Adapton: Composable Demand-Driven Incremental Computation

Matthew A. Hammer
Khoo Yit Phang, Michael Hicks and Jeffrey S. Foster
Incremental Computation

Input

Application

IC Framework

Output
Incremental Computation

- Input
- Trace
- Output

Trace records
- dynamic data
- dependencies

Application

IC Framework
Incremental Computation

Run 1

Mutations
Incremental Computation

Run 1

Mutations

Inconsistencies

Input
Trace
Output

Input
Trace
Output
Incremental Computation

Run 1
Input
Trace
Output

Run 2
Input
Trace
Output

Change-Propagation
Incremental Computation

Run 1
Input
Trace
Output

Run 2
Input
Trace
Output

Change-Propagation
Updates
Incremental Computation

Run 1

Input

Trace

Output

Run 2

Input

Trace

Output

Change-Propagation

Updates
Incremental Computation
Incremental Computation

Run 1
Input
Trace
Output

Run 2
Input
Trace
Output
Incremental Computation

Run 2

Input

Trace

Output

Observations
Incremental Computation

Run 2

Mutations

Input

Trace

Output

loop..

Observations
Incremental Computation

Propagation respects program semantics:

Theorem: Trace and output are "from-scratch"-consistent

Equivalently: Change propagation is History independent
Existing Limitations
(self-adjusting computation)

- **Change propagation is eager**
  - Not driven by output observations

- **Trace representation**
  - = **Total ordering**
  - Limits reuse, excluding certain patterns

*Interactive* settings suffer in particular
Adapton: Composable, Demand-Driven IC

- Key concepts:
  - **Lazy thunks**: programming interface
  - **Demanded Computation Graph (DCG)**: represents execution trace

- Formal semantics, proven sound
- Implemented in OCaml (and Python)
- Speedups for all patterns (unlike SAC)
- Freely available at [http://ter.ps/adapton](http://ter.ps/adapton)
Interaction Pattern: **Laziness**

Do not (re)compute obscured sheets

Sheet A  Sheet B  Sheet C

(Independent sheets)

Legend

- **Consistent**
- **No cache**
- **Inactive**
Interaction Pattern: **Laziness**

Do not (re)compute obscured sheets

Sheet A  Sheet B  Sheet C

(Independent sheets)

Legend
- Consistent
- No cache
- Inactive
Interaction Pattern: **Laziness**

Do not (re)compute obscured sheets

Sheet A \quad Sheet B \quad Sheet C

(Independent sheets)

Legend

- **Consistent**
- **No cache**
- **Inactive**
Interactive Pattern: **Switching**

Demand / control-flow change

**Legend**
- **Consistent**
- **No cache**
- **Inactive**

Sheet \( C = f(A) \)
Interactive Pattern: **Switching**

Demand / control-flow change

**Legend**
- **Consistent**
- **No cache**
- **Inactive**

**Sheet A**

\[ C = f(A) \]

**Sheet B**

\[ C = g(B) \]

\[ \]
Interactive Pattern: **Switching**

Demand / control-flow change

Sheet A

Sheet B

Legend

- Consistent
- No cache
- Inactive

Sheet

\[ C = f(A) \]

\[ C = g(B) \]
Interaction Pattern: **Sharing**

B and C share work for A

\[ B = f(A) \]

\[ C = g(A) \]

Legend:
- **Consistent**
- **No cache**
- **Inactive**
Interaction Pattern: **Sharing**

B and C share work for A

---

**Legend**

- **Consistent**
- **No cache**
- **Inactive**

---

Sheet A

Sheet B = $f(A)$

Sheet C = $g(A)$
Interaction Pattern: **Sharing**

B and C share work for A

B = f(A)

C = g(A)

**Legend**

- Consistent
- No cache
- Inactive
Interactive Pattern: **Swapping**

Swaps input / evaluation order

Sheet A

Sheet B

Sheet

\[ C = f(A, B) \]

Legend

- **Consistent**
- **No cache**
- **Inactive**
Interactive Pattern: **Swapping**

Swaps input / evaluation order

**Legend**
- Consistent
- No cache
- Inactive

Sheet A

Sheet B

Sheet

\[ C = f(A, B) \]

\[ C = f(B, A) \]
Interactive Pattern: **Swapping**

Swaps input / evaluation order

**Legend**
- **Consistent**
- **No cache**
- **Inactive**

Sheet B

Sheet A

Sheet

\[ C = f(A, B) \]

\[ C = f(B, A) \]
Adapton’s Approach

• When we **mutate an input**, we mark dependent computations as **dirty**

• When we **demand a thunk**:
  • **Memo-match** equivalent thunks
  • **Change-propagation** repairs inconsistencies, **on demand**
Spread Sheet Evaluator

type cell = formula ref

and formula =
  | Leaf of int
  | Plus of cell * cell
Spread Sheet Evaluator

type cell = formula ref

and formula =
  | Leaf of int
  | Plus of cell * cell

Mutable

Depends on cells
type cell = formula ref
and formula =
  | Leaf of int
  | Plus of cell * cell

Example

let n₁ = ref (Leaf 1)
let n₂ = ref (Leaf 2)
let n₃ = ref (Leaf 3)
let p₁ = ref (Plus (n₁, n₂))
let p₂ = ref (Plus (p₁, n₃))
Spread Sheet Evaluator

type cell = formula ref
and formula =
  | Leaf of int
  | Plus of cell * cell

Example

let n1 = ref (Leaf 1)
let n2 = ref (Leaf 2)
let n3 = ref (Leaf 3)
let p1 = ref (Plus (n1, n2))
let p2 = ref (Plus (p1, n3))

“User interface” (REPL)
Spread Sheet Evaluator

Evaluator logic

eval : cell → (int thunk)
eval c = thunk ((
  case (get c) of
    | Leaf n ⇒ n
    | Plus(c1, c2) ⇒
      force (eval c1) +
      force (eval c2)
  )))

type cell = formula ref
and formula =
  | Leaf of int
  | Plus of cell * cell
Spread Sheet Evaluator

set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

type cell = formula ref

and formula =
  | Leaf of int
  | Plus of cell * cell
Spread Sheet Evaluator

[type] cell = formula ref

and formula =
| Leaf of int
| Plus of cell * cell

set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

Demands evaluation
Spread Sheet Evaluator

```
set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

type cell = formula ref

and formula =
  | Leaf of int
  | Plus of cell * cell
```
Spread Sheet Evaluator

```
set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

"User interface" (REPL)

[Diagram of a spreadsheet with nodes and operations]

type cell = formula ref

and formula =
  | Leaf of int
  | Plus of cell * cell

☞ let t₁ = eval p₁
```
Spread Sheet Evaluator

set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

let \( t_1 = \text{eval } p_1 \)
Spread Sheet Evaluator

```
set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

let t1 = eval p1

let t2 = eval p2
```
Spread Sheet Evaluator

set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

let t₁ = eval p₁

let t₂ = eval p₂

display t₁

demand!
Spread Sheet Evaluator

set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

let t₁ = eval p₁

let t₂ = eval p₂

display t₁

Demand Computation Graph (DCG)
Spread Sheet Evaluator

\[ \text{set : cell} \times \text{formula} \rightarrow \text{unit} \]

\[ \text{eval : cell} \rightarrow (\text{int thunk}) \]

\[ \text{display : (int thunk)} \rightarrow \text{unit} \]

“User interface” (REPL)

\[ \text{let } t_1 = \text{eval } p_1 \]

\[ \text{let } t_2 = \text{eval } p_2 \]

\[ \text{display } t_1 \]

\[ \text{display } t_2 \]

\[ \text{demand!} \]
Spread Sheet Evaluator

set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

let t₁ = eval p₁

let t₂ = eval p₂

display t₁

display t₂

Memo match!

Share
Spread Sheet Evaluator

set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

let t₁ = eval p₁
let t₂ = eval p₂
display t₁
display t₂
clear
Spread Sheet Evaluator

set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)
Spread Sheet Evaluator

- set : cell x formula → unit
- eval : cell → (int thunk)
- display : (int thunk) → unit

"User interface" (REPL)

☞ set n₁ ← Leaf 5
Spread Sheet Evaluator

set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

Dirty dep

Dirty phase

set n₁ ← Leaf 5
Spread Sheet Evaluator

set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

let t1 = eval p1

data display t1

data display t2

force

def set n1 ← Leaf 5

def display t1
Spread Sheet Evaluator

set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

let \( t_1 = \text{eval } p_1 \)

let \( t_2 = \text{eval } p_2 \)

display \( t_1 \)

display \( t_2 \)

Memo match!

force

get

set \( n_1 \leftarrow \text{Leaf 5} \)

display \( t_1 \)
Spread Sheet Evaluator

set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

let t₁ = eval p₁

let t₂ = eval p₂

display t₁

display t₂

set n₁ ← Leaf 5

display t₁
Spread Sheet Evaluator

set : cell x formula $\rightarrow$ unit

eval : cell $\rightarrow$ (int thunk)

display : (int thunk) $\rightarrow$ unit

“User interface” (REPL)

- set $n_1 \leftarrow$ Leaf 5
- display $t_1$
- set $p_2 \leftarrow$ Plus($n_3$, $p_1$)
Spread Sheet Evaluator

set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

set n₁ ← Leaf 5

display t₁

set p₂ ← Plus(n₃, p₁)
Spread Sheet Evaluator

set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

- set n₁ ← Leaf 5
- display t₁
- set p₂ ← Plus(n₃, p₁)

Dirty

Swap!
Spread Sheet Evaluator

set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

- set n₁ ← Leaf 5
- display t₁
- set p₂ ← Plus(n₃, p₁)
- display t₂
Spread Sheet Evaluator

set : cell x formula → unit

eval : cell → (int thunk)

display : (int thunk) → unit

“User interface” (REPL)

- set n₁ ← Leaf 5
- display t₁
- set p₂ ← Plus(n₃, p₁)
- display t₂

Swap!
**Spread Sheet Evaluator**

- **set**: cell x formula $\rightarrow$ unit
- **eval**: cell $\rightarrow$ (int thunk)
- **display**: (int thunk) $\rightarrow$ unit

**“User interface” (REPL)**

- set $n_1 \leftarrow$ Leaf 5
- display $t_1$
- set $p_2 \leftarrow$ Plus($n_3, p_1$)
- display $t_2$
Lazy Structures

Laziness generalizes **beyond scalars**

Recursive structures: **lists, trees and graphs**

```
type 'a lzlist =
| Nil
| Cons of 'a * ('a lzlist) thunk
```
Merging Lazy Lists
As in conventional lazy programming

```ocaml
let rec merge l1 l2 = function
  | l1, Nil    ⇒ l1
  | Nil, l2    ⇒ l2
  | Cons(h1,t1), Cons(h2,t2) ⇒
      if h1 <= h2 then
        Cons(h1, thunk(merge (force t1) l2))
      else
        Cons(h2, thunk(merge l1 (force t2)))
```

Merging Lazy Lists
Mergesort DCG Viz.

Graphics by Piotr Mardziel
Micro Benchmarks

List and tree applications:
filter, map
fold{min,sum}
quicksort, mergesort
expression tree evaluation
**Batch Pattern:** Experimental procedure:

- **Mutate random input**
- **Demand full output**

<table>
<thead>
<tr>
<th>Batch</th>
<th>Baseline time (s)</th>
<th>Adaption speedup</th>
<th>SAC speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>0.6</td>
<td>2.0</td>
<td>4.11</td>
</tr>
<tr>
<td>map</td>
<td>1.2</td>
<td>2.2</td>
<td>3.32</td>
</tr>
<tr>
<td>fold min</td>
<td>1.4</td>
<td>4350</td>
<td>3090</td>
</tr>
<tr>
<td>fold sum</td>
<td>1.5</td>
<td>1640</td>
<td>4220</td>
</tr>
<tr>
<td>exptree</td>
<td>0.3</td>
<td>497</td>
<td>1490</td>
</tr>
</tbody>
</table>
**Swap Pattern:** Experimental procedure:

Swap input halves

<table>
<thead>
<tr>
<th>Swap</th>
<th>Baseline time (s)</th>
<th>Adaptation speedup</th>
<th>SAC speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>0.5</td>
<td>2.0</td>
<td>0.14</td>
</tr>
<tr>
<td>map</td>
<td>0.9</td>
<td>2.4</td>
<td>0.25</td>
</tr>
<tr>
<td>fold min</td>
<td>1.0</td>
<td>472</td>
<td>0.12</td>
</tr>
<tr>
<td>fold sum</td>
<td>1.1</td>
<td>501</td>
<td>0.13</td>
</tr>
<tr>
<td>exptree</td>
<td>0.3</td>
<td>667</td>
<td>10</td>
</tr>
</tbody>
</table>
**Lazy Pattern:** Experimental procedure:

- **Mutate random input**
- **Demand first output**

<table>
<thead>
<tr>
<th>Lazy</th>
<th>Baseline time (s)</th>
<th>Adapton speedup</th>
<th>SAC speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>1.16E-05</td>
<td>12.8</td>
<td>2.2</td>
</tr>
<tr>
<td>map</td>
<td>6.86E-06</td>
<td>7.8</td>
<td>1.5</td>
</tr>
<tr>
<td>quicksort</td>
<td>7.41E-02</td>
<td>2020</td>
<td>22.9</td>
</tr>
<tr>
<td>mergesort</td>
<td>3.46E-01</td>
<td>336</td>
<td>0.148</td>
</tr>
</tbody>
</table>
Switch Pattern: Experimental procedure:

1. Remove
2. Insert
3. Toggle order

<table>
<thead>
<tr>
<th>Switch</th>
<th>Baseline time (s)</th>
<th>Adaptation speedup</th>
<th>SAC speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>updown1</td>
<td>3.28E-02</td>
<td>22.4</td>
<td>2.47E-03</td>
</tr>
<tr>
<td>updown2</td>
<td>3.26E-02</td>
<td>24.7</td>
<td>4.28</td>
</tr>
</tbody>
</table>
Spreadsheet Experiments

Sheet 1

Sheet 2

Random binary formula
Spreadsheet Experiments

Sheet 1

Sheet 2

Random binary formula

Fixed Depth
Spreadsheet Experiments

Random binary formula

Sheet 1

Sheet 2

Fixed Depth

1. Random Mutations
Spreadsheet Experiments

1. Random Mutations
2. Observe last sheet

Random binary formula
Spreadsheet Experiments

Random binary formula

Sheet 1

Sheet 2

Fixed Depth

1. Random Mutations

2. Observe last sheet
Speedup vs # Changes
(15 sheets deep)

Number of Changes

Speedup

Adaption
SAC
59 uses memoization to avoid unnecessary recomputation when computing.

Related Work

Incremental computation.

The chief aim of RP is similar to self-adjusting computation in that it is a recent approach to I) that uses a positional model that supports several key incremental patterns—hence are still of limited use for interactive settings, as discussed in prior work in that it is dynamically determined by an outer-layer computation rather than pre-determined by the prior inner-layer computation. Hoover and Teitelbaum optimize change propagation for dynamic graphs with a special form of memoization, making the adaptation strategy similar to ours and treats “demand” as coming from their inputs are unchanged. Hudson describes a lazy change propagation strategy similar to ours and treats “demand” as coming from a convention-looking language and tracks dynamic dependency graphs easier to write and reason about and better performance.

However, the mechanism is potentially inefficient: demand-driven functions are sometimes unnecessarily re-evaluated, even if their inputs are unchanged. In contrast, in most prior I) techniques, self-adjusting computation tolerates store-based differences between changes.

Some prior I) work is motivated at least in part by being cognizant of dynamic graphs with a special form of memoization. Incorporating memoization limits reuse of prior computation. Prior work in that it is dynamically determined by an outer-layer computation, but it is always the attribute grammar based on keys “demanded” from an aggregate data structure during a particular computation, but it is always the data structure.

The right plot shows that even for a fixed-sized spreadsheet, the naive implementation resulting in slowdowns that worsen as the number of changes grows. The benefit of — again offers no incremental benefit and the performance is exponentially with more sheets. (y contrast, our measurements show that with only four sheets, the performance to that of naive, stateless evaluation)

In both cases, more sheets and more changes — figure 1 shows the performance of this test script. In the left plot, we consistently measure slowdowns of exponentially —s with the left plot, the number of changes varies.

We plot an average of eight randomized runs for each coordinate.

Scramble-all

Scramble-one

Speedup

Speed vs # Changes

Adapton

SAC

Exponential Speedup

100x Slowdown!
Spreadsheet Experiments

1. Random Mutations

2. Observe last sheet
Spreadsheet Experiments

1. Random Mutations
   - Random binary formula

2. Observe
   - Vary Depth

   - 1. Random Mutations
   - 2. Observe last sheet
While some commonalities with incremental computation, many approaches were promising but many are still assume a totally ordered monolithic trace representation and view of past computation simply cannot handle these patterns. The semantics of the inner and outer layers working in concert is based on maintaining a single totally ordered positional model that supports several key incremental patterns—eager, total order, eager total order, and for some changes a randomly chosen cell to a random value.

Researchers later combined these dynamic graphs with a special form of memoization making the approach of incorporating memoization limiting reuse of prior computation rather than general-purpose computation. His strategy also does not have no explicit notion of outer-layer demand but applied to attribute grammars during change propagation are only reevaluated when called but applied to attribute grammars during change propagation are only reevaluated when called.

Some prior I/O work is motivated at least in part by being cognizant of some commonalities with incremental computation, many approaches were promising but many are still assume a totally ordered monolithic trace representation and view of past computation simply cannot handle these patterns. The semantics of the inner and outer layers working in concert is based on maintaining a single totally ordered positional model that supports several key incremental patterns—eager, total order, eager total order, and for some changes a randomly chosen cell to a random value.

The idea of memoization—improving efficiency of computation when an input signal is updated due to an event like a key press or simply some commonalities with incremental computation, many approaches were promising but many are still assume a totally ordered monolithic trace representation and view of past computation simply cannot handle these patterns. The semantics of the inner and outer layers working in concert is based on maintaining a single totally ordered positional model that supports several key incremental patterns—eager, total order, eager total order, and for some changes a randomly chosen cell to a random value.
The right plot shows that even for a fixed-sized spreadsheet as the number of changes grows, the benefit of -s with the left plot, we consistently measure slowdowns of exponentially —s with the left plot, we consistently measure slowdowns of exponentially for more sheets (y contrast, our measurements show that with only four sheets, the performance to that of naive stateless evaluation. The left plot shows that as the relative speedup/slowdown of -fifteen, and the number of changes varies. In both plots, we show fixed at ten, in the right plot, the number of sheets is fixed at -10 changes) Num. Changes (15 Sheets) Exponential

- Figure . shows the performance of this test script. In the left Speedup vs Sheet Depth figure .: -

- Speedup (over Naive)

- AS

- SAC

- Exponential Speedup

- 100x Slowdown!

- Hoover proposes a programming system of incremental computation, simulates a user loading dense spreadsheet with random choices made by both the naive implementation, resulting in slowdowns that worsen as exponentially for more sheets (y contrast, our measurements show that with only four sheets, the performance to that of naive stateless evaluation. The left plot shows that as the relative speedup/slowdown of -fifteen, and the number of changes varies. In both plots, we show fixed at ten, in the right plot, the number of sheets is fixed at -10 changes) Num. Changes (15 Sheets) Exponential

- Figure . shows the performance of this test script. In the left Speedup vs Sheet Depth figure .: -

- Speedup (over Naive)

- AS

- SAC

- Exponential Speedup

- 100x Slowdown!
Paper and Technical Report

- *Formal semantics* of Adapton
- *Algorithms* to implement Adapton
- More empirical data and analysis
Aside: Formal Semantics

- **CBPV + Refs + Layers** *(outer versus inner)*
- Syntax for **traces** and **knowledge** formally represents **DCG** structure
- Formal specification of change propagation

**Theorems:**

- **Type soundness**
- **Incremental soundness** *(“from-scratch consistency”)*
Summary

- **Adapton**: Composable, Demand-Driven IC
  - *Demand-driven* change propagation
  - Reuse patterns: *Sharing, swapping and switching*
- Formal specification (see paper)
- Implemented in OCaml (and Python)
- Empirical evaluation shows *speedups*

[http://ter.ps/adapton](http://ter.ps/adapton)
<table>
<thead>
<tr>
<th>pattern</th>
<th>input #</th>
<th>LazyNonInc baseline</th>
<th>ADAPTON vs. LazyNonInc</th>
<th>EagerTotalOrder vs. LazyNonInc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>time (s)</td>
<td>mem (MB)</td>
<td>time spdup</td>
</tr>
<tr>
<td>filter lazy</td>
<td>1e6</td>
<td>1.16e-5</td>
<td>96.7</td>
<td>12.8</td>
</tr>
<tr>
<td>map lazy</td>
<td>1e6</td>
<td>6.85e-6</td>
<td>96.7</td>
<td>7.80</td>
</tr>
<tr>
<td>quicksort lazy</td>
<td>1e5</td>
<td>0.0741</td>
<td>18.6</td>
<td>2020</td>
</tr>
<tr>
<td>mergesort lazy</td>
<td>1e5</td>
<td>0.346</td>
<td>50.8</td>
<td>336</td>
</tr>
<tr>
<td>filter swap</td>
<td>1e6</td>
<td>0.502</td>
<td>157</td>
<td>1.99</td>
</tr>
<tr>
<td>map swap</td>
<td>1e6</td>
<td>0.894</td>
<td>232</td>
<td>2.36</td>
</tr>
<tr>
<td>fold(min) swap</td>
<td>1e6</td>
<td>1.04</td>
<td>179</td>
<td>472</td>
</tr>
<tr>
<td>fold(sum) swap</td>
<td>1e6</td>
<td>1.11</td>
<td>180</td>
<td>501</td>
</tr>
<tr>
<td>exptree swap</td>
<td>1e6</td>
<td>0.307</td>
<td>152</td>
<td>667</td>
</tr>
<tr>
<td>updown1 switch</td>
<td>4e4</td>
<td>0.0328</td>
<td>8.63</td>
<td>22.4</td>
</tr>
<tr>
<td>updown2 switch</td>
<td>4e4</td>
<td>0.0326</td>
<td>8.63</td>
<td>24.7</td>
</tr>
<tr>
<td>filter batch</td>
<td>1e6</td>
<td>0.629</td>
<td>157</td>
<td>2.04</td>
</tr>
<tr>
<td>map batch</td>
<td>1e6</td>
<td>1.20</td>
<td>232</td>
<td>2.21</td>
</tr>
<tr>
<td>fold(min) batch</td>
<td>1e6</td>
<td>1.43</td>
<td>179</td>
<td>4350</td>
</tr>
<tr>
<td>fold(sum) batch</td>
<td>1e6</td>
<td>1.48</td>
<td>180</td>
<td>1640</td>
</tr>
<tr>
<td>exptree batch</td>
<td>1e6</td>
<td>0.308</td>
<td>152</td>
<td>497</td>
</tr>
</tbody>
</table>