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 2024, DPP Lecture Slides

Material for This Lecture

This lecture material corresponds to the SC'22 paper:

[1] "AD for an Array Language with Nested Parallelism", Robert Schenck, Ola Rønning, Troels Henriksen and Cosmin Oancea, SC, 2020 author copy pdf

[2] "Reverse-Mode AD of Multi-Reduce and Scan in Futhark", Lotte M. Bruun, Ulrik S. Larsen, Nikolaj H. Hinnerskov and Cosmin E. Oancea, IFL, 2023 author copy pdf

Inspiration sources:

[3] "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, pdf

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

- High-School Math Curiculum, which can possibly be 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:

- 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 $\nabla f(a) = \left[\frac{\partial f(a)}{\partial a_1}(a_1) \ldots \frac{\partial f(a)}{\partial a_n}(a_n)\right]$ is the gradient

Math: Chain Rule for Multi-Variate Functions (Jacobian)

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

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

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?

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.

However, if your implementation is not differentiable AD will compute the wrong value, even if the actual math function is, e.g.,

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.

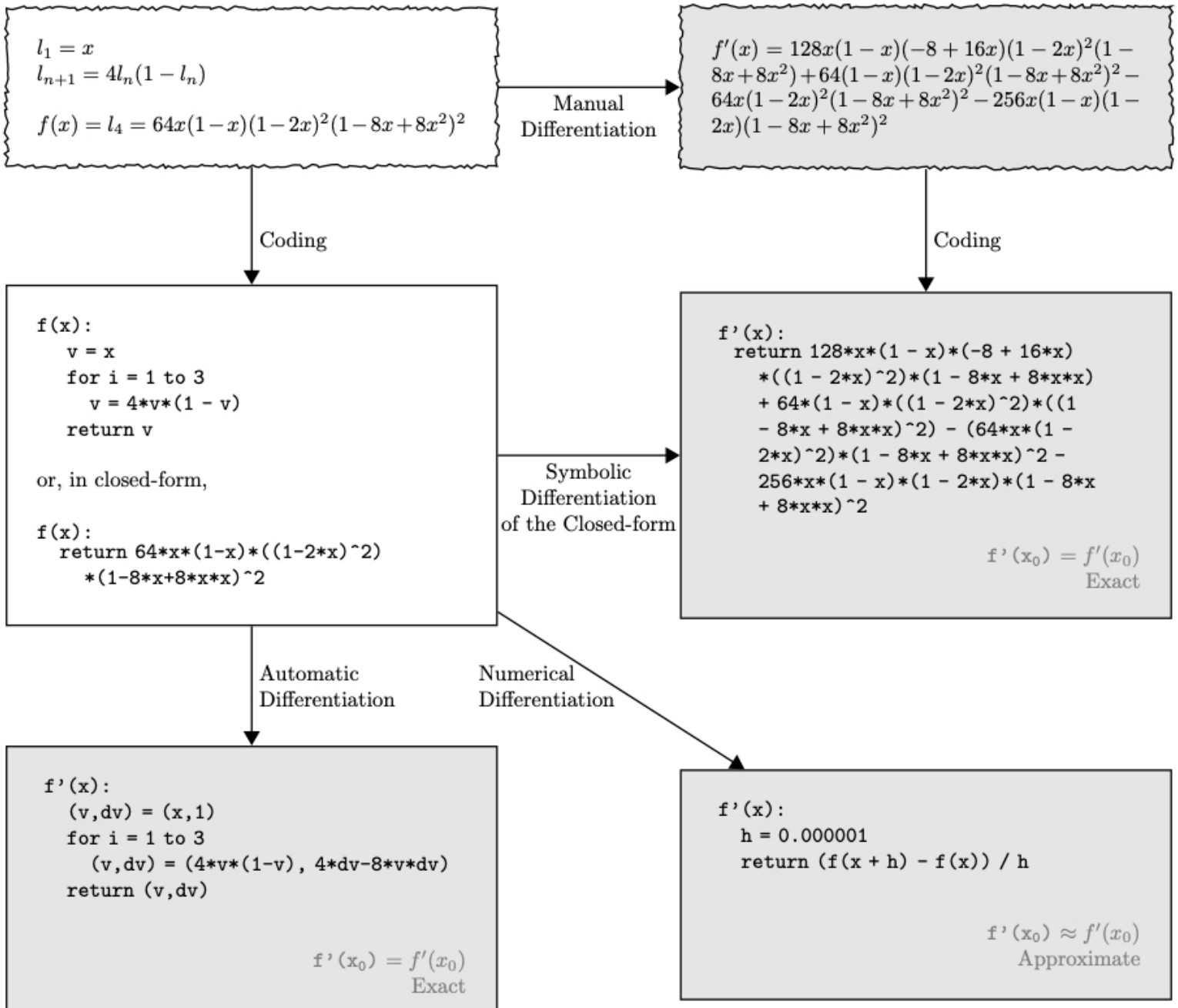
However, if your implementation is not differentiable AD will compute the wrong value, even if the actual math function is, e.g.,

$f\ x\ =\ 4\ \cdot\ x$ is differentiable, having its derivative at any point equal to: $f'\ x\ =\ 4$, but the implementation:

```
def f (x: f64) = if x == 1 then 4 else 4*x
```

will result in the derivative at point $x_0 = 1$ being: $f'_1\ x = 0$

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, denoted *tangent*:

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

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:
  $f(g(v + \dot{v}\,\epsilon)) = f(g(v)) + f'(g(v))\,g'(v)\,\dot{v}\,\epsilon$

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

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

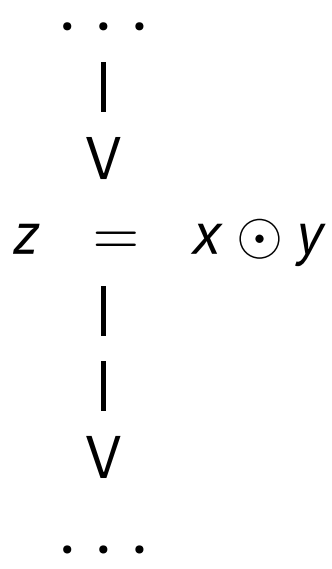
[Baydin, Pearlmutter, Radul, Siskind, "Automatic Differentiation in Machine Learning: a Survey", J. Mach. Learn. Res., 2017]

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 $\bar{v}_5 = \bar{y} = \frac{\partial y}{\partial y} = 1$.

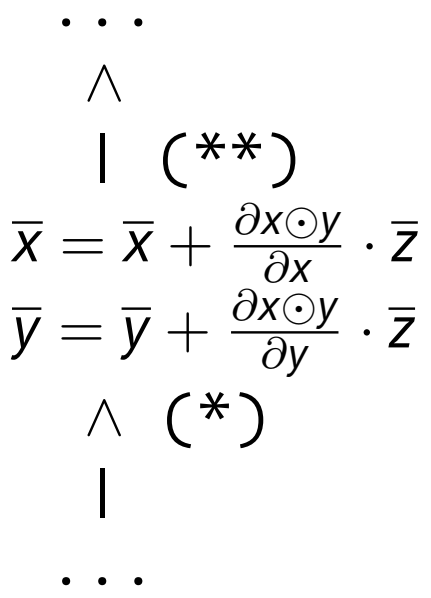
Forward Primal Trace			Reverse Adjoint (Derivative) Trace		
$v_{-1} = x_1$		$= 2$	$\bar{x}_1 = \bar{v}_{-1}$		$= \mathbf{5.5}$
$v_0 = x_2$		$= 5$	$\bar{x}_2 = \bar{v}_0$		$= \mathbf{1.716}$
$v_1 = \ln v_{-1}$		$= \ln 2$	$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$=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

Reverse AD: Core Rewrite Rule

Primal Trace:



Reverse Trace:



(*) 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 their first use.

Important: The rule performs one update of the adjoint of each distinct variable appearing in a statement, e.g., `let y = x*x` will be translated to one update of the adjoint of x: $\overline{x}$ `+= 2*x*`$\overline{y}$

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);
- **perfect scopes (except for loops) do not introduce re-execution overhead** (flattening-like procedure);
- "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).

Asymptotic Preserving & No Overhead for Perfect Nests

Original/Primal Code is a perfect nest of depth 4:

```
let xss = map (λ c as → if c
                        then ...
                        else map (λa → a*a) as
              ) cs ass
```

Asymptotic Preserving & No Overhead for Perfect Nests

Original/Primal Code is a perfect nest of depth 4:

```
let xss = map  (λ c as  →  if c
                           then  ...
                           else  map (λa  →  a*a) as
               ) cs ass
```

Differentiated Code displays in red the re-execution of the primal:

```
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
```

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 does not increase runtime by $2\times$ (e.g., $1.3\times$).

Classical API for AD

Notation: f the function subject to differentiation; x the point in which we apply differentiation, dx the tangent of the input; $\overline{y}$ the adjoint of the result.

Forward mode:

$$\text{jvp} : (f : \alpha \to \beta) \to (x : \alpha) \to (dx : \alpha) \to \beta$$

$$\text{jvp2} : (f : \alpha \to \beta) \to (x : \alpha) \to (dx : \alpha) \to (\beta, \beta)$$

`jvp2` returns the dual result, i.e., the result of the original trace tupled with its tangent

Reverse-mode:

$$\text{vjp} : (f : \alpha \to \beta) \to (x : \alpha) \to (\overline{y} : \beta) \to \alpha$$

$$\text{vjp2} : (f : \alpha \to \beta) \to (x : \alpha) \to (\overline{y} : \beta) \to (\beta, \alpha)$$

`vjp2` returns the result of the original trace tupled with the adjoint of the input

Differentiating Map: The Simple Case

$$x_i = f(a_i)$$

Applying the differentiation rule results in:

$$\overline{a_i} \quad \overline{+}= \quad \frac{\partial f(a_i)}{\partial a_i} \cdot \overline{x_i}$$

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

Original Program:

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

Reverse Trace:

```
let ās =
  map3(\ a ā⁰ x̄ ->
         let x = f a
         let ā = vjp f a x̄
         in  ā⁰ ⊹ ā
       ) as ās 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.**
- Troels 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 scatter, reduce and reduce-by-index.

Optimizing Generalized Reductions

Optimizing generalized reductions means translating them to more specialized constructs such as reduce 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]

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]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 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;
3. Assumed 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 (+`$\overline{y}$`)` $\overline{as}$
  Rationale: $y = as_0 + as_1 + \ldots + as_{n-1} \Rightarrow \overline{as_i}$ += $\overline{y}$

* 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.

Special Case: Min/Max

`let y = reduce_comm f32.max f32.lowest [`a_0`, ..., `a_{n-1}`]`

Denoting by i_{max} the index of the element that gives the result, i.e., $y = a_{i_{max}}$ then applying the differentiation rewrite rule yields:

$$\overline{a_i} \mathrel{+}= \begin{cases} \overline{y} & \text{if } i = i_{max} \\ 0 & \text{otherwise} \end{cases}$$

Special Case: Min/Max

`let y = reduce_comm f32.max f32.lowest [`a_0`, ..., `a_{n-1}`]`

Denoting by i_{max} the index of the element that gives the result, i.e., $y = a_{i_{max}}$ then applying the differentiation rewrite rule yields:

$$\overline{a_i} \mathrel{+}= \begin{cases} \overline{y} & \text{if } i = i_{max} \\ 0 & \text{otherwise} \end{cases}$$

Primal trace is extended to compute also, e.g., the smallest index at which the maximal element appears:

```
let (i_max, y) = zip (iota n) as
                 |> reduce_comm (max^L) (n, lowest)
```

Reverse trace:

```
if i_max >= n then as_bar
else let as_bar[i_max] += y_bar in as_bar
```

Special Case: Multiplication

$$y = a_0 \cdot a_1 \cdot \ldots \cdot a_{n-1} = a_i \cdot \prod_{k \neq i} a_k$$

Applying the reverse-AD rewrite:

$$\overline{a_i} \mathrel{+}= (\prod_{k \neq i} a_k) \cdot \overline{y}$$

Not entirely correct but good intuition:

$$\overline{a_i} \mathrel{+}= \frac{y}{a_i} \cdot \overline{y}$$

Special Case: Multiplication

$$y = a_0 \cdot a_1 \cdot \ldots \cdot a_{n-1} = a_i \cdot \prod_{k \neq i} a_k$$

Applying the reverse-AD rewrite:

$$\overline{a_i} \mathrel{+}= (\prod_{k \neq i} a_k) \cdot \overline{y}$$

Not entirely correct but good intuition:

$$\overline{a_i} \mathrel{+}= \frac{y}{a_i} \cdot \overline{y}$$

Solution: lift multiplication to compute the number of zeros nz and the product of non-zero elements $p^{\neq 0}$:

$$\overline{a_i} \mathrel{+}= \begin{cases} \frac{p^{\neq 0}}{a_i} \cdot \overline{y} & \text{if } nz = 0 \\ p^{\neq 0} \cdot \overline{y} & \text{if } a_i = 0 \text{ and } nz = 1 \\ 0 & \text{otherwise} \end{cases}$$

Generalization based on "Invertible Operators"

The case of multiplication hints to a more general treatment based on **commutative & invertible operators**, which comes down to specifying:

- conversions `cfwd` and `cbwd` between the datatype of the original and lifted op
- a lifted operator ($*^L$) together with its neutral element `ne`
- an inverse operator ($*^{inv}$) that is given the lifted reduction result and an original element *a* and returns the original-reduction result on all elements excluding *a*

```
def cfwd a = if a==0 then (1, 1.0) else (0, a)
def cbwd (nz, p^>0) = if nz == 0 then p^>0 else 0
def ne = (0, 1)
def *^L (nz_1, p_1^>0) (nz_2, p_2^>0) = (nz_1 + nz_2, p_1^>0 * p_2^>0)
def *^inv (nz, p^>0) a =
  if nz == 1 && a == 0 then p^>0
  else if nz == 0 then p^>0 / a  else 0
```

Primal trace:

```
let (ne, p^>0) = map cfwd as |> reduce_comm (*^L) ne
let y = cbwd (ne, p^>0)
```

Reverse trace:

```
let f a = let p' = *^inv (ne, p^>0) a in vjp (\ x -> x * p') a ȳ
let ās = map f as |> map2 (+) ās
```

Differentiating General-Case Reduce-By-Index

Reduce-by-index is re-written based on `hist` and `map`:

```
def reduce_by_index[n][w] (⊙ : α → α → α) (e⊙ : α)
                          (dst: [w]α) (ks: [n]i64) (vs: [n]α): [w]α =
  let hs = hist ⊙ e⊙ w ks vs
  let rs = map2 ⊙ dst hs in rs
```

Following a rationale similar to the one used for reduce, we have:

$$\overline{dst_i} \mathrel{\overline{+}}= \frac{\partial\,(dst_i \odot hs_i)}{\partial\, dst_i} \cdot \overline{rs_i} \tag{1}$$

$$\overline{hs_i} = \frac{\partial\,(dst_i \odot hs_i)}{\partial\, hs_i} \cdot \overline{rs_i} \tag{2}$$

$$\overline{v_i} \mathrel{\overline{+}}= \frac{\partial\,(l_i \odot v_i \odot r_i)}{\partial\, v_i} \cdot \overline{hs_{ks[i]}} \tag{3}$$

- v_i corresponds to `vs[i]`,
- l_i and r_i correspond to the forward and reverse scans (by $\odot$) of elements up to position i that have the same key as v_i ($ks[i]$)

Multi-scans needed to compute l_i and r_i: implemented by (radix) sorting the key-value pairs according to the keys, expensive!

Differentiating Red-by-Index with Invertible Operator

Using the same notation as for reduce and considering $\odot$ to be multiplication:

```
def cfwd a = if a==0 then (1, 1.0) else (0, a)
def cbwd (nz, p^>0) = if nz == 0 then p^>0 else 0
def ne = (0, 1)
def *^L (nz_1, p_1^>0) (nz_2, p_2^>0) = (nz_1 + nz_2, p_1^>0 * p_2^>0)
def *^inv (nz, p^>0) a = if nz == 1 && a == 0 then p^>0
                         else if nz == 0 then p^>0 / a  else 0
```

Primal trace:

```
let hs^L = map cfwd vs |> hist (*^L) ne w ks
let hs   = map cbwd hs^L
let rs   = map2 (*) dst hs
```

Reverse trace:

```
let g d h r̄ = vjp (\x -> x * d) h r̄
let hs̄    = map3 g dst hs rs̄
let dst̄ +̄= map3 g hs dst rs̄

let f k v = let p' = *^inv hs^L[k] v in vjp (\ x -> x * p') v hs̄[k]
let vs̄ +̄= map2 f ks vs
```

Differentiating Scan and reduce-by-index: Literature

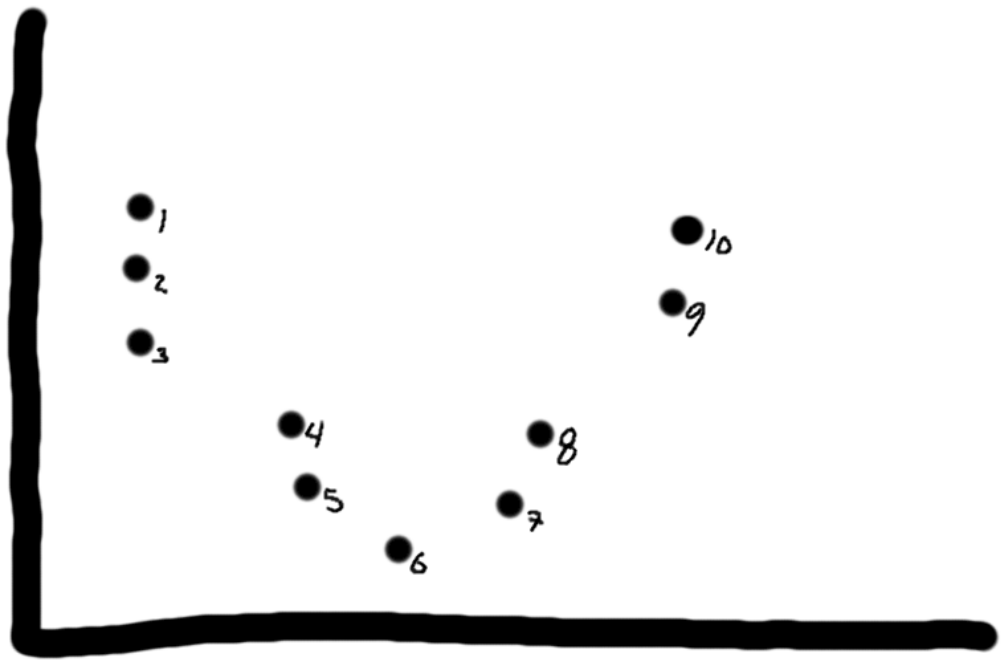
More material about the differentiation of reduce, reduce-by-index and scan:

[2] "Reverse-Mode AD of Multi-Reduce and Scan in Futhark", Lotte M. Bruun, Ulrik S. Larsen, Nikolaj H. Hinnerskov and Cosmin E. Oancea, IFL, 2023, author copy pdf

[4] A. Paszke, M. J. Johnson, R. Frostig, and D. Maclaurin, "Parallelism-Preserving Automatic Differentiation for Second-Order Array Languages", FHPNC, 2021, link to pdf

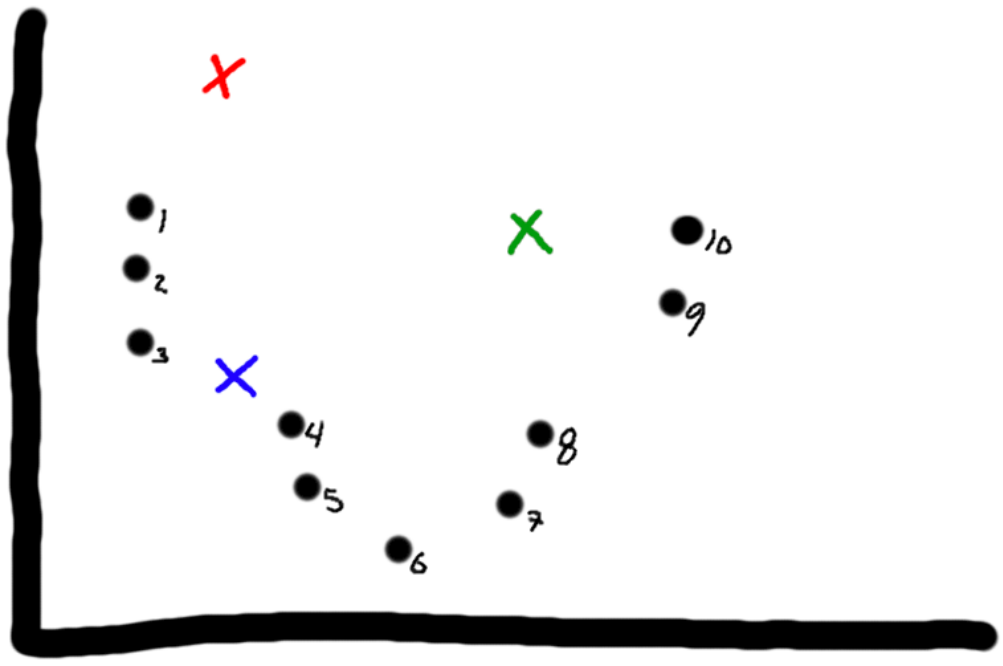
Illustrative Example: k-means clustering in 2D

Credit to Troels for pictures.



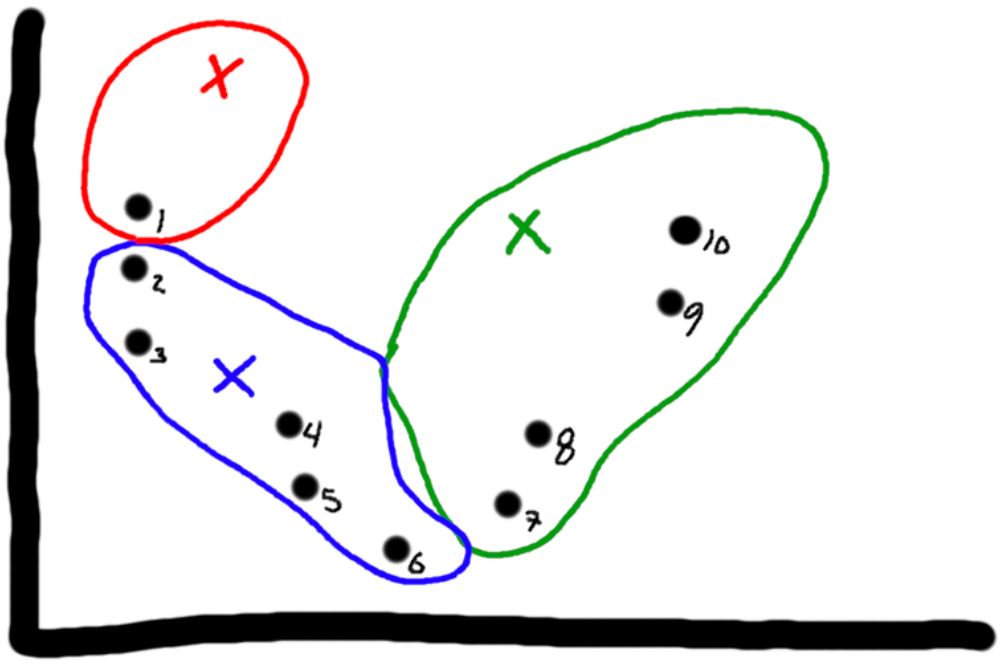
Illustrative Example: k-means clustering in 2D

Credit to Troels for pictures.



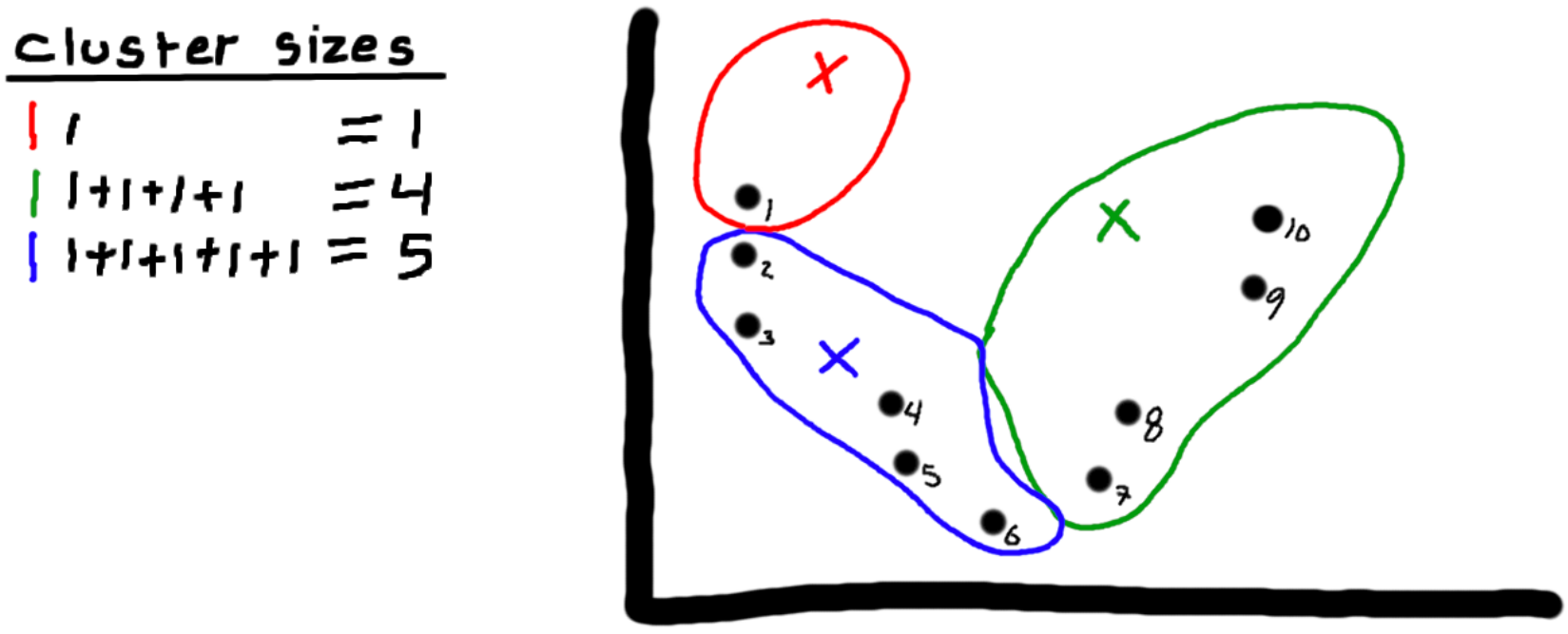
Illustrative Example: k-means clustering in 2D

Credit to Troels for pictures.



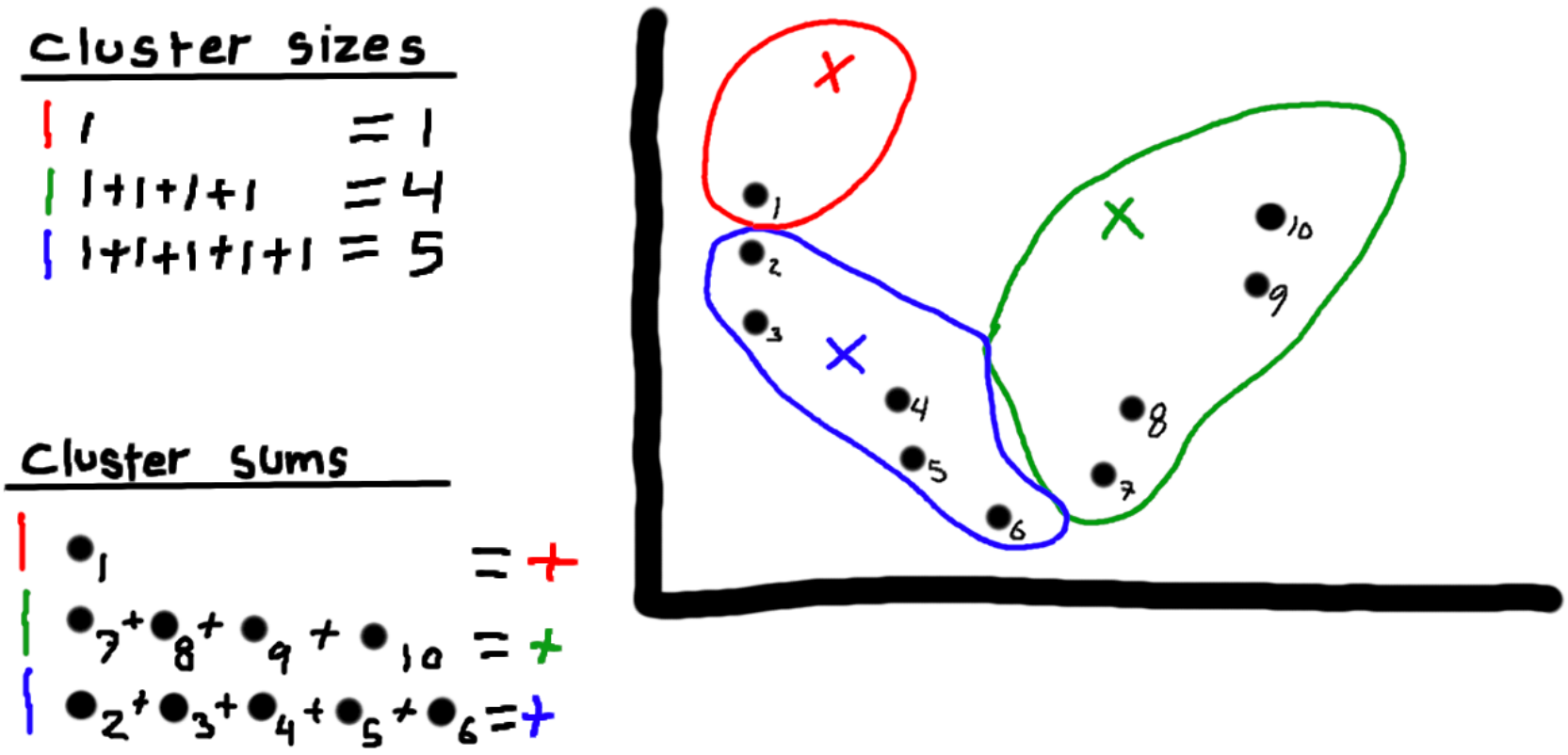
Illustrative Example: *k*-means clustering in 2D

Credit to Troels for pictures.



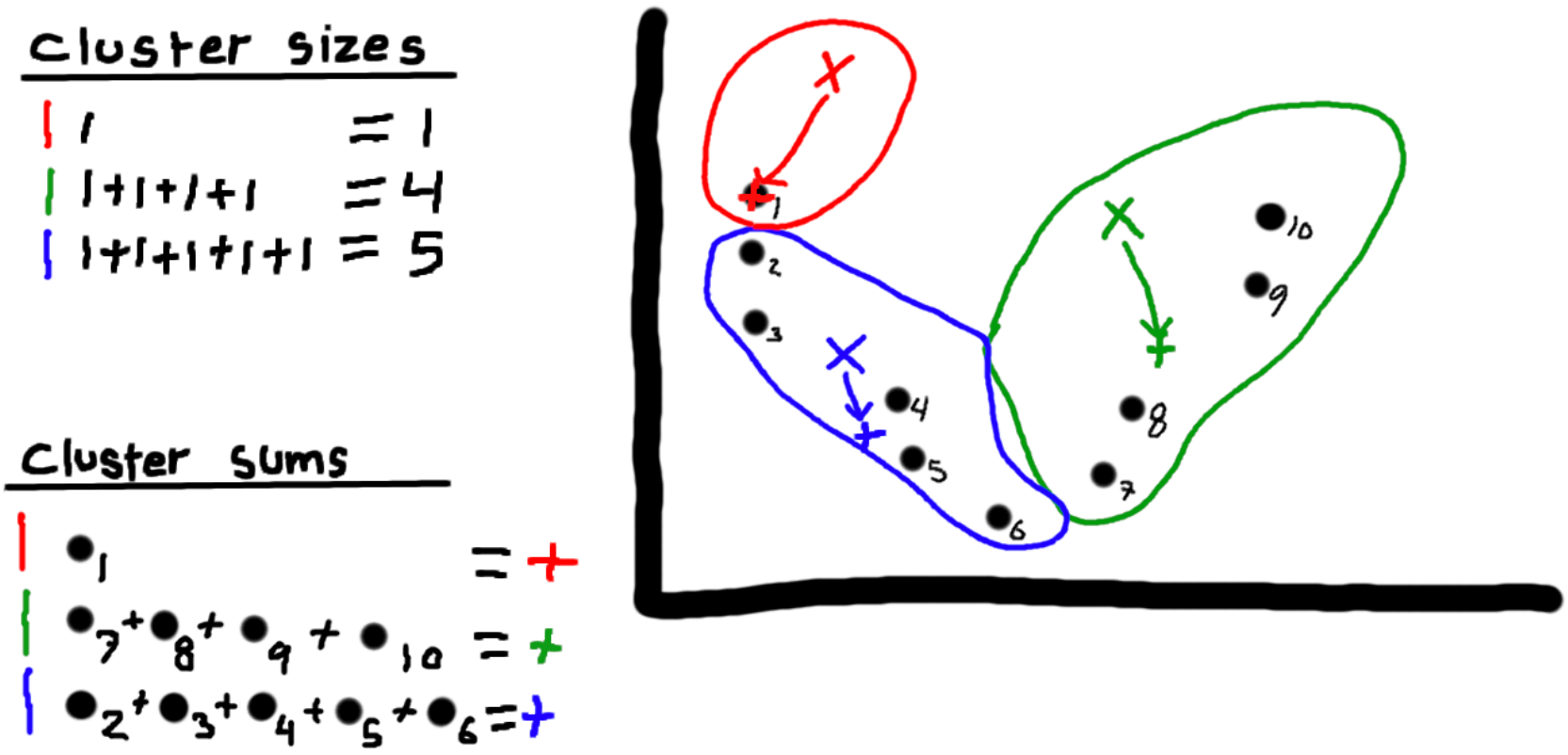
Illustrative Example: *k*-means clustering in 2D

Credit to Troels for pictures.



Illustrative Example: *k*-means clustering in 2D

Credit to Troels for pictures.



Illustrative Example: *k*-means clustering in 2D

Cluster sizes

1 = 1

1+1+1+1 = 4

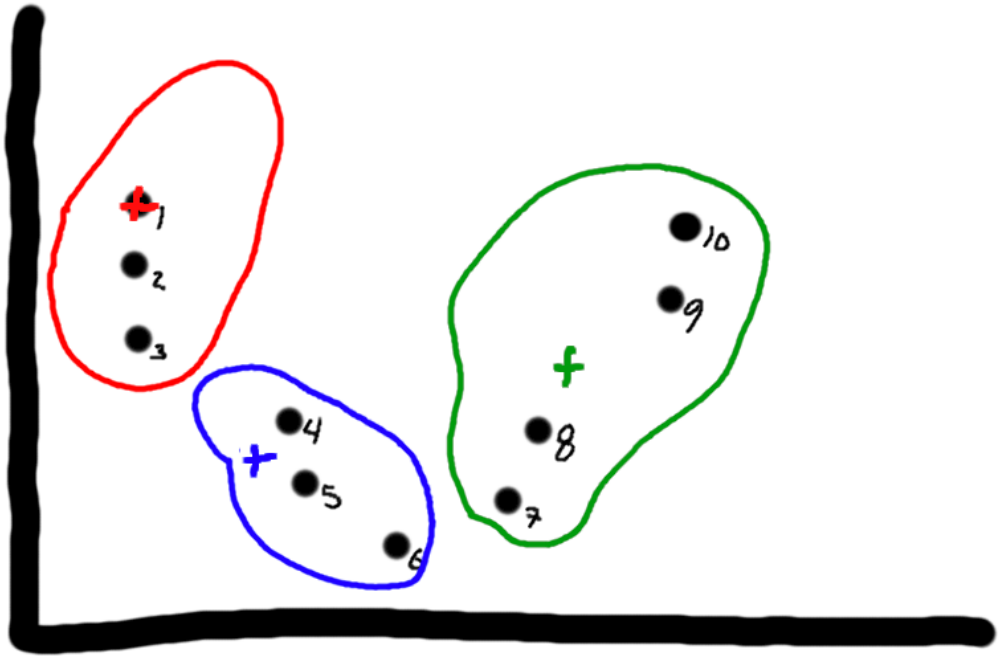
1+1+1+1+1 = 5

Cluster sums

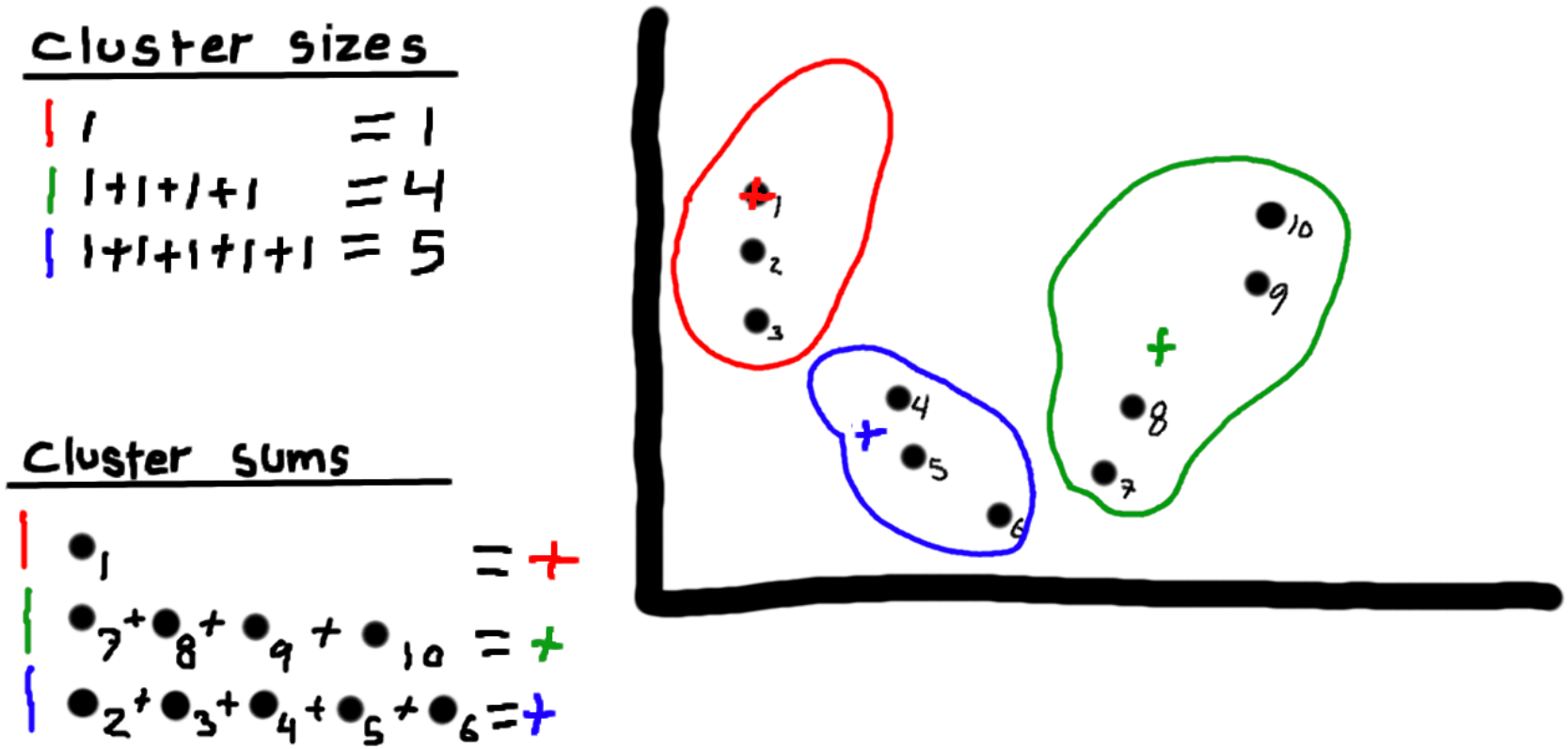
$\bullet_1 = +$

$\bullet_7 + \bullet_8 + \bullet_9 + \bullet_{10} = +$

$\bullet_2 + \bullet_3 + \bullet_4 + \bullet_5 + \bullet_6 = +$



Illustrative Example: *k*-means clustering in 2D



Genralized histograms allow a two-slide efficient implementation.

[2] Troels Henriksen, Sune Hellfritzsch, P. Sadayappan and Cosmin Oancea, "Compiling Generalized Histograms for GPU", In Procs of SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, 2020.

Mathematical Formulation of k-means clustering

Given a set of n points P in a d-dimensional space, one must find the k points C that minimize the cost function:

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

where

$$||p - c||^2 = \sum_{j=0}^{d-1} (p_j - c_j)^2$$

Mathematical Formulation of k-means clustering

Given a set of n points P in a d-dimensional space, one must find the k points C that minimize the cost function:

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

where

$$||p - c||^2 = \sum_{j=0}^{d-1} (p_j - c_j)^2$$

This problem can be solved by applying Newton's Method, i.e.,

$$C^{q+1} = C^q - \nabla f(C^q) \cdot H_f^{-1}(C^q)$$

where $\nabla f = f'$ is the Jacobian and $H_f = f''$ the Hessian of f.

∇f computed with reverse AD (`vjp`), since f has one result.

H_f is to be computed by differentiating ∇f with forward AD (`jvp2`), but in a way that takes advantage of sparsity (next).

The Hessian is a Diagonal Matrix

Given a set of n points P in a d-dimensional space, one must find the k points C that minimize the cost function:

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

For any fix p, the function g defined as:

$$g(c) = \sum_{j=0}^{d-1} (p_j - c_j)^2$$

has the second-order derivatives:

$$\frac{\partial^2 g}{\partial c_{j_1} \partial c_{j_2}} = \begin{cases} 2 & \text{if } j_1 = j_2 \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

It follows that the Hessian $H_f \in \mathcal{M}^{k \cdot d \times k \cdot d}$ is a diagonal matrix, in which the d entries of the i^{th} center are equal to $2 \cdot n_i$,
where n_i denotes the number of points clustered to that center.

Compressing the Hessian

Rather than computing each column of the Hessian by one application of forward AD (`jvp`) to the corresponding unit vector:

$$\begin{bmatrix} \mathbf{h_{1,1}} & 0 & \cdots & 0 \\ 0 & \mathbf{h_{2,2}} & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & \mathbf{h_{k \cdot d, k \cdot d}} \end{bmatrix} \times \begin{bmatrix} \mathbf{1} \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{h_{1,1}} \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

we could compute all Hessian's diagonal values by applying `jvp` once to the vector formed entirely by ones.

$$\begin{bmatrix} \mathbf{h_{1,1}} & 0 & \cdots & 0 \\ 0 & \mathbf{h_{2,2}} & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & \mathbf{h_{k \cdot d, k \cdot d}} \end{bmatrix} \times \begin{bmatrix} \mathbf{1} \\ \mathbf{1} \\ \vdots \\ \mathbf{1} \end{bmatrix} = \begin{bmatrix} \mathbf{h_{1,1}} \\ \mathbf{h_{2,2}} \\ \vdots \\ \mathbf{h_{k \cdot d, k \cdot d}} \end{bmatrix}$$

Inverting a Diagonal Matrix

Given a set of n points P in a d-dimensional space, one must find the k points C that minimize the cost function:

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

This problem can be solved by applying Newton's Method, i.e.,

$$C^{q+1} = C^q - \nabla f(C^q) \cdot H_f^{-1}(C^q)$$

Inverting a diagonal matrix, correspond to inverting each of its elements:

$$\begin{bmatrix} \mathbf{h_{1,1}} & 0 & \cdots & 0 \\ 0 & \mathbf{h_{2,2}} & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & \mathbf{h_{k \cdot d, k \cdot d}} \end{bmatrix}^{-1} = \begin{bmatrix} \frac{\mathbf{1}}{\mathbf{h_{1,1}}} & 0 & \cdots & 0 \\ 0 & \frac{1}{\mathbf{h_{2,2}}} & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & \frac{1}{\mathbf{h_{k \cdot d, k \cdot d}}} \end{bmatrix}$$

On Implementing the Newton Update Step

Given a set of n points P in a d-dimensional space, one must find the k points C that minimize the cost function:

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

This problem can be solved by applying Newton's Method, i.e.,

$$C^{q+1} = C^q - \nabla f(C^q) \cdot H_f^{-1}(C^q)$$

Denoting by f' and f'' the two-dimensional $k \times d$ arrays corresponding to the Jacobian vector and the compressed Hessian vector $[h_{1,1}, \ldots, h_{k \cdot d, k \cdot d}]$ then the update by Newton's method is:

$$C_{i,j}^{q+1} = C_{i,j}^q - \frac{f'_{i,j}}{f''_{i,j}}, \quad \forall\, 0 \leq i < k \text{ and } 0 \leq j < d$$

Classical API for AD

Notation: f the function subject to differentiation; x the point in which we apply differentiation, dx the tangent of the input; $\overline{y}$ the adjoint of the result.

Forward mode:

$$\text{jvp} : (f : \alpha \to \beta) \to (x : \alpha) \to (dx : \alpha) \to \beta$$

$$\text{jvp2} : (f : \alpha \to \beta) \to (x : \alpha) \to (dx : \alpha) \to (\beta, \beta)$$

`jvp2` returns the dual result, i.e., the result of the original trace tupled with its tangent

Reverse-mode:

$$\text{vjp} : (f : \alpha \to \beta) \to (x : \alpha) \to (\overline{y} : \beta) \to \alpha$$

$$\text{vjp2} : (f : \alpha \to \beta) \to (x : \alpha) \to (\overline{y} : \beta) \to (\beta, \alpha)$$

`vjp2` returns the result of the original trace tupled with the adjoint of the input

Implementing the Jacobian and Hessian Computation

Given a set of n points P in a d-dimensional space, one must find the k points C that minimize the cost function:

$$f(C) = \sum_{p \in P} \min \left\{ ||p - c||^2, c \in C \right\} \text{ where } ||p - c||^2 = \sum_{j=0}^{d-1} (p_j - c_j)^2$$

This problem can be solved by applying Newton's Method, i.e.,

$$C^{q+1} = C^q - \nabla f(C^q) \cdot H_f^{-1}(C^q)$$

where $\nabla f = f'$ is the Jacobian and $H_f = f''$ the Hessian of f.

Since P has to come from somewhere, let us use f `= cost` P:

```
def cost [n][k][d] (P: [n][d]f64) (C: [k][d]f64) = ...
```

- ∇f computed with reverse AD `vjp` (f has one result), i.e., it is something like `vjp (cost P) x 1` where `x` holds C
- H_f is computed by differentiating ∇f with forward AD (`jvp2`), so that both f' and f'' are returned. We should take advantage of the diagonal structure of the Hessian, and apply `jvp2` once to the initial tangent dx in which each element is one.

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].

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

Table: Results for RSBench, XSBench and LBM

K-Means: Parallel GPU Performance vs PyTorch & JAX

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.

	Data	Futhark (ms)		PyTorch	JAX	JAX(vmap)
		Manual	AD	(ms)	(ms)	(ms)
A100	D_0	12.6	41.1	41.1	15.5	27.5
	D_1	19.0	10.6	8.7	2.1	107.9
	D_2	94.3	108.9	922.0	206.5	976.4
MI100	D_0	24.6	35.6	94.5	–	–
	D_1	22.5	10.5	40.2	–	–
	D_2	309.5	264.2	2303.2	–	–

Sparse K-Means: GPU Performance vs PyTorch & JAX

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.

		Futhark (s)		**PyTorch** (s)	**JAX** (s)
	Workload	**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	–

GMM: GPU Performance vs PyTorch on A100 & MI100

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.

	Measurement	D_0	D_1	D_2	D_3	D_4	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

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			
		PyTorch Jacob.	**Futhark**	**nn.LSTM**	**JAX**	**JAX(vmap)**
A100	D_0	45.4 ms	3.0	11.6	4.5	0.3
	D_1	740.1 ms	3.3	22.1	6.4	0.9
MI100	D_0	89.8 ms	2.6	4.0	–	–
	D_1	1446.9 ms	1.8	5.4	–	–

		Overheads				
		PyTorch	**Futhark**	**nn.LSTM**	**JAX**	**JAX(vmap)**
A100	D_0	4.1	2.1	2.7	3.5	1.4
	D_1	4.3	3.9	2.2	3.7	0.8
MI100	D_0	5.0	4.2	7.2	–	–
	D_1	7.9	3.9	6.6	–	–

cuDNN-based refers to the (manual) `torch.nn.LSTM` library.

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!**