AD for an Array Language Supporting Nested Parallelism

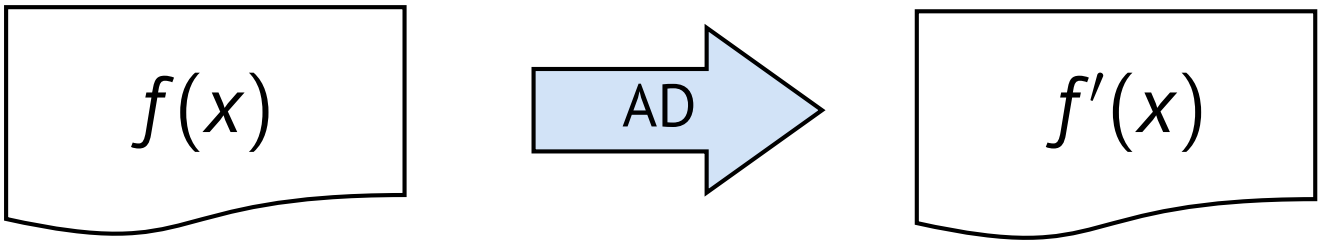
Cosmin E. Oancea (cosmin.oancea@diku.dk)
in collaboration with
Troels Henriksen, Ola Rønning and Robert Schenck

Department of Computer Science (DIKU)
University of Copenhagen

7th of June 2023, MIAPbP Workshop

Overview

- **Automatic differentiation (AD)** is a program transformation for differentiation.



- This talk presents an AD technique [1] for a functional, **high-level**, and **nested-parallel** array language (Futhark) .
- All parallelism is made explicit via **parallel combinators**—map, reduce, scan (prefix sum), scatter, etc.

[1] Robert Schenck, Ola Rønning, Troels Henriksen and Cosmin Oancea, ”AD for an Array Language with Nested Parallelism”, In Procs of SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, 2022.

Motivating Example for AD

Gentle Introduction to AD

Key Idea #1: Redundant Execution Instead of Tape

Key Idea #2: Parallel Constructs are Differentiated at a High Level

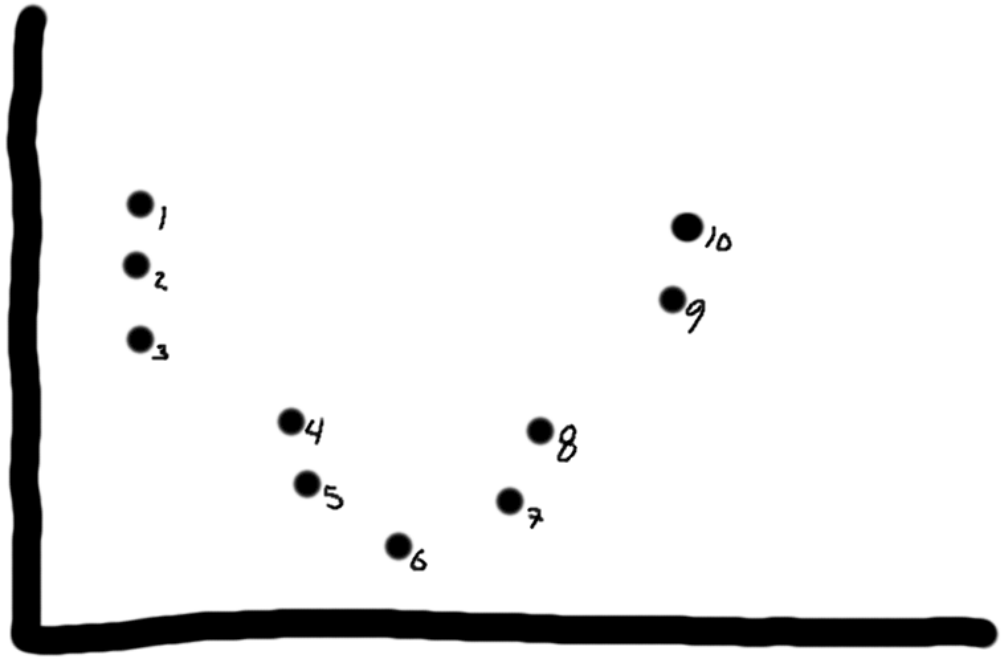
Key Idea #3: Translate Accumulators to Specialized Constructs

Experimental Results: Competitive with State of the Art

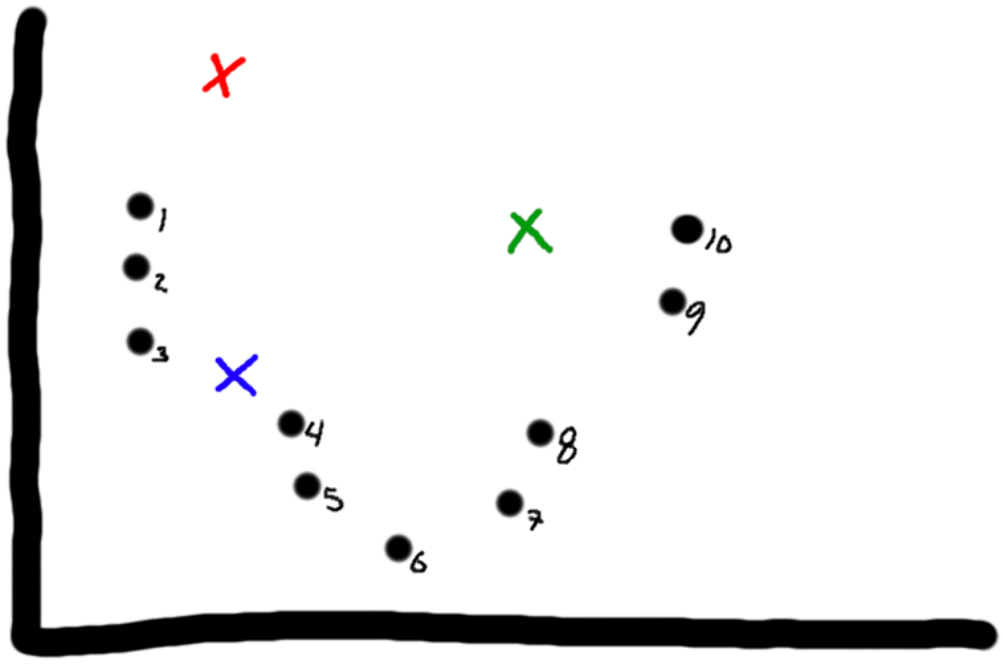
Demo

Extra Slides

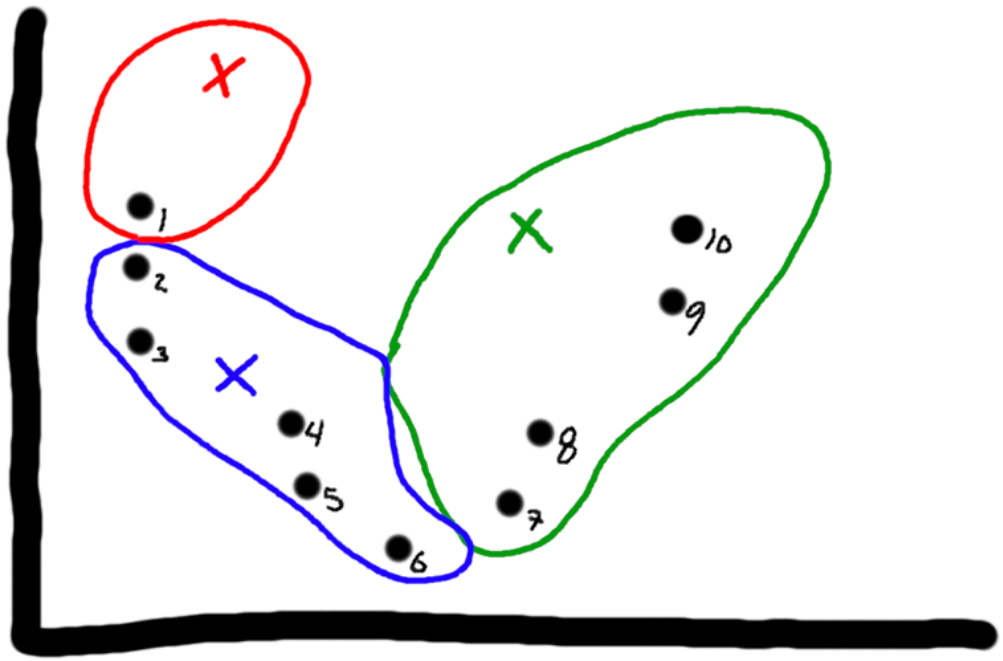
Illustrative Example: k-means clustering in 2D



Illustrative Example: k-means clustering in 2D



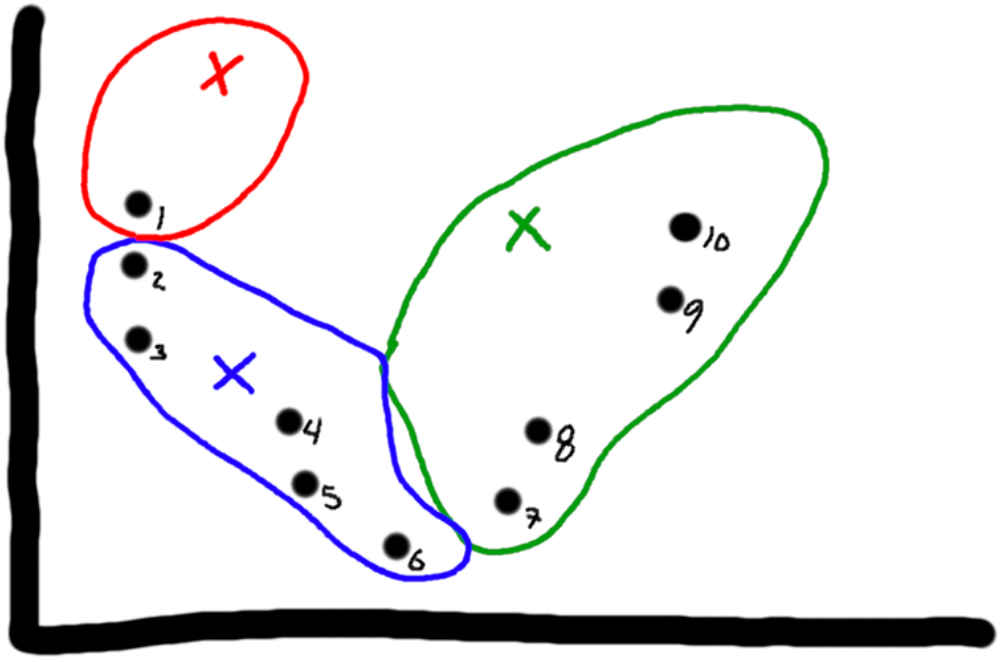
Illustrative Example: k-means clustering in 2D



Illustrative Example: k-means clustering in 2D

Cluster sizes

1	= 1
1+1+1+1	= 4
1+1+1+1+1	= 5



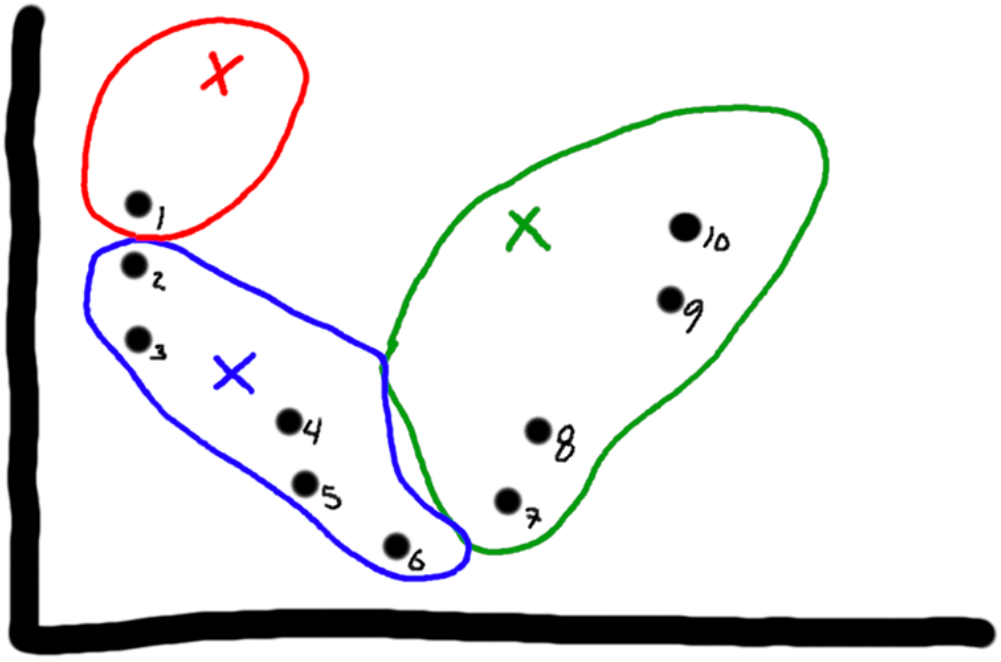
Illustrative Example: k-means clustering in 2D

Cluster sizes

1	= 1
1+1+1+1	= 4
1+1+1+1+1	= 5

Cluster sums

$\bullet_1$	= +
$\bullet_7 + \bullet_8 + \bullet_9 + \bullet_{10}$	= +
$\bullet_2 + \bullet_3 + \bullet_4 + \bullet_5 + \bullet_6$	= +



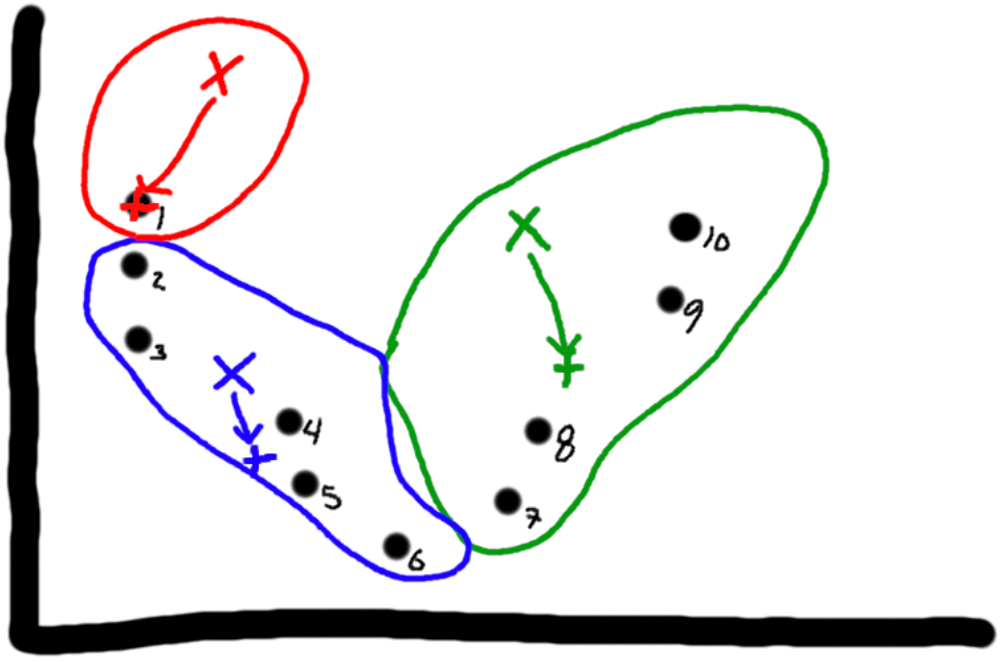
Illustrative Example: k-means clustering in 2D

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

Cluster sizes

1 = 1

1+1+1+1 = 4

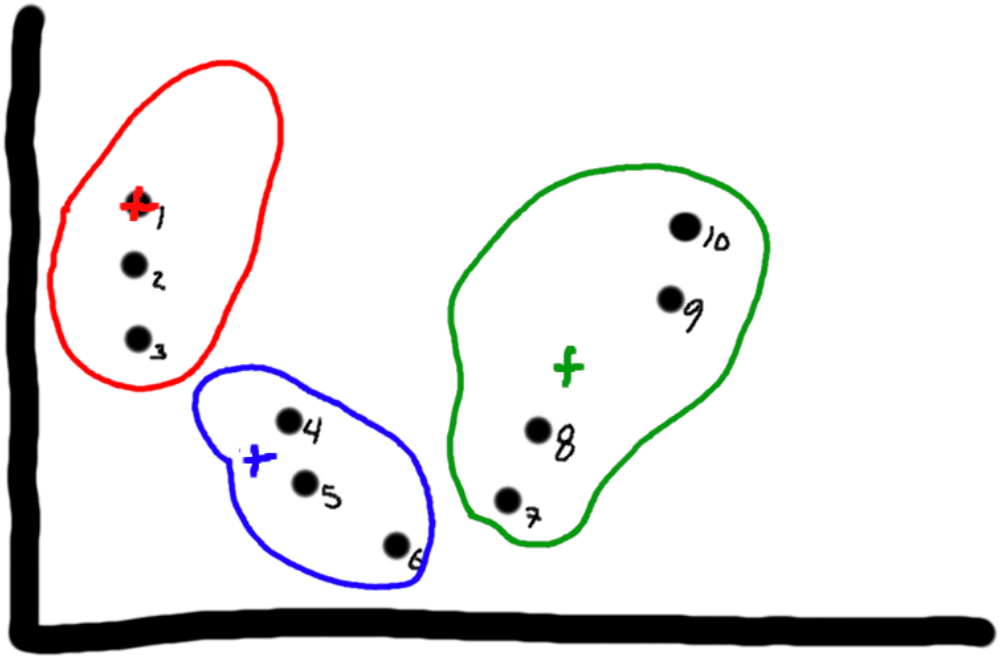
1+1+1+1+1 = 5

Cluster sums

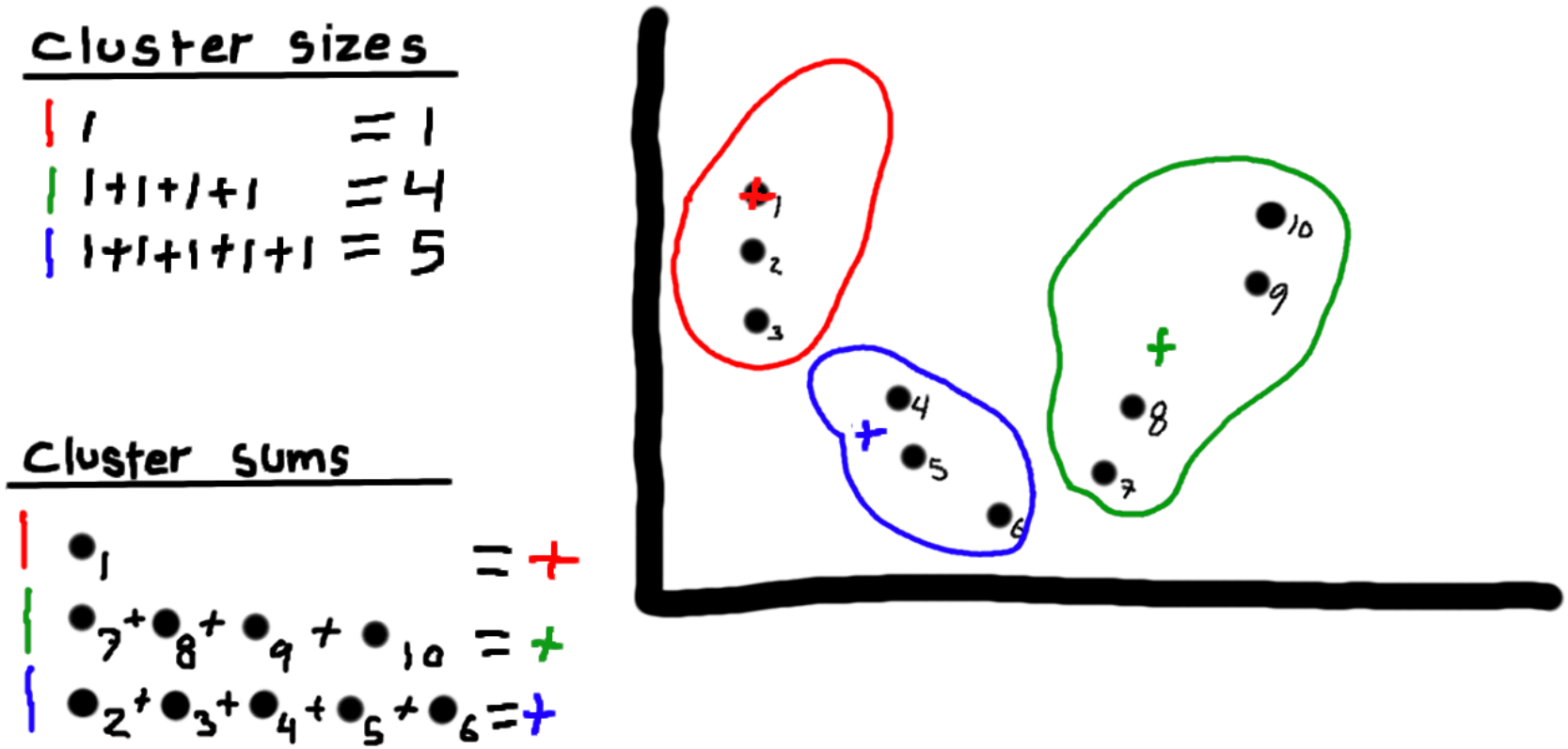
$\bullet_1 = +$

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

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



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:

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

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:

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

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

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

Motivation for AD

- Initially: because we heard that AD is a challenging code transformation, and we are always up to a good challenge.

- Because AD enables simple and generic optimization recipes
  - Gradient descent and Newton's Method
  - In Probabilistic ML:
    - Hamiltonian MC,
    - Newtonian MC,
    - variational inference,
    - …

AD: worthy to be supported as a first-class citizen of parallel languages?

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.

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

...

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

Power: If $h(x) = f(x)^r$ then $\frac{\partial h}{\partial x} = n \cdot f^{r-1} \cdot \frac{\partial f}{\partial x}$

...

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.

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

...

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

Power: If $h(x) = f(x)^r$ then $\frac{\partial h}{\partial x} = n \cdot f^{r-1} \cdot \frac{\partial f}{\partial x}$

...

The Chain Rule (the rule to bind them all):

If $h(x) = (f \circ g)(x) = f(g(x))$ then

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

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:

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

Math Intuition for Differentiating a Program

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:

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

Forward *vs* Reverse-Mode AD answers the question:
Should Jacobians be multiplied from right-to-left or left-to-right?

Intuition: Reverse-Mode AD

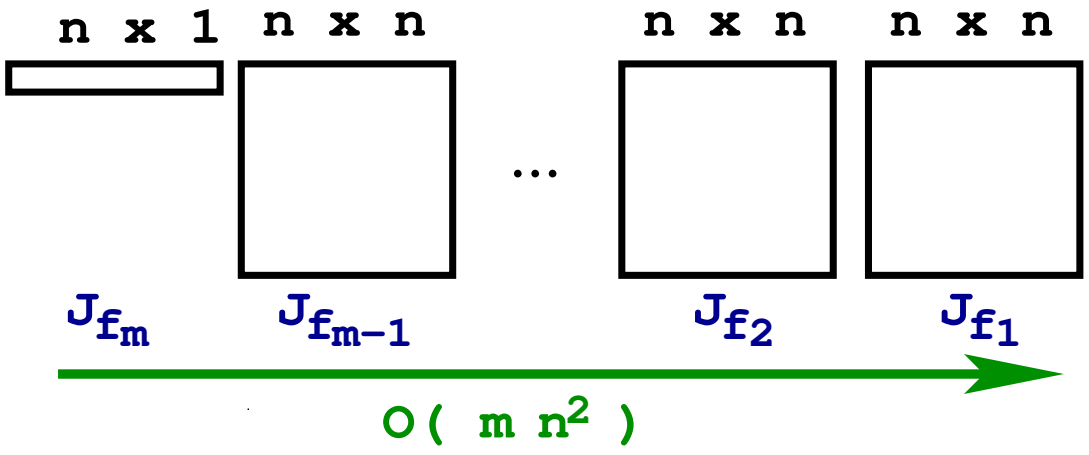
Consider $f_i : \mathbb{R}^{n_{i-1}} \rightarrow \mathbb{R}^{n_i}, \forall i = 1 \ldots m$, and "program" P defined as:

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

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

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

Reverse-Mode AD multiplies Jacobians from left to right.



Intuition: Reverse-Mode AD

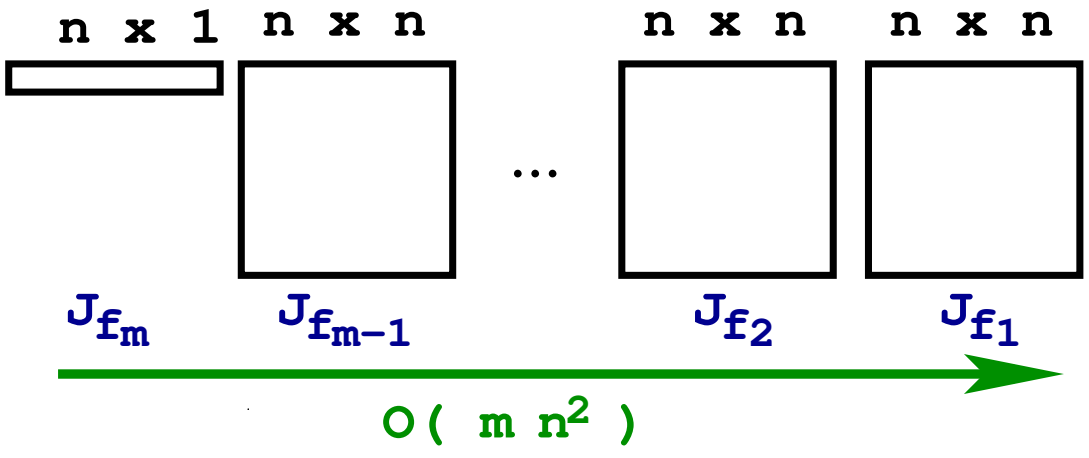
Consider $f_i : \mathbb{R}^{n_{i-1}} \rightarrow \mathbb{R}^{n_i}, \forall i = 1 \ldots m$, and "program" P defined as:

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

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

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

Reverse-Mode AD multiplies Jacobians from left to right.



- Prefered when the input size is larger than the size of the result (ML case);
- The *primal trace* (original program) is first executed to save all intermediate program values on a *tape*. The tape is subsequently used by the *return sweep*, which computes the *adjoint* of each variable in reverse program order.

Reverse AD: Core Re-Write Rule for Scalar Code

For program $P(\ldots x_i \ldots) = y \in \mathbb{R}$, reverse AD computes the **adjoint** of each (intermediate) program variable t, denoted $\bar{t} = \frac{\partial y}{\partial t}$, i.e., the sensitivity of the output to changes in t.

Reverse AD: Core Re-Write Rule for Scalar Code

For program $P(\ldots x_i \ldots) = y \in \mathbb{R}$, reverse AD computes the **adjoint** of each (intermediate) program variable t, denoted $\bar{t} = \frac{\partial y}{\partial t}$, i.e., the sensitivity of the output to changes in t.

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

Primal Trace:

$$\begin{array}{c} \cdots \\ \downarrow \\ z = x \odot y \\ \downarrow \\ \cdots \end{array}$$

Return Sweep:

$$\begin{array}{c} \cdots \\ \wedge \\ \downarrow \quad (**) \\ \bar{x} = \bar{x} + \frac{\partial x \odot y}{\partial x} \cdot \bar{z} \\ \bar{y} = \bar{y} + \frac{\partial x \odot y}{\partial y} \cdot \bar{z} \\ \wedge \quad (*) \\ \downarrow \\ \cdots \end{array}$$

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

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

- adjoints are initialized (zeroed) before their first use.

Illustrating Reverse AD on straightline scalar code

Adjoint of var t is $\bar{t} \equiv \frac{\partial y}{\partial t}$. **Goal:** compute the adjoints of the input.

Core Rewrite Rule: $z = f(x, y) \Rightarrow$ $\bar{x} += \frac{\partial f(x,y)}{\partial x} \cdot \bar{z}$, $\bar{y} += \frac{\partial f(x,y)}{\partial x} \cdot \bar{z}$

$P(x_0,\ x_1)$:
$t_0 = \sin(x_0)$
$t_1 = x_1 \cdot t_0$
$y = x_0 + t_1$
return y

$\Longrightarrow$

$P'(x_0,\ x_1)$:
$t_0 = \sin(x_0)$
$t_1 = x_1 \cdot t_0$
$y = x_0 + t_1$

Illustrating Reverse AD on straightline scalar code

Adjoint of var t is $\bar{t} \equiv \frac{\partial y}{\partial t}$. **Goal:** compute the adjoints of the input.

Core Rewrite Rule: $y = f(x_1, x_2) \Rightarrow$

$$\overline{x_1} += \frac{\partial f(x_1,x_2)}{\partial x_1} \cdot \bar{z}$$
$$\overline{x_2} += \frac{\partial f(x_1,x_2)}{\partial x_2} \cdot \bar{z}$$

$P(x_0,\ x_1)$:
$t_0 = \sin(x_0)$
$t_1 = x_1 \cdot t_0$
$y = x_0 + t_1$
return y

$\Longrightarrow$

$P'(x_0,\ x_1)$:
$t_0 = \sin(x_0)$
$t_1 = x_1 \cdot t_0$
$y = x_0 + t_1$
$\bar{y} = 1; \quad \overline{x_0} = 0; \quad \overline{t_1} = 0$
$\overline{x_0} = \overline{x_0} + \frac{\partial(x_0+t_1)}{\partial x_0} \cdot \bar{y} = 1 \cdot 1 = 1$
$\overline{t_0} = \overline{t_0} + \frac{\partial(x_0+t_1)}{\partial t_1} \cdot \bar{y} = 1 \cdot 1 = 1$

Illustrating Reverse AD on straightline scalar code

Adjoint of var t is $\bar{t} \equiv \frac{\partial y}{\partial t}$. **Goal:** compute the adjoints of the input.

Core Rewrite Rule: $y = f(x_1, x_2) \Rightarrow$

$$\overline{x_1} += \frac{\partial f(x_1,x_2)}{\partial x_1} \cdot \bar{z}$$
$$\overline{x_2} += \frac{\partial f(x_1,x_2)}{\partial x_2} \cdot \bar{z}$$

$P(x_0,\ x_1)$:
$t_0 = \sin(x_0)$
$t_1 = x_1 \cdot t_0$
$y = x_0 + t_1$
return y

$\Longrightarrow$

$P'(x_0,\ x_1)$:
$t_0 = \sin(x_0)$
$t_1 = x_1 \cdot t_0$
$y = x_0 + t_1$
$\bar{y} = 1; \quad \overline{x_0} = 0; \quad \overline{t_1} = 0$
$\overline{x_0} = \overline{x_0} + \frac{\partial(x_0+t_1)}{\partial x_0} \cdot \bar{y} = 1 \cdot 1 = 1$
$\overline{t_1} = \overline{t_0} + \frac{\partial(x_0+t_1)}{\partial t_1} \cdot \bar{y} = 1 \cdot 1 = 1$
$\overline{x_1} = 0; \quad \overline{t_0} = 0$
$\overline{x_1} = \overline{x_1} + \frac{\partial(x_1 \cdot t_0)}{\partial x_1} \cdot \overline{t_1} = \overline{x_1} + t_0 \cdot \overline{t_1}$
$\overline{t_0} = \overline{t_0} + \frac{\partial(x_1 \cdot t_0)}{\partial t_0} \cdot \overline{t_1} = \overline{t_0} + x_1 \cdot \overline{t_1}$

Illustrating Reverse AD on straightline scalar code

Adjoint of var t is $\bar{t} \equiv \frac{\partial y}{\partial t}$. **Goal:** compute the adjoints of the input.

Core Rewrite Rule: $y = f(x_1, x_2) \Rightarrow$ $\overline{x_1} += \frac{\partial f(x_1,x_2)}{\partial x_1} \cdot \bar{z}$, $\overline{x_2} += \frac{\partial f(x_1,x_2)}{\partial x_2} \cdot \bar{z}$

$P(x_0,\ x_1)$:
$t_0 = \sin(x_0)$
$t_1 = x_1 \cdot t_0$
$y = x_0 + t_1$
return y

$\Longrightarrow$

$P'(x_0,\ x_1)$:
$t_0 = \sin(x_0)$
$t_1 = x_1 \cdot t_0$
$y = x_0 + t_1$
$\bar{y} = 1; \quad \overline{x_0} = 0; \quad \overline{t_1} = 0$
$\overline{x_0} = \overline{x_0} + \frac{\partial(x_0+t_1)}{\partial x_0} \cdot \bar{y} = 1 \cdot 1 = 1$
$\overline{t_1} = \overline{t_0} + \frac{\partial(x_0+t_1)}{\partial t_1} \cdot \bar{y} = 1 \cdot 1 = 1$
$\overline{x_1} = 0; \quad \overline{t_0} = 0$
$\overline{x_1} = \overline{x_1} + \frac{\partial(x_1 \cdot t_0)}{\partial x_1} \cdot \overline{t_1} = \overline{x_1} + t_0 \cdot \overline{t_1}$
$\overline{t_0} = \overline{t_0} + \frac{\partial(x_1 \cdot t_0)}{\partial t_0} \cdot \overline{t_1} = \overline{t_0} + x_1 \cdot \overline{t_1}$
$\overline{x_0} = \overline{x_0} + \frac{\partial \sin(x_0)}{\partial x_0} \cdot \overline{t_0} = \overline{x_0} + \cos(x_0) \cdot \overline{t_0}$
return $\overline{x_0},\ \overline{x_1}$

Classical API for AD

Reverse-mode:

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

Forward mode:

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

Classical API for AD

Reverse-mode:

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

Forward mode:

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

Matrix Multiplication in Futhark (C = A*B):

```
let C =
  map (\A_row ->
         map (\B_col ->
                map2 (*) A_row B_col
                |> reduce (+) 0
             ) (transpose B)
      ) A
```

AD by Re-execution

- In the sequential context, the tape is elegantly modeled by means of powerful programming abstractions (closures, co-routines), but these are not suitable for GPU execution;
- In our **nested-parallel** context, the tape is **complex/irregular** and must be passed across **deeply nested scopes**. Challenging to implement efficiently.

AD by Re-execution

- In the sequential context, the tape is elegantly modeled by means of powerful programming abstractions (closures, co-routines), but these are not suitable for GPU execution;
- In our **nested-parallel** context, the tape is **complex/irregular** and must be passed across **deeply nested scopes**. Challenging to implement efficiently.

Key idea #1: Re-execution

Whenever a new scope is entered, the code generation of the return sweep first re-executes the primal trace of that scope.

- **Asymptotics-preserving**: re-execution overhead is constant for non-recursive programs;
- No overhead for **perfectly nested scopes** (other than loops);
- Loops require **checkpointing** & Loop stripmining provides an easy way to navigate an effective **space-time tradeoff**.

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
```

Differentiating Reduce at a High Level

- Reduce combines all elements of an array with a binary associative operator $\odot$:

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

$$\equiv$$

$$\texttt{let}\ y = a_0 \odot a_1 \odot \cdots \odot a_{n-1}$$

Differentiating Reduce at a High Level

- Reduce combines all elements of an array with a binary associative operator $\odot$:

$$\begin{aligned} \textbf{let}\ y &= \textbf{reduce}\ \odot\ e_{\odot}\ [a_0, a_1, \ldots, a_{n-1}] \\ &\equiv \\ \textbf{let}\ y &= a_0 \odot a_1 \odot \cdots \odot a_{n-1} \end{aligned}$$

- For each a_i in the array, we can group the terms of reduce as:

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

Differentiating Reduce at a High Level

- Reduce combines all elements of an array with a binary associative operator $\odot$:

$$\begin{aligned} \textbf{let}\ y &= \textbf{reduce}\ \odot\ e_{\odot}\ [a_0, a_1, \ldots, a_{n-1}] \\ &\equiv \\ \textbf{let}\ y &= a_0 \odot a_1 \odot \cdots \odot a_{n-1} \end{aligned}$$

- For each a_i in the array, we can group the terms of reduce as:

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

  And then directly apply the AD rewrite rule

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

- We compute all $\overline{a_i}$ in parallel by mapping over all instances of l_i, a_i, r_i. How do we compute all instances of l_i and r_i?

Computing l_i and r_i

- For each $i \in \{0, \ldots, n-1\}$, need to compute l_i and r_i

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

- Compute all l_i with a (parallel) prefix sum operation, a.k.a., scan:

$$\begin{aligned} \textbf{let } ls = \ & \textbf{scan } \odot \ e_\odot \ [a_0, a_1, \ldots, a_{n-1}] \\ \equiv \ & [\underbrace{e_\odot}_{l_0}, \underbrace{a_0}_{l_1}, \underbrace{a_0 \odot a_1}_{l_2}, \ \ldots, \ \underbrace{a_0 \odot \ldots \odot a_{n-2}}_{l_{n-1}}] \end{aligned}$$

- For the r_is, do a backward scan, i.e., on the reversed array

$$\begin{aligned} \textbf{let } rs = \ & \textbf{reverse } as \ \triangleright \ \textbf{scan } (\lambda x \ y \to y \odot x) \ e_\odot \ [a_0, a_1, \ldots, a_{n-1}] \\ & \triangleright \ \textbf{reverse} \\ \equiv \ & [\underbrace{a_0 \odot \ldots \odot a_{n-2}}_{r_0}, \ldots, \underbrace{a_{n-2} \odot a_{n-1}}_{r_{n-3}}, \underbrace{a_{n-1}}_{r_{n-2}}, \underbrace{e_\odot}_{r_{n-1}}] \end{aligned}$$

The Reduce Rule

- The differentiation of reduce results in the following statements

$$
\begin{aligned}
&\textbf{let } y = \textbf{reduce } \odot\ e_{\odot}\ [a_0, a_1, \ldots, a_{n-1}] \\
&\qquad\vdots \\
&\textbf{let } ls = \textbf{scan } \odot\ e_{\odot}\ as \\
&\textbf{let } rs = \textbf{reverse } as \triangleright \textbf{scan } (\lambda x\ y \to y\ \odot\ x)\ e_{\odot} \triangleright \textbf{reverse} \\
&\textbf{let } \overline{as}\ \overline{+}\!= \textbf{map } \left(\lambda l_i\ a_i\ r_i \to \frac{\partial (l_i \odot a_i \odot r_i)}{\partial a_i}\ \overline{y}\right)\ ls\ as\ rs
\end{aligned}
$$

The Reduce Rule

- The differentiation of reduce results in the following statements

$$
\begin{aligned}
&\textbf{let } y = \textbf{reduce } \odot \ e_{\odot}\ [a_0, a_1, \ldots, a_{n-1}] \\
&\qquad\vdots \\
&\textbf{let } ls = \textbf{scan } \odot \ e_{\odot}\ as \\
&\textbf{let } rs = \textbf{reverse } as \triangleright \textbf{scan } (\lambda x\ y \to y \ \odot \ x)\ e_{\odot} \triangleright \textbf{reverse} \\
&\textbf{let } \overline{as} \mathbin{\overline{+}}= \textbf{map } \left(\lambda l_i\ a_i\ r_i \to \frac{\partial(l_i \odot a_i \odot r_i)}{\partial a_i}\ \overline{y}\right)\ ls\ as\ rs
\end{aligned}
$$

- The rule is asymptotics-preserving: scan has the same asymptotics as reduce.
- Specialized rules for other operators ($+$, min, max, $*$) admit significantly more efficient implementations (map-reduce).

Optimizing Accumulators: Matrix-Multiplication Eg

A class of optimizations refers to translating accumulators to more specialized constructs, e.g., reductions.

Matrix Multiplication Original:

```
forall i = 0..n—1                    // map
  forall j = 0 ..n—1                 // map
    for k = 0 .. n—1                 // map—reduce
      c[i,j] += a[i,k] * b[k,j]
```

Optimizing Accumulators: Matrix-Multiplication Eg

A class of optimizations refers to translating accumulators to more specialized constructs, e.g., reductions.

Matrix Multiplication Original:

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

Matrix Multiplication Reverse-AD with accumulators:

```
forall i = 0..n-1
  forall j = 0 ..n-1
    forall 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:

Optimizing Accumulators: Matrix-Multiplication Eg

A class of optimizations refers to translating accumulators to more specialized constructs, e.g., reductions.

Matrix Multiplication Original:

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

Matrix Multiplication Reverse-AD with accumulators:

```
forall i = 0..n-1
  forall j = 0 ..n-1
    forall 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.

MMM Example: Optimizing Generalized Reductions

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

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

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

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

MMM Example: Optimizing Generalized Reductions

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

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

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

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

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

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

MMM Example: Optimizing Generalized Reductions

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

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

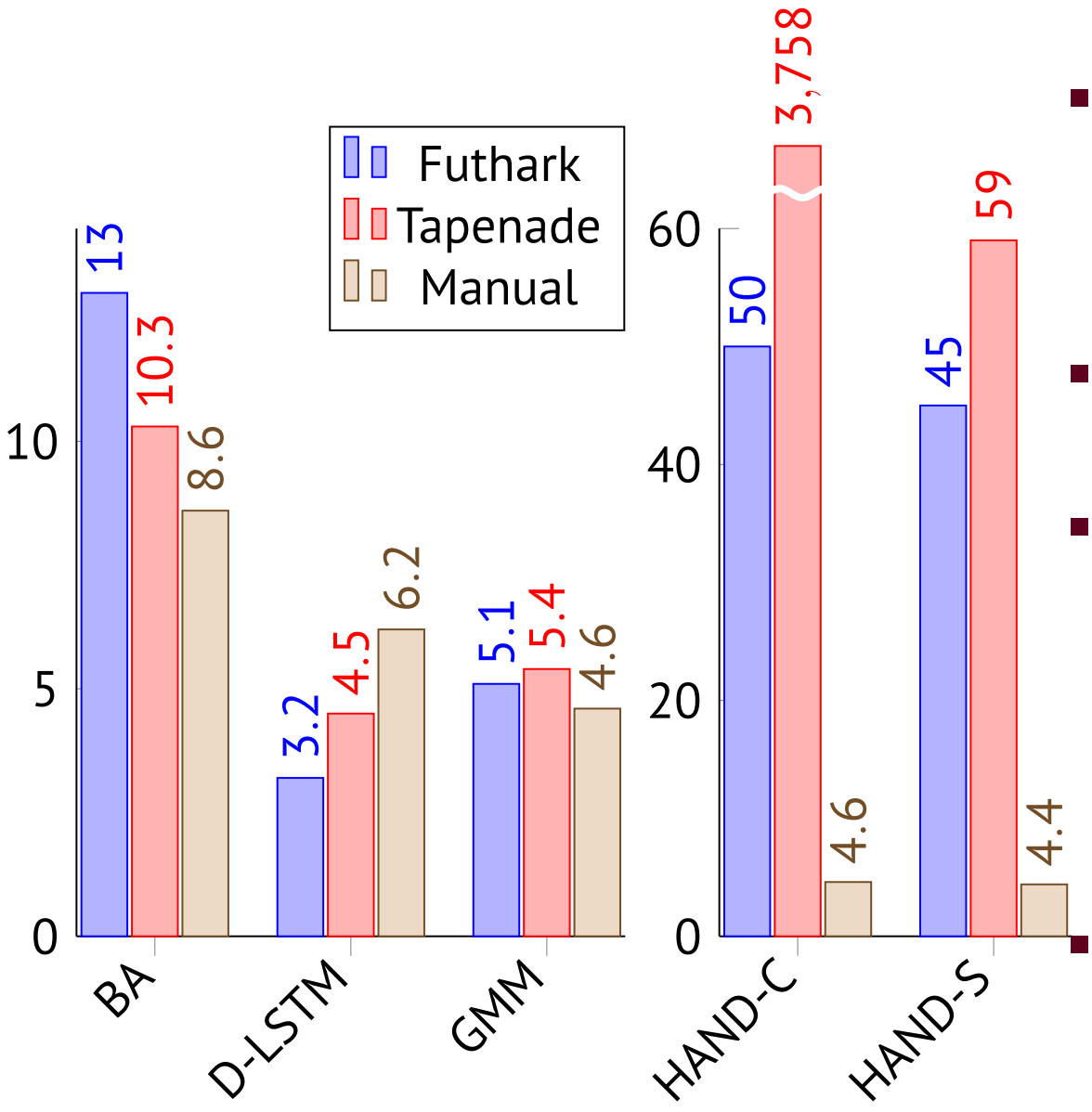
Now re-write (most of) the accumulations as classical reductions:

```
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]

CPU Benchmarks - ADBench



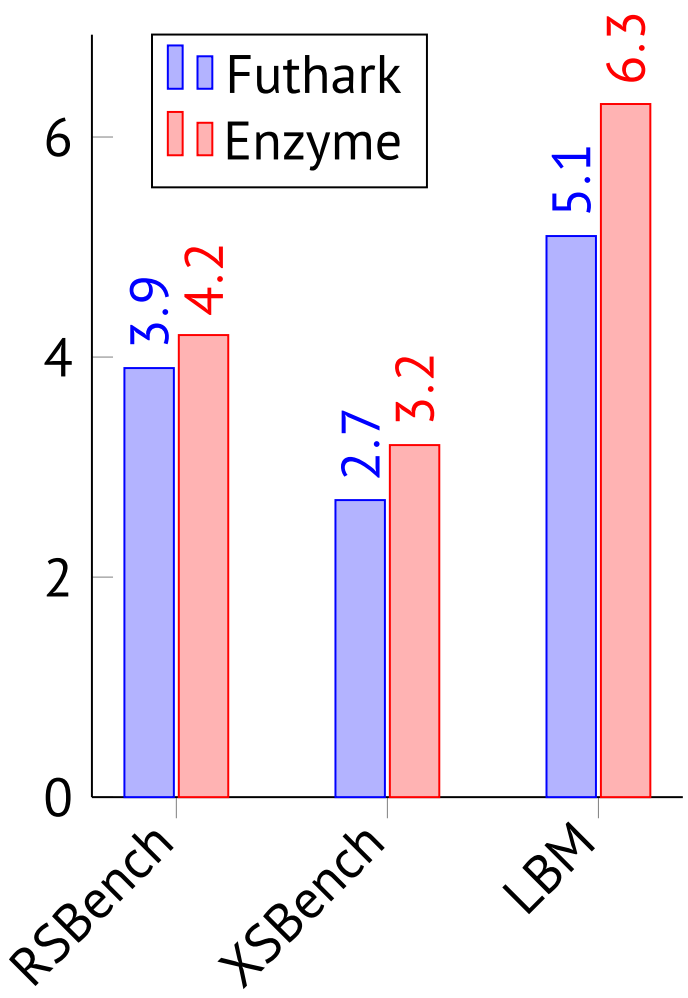
- ADBench: a collection of AD benchmarks for comparing sequential AD tools.
- Benchmarked Futhark using its C backend.
- Performance measured in **AD overhead**:

$$\frac{\text{differentiated runtime}}{\text{original runtime}}$$

the lower the better

- On BA, bottleneck is due to packing the result in CSR.

GPU Benchmarks - vs. Enzyme

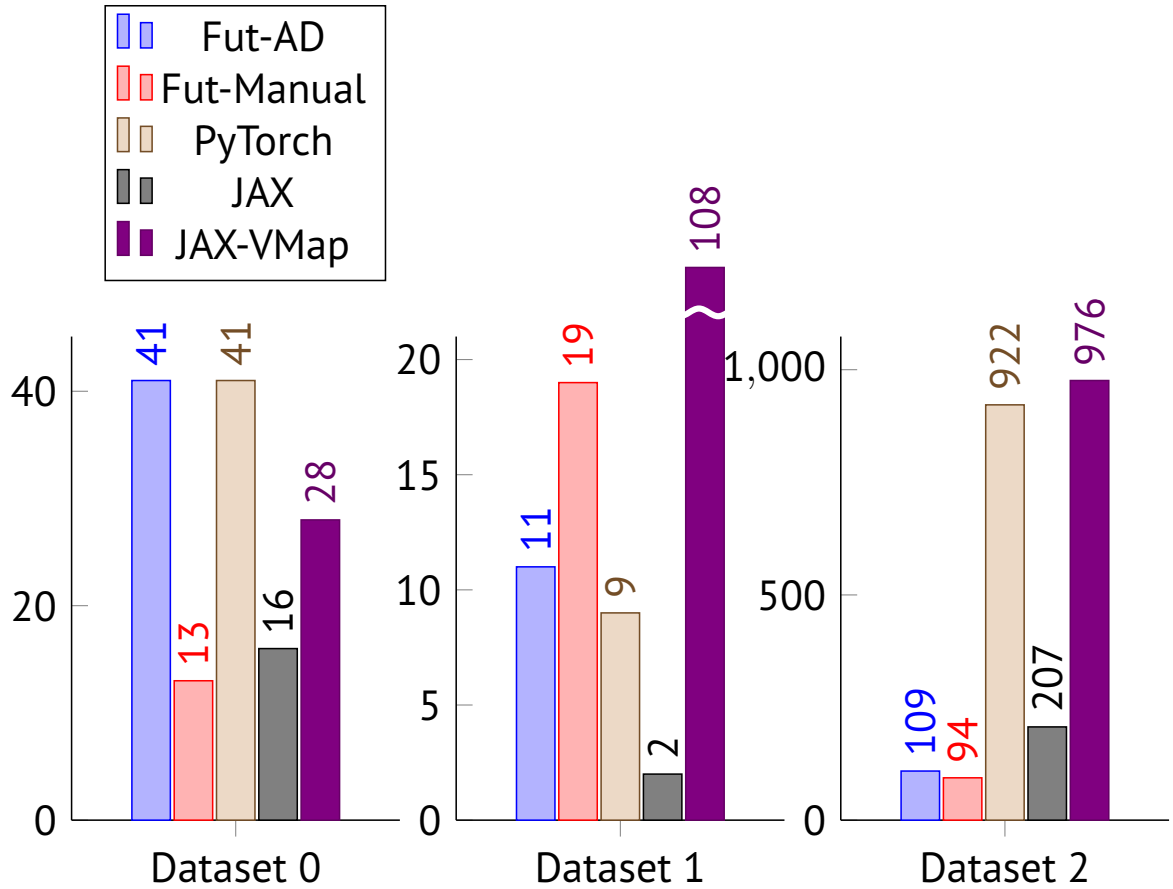


- Performance measured in **AD overhead**:

$$\frac{\text{differentiated runtime}}{\text{original runtime}}$$

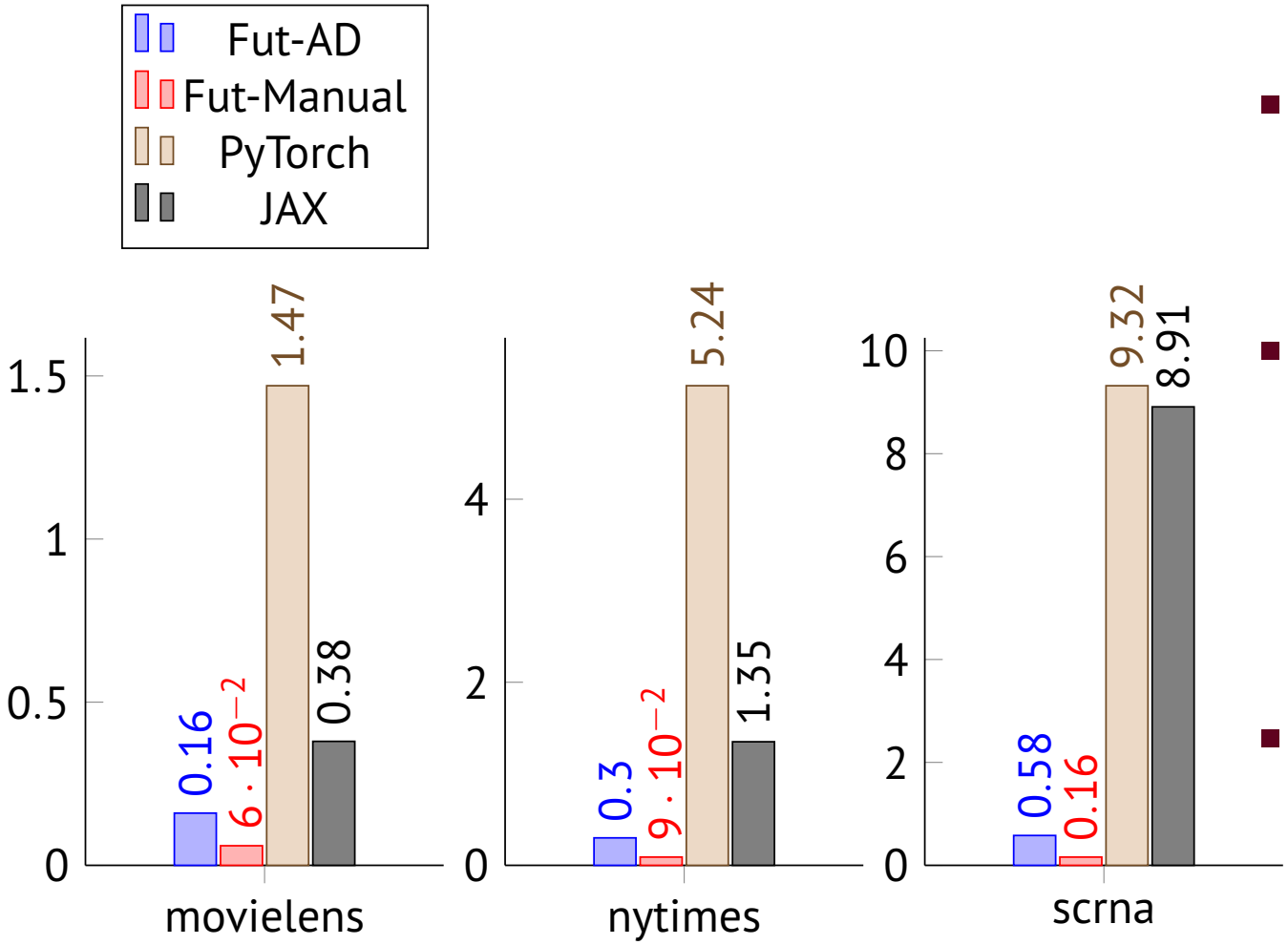
- Enzyme is state-of-the-art LLVM compiler plugin that performs AD on a low-level imperative IR.
- RSBench and XSBench are comprised of a large parallell loop with inner sequential loops and branches.
- LBM consists of a large sequential loop containing a parallel loop.

k-means by Newton's Method on NVIDIA's A100 GPU



- Performance measured in **miliseconds**.
- PyTorch and JAX use hand-tuned matrix primitives;
- JAX(vmap) instead uses JAX's vectorizing map operation.

Sparse k-means on Nvidia's A100 GPU



- Performance measured in **seconds**.
- PyTorch and JAX use hand-tuned matrix primitives and sparse libraries.
- Futhark just uses a standard CSR implementation.

GMM: GPU Performance vs PyTorch on A100 & MI100

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 & JAX

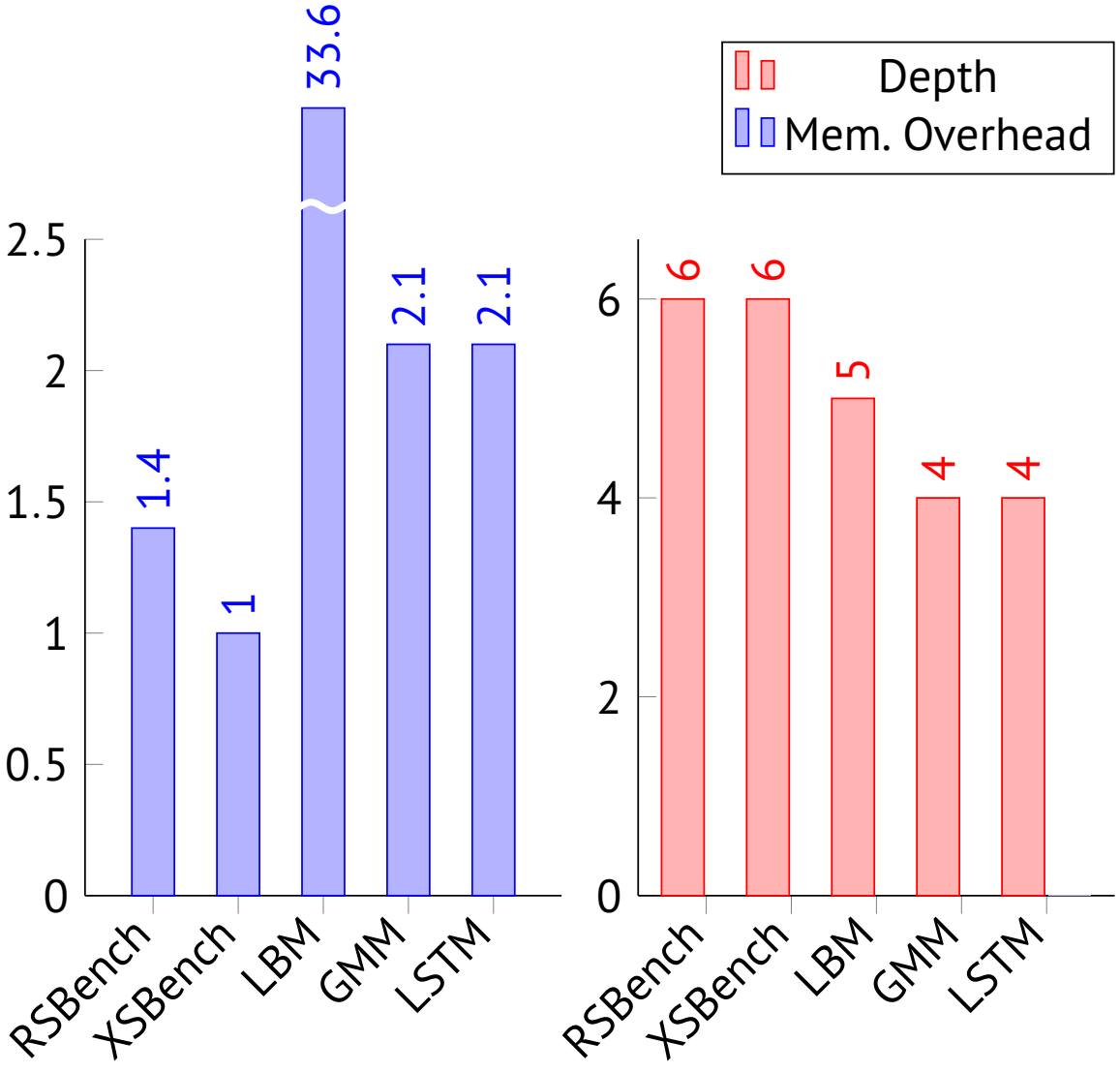
This is PyTorch's home ground, because matrix-multiplications take about 75% of runtime, and matrix-multiplication is a primitive in PyTorch, while Futhark needs to work hard for it.

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

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

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

GPU Benchmarks - Depth and Memory Consumption



- **AD Memory overhead**:

$$\frac{\text{differentiated mem.}}{\text{original mem.}}$$

- **Loop strip-mining** reduces LBM's memory overhead to 8.7×, at the cost of 1.3× increase in runtime.

Strong performance on programs with non-trivial depth demonstrates the viability of a recomputation-based approach to AD.

DEMO

Reduce with Multiplication

$$\textbf{let}\ y = w_0 \cdot w_1 \cdot \ldots \cdot w_{n-1}$$

Sequential:

```
loop y = 1
  for i = 0 .. n-1 do
    y * ws[i]
```

Parallel:

```
let y = reduce (*) 1 ws
```

Why Reduce with Arbitrary Operators?

$$\underbrace{\Sigma^{j<i}_{i,j=1..k} w_j \cdot w_i}_{P_k} = \underbrace{\left(\Sigma^{j<i}_{i,j=1..k-1} w_j \cdot w_i\right)}_{P_{k-1}} + \underbrace{\left(\Sigma_{i=1..k-1} w_i\right)}_{S_{k-1}} \cdot w_k$$

Why Reduce with Arbitrary Operators?

$$\underbrace{\Sigma^{j<i}_{i,j=1..k} w_j \cdot w_i}_{P_k} = \underbrace{\left(\Sigma^{j<i}_{i,j=1..k-1} w_j \cdot w_i\right)}_{P_{k-1}} + \underbrace{\left(\Sigma_{i=1..k-1} w_i\right)}_{S_{k-1}} \cdot w_k$$

Sequential:

```
let seqop (P,S) w_new = (P + S*w_new, S + w_new)

...
let (res, _) =
  loop y = (0,0)
    for i = 0 .. n-1 do
      seqop y ws[i]
```

Why Reduce with Arbitrary Operators?

$$\underbrace{\Sigma^{j<i}_{i,j=1..k} w_j \cdot w_i}_{P_k} = \underbrace{\left(\Sigma^{j<i}_{i,j=1..k-1} w_j \cdot w_i\right)}_{P_{k-1}} + \underbrace{\left(\Sigma_{i=1..k-1} w_i\right)}_{S_{k-1}} \cdot w_k$$

Sequential:

```
let seqop (P,S) w_new = (P + S*w_new, S + w_new)

...
let (res, _) =
  loop y = (0,0)
    for i = 0 .. n-1 do
      seqop y ws[i]
```

Parallel:

```
let parop (P1, S1) (P2, S2) = (P1 + P2 + S1*S2, S1+S2)

let (res, _) = map (\w -> (0, w)) ws -- lifting
            |> reduce parop (0,0)
```

Conclusions

- AD in a **nested-parallel**, **high-level** and **hardware-neutral** functional language.
- **Key idea:** high-level differentiation using specialized rules for parallel combinators.
- **Key idea:** re-computation instead of a tape (except for loops!).
- Strong performance against state-of-the-art AD competitors.
- The implementation is available now in the Futhark compiler—try it out!

`https://futhark-lang.org`

Motivating Example for AD

Gentle Introduction to AD

Key Idea #1: Redundant Execution Instead of Tape

Key Idea #2: Parallel Constructs are Differentiated at a High Level

Key Idea #3: Translate Accumulators to Specialized Constructs

Experimental Results: Competitive with State of the Art

Demo

Extra Slides

Loops

- **Sequential** loops are sugar for tail-recursive functions.
- **Loop parameters** are variables which are variant through the loop and are returned as the result of the loop.

```
loop y = 2 for i = 0 ... n - 1 do
    let y' = y * y
    in y'
```

```
y = 2
for i = 0 ... n - 1 do
   y = y * y
```

(Imperative analog)

- Storing the loop parameter y on the tape for each iteration is required in order to be able to execute the loop backwards.
- Parallel constructs do not require such check-pointing.

Differentiating Loops

```
let y'' =
 loop y = y_0 for i = 0 ... n - 1 do
  stms_loop
  in y'
```

1. Re-execute the original loop, save the value of *y* in each iteration in *ys*.

```
 2 let ys_0 = scratch(n,
 3                    sizeOf(y_0))
 4 let (y'', ys) =
 5  loop (y, ys) = (y_0, ys_0)
 6   for i = 0 ... n - 1 do
 7    let ys[i] = y
 8    stms_loop
 9    in (y', ys)
12 let (ȳ'', f̄vs_l) =
13  loop (ȳ, f̄vs_l) = (ȳ'', f̄vs_l0)
14   for i = n - 1 ... 0 do
15    let y = ys[i]
16    stms_loop  (→)
17    stms_loop  (←)
18    in (ȳ', f̄vs'_l)
19 let ȳ_0 += ȳ'''
```

Primal Trace

Return Sweep

Differentiating Loops

```
let y'' =
 loop y = y_0 for i = 0 ... n - 1 do
  stms_loop
  in y'
```

1. Re-execute the original loop, save the value of y in each iteration in ys.
2. Compute the adjoint contributions of the loop.

```
 2 let ys_0 = scratch(n,
 3                    sizeOf(y_0))
 4 let (y'', ys) =
 5  loop (y, ys) = (y_0, ys_0)
 6   for i = 0 ... n - 1 do
 7    let ys[i] = y
 8    stms_loop
 9    in (y', ys)
12 let (ȳ'', f̄vs_l) =
13  loop (ȳ, f̄vs_l) = (ȳ'', f̄vs_l0)
14   for i = n - 1 ... 0 do
15    let y = ys[i]
16    stms_loop →
17    stms_loop ←
18    in (ȳ', f̄vs'_l)
19 let ȳ_0 += ȳ''
```

Primal Trace

Return Sweep

Differentiating Loops

```
let y'' =
 loop y = y_0 for i = 0 ... n - 1 do
  stms_loop
  in y'
```

1. Re-execute the original loop, save the value of y in each iteration in ys.
2. Compute the adjoint contributions of the loop.
   - Run the loop backwards

```
 2 let ys_0 = scratch(n,
 3                    sizeOf(y_0))
 4 let (y'', ys) =
 5  loop (y, ys) = (y_0, ys_0)
 6   for i = 0 ... n - 1 do
 7    let ys[i] = y
 8    stms_loop
 9    in (y', ys)
12 let (ȳ'', f̄vs_l) =
13  loop (ȳ, f̄vs_l) = (ȳ'', f̄vs_{l_0})
14   for i = n - 1 ... 0 do
15    let y = ys[i]
16    stms_loop →
17    stms_loop ←
18    in (ȳ', f̄vs'_l)
19 let ȳ_0 += ȳ''
```

Primal Trace

Return Sweep

Differentiating Loops

```
let y'' =
 loop y = y_0 for i = 0 ... n - 1 do
  stms_loop
  in y'
```

1. Re-execute the original loop, save the value of y in each iteration in ys.
2. Compute the adjoint contributions of the loop.
   - Run the loop backwards
   - Restore the value of y from ys

```
 2 let ys_0 = scratch(n,
 3                    sizeOf(y_0))
 4 let (y'', ys) =
 5  loop (y, ys) = (y_0, ys_0)
 6   for i = 0 ... n - 1 do
 7    let ys[i] = y
 8    stms_loop
 9    in (y', ys)
12 let (ȳ'', f̄vs_l) =
13  loop (ȳ, f̄vs_l) = (ȳ'', f̄vs_{l_0})
14   for i = n - 1 ... 0 do
15    let y = ys[i]
16    stms_loop (→)
17    stms_loop (←)
18    in (ȳ', f̄vs'_l)
19 let ȳ_0 += ȳ'''
```

Primal Trace

Return Sweep

Differentiating Loops

```
let y'' =
 loop y = y_0 for i = 0 ... n - 1 do
  stms_loop
  in y'
```

1. Re-execute the original loop, save the value of y in each iteration in ys.
2. Compute the adjoint contributions of the loop.
   - Run the loop backwards
   - Restore the value of y from ys
   - Re-execute the body of the original loop

```
 2 let ys_0 = scratch(n,
 3                    sizeOf(y_0))
 4 let (y'', ys) =
 5  loop (y, ys) = (y_0, ys_0)
 6   for i = 0 ... n - 1 do
 7    let ys[i] = y
 8    stms_loop
 9    in (y', ys)
12 let (ȳ'', f̄vs_l) =
13  loop (ȳ, f̄vs_l) = (ȳ'', f̄vs_{l_0})
14   for i = n - 1 ... 0 do
15    let y = ys[i]
16    stms_loop (→)
17    stms_loop (←)
18    in (ȳ', f̄vs'_l)
19 let ȳ_0 += ȳ''
```

Primal Trace

Return Sweep

Differentiating Loops

```
let y'' =
 loop y = y_0 for i = 0 ... n - 1 do
  stms_loop
  in y'
```

1. Re-execute the original loop, save the value of y in each iteration in ys.
2. Compute the adjoint contributions of the loop.
   - Run the loop backwards
   - Restore the value of y from ys
   - Re-execute the body of the original loop
   - Compute the adjoints of the body

```
 2 let ys_0 = scratch(n,
 3                    sizeOf(y_0))
 4 let (y'', ys) =
 5  loop (y, ys) = (y_0, ys_0)
 6   for i = 0 ... n - 1 do
 7    let ys[i] = y
 8    stms_loop
 9    in (y', ys)
12 let (ȳ''', f̄vs_l) =
13  loop (ȳ, f̄vs_l) = (ȳ'', f̄vs_l0)
14   for i = n - 1 ... 0 do
15    let y = ys[i]
16    stms_loop  (→)
17    stms_loop  (←)
18    in (ȳ', f̄vs'_l)
19 let ȳ_0 += ȳ'''
```

Primal Trace

Return Sweep

Loop Strip-mining implements the Space-Time Trade-off

- **Loop strip-mining** partitions a loop into a loop nest

$$\textbf{loop}\ y = y_0 \quad \textbf{for}\ i = 0 \ldots n^3\text{-}1\ \textbf{do} \quad \textit{stms} \quad \Rightarrow \quad \textbf{loop}\ y_j = y_0\ \textbf{for}\ j = 0 \ldots n\text{-}1\ \textbf{do}\ \ \textbf{loop}\ y_k = y_j\ \textbf{for}\ k = 0 \ldots n - 1\ \textbf{do}\ \ \textbf{loop}\ y_m = y_k\ \textbf{for}\ m = 0 \ldots n - 1\ \textbf{do}\ \ \textbf{let}\ i = j * n^2 + k * n + m\ \ \textit{stms}$$

- For the original loop, we save n^3 versions of y on the tape.
- For the strip-mined loop, only $3n$ versions are saved.
- Strip-mining is controlled by the user via a simple annotation.

Strip-mining a loop k times results in **up to $k\times$ slowdown**, but memory overhead decreases from a factor of $n^k\times$ to $n \cdot k\times$.

Summary: Tape vs Redundant Execution

- 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).

Map without Free Variables

- Map is equivalent to a parallel for-loop

$$\texttt{let}\ xs = \texttt{map}\ (\lambda a\ b \rightarrow \texttt{let}\ res = a\ *\ b\ \texttt{in}\ res)\ as\ bs$$

$$\Updownarrow$$

$$\texttt{forall}\ i = 0 \ldots n - 1$$

$$xs[i] = as[i] * bs[i]$$

Map without Free Variables

- Map is equivalent to a parallel for-loop

$$\texttt{let } xs = \texttt{map } (\lambda a\ b \rightarrow \texttt{let } res = a * b \texttt{ in } res)\ as\ bs$$

$$\Updownarrow$$

$$\texttt{forall } i = 0 \ldots n - 1$$
$$xs[i] = as[i] * bs[i]$$

- Differentiating map is straightforward in the absence of free variables

$$\texttt{let } \overline{as}, \overline{bs} = \texttt{map } (\lambda a\ b\ \overline{x}\ \overline{a_0}\ \overline{b_0} \rightarrow$$
$$\texttt{let } res = a * b$$
$$\texttt{let } \overline{a} = \overline{a_0} + b * \overline{x}$$
$$\texttt{let } \overline{b} = \overline{b_0} + a * \overline{x}$$
$$\texttt{in } \overline{a}, \overline{b})\ as\ bs\ \overline{xs}\ \overline{as_0}\ \overline{bs_0}$$

Map with Free Variables

- Maps involving **free variables** are more complicated to differentiate

$$\mathbf{let}\ xs = \mathbf{map}\ (\lambda a \to a * b)\ as$$

- Naive approach: turn free variables into bound variables.

$$\mathbf{let}\ xs = \mathbf{map}\ (\lambda a\ b' \to a * b')\ as\ (\mathtt{replicate}\ n\ b)$$

- Problem: asymptotically inefficient for partially used free arrays.

Efficient Maps with Free Variables

- In an impure language, asymptotics-preserving adjoint updates for free array variables can be implemented as a **generalized reduction**:
  - preserves the useful properties of maps (parallel loops),
  - generalization of map, reduce, reduce-by-index, scatter.

- In this setting, the adjoint of a free aray variable $as[i]$ can be updated with an operation $\overline{as}[i] \ += \ v$.

- In our pure setting, we introduce **accumulators**.
  - **Write-only** view of an array.
  - Guarantees the generalized reduction properties at the type level.

- **An important set of optimizations refer to specializing generalized reductions to maps, reduce, reduce-by-index.**

Handling of Loops with In-Place Updates

The brownian loop brindge from OptionPricing, FinPar suite:

```
let bbrow =
  loop bbrow for i in 1..<num_dates do
    let bbrow[ bi[i] - 1 ] =
          sd[i] * gauss[i] +
          rw[i] * bbrow[ ri[i] - 1 ] +
          lw[i] * bbrow[ li[i] - 1 ]
in  bbrow
```

Checkpointing `bbrow` for each iteration breaks work asymptotics!

If the loop exhibits only true (RAW) dependencies, then:

Handling of Loops with In-Place Updates

The brownian loop brindge from OptionPricing, FinPar suite:

```
let bbrow =
  loop bbrow for i in 1..<num_dates do
    let bbrow[ bi[i] - 1 ] =
          sd[i] * gauss[i] +
          rw[i] * bbrow[ ri[i] - 1 ] +
          lw[i] * bbrow[ li[i] - 1 ]
in  bbrow
```

Checkpointing `bbrow` for each iteration breaks work asymptotics!

If the loop exhibits only true (RAW) dependencies, then:

On primal: before the loop, checkpoint the indices of `bbrow` written through the loop (e.g., the whole array);

On return: after computing the adjoint of the loop, restore the `bbrow` array to the state before the loop.

Rationale:

Handling of Loops with In-Place Updates

The brownian loop brindge from OptionPricing, FinPar suite:

```
let bbrow =
  loop bbrow for i in 1..<num_dates do
    let bbrow[ bi[i] - 1 ] =
          sd[i] * gauss[i] +
          rw[i] * bbrow[ ri[i] - 1 ] +
          lw[i] * bbrow[ li[i] - 1 ]
in  bbrow
```

Checkpointing `bbrow` for each iteration breaks work asymptotics!

If the loop exhibits only true (RAW) dependencies, then:

On primal: before the loop, checkpoint the indices of `bbrow` written through the loop (e.g., the whole array);

On return: after computing the adjoint of the loop, restore the `bbrow` array to the state before the loop.

Rationale:

- Parallel prog means avoiding (cross-iteration) dependencies;
- WAR & WAW are named **false** dependencies for a reason!