AD for an Array Language with Nested Parallelism

Robert Schenck
DIKU
University of Copenhagen
r@bert.lol

Ola Rønning
DIKU
University of Copenhagen
ola@di.ku.dk

Troels Henriksen
DIKU
University of Copenhagen
athas@sigkill.dk

Cosmin E. Oancea
DIKU
University of Copenhagen
cosmin.oancea@diku.dk

***Abstract*—We present a technique for applying reverse mode automatic differentiation (AD) on a non-recursive second-order functional array language that supports nested parallelism and is primarily aimed at efficient GPU execution.**

The key idea is to eliminate the need for a tape by relying on redundant execution to bring into each new scope all program variables that may be needed by the differentiated code. Efficient execution is enabled by the observation that perfectly nested scopes do not introduce re-execution and that such perfect nests can be readily produced by application of known compiler transformations. Our technique differentiates loops and bulk-parallel operators—e.g., map, reduce(-by-index), scan, and scatter—by specific rewrite rules and aggressively optimizes the resulting nested-parallel code. We report an evaluation that compares with established AD solutions and demonstrates competitive performance on ten common benchmarks from recent applied AD literature.

***Index Terms*—automatic differentiation, functional data parallel language, compilers, GPGPU**

I. 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 [1], ADOL-C [2], and Stalingrad [3]. Modern deep learning is built on array programming frameworks such as Tensorflow [4], PyTorch [5], and JAX [6] which provide implicitly parallel bulk operations that support AD.

A largely unsolved challenge is supporting AD for high-level parallel languages [7]–[9] that permit arbitrary nesting of sequential and parallel constructs. Such solutions may in principle act as a catalyst for prototyping and training of more advanced machine learning (ML) models.

ML often involves minimizing a cost function, a procedure which generally involves computing its derivative. Cost functions in ML workloads typically have far more inputs than outputs; the *reverse* mode of AD is the most efficient in such cases [10] but is challenging to implement because intermediate program values are required by the differentiated code. The program must first run a *forward sweep* (a.k.a., primal trace) that stores intermediate program states on the *tape*. The tape is read from in the *return sweep*, which executes the program in reverse to compute 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 (dynamic) closures [3] and delimited continuations [11]. These abstractions are not suited for efficient parallel execution on manycore hardware such as GPUs.

This work is, to our knowledge, the first to demonstrate an efficient GPU implementation of reverse mode AD as a compiler transformation on a data-parallel language that supports nested parallelism by means of higher-order array combinators such as map, reduce(-by-index), scan, and scatter.

A key difference is that related approaches save all variables on the tape by default and support checkpointing annotations as a memory footprint optimization. However, in a nested-parallel context, the tape may give raise to complex, irregular data-structures which are passed across deep nests and are challenging to implement efficiently in regards to optimizing spatial (coalescing) and temporal locality. Our approach exploits the fact that applying reverse AD to a straight line of side effect-free code does not require any tape because all intermediate values remain available [10]. 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 needed variables into scope.

Our technique preserves work-span asymptotics because the recomputation overhead is at worst proportional to the depth of the deepest nest of scopes, which is constant for a given non-recursive program. Moreover, perfectly nested scopes (other than loops) are guaranteed to not introduce re-execution, hence the overhead can be minimized by classic compiler transformations such as flattening nested parallelism [12] and polyhedral-like optimizations [13]. Since we forgo passing a tape across scopes, scalars are efficiently accessed from registers rather than global memory, and the code resulting from the AD transformation fully benefits from the existent compiler optimization repertoire.[1]

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 (eq. (2)) and extend it by applying reasoning that combines imperative (dependence analysis, loop distribution) and functional thinking (rewrite rules, recurrences as scans).

[1]Our approach is not "AD-efficient" because there is no constant bound for the depth (of scopes) that programs may have. We view this more as a tradeoff that needs to be demonstrated rather than a weakness because tape-based systems may incur order-of-magnitude overheads (due to inefficient utilization of locality), which are higher than the depth of most programs.

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00

(a)

$P(x_0,\ x_1):$
$v_0 = \sin(x_0)$
$v_1 = x_1 * v_0$
$v_2 = v_0 * v_1$
$y_0 = v_1 + v_2$
return y_0

(b)

$\overrightarrow{P}(x_0,\ x_1,\ \dot{x_0},\ \dot{x_1}):$
$v_0 = \sin(x_0)$
$\dot{v_0} = \cos(x_0) * \dot{x_0}$
$v_1 = x_1 * v_0$
$\dot{v_1} = v_0 * \dot{x_1} + x_1 * \dot{v_0}$
$v_2 = v_0 * v_1$
$\dot{v_2} = v_1 * \dot{v_0} + v_0 * \dot{v_1}$
$y_0 = v_1 + v_2$
$\dot{y_0} = \dot{v_1} + \dot{v_2}$
return $\dot{y_0}$

(c)

$\overleftarrow{P}(x_0,\ x_1,\ \overline{y_0}):$
$v_0 = \sin(x_0)$
$v_1 = x_1 * v_0$
$v_2 = v_0 * v_1$
$y_0 = v_1 + v_2$
$\overline{v_1} = \overline{y_0}$
$\overline{v_2} = \overline{y_0}$
$\overline{v_0} = v_1 * \overline{v_2}$
$\overline{v_1}\ += v_0 * \overline{v_2}$
$\overline{x_1} = v_0 * \overline{v_1}$
$\overline{v_0}\ += x_1 * \overline{v_1}$
$\overline{x_0} = \cos(x_0) * \overline{v_0}$
return $\overline{x_0}, \overline{x_1}$

Fig. 1. (a) Program $P(x_0, x_1) = x_1 \cdot \sin(x_0) + \sin(x_0) * x_1 * \sin(x_0)$, (b) the forward and (c) reverse mode AD transformation of P.

In particular, the simplest parallel operator, map, is the most difficult one to differentiate because its purely functional semantics allow free variables to be freely read inside it, but reverse AD replaces reads with accumulations, which are not representable as a combination of classical data-parallel constructs. We report safe support for accumulations inside maps via *accumulators*, together with optimizations aimed at transforming accumulators into more specialized constructs such as reductions, which are further optimized for locality and may yield speedups close to one order of magnitude.

Our overall contribution is an end-to-end AD algorithm that supports nested parallel combinators as well as nesting of forward and reverse mode, which is implemented as a compiler pass for the Futhark programming language. Our specific contributions are:

- A redundant execution technique for reverse AD that eliminates the need for tape and does not introduce re-execution for perfectly nested scopes other than loops.
- A set of rewrite rules for differentiating higher-order parallel combinators, including in the presence of free variables.
- A collection of optimizations that rewrite common cases of accumulators to reductions, which benefit from specialized code generation.
- An experimental evaluation that demonstrates sequential and GPU performance competitive with Tapenade [1], Enzyme [14], PyTorch [5], and JAX [6].

II. Preliminaries

In AD we seek to answer a basic question: how do changes to the inputs of a program affect its outputs? Consider variable v_2 from program P in fig. 1 (a): the *tangent* of v_2, written $\dot{v_2}$, measures how v_2 changes as the inputs x_0 and x_1 change.[2] Since $v_2 = v_0 * v_1$, x_0 and x_1 affect v_2 indirectly via v_0 and v_1. The chain rule of calculus says that we can express $\dot{v_2}$ in terms of $\dot{v_0}$ and $\dot{v_1}$, scaled by the sensitivity of v_2 to v_0 and v_2 to v_1, respectively. By computing the tangent of all intermediate variables in program order, we can compute $\dot{y_0}$—the answer to our question. The forward mode rewrite rule expresses precisely this program transformation:

$$v = f(u,w) \implies \begin{aligned} v &= f(u,w) \\ \dot{v} &= \frac{\partial f(u,w)}{\partial u}\dot{u} + \frac{\partial f(u,w)}{\partial w}\dot{w} \end{aligned} \tag{1}$$

The original statement $v = f(u, w)$ is preserved in the rule because the derivatives appearing in tangent expressions may depend on variables in the original program. Application of the rule to program P in fig. 1 (a) yields the differentiated program $\overrightarrow{P}$, shown in fig. 1 (b). For example, applying eq. (1) to the statement $v_2 = v_0 * v_1$ in P yields the lines

$$\begin{aligned} v_2 &= v_0 * v_1 \\ \dot{v_2} &= v_1 * \dot{v_0} + v_0 * \dot{v_1} \end{aligned}$$

in $\overrightarrow{P}$. The inputs of $\overrightarrow{P}$ are augmented with their respective tangents; to find the derivative with respect to the i-th input, we set $\dot{x}_i = 1$ and all other input tangents to 0. So, for n inputs to the original program, n executions of $\overrightarrow{P}$ are required to yield the derivative with respect to each input.

In contrast to forward mode AD, reverse mode AD computes how a program's outputs are affected by its inputs from the bottom up. Consider the variable v_0 in program P; the *adjoint* of v_0, written $\overline{v_0}$, measures how the output y_0 changes as v_0 changes.[3] Notice that y_0 depends on v_0 indirectly through v_1 and v_2; the chain rule says that $\overline{v_0}$ is simply the addition of $\overline{v_1}$ and $\overline{v_2}$, each scaled by the sensitivity of v_1 to v_0 and v_2 to v_0, respectively. This approach is inherently bottom-up: in reverse mode AD the adjoint of each variable is determined in reverse program order. Since adjoint variables have the reverse dependencies of the original program variables, this necessitates that P must first be executed (to bring all variables into scope as the adjoints may depend on them) before any adjoints can be computed. Variables may be used on the right-hand side of multiple statements, meaning that their adjoints must in general accumulate contributions throughout the program. All of this is captured in the reverse mode rewrite rule

$$v = f(u,w) \implies \begin{aligned} v &= f(u,w) \\ &\vdots \\ \overline{u}\ &+= \frac{\partial f(u,w)}{\partial u}\overline{v} \\ \overline{w}\ &+= \frac{\partial f(u,w)}{\partial w}\overline{v} \end{aligned} \tag{2}$$

where the $\vdots$ indicates that the statements are separated by the result of applying eq. (2) to all statements following the $v = f(u, w)$ statement in the original program. Application of the rewrite rule on P yields the reverse mode differentiated

[2] Formally, $\dot{v_i} = \sum_{j=0}^{n-1} \partial v_i / \partial x_j$ where $x_0, \ldots, x_{n-1}$ are the n inputs to the program.

[3] Formally, $\overline{v_i} = \sum_{j=0}^{m-1} \partial y_j / \partial v_i$ where $y_0, \ldots, y_{m-1}$ are the m ouputs of the program.

program $\overleftarrow{P}$, shown in fig. 1 (c).[4] As an example of the rewrite rule, applying eq. (2) to the statement $v_2 = v_0 * v_1$ in P yields the lines

$$\begin{aligned} v_2 &= v_0 * v_1 \\ &\vdots \\ \overline{v_0} &= v_1 * \overline{v_2} \\ \overline{v_1} &+= v_0 * \overline{v_2} \end{aligned}$$

in $\overleftarrow{P}$. The inputs of $\overleftarrow{P}$ are augmented with the adjoints of the outputs; to find the derivative with respect to the i-th output, we set $\overline{y}_i = 1$ and all other output adjoints to 0. A program P with m outputs requires m evaluations of $\overleftarrow{P}$ to yield all input adjoints.

A. AD Interface

We expose AD to the user via two higher-order functions, **jvp** and **vjp**, which correspond to forward and reverse mode AD, respectively.[5] The types of **jvp** and **vjp** are

$$\begin{aligned} \mathbf{jvp} &: (f : \alpha \to \beta) \to (\mathbf{x} : \alpha) \to (\mathbf{dx} : \alpha) \to \beta \\ \mathbf{vjp} &: (f : \alpha \to \beta) \to (\mathbf{x} : \alpha) \to (\mathbf{dy} : \beta) \to \alpha \end{aligned}$$

Each transforms a function $f : \alpha \to \beta$ into its derivative (if it exists) at $\mathbf{x} : \alpha$. The additional arguments $\mathbf{dx} : \alpha$ and $\mathbf{dy} : \beta$ for **jvp** and **vjp** correspond to the input tangents and output adjoints, respectively. That is, if the input $\mathbf{x}$ to f is perturbed by $\mathbf{dx}$, **jvp** reports how much the output changes, which is why it returns something of type β. On the other hand, **vjp** answers how much $\mathbf{x}$ must change to observe an output difference of $\mathbf{dy}$, returning something of type α.

Forward mode (**jvp**) is implemented in close analog to the dual number formulation described in [10] and won't be discussed further. Reverse mode is more complex due to computing adjoints in reverse program order, storing of intermediate program variables, and the accumulation of derivatives into adjoint variables, which turn any read of a variable in the original program into an accumulation in the transformed program.

B. Source Language

We perform our transformation on a data-parallel language which expresses all available parallelism by arbitrary nesting of *second-order array combinators* (SOACs), e.g., map, reduce, and scan.

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

$$\mathbf{map}\ f\ [a_0, a_1, \ldots, a_{n-1}] \ \equiv\ [f\ a_0, f\ a_1, \ldots, f\ a_{n-1}]$$

[4]The first contribution to each adjoint in $\overleftarrow{P}$ is written with $=$ instead of $+=$ for brevity; instead, adjoints could first be initialized to 0 and all accumulations could be written with $+=$.

[5]**jvp** stands for "Jacobian-vector product"; **jvp** f **x dx** computes $\mathbf{J}(f(\mathbf{x})) \cdot \mathbf{dx}$, where $\mathbf{J}(f(\mathbf{x}))$ is the Jacobian of f at $\mathbf{x}$. **vjp** stands for "vector-Jacobian product"; **vjp** f **x dy** computes $\mathbf{dy}^\intercal \cdot \mathbf{J}(f(\mathbf{x}))$.

Note that function application is curried and is denoted by spaces, e.g., the imperative $f(x, y, z)$ is written as $f\ x\ y\ z$. Map is analogous to the imperative parallel loop

$$\begin{aligned} &\mathbf{forall}\ i = 0 \ldots n - 1 \\ &\quad as[i] = f(as[i]) \end{aligned}$$

where $as = [a_0, a_1, \ldots, a_{n-1}]$. Further SOACs will be introduced as necessary. For simplicity, we allow SOACS to be called with k-ary functions wherein the SOAC is applied to k equal-length arrays. For example,

$$\mathbf{map}\ g\ as\ bs \equiv\ [g\ a_0\ b_0, g\ a_1\ b_1, \ldots, g\ a_{n-1}\ b_{n-1}]$$

where $as = [a_0, a_1, \ldots, a_{n-1}]$ and $bs = [b_0, b_1, \ldots, b_{n-1}]$.

The source language supports higher-order functions, polymorphism, modules, and similar high-level features, which are compiled away using a variety of techniques [15], [16] before we perform AD. The only remaining second-order functions are the SOACs. Lambdas (i.e., unnamed functions) can only appear syntactically in SOACs and **vjp**/**jvp**, and are not values. As such we do not suffer from "perturbation confusion" [17]. Further, a significant battery of standard optimizations (CSE, constant folding, aggressive inlining) is also applied prior to AD.

The language is written in A-normal form [18] (ANF): all subexpressions are variable names or constants with the exception of the *body* expression of **loops**, **if-then-else**-expressions and **let**-expressions. **let**-expressions consist of a series of bindings—which we also call statements—followed by a sequence of one or more returns which follow the **in** keyword (i.e., **in** separates the bindings from the returns), for example,

$$\mathbf{let}\ a = 5\ \mathbf{in}\ (\mathbf{let}\ b = a * a\ \mathbf{in}\ b)$$

evaluates to 25. The language is purely functional: redefinitions of the same variable should be understood as a notational convenience for variable shadowing. **let** x += y is syntactic sugar for **let** x = x + y, where x is shadowed. The language supports a functional flavor of in-place updates based on uniqueness types [7]. We write **let** $xs[i] = x$ as syntactic sugar for **let** $xs' = xs$ **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 loops, which have the semantics of a tail-recursive function:

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

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. The loop terminates after n iterations and the final result of e is bound to y.

```
def cost points centers =
  sum (map (λp → min (map (dist2 p) centers))
           points)

def kmeans (k: i64) (n: i64) (d: i64) (b: i64)
           (points: [n][d]f32) =
  loop (centers: [k][d]f32 = ...) for i < b do
    let (cost', cost'') =
      jvp2 (λx → vjp (cost points) x 1) centers
           (replicate k (replicate d 1))
    let dx = map (map (/)) cost' cost''
    in map (map (-)) centers dx
```

Fig. 2. Applying reverse (**vjp**) and forward (**jvp2**) AD to solve k-means, as described in section VI-D. **jvp2** $f\ x\ dx$ returns both the primal value and the derivative (**jvp** only returns the derivative); this is useful because Newton's method requires both the gradient (`cost'`) and the diagonal of the Hessian (`cost''`). `dist2` computes the Euclidean distance.

C. Example: using *vjp* and *jvp* to solve k-means.

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

$$f(C) = \sum_{p \in P} \min \left\{ ||p - c||^2, c \in C \right\} \tag{3}$$

f can be minimized using Newton's method [19], which iteratively finds the minimizing cluster locations via the recurrence

$$C_{i+1} = C_i - \mathbf{H}(f(C_i))^{-1} \nabla f(C_i)$$

until convergence, where $\mathbf{H}(f(C_i))$ is the Hessian of f and $\nabla f(C_i)$ is the gradient of f. Since the centroids are independent from each other, $\mathbf{H}(f(C_i))$ is a diagonal matrix and the above computation can instead be written as

$$C_{i+1} = C_i - \nabla f(C_i) \oslash \text{diag}(\mathbf{H}(f(C_i))$$

where $\oslash$ is element-wise division and $\text{diag}(\mathbf{H}(f(C_i))$ is the vector containing the diagonal elements of $\mathbf{H}(f(C_i))$. This avoids computing the full Hessian as well as its inverse—an expensive operation! Figure 2 shows Futhark code with a function `cost` which implements the cost function (eq. (3)) along with a function `kmeans` which minimizes it and exploits sparsity of the Hessian via the recurrence above (realized by a **loop**); this shows how the **jvp/vjp** interface allows the programmer to exploit sparsity. Note the nesting of **vjp** inside of **jvp2**, which allows the gradient to be differentiated to produce the Hessian.[6]

III. Reverse Mode AD by Redundant Execution

This section discusses the manner in which our transformation bridges scopes without requiring an explicit tape: section III-A gives an example and sketches the overall structure of the analysis, section III-B demonstrates the analysis of loops, and section III-C discusses the trade-offs related to redundant execution.

[6] $\nabla f(C_i)$ can be directly computed by **vjp**; since $\mathbf{H}(f(C_i)) = \mathbf{J}(\nabla f(C_i))$, $\text{diag}(\mathbf{H}(f(C_i))) = \mathbf{H}(f(C_i)) \cdot \mathbf{1}$ is just a Jacobian-vector product and may be directly computed with a single iteration of **jvp** on $\nabla f(C_i)$ (computed by **vjp**), where $\mathbf{1}$ is a vector of all 1s.

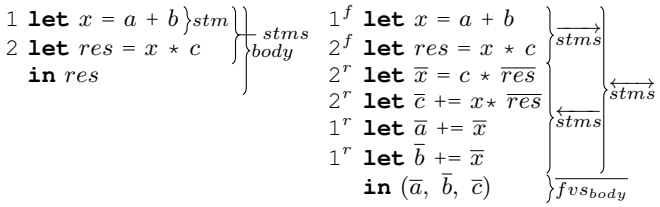

Fig. 3. An example of applying the vjp_{body} rule (see fig. 4) to a code body: the code on the left shows the original body and the code on the right shows the resulting differentiated body. The line numbers 1^f and 2^f indicate that the line corresponds to the forward sweep of lines 1 and 2 (of the original program), respectively. 1^r and 2^r indicate that the line is part of the return sweep corresponding to lines 1 and 2, respectively.

Rule vjp_{body} **refers to a body** $body = stms$ **in** res**:**
$vjp_{body}(\overline{res},\ stms\ \textbf{in}\ res) \Rightarrow \overleftrightarrow{stms}\ \textbf{in}\ (res,\ \overline{fvs_{body}})$
where $\overleftrightarrow{stms} \leftarrow vjp_{stms}(stms)$ **and** $\overline{fvs_{body}} \leftarrow \overline{\mathcal{FV}(body)}$

Rule vjp_{stms} **simply folds over each statement:**
$vjp_{stms}(stm, stms) \Rightarrow (\overrightarrow{stms},\ vjp_{stms}(stms),\ \overleftarrow{stms})$
where $(\overrightarrow{stms},\ \overleftarrow{stms}) \leftarrow vjp_{stm}(stm)$

Rule vjp_{stm} **for a scalar multiplication statement:**
$vjp_{stm}(\textbf{let}\ x = a \star b) \Rightarrow (\overrightarrow{stms},\ \overleftarrow{stms})$
where $\overrightarrow{stms} \leftarrow \textbf{let}\ x = a \star b$
$\overleftarrow{stms} \leftarrow \textbf{let}\ \overline{a}\ \mathrel{+}= b \star \overline{x},\ \textbf{let}\ \overline{b}\ \mathrel{+}= a \star \overline{x}$

Rule vjp_{stm} **for an array indexing statement:**
$vjp_{stm}(\textbf{let}\ y = a[i]) \Rightarrow (\textbf{let}\ y = a[i],\ \textbf{let}\ \overline{y} = \textbf{upd}\ i\ \overline{y}\ \overline{a})$

Rule vjp_λ **refers to a lambda function** $\lambda x_1 \ldots x_n \rightarrow body$**:**
$vjp_\lambda(\overline{res},\ \lambda x_1 \ldots x_n \rightarrow stms\ \textbf{in}\ res) \Rightarrow \lambda x_1 \ldots x_n \rightarrow \overleftrightarrow{body}$
where $\overleftrightarrow{stms}\ \textbf{in}\ (\_,\ \overline{fvs_{body}}) \leftarrow vjp_{body}(\overline{res},\ body)$
$\overleftrightarrow{body} \leftarrow \overleftrightarrow{stms}\ \textbf{in}\ \overline{fvs_{body}}$

Fig. 4. The vjp code transformation for several syntactic categories. $\overline{x}$ denotes the adjoint of x, $\mathcal{FV}(body)$ denotes the free variables of $body$, $\overrightarrow{stms}$ and $\overleftarrow{stms}$ denote the forward and return sweeps generated for $stms$.

A. Transformation Rules Across Scopes

Figure 3 illustrates the reverse AD transformation on simple example. The transformation acts on a $body$, which consist of a list of statements $(stms)$ followed by a result $(\textbf{in}\ res)$, as depicted in the code on the left of fig. 3. The right side of the figure shows the result of the transformation: it consists of a forward sweep, $\overrightarrow{stms}$, which is a re-execution of the statements from the original program to bring into scope any variables which may be needed in the return sweep, and the reverse sweep itself, $\overleftarrow{stms}$, which computes the new adjoint contributions to each variable, in reverse program order. The forward and return sweeps together constitute the statements of the differentiated body, which we label $\overleftrightarrow{stms}$. Finally, the result of the differentiated body consists of the adjoints of the free variables in the body, $\overline{fvs_{body}}$, which are $\overline{a}$, $\overline{b}$, and $\overline{c}$. Only

these adjoints can contribute to the adjoints of other program variables: all bound variables within the body will be out of scope once the body returns. In the following sections, the reverse mode transformation vjp is implemented as a syntax-directed translation [20], where translation rules are defined for each syntactic category of the language. Figure 4 shows the primary translation rules for the vjp transformation and illustrates the treatment of new scopes.

We elide symbol table bookkeeping and assume that the adjoint $\overline{x}$ of a variable x is always available.[7] The vjp_{body} rule refers to a body of statements, which *always starts a new scope*. The rule first extends the environment by binding the body result res to its adjoint $\overline{res}$ (not shown).[8] The statements of the transformed body ($\overleftarrow{stms}$) are those generated by vjp_{stms}; 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 ($\overrightarrow{stms}$), that brings into scope whatever information is necessary to execute the return sweep for that statement ($\overleftarrow{stms}$), which is always organized in the reverse order of the original statements.

The return sweep of an array-indexing statement **let** $y = a[i]$ must update $\overline{a}[i]$ with the contribution of $\overline{y}$, which is accomplished in the rule by **let** $\overline{y}$ = **upd** i $\overline{y}$ $\overline{a}$. *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 are never accessed again, similar to the in-place updates of section II-B.

Finally, rule vjp_λ transforms an anonymous function; its result is obtained by calling vjp_{body} on the function body. Note that $x_1, \ldots, x_n$ are free variables in $body$, so their adjoints are among $\overline{fvs_{body}}$.

B. Demonstrating the Transformation of Loops

Figure 5 (b) shows the result of vjp_{stms} applied to the loop in fig. 5 (a). The forward sweep of the loop (lines $2-9$) consists of the original loop except that its result and body are modified to checkpoint into array ys the value of y at the start of each iteration. ys is also declared as loop variant and initialized to ys_0, which is allocated (by **scratch**) just before the loop statement. Importantly, *only the loops of the current scope are checkpointed*; an inner-nested loop would be re-executed, not checkpointed.

The return sweep of the loop (lines $12-19$) consists of a loop that iterates backward from $m^k - 1$ down to 0: the first statement (line 15) re-installs the value of y of the current iteration from the checkpoint, so that the forward sweep of the loop body $\overrightarrow{stms_{loop}}$ can be re-executed to bring into scope the variables needed by the return sweep of the loop body

[7] In practice, either the adjoint is available or it hasn't had any contributions yet and hence a statement can be inserted which intializes the adjoint to a zero element of the appropriate type and shape.

[8] This is safe since the transformation works backwards, hence the adjoint $\overline{res}$ is already available from the outer scope.

(a) Original loop

$stms_{before}$
let $y'' =$
 loop $y = y_0$
 for $i = 0 \ldots m^k - 1$ **do**
  $stms_{loop}$
 in y'
$stms_{after}$

(c) Strip-mined loop

$stms_{before}$
let $y'' =$
 loop $y_1 = y_0$
  for $i_1 = 0 \ldots m - 1$ **do**
   ...
   loop $y_k = y_{k-1}$
    for $i_k = 0 \ldots m - 1$ **do**
     let $y = y_k$
     let $i =$
      $i_1 * m^{k-1} + \ldots + i_k$
     $stms_{loop}$
     in y'
$stms_{after}$

(b) Reverse AD of loop

```
1  →stms_before
2  let ys_0 = scratch(m^k,
3                     sizeOf(y_0))
4  let (y'', ys) =
5   loop (y, ys) = (y_0, ys_0)
6    for i = 0...m^k - 1 do
7     let ys[i] = y
8     stms_loop
9     in (y', ys)
10 →stms_after
11 ←stms_after
12 let (y''', fvs_l) =
13  loop (ȳ, fvs_l) = (y''', fvs_l0)
14   for i = m^k - 1...0 do
15    let y = ys[i]
16    →stms_loop
17    ←stms_loop
18    in (y', fvs'_l)
19 let ȳ_0 += y'''
20 ←stms_before
```

Fig. 5. (a) A loop, (b) the result of applying reverse AD to it, and (c) the result of strip-mining it into a depth-k loop nest. In (b), $vjp_{body}(\overline{y},\ stms_{loop}$ **in** $y')$ is called to generate $\overrightarrow{stms_{loop}}$, $\overleftarrow{stms_{loop}}$, and $\overline{fvs'_l}$, as defined in fig. 4.

$\overleftarrow{stms_{loop}}$. The result of the reversed loop body (line 18) is the adjoint of the original result, $\overline{y'}$, together with the adjoints of the free variables used in the loop, $\overline{fvs'_l}$. These are declared as variant through the loop (line 13) such that the updates of all iterations are recorded.[9] Finally, the statement at line 19 semantically updates the adjoint of the loop initializer $\overline{y_0}$ with the result of the loop $\overline{y'''}$. This is because the first executed instruction of the source loop sets $y = y_0$ and the adjoint of y corresponding to the first iteration is the same as the adjoint of the loop result since the return sweep executes backwards.

With our strategy, the original loop is executed twice (lines 8 and 16). Strip-mining a loop of count m^k into a depth-k loop nest, as shown in fig. 5 (c), would suffer at worst a re-execution overhead factor of $k+1$ when differentiated; however, it would have a much smaller memory overhead than the original loop as each of the k loops of count m checkpoints its own variable, resulting in a memory footprint that is proportional to mk rather than m^k.[10] When m is constant, this results in the logarithmic space and time overhead case of the time-space tradeoff studied by Siskind and Pearlmutter [21]. We exploit the tradeoff in a simple and practical way by allowing the user to annotate the loops with a constant strip-mining depth, which is applied automatically before AD.

We conclude by noting that sequential loops are the only construct that require iteration checkpointing and that, importantly, *parallel constructs do not* because the input of parallel loops do not depend on the result of other iterations.

[9] $\overline{fvs_{l_0}}$ is the adjoint of the free variables prior to entering the loop.

[10] The checkpoint of each of the k loops stores m versions of y.

```
let xss = map (λc as → if c then ... else map (λa → a*a) as) cs ass
let ass = map (λc as xs →
                let xs = if c then ... else map (λa → a*a) as
                in if c then ...
                    else let xs' = map (λa → a*a) as
                         let as = map (λa x → let x = a * a
                                              in 2 * a * x
                                      ) as xs
                         in as
              ) ass xss
```

Fig. 6. The body of the function generated by applying reverse AD to $\lambda ass \to \mathbf{map}(\lambda c\ as \to \mathbf{if}\ c\ \mathbf{then} \ldots \mathbf{else}\ \mathbf{map}(\lambda a \to a{*}a)\ as)\ cs\ ass$. Red denotes the re-execution of the forward trace in each (new) scope. Note that all re-executions are dead code; this is guaranteed when the original code consists of perfectly nested scopes.

C. Perfect Nests Do Not Incur Redundant Execution

Figure 6 shows the code generated by applying vjp to a function whose body consist of perfectly nested scopes. While we have not yet discussed how all constituents are differentiated (map is discussed in section IV-E), this is not important for the moment. What is important is to notice that the re-execution of the forward trace for each of the four scopes[11]—denoted in red in the figure—is dead code.

The rationale is that perfectly nested scopes (other than loops) are guaranteed to not introduce recomputation because, by definition, their bodies consist of one (composed) statement. Hence their result cannot possibly be used in the return sweep[12] and it does not have to be returned because it has been recomputed in the parent scope.

It follows that overheads can be optimized by known transformations that create perfect nests [13], [22] rooted in loop distribution and interchange, which are also supported by the Futhark compiler [12]. With their help, we commonly expect the forward sweep to be executed twice:

1) Once for the outermost scope because programs typically consist of multiple nests and also the user may require the result of the original program.
2) Once for the innermost scope that typically performs scalar computation, which is cheap to recompute. In comparison, vectorization or the use of tape would require such scalars to be retrieved from global memory, which has (an) order(s) of magnitude higher latency.

Note that loops whose variant values are addition-based accumulations also do not introduce recomputation, since the differentiation of plus does not require primal values.

IV. Rewrite Rules for Parallel Constructs

This section presents in detail the differentiation rules for reduce, reduce-by-index, and map. Additionally, an overview for the rules for scan and scatter is given.

[11]The outermost scope (scope 0) is the function's body, which consists of an outer map whose function's body (scope 1) consists of an **if**, whose **else**-body (scope 2) consists of an inner map, whose function body (scope 3) consists of a multiplication.

[12]Nested loops also need to be flattened into a single loop, because, even if perfectly nested, they may be kept alive due to checkpointing needs.

A. Reduce

A reduce combines all elements of an array with a binary associative operator $\odot$:

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

where $e_\odot$ is the neutral element of $\odot$. For each a_i, we can group the terms of the reduce as

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

and then directly apply the main rule for reverse AD given in eq. (2), 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 constants and can be computed by exclusive scans,[13] $\overline{+}$ denotes a potentially vectorized addition that matches the datatype, and y is the variable bound to the result of **reduce**. The code for the right-hand side can be generically generated as a function f that is *mapped* over each a_i, l_i, r_i:

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

except f is modified to not return the adjoints of l_i and r_i since they aren't needed.

In the general case, the forward sweep is the original reduce statement. Letting $as = [a_0, a_1, \ldots, a_{n-1}]$ 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
```

where $\triangleright$ is the *pipe forward operator* which enables composing functions left-to-right, e.g., $x \triangleright f \triangleright g = g\ (f\ x)$.

This translation is AD-efficient, but requires 8 memory accesses per element in comparison to one in the original reduce. Fortunately, standard operators admit more efficient translations. Specialized rules for addition, min, and max are known [23].[14]

When the reduction operator is ***multiplication***, we have

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

and discriminate three cases:

1) If all as's elements are nonzeros, then $l_i * r_i = {}^{y}/_{a_i}$ and $y \neq 0$, hence we update each element as $\overline{a_i}\ {+}{=}\ {}^{y}/_{a_i}\ \overline{y}$.
2) If exactly one element at index i_0 is zero, then $l_i * r_i$ is zero for all other elements and $\overline{a_{i_0}}\ {+}{=}\ y * \overline{y}$.
3) Otherwise, $\forall i,\ l_i * r_i = 0$ and $\overline{as}$ remains unchanged.

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

[14]The return sweep for (vectorized) addition adds $\overline{y}$ to each element of $\overline{as}$. For min/max, the forward sweep computes the minimal/maximal element together with its first occurring index i_y, i.e., **let** $(y,\ i_y)$ = argmin as, and the return sweep updates only the adjoint of that element: **let** $\overline{as}[i_y]\ \overline{+}{=}\ \overline{y}$.

The forward sweep is a reduce applied to an extended operator that computes 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.

B. Reduce-by-index

Reduce-by-index generalizes a histogram computation [24] by allowing the values from an array (as) that fall into the same bin (i.e., an index from $inds$) to be reduced with an arbitrary associative and commutative operator $\odot$ with neutral element $e_\odot$, where the number of bins m is typically smaller than the number of index-value pairs n:

$$\texttt{let}\ hs = \texttt{reduce\_by\_index}\ \odot\ e_\odot\ inds\ as$$

which has the semantics

$$\begin{array}{l}\texttt{loop}\ hs = \texttt{replicate}\ m\ e_\odot\ \texttt{for}\ i\ =\ 0 \ldots n-1\ \texttt{do}\\ \quad \texttt{let}\ hs[inds[i]]\ \odot = as[i]\ \texttt{in}\ hs\end{array}$$

Similar reasoning to that used for reduce suggests that two scans need to be applied to each subset of elements that fall in the same bin in order to compute the l_i and r_i terms for each i. The contributions to the adjoint $\overline{as}$ are then computed—as in the reduce case—by $\texttt{map}\ f\ ls\ as\ rs$. Assuming a constant key size, the scans can be implemented with the right asymptotic complexity by radix sorting as according to the corresponding bins (to ensure elements falling in the same bin are consecutive) and then by applying irregular segmented scans (forward and reverse) on the result. Since sorting is expensive in practice, we support specialized implementations for common operators:[15] the forward sweep consists of the **reduce_by_index** statement, but enhanced with extended operators, and the return sweep is similar to reduce, except that in the update formula of $\overline{as}[i]$, one replaces $\overline{y}$ with $\overline{hs}[inds[i]]$.

C. Scan

The rule for scan can be derived by differentiating its imperative, loop-based formulation and then applying loop distribution to separate a map from a recurrence of the form

$$\overline{x}_{n-1} = \overline{a}_{n-1},\ \overline{x}_i = \overline{a}_i + b_i \cdot \overline{x}_{i+1}, \qquad i = n-2 \ldots 0$$

which is known to accept a parallel formulation rooted in a scan whose operator is a generalization of linear function composition [25]. Here, b_i is a $d \times d$ matrix and a_i and x_i are vectors of length d. If d is a constant, e.g., the array elements are tuples of scalars, the work-depth asymptotic is preserved but the translation is still not AD-efficient. We however support vectorized operators for scan efficiently by interchanging the scan inside the map:

$$\begin{array}{l}\texttt{scan}\ (\texttt{map}\ \odot)\ \overline{e_\odot}\ xs\ \Rightarrow\\ \quad \texttt{transpose}\ xs\ \triangleright\ \texttt{map}\ (\texttt{scan}\ \odot\ e_\odot)\ \triangleright\ \texttt{transpose}\end{array}$$

and generate specialized code for the vectorized-plus operator.

[15] Addition, multiplication, min, and max.

D. Parallel Scatter

$\texttt{let}\ ys = \texttt{scatter}\ xs\ is\ vs$ produces an array ys by updating in-place the array xs at the m indices in is[16] with corresponding values of vs. The forward sweep saves the elements of xs that are about to be overwritten prior to performing the update:

$$\begin{array}{l}\texttt{let}\ xs_{saved} = \texttt{gather}\ xs\ is\\ \texttt{let}\ ys = \texttt{scatter}\ xs\ is\ vs\end{array}$$

where $\texttt{gather}\ xs\ is$ has the same semantics as $\texttt{map}\ (\lambda i \to xs[i])\ is$. The return sweep (1) first updates the adjoint of vs with the elements gathered from $\overline{ys}$, then (2) creates the adjoint of xs by zeroing out the elements from $\overline{ys}$ that were subject to the update, and (3) restores xs to its state before the update (from xs_{saved}):

$$\begin{array}{l}\texttt{let}\ \overline{vs}\ \overline{+}\!= \texttt{gather}\ \overline{ys}\ is\\ \texttt{let}\ \overline{xs} = \texttt{scatter}\ \overline{ys}\ is\ (\texttt{replicate}\ m\ 0)\\ \texttt{let}\ xs = \texttt{scatter}\ ys\ is\ xs_{saved}\end{array}$$

Both sweeps have $O(m)$ work and $O(1)$ depth.

E. Map

Consider the map

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

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

$$\begin{array}{l}\texttt{let}\ \overline{as} = \texttt{map}\ (\lambda(a,\ \overline{a}_0,\ \overline{x}) \to\\ \qquad\qquad \overrightarrow{stms}\ \overleftarrow{stms}\ \texttt{in}\ \overline{a}_0\ +\ \overline{a})\ as\ \overline{as}\ \overline{xs}\end{array}$$

where $\overline{a}$ is the new contribution to the adjoint of each a, computed in $\overleftarrow{stms}$.

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

In an impure language, adjoint updates for free variables can be implemented as a generalized reduction [26],[17] wherein the adjoint of a free array variable $as[i]$ could be updated with an operation $\overline{as}[i]\ +\!=\ v$, implemented with atomics or locks in the parallel case. In our pure setting, we instead introduce *accumulators* that preserve purity and guarantee the generalized reduction properties at the type level. An array can be temporarily turned into an accumulator with **withacc**:[18]

$$\texttt{withacc}\ :\ [d]\alpha\ \to (\texttt{acc}(\alpha) \to \texttt{acc}(\alpha)) \to [d]\alpha$$

Intuitively we view accumulators as a write-only view of an array. *Semantically*, accumulators are lists of index/value pairs,

[16] No duplicates are allowed in is.

[17] A loop is a generalized reduction if all its cross-iteration dependencies are due to variables X that only appear in "reduction" statements of the form $X[is[i]]\ \odot\!= exp$, where exp does not contain X, and $\odot$ is associative and commutative.

[18] 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.

each denoting an update of an array. When we use **upd** on an accumulator, we add an index/value pair to this list, returning a new accumulator. *Operationally*, **upd** immediately writes to the underlying array and does not actually maintain a list of updates in memory. The purpose of accumulators is to allow the compiler to reason purely functionally, in particular that all data dependencies are explicit, while allowing efficient code generation. Accumulators are similar to the accumulation effects in Dex [8] and have the same motivation. The main difference is that Dex 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.,

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

results in the return sweep code

$$\begin{aligned}&\textbf{let}\ \overline{as} = \textbf{withacc}\ \overline{as}\ (\lambda \overline{as}_{\text{acc}} \to \\ &\quad \textbf{map}\ (\lambda(i,\ \overline{x},\ \overline{as}) \to \textbf{upd}\ i\ \overline{x}\ \overline{as})\ is\ \overline{xs}\ \overline{as}_{\text{acc}})\end{aligned}$$

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 runtime representation and it saves us from tedious boilerplate. The equivalent imperative (generalized reduction) code is

$$\begin{aligned}&\textbf{forall}\ k = 0 \ldots \texttt{length}(is) - 1 \\ &\quad \overline{as}[is[k]]\ \mathrel{+}= \overline{xs}[k]\end{aligned}$$

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 an array $a[i]$ is turned into an accumulation on $\overline{a}[i]$ 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 an array there.

V. Implementation and Optimizations

We have implemented the reverse AD transformation as a pass in the publicly available Futhark compiler.[19] The presented transformation rules were tuned to preserve fusion opportunities, both with constructs from a statement's sweep and across statements.

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 an uncoalesced fashion and are subject to conflicts, i.e., threads simultaneously accessing the same location. In this regard, section V-A presents optimizations aimed at turning accumulators into more specialized constructs (e.g., map-reduce) that can be better optimized. Section V-B discusses several omitted issues, namely how to optimize checkpointing for arrays that are constructed by in-place updates inside loop nests and how to support while loops.

A. Optimizing Accumulators

We demonstrate our accumulator optimizations on matrix-matrix multiplication. To aid readability, we use an imperative notation in which we omit **withacc**: **forall** loops denote **map** operations and $as[i,k]\ \mathrel{+}=\ v$ denotes **let** $\overline{as}$ = **upd** $(i,\ k)\ v\ \overline{as}$.

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 row of as and column of bs.

$$\begin{aligned}&\textbf{forall}\ i = 0 \ldots m - 1 \\ &\quad \textbf{forall}\ j = 0 \ldots n - 1 \\ &\qquad cs[i,j] = \textbf{sum}\ (\textbf{map}\ (\star)\ as[i,:]\ bs[:,j])\end{aligned}$$

Differentiating the code above results in the return sweep

$$\begin{aligned}&\textbf{forall}\ i = 0 \ldots m - 1 \\ &\quad \textbf{forall}\ j = 0 \ldots n - 1 \\ &\qquad \textbf{forall}\ k = 0 \ldots q - 1 \\ &\qquad\quad \overline{as}[i,k]\ \mathrel{+}= bs[k,j] \star \overline{cs}[i,j] \\ &\qquad\quad \overline{bs}[k,j]\ \mathrel{+}= as[i,k] \star \overline{cs}[i,j]\end{aligned}$$

which is not efficient, because (temporal) locality is sub-utilized. To address this, we have designed and implemented a pass aimed at turning common cases of accumulator accesses into reductions. 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 the example above, $\overline{as}$ is accumulated on indices $[i,k]$ that are both invariant to the parallel index j. In such a case, the map nest is split into two: the code on which the accumulated statement depends and the code without the accumulator statement,[20] which is simplified and treated recursively. The map nest encapsulating the accumulation is reorganized such that the invariant dimension (j) is moved innermost.[21] The accumulation statement is taken out of this innermost map, which is modified to produce (only) the accumulated values, whose sum is rewritten to be the value accumulated by $\overline{as}$:

$$\begin{aligned}&\textbf{forall}\ i = 0 \ldots m - 1 \\ &\quad \textbf{forall}\ k = 0 \ldots q - 1 \\ &\qquad s_a\ =\ \textbf{sum}\ (\textbf{map}\ (\star)\ bs[k,:]\ \overline{cs}[i,:]) \\ &\qquad \overline{as}[i,k]\ \mathrel{+}= s_a \\ &\textbf{forall}\ k = 0 \ldots q - 1 \\ &\quad \textbf{forall}\ j = 0 \ldots n - 1 \\ &\qquad s_b\ =\ \textbf{sum}\ (\textbf{map}\ (\star)\ as[:,k]\ \overline{cs}[:,j]) \\ &\qquad \overline{bs}[k,j]\ \mathrel{+}= s_b\end{aligned}$$

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

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

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

The code now consists of two matrix multiplication-like kernels (with different parallel **forall** dimensions than the original). 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, inspired from [27].

This optimization is responsible for nearly a one-order-of-magnitude speedup at the application level for benchmarks dominated by matrix multiplication.[22]

B. Loop Optimizations and Limitations

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

$$\textbf{loop}\ xs = xs_0\ \textbf{for}\ i\ =\ 1 \ldots n-1\ \textbf{do}$$
$$\quad \textbf{let}\ xs[i,j] = as[i,j] + xs[i-1,j]\ \textbf{in}\ xs$$

Iteration-level checkpointing is not needed if the loop nest does not have any false dependencies (WAR+WAW):[23] since no value is "lost" through the loop nest, it is sufficient to checkpoint xs only once at the entry to the outermost loop of the nest. Moreover, re-execution is safe because all the overwrites are idempotent. We allow the user to annotate loop parameters that are free of false dependencies: they're checkpointed upon entry to the loop nest and restored just before entering the return sweep of the nest. Techniques in automatic parallelization can be used to automatically check the safety of such annotations, statically [28], dynamically [29], and anywhere in between [30].

A second issue relates to while loops, on which we cannot perform AD directly because their statically unknown iteration count hinders the allocation of checkpointing arrays. To address this issue, 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-then-else expression, which only executes the valid iterations of the while loop. In the absence of such annotation, the loop count is computed dynamically by an inspector.

Finally, a limitation of the current implementation is that it does not support loop-variant parameters that change their shape throughout the loop. In principle this can be handled by dynamic re-allocations, but this might be expensive on GPUs.

VI. Experimental Evaluation

A. Parallel Hardware and Methodology

We benchmark on three different Linux systems, detailed in fig. 7. We report mean runtime for 10 runs (following an initial run that is discarded), which includes all overheads, except transferring 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 [31], hence the ratio serves as a good measure of the efficiency of an AD implementation. We also report the memory usage of the primal (when applicable) and differentiated program on the dataset with maximal memory consumption for each experiment.

System	CPU	GPU	API
A100	2×AMD EPYC 7352	NVIDIA A100	CUDA 11.6
MI100	2×AMD EPYC 7352	AMD MI100	ROCm 5.0.1
2080Ti	2×Intel E5-2650	NVIDIA 2080Ti	CUDA 11.3

Fig. 7. Systems used for benchmarking.

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

Fig. 8. ADBench sequential overheads; lower is better.

B. ADBench: Sequential AD Overhead

ADBench is a collection of benchmarks for comparing different AD tools [32], to which we have added Futhark implementations. We compile to sequential CPU code on the A100 system and report the the AD overhead using the largest default dataset for each benchmark. We compare against Tapenade [1] and manually differentiated programs. The results are shown in fig. 8.

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. On HAND, both Tapenade and Futhark perform poorly compared to manually differentiated 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**.

C. Comparison with Enzyme

Enzyme is an LLVM compiler plugin that performs reverse mode AD, including support for GPU kernels [14]. We have ported several benchmarks in order to compare our solution with Enzyme, with results in fig. 9. The Enzyme overheads are copied directly from [14]. RSBench and XSBench each constitute a large parallel loop that contains inner sequential loops and control flow, as well as indirect indexing of arrays. Our overhead is slightly smaller, although this may come

[22] Such as GMM and LSTM, which are evaluated in sections VI-F and VI-G.

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

Benchmark	Primal runtimes (s)		AD overhead	
	Original	Futhark	Futhark	Enzyme
RSBench	2.311	2.127	3.9	4.2
XSBench	0.244	0.239	2.7	3.2
LBM	0.071	0.042	3.4	6.3

Fig. 9. Enzyme results, showing absolute runtimes and AD overheads. The Enzyme AD overheads are taken from [14]. RSBench and XSBench were measured on the 2080Ti, while LBM is measured on the A100, similar to the systems used in the Enzyme paper. For LBM the workload is $120 \times 120 \times 150$ for 100 iterations. RSBench and XSBench use the "small" datasets with $10,200,000$ and $17,000,000$ "lookups", respectively.

	Data	Futhark (ms)		PyTorch	JAX	JAX(vmap)
		Manual	AD	(ms)	(ms)	(ms)
A100	$\mathbf{D}_0$	12.6	41.1	41.1	15.5	27.5
	$\mathbf{D}_1$	19.0	10.6	8.7	2.1	107.9
	$\mathbf{D}_2$	94.3	108.9	922.0	206.5	976.4
MI100	$\mathbf{D}_0$	24.6	35.6	94.5	–	–
	$\mathbf{D}_1$	22.5	10.5	40.2	–	–
	$\mathbf{D}_2$	309.5	264.2	2303.2	–	–

Fig. 10. k-means clustering performance measurements for three different workloads. The **JAX** and **JAX**(vmap) implementations use array intrinsics and a vectorizing map, respectively. $\mathbf{D}_0 = (5, 494019, 35)$, $\mathbf{D}_1 = (1024, 10000, 256)$, $\mathbf{D}_2 = (1024, 2000000, 10)$ where each tuple is formatted as (k, n, d); k is the number of clusters and n the number of d-dimensional points. $\mathbf{D}_0$ corresponds to the KDD Cup dataset [33]. $\mathbf{D}_1$ and $\mathbf{D}_2$ were randomly generated. All data consists of 32-bit floating points.

down to micro-optimizations. LBM comprises a sequential loop containing a parallel loop. As Enzyme currently only supports differentiation of a single kernel, this requires some manual bookkeeping of the tape, whereas Futhark automatically handles the loop. Our overhead is significantly lower than Enzyme's; possibly because we can handle the tape more efficiently across the outer sequential loop.

D. Case Study 1: Dense k-means clustering

In this section, we benchmark the k-means example of section II-C. As shown in fig. 2, in Futhark the cost function (eq. (3)) is written via nested map and reduce operations. In first-order languages like PyTorch, the cost function must be realized via array primitives: to efficiently compute the cost function, we expand the all pairs norm between points P and centroids C: $||P - C||^2 = P^2 + C^2 - 2PC^T$. In expanded form, all terms can be computed using vectorized operations, with the PC^T term being computed by matrix multiplication.

We compare against a hand-written Futhark histogram-based implementation as well as AD-based implementations in PyTorch and JAX on three qualitatively different datasets. In PyTorch and JAX, array intrinsics like matrix multiplication (and the differentiation thereof) are compiled to extremely efficient hand-tuned GPU code; to better compare with Futhark's programming model, a second JAX implementation using JAX's vectorizing map operation, `vmap`, was written in close analog to the Futhark implementation. The results are shown in fig. 10. When the histograms benefit from the optimizations discussed in [24], the hand-written implementation can show significant speedup over our AD approach: up to $3.3\times$ on $\mathbf{D}_0$ on the A100. When they cannot, the AD approach can be faster

	Workload	Futhark (s)		PyTorch (s)	JAX (s)
		Manual	AD		
A100	movielens	0.06	0.16	1.47	0.38
	nytimes	0.09	0.30	5.24	1.35
	scrna	0.16	0.58	9.32	8.91
MI100	movielens	0.44	5.32	3.24	–
	nytimes	0.44	9.55	11.58	–
	scrna	0.42	2.87	20.81	–

Fig. 11. Sparse k-means performance measurements for three NLP workloads. $k = 10$ for all datasets, with a fixed iteration count of 10 and a 32-bit representation. The movielens dataset uses data from the ML 20M dataset described in [34] with dimensions $(139K, 131K)$ and a density of 0.11%. The nytimes and scrna datasets are the same as used in [35], with dimensions $(300K, 102K)$ and $(66K, 27K)$ and densities of 0.23% and 7.3%, respectively.

due to differing amounts of parallelism. Additionally, note that the MI100 uses Futhark's OpenCL backend, which—unlike CUDA—doesn't support floating-point atomic add operations. Instead, atomic updates are implemented via a spinlock which can result in significant additional overhead in the atomic histogram updates of the manual implementation and, to a lesser extent, accumulator updates in the AD implementation. Futhark AD is on par with or significantly faster than PyTorch on all datasets (as high as $8.5\times$ on the A100 and $8.7\times$ on the MI100). The intrinsics-based JAX implementation demonstrates significant speedup over Futhark on the $\mathbf{D}_0$ and $\mathbf{D}_1$ datasets, but Futhark demonstrates a $1.9\times$ speedup on the larger $\mathbf{D}_2$ dataset. On $\mathbf{D}_0$, the `vmap`-based JAX implementation has a $1.5\times$ speedup over Futhark, but Futhark demonstrates speedups of around $10\times$ on the other two datasets, likely a product of the fact that preservation of nested parallelism becomes more impactful as the number of clusters increases—we surmise JAX's vectorizing/flat approach limits locality optimizations. Our approach pays further dividends still: if the number of points is made larger, beyond $\mathbf{D}_2$'s two million, the PyTorch and JAX implementations run out of memory due manifestating the entire $n \times k$ array of point-cluster distances.

E. Case Study 2: Sparse k-means clustering

We have implemented a sparse formulation of k-means clustering, which uses a dense representation for the centroids and a sparse representation for the input points. The Futhark implementations use the CSR format, while PyTorch and JAX both use the COO format.[24] As explained in the previous section, we compute the cost with vectorized operations in the PyTorch and JAX implementations.

Figure 11 shows runtimes on three publicly available sparse NLP workloads. On the A100, our AD is slower than the manual code by a factor between $2.5 - 3.7\times$ due to an optimization allowing updates to fit in the L2 cache [24]. On the A100, Futhark AD demonstrates significant speedups as high as $17.5\times$ against AD competitors. On the MI100,

[24]PyTorch's functional AD constructs (`jvp` and `hvp`) currently raise runtime errors with CSR format. JAX's transformations only support batch COO representation.

Data	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

Fig. 12. GMM ADBench parameters for the datasets used in fig. 13; n is the number of points, d the dimensionality of the input data, and K the number of Gaussian distributions. All datasets use 64-bit floats. The corresponding datasets may be found on the ADBench GitHub repository (https://github.com/microsoft/ADBench).

	Measurement	$\mathbf{D}_0$	$\mathbf{D}_1$	$\mathbf{D}_2$	$\mathbf{D}_3$	$\mathbf{D}_4$	$\mathbf{D}_5$
A100	**PyT. Jacob.** (ms)	7.4	15.8	15.2	5.9	12.5	64.8
	Fut. Speedup	2.1	2.2	1.4	1.6	1.5	1.0
	PyT. Overhead	3.5	4.9	2.8	3.2	4.0	3.2
	Fut. Overhead	2.0	1.8	1.9	2.7	2.8	2.8
MI100	**PyT. Jacob.** (ms)	20.9	51.5	42.5	20.7	38.5	193.1
	Fut. Speedup	3.3	4.0	2.1	2.9	2.5	1.7
	PyT. Overhead	5.9	5.3	2.4	2.6	3.1	2.8
	Fut. Overhead	3.0	2.9	3.0	2.8	2.8	2.8

Fig. 13. GMM benchmark results on the A100 and MI100 systems. **Fut.** and **PyT.** refer to Futhark and PyTorch, respectively. **PyT. Jacob.** is the time to compute the full Jacobian of the objective function in PyTorch. On Futhark, block and register tile sizes of 16 and 3 were used, respectively.

PyTorch has modest speedup over Futhark on the movielens dataset, but Futhark is faster on the two other datasets.

F. Case Study 3: GMM

To evaluate the parallelism preservation of our AD transformation, we compile the Futhark implementation of the GMM benchmark from the ADBench suite to parallel CUDA. We compare against ADBench's implementation of GMM in PyTorch (also run on CUDA), which we have improved (by a $> 10\times$ factor) by vectorizing all comprehensions. We benchmark on a selection of 1,000 and 10,000-point datasets from ADBench, see fig. 12. The runtime of the primal program is dominated by matrix multiplication ($\sim 70\%$).

As discussed in section VI-D, matrix multiplication is a primitive in PyTorch [5]; we expect that differentiation of matrix multiplication is 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 V-A. The benchmark results are shown in fig. 13. The results demonstrate significant speedups over PyTorch on both systems, with an average speedup of $1.65\times$ on the A100 and of $2.75\times$ on the MI100. This demonstrates the feasibility of competitive AD performance in the absence of array primitives.

G. Case Study 4: LSTM

Long Short-Term Memory (LSTM) [36] is a type of recurrent neural network architecture popular in named entity recognition and part-of-speech tagging [37], [38]. We benchmark two LSTM networks with hyperparameters common in natural language processing [37], [39]–[41]. The AD-based implementations are based on the architecture in [36]. We also compare against PyTorch's `torch.nn.LSTM` class, which wraps the NVIDIA cuDNN LSTM implementation [42] on the A100 and AMD's MIGraphX on the MI100; note that

			Speedups			
		PyTorch Jacob.	**Futhark**	**nn.LSTM**	**JAX**	**JAX**(`vmap`)
A100	$\mathbf{D}_0$	45.4 ms	3.0	11.6	4.5	0.3
	$\mathbf{D}_1$	740.1 ms	3.3	22.1	6.4	0.9
MI100	$\mathbf{D}_0$	89.8 ms	2.6	4.0	–	–
	$\mathbf{D}_1$	1446.9 ms	1.8	5.4	–	–

		Overheads				
		PyTorch	**Futhark**	**nn.LSTM**	**JAX**	**JAX**(`vmap`)
A100	$\mathbf{D}_0$	4.1	2.1	2.7	3.5	1.4
	$\mathbf{D}_1$	4.3	3.9	2.2	3.7	0.8
MI100	$\mathbf{D}_0$	5.0	4.2	7.2	–	–
	$\mathbf{D}_1$	7.9	3.9	6.6	–	–

Fig. 14. LSTM speedups and overheads on $\mathbf{D}_0 = (1024, 20, 300, 192)$ and $\mathbf{D}_1(1024, 300, 80, 256)$ where each tuple is formatted as (bs, n, d, h); bs is the batch size, n the sequence length, d the dimensionality of the input data, and h the dimensionality of the hidden state. All data consists of 32-bit floating points. **nn.LSTM** refers to the `torch.nn.LSTM` implementation. Futhark was run with block and register tile sizes of 16 and 4.

Benchmark	Mem.	Depth	Mem. Overhead
RSBench	9.9 MiB	7	1.4
XSBench	225 MiB	7	1.0
LBM	550 MiB	5	33.6
GMM	4.1 GiB	5	2.1
LSTM	0.84 GiB	5	2.1

Fig. 15. Primal memory footprints, maximal program depth, and the memory AD overhead of the benchmarks. For benchmarks with multiple datasets, the memory footprint is reported for the dataset with the largest memory consumption ($\mathbf{D}_5$ for GMM and $\mathbf{D}_1$ for LSTM).

Benchmark	Dataset	Futhark Manual	Futhark AD
Dense k-means	$\mathbf{D}_0$	188 MiB	172 MiB
Sparse k-means	scrna	2.7 GiB	2.7 GiB

Fig. 16. Memory footprint comparison between the manual and AD Futhark implementations for dense and sparse k-means.

both implementations are hand-written/optimized and feature manual differentiation. The results are shown in fig. 14. Futhark is about $3\times$ faster than PyTorch on the A100 and slightly less on the MI100. As with GMM, LSTM is dominated by matrix multiplications. None of the AD-based implementations are competitive against the manual `torch.nn.LSTM` implementations.[25] JAX performs up to two times faster than Futhark when matrix multiplication is a highly optimized primitive and up to ten times slower when it's not.

H. Depth and Memory Consumption

Figure 15 shows the memory overhead (the ratio of the memory footprint of the differentiated and primal programs) and maximal program depths of the benchmarks. Futhark's memory overhead on LBM is large: this can be ameliorated by annotating the outer loop to be strip-mined; doing so modestly increases the AD overhead from 3.4 to 4.5, but decreases the memory overhead from 33.6 to 8.7. The memory overheads of the remaining benchmarks demonstrate the efficiency of our approach: the forward and reverse passes combined should

[25]The results appear correlated with the ratio between peak FLOPS with and without the usage of tensor cores.

use roughly twice as much memory as the primal program—the remaining benchmarks achieve this or better. The GPU benchmarks also feature non-trivial depth; nevertheless we achieve competitive results, in part because the forward sweep is often executed only once or twice, irrespective of depth—see section III-C. Memory overheads aren't applicable to the k-means benchmarks, but it's informative to compare the AD-based implementations to the manual ones; fig. 16 shows that the implementations all use a similar amount of memory.

VII. Related Work

Reverse mode differentiation of reduce and scan is discussed in [43]. Our rule for reduce is similar but was developed independently [44] and we handle scan differently: our approach is less efficient for complex operands because we manifest the Jacobian matrices, but more efficient for single-value operands on GPUs as it requires less shared memory to implement the derived scan operator. Neither our scan rule nor the one from [43] is asymptotics-preserving in general, but they are for most scans that occur in practice.

$\widetilde{F}$ is a functional array language that supports nested parallelism. Its AD implementation uses the forward mode, along with rewrite rules for exploiting sparsity in some cases [45].

Dex is a recent language built specifically to support efficient AD. Empirical benchmarks for AD in Dex have not yet been published, but we can compare with their approach [8]. In contrast to our conventional "monolithic" approach where reverse mode AD is a transformation completely distinct from forward mode, Dex uses a technique where the program is first linearized, producing a linear map, after which this linear map is then transposed, producing the adjoint code. Like Dex, we do not support recursion or AD of higher-order functions. Dex does not make direct use of a tape in the classical sense, but instead constructs arrays of closures followed by defunctionalization. The actual runtime data structures will conceptually consist of *multiple* tapes in the form of multidimensional irregular arrays. Dex does not report strategies for checkpointing, or optimization of particular accumulation patterns as in section V-A, or of tape accesses.

The time-space tradeoff for reverse mode AD is systematically studied by Siskind and Pearlmutter [21]. Tapenade [1] supports a wealth of checkpointing techniques; our loop strip-mining technique is a practical and simple special case of that.

Enzyme shows the advantage of performing AD after standard compiler optimizations have simplified the program [46]. Like Enzyme, we 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-optimization low-level representation, our work takes advantage of pre-AD optimization, the information provided by high-level parallel constructs, and post-AD optimization.

Enzyme has also been applied to GPU kernels, where it makes use of AD-specific GPU memory optimizations including caching tape values in thread-local storage as well as memory-aware adjoint updates [14]. 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 discussed in section V-A.

Recent AD work on OpenMP details an approach to reverse mode AD for non-nested parallel loops [47]. Updates to free variables in loops are handled by sharing adjoints across threads and atomic updates; our approach to map is similar, but preservation of nested parallelism allows us to identify and reduce some updates into a single atomic update.

ML practitioners use tools such as PyTorch [5] 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. JAX is another such example; it supports automatic differentiation of pure Python code and just-in-time (JIT) compilation to XLA HLO [6], [48]. Unlike PyTorch, JAX also features a vectorizing map operation which often yields good performance on some workloads. However, applying (reverse) AD on flat-parallel (vectorized) code may prevent further memory-locality optimizations; our approach applies AD to nested-parallel code, which preserves further optimization opportunities, as demonstrated by our treatment of matrix multiplication-like computations.

Reverse AD has also been implemented in DSLs aimed at stencil computations [49]. The challenge here is to combine AD with loop transformations such that the resulting code is a stencil itself and thus can be optimized with the repertoire of existent optimizations. AD has also been described for tensor languages that support constrained forms of loops, which in particular has the benefit of not requiring the use of tapes [50].

VIII. Conclusions

We have presented a fully operational compiler implementation of both reverse and forward mode AD in a nested-parallel, hardware-neutral functional language. Our transformation is based on using redundant execution to eliminate the need for an explicit tape and is performed before the parallelism of a program is mapped to hardware. It thus benefits from specialized rules for parallel constructors and flexibility in aggressively optimizing the original and AD code independently. Our experimental evaluation shows that our approach is effective in practice and competitive with both well-established frameworks that encompass more specialized languages such as PyTorch and JAX and with newer research efforts aimed at a lower-level language, such as Enzyme.

Acknowledgments

We are grateful to Rory Mitchell for suggesting the AD formulation of k-means, to Lotte Bruun and Ulrik Larsen for integrating the AD rules for scan and reduce-by-index, and to Martin Elsman for using Futhark's AD infrastructure.

This work has been supported by the Independent Research Fund Denmark (DFF) under the grants *Deep Probabilistic*

Programming for Protein Structure Prediction and *FUTHARK: Functional Technology for High-performance Architectures*, and by the UCPH Data-Plus grant: *High-Performance Land Change Assessment*.

REFERENCES

[1] M. Araya-Polo and L. Hascoët, "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. Jyväskylä, Finland: University of Jyväskylä, 2004, online at http://www.mit.jyu.fi/eccomas2004/proceedings/pdf/550.pdf.

[2] A. Griewank, D. Juedes, and J. Utke, "Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++," *ACM Transactions on Mathematical Software (TOMS)*, vol. 22, no. 2, pp. 131–167, 1996.

[3] B. A. Pearlmutter and J. M. Siskind, "Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 2, Mar. 2008. [Online]. Available: https://doi.org/10.1145/1330017.1330018

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "PyTorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[6] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018. [Online]. Available: http://github.com/google/jax

[7] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea, "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*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 556–571. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062354

[8] A. Paszke, D. D. Johnson, D. Duvenaud, D. Vytiniotis, A. Radul, M. J. Johnson, J. Ragan-Kelley, and D. Maclaurin, "Getting to the point: Index sets and parallelism-preserving autodiff for pointful array programming," *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, aug 2021. [Online]. Available: https://doi.org/10.1145/3473593

[9] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. Devito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated GPU kernels, automatically," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, oct 2019. [Online]. Available: https://doi.org/10.1145/3355606

[10] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: A survey," *J. Mach. Learn. Res.*, vol. 18, no. 1, p. 5595–5637, Jan. 2017.

[11] F. Wang, D. Zheng, J. Decker, X. Wu, G. M. Essertel, and T. Rompf, "Demystifying differentiable programming: Shift/reset the penultimate backpropagator," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3341700

[12] T. Henriksen, F. Thorøe, M. Elsman, and C. Oancea, "Incremental flattening for nested data parallelism," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: ACM, 2019, pp. 53–67. [Online]. Available: http://doi.acm.org/10.1145/3293883.3295707

[13] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 101–113. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375595

[14] W. S. Moses, V. Churavy, L. Paehler, J. Hückelheim, S. H. K. Narayanan, M. Schanen, and J. Doerfert, "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*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476165

[15] M. Elsman, T. Henriksen, D. Annenkov, and C. E. Oancea, "Static interpretation of higher-order modules in Futhark: Functional GPU programming in the large," *Proceedings of the ACM on Programming Languages*, vol. 2, no. ICFP, pp. 97:1–97:30, Jul. 2018.

[16] A. K. Hovgaard, T. Henriksen, and M. Elsman, "High-performance defunctionalization in Futhark," in *Symposium on Trends in Functional Programming (TFP'18)*, September 2018.

[17] O. Manzyuk, B. A. Pearlmutter, A. A. Radul, D. R. Rush, and J. M. Siskind, "Perturbation confusion in forward automatic differentiation of higher-order functions," *Journal of Functional Programming*, vol. 29, p. e12, 2019.

[18] A. Sabry and M. Felleisen, "Reasoning about programs in continuation-passing style." *SIGPLAN Lisp Pointers*, vol. V, no. 1, pp. 288–298, Jan. 1992.

[19] L. Bottou and Y. Bengio, "Convergence properties of the k-means algorithms," in *Advances in Neural Information Processing Systems*, G. Tesauro, D. Touretzky, and T. Leen, Eds., vol. 7. MIT Press, 1994. [Online]. Available: https://proceedings.neurips.cc/paper/1994/file/a1140a3d0df1c81e24ae954d935e8926-Paper.pdf

[20] T. A. Mogensen, *Introduction to Compiler Design*, 1st ed. Springer Publishing Company, Incorporated, 2011.

[21] J. M. Siskind and B. A. Pearlmutter, "Divide-and-conquer checkpointing for arbitrary programs with no user annotation," *Optimization Methods and Software*, vol. 33, no. 4-6, pp. 1288–1330, 2018. [Online]. Available: https://doi.org/10.1080/10556788.2018.1459621

[22] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 54:1–54:23, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400713

[23] P. Hovland and C. Bischof, "Automatic differentiation for message-passing parallel programs," in *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, 1998, pp. 98–104.

[24] T. Henriksen, S. Hellfritzsch, P. Sadayappan, and C. Oancea, "Compiling generalized histograms for GPU," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[25] G. E. Blelloch, "Prefix sums and their applications," 1990.

[26] B. Lu and J. Mellor-Crummey, "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*, 1998, pp. 42–51.

[27] A. Rasch, R. Schulze, and S. Gorlatch, "Generating portable high-performance code via multi-dimensional homomorphisms," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019, pp. 354–369.

[28] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam, "Interprocedural Parallelization Analysis in SUIF," *Trans. on Prog. Lang. and Sys. (TOPLAS)*, vol. 27(4), pp. 662–731, 2005.

[29] F. Dang, H. Yu, and L. Rauchwerger, "The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops," in *Int. Par. and Distr. Processing Symp. (PDPS)*, 2002, pp. 20–29.

[30] C. E. Oancea and L. Rauchwerger, "Logical inference techniques for loop parallelization," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 509–520. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254124

[31] A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[32] F. Srajer, Z. Kukelova, and A. Fitzgibbon, "A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning," *Optimization Methods & Software*, vol. 33, no. 4–6, pp. 889–906, 2018. [Online]. Available: https://doi.org/10.1080/10556788.2018.1435651

[33] (1999) KDD Cup 1999 data. [Online]. Available: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

[34] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Trans. Interact. Intell. Syst.*, vol. 5, no. 4, dec 2015. [Online]. Available: https://doi.org/10.1145/2827872

[35] C. J. Nolet, D. Gala, E. Raff, J. Eaton, B. Rees, J. Zedlewski, and T. Oates, "GPU semiring primitives for sparse neighborhood methods," 2021. [Online]. Available: https://arxiv.org/abs/2104.06357

[36] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition," 2014. [Online]. Available: https://arxiv.org/abs/1402.1128

[37] J. P. Chiu and E. Nichols, "Named entity recognition with bidirectional LSTM-CNNs," *Transactions of the Association for Computational Linguistics*, vol. 4, pp. 357–370, 2016.

[38] O. Rønning, D. Hardt, and A. Søgaard, "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)*, 2018, pp. 236–241.

[39] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[40] M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz, "Building a large annotated corpus of English: The Penn Treebank," *Computational Linguistics*, vol. 19, no. 2, pp. 313–330, 1993. [Online]. Available: https://aclanthology.org/J93-2004

[41] A. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*, 2011, pp. 142–150.

[42] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient primitives for deep learning," 2014. [Online]. Available: https://arxiv.org/abs/1410.0759

[43] A. Paszke, M. J. Johnson, R. Frostig, and D. Maclaurin, "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*, ser. FHPNC 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 13–23. [Online]. Available: https://doi.org/10.1145/3471873.3472975

[44] C. E. Oancea, T. Henriksen, and R. Schenck, "Reverse mode automatic differentiation. *Lecture Slides for the Parallel Functional Programming MSc Course*," Dec. 2020. [Online]. Available: https://github.com/diku-dk/pfp-e2020-pub/blob/master/slides/L8-reverse-ad.pdf

[45] A. Shaikhha, A. Fitzgibbon, D. Vytiniotis, and S. Peyton Jones, "Efficient differentiable programming in a functional array-processing language," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, jul 2019. [Online]. Available: https://doi.org/10.1145/3341701

[46] W. S. Moses and V. Churavy, "Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients," in *Advances in Neural Information Processing Systems 33*, 2020.

[47] J. Hückelheim and L. Hascoët, "Source-to-source automatic differentiation of OpenMP parallel loops," 2021. [Online]. Available: https://arxiv.org/abs/2111.01861

[48] R. Frostig, M. J. Johnson, and C. Leary, "Compiling machine learning programs via high-level tracing," *Systems for Machine Learning*, pp. 23–24, 2018.

[49] J. Hückelheim, N. Kukreja, S. H. K. Narayanan, F. Luporini, G. Gorman, and P. Hovland, "Automatic differentiation for adjoint stencil loops," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP 2019. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3337821.3337906

[50] G. Bernstein, M. Mara, T.-M. Li, D. Maclaurin, and J. Ragan-Kelley, "Differentiating a tensor language," 2020. [Online]. Available: https://arxiv.org/abs/2008.11256