0×
$D_{2,3}$	265	2.1×	80	12.4×	141	8.0×	264	19.5×

Figure 10: Reverse-AD Performance of `reduce_by_index`.

should always apply it. In comparison, PPAD (**Cmp**) is 76× slower, for the same reasons as the ones discussed for reduce.

- For matrix multiplication, which benefits from the RBD sparsity optimization, the AD overhead reaches a peak of 8.1× for 3×3 matrices, but then decreases for 4×4 and 5×5 matrices up until 6.9×, which seems to indicate that performance remains a constant factor away from the primal.
- Our algorithm is faster than PPAD in all evaluated case, with speedups typically ranging between 1.3×−2.25×, but we surmise that on operators defined on high-dimensional tuples ($d \geq 3$) that do not benefit from the sparsity optimizations, PPAD will be significantly more efficient than ours.

6.4 Reduce By Index

Figure 10 shows the performance of differentiating reduce-by-index. We do not use a competitor since we are not aware of work on differentiating this construct at a high-level. Key observations are:

- The base cases (+,$*$,`min`) are efficiently differentiated with overheads under 2.4×, except for `min` on $D_{1/2,3}$, which corresponds to the largest histogram of length 50K that fits only in global memory. The slowdown is due to the lifted operator requiring a mutex lock instead of using atomic primitives such as `atomicAdd` or `atomicMul` as with the other cases.
- Vectorized operators incurs larger overheads, due to reasons similar to the ones discussed for `min` above (see also section 4.2.6). However, applying by hand the adaptation of IRWIM for reduce by index (see re-write 6) is very beneficial, resulting in the following column of AD overheads: $[2.7\times, 2.8\times, 2.9\times, 3.6\times, 3.5\times, 1.7\times]^T$ for vectorized multiplication (and similar for `min`). We are planning to implement this transformation in the differentiation pass.
- `sumOfProd` and `satAdd` are dispatched to the general-case algorithm that involves sorting and is inefficient, resulting in AD overheads as high as 92×. The rationale behind this is discussed at the end of section 4.1. An obvious optimizations would be to improve the underlying sorting implementation.
- Supporting the invertible-operator refinement, discussed in section 4.2.4 and applied by hand to `sumOfProd`, results in very efficient differentiation, i.e. the following AD overhead column $[1.8\times, 1.8\times, 1.1\times, 2.1\times, 2.1\times, 1.1\times]^T$.

7 RELATED WORK

The most related work is the one of PPAD [44] that presents algorithms for high-level reverse-mode differentiation of reduce and scan. We have compared with it throughout the paper: Essentially our treatment of reduce is superior, as ours is AD efficient, and their treatment of scan is theoretically superior to the one presented in this paper and also practically superior in the case of high-dimensional tuples when the RBD sparsity does not apply.

A body of work has investigated how to differentiate (parallel) functional array languages at a high level, i.e., before parallelism is mapped to the hardware. Dex [43] uses a technique where the program is first linearized, producing a linear map, then this linear map is transposed producing the adjoint code. Dex supports accumulators, which are discriminated by the type system, intuitively, into parallel or sequential loops, but does not support second-order parallel constructs such as scan and reduce-by-index.

$\widetilde{F}$ [51] proposes an AD implementation that is applied to a nested-parallel program and uses the forward mode, along with rewrite rules for exploiting sparsity in certain cases. DiffSharp [4] is a library for AD that aims to make available to the machine learning (ML) community, in convenient form, a range of AD techniques, including, among others, nesting of forward/reverse mode AD operations, efficient linear algebra primitives, and a functional API that emphasizes the use of higher-order functions and composition.

PRAD [26] is a parallel algorithm for reverse-mode AD of recursive fork-join programs that (i) is provably work efficient, (ii) has span within a polylogarithmic factor of the original program, and (iii) supports Cilk fork-joint parallelism, without requiring parallel annotations. Evaluated on 8 ML applications, PRAD is reported to achieve 1.5× AD overhead and 8.9× speedup on 18 cores.

None of these approaches propose specific AD algorithms for differentiating reduce, scan or reduce-by-index. The algorithms presented in this paper are part of Futhark's AD system [50], that supports (forward and) reverse-mode differentiation of nested-parallel programs. The key difference is that reverse-AD avoids using tape by a redundant-execution technique and by techniques that are aimed to rely on dependency analysis of loops [38, 39]. The AD implementation benefits from various compiler optimizations [16, 19, 20, 22, 34, 35] and specialized code generation [17, 18, 27, 36].

Another rich body of work refers to the implementation of AD algorithms in the imperative context, where parallelism is already mapped to hardware, e.g, by means of low-level APIs such as OpenMP and Cuda. Enzyme [31] applies AD on low-level compiler representation, thus taking advantage of both pre- and post-AD compiler optimizations. Since the support for AD is built in the low-level compiler (LLVM), their approach naturally achieves AD interoperability [33] across languages, e.g., Julia, and parallel APIs, such as OpenMP, MPI and Cuda. In particular, the AD algorithm for Cuda [32] makes use of AD-specific GPU memory optimizations including caching tape values in thread-local storage as well as memory-aware adjoint updates. However, we speculate that if the target Cuda kernels is already maxed out in terms of resource usage then the tape would need to be mapped in global memory, which will degrade the AD performance. This is typically the case for the kernels generated from reduce(-by-index) and especially scan [10, 36], which uses a single-pass implementation [30]. Reversely, Enzyme's low-level approach of differentiating memory would likely offer better performance for cases such as reduce-by-index with non-invertible operators (e.g., `satAdd`).

In the context of parallel API such as OpenMP and MPI, other reverse-AD implementations have been proposed, either by compiler transformations [25] or by overloading techniques [47, 48].

Reverse AD has also been implemented in DSLs for stencil computations [24], and tensor languages [5] that support constrained forms of loops, which do not require the use of tapes.

ML practitioners use tools such as Tensorflow [1], PyTorch [42] or JAX [8, 11] that restrict the programming interface, but offer well-tuned primitives for AI. A practically important direction is to promote AD interoperability across popular languages [40, 41], which bears similarities to prior work on supporting generics in computer algebra [9].

Finally, the time-space tradeoff for reverse-mode AD is systematically studied by Siskind and Pearlmutter [52], and Tapenade [2] supports a wealth of checkpointing techniques. Other approaches aimed at sequential code include ADOL-C [14], and Stalingrad [45].

In conclusion, none of the imperative or functional approaches (other than PPAD) have proposed AD algorithms specific to reduce, scan and reduce-by-index, or evaluated them for GPU execution.

8 CONCLUSIONS

We have presented reverse-mode differentiation algorithms for reduce, scan and reduce by index second-order parallel array combinators. Interestingly, the general-case algorithm re-writes the differentiation of a construct in terms of other, less-efficient ones: reduce's re-write uses scans, reduce-by-index uses multi-scan (implemented by sorting), and scan's re-write is not AD efficient.

However, we have also shown that for most cases of practical interest, specializations that enable efficient differentiation are possible: (i) vectorized operators are reduced to scalar ones and then differentiated, (ii) invertible operators allow reduce(-by-index) to be differentiated in terms of map/reduce-by-index constructs, and (iii) sparsity optimization allows reasonably-efficient differentiation of scans with (tuple-based) matrix-multiplication operators, which seem to remain a constant-factor away from the primal.

Most important, we have reported, to our knowledge, the first evaluation of reverse AD of said constructs in the context of GPU execution, which constitutes a useful baseline for future work.

ACKNOWLEDGMENTS

We would like to acknowledge Troels Henriksen and Robert Schenck for their invaluable contributions to implementing AD in Futhark. We credit Troels with the idea of differentiating the classical work-efficient implementation of scan in the case of un-vectorized operators on arrays. This work has been supported by the UCPH Data+ grant: *High-Performance Land Change Assessment* and by the the Independent Research Fund Denmark (DFF) under the grant *Monitoring Changes in Big Satellite Data via Massively Parallel AI*.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] M. Araya-Polo and Laurent Hascoët. 2004. Data Flow Algorithms in the Tapenade Tool for Automatic Differentiation. In *Proceedings of the European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS 2004)*, P. Neittaanmäki, T. Rossi, S. Korotov, E. Oñate, J. Périaux, and D. Knörzer (Eds.). University of Jyväskylä, Jyväskylä, Finland. online at http://www.mit.jyu.fi/eccomas2004/proceedings/pdf/550.pdf.

[3] Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2017. Automatic Differentiation in Machine Learning: A Survey. *J. Mach. Learn. Res.* 18, 1 (Jan. 2017), 5595–5637.

[4] Atilim Gunes Baydin, Barak A. Pearlmutter, and Jeffrey Mark Siskind. 2015. DiffSharp: Automatic Differentiation Library. arXiv:cs.MS/1511.07727

[5] Gilbert Bernstein, Michael Mara, Tzu-Mao Li, Dougal Maclaurin, and Jonathan Ragan-Kelley. 2020. Differentiating a Tensor Language. https://doi.org/10.48550/ARXIV.2008.11256

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

[7] Guy E. Blelloch. 1990. Prefix sums and their applications.

[8] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/google/jax

[9] Y. Chicha, M. Lloyd, C. Oancea, and S. M. Watt. 2004. Parametric Polymorphism for Computer Algebra Software Components. In *Proc. 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Comput.* Mirton Publishing House, 119–130.

[10] Morten Tychsen Clausen. 2021. *Regular Segmented Single-pass Scan in Futhark.* Master's thesis. Department of Computer Science, Faculty of Science, University of Copenhagen, https://futhark-lang.org/student-projects/morten-msc-thesis.pdf. https://futhark-lang.org/student-projects/morten-msc-thesis.pdf

[11] Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning* (2018), 23–24.

[12] Fabian Gieseke, Sabina Rosca, Troels Henriksen, Jan Verbesselt, and Cosmin E. Oancea. 2020. Massively-Parallel Change Detection for Satellite Time Series Data with Missing Values. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 385–396. https://doi.org/10.1109/ICDE48307.2020.00040

[13] Sergei Gorlatch. 1996. Systematic extraction and implementation of divide-and-conquer parallelism. In *Programming Languages: Implementations, Logics, and Programs*, Herbert Kuchen and S. Doaitse Swierstra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 274–288.

[14] Andreas Griewank, David Juedes, and Jean Utke. 1996. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software (TOMS)* 22, 2 (1996), 131–167.

[15] Marc Henrard. 2017. *Algorithmic Differentiation in Finance Explained.* https://doi.org/10.1007/978-3-319-53979-9

[16] Troels Henriksen and Martin Elsman. 2021. Towards Size-Dependent Types for Array Programming. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (Virtual, Canada) *(ARRAY 2021)*. Association for Computing Machinery, New York, NY, USA, 14. https://doi.org/10.1145/3460944.3464310

[17] Troels Henriksen, Sune Hellfritzsch, Ponnuswamy Sadayappan, and Cosmin Oancea. 2020. Compiling Generalized Histograms for GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 97, 14 pages.

[18] Troels Henriksen, Ken Friis Larsen, and Cosmin E. Oancea. 2016. Design and GPGPU Performance of Futhark's Redomap Construct. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (Santa Barbara, CA, USA) *(ARRAY 2016)*. ACM, New York, NY, USA, 17–24.

[19] Troels Henriksen and Cosmin Eugen Oancea. 2013. A T2 Graph-reduction Approach to Fusion. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing* (Boston, Massachusetts, USA) *(FHPC '13)*. ACM, New York, NY, USA, 47–58. https://doi.org/10.1145/2502323.2502328

[20] Troels Henriksen and Cosmin E. Oancea. 2014. Bounds Checking: An Instance of Hybrid Analysis. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (Edinburgh, United Kingdom) *(ARRAY'14)*. ACM, New York, NY, USA, Article 88, 7 pages. https://doi.org/10.1145/2627373.2627388

[21] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[22] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. 2019. Incremental Flattening for Nested Data Parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. ACM, New York, NY, USA, 53–67. https://doi.org/10.1145/3293883.3295707

[23] P. Hovland and C. Bischof. 1998. Automatic differentiation for message-passing parallel programs. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing.* 98–104. https://doi.org/10.1109/IPPS.1998.669896

[24] Jan Hückelheim, Navjot Kukreja, Sri Hari Krishna Narayanan, Fabio Luporini, Gerard Gorman, and Paul Hovland. 2019. Automatic Differentiation for Adjoint Stencil Loops. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP 2019)*. Association for Computing Machinery, New York, NY, USA, Article 83, 10 pages. https://doi.org/10.1145/3337821.3337906

[25] Jan Hückelheim and Laurent Hascoët. 2021. Source-to-Source Automatic Differentiation of OpenMP Parallel Loops. https://doi.org/10.48550/ARXIV.2111.01861

[26] Tim Kaler, Tao B. Schardl, Brian Xie, Charles E. Leiserson, Jie Chen, Aldo Pareja, and Georgios Kollias. [n.d.]. *PARAD: A Work-Efficient Parallel Algorithm for Reverse-Mode Automatic Differentiation.* 144–158. https://doi.org/10.1137/1.9781611976489.11 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611976489.11

[27] Rasmus Wriedt Larsen and Troels Henriksen. 2017. Strategies for Regular Segmented Reductions on GPU. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing* (Oxford, UK) *(FHPC 2017)*. ACM, New York, NY, USA, 42–52. https://doi.org/10.1145/3122948.3122952

[28] Claire Lauvernet, Laurent Hascoët, François-Xavier Le Dimet, and Frédéric Baret. 2012. Using Automatic Differentiation to Study the Sensitivity of a Crop Model. In *Recent Advances in Algorithmic Differentiation*, Shaun Forth, Paul Hovland, Eric Phipps, Jean Utke, and Andrea Walther (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 59–69.

[29] Lianfa Li. 2020. Optimal Inversion of Conversion Parameters from Satellite AOD to Ground Aerosol Extinction Coefficient Using Automatic Differentiation. *Remote Sensing* 12, 3 (2020). https://doi.org/10.3390/rs12030492

[30] Duane Merrill and Michael Garland. 2016. *Single-pass Parallel Prefix Scan with Decoupled Lookback.* NVIDIA Technical Report NVR-2016-002, March 2016. NVIDIA. https://research.nvidia.com/sites/default/files/pubs/2016-03_Single-pass-Parallel-Prefix/nvr-2016-002.pdf

[31] William S. Moses and Valentin Churavy. 2020. Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. In *Advances in Neural Information Processing Systems 33.*

[32] William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. 2021. Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 61, 16 pages. https://doi.org/10.1145/3458817.3476165

[33] William S. Moses, Sri Hari Krishna Narayanan, Ludger Paehler, Valentin Churavy, Michel Schanen, Jan Hückelheim, Johannes Doerfert, and Paul Hovland. 2022. Scalable Automatic Differentiation of Multiple Parallel Paradigms through Compiler Augmentation. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–18. https://doi.org/10.1109/SC41404.2022.00065

[34] Philip Munksgaard, Svend Lund Breddam, Troels Henriksen, Fabian Cristian Gieseke, and Cosmin Oancea. 2021. Dataset Sensitive Autotuning of Multi-versioned Code Based on Monotonic Properties. In *Trends in Functional Programming*, Viktória Zsók and John Hughes (Eds.). Springer International Publishing, Cham, 3–23.

[35] P. Munksgaard, T. Henriksen, P. Sadayappan, and C. Oancea. 2022. Memory Optimizations in an Array Language. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*. IEEE Computer Society, Los Alamitos, CA, USA, 424–438. https://doi.ieeecomputersociety.org/

[36] Andreas Nicolaisen and Marco Aslak Persson. 2020. *Implementing Single-Pass Scan in the Futhark Compiler.* Master's thesis. Department of Computer Science, Faculty of Science, University of Copenhagen, https://futhark-lang.org/student-projects/marco-andreas-scan.pdf. https://futhark-lang.org/student-projects/marco-andreas-scan.pdf

[37] Cosmin E. Oancea, Christian Andreetta, Jost Berthold, Alain Frisch, and Fritz Henglein. 2012. Financial Software on GPUs: Between Haskell and Fortran. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing* (Copenhagen, Denmark) *(FHPC '12)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2364474.2364484

[38] Cosmin E. Oancea and Alan Mycroft. 2008. Set-Congruence Dynamic Analysis for Software Thread-Level Speculation. In *Procs. Langs. Comp. Parallel Computing.* 156–171.

[39] Cosmin E. Oancea and Lawrence Rauchwerger. 2013. A Hybrid Approach to Proving Memory Reference Monotonicity. In *Languages and Compilers for Parallel Computing*, Sanjay Rajopadhye and Michelle Mills Strout (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 61–75.

[40] Daniel O'Malley, Javier E. Santos, and Nicholas Lubbers. 2022. Interlingual Automatic Differentiation: Software 2.0 between PyTorch and Julia. Association for the Advancement of Artificial Intelligence.

[41] Valérie Pascual and Laurent Hascoët. 2018. Mixed-language automatic differentiation. *Optimization Methods and Software* 33, 4-6 (2018), 1192–1206. https://doi.org/10.1080/10556788.2018.1435650 arXiv:https://doi.org/10.1080/10556788.2018.1435650

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.

[43] Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the Point: Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming. *Proc. ACM Program. Lang.* 5, ICFP, Article 88 (aug 2021), 29 pages. https://doi.org/10.1145/3473593

[44] Adam Paszke, Matthew J. Johnson, Roy Frostig, and Dougal Maclaurin. 2021. Parallelism-Preserving Automatic Differentiation for Second-Order Array Languages. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing* (Virtual, Republic of Korea) *(FHPNC 2021)*. Association for Computing Machinery, New York, NY, USA, 13–23. https://doi.org/10.1145/3471873.3472975

[45] Barak A. Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-Mode AD in a Functional Framework: Lambda the Ultimate Backpropagator. *ACM Trans. Program. Lang. Syst.* 30, 2, Article 7 (March 2008), 36 pages. https://doi.org/10.1145/1330017.1330018

[46] B. Pinty, M. Clerici, T. Lavergne, T. Kaminski, M. Taberner, and I. Andredakis. 2009. Application of Automatic Differentiation technique to retrieve land surface parameters and associated uncertainties from satellite products.. In *EGU General Assembly Conference Abstracts (EGU General Assembly Conference Abstracts)*. 7439.

[47] Max Sagebaum, Tim Albring, and Nicolas R. Gauger. 2018. Expression templates for primal value taping in the reverse mode of algorithmic differentiation. *Optimization Methods and Software* 33 (2018), 1207 – 1231. https://api.semanticscholar.org/CorpusID:52985890

[48] Max Sagebaum, Tim Albring, and Nicolas R. Gauger. 2019. High-Performance Derivative Computations Using CoDiPack. 45, 4, Article 38 (dec 2019), 26 pages. https://doi.org/10.1145/3356900

[49] M. Sambridge, P. Rickwood, N. Rawlinson, and S. Sommacal. 2007. Automatic differentiation in geophysical inverse problems. *Geophysical Journal International* 170, 1 (07 2007), 1–8. https://doi.org/10.1111/j.1365-246X.2007.03400.x arXiv:https://academic.oup.com/gji/article-pdf/170/1/1/39581500/gji_170_1_1.pdf

[50] R. Schenck, O. RÃ¸nning, T. Henriksen, and C. E. Oancea. 2022. AD for an Array Language with Nested Parallelism. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*. IEEE Computer Society, Los Alamitos, CA, USA, 829–843. https://doi.ieeecomputersociety.org/

[51] Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. 2019. Efficient Differentiable Programming in a Functional Array-Processing Language. *Proc. ACM Program. Lang.* 3, ICFP, Article 97 (jul 2019), 30 pages. https://doi.org/10.1145/3341701

[52] Jeffrey Mark Siskind and Barak A. Pearlmutter. 2018. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimization Methods and Software* 33, 4-6 (2018), 1288–1330. https://doi.org/10.1080/10556788.2018.1459621 arXiv:https://doi.org/10.1080/10556788.2018.1459621

[53] Chengbo Wang. 2020. *Financial Applications of Algorithmic Differentiation.* Ph.D. Dissertation. Advisor(s) Maciej, Klimek,. AAI28078040.

9 APPENDIX

For completeness, figure 11 shows the Futhark implementation of the PPAD rule for reverse differentiating scan [44], which we have used in our evaluation.

```
def op_bar_1 't (op : t → t → t)
                (x: t, y: t, r_b: t) : t =
  let op' b a = op a b in  vjp (op' y) x r_b

def op_bar_2 't (op : t → t → t)
                (x: t, y: t, r_b: t) : t =
  vjp (op x) y r_b

def op_lft 't (plus: t → t → t) (op : t → t → t)
              (x1: t, a1: t, y1_h: t)
              (_x2: t, a2: t, y2_h: t) : (t, t, t) =
  let z = plus (op_bar_1 op (x1, a1, y2_h)) y1_h
  in  (x1, op a1 a2, z)

def scan_bar [n] 't (zero: t) (plus: t → t → t)
       (op :  t → t → t) (e : t) (u : [n]t)
       (x_b : [n]t) : [n]t =
  let x = scan op e u
  let u_lft = map (λi → if i<n-1 then u[i+1] else e)
                  (iota n)
  let m = zip3 x u_lft x_b
  let (_, _, x_hat) = unzip3 <|
        scan_right (op_lft plus op) (e, e, zero) m
  let x_rht = map (λi → if i==0 then e else x[i-1])
                  (iota n)
  in  map (op_bar_2 op) (zip3 x_rht u x_hat)
```

Figure 11: Futhark Implementation of PPAD [44] Reverse-AD Rule for Scan