AD for an Array Language with Nested Parallelism

Anonymous Author(s)

Abstract

We present a technique for applying AD on a higher-order functional array language with support for nested parallelism. We have implemented the AD transformation in the optimising compiler for the Futhark language, in order to generate high-performance code for parallel execution on GPUs. We show how to efficiently differentiate standard constructs such as map, scan, and reduce, including how to handle free variables in lambda terms. We also discuss how to view AD of sequential loops as a time/space tradeoff that can be balanced via stripmining. We benchmark our implementation of AD on nine benchmarks against other state-of-the-art implementations, and demonstrate competitive performance on both sequential CPU and parallel GPU execution.

Keywords: automatic differentiation, functional data parallel language, compilers, GPGPU.

1 Introduction

Automatic differentiation (AD) is a practical way for computing derivatives of functions that are expressed as programs. AD of sequential code is implemented in tools such as Tapenade [2], ADOL-C [9], and Stalingrad [21]. Modern deep learning is built on array programming frameworks such as Tensorflow [1] and PyTorch [19], which provide implicitly parallel bulk operations that support AD.

An unsolved challenge is applying AD on *higher-order* parallel programming languages that combine the performance of low-level numerical languages with the flexibility of high-level functional languages, such as Dex [20] and Futhark [12]. Efficient AD for such languages would allow even more advanced machine learning models to be trained.

Machine learning involves minimising a *cost function*, which typically has far more inputs than outputs. The *reverse mode* of AD is the most efficient in such cases [3], but is challenging to implement because intermediate program values are required by the differentiated code. The program must first run a *forward sweep* that stores intermediate program states on the *tape*, which is then read during the *return sweep*, which essentially runs the program in reverse, and actually computes the derivative.

A significant amount of work has studied how to elegantly model reverse mode AD as a compiler transformation, and how to hide the tape under powerful programming abstractions such as closures [21] and delimited continuations [29]. These abstractions are not suited for efficient parallel execution on manycore hardware such as GPUs, which is the popular choice for running ML algorithms.

This paper is to our knowledge the first to efficiently implement reverse mode AD as a compiler transformation on a purely functional data parallel language that supports nested parallelism by means of higher-order array combinators, such as map, reduce, scan, and scatter.

Specifically, in order to model the tape we take inspiration from the idea [3] that applying reverse AD to a straight line of side-effect-free code does not require any tape per se, because all intermediate values remain available. We expand this idea to drive the code-transformation across lexical scopes by requiring that whenever the return sweep enters a new scope s, it first redundantly re-executes the forward sweep of s in order to bring all the variables that are needed into scope. This of course requires checkpointing the variables that are variant through sequential loops.

This technique preserves the work and span asymptotics because the recomputation overhead is at worst proportional to the depth of the deepest nest of scopes, which is constant for a nonrecursive program. Moreover, the overhead is tunable by classic compiler transformations: flattening parallel constructs eliminates the redundant-execution overhead free of charge. Loops unveil an essential time-space tradeoff: stripmining a loop k times increases the re-execution overhead by a $k\times$ factor, but exponentially decreases the memory overhead of checkpointing.

Having designed the glue that binds scopes together, we turn our attention to deriving high-level rewrite rules for differentiating the parallel operators of the language. We achieve this by starting from the main rewrite rule of the reverse mode[1] and extend it by applying reasoning that combines imperative (dependence analysis, loop distribution) and functional thinking (rewrite rules, recurrences as scans).

In particular, the simplest parallel operator, namely map, is the most difficult one to translate, because its purely-functional semantics allows free variables to be freely read inside it, but reverse AD replaces a read with an accumulation, which cannot be in general represented as a combination of classical data-parallel constructs.

Our overall contribution is an end-to-end algorithm of AD with support for nested parallel combinators, implemented inside the compiler for the Futhark language. Our specific contributions are:

PL'18, January 01–03, 2018, New York, NY, USA
2018.

[1] An original-program statement $x = f(a)$ produces the differentiated code $\overline{a} \mathrel{+}= \frac{\partial f(a)}{\partial a} \cdot \overline{x}$. where $\overline{x}$ denotes the adjoint of program variable x.

- An analysis of the time/space tradeoff when checkpointing loops for AD, where one can balance recomputation and memory usage through stripmining.
- A set of rewrite rules for differentiating higher-order parallel combinators, including uses of free variables.
- A collection of optimisations that detect patterns in the differentiated code and rewrites it to make use of specific language constructs that can benefit from specialized code generation.
- An experimental evaluation that validates the efficiency of our framework, as implemented in a real optimising compiler, on a sequential benchmark suite and on a collection of GPU benchmarks.

2 Preliminaries

This section provides the gist of how automatic differentiation of programs may be derived from its mathematical foundation. Given a function $P : \mathbb{R}^a \to \mathbb{R}^d$, the (total) derivative of P at a point $\mathbf{x} \in \mathbb{R}^a$ (where P is differentiable) can be written as a function of $\mathbf{y} \in \mathbb{R}^a$ in terms of its Jacobian as $\frac{\partial P(\mathbf{x})}{\partial \mathbf{x}}(\mathbf{y}) = J_P(\mathbf{x}) \cdot \mathbf{y}$, where $\cdot$ is matrix multiplication.

The Jacobian is a representation of a function derivative at a specific point. The *chain rule* provides a straightforward way of deriving the derivative across function composition. For example, if $P(\mathbf{x}) = h(g(f(\mathbf{x})))$, and the types of f, g, h are $f : \mathbb{R}^a \to \mathbb{R}^b, g : \mathbb{R}^b \to \mathbb{R}^c$, and $h : \mathbb{R}^c \to \mathbb{R}^d$, then:

$$J_P(\mathbf{x}) = J_h\left(g(f(\mathbf{x}))\right) \cdot J_g\left(f(\mathbf{x})\right) \cdot J_f(\mathbf{x}). \tag{1}$$

Since matrix multiplication is associative, there are multiple ways to evaluate $J_P(\mathbf{x})$ when it's viewed as a program. For example, from right-to-left:

$$\overrightarrow{J_P}(\mathbf{x}) = J_h\left(g(f((\mathbf{x}))\right) \underbrace{\left[J_g\left(f(\mathbf{x})\right) J_f(\mathbf{x})\right]}_{\bigstar},$$

or left-to-right:

$$\overleftarrow{J_P}(\mathbf{x}) = \overbrace{\left[J_h\left(g(f(\mathbf{x}))\right) J_g\left(f(\mathbf{x})\right)\right]}^{\blacksquare} J_f(\mathbf{x}).$$

Notice that if h is a scalar function (i.e., $d = 1$), then $J_h\left(g(f(\mathbf{x})\right)$ is a gradient vector of dimension $\mathbb{R}^d$. In the right-to-left formulation, $\bigstar$ is a large matrix of dimension $\mathbb{R}^{c \times a}$, whereas in the left-to-right formula, $\blacksquare$ is a vector of dimension $\mathbb{R}^b$. When $d \ll a$, it is much more expensive to compute/manifest the larger intermediate Jacobian matrices in the right-to-left evaluation order. When $a \ll d$, it's more efficient to choose the right-to-left evaluation order. There's a catch: the Jacobians depend on intermediate results, e.g., $J_g(f(\mathbf{x}))$ depends on $f(\mathbf{x})$. In the right-to-left evaluation order, the computation of these intermediate results can be interwoven with the computation of the Jacobians, as the Jacobians are computed in program-order. In the left-to-right evaluation there's no such luck: $P(\mathbf{x})$ must first be executed, with all intermediate results saved, before the $J_P(\mathbf{x})$ may be computed. For this reason, when $a \approx d$ the right-to-left evaluation order is preferred.

(a)

```
let P(x0, x1) =
  let v0 = sin(x0)
  let v1 = x1 * v0
  let v2 = x0 * x1
  let y0 = v1
  let y1 = v2
  in (y0, y1)
```

(b)

```
let →P(x0, x1, ẋ0, ẋ1) =
  let v0 = sin(x0)
  let v̇0 = sin(x0) * ẋ0
  let v1 = x1 * v0
  let v̇1 = v0 * ẋ1 + x1 * v̇0
  let v2 = x0 * x1
  let v̇2 = x1 * ẋ0 + x0 * ẋ1
  let y0 = v1
  let ẏ0 = v̇1
  let y1 = v2
  let ẏ1 = v̇2
  in (y0, y1, ẏ0, ẏ1)
```

(c)

```
let ←P(x0, x1, ȳ0, ȳ1) =
  let v0 = sin(x0)
  let v1 = x1 * v0
  let v2 = x0 * x1
  let y0 = v1
  let y1 = v2
  let v̄2 = ȳ1
  let v̄1 = ȳ0
  let x̄0 = x1 * v̄2
  let x̄1 = x0 * v̄2
  let x̄1 += v0 * v̄1
  let v̄0 += x1 * v̄1
  let x̄0 += cos(x0) * v̄0
  in (y0, y1, x̄0, x̄1)
```

Figure 1. (a) A program P which computes the function $f(x_0, x_1) = (x_1 \cdot sin(x_0), x_0 \cdot x_1)$, **(b)** the forward mode AD transformation of the program, and **(c)** the reverse mode AD transformation of the program.

2.0.1 Forward mode AD.

Consider the example program P in Figure 1a. The *forward mode* AD transformation of P, $\overrightarrow{P}$, is shown in Figure 1b; the transformation involves the core rewrite rule:[2]

$$\texttt{let } v = f(a,b) \implies \begin{array}{l}\texttt{let } v = f(a,b)\\ \texttt{let } \dot{v} = \dfrac{\partial f(a,b)}{\partial a}\dot{a} + \dfrac{\partial f(a,b)}{\partial b}\dot{b}\end{array} \tag{2}$$

where we call $\dot{v}$ the *tangent* of v, which is the derivative of v with respect to a chosen direction $(\dot{x_0}, \dot{x_1})$ (now appearing as an additional argument to $\overrightarrow{P}$). $\overrightarrow{P}$ also returns the tangents of its outputs, $\dot{y_0}$ and $\dot{y_1}$, which make-up the derivative of P. We expose forward mode AD to the user via a function

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

which, for any differentiable program P computes $\overrightarrow{P}$, with the property that $(\overrightarrow{P}(\mathbf{x}))(\dot{\mathbf{x}}) = \overrightarrow{J}_P(\mathbf{x})\dot{\mathbf{x}}$. [3] In simpler terms, $\overrightarrow{P}$ takes a point and a direction and returns the derivative of P at the point and in the given direction by multiplying the direction on the right of the right-to-left association of the Jacobian for P. In order to recover the full Jacobian $\overrightarrow{J}_P(\mathbf{x})$, $\overrightarrow{P}(\mathbf{x})$ is mapped over the standard basis of $\mathbb{R}^n$.

2.0.2 Reverse mode AD.

The *reverse mode* AD transformation of P, $\overleftarrow{P}$, is shown in Figure 1c. As with the forward mode transformation, it involves the core rewrite rule:[4]

$$\texttt{let } v = f(a,b) \implies \begin{array}{l}\texttt{let } v = f(a,b)\\ \vdots\\ \texttt{let } \overline{a} \mathrel{+}= \dfrac{\partial f(a,b)}{\partial a}\overline{v}\\ \texttt{let } \overline{b} \mathrel{+}= \dfrac{\partial f(a,b)}{\partial b}\overline{v}\end{array} \tag{3}$$

[2] This rule generalizes for n-array functions in the obvious way: for any statement $\texttt{let } v = f(a_0, \ldots, a_{n-1})$, we have $\dot{v} = \sum_{i=0}^{n-1} \frac{\partial f}{\partial a_i}\dot{a}_i$.

[3] jvp stands for *Jacobian-vector product*, named as such because we apply the direction vector on the right of the Jacobian.

[4] In general, the adjoint of a variable a appearing on the RHS of statements $\texttt{let } v_0 = f_0(.., a, ..), \ldots, \texttt{let } v_{n-1} = f_{n-1}(.., a, ..)$ is $\overline{a} = \sum_{i=0}^{n-1} \frac{\partial f}{\partial a}\overline{v_i}$.

where $\overline{v}$ is the *adjoint* of v, which is the derivative of P with respect to v. The adjoint of a variable v may be updated multiple times–any use of v may contribute to the derivative (see, e.g., x_1 in Figure 1c). Reverse mode AD corresponds to computing the left-to-right Jacobian $\overleftarrow{J_P}(\mathbf{x})$. All intermediate results of P which appear in adjoint expressions must first be saved, which corresponds to computing P before the adjoint of each program variable. The $\vdots$ indicates the presence of these statements, along with any preceding adjoint expressions. The adjoints in Figure 1c are computed in the reverse program order, corresponding to a left-to-right association of the Jacobian of P. Any adjoint $\overline{v}$ is necessarily finalized before it is used: all uses of v must occur after assignment; any contributions to $\overline{v}$ must necessarily appear before any uses of $\overline{v}$ in reverse program order. $\overleftarrow{P}$ takes as additional arguments the initial adjoints of the output $\overline{y_0}$, $\overline{y_1}$ and additionally returns the adjoints of the input, $\overline{x_0}$, $\overline{x_1}$. Reverse mode AD is made available to the user via a function:

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

which computes $\overleftarrow{P}$ for any differentiable program P, with the property that $(\overleftarrow{P}(\mathbf{x}))(\overline{\mathbf{y}}) = \overline{\mathbf{y}}\overleftarrow{J}_P(\mathbf{x})$. [5] As $\overleftarrow{P}$ has the same computational properties as $\overleftarrow{J}_P(\mathbf{x})$, it is the preferred choice when $m \ll n$. When P computes a scalar function (i.e., $m = 1$), vjp returns the complete gradient of P in a single pass; jvp requires n passes, one for each basis vector of $\mathbb{R}^n$. Programs with very high-dimensional inputs and low-dimensional outputs appear frequently in the fields of optimization and machine learning, making vjp the computationally superior choice. The vjp transformation is also more complex due to computing adjoints in reverse order, storing of intermediate program variables, and accumulation of derivatives into adjoint variables, which turn any read of a variable in the original program into a write in the transformed program. For these reasons, we concentrate our discussion on reverse mode AD.

2.1 Language

We perform our transformation on a data-parallel language which features arbitrary nesting of *second-order array combinators* (SOACs), e.g., **map**, **reduce**, and **scan**. SOACs are variadic in their number of arguments and returns, i.e., **zip**s and **unzip**s are implicit. For example,

$$\textbf{unzip3}\ (\textbf{map}\ (\lambda(a,b,c) \to (c,b,a))\ (\textbf{zip3}\ as\ bs\ cs))$$

may be equivalently written as simply

$$\textbf{map}\ (\lambda a\ b\ c \to (c,b,a))\ as\ bs\ cs$$

The source language supports higher-order functions, polymorphism, modules, and similar high-level features, which are compiled away using a variety of techniques [8, 13] before we perform AD. The only remaining higher-order functions are the SOACs. Further, a significant battery of standard optimisations (CSE, constant folding, aggressive inlining) is also applied prior to AD.

The language is written in A-normal form [25] (ANF): all subexpressions are variables names or constants with the exception of the body expression of **loops**, **if**-expressions and **let**-expressions. **let**-expressions consist of a series of bindings—that we also call statements—followed by a sequence of one or more returns:

$$\begin{aligned}&\textbf{let}\ a = 5 * x\\&\textbf{let}\ b = a * a\\&\textbf{in}\ b\end{aligned}$$

The language is purely functional: re-assignments to the same variable (in a given scope) should be understood as a notational convenience for variable shadowing. It supports a functional flavor of in-place updates based on uniqueness types [12]. The binding $\textbf{let}\ xs[i] = x$ is syntactic sugar for $\textbf{let}\ xs' = xs\ \textbf{with}\ [i] \leftarrow x$, which has the semantics that xs' is a copy of xs in which the element at index i is updated to x, but also provides the operational guarantee that the update will be realized in place.

The language also features sequential and pure loops, which have the semantics of a tail-recursive function:

$$\textbf{let}\ y = \textbf{loop}\ (x) = (x_0)\ \textbf{for}\ i = 0 \ldots n-1\ \textbf{do}\ e$$

The loop is initialized by binding x_0 to x. Each iteration of the loop executes e , and binds the result of the expression to x, which is used on the subsequent iteration of the loop. The loop terminates after n iterations and the final result of e is bound to y. We introduce the remainder of the language on a by-need basis.

3 Forward mode

We implement forward mode AD by straight-forward application of the forward mode rewrite rule (2). That is, tangent variables statements are interleaved with primal variable statements (tangent variables are associated with primal variables via a simple mapping). In parameters, body results, and SOAC arguments we bundle tangent variables together with their primal counterpart.

As an alternative to a program transformation, forward mode AD can be implemented by coupling primal and tangent variables together in a tuple, e.g., $(v, \dot{v}) = (f(a), \frac{df}{da}\dot{a})$. Via overloading, the operators in the program may operate directly on such tuples, giving rise to the operator-overloading/dual number formulation of forward-mode AD described in [3]. Our approach of primal and tangent statements and bundling of primal and tangent variables is equivalent to this formulation of AD, implemented as a compiler pass instead of via operator overloading.

Forward AD is total and closed on the IR: that is, any Futhark IR program can be differentiated an arbitrary number of times. This is crucial for supporting higher-order derivatives, as our reverse mode AD transformation is *not* total and closed: the reverse-mode derivative of a program can,

[5] vjp stands for *vector-Jacobian product*.

Rule vjp_{body} refers to a body $body = stms$ `in` res:

$$vjp_{body}(\overline{res},\ stms \texttt{ in } res) \Rightarrow \overleftarrow{stms} \texttt{ in } (res,\ \overline{fvs_{body}})$$
$$\texttt{where } \overleftarrow{stms} \leftarrow vjp_{stms}(stms) \texttt{ and } \overline{fvs_{body}} \leftarrow \overline{\mathcal{FV}(body)}$$

Rule vjp_λ refers to a lambda function $\lambda x_1 \ldots x_n \rightarrow body$:

$$vjp_\lambda(\overline{res},\ \lambda x_1 \ldots x_n \rightarrow stms \texttt{ in } r) \Rightarrow \lambda x_1 \ldots x_n \rightarrow \overleftarrow{body}$$
$$\texttt{where } \overleftarrow{stms} \texttt{ in } (\_, \overline{fvs_{body}}) \leftarrow vjp_{body}(\overline{res},\ body)$$
$$\overleftarrow{body} \leftarrow \overleftarrow{stms} \texttt{ in } \overline{fvs_{body}}$$

Rule vjp_{stms} simply folds over each statement:

$$vjp_{stms}(stm, stms) \Rightarrow \overrightarrow{stms},\ vjp_{stms}(stms),\ \overleftarrow{stms}$$
$$\texttt{where } (\overrightarrow{stms}, \overleftarrow{stms}) \leftarrow vjp_{stm}(stm)$$

Rule vjp_{stm} for scalar multiplication (simple) statement:

$$vjp_{stm}(\texttt{let } x = a * b) \Rightarrow (\overrightarrow{stms},\ \overleftarrow{stms})$$
$$\texttt{where } \overrightarrow{stms} \leftarrow \texttt{let } x = a * b$$
$$\overleftarrow{stms} \leftarrow \texttt{let } \overline{a} = \overline{a} + b * \overline{x},\ \texttt{let } \overline{b} = \overline{b} + a * \overline{x}$$

Rule vjp_{stm} for loop statement: checkpoints and re-executes body
(Applies only to loops whose variants do not change their shape within the loop)

$$vjp_{stm}(\texttt{let } y = \texttt{loop } (x) = (x_0) \texttt{ for } i = 0 \ldots n-1 \texttt{ do } body) \Rightarrow (\overrightarrow{loop},\ \overleftarrow{loop})$$
$$\texttt{where}$$
$$stms \texttt{ in } r \leftarrow body$$
$$\overrightarrow{loop} \leftarrow \texttt{let } xs_0 = \texttt{scratch}(n, \texttt{size}(\texttt{typeOf}(x))),$$
$$\texttt{let } (y, xs) = \texttt{loop } (x, xs) = (x_0, xs_0)$$
$$\texttt{for } i = 0 \ldots n-1 \texttt{ do let } xs[i] = x,\ stms \texttt{ in } (r, xs)$$
$$\overline{fvs_{bdy}} \leftarrow \overline{\mathcal{FV}(body)}$$
$$\overleftarrow{stms} \texttt{ in } (\_, \overline{fvs'_{bdy}}) \leftarrow vjp_{body}(\overline{x},\ body)$$
$$(\_, \overleftarrow{stm_{x_0}}) \leftarrow vjp_{stm}(\texttt{let } x' = x_0)$$
$$\overleftarrow{loop} \leftarrow \texttt{let } \overline{fvs_{bdy_0}} = \texttt{get or initialize adjoints},$$
$$\texttt{let } (\overline{x'}, \overline{fvs_{bdy}}) =$$
$$\texttt{loop } (\overline{x}, \overline{fvs_{bdy}}) = (\overline{y}, \overline{fvs_{bdy_0}})$$
$$\texttt{for } i = n-1 \ldots 0 \texttt{ do}$$
$$\texttt{let } x = xs[i],\ \overleftarrow{stms} \texttt{ in } (\overline{r}, \overline{fvs'_{bdy}}),$$
$$\overleftarrow{stm_{x_0}}$$

Figure 2. Reverse AD transformation rules for body of statements, scalar multiplication and do loops.

in general, not be reverse differentiated again due to the possible presence of accumulators (discussed in Sec. 5.4). However, this limitation is of no concern: a program $P : \mathbb{R}^a \rightarrow \mathbb{R}^b$'s n-th derivative P^n (obtain by n applications of automatic differentiation) has type $P^n : \mathbb{R}^a \rightarrow \mathbb{R}^{b \times a^n}$. For all $n \geq 1$, forward mode is more efficient than reverse because $b \times a^n > a$. That is, reverse mode differentiation is only more efficient on primal programs and never on the derivatives of programs.

4 Redundant Execution Instead of Tape

Figure 2 sketches the reverse-mode AD code transformation at the lexical levels of a body of statements (vjp_{body}), a sequence of statements (vjp_{stms}) and individual statements (vjp_{stm}). We recall that *body* refers to a sequence of **let** bindings (statements) followed by an **in** result. Our presentation sacrifices some formalism for readability, for example, by omitting the environment, which essentially carries out a mapping from the variables of the source program to their respective adjoints. Instead, we assume that the adjoint $\overline{x}$ of a variable x is always available.

The vjp_{body} rule refers to transforming a body of statements, which defines a new scope. The rule starts by binding the body result to its adjoint $res \mapsto \overline{res}$ (not shown). This is safe because the transformation works backwards, hence $\overline{res}$ is already available from the outer scope (ultimately passed as parameter to $vjp_{function}$ by the user). The statements of the transformed body $\overleftarrow{stms}$ are those generated by vjp_{stms} and the result consists of the original result res, extended with the adjoints $\overline{fvs_{body}}$, consisting of all free variables that are used inside the body—$\mathcal{FV}(body)$ finds the free variables used in the body and $\overline{\mathcal{FV}(body)}$ refers to their adjoints.

Rule vjp_λ refers to an un-named (lambda) function, and its reverse-AD code is essentially obtained by calling vjp_{body} on lambda's body. Note that $x_{1,\ldots,n}$ are free variables in $body$, and as such their adjoints are among $\overline{fvs_{body}}$.

The vjp_{stms} rule highlights the redundant-execution mechanism that removes the need to implement the tape as a separate abstraction: each statement stm is processed individually (by vjp_{stm}), producing a sequence of statements on the forward sweep, denoted $\overrightarrow{stms}$, that brings into scope whatever information is necessary to execute the return sweep for that statement, denoted by $\overleftarrow{stms}$.[6]

For example, the forward sweep of a multiplication statement $\texttt{let } x = a * b$ is the statement itself. Re-executing it brings into scope the (value of) variable x which may be needed in the return sweep of a following statement, e.g., $\texttt{let } y = x * a$, which would require an update to the adjoint of a according to the rewrite rule of Equation (3): $\overline{a}\ +\!= \frac{\partial(x*a)}{\partial a} * \overline{y}$, which results in $\overline{a}\ +\!= x * \overline{y}$.

The rule for a loop statement is more complex. Figure 2 shows a loop that exhibits a loop-variant variable x—whose value changes through the loop—which is initialized with x_0 just before the loop is started, and whose value for the next iteration is bounded to the result of the loop body. The result of the entire loop is bounded to variable y. The rule assumes that the shape of x does not change throughout the loop.

The forward sweep $\overrightarrow{loop}$ is the original loop, except that its body is modified to checkpoint into array xs the value of x at the entry of each iteration. As such xs is also declared as loop variant and initialized to xs_0, which, in turn, is allocated (by **scratch**) just before the loop statement. Note that *only the loops of the current scope are checkpointed*; an inner-nested loop would be re-executed but not checkpointed.

The return sweep consists mainly of a loop that iterates with i from $n-1$ down to 0: the first statement re-installs the value of x for the current iteration from the checkpoint (i.e., $xs[i]$). The remaining statements, $\overleftarrow{stms}$, are those generated by vjp_{body} from the original loop body, including its forward

[6] The statements of the forward and return sweeps are arranged symmetrically; all statements of the forward sweep come before any of the return sweep, and the latter is organized in the reverse order of the original statements: $vjp_{stms}(stm, stms) \Rightarrow \overrightarrow{stms}, vjp_{stms}(stms), \overleftarrow{stms}$.

sweep. As $\overleftarrow{stms}$ may use x, re-installment is necessary. Re-execution of the forward sweep brings into scope all variables that might possibly be used by the return sweep of the body.

The result of a reversed iteration is the adjoint of the original result $\overline{r}$, together with the (updated) adjoints of the free variables used in the loop $\overline{fvs'_{bdy}}$. These are declared as variant through the loop ($\overline{fvs_{bdy}}$), such that the updates of all iterations are recorded. $\overline{fvs_{bdy_0}}$ is the adjoint of the free variables prior to entering the loop; if a free variable was used in a prior statement of the same scope (and hence has an existing potentially non-zero adjoint) its adjoint is simply retrieved. Otherwise, its adjoint is initialized to a zero element of the appropriate type and shape.

The final statement of the return sweep is $\overleftarrow{stm_{x_0}}$; it semantically updates the adjoint of the loop-variant initializer $\overline{x_0}$ with the (adjoint) result $\overline{x'}$ of the return sweep loop—the code is generated by $vjp_{stm}(\textbf{let } x' = x_0)$.[7]

Sequential loops are the only construct that require iteration checkpointing: *parallel constructs do not.*

4.1 Array indexing

The adjoint $\overline{a}$ of an an array variable a of type $[d]\tau$ is itself of type $[d]\tau$. When generating code for the return sweep of a statement $\textbf{let } y = a[i]$, we must update $\overline{a[i]}$ with the "contribution" of $\overline{y}$. This leads us to the following rule:

$$vjp_{stm}(\textbf{let } y = a[i]) \Rightarrow (\textbf{let } y = a[i], \textbf{let } \overline{y} = \textbf{upd } i\ \overline{y}\ \overline{a})$$

The **upd** construct merits further elaboration. *Semantically*, **upd** $i\ v\ a$ returns a but with the value at index i changed to be $v + a[i]$. *Operationally*, the array a is directly modified in-place. To preserve purely functional semantics, we require that the "old" a and its aliases is never accessed again, similar to the in-place updates of Section 2.1

4.2 Navigating the Optimization & Trade-off Space

Figure 3, Listing 3, demonstrates the result of the vjp_{stms} transformation for the source code shown in Listing 1, which consists of a loop surrounded by statements in the same scope ($stms^{bef}_{out}, stms^{after}_{out}$). The drawback of our strategy is that, essentially, the original loop is executed twice: once on the forward sweep when it is checkpointed ($stms_{loop}$), and a second time on the return sweep, where the forward sweep of the loop body ($\overrightarrow{stms_{loop}}$) is re-executed to bring into scope all variables in the loop body that might be needed by the return sweep of the current iteration ($\overleftarrow{stms_{loop}}$). In general, the worst-case recomputation overhead is proportional with the depth of the deepest scope nest of the program, which is constant for a given program, hence the work complexity is preserved.

[7] This is because the first (implicit) statement of the original loop is $\textbf{let } x = x_0$, hence it is differentiated last on the return sweep. Moreover, the loop returns the (updated) adjoint of its variant variable x, which is bound/saved in $\overline{x'}$, and is thus used to update the adjoint of x_0.

Listing (1) Original Loop

```
stms^bef_out
let y'' =
  loop (y)=(y_0)
  for i = 0...m^k - 1 do
    stms_loop
    in y'
stms^after_out
```

Listing (2) Original loop Stripmined $k\times$

```
stms^bef_out
let y'' =
  loop (y_1)=(y_0)
  for i_1 = 0...m - 1 do
    ...
    loop(y_k)=(y_{k-1})
    for i_k = 0...m - 1 do
      let y = y_k
      let i = i_1*m^{k-1}+...+i_k
      stms_loop
      in y'
stms^after_out
```

Listing (3) Reverse AD Result on Original Loop

```
→stms^bef_out
let ys_0 = scratch(m^k,
                   sizeOf(y_0))
let (y'', ys) =
  loop (y,ys)=(y_0,ys_0)
  for i = 0...m^k - 1 do
    let ys[i] = y
    stms_loop
    in (y', ys)
→stms^after_out
←stms^after_out
let (ȳ''', ...) =
  loop (ȳ,...)=(ȳ'',...)
  for i = m^k - 1...0 do
    let y = ys[i]
    →stms_loop
    ←stms_loop
    in (ȳ', ...)
vjp_smt(let y''' = y_0)
←stms^bef_out
```

Figure 3. The figure shows the result of the reverse-AD transformation (3) applied to code containing one loop (1). One can notice that the original-loop statements are re-executed twice ($stms_{loop}$ and $\vec{stms}_{loop}$). One can generalize that if we stripmine the original loop k times (as in 2), and apply reverse AD on it, the re-execution factor increases to $(k+2)\times$, but the memory overhead of checkpointing increases only by $(k \cdot m)\times$ rather than $m^k\times$ with the original.

In practice, we have observed that standard optimisations (CSE, constant/copy-propagation, etc.) effectively eliminate redundancies, especially in a purely functional setting in which aggressive dead code elimination is trivial. Furthermore, the re-execution strategy is beneficial for scalars: storing and retrieving them from the tape (global memory) is typically significantly more expensive than recomputing them.

More importantly, one can observe that perfectly-nested constructs—e.g., a perfect **map** or loop nest—do not introduce any recomputation overhead.[8] It follows that this overhead can be minimized by classical compiler transformations such as map fission or loop distribution, aimed at creating perfectly-nested recurrences.[9]

In the case of perfect-loop nests, they should also be flattened into one loop because otherwise the need for checkpointing might keep them alive. This observation reveals *an essential time-space tradeoff*, which is demonstrated in Listing 2: applying the reverse AD to the stripmined loop (k

[8] This is because the result of the forward sweep—which is re-executed as part of the return sweep—cannot possibly be used in the return sweep of that scope (if the construct is perfectly nested), hence it is dead code.

[9] In the least, the reverse AD transformation should be performed before fusion.

times) would increase the re-execution factor from 2× to $k+2$×, but would exponentially decrease the memory expansion factor from m^k× (the loop count) to only $(k \cdot m)$×—because the checkpointing of each of the k stripmined loops of count m store m versions of the loop-variant variable y. Essentially, if reverse AD runs out of memory, the user can exponentially reduce the memory requirements by stripmining the guilty loop, at the expense of (only) a linear slowdown. To demonstrate, consider the following loop which computes x^y:

```
let pow y x = loop acc = 1 for _i < y do
              acc * x
```

Computing the reverse mode derivative of `pow y` at $(x, y) = (2, 10^{10})$ (without regard for overflow) utilizes 78GB of memory. Not only does stripmining reduce the memory footprint to a mere 300MB, but in this (exaggerated) example, recomputation is cheaper by a factor of 1.65×. We provide a simple annotation that performs loop stripmining.[10]

5 Rewrite Rules for Parallel Constructs

Having introduced the glue that binds the transformation across scopes, we turn our attention to the mapping of individual parallel constructs within a scope, albeit we allow arbitrary nesting of them and loops together. We organize the discussion in free-writing style fashion that focuses on the reasoning that led the derivation on the rules rather than presenting them in a formal, but dry and verbose notation.

5.1 Reduce and Multi-Reduce

We recall that the semantics of reduce, for an arbitrary binary associative operator $\odot$ and its neutral element $e_\odot$ is:

$$\texttt{reduce}\ \odot\ e_\odot\ [a_0, a_1, \ldots, a_{n-1}] \equiv a_0 \odot a_1 \odot \ldots \odot a_{n-1}$$

If the result of **reduce** is let-bound to variable y, we can reason more easily about the contribution of the reduce statement to the adjoint of a_i if we group terms as:

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

If we would know, for every i, the terms $l_i = a_0 \odot \ldots \odot a_{i-1}$ and $r_i = a_{i+1} \odot \ldots \odot a_{n-1}$, we could directly apply the main rule for reverse AD given in Equation (3), which results in:

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

where l_i and r_i are considered constants and $\overline{+}$ denotes a potentially vectorized addition that matches the datatype. The code for the right-hand side (RHS) can be generically generated as a function f that can be *mapped* to all a_i, l_i, r_i:

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

except that the adjoints of l_i and r_i are not returned by f.

Finally, all the l_i and r_i values can be computed by a forward and reverse exclusive scan (prefix sum), respectively. Essentially, the forward sweep is the reduce statement. Denoting by $as = [a_0, a_1, \ldots, a_{n-1}]$, and assuming for simplicity that $\odot$ has no free variables, the return sweep is:

```
let ls = scan^exc ⊙ e_⊙ as
let rs = reverse as ▷
         scan^exc (λx y → y ⊙ x) e_⊙ ▷ reverse
let as' += map f ls as rs
```

Essentially, the reverse AD code for an arbitrary reduce requires a map, two scan[11] and two reverse operations, which preserve the work-depth parallel asymptotics of the original program. In practice, this is quite expensive,[12] but luckily most standard operators admit more efficient translations:

5.1.1 Special Cases of Reduce Operators.

When the reduce operator is (vectorized) ***plus***, we have $\frac{\partial(l_i + a_i + r_i)}{\partial a_i}\,\overline{y} = \overline{y}$, hence the return sweep becomes **let** $\overline{as}\ \overline{+}=\ \overline{y}$, which is also derived automatically by the simplification engine.

When the reduction operator is ***multiplication***, we have $\frac{\partial(l_i * a_i * r_i)}{\partial a_i}\,\overline{y} = l_i * r_i * \overline{y}$. We discriminate three cases here:

- if all as' elements are nonzeros, then $l_i * r_i = \frac{y}{a_i}$ and $y \neq 0$, hence we update each element: $\overline{a_i}\ +=\ \frac{y}{a_i}\overline{y}$,
- if exactly one element at index i_0 is zero, then $l_i * r_i$ is zero for all other elements, and $\overline{a_{i_0}}\ +=\ y * \overline{y}$
- otherwise, $\forall i, l_i * r_i \equiv 0$, and $\overline{as}$ remains unchanged.

The forward sweep is modified to compute the number of zeros in as and the product of non-zero elements (by a map-reduce operation), followed by setting the reduced result y accordingly. The return sweep computes the contributions by a parallel map and updates adjoints as discussed before.

Finally, when the reduction operator is ***min*** (or ***max***), then only the adjoint of (one of) the minimal array element should receive the contribution of y, because it is the only one that was used to compute the result. As such the forward sweep is modified to compute the minimal element together with its (first) index i_y (the common "argmin" operation), which can be done with a parallel reduction. The forward sweep is then **let** $(y, i_y) =$ argmin as, and the return sweep could be **let** $\overline{as}[i_y]\ \overline{+}=\ \overline{y}$.

However, at this point we add a compiler improvement that exploits sparsity: For example, if $\overline{as}$ has not been created yet, then instead of inserting the update statement, we (statically) record in vjp's environment that $\overline{as}$ is sparse (currently one non-zero element). Further operators such as **reduce** `min`, may refine the sparse structure of $\overline{as}$ by adding more points to it. If as is created by a **map** operation, and $\overline{as}$ is still sparse, then we (statically) replace the adjoint code of the **map** with (only) the adjoint code of the iterations corresponding to the non-zeros in $\overline{as}$—because the other iterations have no effect.

[10] Since the loop count n is not always a power of k, we use $\lfloor \sqrt[k]{n} \rfloor$ as the count of the stripmined loops, and conclude with a loop of count $n - \lfloor \sqrt[k]{n} \rfloor^k$.

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

[12] In theory, the adjoint code should not be more than 4× slower than the original, but our general rule requires at least 5 global memory accesses per array element (2 for each scan), while the original reduce requires only one.

5.1.2 Reduce-by-Index.

Reduce-by-index [11], a.k.a. multi-reduce, essentially generalizes a histogram computation by allowing the values from an array (*as*) that fall into the same bin (index from `inds`) to be reduced with an arbitrary associative and commutative operator $\odot$ having neutral element $e_\odot$, where the number of bins m is typically assumed smaller than that of index-value pairs n, i.e.,

```
let hs = reduce_by_index (⊙) e⊙ inds as
```

has the semantics:

```
loop hs = replicate m e⊙
for i = 0 ... n-1 do
  let hs[ inds[i] ] ⊙= as[i] in hs
```

A similar reasoning as for the arbitrary **reduce** suggests that two scans need to be applied to each subset of elements that fall in the same bin, a.k.a., multi-scan, in order to compute the l_i and r_i terms for every i. Then the contributions to the adjoint of *as* of a reduce-by-index statement are computed as before by **map** `f ls as rs`.

Assuming a constant key size, the multi-scan can be implemented within the right work-depth asymptotic by (radix) sorting *as* according to the corresponding bins, and by applying an irregular segmented reduction on the result. This is straightforward but tedious, and we have not implemented it, because user-defined operators are rarely used.

Instead we have implemented the special-case operators discussed for reduce. Essentially, the forward sweep consists of the reduce-by-index statement, but enhanced with the extended operators, and the return sweep is similar to reduce, except that *as*[i] is updated with $\overline{\texttt{hs[inds[i]]}}$.

5.2 Scan or Prefix Sum

An inclusive scan [4] computes all prefixes of an array by means of an associative operator $\odot$ with neutral element $e_\odot$:

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

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 reals, and generalize later:

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

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 (3) to each statement and finally gather them back into the loop below:

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

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:

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

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

The second loop (computing $\overline{as}$) is essentially a **map** once we know the values of $\overline{rs'}$. Denoting by $c_{n-1} = 1$ and $c_i = \frac{\partial\,(rs_i \odot as_{i+1})}{\partial\, rs_i}$, the first loop (computing $\overline{rs}$) is a backward linear recurrence of the form

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

where $\overline{ys}$ is the adjoint of the result of the original loop statement, which is known before the reversed loop is entered.

Such a recurrence is known to be solved with a scan whose operator is linear-function composition [5]. To summarize: the forward sweep is the original scan. The reverse sweep consists of (1) the **map** that computes the c_i values, (2) the **scan** that computes the backward linear recurrence, and (3) the **map** that computes that updates $\overline{as}$:

```
let (ds, cs) =
    map ( λi → if (i == n-1) then (0, 1)
               else (ys'[i], ∂(rs[i] ⊙ as[i+1])/∂ rs[i]) ) [0..n-1]
let lin_o (d1,c1) (d2,c2) = (d2 + c2 · d1, c2 × c1)
let rs' = scan lin_o (0,1) (reverse ds) (reverse cs)
      ▷ map (λ di ci → di + ci · ys'[n-1]) ▷ reverse

let as' += map (λ (i, ai) → if i==0 then rs'[0]
                            else ∂(rs[i-1] ⊙ ai)/∂ ai · rs'[i]
               ) [0..n-1] as
```

For generalization, we observe that any element type can be linearized to a vector, say of size d. Then we observe that

- a term like $\frac{\partial\,(rs[i-1], a_i)}{\partial\, a_i}$ corresponds to the Jacobian of the function $\odot_{right}(y) = rs[i-1] \odot y$ at point a_i, denoted by $J_{\odot_{right}}(a_i) \in \mathbb{R}^{d \times d}$,
- a term like $\frac{\partial\,(rs[i-1], a_i)}{\partial\, a_i} \cdot \overline{rs}[i] \in \mathbb{R}^d$ applies the Jacobian to a vector and corresponds to $vjp_\lambda(\overline{rs}[i], \odot_{right})$.

It follows that in the rewrite rule above, $\cdot$ and $\times$ are generalized from scalar multiplication to matrix-vector and matrix-matrix multiplication, respectively, and $+$ to vector addition. In particular, the lin_o operator of scan, has neutral element $(0 \in \mathbb{R}^d, I \in \mathbb{R}^{d \times d})$, where I is the identity matrix, and lin_o computes, among others, one matrix multiplication. If d is a constant, e.g., tuples of scalars, the work-depth asymptotic of the original program is preserved, otherwise, this rule provides no such guarantee. We have implemented the case when scan's element type is one scalar. We also support vectorized such operators by turning them beforehand to a regular-segmented scan, by the rule:

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

We treat separately the case of scan with (vectorized) plus operator, in which the contributions to be accumulated by $\overline{as}$ are given by **scan** $(\overline{+})$ $\overline{0}$ (**reverse** $\overline{ys}$) ▷ **reverse**.

5.3 Parallel Scatter

The statement `let ys = scatter xs is vs` produces an array `ys` by updating in-place the array `xs` (which is consumed) at the m indices specified in array `is` with corresponding values taken from `vs`. Scatter has constant depth, and work proportional with m, the size of array `vs`, i.e., it does not depends of the length n of the updated array `ys`. Our rule assumes that `is` contains no duplicates, i.e., idempotent updates are currently not supported.

Denoting by **gather** the operation that reads from a support array elements at a given subset of indices, i.e.,

```
let gather arr inds = map (λi → xs[i]) inds
```

the forward sweep saves (in $\mathtt{xs}_{saved}$) the elements of xs that are about to be overwritten prior to performing the update:

```
let xs_saved = gather xs is
let ys = scatter xs is vs
```

Since the original statement performs `ys[ is[i] ] = vs[i]` for any $i \in$ `is`, we have by Equation (3) $\overline{vs}$`[i] +=` $\overline{ys}$`[is[i]]`. As such, the return sweep (1) first updates the adjoint of `vs`, then (2) creates the adjoint of `xs` by zeroing out the elements from $\overline{ys}$ that were never in `xs`, and finally, (3) restores `xs` to its state before the update:

```
let vs̄ += gather is ys̄
let xs̄ = scatter ys̄ is (replicate m 0)
let xs = scatter ys is xs_saved
```

Note that both forward and return sweep preserve the original work-depth asymptotics, because all operations have work proportional to m (not n). This would not hold if, e.g., the forward sweep would make a copy of the whole `xs`.

5.4 Map

A **map** applies a lambda function to each element of an array, producing an array of same length:

$$\textbf{let}\ xs\ =\ \textbf{map}\ (\lambda a \rightarrow stms\ \textbf{in}\ x)\ as$$

If the lambda has no free variables, the return sweep is:

$$\textbf{let}\ \overline{as} = \textbf{map}\ (\lambda(a, \overline{a}, \overline{x}) \rightarrow \overrightarrow{stms}\ \overleftarrow{stms}\ \textbf{in}\ \overline{a_0} + \overline{a})\ as\ \overline{as}\ \overline{xs}$$

where $\overrightarrow{stms}$ and $\overleftarrow{stms}$ denote the forward and reverse statements of lambda's body, and $\overline{a}$ is shadowed by $\overleftarrow{stms}$.

A naive way of handling free variables is to turn them into bound variables. E.g. converting **map** $(\lambda i \rightarrow as[i])\ is$ into **map** $(\lambda(i, as') \rightarrow as'[i])\ is$ (**replicate** $n\ as$) where n is the size of is. This is fine for scalars, but asymptotically inefficient for arrays that are only partially used, as here, as the adjoint will be mostly zeroes.

In an impure language, we could update the adjoint of a free array variable $as[i]$ with an operation $\overline{as}[i]$+=v, implemented with atomics or locks in the parallel case. In our pure setting, we instead introduce *accumulators*. An array can be "temporarily" turned into an accumulator with **withacc**[13]:

$$\textbf{withacc} : [d]\alpha \rightarrow (\textbf{acc}(\alpha) \rightarrow \textbf{acc}(\alpha)) \rightarrow [d]\alpha$$

Intuitively we view an accumulator as a "write-only" view of an array. *Semantically*, accumulators are lists of index/value pairs, each denoting an update of an array. When we use **upd** on an accumulator, we prepend index/value pair to this list, returning a new accumulator. *Operationally*, **upd** on an accumulator is implemented by immediately updating the underlying array, not by actually maintaining a list of updates in memory (although such an implementation would be semantically valid). The purpose of accumulators is to allow the compiler to continue to reason purely functionally, in particular that all data dependencies are explicit, while allowing efficient code generation based on in-place updates. Our accumulators are similar to generalized reductions [14] or the accumulation effects in Dex [20] and have the same underlying motivation. The main difference is that in Dex, they are an effect, which requires every part of the compiler to be effect-aware.

Free array-typed variables in **map** are thus turned into accumulators while generating return sweep code for the **map**, during which we can perform the updates directly. We allow implicit conversion between accumulators and arrays of accumulators, as this allows us to directly **map** them. E.g.

$$\textbf{let}\ xs\ =\ \textbf{map}\ (\lambda i \rightarrow as[i])\ is$$

results in the return sweep code

$$\textbf{let}\ \overline{as}\ =\ \textbf{withacc}\ \overline{as}\ (\lambda \overline{as}_{\text{acc}} \rightarrow \textbf{map}\ (\lambda(i, \overline{x}, \overline{as}) \rightarrow \textbf{upd}\ i\ v\ \overline{as})\ is\ \overline{xs}\ \overline{as}_{\text{acc}})$$

where we treat $\overline{as}_{\text{acc}}$ as an array of accumulators when passed to **map**, and treat the result of the **map** as a single accumulator. This is efficient because accumulators have no run-time representation, and saves us from tedious boilerplate.

During the lifetime of the accumulator, the underlying array may not be used—this prevents observation of intermediate state. These rules can be encoded in a linear type system and mechanically checked, which we do in our implementation, but exclude from the paper for simplicity.

Accumulators are sufficient to express the adjoint computation inside maps because (1) any read from a array $a[i]$ is turned into an accumulation on $\overline{a}[i]$ as discussed in Section 4.1, and (2) the only place on the return sweep where $\overline{a}$ can be read outside an accumulation statement is the definition of a, which by definition is the last use of $\overline{a}$, hence it is safe to turn it back into a normal variable at that point.

[13] For simplicity we treat only single-dimensional arrays in this section, but the idea also works in the multi-dimensional case. This type for **withacc** allows only a single result corresponding to the array being updated. In practice, we also need to be able to return an arbitrary secondary result.

6 Implementation and Optimizations

We have implemented the reverse AD transformation as a pass in a copy of the publicly available Futhark compiler.[14] The presented transformation rules were tuned to preserve fusion opportunities, both with constructs from the statement's trace, and across statements. Section 6.2 discusses several omitted issues, namely how to optimize checkpointing for the arrays that are constructed by in-place updates inside loop nests, and how to support while loops, and Section 6.3 presents the limitations of our implementation.

Since accumulators were not supported in the original language, we have implemented them throughout the compiler—for the GPU backends, they ultimately boil down to atomic updates, such as `atomicAdd` in CUDA. Accumulators however, often result in suboptimal performance, because they access memory in uncoalesced fashion and are subject to conflicts, i.e., threads simultaneously accessing the same location.

In this sense, Section 6.1 presents optimizations aimed at turning accumulators into more specialized constructs (e.g., map-reduce) that can be better optimized.

6.1 Optimizing Accumulators

We demonstrate our optimizations of accumulator on the matrix-matrix multiplication. The code below assumes $as \in \mathbb{R}^{m\times q}$ and $bs \in \mathbb{R}^{q\times n}$ and computes $cs \in \mathbb{R}^{m\times n}$ by taking the dot product of each (pair of) row of as and column of bs:

```
let xs = map (λi → map (λj → sum (map (*) as[i,:] bs[:,j])
             ) [0...n-1]) [0...m-1]
```

Differentiating the code above results in the reverse sweep:

```
let (as̄, b̄s) = withacc(as̄ b̄s λas̄ b̄s →
  map (λi as̄ b̄s → map (λj as̄ b̄s → map (λk as̄ b̄s →
        -- as̄[i,k] += bs[k,j]*c̄s[i,j]
        let as̄ = upd (i,k) (bs[k,j]*c̄s[i,j]) as̄
        -- b̄s[k,j] += as[i,k]*c̄s[i,j]
        let b̄s = upd (k,j) (as[i,k]*c̄s[i,j]) b̄s in (as̄, b̄s)
      ) [0...q-1] as̄ b̄s) [0...n-1] as̄ b̄s) [0...m-1] as̄ b̄s
```

which is not efficient, because (temporal) locality is subutilized. In this sense, we have designed and implemented a pass aimed at turning (common cases of) accumulator accesses into reductions. We present the analysis at an intuitive level: The analysis searches for the first accumulator directly nested in a perfect map-nest and checks whether its indices are invariant across any of the parallel dimensions. In our case as is accumulated on indices $[i,k]$ that are both invariant to the outermost parallel index j. In such a case, the map-nest is split into two: the code on which the accumulated statement depends on, and the code without the accumulator statement,[15] which is simplified and treated recursively. The map-nest encapsulating the accumulation is reorganized such that the invariant dimension (j) is moved innermost.[16] The accumulated statement is taken out of this innermost map, which is modified to produce (only) the accumulated values, whose sum (`reduce (+) 0`) is re-written to be the value accumulated by $\overline{as}$:

```
let (as̄) = withacc(as̄ λas̄ →
  map (λi as̄ → map (λk as̄ →
        let s = sum (map (*) bs[:,j] c̄s[i,:])
        in  upd (i,k) s as̄ -- as̄[i,k] += bs[k,j]*c̄s[i,j]
      ) [0...q-1] as̄) [0...m-1] as̄
let (b̄s) = withacc(b̄s λb̄s →
  map (λk b̄s → map (λj b̄s →
        let s = sum (map (*) as[:,k] c̄s[:,j])
        in  upd (k,j) s b̄s -- b̄s[i,k] += as[i,k]*c̄s[i,j]
      ) [0...n-1] b̄s) [0...q-1] b̄s
```

One can notice that the code now essentially consists (as expected) of two matrix-multiplication-like kernels. These are optimized by a later pass that performs block and register tiling whenever it finds an innermost map-reduce whose input arrays are invariant to one of the outer-parallel dimensions. We have extended this pass (1) to support accumulators, (2) to keep track of the array layout—i.e., transposed or not, (3) to copy from global to shared memory in coalesced fashion for any layout, and (4) to exploit some of the parallelism of the innermost dot product as inspired from [23].

6.2 Loop optimizations

As discussed in section 4, by default, loop-variant variables are saved/checkpointed at the entry of each iteration. This technique does not preserve the work asymptotic of the original program when a loop variant array is modified in place. For example, the contrived dependent loop below constructs an array of length n in $O(n)$ work, but the checkpointing of the forward sweep would require $O(n^2)$ work:

```
loop (xs) = (xs_0) for i = 1...n-1 do
  let xs[i,j] = ass[i,j] + xs[i-1]  in xs
```

One can observe however that iteration-level checkpointing is not needed if the loop nest does not exhibit *any false dependencies* (WAR+WAW):[17] since no value is "lost" through the loop nest, it is enough to checkpoint `xs` only once at the entry to the outermost loop of the nest. Moreover, re-execution is safe because all the over-writes are idempotent. In this sense, we allow the user to annotate such loop parameters that are free of false dependencies: we checkpoint them at the loop-nest entry an restore this copy just before entering the reverse sweep of that nest. Essentially, we consider that false dependencies are properly named as such, and the user should not rely on them, especially in a data-parallel context.

A second issue relates to while loops, on which we cannot perform AD directly, because their statically-unknown iteration count hinders the allocation of the checkpointing

[14] https://github.com/diku-dk/futhark

[15] The optimization fires only if the number of redundant access to global memory introduced by splitting the map nest is less than two.

[16] It is always safe to interchange parallel loops inwards.

[17] The absence of false dependencies, means that the loop has only true (RAW) dependencies or no dependencies at all.

Tool	BA	D-LSTM	GMM	HAND	
				Comp.	Simple
Futhark	13.0×	3.2×	5.1×	49.8×	45.4×
Tapenade	10.3×	4.5×	5.4×	3758.7×	59.2×
Manual	8.6×	6.2×	4.6×	4.6×	4.4×

Table 1. Time to compute the full Jacobian relative to time to compute the objective function. Lower is better.

tape. We provide two mechanisms to address this issue: first, the user may annotate a while-loop with an iteration bound n. The while loop is then transformed into an n-iteration for-loop that contains a perfectly nested **`if`**, which only executes the valid iterations of the while loop. Alternatively, in the absence of such annotation, the loop count can be computed dynamically by an inspector—i.e., loop slice that computes only the number of iterations of the loop.

6.3 Current Implementation Limitations

We have already discussed limitations during the presentation of individual constructs. The main remaining one is that we do not currently support loop-variant parameters that change their shape throughout the loop. In principle this can be handled by dynamic re-allocations, albeit this might prove expensive in a GPU setting.

Another shortcoming is that when an in-place update occurs inside an **`if-then-else`**, we currently save the target array at the branch's entry: in principle, we should instead propagate the saved element(s)—denoted xs_{saved} in Section 5.3—outside branches, and back in to reach the reverse trace.[18] By construction, this propagation cannot escape the scope of the innermost-enclosing recurrence.

7 Experimental Evaluation

7.1 AD-Bench: Sequential AD Performance

ADBench is a collection of benchmarks aimed at comparing different AD tools [28]. We have used the ADBench framework to measure Futhark versions of the four ADBench problems. We compile to sequential CPU code and report the time taken to compute the full Jacobian relative to the time taken to compute the objective function, using the largest default dataset for each benchmark. We compare against Tapenade [2] and manually differentiated programs. The results are shown in Table 1.

Futhark does well, in particular managing to outperform Tapenade in four out of five cases. For the exception, BA, the bottleneck is packing the result (which is a sparse Jacobian) in the CSR format expected by the tooling, which is code that is not subject to AD. The HAND benchmark has two variants: a "simple" one that computes only the dense part of the Jacobian, and a "complicated" one that also computes a sparse part. Tapenade handles the latter poorly. Both Tapenade and Futhark perform poorly when compared to the hand-written code. Both BA and HAND produce sparse Jacobians where the sparsity structure is known in advance, which is exploited by passing appropriate seed vectors to `jvp`/`vjp`.

[18] Of course, if the in-place update occuring inside the **`if`** is protected by the annotation discussed in Section 6.2, then they do not require any instrumentation.

7.2 Parallel Hardware and Methodology

We benchmarks on two different Linux systems: one with two AMD EPYC 7352 CPUs and an NVIDIA A100 GPU running CUDA 11.2, which we denote **A100**, and another with two Intel E5-2650 CPUs and an NVIDIA RTX 2080Ti GPU running CUDA 11.3, which we denote **2080Ti**. We report mean runtime for 10 runs, which includes all overheads, except transfering program input and result arrays between device and host. We report the absolute runtime of the differentiated and primal program, and the "overhead" of differentiation that corresponds to the ratio between the two. In optimal AD, this ratio (counted in number of operations) is supposed to be a small constant [10], hence the ratio serves as a good measure of the efficiency of an AD implementation. Futhark benchmarks using 32-bit floats were performed with block and register tile sizes of 16 and 4, respectively; on 64-bit floats the register tile size is 2.

We have developed our framework on NVIDIA Turing GPUs including the RTX 2080Ti, RTX 2070, and Quadro RTX 6000 GPUs and found consistent performance across all development GPUs. On the **A100** benchmark system, we observed that PyTorch receives a significant boost, but Futhark does not, especially for benchmarks dominated by matrix multiplication (GMM and LSTM).[19] We are investigating this puzzling behavior.

7.3 Compared to Enzyme

RSBench and XSBench are CUDA implementations of Monte Carlo neutron transport algorithms that are reported in the Enzyme paper [18], and which we have ported to Futhark in order to quantiatively compare our solution with Enzyme. They each constitute a large `map` operation that contains inner loops and control flow, as well as indirect indexing of arrays. We benchmark on the **2080Ti** system, which features a GPU similar to the one used to report the Enzyme results. We use the "small" datasets. The results (Table 2) show the overhead caused by our AD and the one reported in the Enzyme paper. Our overhead is slightly smaller, although this may come down to micro-optimisations.

7.4 Case Study 1: K-means clustering

k-means clustering is an optimization problem where given n points P in a d-dimensional space we must find k points C that minimize the cost function

[19] For example, LSTM on the **A100** system is slower than on the **2080Ti** system (both on floats).

	Primal runtimes		AD overhead	
	Original	Futhark	Futhark	Enzyme
RSBench	2.311*s*	2.118*s*	3.6×	4.2×
XSBench	0.244*s*	0.235*s*	2.6×	3.2×

Table 2. Results for RSBench and XSBench. We report the absolute runtimes of the original program and the un-differentiated Futhark program, as well as the overhead of a single forward and return sweep of the reverse-mode differentiated program compared to the un-differentiated one.

	(k, n, d)	Futhark Manual	Futhark AD	PyTorch
A100	(5, 494019, 35)	12.79*ms*	133.8*ms*	54.6*ms*
	(1024, 10000, 256)	17.5*ms*	16.9*ms*	12.0*ms*
2080Ti	(5, 494019, 35)	17.7*ms*	80.9*ms*	71.4*ms*
	(1024, 10000, 256)	18.5*ms*	18.9*ms*	22.6*ms*

Table 3. Performance measurements for k-means clustering for two different workloads, implemented both manually in Futhark, with AD in Futhark, and with AD in PyTorch.

$$f(C) = \sum_{p \in P} \min\{||p - c||, c \in C\}$$

Solving this with Newton's Method requires computing the Jacobian and Hessian of f. The cost function is easily written with nested `map` and `reduce` operations, for which the Hessian can be computed by nesting `vjp` inside of `jvp`. Note that while

$$f : \mathbb{R}^{k \times d} \to \mathbb{R}$$

and so needs reverse-mode `vjp`, the differentiated function

$$\text{vjp } f : \mathbb{R}^{k \times d} \to \mathbb{R} \to \mathbb{R}^{k \times d}$$

is suitable for use with forward-mode `jvp` (the $\mathbb{R}$ argument is set to 1). Further, as the Hessian for f has nonzero entries only along the diagonal, we need only a single invocation of `jvp` to compute it, returning exactly this diagonal. This shows how `jvp`/`vjp` allows the user to exploit sparsity. Further, the nesting of `jvp`/`vjp` allows us to avoid duplicating the expensive search operation in the cost function, which would not be the case if we were limited to dedicated constructs for Jacobian and Hessian computation.

k-means clustering can also be solved using a method based on repeatedly calculating histograms [11], which can be implemented using Futhark's dedicated histogram construct. The code that is generated by AD consists of `map`s that perform semantically equivalent histogram-like accumulator updates. Essentially, using AD leads to basically the same code one would write by hand. However, while the histogram construct uses specialized GPU code generation based on conflict-reduction via multihistogramming and caching the histograms in CUDA shared memory when possible, the accumulator updates are implemented somewhat naively with atomic updates of a single shared array in global memory. For some workloads the performance difference is significant, but translating such uses of accumulators into histogram constructs remains future work.

	n	d	K		n	d	K
$\mathbf{D}_0$	1k	64	200	$\mathbf{D}_3$	10k	64	25
$\mathbf{D}_1$	1k	128	200	$\mathbf{D}_4$	10k	128	25
$\mathbf{D}_2$	10k	32	200	$\mathbf{D}_5$	10k	128	200

(a) Parameters of the ADBench datasets used for benchmarking. The corresponding datasets may be found on the ADBench GitHub repository (https://github.com/microsoft/ADBench).

	$\mathbf{D}_0$	$\mathbf{D}_1$	$\mathbf{D}_2$	$\mathbf{D}_3$	$\mathbf{D}_4$	$\mathbf{D}_5$
PyT. Jacob. (ms)	7.8	19.4	20.3	8.1	18.7	94.3
Fut. Speedup (×)	1.20	1.06	1.11	1.20	0.91	0.58
PyT. Overhead (×)	3.0	5.16	2.55	3.18	4.21	3.23
Fut. Overhead (×)	3.3	3.2	2.9	2.8	3.0	3.1

(b) GMM benchmark results on the **A100** system with 64-bit floats. **Fut.** and **PyT.** refer to Futhark and PyTorch. **PyT. Jacob.** is the time to compute the full Jacobian of the objective function in PyTorch.

	$\mathbf{D}_0$	$\mathbf{D}_1$	$\mathbf{D}_2$	$\mathbf{D}_3$	$\mathbf{D}_4$	$\mathbf{D}_5$
PyT. Jacob. (ms)	20.2	57.9	98.8	31.1	72.5	∅
Fut. Speedup (×)	4.5	5.29	4.78	7.31	6.20	∅
PyT. Overhead (×)	2.28	2.41	3.07	3.20	2.62	∅
Fut. Overhead (×)	2.8	2.5	3.2	2.6	3.5	3.27

(c) GMM benchmark results on the **2080Ti** system with 32-bit floats. The PyTorch implementation ran out of memory on the $\mathbf{D}_5$ dataset, indicated with ∅. The Futhark Jacobian-runtime was 96.7*ms* on $\mathbf{D}_5$.

Table 4. Benchmarking results for the GMM case study.

In PyTorch we must take care when computing the all-pairs Euclidean distance from data points to the centers. Expressing this computation with broadcasting suffers from massive memory overhead. We get around this by explictly distributing the quadratic terms in the Euclidean distance. The AutoGrad module computes the Jacobian and Hessian, instead of in one go as we can when nesting `vjp`/`jvp`.

We benchmark with two qualitiatively different datasets, with results in Table 3. When the histograms benefit from the optimisations discussed in [11], the hand-written implementation has a speedup of 10.4× over the AD approach, while there is no significant difference if they cannot. Our AD approch and PyTorch are close for all dataset except the KDD on A100, where PyTorch is 2× faster. As PyTorch does twice the number of operations, the difference comes down to faster primitive operations in PyTorch.

7.5 Case Study 2: GMM

To evaluate the parallelism-preservation of our AD transformation, we compile the GMM benchmark from the ADBench

			Speedups (×)		Overheads (×)		
		PyT. Jacob.	Fut.	cuDNN	PyT.	Fut.	cuDNN
A100	$\mathbf{D_0}$	46.8*ms*	1.3	12.9	3.4	3.4	2.7
	$\mathbf{D_1}$	739.0*ms*	1.5	14.8	3.5	3.8	3.0
2080Ti	$\mathbf{D_0}$	76.8*ms*	3.1	8.3	2.6	2.6	3.2
	$\mathbf{D_1}$	1175.0*ms*	3.2	12.8	2.5	4.0	2.5

Table 5. LSTM measurements on datasets $\mathbf{D_0} : (bs, n, d, h) = (1024, 20, 300, 192)$ and $\mathbf{D_1} : (bs, n, d, h) = (1024, 300, 80, 256)$. **Py.T**, **Fut.**, and **cuDNN** refer to the PyTorch, Futhark, and the cuDNN-based `torch.nn.LSTM` implementations, respectively.

suite to parallel CUDA. We compare against ADBench's implementation of GMM in PyTorch (also run on CUDA), which we have improved (by a $\sim$ 40× factor) by vectorizing all comprehensions. We benchmark on a selection of 1,000 and 10,000-point datasets from ADBench featuring a variety of d (the dimensionality of the input data) and K (the number of Gaussian distributions used in the model) values, see Table 4a. We observed that the runtime of the primal program is dominated by matrix multiplication ($\sim$ 70%).

Matrix multiplication is a primitive in PyTorch [19]; we reasonably expect that differentiation of matrix multiplication may be readily implemented very efficiently. In Futhark there are no such primitives: matrix multiplication is written with maps, whose differentiation yield accumulators, which are further optimized as described in Section 6.1.

The benchmark results are shown in Tables 4b and 4c for the **A100** and **2080Ti** systems, respectively. On the **A100** 64-bit floats were used; on the **2080Ti** system 32-bit floats were used due to the limited double-precision performance of the system. On the **2080Ti** system, the AD overhead for Futhark and Python are quite comparable. The results demonstrates a significant speedup of Futhark over PyTorch (as high as 7.3×) in the Jacobian runtimes on all inputs. Futhark performs mostly on-par with PyTorch on the **A100** system with some notable improvements in AD overhead, except $\mathbf{D_5}$–this is due to the (mentioned) matrix multiplication bottleneck.

7.6 Case Study 3: LSTM

Long Short-Term Memory (LSTM) [26] is a type of recurrent neural network architecture popular in named entity recognition and part-of-speech tagging [7][24]. We benchmark two LSTM networks with hyperparameters common in natural language processing. The LSTM's input dimensions (d) are 80 and 300, respectively. These correspond to GloVe embeddings [22]. The recurrent units have hidden dimensions (h) 192 and 256 [7]. The sequence lengths (n) are 20 and 300. The lengths are close to the average sentence length for the Penn Treebank dataset [16] and IMDB dataset [15]. Batch sizes (bs) were chosen to fully saturate the GPUs.

Both the Futhark and PyTorch implementations are based on the architecture in [26]. We also compare against PyTorch's `torch.nn.LSTM` class, which wraps the NVIDIA cuDNN LSTM implementation [6]; note that this implementation is hand-written/optimized and features manual differentiation. All benchmarks were on 32-bit floating points and were initialized with the same parameter data, which was generated by `torch.nn.LSTM`. The results are shown in Table 5.

Futhark shows significant speedups over PyTorch on the **2080Ti** system and moderate speedups on the **A100** system. As with GMM, LSTM is dominated by matrix-multiplication computations which offers insight into the lesser relative performance on the **A100**. Neither AD-based implementation is competitive against the manual cuDNN-based implementation, but this is to be expected. Futhark and PyTorch share similar overheads to the cuDNN-based implementation; this suggests that the AD in Futhark on LSTM is close to optimal.

8 Related Work

$\widetilde{F}$ is a functional array language that supports nested parallelism, but its AD implementation uses the forward mode, along with a handful of rewrite rules that allow it to exploit sparsity in some cases [27].

Enzyme shows the advantage of of performing AD after standard compiler optimisations has simplified the program [17]. Like Enzyme, we also apply our AD transformation on a program that has already been heavily optimized by the compiler. But where Enzyme is motivated by performing AD on a post-optimisation low-level representation, our work takes advantage of *both* pre-AD optimisation, as well as the information provided by high-level parallel constructs. Enzyme has also been applied to GPU kernels [18]. We achieve equivalent performance, but our approach is not based on differentiating single kernels—indeed, the GPU code we generate for a differentiated program may have a significantly different structure than the original program. For example, the optimized adjoint code for a matrix multiplication requires *two* matrix multiplications, each its own kernel, as in the LSTM and GMM benchmarks.

ML practitioners use tools such as PyTorch [19] that incorporate AD. These are less expressive than our language and do not support true nested parallelism, but instead require the program to use flat (although vectorized) constructs. On the other hand, they can provide hand-tuned adjoints for the primitives they do support.

9 Conclusions

We have presented a fully operational compiler implementation of both reverse and forward mode AD in a nested-parallel functional language. Our experimental evaluation shows that our mapping of parallel construct and our technique of implementing the tape based on redundant computation is practically effective, and competitive with both (1)

well-established frameworks that encompass more specialized languages such as PyTorch, and with (2) newer research efforts aimed at a lower-level language, such as Enzyme.

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

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

[6] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[7] Jason PC Chiu and Eric Nichols. 2016. Named entity recognition with bidirectional LSTM-CNNs. *Transactions of the Association for Computational Linguistics* 4 (2016), 357–370.

[8] Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. 2018. Static Interpretation of Higher-order Modules in Futhark: Functional GPU Programming in the Large. *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 97 (July 2018), 30 pages.

[9] 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.

[10] Andreas Griewank and Andrea Walther. 2008. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM.

[11] 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.

[12] 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

[13] Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. 2018. High-performance defunctionalization in Futhark. In *Symposium on Trends in Functional Programming (TFP'18)*.

[14] B. Lu and J. Mellor-Crummey. 1998. Compiler optimization of implicit reductions for distributed memory multiprocessors. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. 42–51. https://doi.org/10.1109/IPPS.1998.669887

[15] Andrew Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. 2011. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*. 142–150.

[16] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. (1993).

[17] 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*.

[18] 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

[19] 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.

[20] 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

[21] 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

[22] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

[23] Ari Rasch, Richard Schulze, and Sergei Gorlatch. 2019. Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 354–369. https://doi.org/10.1109/PACT.2019.00035

[24] Ola Rønning, Daniel Hardt, and Anders Søgaard. 2018. Sluice resolution without hand-crafted features over brittle syntax trees. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. 236–241.

[25] Amr Sabry and Matthias Felleisen. 1992. Reasoning About Programs in Continuation-passing Style. *SIGPLAN Lisp Pointers* V, 1 (Jan. 1992), 288–298.

[26] Haşim Sak, Andrew Senior, and Françoise Beaufays. 2014. Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition. arXiv:1402.1128 [cs.NE]

[27] 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

[28] Filip Srajer, Zuzana Kukelova, and Andrew Fitzgibbon. 2018. A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning. *Optimization Methods & Software* 33, 4–6 (2018), 889–906. https://doi.org/10.1080/10556788.2018.1435651 arXiv:https://doi.org/10.1080/10556788.2018.1435651

[29] Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. 2019. Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator. *Proc. ACM Program. Lang.* 3, ICFP, Article 96 (July 2019), 31 pages. https://doi.org/10.1145/3341700