Reverse AD for a Functional Array Language with Nested Parallelism

Cosmin E. Oancea in collaboration with
Troels Henriksen, Ola Rønning and Robert Schenck

Department of Computer Science (DIKU)
University of Copenhagen

December 2021, DPP Lecture Slides

Material for This Lecture

This lecture material corresponds to the PLDI'22 submission:

- "AD for an Array Language with Nested Parallelism", anonymous authors (due to double-blind review, but available on Absalon),
- whose content is surprisingly similar to the lecture slides presented at DPP last year, but now it is also backed up by an end-to-end compiler implementation, and a performance evaluation of results.

Inspiration sources:

- "Automatic Differentiation in Machine Learning: a Survey", A. G. Baydin, B. A. Pearlmutter, A. A. Radul and J. M. Siskind, *Journal of Machine Learning Research*, Vol. 18, pages: 1–43, 2018;

  This is a paper that presents in a very-accessible, concise and crystal-clear way the intuition behind automatic differentiation. We credit this paper for "opening our eyes", and for being the main source of inspiration for the work/ideas presented in these slides.

- High-School Math Curiculum, which was brushed up with the help of wikipedia-1 and wikipedia-2.

Motivation for Automatic Differentiation (AD)

What is AD?

- A practical way for computing derivatives of functions that are expressed as programs.

Motivation from Application Perspective:

- one of the driving forces for smartening ML algorithms;
- financial algorithms, computational fluid dynamics, atmospheric sciences, etc.

Motivation from Our Perspective:

Motivation for Automatic Differentiation (AD)

What is AD?

- A practical way for computing derivatives of functions that are expressed as programs.

Motivation from Application Perspective:

- one of the driving forces for smartening ML algorithms;
- financial algorithms, computational fluid dynamics, atmospheric sciences, etc.

Motivation from Our Perspective:

- we heard it is a difficult problem (code transformation),
- and took it as a challenge ...

Math: Simple Differentiation Rules $h : \mathbb{R} \to \mathbb{R}$

Basic Rules:

sin: $\frac{\partial sin\ x}{\partial x} = cos\ x$

cos: $\frac{\partial cos\ x}{\partial x} = -sin\ x$

pow: $\frac{\partial x^r}{\partial x} = r \cdot x^{r-1}, \quad \forall r \in \mathbb{R},\ r \neq 0$

log: $\frac{\partial log_c\ x}{\partial x} = \frac{1}{x \cdot ln\ c}$ where
$ln\ x = log_e\ x, e = 2.7182818284 59\ldots$ Euler's number.

- ...

How do we combine this rules?

Math: Simple Differentiation Rules $h : \mathbb{R} \to \mathbb{R}$

Basic Rules:

sin: $\frac{\partial sin\ x}{\partial x} = cos\ x$

cos: $\frac{\partial cos\ x}{\partial x} = -sin\ x$

pow: $\frac{\partial x^r}{\partial x} = r \cdot x^{r-1}, \quad \forall r \in \mathbb{R},\ r \neq 0$

log: $\frac{\partial log_c\ x}{\partial x} = \frac{1}{x \cdot ln\ c}$ where
$ln\ x = log_e\ x, e = 2.718281828459\ldots$ Euler's number.

- ...

How do we combine this rules?

Linear: If $h(x) = a \cdot f(x) + b \cdot g(x)$ then $\frac{\partial h}{\partial x} = a \cdot \frac{\partial f}{\partial x} + b \cdot \frac{\partial g}{\partial x}$

Product: If $h(x) = f(x) \cdot g(x)$ then $\frac{\partial h}{\partial x} = \frac{\partial f}{\partial x} \cdot g + f \cdot \frac{\partial g}{\partial x}$

Quotient: If $h(x) = \frac{f(x)}{g(x)}$ then $\frac{\partial h}{\partial x} = \frac{\frac{\partial f}{\partial x} \cdot g + \frac{\partial g}{\partial x} \cdot f}{g^2}$

GenPower: If $h(x) = f(x)^{g(x)}$ then $\frac{\partial h}{\partial x} = f^g \cdot (\frac{\partial f}{\partial x} \cdot \frac{g}{f} + \frac{\partial g}{\partial x} \cdot ln\ f)$

GenLog: If $h(x) = ln(f(x))$ then $\frac{\partial h}{\partial x} = \frac{\frac{\partial f}{\partial x}}{f}$, where f is positive.

I have not mentioned the most important rule (?)

Math: Chain Rule

If $h(x) = (f \circ g)(x) = f(g(x))$ then $\frac{\partial h(x)}{\partial x} = \frac{\partial f(z)|_{z=g(x)}}{\partial z} \cdot \frac{\partial g(x)}{\partial x}$

1^{st} Generalization: partial derivatives of $f : \mathbb{R}^n \to \mathbb{R}$:

Intuition. $f : \mathbb{R}^2 \to \mathbb{R}$, $f(y, x) = x^2 + xy + y^2$
Fix $y = a$, define $f_a : \mathbb{R} \to \mathbb{R}$, $f_a(x) = f(a, x) = x^2 + ax + a^2$
$\frac{\partial f_a(x)}{\partial x} = 2x + a$, i.e., a different function for each instance of y.

Math: Chain Rule

If $h(x) = (f \circ g)(x) = f(g(x))$ then $\frac{\partial h(x)}{\partial x} = \frac{\partial f(z)|_{z=g(x)}}{\partial z} \cdot \frac{\partial g(x)}{\partial x}$

1^{st} Generalization: partial derivatives of $f : \mathbb{R}^n \to \mathbb{R}$:

Intuition. $f : \mathbb{R}^2 \to \mathbb{R}$, $f(y, x) = x^2 + xy + y^2$
Fix $y = a$, define $f_a : \mathbb{R} \to \mathbb{R}$, $f_a(x) = f(a, x) = x^2 + ax + a^2$
$\frac{\partial f_a(x)}{\partial x} = 2x + a$, i.e., a different function for each instance of y.

The i^{th} partial derivative of f in point $a \in \mathbb{R}^n$ is defined as:
$\frac{\partial f(a_1,\ldots,a_n)}{\partial x_i}(x) = lim_{h \to 0} \frac{f(a_1,\ldots,a_{i-1},x+h,a_{i+1},\ldots,a_n) - f(a_1,\ldots,a_{i-1},x,a_{i+1},\ldots,a_n)}{h}$

- Hence the differential (derivative) is a

Math: Chain Rule

If $h(x) = (f \circ g)(x) = f(g(x))$ then $\frac{\partial h(x)}{\partial x} = \frac{\partial f(z)|_{z=g(x)}}{\partial z} \cdot \frac{\partial g(x)}{\partial x}$

1^{st} Generalization: partial derivatives of $f : \mathbb{R}^n \to \mathbb{R}$:

Intuition. $f : \mathbb{R}^2 \to \mathbb{R}$, $f(y, x) = x^2 + xy + y^2$
Fix $y = a$, define $f_a : \mathbb{R} \to \mathbb{R}$, $f_a(x) = f(a, x) = x^2 + ax + a^2$
$\frac{\partial f_a(x)}{\partial x} = 2x + a$, i.e., a different function for each instance of y.

The i^{th} partial derivative of f in point $a \in \mathbb{R}^n$ is defined as:
$\frac{\partial f(a_1,\ldots,a_n)}{\partial x_i}(x) = lim_{h \to 0} \frac{f(a_1,\ldots,a_{i-1},x+h,a_{i+1},\ldots,a_n) - f(a_1,\ldots,a_{i-1},x,a_{i+1},\ldots,a_n)}{h}$

- Hence the differential (derivative) is a **higher-order function**.
- The differential of $f : \mathbb{R} \to \mathbb{R}$ is

Math: Chain Rule

If $h(x) = (f \circ g)(x) = f(g(x))$ then $\frac{\partial h(x)}{\partial x} = \frac{\partial f(z)|_{z=g(x)}}{\partial z} \cdot \frac{\partial g(x)}{\partial x}$

1^{st} Generalization: partial derivatives of $f : \mathbb{R}^n \to \mathbb{R}$:

Intuition. $f : \mathbb{R}^2 \to \mathbb{R}$, $f(y, x) = x^2 + xy + y^2$
Fix $y = a$, define $f_a : \mathbb{R} \to \mathbb{R}$, $f_a(x) = f(a, x) = x^2 + ax + a^2$
$\frac{\partial f_a(x)}{\partial x} = 2x + a$, i.e., a different function for each instance of y.

The i^{th} partial derivative of f in point $a \in \mathbb{R}^n$ is defined as:
$\frac{\partial f(a_1,\ldots,a_n)}{\partial x_i}(x) = lim_{h \to 0} \frac{f(a_1,\ldots,a_{i-1},x+h,a_{i+1},\ldots,a_n) - f(a_1,\ldots,a_{i-1},x,a_{i+1},\ldots,a_n)}{h}$

- Hence the differential (derivative) is a **higher-order function**.
- The differential of $f : \mathbb{R} \to \mathbb{R}$ is the same function no matter the point, but this does not hold for $f : \mathbb{R}^{n>1} \to \mathbb{R}^{m \geq 1}$.
- $df_a(v) = \frac{\partial f(a)}{\partial a}(v) = \nabla f(a) \cdot v$, where $\cdot$ is the dot product and $\nabla f(a) = \left[\frac{\partial f(a)}{\partial a_1}(a_1) \ldots \frac{\partial f(a)}{\partial a_n}(a_n)\right]$

Math: Chain Rule for Multi-Variate Functions

Given $f : \mathbb{R}^n \to \mathbb{R}^m$, $f(x) = (f_1(x), \ldots, f_m(x))$, $f_i : \mathbb{R}^n \to \mathbb{R}$, $a \in \mathbb{R}^n$

$$J_f(a) = \left[\frac{\partial f(a)}{\partial a_1}(a_1) \ldots \frac{\partial f(a)}{\partial a_n}(a_n)\right] = \begin{bmatrix} \frac{\partial f_1(a)}{\partial a_1}(a_1) & \ldots & \frac{\partial f_1(a)}{\partial a_n}(a_n) \\ & \ldots & \\ \frac{\partial f_m(a)}{\partial a_1}(a_1) & \ldots & \frac{\partial f_m(a)}{\partial a_n}(a_n) \end{bmatrix}$$

$$\frac{\partial f(a)}{\partial a}(y) = J_f(a) \cdot y$$

Chain Rule for $f : \mathbb{R}^n \to \mathbb{R}^m, g : \mathbb{R}^m \to \mathbb{R}^k$ and $p \in \mathbb{R}^n$ is:

$$J_{g \circ f}(p) = J_g(f(p)) \cdot J_f(p)$$

Intuition: Differentiating a Program in Forward Mode

Consider $f_i : \mathbb{R}^{n_{i-1}} \to \mathbb{R}^{n_i}, \forall i = 1 \ldots m$, and "program" P defined as:

$$P : \mathbb{R}^{n_0} \to \mathbb{R}^{n_m} = f_m \circ f_{m-1} \circ \ldots \circ f_2 \circ f_1$$

Applying the Chain Rule for some $x \in \mathbb{R}^{n_0}$ results in:

$$J_P(x) = J_{f_m}(f_{m-1}(\ldots f_1(x))) \cdot \ldots \cdot J_{f_2}(f_1(x)) \cdot J_{f_1}(x)$$

Should compute $J_P(x)$, from right-to-left or from left-to-right?

$\leftarrow$ Forward mode (from right to left) $\longleftarrow$

- **better when input size is less than or comparable to the length of the output,** i.e., $n_0 \leq n_m$ or $n_0 \sim n_m$;
- the Jacobian J_{f_i} is computed in the same time as $f_i(x)$;
- Think $J_{f_1}(x) \in \mathbb{R}^{n \times 1}$, the others $J_{f_{i>1}}(x) \in \mathbb{R}^{n \times n}$.
  Complexity: $O(n^2)$ **instead of** $O(n^3)$**.**

Intuition: Differentiating a Program in Backward Mode

Consider $f_i : \mathbb{R}^{n_{i-1}} \to \mathbb{R}^{n_i}, \forall i = 1 \ldots m$, and "program" P defined as:

$$P : \mathbb{R}^{n_0} \to \mathbb{R}^{n_m} = f_m \circ f_{m-1} \circ \ldots \circ f_2 \circ f_1$$

Applying the Chain Rule for some $x \in \mathbb{R}^{n_0}$ results in:

$$J_P(x) = J_{f_m}(f_{m-1}(\ldots f_1(x))) \cdot \ldots \cdot J_{f_2}(f_1(x)) \cdot J_{f_1}(x)$$

Should compute $J_P(x)$, from right-to-left or from left-to-right?

$\to$ Reverse mode (from left to right) $\longrightarrow$

- **better when** $n_0 \gg n_m$, e.g., if $J_{f_m} \in \mathbb{R}^n$ and $J_{f_{i<m}} \in \mathbb{R}^{n \times n}$, then complexity is $O(n^2)$ instead of $O(n^3)$ with the forward mode!
- typically the common case for AI;
- Big Challenge: $f_{m-1}(\ldots f_1(x))$ needs to be computed before we start computing the Jacobians $\Rightarrow$ tape abstraction.

What is NOT Automatic Differentiation?

What is not AD:

- manually working out derivatives and coding them:
  time consuming and error prone;

- numerical differentiation using finite-difference approximation:
  - may be highly inaccurate due to round-off & truncation error;
  - it scales poorly to gradients

    e.g., where gradients w.r.t. million of parameters are needed.

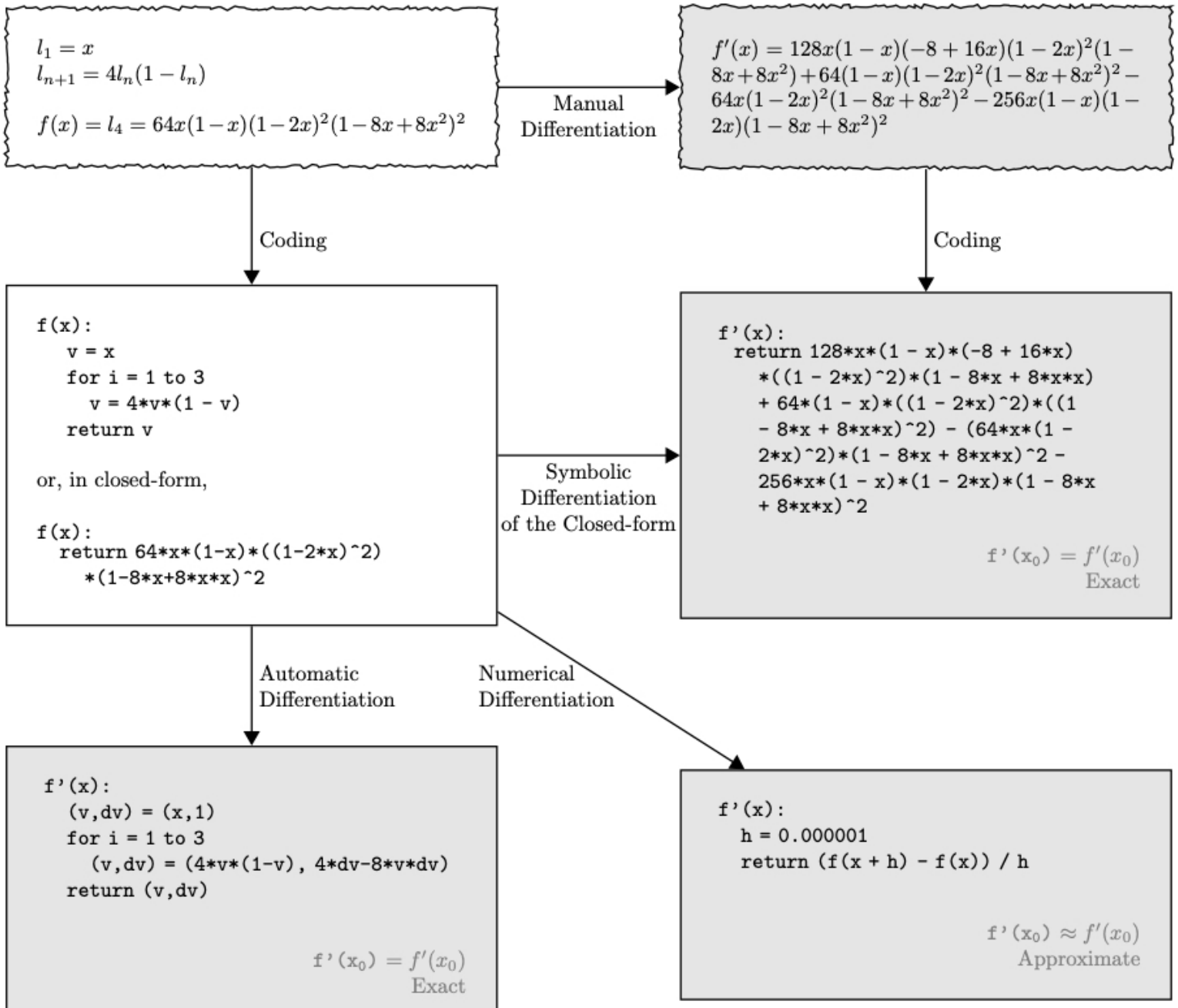
- symbolic differentiation in compute-algebra systems such as Mathematica, Maxima, Maple:
  - solves the problems above, but may suffer ”expression swell” and result in cryptic/complex expressions;
  - requires the problem to be modeled as closed-formed expressions (formula), severely limiting control flow, expressiveness, and the application of compiler-like optimizations.

What is Automatic Differentiation (AD)?

"Automatic (or algorithmic) differentiation performs a non-standard interpretation of a given program by replacing the domain of variables to incorporate derivative values and redifinging the semantics of operators to propagate derivatives per the chain rule of differential calculus." [Baydin et. al. 2018].

AD can be applied to regular code with minimal change, allowing branching, loops, and even recursion.

What is AD? [Baydin et. al. 2018]



Main Mathematical Re-Write Rule for Forward Mode

$$\frac{\partial w_i}{\partial x} = \frac{\partial w_i}{\partial w_{i-1}} \cdot \frac{\partial w_{i-1}}{\partial x}$$

Quantity of interest is the derivative of some variable w, stored numerically and not as a symbolic expression, and denoted by

$$\dot{w} = \frac{\partial w}{\partial x}$$

Assume $y = f(\, g(\, h(\, x\,)\,)\,)$

w_0 = x
w_1 = h(w_0)
w_2 = g(w_1)
w_3 = f(w_2)
y = w_3

$$\dot{y} = \frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_2} \cdot \frac{\partial w_2}{\partial x} = \frac{\partial y}{\partial w_2} \cdot (\frac{\partial w_2}{\partial w_1} \cdot \frac{\partial w_1}{\partial x}) = \frac{\partial y}{\partial w_2} \cdot (\frac{\partial w_2}{\partial w_1} \cdot (\frac{\partial w_1}{\partial x} \cdot \frac{\partial x}{\partial x}))$$

Essentially, derivatives are computed in order, at the place where each variable is defined (once). Generalized to multiple input variables as a matrix product of Jacobians.

Running Example and Notation

[Baydin et. al., Automatic Differentiation in Machine Learning: a Survey, 2018]

Running example: $y = f(x_1, x_2) = ln(x_1) + x_1 \cdot x_2 - sin(x_2)$

Will use the notation used by Griewank and Walther (2008), where a function $f : \mathbb{R}^n \to \mathbb{R}^m$ is constructed from variables v_i:

- $v_{i-n} = x_i,\ i = 1, \ldots, n$ are the input variables;
- $v_i,\ i = 1, \ldots, l$ are the intermediate variables;
- $y_{m-i} = v_{l-i},\ i = m-1, \ldots, 0$ are the output variables.

AD is blind w.r.t. any operation, including control flow statements, which do not directly alter numeric values.

Forward Mode: Main Intuition

[Baydin et. al., Automatic Differentiation in Machine Learning: a Survey, 2018]

Running example: $y = f(x_1, x_2) = ln(x_1) + x_1 \cdot x_2 - sin(x_2)$

- For computing derivative w.r.t. x_1, we associate with each intermediate value v_i a derivative $\dot{v}_i = \frac{\partial v_i}{\partial x_1}$
- applying the chain rule to each elementary operation in the formal primal trace (i.e., original program) results in the corresponding tangent (derivative) trace on the right.
- evaluating the primals v_i in lockstep with their tangents $\dot{v}_i$ gives the required derivative in final variable $\dot{v}_5 = \frac{\partial y}{\partial x_1}$

Forward Mode: Applied to Basic-Block Code

[Baydin et. al., Automatic Differentiation in Machine Learning: a Survey, 2018]

Running example: $y = f(x_1, x_2) = ln(x_1) + x_1 \cdot x_2 - sin(x_2)$

Table 2: Forward mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ evaluated at $(x_1, x_2) = (2, 5)$ and setting $\dot{x}_1 = 1$ to compute $\frac{\partial y}{\partial x_1}$. The original forward evaluation of the primals on the left is augmented by the tangent operations on the right, where each line complements the original directly to its left.

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

Forward Mode: Generalization to Multiple Dimensions

[Baydin et. al., Automatic Differentiation in Machine Learning: a Survey, 2018]

Assume $f : \mathbb{R}^n \to \mathbb{R}^m$ with input $x_{1\ldots n}$ and output $y_{1\ldots m}$:

- requires n runs of the code,
- in which run i initializes $\dot{x} = e_i$ (the i-th unit vector), $x = a$
- and computes one column of the Jacobian matrix $J_f(a)$:

$$\dot{y_j} = \frac{\partial y_j}{\partial x_i}|_{x=a},\ j = 1, \ldots, m$$

Essentially, a `map` operation over the n unit vectors, where the unnamed function $\lambda \dot{x_i} \to \ldots$ implements the forward mode generically for each value of $\dot{x_i} = e_i$.

Forward Mode: Dual Numbers

[Baydin et. al., Automatic Differentiation in Machine Learning: a Survey, 2018]

Forward mode seen as evaluating a function by dual numbers:

$v + \dot{v}\epsilon$, where $v, \dot{v} \in \mathbb{R}$ and $\epsilon \neq 0$, but $\epsilon^2 = 0$:

- $(v + \dot{v}\epsilon) + (u + \dot{u}\epsilon) = (v + u) + (\dot{v} + \dot{u})\epsilon$
- $(v + \dot{v}\epsilon) \cdot (u + \dot{u}\epsilon) = (vu) + (v\dot{u} + \dot{v}u)\epsilon$
- Seting up a regime such as $f(v + \dot{v}\epsilon) = f(v) + f'(v)\,\dot{v}\epsilon$ makes the chain rule work as expected:
  $f(g(v + \dot{v}\,\epsilon)) = f(g(v)) + f'(g(v))\,g'(v)\,\dot{v}\,\epsilon$

$\frac{\partial f(x)}{\partial x}|_{x=v} = \texttt{espsilon-coeff(dual-version(f)}(v + 1\epsilon)\texttt{)}$

Main Mathematical Re-Write Rule for Backward Mode

$$\frac{\partial y}{\partial w_i} = \frac{\partial y}{\partial w_{i+1}} \cdot \frac{\partial w_{i+1}}{\partial w_i}$$

Quantity of interest is the adjoint—i.e., the derivative of a chosen variable w.r.t. an expression—computed numerically.

The adjoint of some w is denoted by: $\overline{w} = \frac{\partial y}{\partial w}$

Since *w* can be used/read many times, its adjoint is accumulated.

Assume $y = f(\, g(\, h(\, x\,)\,)\,)$

w_0 = x
w_1 = h(w_0)
w_2 = g(w_1)
w_3 = f(w_2)
y = w_3

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

Reverse Mode: Main Intution

[Baydin, Pearlmutter, Radul, Siskind, "Automatic Differentiation in Machine Learning: a Survey", J. Mach. Learn. Res., 2017]

Propagates derivatives backward from a given output: achieved by complementing each intermediate variable v_i with an adjoint $\overline{v_i} = \frac{\partial y_j}{v_i}$, representing the sensitivity of output y_j to changes in v_i.

Phase 1: original code run forward, populating intermediate vars v_i

Phase 2: derivatives are calculated by propagating adjoints in reverse, from the outputs to the inputs.

- computes a row of the Jacobian at a time!

If one output y, then start with $\overline{y} = \frac{\partial y}{\partial y} = 1$.
If m outputs, then start a computation for each output with $\overline{y} = e_i,\ i = 1 \ldots m$, where e_i is the i-th unit vector as before!

Reverse Mode: Applied to Basic-Block Code

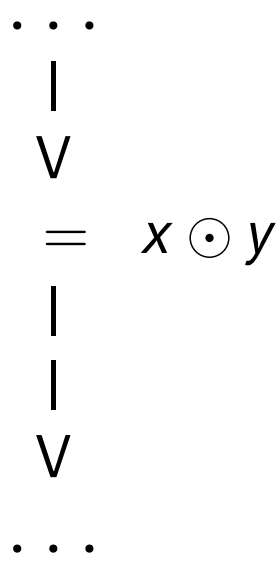
[Baydin, Pearlmutter, Radul, Siskind, "Automatic Differentiation in Machine Learning: a Survey", J. Mach. Learn. Res., 2017]

Table 3: Reverse mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ evaluated at $(x_1, x_2) = (2, 5)$. After the forward evaluation of the primals on the left, the adjoint operations on the right are evaluated in reverse (cf. Figure 1). Note that both $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$ are computed in the same reverse pass, starting from the adjoint $\bar{v}_5 = \bar{y} = \frac{\partial y}{\partial y} = 1$.

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

Reverse AD: Core Rewrite Rule

Primal Trace:



Reverse Trace:

$$\overline{x} = \overline{x} + \frac{\partial x \odot y}{\partial x} \cdot \overline{z}$$
$$\overline{y} = \overline{y} + \frac{\partial x \odot y}{\partial y} \cdot \overline{z}$$

(*) final value of $\overline{z}$ is known;
the values of x and y need to be available, e.g., used in $\frac{\partial x \odot y}{\partial x}$

(**) $\overline{z}$ is dead after this point;
$\overline{x}$ and $\overline{y}$ still under computation;

- the adjoint are initialized (zeroed) before the first use.

Related Work in the Sequential Context

One difficulty is the tape.

- Reverse AD elegantly modeled as a compiler transformation.
- The tape is hidden by powerful programming abstractions:
  - closures [Pearlmutter and Siskind]: the ultimate backpropagator
  - delimited continuations [Wang, Zheng, Decker, Wu, Essertel, Rompf]: the penultimate backpropagator.
- not suited for parallel execution, e.g., GPUs.

Reverse Mode: Applied to Basic-Block Code

[Baydin, Pearlmutter, Radul, Siskind, "Automatic Differentiation in Machine Learning: a Survey", J. Mach. Learn. Res., 2017]

Table 3: Reverse mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ evaluated at $(x_1, x_2) = (2, 5)$. After the forward evaluation of the primals on the left, the adjoint operations on the right are evaluated in reverse (cf. Figure 1). Note that both $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$ are computed in the same reverse pass, starting from the adjoint $\bar{v}_5 = \bar{y} = \frac{\partial y}{\partial y} = 1$.

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

Key Ideas for a Functional Data-Parallel Language

Key Simple Idea 1:

- Make **the tape** part of the program by
- **redundantly executing the forward trace on each new scope.**
- Preserves the work (and depth) asymptotics of the program;
- Unveils important time-space trade-off:
  loop merging/flattening *vs.* loop strip mining/nesting.
- Optimization space navigated by classical transformations.

Key Ideas for a Functional Data-Parallel Language

Key Simple Idea 1:

- Make **the tape** part of the program by
- **redundantly executing the forward trace on each new scope.**
- Preserves the work (and depth) asymptotics of the program;
- Unveils important time-space trade-off:
  loop merging/flattening *vs.* loop strip mining/nesting.
- Optimization space navigated by classical transformations.

Key Idea 2:

- A functional data-parallel array language is easier to AD.
- SOACs $\Rightarrow$ high-level reasoning (by means of re-write rules).
- A good example where PL-based reasoning helps Math:
  - we use only basic differentiation rules (for +, *, etc.), and
  - the core re-write rule (mentioned before).
  - everything else emerges naturally from rewrite-rule reasoning and dependency analysis.

Tape by Redundant Execution & Checkpointing

- whenever a new scope is entered, the code generation of the reverse trace first re-executes the primal trace of that scope;
- the re-execution overhead is at worst proportional with the deepest scope nest of the program (which is constant);
- “the tape” is part of the program and subject to aggressive optimizations (especially in a purely functional context);
- in most cases, it is more efficient to re-compute scalars rather than to access them from the tape (global memory);
- **loops require checkpointing, parallel constructs do not.**
- Subject to checkpointing are only the loops appearing directly in the current scope of the reverse-trace code generation (i.e., inner loops of the primal trace are not).

Tape by Redundant Execution & Checkpointing

Original Loop

$stms^{bef}_{out}$
let y'' =
  loop (y)=(y_0)
  for i = $0 \ldots m^k - 1$ **do**
    $stms_{loop}$ **in** y'
$stms^{after}_{out}$

Original loop Stripmined $k\times$

$stms^{bef}_{out}$
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'
$stms^{after}_{out}$

Reverse AD Result on Original Loop

$\overrightarrow{stms^{bef}_{out}}$
let ys_0 = scratch(m^k, sizeOf(y_0))
let (y'', ys) =
  loop (y,ys)=(y_0,ys_0)
  for i = $0 \ldots m^k - 1$ **do**
    let ys[i] = y
    $stms_{loop}$ **in** (y', ys)
$\overrightarrow{stms^{after}_{out}}$
$\overleftarrow{stms^{after}_{out}}$
let ($\overline{y'''}$, ...) =
  loop ($\overline{y}$,...)=($\overline{y''}$,...)
  for i = $m^k - 1 \ldots 0$ **do**
    let y = ys[i]
    $\overrightarrow{stms_{loop}}$
    $\overleftarrow{stms_{loop}}$
    in ($\overline{y'}$, ...)
vjp_{smt}(**let** $y''' = y_0$)
$\overleftarrow{stms^{bef}_{out}}$

Stripmining $k\times$: **up to $k\times$ slower,** memory overhead: $k \times m$ vs m^k

Tape by Redundant Execution & Checkpointing

- **Stripmining a loop** $k\times$ results in a classical time-space trade-off: **linear slowdown** (up to $k\times$) but the **memory overhead is reduced exponentially** (from m^k to $k \times m$).

- Creating perfect nests of parallel constructs or loops minimizes the re-computation overhead $\Rightarrow$ achieved with classical code transformation such as map fission or loop distribution. (Loops should be also flattened.)

- In most cases, it is more efficient to re-compute scalars rather than to access them from the tape (global memory);

- In practice, stripmining a loop twice does not increase runtime by $2\times$ (think more like $1.1\times$), and we have also observed speedups (due to caching).

Scatter or Parallel Write Operator

scatter : *[m]α $\rightarrow$ [n]int $\rightarrow$ [n]α $\rightarrow$ *[m]α

A (data vector) =[b0, b1, b2, b3]
I (index vector) =[2, 4, 1, -1]
X (input array) =[a0, a1, a2, a3, a4, a5]
scatter X I A =[a0, b2, b0, a3, b1, a5]

Scatter or Parallel Write Operator

scatter : *[m]$\alpha \to$ [n]int $\to$ [n]$\alpha \to$ *[m]α

A (data vector) =[b0, b1, b2, b3]
I (index vector) =[2, 4, 1, -1]
X (input array) =[a0, a1, a2, a3, a4, a5]
scatter X I A =[a0, b2, b0, a3, b1, a5]

scatter has $D(n) = \Theta(1)$ and $W(n) = \Theta(n)$,

1. X is consumed; following uses of X or aliases are illegal;
2. Writes to out-of-bounds indices are ignored;
2. Assume scatter semantics disallows idempotent writes.

For convenience lets also define gather:

```
let 't [n] gather (xs: []t) (is: [n]i64) : [n]t =
  map (\ind -> xs[ind]) is
```

Rev-AD Rewrite Rule for Scatter

Original:

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

Primal trace:

```
-- let is = remove_duplicates is
xs_save = gather xs is      -- checkpointing
ys = scatter xs is vs
```

Original Semantics: $\forall i : \; ys[\, is[i] \,] = vs[i]$
By AD re-write rule: $\forall i : \; \overline{vs}[i] \mathrel{+}= \overline{ys}[\, is[i] \,]$

Reverse trace: we have the final $\overline{ys}$ and a partial $\overline{vs}$:

```
vs_bar +=  gather is ys_bar
xs_bar =  scatter ys_bar is 0
xs = scatter ys is xs_save    -- restoring xs
```

$\overline{xs}$ created there since `xs` could not have been read after.

All operations are proportional with the size of the updates!

Differentiating an Arbitrary Reduce Operation

$\odot : \alpha \to \alpha \to \alpha$ an arbitrary associative operator. Recall:

$\texttt{y = reduce } \odot \texttt{ e}_{\odot} \texttt{ } [a_0, a_1, \ldots, a_{n-1}] \texttt{ = } a_0 \odot a_1 \odot \ldots \odot a_{n-1}$

Let's group it for some i:

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

Denoting by $l_i = a_0 \odot \ldots \odot a_{i-1}$ and $r_i = a_{i+1} \odot \ldots \odot a_{n-1}$

$$y = l_i \odot a_i \odot r_i$$

Differentiating an Arbitrary Reduce Operation

$\odot : \alpha \to \alpha \to \alpha$ an arbitrary associative operator. Recall:

$$\texttt{y = reduce } \odot \texttt{ e}_{\odot} \texttt{ } [a_0, a_1, \ldots, a_{n-1}] \texttt{ = } a_0 \odot a_1 \odot \ldots \odot a_{n-1}$$

Let's group it for some i:

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

Denoting by $l_i = a_0 \odot \ldots \odot a_{i-1}$ and $r_i = a_{i+1} \odot \ldots \odot a_{n-1}$

$$y = l_i \odot a_i \odot r_i$$

By AD re-write rule:

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

Denote by f the recursively computed derivative

$$f(l_i, x, r_i) = \frac{\partial(l_i \odot x \odot r_i)}{\partial x}$$

If we could compute all $ls = [l_0, \ldots, l_{n-1}]$ and $rs = [r_0, \ldots, r_{n-1}]$:

$$\overline{a_i} \overline{+}= \textbf{map } f \ (\textbf{zip3 } ls \ as \ rs)$$

Differentiating an Arbitrary Reduce Operation

Original

```
y = reduce ⊙ e⊙ as
```

Primal trace:

```
y = reduce ⊙ e⊙ as
```

Denoting by $l_i = a_0 \odot \ldots \odot a_{i-1}$ and $r_i = a_{i+1} \odot \ldots \odot a_{n-1}$,
If we could compute all $ls = [l_0, \ldots, l_{n-1}]$ and $rs = [r_0, \ldots, r_{n-1}]$:

$$\overline{a_i} \ \overline{+} = \ \mathbf{map}\ f\ (\mathbf{zip3}\ ls\ as\ rs)$$

Reverse trace:

```
ls = scan^exc ⊙ e⊙ as
rs = reverse <| scan^exc ⊙' e⊙ (reverse as)
ā_i +̄= map f (zip3 ls as rs)
where x ⊙' y = y ⊙ x
```

Essentially a reduce is implemented with two scans and a map!

Special Cases of Reduce & Multi-Reduce

$+$ `let y=reduce (+) 0 as` $\Rightarrow$ `let` $\overline{as}$ `= map (+`$\overline{y}$`)` $\overline{as}$

$*$ primal modified to compute the product of non-zero elements *and* the number of zero elements;

min/max primal modified to compute `argmin/max`

Reduce by index, a.k.a. multi reduce or generalized histograms, follows the same rationale as for reduce, but applied to each bin.

Differentiating Arbitrary Scans

$\odot : \alpha \to \alpha \to \alpha$ an associative operator (for now on scalars)
Input array $as = [a_0, a_1, \ldots, a_{n-1}]$.

$\texttt{ys = scan } \odot \texttt{ e}_{\odot} \texttt{ as} = [a_0, a_0 \odot a_1, \ldots, a_0 \odot \ldots \odot a_{n-1}]$

So if we have $\overline{ys}$, what does it contribute to some element a_i?

Differentiating Arbitrary Scans

$\odot : \alpha \to \alpha \to \alpha$ an associative operator (for now on scalars)
Input array $as = [a_0, a_1, \ldots, a_{n-1}]$.

$$\texttt{ys = scan } \odot \texttt{ e}_{\odot} \texttt{ as = } [a_0, a_0 \odot a_1, \ldots, a_0 \odot \ldots \odot a_{n-1}]$$

So if we have $\overline{ys}$, what does it contribute to some element a_i?

Actually, it is easier to reason imperatively (writing scan as a loop):

```
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
```

Differentiating the loop implementation using human knowledge:

Differentiating Arbitrary Scans

$\odot : \alpha \to \alpha \to \alpha$ an associative operator (for now on scalars)
Input array $as = [a_0, a_1, \ldots, a_{n-1}]$.

$$\texttt{ys = scan } \odot \texttt{ e}_\odot \texttt{ as} = [a_0, a_0 \odot a_1, \ldots, a_0 \odot \ldots \odot a_{n-1}]$$

So if we have $\overline{ys}$, what does it contribute to some element a_i?

Actually, it is easier to reason imperatively (writing scan as a loop):

```
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
```

Differentiating the loop implementation using human knowledge:

$$\begin{aligned}
&\textbf{let } (\overline{rs'}, \overline{as}) = \textbf{loop } (\overline{rs}, \overline{as}) = (\text{copy } \overline{ys}, \overline{as}) \textbf{ for } i = n-1..1 \textbf{ do}\\
&\quad \textbf{let } \overline{rs}[i-1] \mathrel{+}= \frac{\partial\,(rs[i-1] \odot as[i])}{\partial\, rs[i-1]} * \overline{rs}[i]\\
&\quad \textbf{let } \overline{as}[i] \mathrel{+}= \frac{\partial\,(rs[i-1] \odot as[i])}{\partial\, as[i]} * \overline{rs}[i] \qquad \textbf{in } (\overline{rs}, \overline{as})\\
&\textbf{let } \overline{as}[0] \mathrel{+}= \overline{rs'}[0]
\end{aligned}$$

Simple dependence analysis proves that loop distribution is safe.

Differentiating Arbitrary Scans

ys = **scan** $\odot$ $e_{\odot}$ as = $[a_0, a_0 \odot a_1, \ldots, a_0 \odot \ldots \odot a_{n-1}]$

It is safe to distributed the outer loop:

let $(\overline{rs'})$ = **loop** $(\overline{rs})$ = (copy $\overline{ys}$) **for** i = n-1...1 **do**
let $\overline{rs}$[i-1] += $\frac{\partial\ (rs[i-1]\ \odot\ as[i])}{\partial\ rs[i-1]}$ $*$ $\overline{rs}[i]$ **in** $\overline{rs}$

The loop above is the tricky one.

let $(\overline{as})$ = **loop** $(\overline{as})$ = $(\overline{as})$ **for** i = n-1...0 **do**
let el = **if** i==0 **then** $\overline{rs'}$[0]
else $\frac{\partial\ (rs[i-1]\ \odot\ as[i])}{\partial\ as[i]}$ $*$ $\overline{rs'}[i]$
let $\overline{as}$[i] += el **in** $\overline{as}$

This loop is a map essentially.

Differentiating Arbitrary Scans

ys = **scan** $\odot$ $e_\odot$ as = $[a_0, a_0 \odot a_1, \ldots, a_0 \odot \ldots \odot a_{n-1}]$

Let us zoom in the difficult recurrence:

let $(\overline{rs'})$ = **loop** $(\overline{rs})$ = (copy $\overline{ys}$) **for** i = n-1...1 **do**
let $\overline{rs}$[i-1] += $\frac{\partial\ (rs[i-1]\ \odot\ as[i])}{\partial\ rs[i-1]}$ $*$ $\overline{rs}[i]$ **in** $\overline{rs}$

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:

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

(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
[Blelloch, "Prefix sums and their applications"]

Differentiating Arbitrary Scans

Original and Primal Trace:

$$\texttt{ys = scan } \odot \texttt{ e}_{\odot} \texttt{ as} = [a_0, a_0 \odot a_1, \ldots, a_0 \odot \ldots \odot a_{n-1}]$$

Reverse trace:

```
let (ds, cs) = map ( \i -> if (i == n-1) then (0, 1)
                           else (ys_bar[i], ∂ (rs[i] ⊙ as[i+1]) / ∂ rs[i])
                   ) [0..n-1]

let rs_bar = scan lin_o (0,1) (reverse ds) (reverse cs)
      |> map (\ d_i c_i -> d_i + c_i · ys_bar[n-1]) |> reverse

let as_bar +=_bar map (\ (i, a_i) -> if i==0 then rs_bar[0]
                                     else ∂ (rs[i-1] ⊙ a_i) / ∂ a_i · rs_bar[i]
                      ) [0..n-1] as

where lin_o (d1,c1) (d2,c2) = (d2 + c2 · d1, c2 × c1)
```

Generalization:

- $\times$ and $\cdot$ are matrix- matrix and vector multiplication, respectively, $+$ is vector addition;
- work-depth complexity preserved when $\odot$ operates on tuples, but not for arrays.

Differentiating Map Simple Case

Original program:

```
let xs = map f as
```

If the mapped function has no free variables then it is trivial to differentiate a map; the translation is simply another map:

```
let as' =
  map3(\ a a'0 x' ->
         let x = f a
         let a' = f' a x'
         in  a'0 + a'
       ) as as' xs'
```

Differentiating Map General Case

Consider a “crazy” example:

```
let ys = map3(\ k ind n ->
                loop (x) = (1.0) for i = 0..n-1 do
                  if    p1(k,ind,i) then x * a[ind+i]
                  elif  p2(k,ind,i) then x + b[2*ind+i]
                  elif  p3(k,ind,i) then x + sin c[3*ind-i]
                  else                   x * d[4*ind-2*i]
              ) keys inds counts
```

Map: writes once per iteration, but has no restrictions on what it reads!

Differentiating Map General Case

Consider a “crazy” example:

```
let ys = map3(\ k ind n ->
                loop (x) = (1.0) for i = 0..n-1 do
                  if    p1(k,ind,i) then x * a[ind+i]
                  elif  p2(k,ind,i) then x + b[2*ind+i]
                  elif  p3(k,ind,i) then x + sin c[3*ind-i]
                  else                   x * d[4*ind-2*i]
              ) keys inds counts
```

Map: writes once per iteration, but has no restrictions on what it reads!

The adjoint code would have to update each of the elements read by the map. This cannot be represented as a map in general. In the example, we need to update a statically-unknown number of elements from statically-unknown arrays.

Differentiating Map General Case

The adjoint of the map can be written as a loop:

```
for q = 0..m-1 do
  x = 1; k = keys[q]; ind = inds[q]; n = counts[q];
  float xs[n]; -- original code (redundant)
  for i = 0..n-1 do
    xs[i] = x;
    if   p1(k,ind,i) { x = x * a[ind+i]; }
    elif p2(k,ind,i) { x = x + b[2*ind+i]; }
    elif p3(k,ind,i) { x = x + sin(c[3*ind-i]); }
    else             { x = x * d[4*ind-2*i]; }
  -- adjoint code
  x̄ = ȳs[i];
  for i = n-1..0 do
    x = xs[i]
    if   p1(k,ind,i) { ā[ind+i] += x * x̄;  x̄ *= a[ind+i];  }
    elif p2(k,ind,i) { b̄[2*ind+i] += x̄; }
    elif p3(k,ind,i) { c̄[3*ind-i] += cos(c[3*ind-i]) * x̄; }
    else             { d̄[4*ind-2*i] += x * x̄; x̄ *= d[4*ind-2*i];  }
```

This is a generalized reduction, i.e., the outer loop can be executed in parallel if the += updates to arrays $\overline{a}, \overline{b}, \overline{c}, \overline{d}$ are executed attomically.

Generalized Reduction

If all the statements in which a variable x appears are of the form x[*ind*] $\oplus$= *exp*, where x does not appear in *exp* and *ind* and $\oplus$ is associative and commutative, then the cross-iteration RAWs on x can be resolved, for example by executing the update atomically.

A loop nest whose inter-iteration dependencies are all due to such reductions, is called a generalized reduction. For all/most purposes, generalized reductions can be treated as parallel loops.

- **The reverse-AD translation of `map` is a generalized reduction.**
- Implemented accumulators, which are created by the `with` construct and can be used inside `map` constructs.
- The properties of the generalized reduction are type-checked throughout the compiler.
- the extended map is a generalization of the classical map, but also of reduce-by-index.

Optimizing Generalized Reductions

Optimizing generalized reductions means translating them to more specialized constructs such as reductions or reduce-by-index.

Matrix Multiplication Original:

```
for i = 0..n-1
  for j = 0 ..n-1
    for k = 0 .. n-1
      c[i,j] += a[i,k] * b[k,j]
```

Matrix Multiplication Reverse-AD with accumulators:

```
for i = 0..n-1
  for j = 0 ..n-1
    for k = 0 .. n-1
      ā[i,k] += b[k,j] * c̄[i,j]
      b̄[k,j] += a[i,k] * c̄[i,j]
```

Rewrite the accumulations as classical reductions. This requires:

- to distribute the loop-nest over each statement
- bring innermost the parallel loop to which the write access is invariant to.

Optimizing Generalized Reductions

- Distribute the loop-nest over each statement
- Bring innermost the parallel loop to which the write access is invariant to.

```
for i = 0..n-1
  for k = 0 .. n-1
    for j = 0 ..n-1
      ā[i,k] += b[k,j] * c̄[i,j]
```

```
for k = 0 .. n-1
  for j = 0 ..n-1
    for i = 0..n-1
      b̄[k,j] += a[i,k] * c̄[i,j]
```

Perform a strength reduction: result can be summed and accumulated once

```
for i = 0..n-1
  for k = 0 .. n-1
    acc = 0
    for j = 0 ..n-1
      acc = acc + b[k,j] * c̄[i,j]
    ā[i,k] += acc
```

```
for k = 0 .. n-1
  for j = 0 ..n-1
    acc = 0
    for i = 0..n-1
      acc = acc + a[i,k] * c̄[i,j]
    b̄[k,j] += acc
```

Optimizing Generalized Reductions

```
for i = 0..n-1
  for k = 0 .. n-1
    acc = 0
    for j = 0 ..n-1
      acc = acc + b[k,j] * c[i,j]
    a[i,k] += acc
```

```
for k = 0 .. n-1
  for j = 0 ..n-1
    acc = 0
    for i = 0..n-1
      acc = acc + a[i,k] * c[i,j]
    b[k,j] += acc
```

Now re-write (most of) the accumulations as classical reductions:

```
map(\i ->
  map(\k ->
      let acc = map2 (*) b[k] c[i]
                |> reduce (+) 0
      let a[i,k] += acc in a
    ) (iota n)
  ) (iota n)
```

```
map(\ k ->
  map(\ j ->
      let acc = map2 (*) a[:,k] c[:,j]
                |> reduce (+) 0
      let b[k,j] += acc in b
    ) (iota n)
  ) (iota n)
```

In this form, the (enhanced) Futhark compiler would apply block and register tiling, and also implement the technique of parallelizing the innermost dimension proposed by [Rasch,Schulze, and Gorlatch, 2019]

Sequential Performance on AD-Bench

We report and evaluate an end-to-end implementation of reverse and forward AD inside the Futhark compiler.

- ADBench: set of benchmarks aimed at comparing AD tools;
- Futhark scores nicely in comparison with more mature frameworks, e.g., Tapenade [Araya-Polo and Hascoët, 2004].

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: displays **AD overhead**: the time to compute the full Jacobian relative to the time to compute the objective function. **Lower is better.**

For BA, Futhark is slower due to packing the result in the CSR format expected by the tooling, this code is not subject to AD.

Parallel GPU Performance vs Enzyme

We report and evaluate an end-to-end implementation of reverse and forward AD inside the Futhark compiler.

We benchmark on an NVIDIA **RTX 2080Ti** system, which features a GPU similar to the one used to report the Enzyme results [Moses et. al., 2021].

	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: Results for RSBench and XSBench

K-Means: Parallel GPU Performance vs PyTorch

k-means is an optimization problem where given n points P in $\mathbb{R}^d$ we must find k points C that minimize the cost function

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

Solving this with Newton's Method requires computing the Jacobian and Hessian of f.

We benchmark on an NVIDIA **RTX 2080Ti** and an **A100** system.
Futhark looses on home ground because accumulations are not yet optimized to a reduce-by-index construct.

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

GMM: GPU Performance vs PyTorch on RTX 2080Ti

We report and evaluate an end-to-end implementation of reverse and forward AD inside the Futhark compiler.

We test benchmark GMM from ADBench on 32-bit floats.
This is neutral ground.

	$\mathbf{D}_0$	$\mathbf{D}_1$	$\mathbf{D}_2$	$\mathbf{D}_3$	$\mathbf{D}_4$	$\mathbf{D}_5$
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

∅ means PyTorch run out of memory.
The Futhark Jacobian-runtime on $\mathbf{D}_5$ was 96.7*ms* on $\mathbf{D}_5$.

GMM: GPU Performance vs PyTorch on A100

We report and evaluate an end-to-end implementation of reverse and forward AD inside the Futhark compiler.

We test benchmark GMM from ADBench on 64-bit floats.
This is neutral ground, albeit GMM uses matrix multiplication.

	D_0	D_1	D_2	D_3	D_4	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

Futhark's matrix matrix multiplication performs very well on RTX 2080Ti, but poorly on A100; we are investigating this puzzling behavior.

LSTM: GPU Performance vs PyTorch

This is PyTorch's home ground, because matrix-multiplications take about 75% of runtime, and matrix-multiplication is a primitive in PyTorch, while Futhark needs to work hard for it.

			Speedups ($\times$)		**Overheads** ($\times$)		
		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

$\mathbf{D_0}$: $(bs, n, d, h) = (1024, 20, 300, 192)$
$\mathbf{D_1}$: $(bs, n, d, h) = (1024, 300, 80, 256)$.
cuDNN-based refers to the (manual) `torch.nn.LSTM` library.

Futhark is slower on A100 than RTX2080Ti; we are investigating this puzzling behavior.

Concluding Remarks

- We have presented a code transformation that implements reverse AD in a context of a functional, array language that supports nested parallelism;
- We have reported an end-to-end implementation in the Futhark compiler, and an experimental evaluation that is encouraging.
- **Our biggest problem right now is that the AD benchmarks that we have fail short of highlighting both the shortcomings and the strengths of our implementation. If you may help with that, we are all ears!**