SC22
Dallas, TX | hpc accelerates.

AD for an Array Language with Nested Parallelism

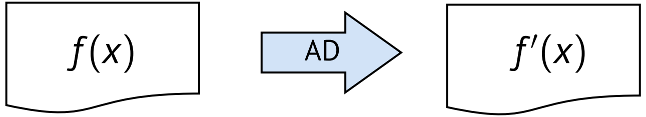
Robert Schenck, Ola Rønning, Troels Henriksen, and Cosmin E. Oancea

Department of Computer Science

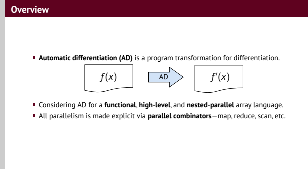
University of Copenhagen

Overview

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



- Considering AD for a **functional**, **high-level**, and **nested-parallel** array language.
- All parallelism is made explicit via **parallel combinators**—map, reduce, scan, etc.


Overview

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

$f(x)$ AD $f'(x)$

- Considering AD for a **functional**, **high-level**, and **nested-parallel** array language.
- All parallelism is made explicit via **parallel combinators**—map, reduce, scan, etc.

- AD is a program transformation which takes as input a program and computes its derivative. For example, the program may compute f(x) and the AD transformation yields a new program which computes f'(x).
- This talk is about AD in the context of pure, functional, high-level, and nested-parallel array language.
- A defining feature in the language is that all parallelism is made explicit with parallel combinators. Things like map, reduce, scan and so forth.

High-level AD

Key idea #1: High-level AD

Parallel constructs are differentiated at a **high-level**.

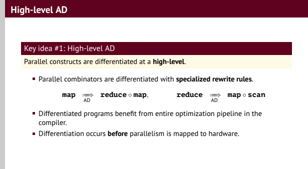
- Parallel combinators are differentiated with **specialized rewrite rules**.

$$\mathtt{map} \xRightarrow[\text{AD}]{} \mathtt{reduce} \circ \mathtt{map}, \qquad \mathtt{reduce} \xRightarrow[\text{AD}]{} \mathtt{map} \circ \mathtt{scan}$$

- Differentiated programs benefit from entire optimization pipeline in the compiler.
- Differentiation occurs **before** parallelism is mapped to hardware.

2024-06-16

└High-level AD



High-level AD

Key idea #1: High-level AD

Parallel constructs are differentiated at a **high-level**.

- Parallel combinators are differentiated with **specialized rewrite rules**.

$$\mathtt{map} \xRightarrow[\text{AD}]{} \mathtt{reduce} \circ \mathtt{map}, \qquad \mathtt{reduce} \xRightarrow[\text{AD}]{} \mathtt{map} \circ \mathtt{scan}$$

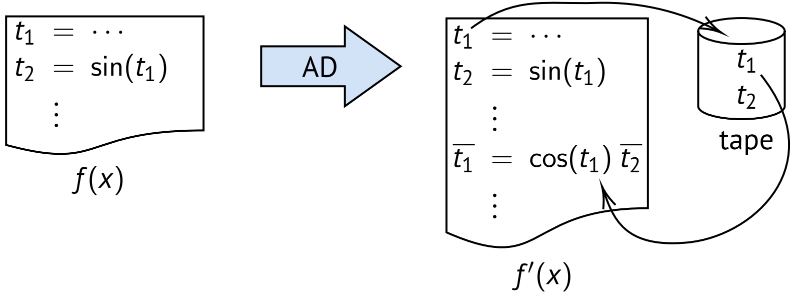
- Differentiated programs benefit from entire optimization pipeline in the compiler.
- Differentiation occurs **before** parallelism is mapped to hardware.

- Specialized rewrite rules: can write the derivative for each parallel construct in terms of parallel constructs themselves.
- Differentiation happens early in the compiler pipeline–differentiated prorams benefit from the *entire* pipeline.
- Before parallelism is mapped: flexibility to aggresively optimize the original code and differentiated code independently.

- Parallel structure of the differentiated code to differ from that of the original code.

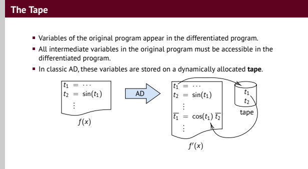
The Tape

- Variables of the original program appear in the differentiated program.
- All intermediate variables in the original program must be accessible in the differentiated program.
- In classic AD, these variables are stored on a dynamically allocated **tape**.



2024-06-16

└The Tape


The Tape

- Variables of the original program appear in the differentiated program.
- All intermediate variables in the original program must be accessible in the differentiated program.
- In classic AD, these variables are stored on a dynamically allocated **tape**.

- Key challenge in AD: variables of the original program appear in the differentiated program.
- Need mechanism to make the original program available to the differentiated program.
- Data structure on which we store original program variables is called the tape.
- Tape must be dynamically allocated since the amount of storage cannot be statically determined due to control flow.

AD by Re-execution

- In our **nested-parallel** context, the tape is **complex/irregular** and must be passed across **deeply nested scopes**. Challenging to implement efficiently.

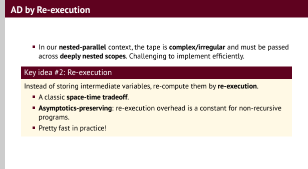
Key idea #2: Re-execution

Instead of storing intermediate variables, re-compute them by **re-execution**.

- A classic **space-time tradeoff**.
- **Asymptotics-preserving**: re-execution overhead is a constant for non-recursive programs.
- Pretty fast in practice!

2024-06-16

└AD by Re-execution



AD by Re-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 #2: Re-execution

Instead of storing intermediate variables, re-compute them by **re-execution**.

- A classic **space-time tradeoff**.
- **Asymptotics-preserving**: re-execution overhead is a constant for non-recursive programs.
- Pretty fast in practice!

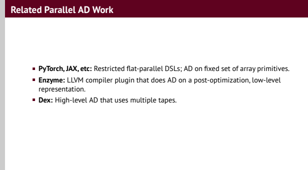
- Challenge to implement efficiently, especially in regards to coalesced memory accesses.
- One of our motivations for this work was demonstrating that the simpler approach of recomputation is not only computationally valid, but can yield excellent performance.

Related Parallel AD Work

- **PyTorch, JAX, etc:** Restricted flat-parallel DSLs; AD on fixed set of array primitives.
- **Enzyme:** LLVM compiler plugin that does AD on a post-optimization, low-level representation.
- **Dex:** High-level AD that uses multiple tapes.

2024-06-16

└Related Parallel AD Work


Related Parallel AD Work

- **PyTorch, JAX, etc:** Restricted flat-parallel DSLs; AD on fixed set of array primitives.
- **Enzyme:** LLVM compiler plugin that does AD on a post-optimization, low-level representation.
- **Dex:** High-level AD that uses multiple tapes.

- To give some context to where our work lies in the parallel AD ecosystem, let's look at some contemporaries that support efficient parallel AD.
- There are restricted, flat-parallel DSLs that do AD on array primitives like PyTorch and JAX. These languages, while often fast in some cases, are limited in what they can efficiently differentiate.
- There's also Enzyme, which, in contrast to us, differentiates on a post-optimization low-levle representation.
- There's Dex, which also takes a high-level approach to AD with a tape-based approach. As of yet they haven't published benchmarks.

A Very Short Introduction to AD

A Very Short Introduction to AD

I’ll now give a quick intro to the basics of AD.

Introduction to AD

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

$\Longrightarrow$

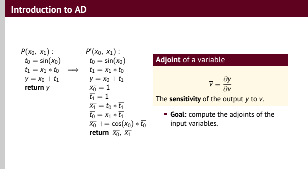
```
$P'(x_0,\ x_1):$
    $t_0 = \sin(x_0)$
    $t_1 = x_1 * t_0$
    $y = x_0 + t_1$
    $\overline{x_0} = 1$
    $\overline{t_1} = 1$
    $\overline{x_1} = t_0 * \overline{t_1}$
    $\overline{t_0} = x_1 * \overline{t_1}$
    $\overline{x_0} \mathrel{+}= \cos(x_0) * \overline{t_0}$
    return $\overline{x_0},\ \overline{x_1}$
```

Adjoint of a variable

$$\overline{v} \equiv \frac{\partial y}{\partial v}$$

The **sensitivity** of the output y to v.

- **Goal:** compute the adjoints of the input variables.


Introduction to AD
$P(x_0, x_1)$: $t_0 = \sin(x_0)$ $t_1 = x_1 * t_0$ $y = x_0 + t_1$ return y $\Longrightarrow$ $P'(x_0, x_1)$: $t_0 = \sin(x_0)$ $t_1 = x_1 * t_0$ $y = x_0 + t_1$ $\overline{x_0} = 1$ $\overline{t_1} = 1$ $\overline{x_1} = t_0 * \overline{t_1}$ $\overline{t_0} = x_1 * \overline{t_1}$ $\overline{x_0} \mathrel{+}= \cos(x_0) * \overline{t_0}$ return $\overline{x_0}, \overline{x_1}$
Adjoint of a variable $\overline{v} \equiv \frac{\partial y}{\partial v}$ The sensitivity of the output y to v.
Goal: compute the adjoints of the input variables.

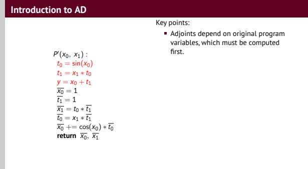
- On the far left we have some program P, with inputs x_0 and x_1 and return y.
- Next to it is its derivative, P', computed by AD. Notice that it has been augmented with a bunch of variables with bars on top of them. These are called *adjoints*.
- The adjoint of a variable is the sensitivty of the output of the original program to that variable.
- Notice that P' returns the adjoint of the inputs; that is the goal of AD. The adjoints of the inputs are exactly the derivative of the program.

Introduction to AD

```
$P'(x_0,\ x_1)$ :
    $t_0 = \sin(x_0)$
    $t_1 = x_1 * t_0$
    $y = x_0 + t_1$
    $\overline{x_0} = 1$
    $\overline{t_1} = 1$
    $\overline{x_1} = t_0 * \overline{t_1}$
    $\overline{t_0} = x_1 * \overline{t_1}$
    $\overline{x_0} \mathrel{+}= \cos(x_0) * \overline{t_0}$
    return $\overline{x_0},\ \overline{x_1}$
```

Key points:

- Adjoints depend on original program variables, which must be computed first.



Introduction to AD

```
$P'(x_0,\ x_1)$ :
    $t_0 = \sin(x_0)$
    $t_1 = x_1 * t_0$
    $y = x_0 + t_1$
    $\overline{x_0} = 1$
    $\overline{t_1} = 1$
    $\overline{x_1} = t_0 * \overline{t_1}$
    $\overline{t_0} = x_1 * \overline{t_1}$
    $\overline{x_0} \mathrel{+}= \cos(x_0) * \overline{t_0}$
    return $\overline{x_0},\ \overline{x_1}$
```

Key points:

- Adjoints depend on original program variables, which must be computed first.

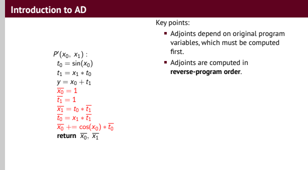
- Also noticce that many adjoints depend on other adjoints. To compute the adjoints of the inputs, we have to first compute the adjoints of any variables whih directly or indirectly depend on the inputs as the inputs can affect the output through these variables.

Introduction to AD

```
$P'(x_0,\ x_1)$ :
  $t_0 = \sin(x_0)$
  $t_1 = x_1 * t_0$
  $y = x_0 + t_1$
  $\overline{x_0} = 1$
  $\overline{t_1} = 1$
  $\overline{x_1} = t_0 * \overline{t_1}$
  $\overline{t_0} = x_1 * \overline{t_1}$
  $\overline{x_0} \mathrel{+}= \cos(x_0) * \overline{t_0}$
  return $\overline{x_0},\ \overline{x_1}$
```

Key points:

- Adjoints depend on original program variables, which must be computed first.
- Adjoints are computed in **reverse-program order**.



Introduction to AD

Key points:

- Adjoints depend on original program variables, which must be computed first.
- Adjoints are computed in **reverse-program order**.

$P'(x_0,\ x_1)$:
$t_0 = \sin(x_0)$
$t_1 = x_1 * t_0$
$y = x_0 + t_1$
$\overline{x_0} = 1$
$\overline{t_1} = 1$
$\overline{x_1} = t_0 * \overline{t_1}$
$\overline{t_0} = x_1 * \overline{t_1}$
$\overline{x_0} \mathrel{+}= \cos(x_0) * \overline{t_0}$
return $\overline{x_0},\ \overline{x_1}$

- Also noticce that many adjoints depend on other adjoints. To compute the adjoints of the inputs, we have to first compute the adjoints of any variables whih directly or indirectly depend on the inputs as the inputs can affect the output through these variables.

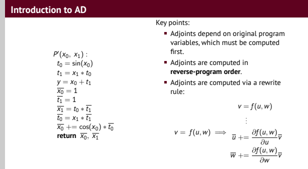
Introduction to AD

```
$P'(x_0,\ x_1)$ :
  $t_0 = \sin(x_0)$
  $t_1 = x_1 * t_0$
  $y = x_0 + t_1$
  $\overline{x_0} = 1$
  $\overline{t_1} = 1$
  $\overline{x_1} = t_0 * \overline{t_1}$
  $\overline{t_0} = x_1 * \overline{t_1}$
  $\overline{x_0} \mathrel{+}= \cos(x_0) * \overline{t_0}$
  **return** $\overline{x_0},\ \overline{x_1}$
```

Key points:

- Adjoints depend on original program variables, which must be computed first.
- Adjoints are computed in **reverse-program order**.
- Adjoints are computed via a rewrite rule:

$$v = f(u, w) \implies \begin{array}{c} v = f(u,w) \\ \vdots \\ \overline{u} \mathrel{+}= \dfrac{\partial f(u,w)}{\partial u}\overline{v} \\ \overline{w} \mathrel{+}= \dfrac{\partial f(u,w)}{\partial w}\overline{v} \end{array}$$



Introduction to AD

Key points:

- Adjoints depend on original program variables, which must be computed first.
- Adjoints are computed in **reverse-program order**.
- Adjoints are computed via a rewrite rule:

$P'(x_0,\ x_1)$: $t_0 = \sin(x_0)$ $t_1 = x_1 * t_0$ $y = x_0 + t_1$ $\overline{x_0} = 1$ $\overline{t_1} = 1$ $\overline{x_1} = t_0 * \overline{t_1}$ $\overline{t_0} = x_1 * \overline{t_1}$ $\overline{x_0} \mathrel{+}= \cos(x_0) * \overline{t_0}$ **return** $\overline{x_0},\ \overline{x_1}$

$v = f(u,w) \implies v = f(u,w)$ $\vdots$ $\overline{u} \mathrel{+}= \frac{\partial f(u,w)}{\partial u}\overline{v}$ $\overline{w} \mathrel{+}= \frac{\partial f(u,w)}{\partial w}\overline{v}$

- Also noticce that many adjoints depend on other adjoints. To compute the adjoints of the inputs, we have to first compute the adjoints of any variables whih directly or indirectly depend on the inputs as the inputs can affect the output through these variables.

AD Transformation

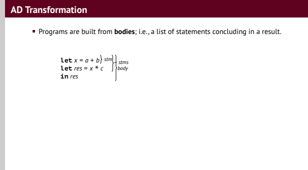
AD Transformation

Let’s now move on to looking at the actual AD transformation in the language.

AD Transformation

- Programs are built from **bodies**; i.e., a list of statements concluding in a result.

$$\left.\begin{array}{l}\left.\begin{array}{l}\left.\mathtt{let}\ x = a + b\right\}\ \mathit{stm}\\ \mathtt{let}\ \mathit{res} = x * c\end{array}\right\}\ \mathit{stms}\\ \mathtt{in}\ \mathit{res}\end{array}\right\}\ \mathit{body}$$



AD Transformation

- Programs are built from **bodies**; i.e., a list of statements concluding in a result.

let $x = a + b$ } stm } stms
let $res = x * c$ } body
in res

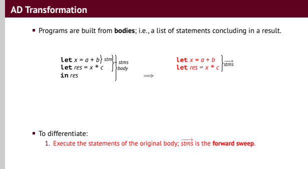
- We build our programs from *bodies*, which are a list of statements that conclude in a result.
- To differentiate, we first execute the statements of the original body, which we call the *forward sweep*.
- Then, we compute the adjoint contributions, which we call the *reverse sweep*, using the AD rewrite rule from before.
- Finally, we return the adjoints of free variables, since any local variables will be out of scope once the body returns.

AD Transformation

- Programs are built from **bodies**; i.e., a list of statements concluding in a result.

$$\left.\begin{array}{l}\left.\begin{array}{l}\left.\mathtt{let}\ x = a + b\right\}\mathit{stm}\\ \mathtt{let}\ \mathit{res} = x * c\end{array}\right\}\mathit{stms}\\ \mathtt{in}\ \mathit{res}\end{array}\right\}\mathit{body} \quad\Longrightarrow\quad \left.\begin{array}{l}\mathtt{let}\ x = a + b\\ \mathtt{let}\ \mathit{res} = x * c\end{array}\right\}\overrightarrow{\mathit{stms}}$$

- To differentiate:
  1. Execute the statements of the original body; $\overrightarrow{\mathit{stms}}$ is the **forward sweep**.



AD Transformation

- Programs are built from **bodies**; i.e., a list of statements concluding in a result.
- To differentiate:
  1. Execute the statements of the original body; $\overrightarrow{\mathit{stms}}$ is the **forward sweep**.

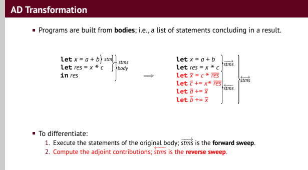
- We build our programs from *bodies*, which are a list of statements that conclude in a result.
- To differentiate, we first re-execute the statements of the original body, which we call the *forward sweep*.
- Then, we compute the adjoint contributions, which we call the *reverse sweep*, using the AD rewrite rule from before.
- Finally, we return the adjoints of free variables, since any local variables will be out of scope once the body returns.

AD Transformation

- Programs are built from **bodies**; i.e., a list of statements concluding in a result.

$$\left.\begin{array}{l}\left.\begin{array}{l}\left.\texttt{let}\ x = a + b\right\}\ stm\\ \texttt{let}\ res = x * c\end{array}\right\}\ stms\\ \texttt{in}\ res\end{array}\right\}\ body \implies \left.\begin{array}{l}\left.\begin{array}{l}\texttt{let}\ x = a + b\\ \texttt{let}\ res = x * c\end{array}\right\}\overrightarrow{stms}\\ \left.\begin{array}{l}\texttt{let}\ \overline{x} = c * \overline{res}\\ \texttt{let}\ \overline{c} \mathrel{+}= x * \overline{res}\\ \texttt{let}\ \overline{a} \mathrel{+}= \overline{x}\\ \texttt{let}\ \overline{b} \mathrel{+}= \overline{x}\end{array}\right\}\overleftarrow{stms}\end{array}\right\}\overleftrightarrow{stms}$$

- To differentiate:
  1. Execute the statements of the original body; $\overrightarrow{stms}$ is the **forward sweep**.
  2. Compute the adjoint contributions; $\overleftarrow{stms}$ is the **reverse sweep**.



AD Transformation
Programs are built from bodies; i.e., a list of statements concluding in a result.
To differentiate:
1. Execute the statements of the original body; $\overrightarrow{stms}$ is the forward sweep.
2. Compute the adjoint contributions; $\overleftarrow{stms}$ is the reverse sweep.

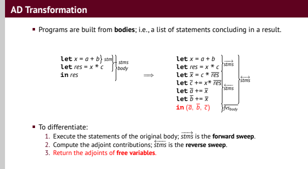
- We build our programs from *bodies*, which are a list of statements that conclude in a result.
- To differentiate, we first re-execute the statements of the original body, which we call the *forward sweep*.
- Then, we compute the adjoint contributions, which we call the *reverse sweep*, using the AD rewrite rule from before.
- Finally, we return the adjoints of free variables, since any local variables will be out of scope once the body returns.

AD Transformation

- Programs are built from **bodies**; i.e., a list of statements concluding in a result.

$$\left.\begin{array}{l}\left.\begin{array}{l}\left.\mathbf{let}\ x = a + b\right\}\ stm\\ \mathbf{let}\ res = x * c\end{array}\right\}\ stms\\ \mathbf{in}\ res\end{array}\right\}\ body \implies \begin{array}{l}\left.\begin{array}{l}\left.\begin{array}{l}\mathbf{let}\ x = a + b\\ \mathbf{let}\ res = x * c\end{array}\right\}\ \overrightarrow{stms}\\ \left.\begin{array}{l}\mathbf{let}\ \overline{x} = c * \overline{res}\\ \mathbf{let}\ \overline{c} \mathrel{+}= x * \overline{res}\\ \mathbf{let}\ \overline{a} \mathrel{+}= \overline{x}\\ \mathbf{let}\ \overline{b} \mathrel{+}= \overline{x}\end{array}\right\}\ \overleftarrow{stms}\end{array}\right\}\ \overleftrightarrow{stms}\\ \left.\mathbf{in}\ (\overline{a},\ \overline{b},\ \overline{c})\right\}\ \overline{fvs_{body}}\end{array}$$

- To differentiate:
  1. Execute the statements of the original body; $\overrightarrow{stms}$ is the **forward sweep**.
  2. Compute the adjoint contributions; $\overleftarrow{stms}$ is the **reverse sweep**.
  3. Return the adjoints of **free variables**.



AD Transformation

- Programs are built from **bodies**; i.e., a list of statements concluding in a result.

$$\left.\begin{array}{l}\left.\begin{array}{l}\left.\mathbf{let}\ x = a + b\right\}\ stm\\ \mathbf{let}\ res = x * c\end{array}\right\}\ stms\\ \mathbf{in}\ res\end{array}\right\}\ body \implies \begin{array}{l}\left.\begin{array}{l}\left.\begin{array}{l}\mathbf{let}\ x = a + b\\ \mathbf{let}\ res = x * c\end{array}\right\}\ \overrightarrow{stms}\\ \left.\begin{array}{l}\mathbf{let}\ \overline{x} = c * \overline{res}\\ \mathbf{let}\ \overline{c} \mathrel{+}= x * \overline{res}\\ \mathbf{let}\ \overline{a} \mathrel{+}= \overline{x}\\ \mathbf{let}\ \overline{b} \mathrel{+}= \overline{x}\end{array}\right\}\ \overleftarrow{stms}\end{array}\right\}\ \overleftrightarrow{stms}\\ \left.\mathbf{in}\ (\overline{a},\ \overline{b},\ \overline{c})\right\}\ \overline{fvs_{body}}\end{array}$$

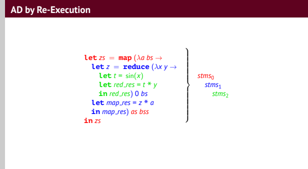
- To differentiate:
  1. Execute the statements of the original body; $\overrightarrow{stms}$ is the **forward sweep**.
  2. Compute the adjoint contributions; $\overleftarrow{stms}$ is the **reverse sweep**.
  3. Return the adjoints of **free variables**.

- We build our programs from *bodies*, which are a list of statements that conclude in a result.
- To differentiate, we first re-execute the statements of the original body, which we call the *forward sweep*.
- Then, we compute the adjoint contributions, which we call the *reverse sweep*, using the AD rewrite rule from before.
- Finally, we return the adjoints of free variables, since any local variables will be out of scope once the body returns.

AD by Re-Execution

```
let zs = map (λa bs →
  let z = reduce (λx y →
    let t = sin(x)
    let red_res = t * y
    in red_res) 0 bs
  let map_res = z * a
  in map_res) as bss
in zs
```

$$\left.\begin{array}{l}\\ \\ \\ \\ \\ \\ \\ \\ \end{array}\right\} \begin{array}{l} stms_0 \\ \quad stms_1 \\ \qquad stms_2 \end{array}$$



AD by Re-Execution

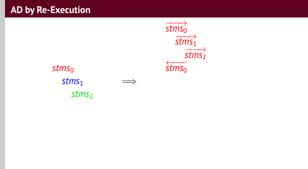
```
let zs = map (λa bs →
  let z = reduce (λx y →
    let t = sin(x)
    let red_res = t * y
    in red_res) 0 bs
  let map_res = z * a
  in map_res) as bss
in zs
```

$stms_0$ $stms_1$ $stms_2$

- Now let’s look at how our re-execution technique works in practice
- Here’s a sample program which contains three different color-coded scopes.
- On the right, we’ve represented the nested structure of this program as a series of indented statements.
- Let’s now see how these statements are re-executed during differentiation.

AD by Re-Execution

$$\begin{array}{l} stms_0 \\ \quad stms_1 \\ \quad\quad stms_2 \end{array} \Longrightarrow \begin{array}{l} \overrightarrow{stms_0} \\ \quad \overrightarrow{stms_1} \\ \quad\quad \overrightarrow{stms_2} \\ \overleftarrow{stms_0} \end{array}$$

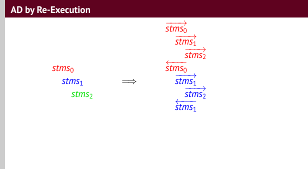


AD by Re-Execution

- First we differentiate the outermost scope. To do so, we re-execute the outermost scope to bring into scope any intermediate variables in the outermost scope. This is shown by the statements with the right arrow.
Then we compute the adjoints of the outermost scope in a reverse sweep, shown with the left arrow.
- Next, to differentiate the middle scope, denoted in blue, we re-execute the middle scope and then compute the adjoint updates.
- Finally, to differentiate the innermost scope, we re-execute only the innermost scope and then do a final reverse sweep for the adjoint updates.
- Notice that the amout of re-execution is proportional to the depth of the deepest scope. So, here, the deepest scope has depth 3 and we re-execute it 3 times.

AD by Re-Execution

$$\begin{array}{l} stms_0 \\ \quad stms_1 \\ \qquad stms_2 \end{array} \Longrightarrow \begin{array}{l} \overrightarrow{stms_0} \\ \quad \overrightarrow{stms_1} \\ \qquad \overrightarrow{stms_2} \\ \overleftarrow{stms_0} \\ \quad \overrightarrow{stms_1} \\ \qquad \overrightarrow{stms_2} \\ \quad \overleftarrow{stms_1} \end{array}$$



AD by Re-Execution

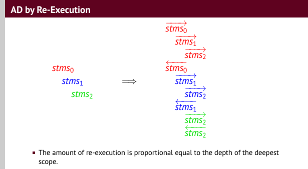
$$\begin{array}{l} stms_0 \\ \quad stms_1 \\ \qquad stms_2 \end{array} \Longrightarrow \begin{array}{l} \overrightarrow{stms_0} \\ \quad \overrightarrow{stms_1} \\ \qquad \overrightarrow{stms_2} \\ \overleftarrow{stms_0} \\ \quad \overrightarrow{stms_1} \\ \qquad \overrightarrow{stms_2} \\ \quad \overleftarrow{stms_1} \end{array}$$

- First we differentiate the outermost scope. To do so, we re-execute the outermost scope to bring into scope any intermediate variables in the outermost scope. This is shown by the statements with the right arrow.
Then we compute the adjoints of the outermost scope in a reverse sweep, shown with the left arrow.
- Next, to differentiate the middle scope, denoted in blue, we re-execute the middle scope and then compute the adjoint updates.
- Finally, to differentiate the innermost scope, we re-execute only the innermost scope and then do a final reverse sweep for the adjoint updates.
- Notice that the amout of re-execution is proportional to the depth of the deepest scope. So, here, the deepest scope has depth 3 and we re-execute it 3 times.

AD by Re-Execution

$$\begin{array}{ll}
\quad & \overrightarrow{stms_0} \\
 & \quad\overrightarrow{stms_1} \\
 & \qquad\overrightarrow{stms_2} \\
stms_0 & \overleftarrow{stms_0} \\
\quad stms_1 \qquad \Longrightarrow & \quad\overrightarrow{stms_1} \\
\qquad stms_2 & \qquad\overrightarrow{stms_2} \\
 & \quad\overleftarrow{stms_1} \\
 & \qquad\overrightarrow{stms_2} \\
 & \qquad\overleftarrow{stms_2}
\end{array}$$

- The amount of re-execution is proportional equal to the depth of the deepest scope.

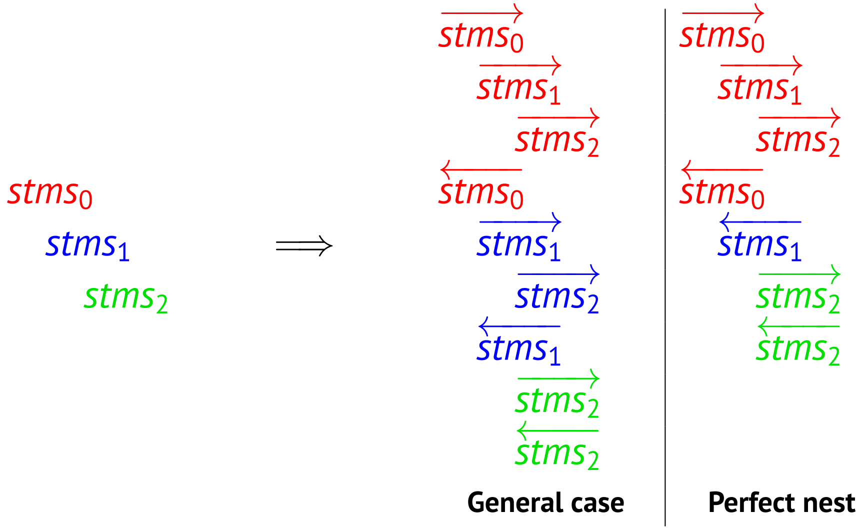


AD by Re-Execution

- The amount of re-execution is proportional equal to the depth of the deepest scope.

- First we differentiate the outermost scope. To do so, we re-execute the outermost scope to bring into scope any intermediate variables in the outermost scope. This is shown by the statements with the right arrow.
Then we compute the adjoints of the outermost scope in a reverse sweep, shown with the left arrow.
- Next, to differentiate the middle scope, denoted in blue, we re-execute the middle scope and then compute the adjoint updates.
- Finally, to differentiate the innermost scope, we re-execute only the innermost scope and then do a final reverse sweep for the adjoint updates.
- Notice that the amout of re-execution is proportional to the depth of the deepest scope. So, here, the deepest scope has depth 3 and we re-execute it 3 times.

Re-execution in Perfect Scope Nests



$stms_0$

$stms_1$

$stms_2$

$\Longrightarrow$

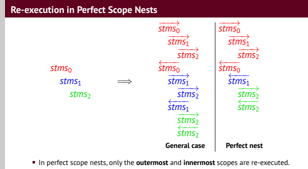
$\overrightarrow{stms_0}$
$\overrightarrow{stms_1}$
$\overrightarrow{stms_2}$
$\overleftarrow{stms_0}$
$\overrightarrow{stms_1}$
$\overrightarrow{stms_2}$
$\overleftarrow{stms_1}$
$\overrightarrow{stms_2}$
$\overleftarrow{stms_2}$

General case

$\overrightarrow{stms_0}$
$\overrightarrow{stms_1}$
$\overrightarrow{stms_2}$
$\overleftarrow{stms_0}$
$\overleftarrow{stms_1}$
$\overrightarrow{stms_2}$
$\overleftarrow{stms_2}$

Perfect nest

- In perfect scope nests, only the **outermost** and **innermost** scopes are re-executed.



Re-execution in Perfect Scope Nests

General case

Perfect nest

In perfect scope nests, only the **outermost** and **innermost** scopes are re-executed.

- In perfect nests, which are nests of parallel or loop constructs without any intermediate statements, re-execution isn't necessary for intermediate constructs in the nest.
- This means that we can use compiler optimizations like loop distribution and interchange to create perfect nests and exploit this fact.
- By applying these optimizations, we commonly expect the re-execution to only be executed twice: once for the outermost scope and once for the innermost scope.
- Actually, the example from before is a perfect nest because there are no intermediate statements between the map and reduce.

Differentiating Parallel Constructs

Differentiating Parallel Constructs

Let’s now look at how we differentiate the parallel constructs in the language.

Reduce

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

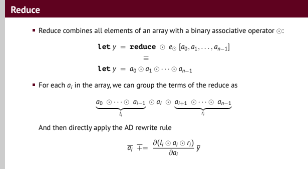
$$\begin{aligned} \texttt{let } y &= \texttt{reduce } \odot\ e_{\odot}\ [a_0, a_1, \ldots, a_{n-1}] \\ &\equiv \\ \texttt{let } y &= a_0 \odot a_1 \odot \cdots \odot a_{n-1} \end{aligned}$$

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

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

And then directly apply the AD rewrite rule

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



Reduce

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

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

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

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

And then directly apply the AD rewrite rule

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

- As you can see on the slide, reduce combines all elements of an array with some binary associative operator.
- To differentiate reduce, we first group the terms of the reduce for each element a_i into the elements which precede it−l_i−and the elements which come after it, r_i.
- At this point, we can just directly apply the AD rewrite rule from the introduction and obtain the adjust contributions for each a_i. The question that remains, then, is how to compute l_i and r_i *efficiently*.

Computing l_i and r_i

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

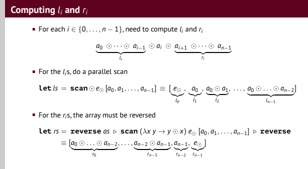
$$\underbrace{a_0 \odot \cdots \odot a_{i-1}}_{l_i} \odot a_i \odot \underbrace{a_{i+1} \odot \cdots \odot a_{n-1}}_{r_i}$$

- For the l_is, do a parallel scan

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

- For the r_is, the array must be reversed

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



Computing l_i and r_i

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

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

- For the l_is, do a parallel scan

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

- For the r_is, the array must be reversed

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

- So we need to compute l_i and r_i for *each* element in the array.
- For the l_is, we can do this in a straightforward way with a standard exclusive scan and for r_is, we first have reverse the list, then scan, and then reverse it back.

The Reduce Rule

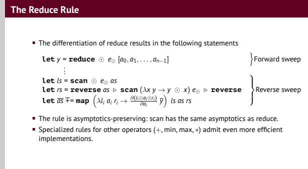
- The differentiation of reduce results in the following statements

$$\left.\begin{array}{l}\mathtt{let}\ y = \mathtt{reduce}\ \odot\ e_{\odot}\ [a_0, a_1, \ldots, a_{n-1}]\end{array}\right\}\text{Forward sweep}$$

$$\vdots$$

$$\left.\begin{array}{l}\mathtt{let}\ ls = \mathtt{scan}\ \odot\ e_{\odot}\ as \\ \mathtt{let}\ rs = \mathtt{reverse}\ as \triangleright \mathtt{scan}\ (\lambda x\ y \to y \odot x)\ e_{\odot} \triangleright \mathtt{reverse} \\ \mathtt{let}\ \overline{as}\ \overline{+}\!= \mathtt{map}\ \left(\lambda l_i\ a_i\ r_i \to \frac{\partial(l_i \odot a_i \odot r_i)}{\partial a_i}\ \overline{y}\right)\ ls\ as\ rs\end{array}\right\}\text{Reverse sweep}$$

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



The Reduce Rule

- The differentiation of reduce results in the following statements

let y = reduce ⊙ e⊙ [a0, a1, ..., an−1] } Forward sweep

let ls = scan ⊙ e⊙ as
let rs = reverse as ▷ scan (λx y → y ⊙ x) e⊙ ▷ reverse
let as += map (λli ai ri → ∂(li⊙ai⊙ri)/∂ai y) ls as rs } Reverse sweep

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

- We now have all the ingredients for our rewrite rule for ‘reduce‘, which consists of just the original statement in the forward sweep, followed by scans to compute the ‘l_i‘s and ‘r_i‘s.
- Finally, we perform a map over ‘l_i‘s, ‘r_i‘s and ‘as‘ to compute the adjoint contribution for each element in the reduce.

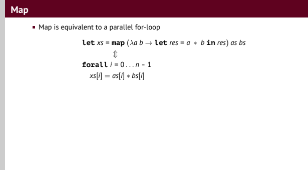
Map

- Map is equivalent to a parallel for-loop

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

$$\Updownarrow$$

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



Map

- Map is equivalent to a parallel for-loop

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

$$\Updownarrow$$

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

- Let’s now move on to differentiating map, which is equivalent to an imperative parallel for loop.
- To differentiate map, we just differentiate the body of the function being mapped and modify the map to recieve additional arguments necessary to compute the adjoints of the body of the function being mapped: namely the adjoints of the arrays being mapped over as well as the adjoint of the LHS of the original statement.

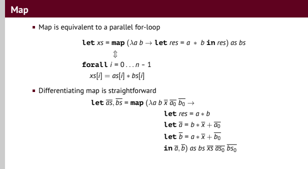
Map

- Map is equivalent to a parallel for-loop

$$
\begin{array}{l}
\mathtt{let}\ xs = \mathbf{map}\ (\lambda a\ b \rightarrow \mathtt{let}\ res = a * b\ \mathtt{in}\ res)\ as\ bs \\
\qquad\qquad \Updownarrow \\
\mathtt{forall}\ i = 0 \ldots n - 1 \\
\quad xs[i] = as[i] * bs[i]
\end{array}
$$

- Differentiating map is straightforward

$$
\begin{array}{ll}
\mathtt{let}\ \overline{as}, \overline{bs} = \mathbf{map}\ (\lambda a\ b\ \overline{x}\ \overline{a_0}\ \overline{b_0} \rightarrow & \\
 & \mathtt{let}\ res = a * b \\
 & \mathtt{let}\ \overline{a} = b * \overline{x} + \overline{a_0} \\
 & \mathtt{let}\ \overline{b} = a * \overline{x} + \overline{b_0} \\
 & \mathtt{in}\ \overline{a}, \overline{b})\ as\ bs\ \overline{xs}\ \overline{as_0}\ \overline{bs_0}
\end{array}
$$



- Let’s now move on to differentiating map, which is equivalent to an imperative parallel for loop.
- To differentiate map, we just differentiate the body of the function being mapped and modify the map to recieve additional arguments necessary to compute the adjoints of the body of the function being mapped: namely the adjoints of the arrays being mapped over as well as the adjoint of the LHS of the original statement.

Map with Free Variables

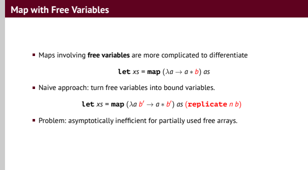
- Maps involving **free variables** are more complicated to differentiate

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

- Naive approach: turn free variables into bound variables.

$$\texttt{let}\ xs = \textbf{map}\ (\lambda a\ b' \rightarrow a * b')\ as\ (\texttt{replicate}\ n\ b)$$

- Problem: asymptotically inefficient for partially used free arrays.



Map with Free Variables

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

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

- Naive approach: turn free variables into bound variables.

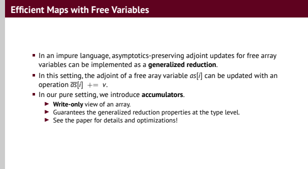
$$\texttt{let}\ xs = \textbf{map}\ (\lambda a\ b' \rightarrow a * b')\ as\ (\texttt{replicate}\ n\ b)$$

- Problem: asymptotically inefficient for partially used free arrays.

- If the map involves a free variable, things are more complicated. By turning free variables into bound variables, we can use our previous approach.
- Unfortunately, this is asymptotically inefficient for partially used 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**.
- In this setting, the adjoint of a free aray variable $as[i]$ can be updated with an operation $\overline{as}[i] \mathrel{+}= v$.
- In our pure setting, we introduce **accumulators**.
  - **Write-only** view of an array.
  - Guarantees the generalized reduction properties at the type level.
  - See the paper for details and optimizations!



Efficient Maps with Free Variables

- In an impure language, asymptotics-preserving adjoint updates for free array variables can be implemented as a **generalized reduction**.
- In this setting, the adjoint of a free aray variable $as[i]$ can be updated with an operation $\overline{as}[i] \mathrel{+}= v$.
- In our pure setting, we introduce **accumulators**.
  - **Write-only** view of an array.
  - Guarantees the generalized reduction properties at the type level.
  - See the paper for details and optimizations!

- In an impure language, asymptotics-preserving adjoint updates can be implemented as a generalized reduction.
- What this basically looks like is just doing a plus-equals to update the adjoint at specific indices.
- In our purely functional setting, such updates aren't possible.
- Instead, to be able to efficiently handle adjoint updates to free array variables, we've introduced an *accumulator* construct.
- See the paper for much more detail as well a discussion on optimizing accumulator accesses to memory.

Loops

Loops

Let’s now move on to our final construct, which are loops.

Loops

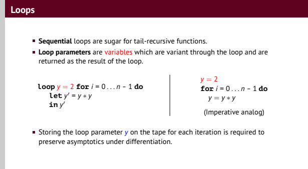
- **Sequential** loops are sugar for tail-recursive functions.
- **Loop parameters** are variables which are variant through the loop and are returned as the result of the loop.

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

```
$y = 2$
for $i = 0 \ldots n - 1$ do
    $y = y * y$
```

(Imperative analog)

- Storing the loop parameter y on the tape for each iteration is required to preserve asymptotics under differentiation.



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 $i = 0 \ldots n - 1$ do let $y' = y * y$ in y'

$y = 2$ for $i = 0 \ldots n - 1$ do $y = y * y$ (Imperative analog)

- Storing the loop parameter y on the tape for each iteration is required to preserve asymptotics under differentiation.

- Loop statements specify one or more *loop parameters* which are the variant through the loop and are also the result of the loop.
- In the example on the left, ‘y‘ is the loop parameter. Each iteration it’s squared and then returned after n iterations.
- Until now, our approach to differentiation has been entirely re-computation based. This strategy doesn’t work for loops: must store the loop parameter for each iteration on a tape in order to preserve the asymptotics of the original program.

Differentiating Loops

```
let $y''$ =
 loop $y = y_0$ for $i = 0 \dots n - 1$ do
  $stms_{loop}$
  in $y'$
```

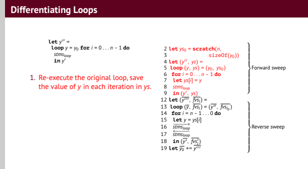
1. Re-execute the original loop, save the value of y in each iteration in ys.

```
let $ys_0$ = scratch($n$,
                    sizeOf($y_0$))
let $(y'', ys)$ =
 loop $(y, ys) = (y_0, ys_0)$
  for $i = 0 \dots n - 1$ do
   let $ys[i] = y$
   $stms_{loop}$
   in $(y', ys)$
```

Forward sweep

```
let $(\overline{y'''}, \overline{fvs_l})$ =
 loop $(\overline{y}, \overline{fvs_l}) = (\overline{y''}, \overline{fvs_{l_0}})$
  for $i = n - 1 \dots 0$ do
   let $y = ys[i]$
   $\overrightarrow{stms_{loop}}$
   $\overleftarrow{stms_{loop}}$
   in $(\overline{y'}, \overline{fvs'_l})$
let $\overline{y_0}$ += $\overline{y'''}$
```

Reverse sweep



- Let's now look at how to differentiate the loop on the top left.
1. First, we re-execute the original loop and store the loop parameters for each iteration.
2. We then compute the adjoint contributions of the loop:
* This consists of runing the loop backwards since we compute adjoints in reverse program order.
* Next we restore the loop parameter from the tape.
* Followed by re-executing the body of the original loop.
* And finally we compute the adjoints of the body.

Differentiating Loops

```
let $y''$ =
 loop $y = y_0$ for $i = 0 \ldots n - 1$ 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.

```
let $ys_0$ = scratch($n$,
                     sizeOf($y_0$))
let $(y'', ys)$ =
 loop $(y, ys) = (y_0, ys_0)$
  for $i = 0 \ldots n - 1$ do
   let $ys[i] = y$
   $stms_{loop}$
   in $(y', ys)$
let $(\overline{y'''}, \overline{fvs_l})$ =
 loop $(\overline{y}, \overline{fvs_l}) = (\overline{y''}, \overline{fvs_{l_0}})$
  for $i = n - 1 \ldots 0$ do
   let $y = ys[i]$
   $\overrightarrow{stms_{loop}}$
   $\overleftarrow{stms_{loop}}$
   in $(\overline{y'}, \overline{fvs'_l})$
let $\overline{y_0}$ += $\overline{y'''}$
```

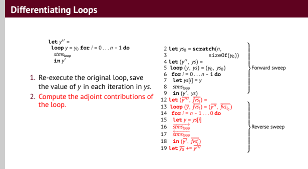
Forward sweep

Reverse sweep

2024-06-16

└Loops

└Differentiating Loops



- Let’s now look at how to differentiate the loop on the top left.
1. First, we re-execute the original loop and store the loop parameters for each iteration.
2. We then compute the adjoint contributions of the loop:
* This consists of runing the loop backwards since we compute adjoints in reverse program order.
* Next we restore the loop parameter from the tape.
* Followed by re-executing the body of the original loop.
* And finally we compute the adjoints of the body.

Differentiating Loops

```
let $y'' =$
 loop $y = y_0$ for $i = 0 \dots n - 1$ do
  $stms_{loop}$
  in $y'$
```

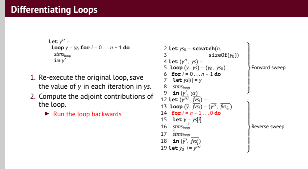
1. Re-execute the original loop, save the value of y in each iteration in ys.
2. Compute the adjoint contributions of the loop.
   - Run the loop backwards

```
let $ys_0$ = scratch($n$,
                     sizeOf($y_0$))
let $(y'', ys)$ =
 loop $(y, ys) = (y_0, ys_0)$
  for $i = 0 \dots n - 1$ do
   let $ys[i] = y$
   $stms_{loop}$
   in $(y', ys)$
```

Forward sweep

```
let $(\overline{y'''}, \overline{fvs_l})$ =
 loop $(\overline{y}, \overline{fvs_l}) = (\overline{y''}, \overline{fvs_{l_0}})$
  for $i = n - 1 \dots 0$ do
   let $y = ys[i]$
   $\overrightarrow{stms_{loop}}$
   $\overleftarrow{stms_{loop}}$
   in $(\overline{y'}, \overline{fvs'_l})$
let $\overline{y_0} \mathrel{+}= \overline{y'''}$
```

Reverse sweep



Differentiating Loops

let $y'' =$ loop $y = y_0$ for $i = 0 \dots n - 1$ 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 $i = 0 \dots n - 1$ do 7 let $ys[i] = y$ 8 $stms_{loop}$ 9 in (y', ys) 12 let $(\overline{y'''}, \overline{fvs_l})$ = 13 loop $(\overline{y}, \overline{fvs_l}) = (\overline{y''}, \overline{fvs_{l_0}})$ 14 for $i = n - 1 \dots 0$ do 15 let $y = ys[i]$ 16 $\overrightarrow{stms_{loop}}$ 17 $\overleftarrow{stms_{loop}}$ 18 in $(\overline{y'}, \overline{fvs'_l})$ 19 let $\overline{y_0} \mathrel{+}= \overline{y'''}$

Forward sweep

Reverse sweep

- Let's now look at how to differentiate the loop on the top left.
1. First, we re-execute the original loop and store the loop parameters for each iteration.
2. We then compute the adjoint contributions of the loop:
* This consists of runing the loop backwards since we compute adjoints in reverse program order.
* Next we restore the loop parameter from the tape.
* Followed by re-executing the body of the original loop.
* And finally we compute the adjoints of the body.

Differentiating Loops

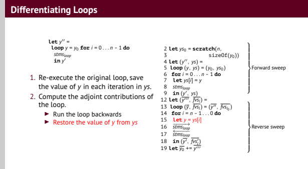
```
let $y''$ =
  loop $y = y_0$ for $i = 0 \ldots n-1$ do
    $stms_{loop}$
    in $y'$
```

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

```
let $ys_0$ = scratch($n$,
                     sizeOf($y_0$))
let ($y''$, $ys$) =
 loop ($y$, $ys$) = ($y_0$, $ys_0$)
  for $i = 0 \ldots n-1$ do
   let $ys[i] = y$
   $stms_{loop}$
   in ($y'$, $ys$)
let ($\overline{y'''}$, $\overline{fvs_l}$) =
 loop ($\overline{y}$, $\overline{fvs_l}$) = ($\overline{y''}$, $\overline{fvs_{l_0}}$)
  for $i = n-1 \ldots 0$ do
   let $y = ys[i]$
   $\overrightarrow{stms_{loop}}$
   $\overleftarrow{stms_{loop}}$
   in ($\overline{y'}$, $\overline{fvs'_l}$)
let $\overline{y_0}$ += $\overline{y'''}$
```

Forward sweep

Reverse sweep



- Let’s now look at how to differentiate the loop on the top left.
1. First, we re-execute the original loop and store the loop parameters for each iteration.
2. We then compute the adjoint contributions of the loop:
* This consists of runing the loop backwards since we compute adjoints in reverse program order.
* Next we restore the loop parameter from the tape.
* Followed by re-executing the body of the original loop.
* And finally we compute the adjoints of the body.

Differentiating Loops

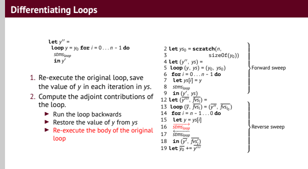
```
let $y'' =$
 loop $y = y_0$ for $i = 0 \ldots n - 1$ do
  $stms_{loop}$
  in $y'$
```

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

```
let $ys_0$ = scratch($n$,
                     sizeOf($y_0$))
let $(y'', ys) =$
 loop $(y, ys) = (y_0, ys_0)$
  for $i = 0 \ldots n - 1$ do
   let $ys[i] = y$
   $stms_{loop}$
   in $(y', ys)$
let $(\overline{y'''}, \overline{fvs_l}) =$
 loop $(\bar{y}, \overline{fvs_l}) = (\overline{y''}, \overline{fvs_{l_0}})$
  for $i = n - 1 \ldots 0$ do
   let $y = ys[i]$
   $\overrightarrow{stms_{loop}}$
   $\overleftarrow{stms_{loop}}$
   in $(\overline{y'}, \overline{fvs'_l})$
let $\overline{y_0}$ += $\overline{y'''}$
```

Forward sweep

Reverse sweep


Differentiating Loops

- Let's now look at how to differentiate the loop on the top left.
1. First, we re-execute the original loop and store the loop parameters for each iteration.
2. We then compute the adjoint contributions of the loop:
* This consists of runing the loop backwards since we compute adjoints in reverse program order.
* Next we restore the loop parameter from the tape.
* Followed by re-executing the body of the original loop.
* And finally we compute the adjoints of the body.

Differentiating Loops

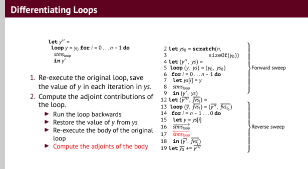
```
let $y'' =$
 loop $y = y_0$ for $i = 0 \ldots n - 1$ do
  $stms_{loop}$
  in $y'$
```

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

```
let $ys_0 =$ scratch($n$,
                  sizeOf($y_0$))
let $(y'', ys) =$
 loop $(y, ys) = (y_0, ys_0)$
  for $i = 0 \ldots n - 1$ do
   let $ys[i] = y$
   $stms_{loop}$
   in $(y', ys)$
let $(\overline{y'''}, \overline{fvs_l}) =$
 loop $(\overline{y}, \overline{fvs_l}) = (\overline{y''}, \overline{fvs_{l_0}})$
  for $i = n - 1 \ldots 0$ do
   let $y = ys[i]$
   $\overrightarrow{stms_{loop}}$
   $\overleftarrow{stms_{loop}}$
   in $(\overline{y'}, \overline{fvs'_l})$
let $\overline{y_0}$ += $\overline{y'''}$
```

Forward sweep

Reverse sweep



Differentiating Loops

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

Forward sweep

Reverse sweep

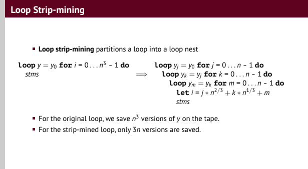
- Let’s now look at how to differentiate the loop on the top left.
1. First, we re-execute the original loop and store the loop parameters for each iteration.
2. We then compute the adjoint contributions of the loop:
* This consists of runing the loop backwards since we compute adjoints in reverse program order.
* Next we restore the loop parameter from the tape.
* Followed by re-executing the body of the original loop.
* And finally we compute the adjoints of the body.

Loop Strip-mining

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

```
loop $y = y_0$ for $i = 0 \ldots n^3 - 1$ do          $\Longrightarrow$    loop $y_j = y_0$ for $j = 0 \ldots n - 1$ do
    $stms$                                                   loop $y_k = y_j$ for $k = 0 \ldots n - 1$ do
                                                                 loop $y_m = y_k$ for $m = 0 \ldots n - 1$ do
                                                                     let $i = j * n^{2/3} + k * n^{1/3} + m$
                                                                     $stms$
```

- For the original loop, we save n^3 versions of y on the tape.
- For the strip-mined loop, only $3n$ versions are saved.



Loop Strip-mining

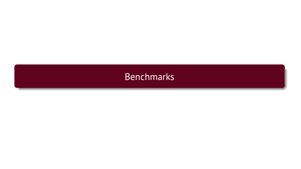
- Loop strip-mining partitions a loop into a loop nest

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

- For the original loop, we save n^3 versions of y on the tape.
- For the strip-mined loop, only $3n$ versions are saved.

- Saving loop parameters for large loops can be expensive and use large amounts of memory.
- One technique to address this is *loop strip-mining*. This partitions a loop into loop nest.
- In the figure, we strip-mine a loop that does n cubed iteration into a nest of three loops which each do n iterations.
- When differentiated, each loop in the loop nest only saves its own loop parameters, meaning that we only save 3*n loop parameters instead of n cubed.
- Strip-mining results in additional re-execution overhead, but the overhead is generally vest modest.

Benchmarks



Benchmarks

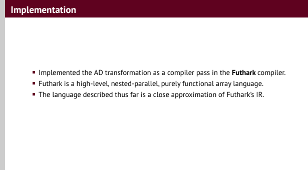
Implementation

- Implemented the AD transformation as a compiler pass in the **Futhark** compiler.
- Futhark is a high-level, nested-parallel, purely functional array language.
- The language described thus far is a close approximation of Futhark's IR.

2024-06-16

└Benchmarks

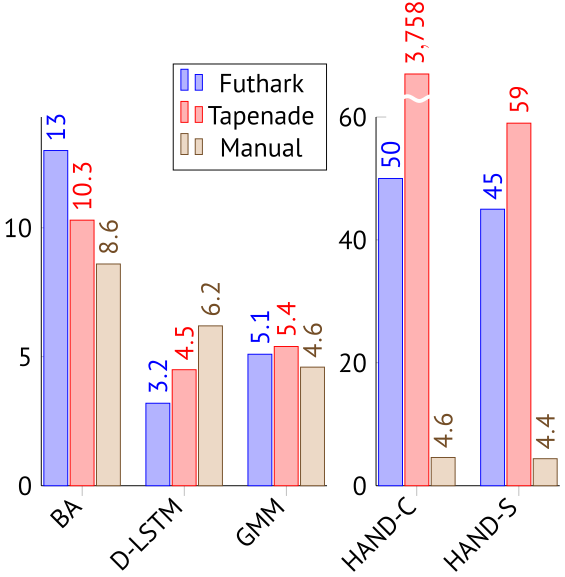
└Implementation



Implementation

- Implemented the AD transformation as a compiler pass in the **Futhark** compiler.
- Futhark is a high-level, nested-parallel, purely functional array language.
- The language described thus far is a close approximation of Futhark's IR.

CPU Benchmarks - ADBench



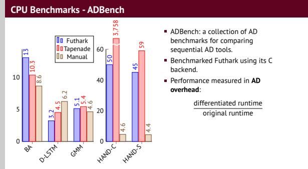
- ADBench: a collection of AD benchmarks for comparing sequential AD tools.
- Benchmarked Futhark using its C backend.
- Performance measured in **AD overhead**:

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

2024-06-16

└Benchmarks

└CPU Benchmarks - ADBench



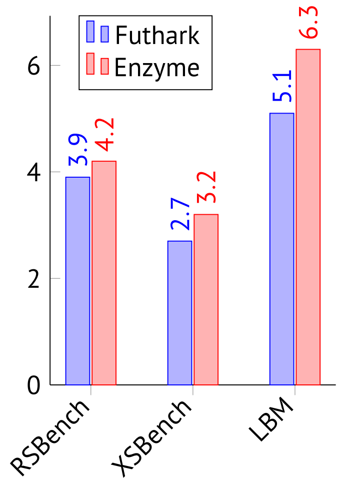
CPU Benchmarks - ADBench

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

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

- First up is ADBench, which is a suite of standardized sequential CPU benchmarks for AD implementations.
- The main metric here is *AD overhead* which is just the ratio of the runtimes of the differentiated program with the original program. Ideally, this ratio should always be a small constant.
- ADBench includes five diferent benchmarks and we show the AD overheads for Futhark and Tapenade, an AD tool for C, on the left.
- Also shown are overheads for manually differentiated implementations, which are hand-derived and optimized to be as fast as possible and don't use AD techniques.
- In the graphs we see that Futhark does very well in comparison to Tapenade in all

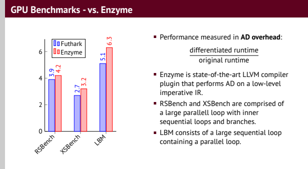
GPU Benchmarks - vs. Enzyme



- 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.



GPU Benchmarks - vs. Enzyme

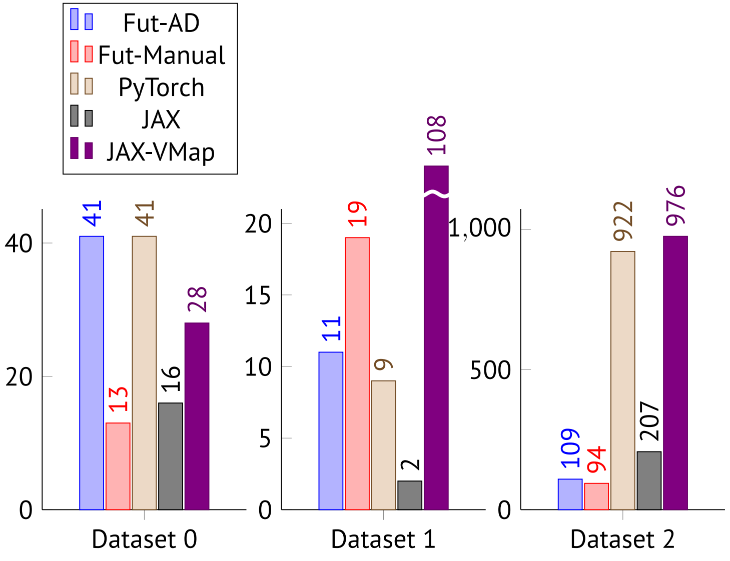
- Performance measured in **AD overhead**:

differentiated runtime / original runtime

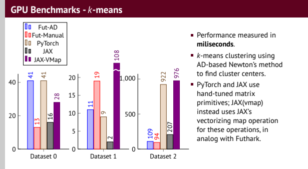
- Enzyme is state-of-the-art LLVM compiler plugin that performs AD on a low-level imperative IR.
- RSBench and XSBench are comprised of a large parallell loop with inner sequential loops and branches.
- LBM consists of a large sequential loop containing a parallel loop.

- Here we compare against Enzyme on the GPU, a state-of-the-art AD system which performs AD on low-level code.
- Futhark is competitive with Enzyme on all of the benchmarks.
- Shows that our high-level, simple re-computation based approach can be competitive with low level approaches.

GPU Benchmarks - *k*-means



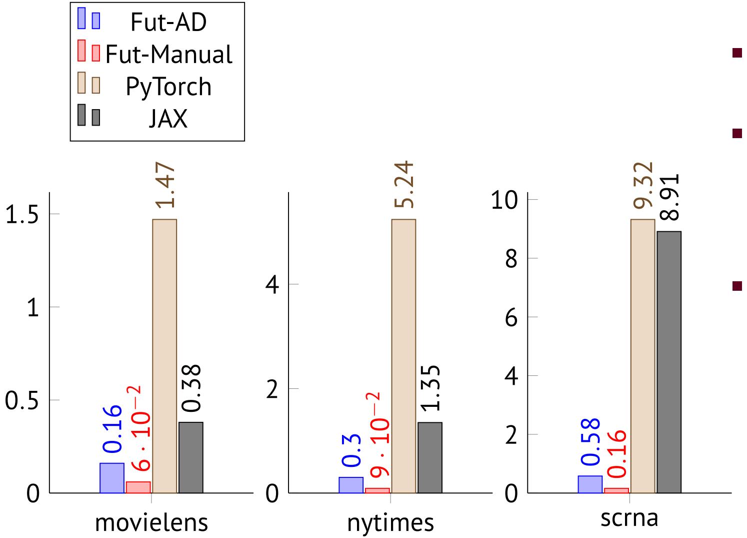
- Performance measured in **miliseconds**.
- *k*-means clustering using AD-based Newton's method to find cluster centers.
- PyTorch and JAX use hand-tuned matrix primitives; JAX(vmap) instead uses JAX's vectorizing map operation for these operations, in analog with Futhark.


GPU Benchmarks - *k*-means

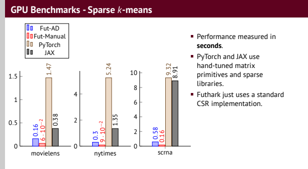
- Performance measured in **miliseconds**.
- *k*-means clustering using AD-based Newton's method to find cluster centers.
- PyTorch and JAX use hand-tuned matrix primitives; JAX(vmap) instead uses JAX's vectorizing map operation for these operations, in analog with Futhark.

- In this benchmark, we use AD to solve k-means clustering.
- We compare Futhark AD against the classic k-means algorithm in Futhark, denoted Fut-Manual, and against AD implementations in PyTorch and JAX.
- JAX and PyTorch rely array primitives to express computations and benefit from hand-written, highly optmized derivatives of these primitives.
- To better compare with Futhark's unrestricted model, we also compare against a version of JAX that only uses general parallel constructs which is denoted JAX-VMap.
- Futhark meets or exceeds the performance of PyTorch in all benchmarks and does significantly better than JAX with array primitives as datasets become larger.
- Without JAX's array primitives, Futhark is on par with or demonstrates magnitude-level speed-ups over JAX.

GPU Benchmarks - Sparse *k*-means

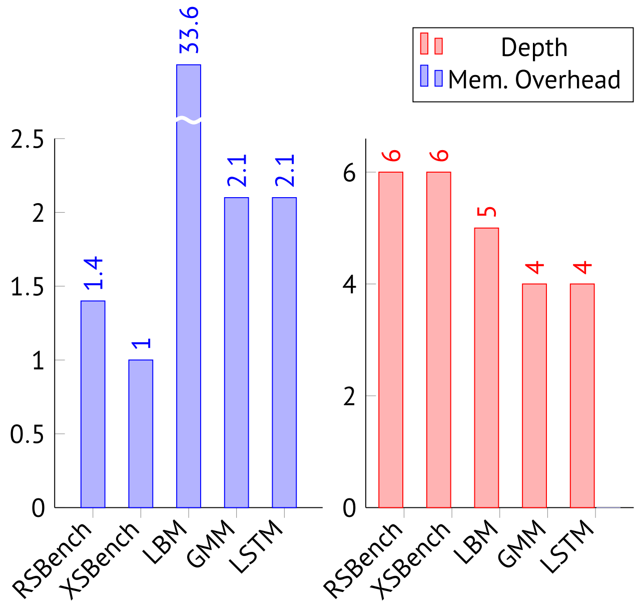


- Performance measured in **seconds**.
- PyTorch and JAX use hand-tuned matrix primitives and sparse libraries.
- Futhark just uses a standard CSR implementation.



- Here we benchmark k-means again, but this time with a sparse representation.
- We see that in all benchmarks, both AD and manual Futhark demonstrate very significant speed-up, often greater than a magnitude.

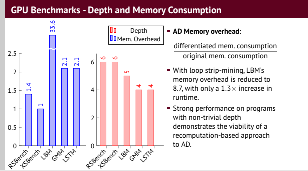
GPU Benchmarks - Depth and Memory Consumption



- **AD Memory overhead**:

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

- With loop strip-mining, LBM's memory overhead is reduced to 8.7, with only a 1.3× increase in runtime.
- Strong performance on programs with non-trivial depth demonstrates the viability of a recomputation-based approach to AD.



GPU Benchmarks - Depth and Memory Consumption

- AD Memory overhead: differentiated mem. consumption / original mem. consumption
- With loop strip-mining, LBM's memory overhead is reduced to 8.7, with only a 1.3× increase in runtime.
- Strong performance on programs with non-trivial depth demonstrates the viability of a recomputation-based approach to AD.

- AD memory overhead is just the ratio of the memory consumption of the differentiated program to the original program.
- We expect this ratio to be a small constant factor and indeed that is the case for most benchmarks, with the exception for LBM.
- LBM features a large outer sequential loop, so we have to store many loop parameters on a tape for each iteration of the loop.
- Using our strip-minig technique, we can decrease the memory overhead to under 10, with only a small 1.3 times increase in runtime.
- Plot in red shows the maximal depth of each benchmark, which shows that these benchmarks include non-trivial depth.

Conclusions

2024-06-16

└Conclusions

Conclusions

Conclusions

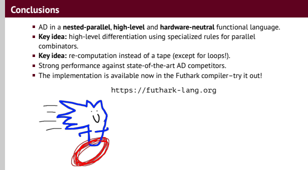
- AD in a **nested-parallel**, **high-level** and **hardware-neutral** functional language.
- **Key idea:** high-level differentiation using specialized rules for parallel combinators.
- **Key idea:** re-computation instead of a tape (except for loops!).
- Strong performance against state-of-the-art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

`https://futhark-lang.org`

2024-06-16

└Conclusions

└Conclusions



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