

より清浄なStream Fusion

小林 友明 Oleg Kiselyov
(所属：東北大学)

目次

- ・ 背景
- ・ 提案手法
- ・ 結果
- ・ まとめ

Stream Fusionとは

- ・ ストリーム = 長さに制約のない系列データ (無限長を許す)
 - ・ 例: Webのログデータ, IoTのセンサーデータ
- ・ 関数型のストリーム処理 = **演算子**によるパイプライン
 - ・ 例: 二乗和のパイプライン

```
of_arr (int_array arr)
```

```
|> map (fun x -> x * x)
```

```
|> fold (fun z x -> z + x) (int 0)
```

$$\sum_i x_i^2$$

Stream Fusionとは

- ・ ストリーム = 長さに制約のない系列データ(無限長を許す)
 - ・ 例: Webのログデータ, IoTのセンサーデータ

- ・ 関数型のストリーム処理 = **演算子**によるパイプライン

- ・ 例: 二乗和のパイプライン

```
of_arr (int_array arr)
```

```
|> map (fun x -> x * x)
```

```
|> fold (fun z x -> z + x) (int 0)
```


$$\sum_i x_i^2$$

- ・ **中間データ構造**のオーバーヘッド

=> 演算子をfusion(融合)して**中間データ構造なしの処理へ変換**

=> **stream fusion**

Stream Fusionを利用するストリームライブラリ

-  **strymonas**^[Kiselyov et al., POPL 2017]
 - OCaml, Scalaをサポート
 - **処理の実行前にstream fusionされたコードを生成・利用**
 - 例: 二乗和のパイプラインから生成されるOCamlのコード

```
let lv_1 = arr in
let lv_2 = (Array.length lv_1) - 1 in
let lv_3 = ref 0 in
for i_4 = 0 to lv_2 do
  let el_5 = Array.get lv_1 i_4 in
  let lv_6 = el_5 * el_5 in
  lv_3 := !lv_3 + lv_6
done;
!lv_3
```

Stream Fusionを利用するストリームライブラリ

- ・ 複雑なパイプライン

```
zip_with pair  
  (of_arr (int_array array)  
    |> map (fun x -> x * x)  
    |> take (int 12)  
    |> filter (fun x -> x mod (int 2) = int 0)  
    |> map (fun x -> x * x))  
  (iota (int 1)  
    |> flat_map (fun x -> iota (x + int 1) |> take (int 3))  
    |> filter (fun x -> x mod (int 2) = int 0))  
|> fold (fun z x -> cons x z) nil
```

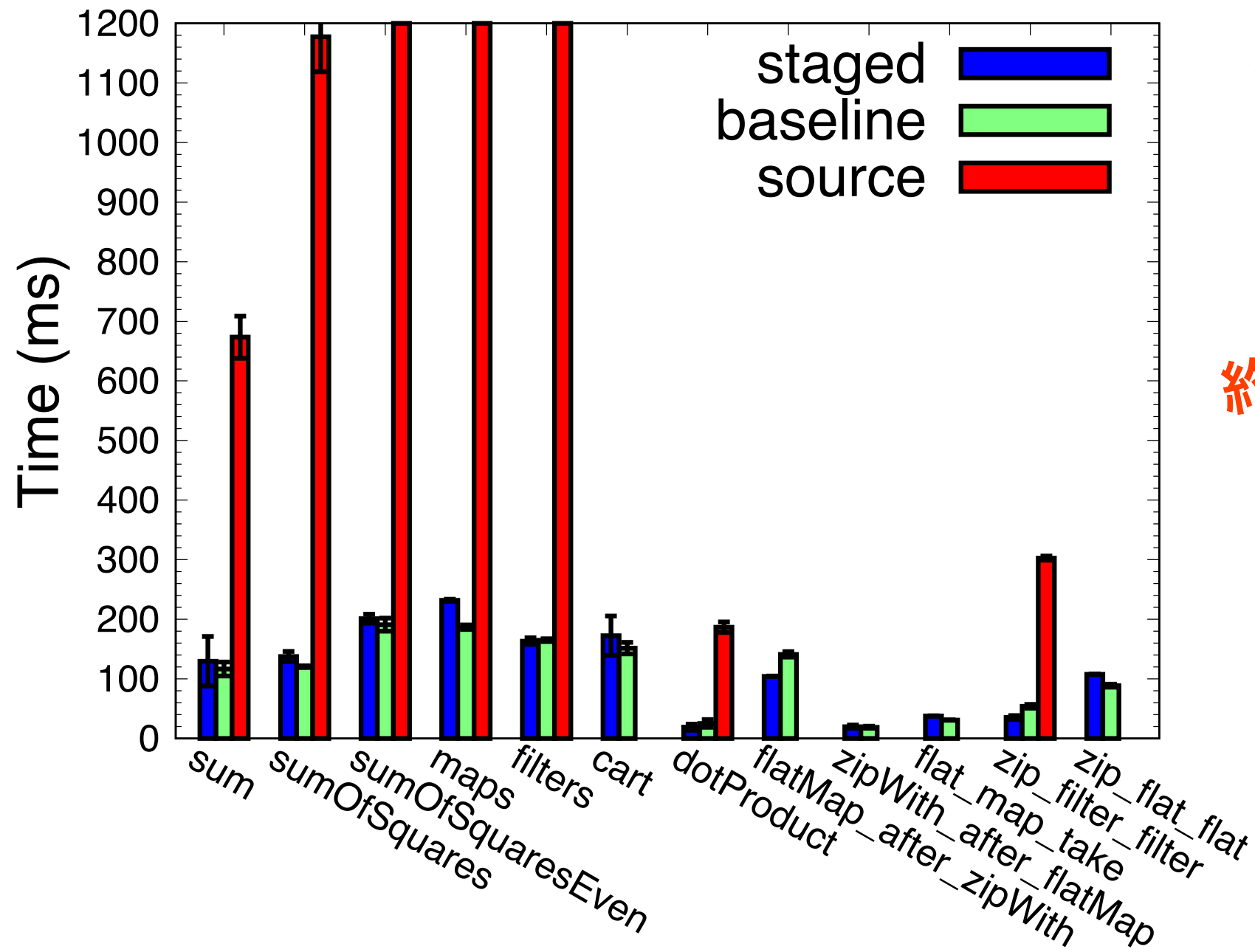
- ・ **strymonas**: 上記のようなパイプラインで中間データ構造が残留
=> 本研究: **strymonas**を改善

改善後のStrymonas: 生成コード

```
let lv_1 = array in let lv_2 = (Array.length lv_1) - 1 in
let lv_3 = ref [] in let lv_4 = ref 0 in let lv_5 = ref 1 in
while (! lv_4) <= (if 11 < lv_2 then 11 else lv_2) do
  let lv_6 = ! lv_5 in incr lv_5;
  let lv_7 = ref (lv_6 + 1) in let lv_8 = ref 0 in
  while ((! lv_8) <= 2) && ((! lv_4) <= (if 11 < lv_2 then 11 else lv_2)) do
    incr lv_8; let lv_9 = ! lv_7 in incr lv_7;
    if (lv_9 mod 2) = 0 then
      let lv_10 = ref true in
      while (! lv_10) && ((! lv_4) <= (if 11 < lv_2 then 11 else lv_2)) do
        let el_11 = Array.get lv_1 (! lv_4) in
        let lv_12 = el_11 * el_11 in
        if (lv_12 mod 2) = 0 then
          let lv_13 = lv_12 * lv_12 in
          incr lv_4; lv_10 := false; lv_3 := ((lv_13, lv_9) :: (! lv_3))
        else (incr lv_4)
      done
    else ()
  done
done;
! lv_3
```

中間的なデータ構造(バッファ, タプル, etc.)や
クロージャは一切ない

改善後のStrymonas: パフォーマンス



後で詳しく紹介

約5 ~ 25倍高性能

strymonas v2

≈ 手書き

≫ streaming

- 改善後のstrymonas (v2)
- 手書きの命令的なコード
- streaming [\[odis-labs.github.io/streaming\]](https://github.com/odis-labs/streaming)

当初のStrymonasの特徴と問題

- 特徴

- 生成コード: **型安全性・スコープ安全性と処理系非依存な効率性の保証**
- 殆どの**パイプライン: **命令型の手書きストリーム処理と同等の性能**

- 問題

- 一般に, 完全なzipやflat_mapのfusionは難しい

=> strymonas: **flat_mapされたストリーム対のzipで不完全なfusion**

```
zip_with f
```

```
  (stream_1 |> flat_map prod_1 |> ...)
```

```
  (stream_2 |> flat_map prod_2 |> ...)
```

```
|> ...
```

- strymonas: filterをflat_mapで実装

=> **問題の影響範囲は広い**

本研究の貢献

- ・ この問題を解決する (BER Meta) OCaml 版 strymonas の再実装
 - ・ zip と flat_map を完全に fusion
 - => **任意の** パイプライン: 命令型の手書きストリーム処理と同等の性能
- ・ その他の改善
 - ・ コード生成の中間言語の洗練化
 - => **内部実装やユーザー用APIの実装が簡単に**
 - ・ 対象言語の抽象化, コード生成機能と最適化機能の分離
 - => **多言語のコード生成へ対応, ステージ化注釈の隠蔽, 最適化の強化**
 - => **ストリーム処理ジェネレータとしての側面の強化 (Yacc の比喩)**

目次

- ・ 背景
- ・ **提案手法**
- ・ 結果
- ・ まとめ

提案手法

- `zip_flat_flat`^[Parreaux et al., GPCE 2017]の生成コードで説明

```
zip_with (fun x y -> x + y)
  (of_arr (int_array arr1)
    |> flat_map (fun x -> of_arr (int_array arr2)
                  |> map (fun y -> x * y)))
  (of_arr (int_array arr2)
    |> flat_map (fun x -> of_arr (int_array arr1)
                  |> map (fun y -> x - y)))
|> take (int 20_000_000)
|> fold (fun z x -> z + x) (int 0)
```

問題解決前の生成コード: クロージャ

```
let ... in let curr_6 = ref None in let nadv_7 = ref None in
let adv_16 () =
  curr_6 := None;
  while ... do
    match ! nadv_7 with
    | Some adv_8 -> adv_8 ()
    | None ->
      let el_9 = arr_4.(! i_5) in ... let i_11 = ref 0 in ...
      let adv1_15 () =
        if (! i_11) <= ((Array.length arr_10) - 1) then
          let el_13 = arr_10.(! i_11) in
          let t_14 = el_9 * el_13 in
          (incr i_11; curr_6 := (Some t_14))
        else nadv_7 := old_adv_12 in
      nadv_7 := (Some adv1_15); adv1_15 ()
    done in
  ...
while ... do
  ...
  while ... do
    ...
    match ! curr_6 with
    | Some el_26 -> (adv_16 (); ...; s_3 := ((! s_3) + (el_26 + t_25)))
  done
done;
! s_3
```

クロージャ

本体

問題解決前の生成コード: option型の値

```
let ... in let curr_6 = ref None in let nadv_7 = ref None in
let adv_16 () =
  curr_6 := None;
  while ... do
    match ! nadv_7 with
    | Some adv_8 -> adv_8 ()
    | None ->
      let el_9 = arr_4.(! i_5) in ... let i_11 = ref 0 in ...
      let adv1_15 () =
        if (! i_11) <= ((Array.length arr_10) - 1) then
          let el_13 = arr_10.(! i_11) in
          let t_14 = el_9 * el_13 in
          (incr i_11; curr_6 := (Some t_14))
        else nadv_7 := old_adv_12 in
      nadv_7 := (Some adv1_15); adv1_15 ()
    done in
  ...
while ... do
  ...
  while ... do
    ...
    match ! curr_6 with
    | Some el_26 -> (adv_16 (); ...; s_3 := ((! s_3) + (el_26 + t_25)))
  done
done;
! s_3
```

参照セル中にNone

= 未初期化のミュータブル変数

本体

問題解決後の生成コード: クロージャ

```
let ... in
let lv_10 = ref (Obj.obj (Obj.new_block 0 0)) in
let lv_12 = ref (Obj.obj (Obj.new_block 0 0)) in
let ... in
while ... do
```

```
  ...
  while ... do
```

```
    ...
    while ... do
```

```
      if not (! lv_9) then
```

```
        (let el_25 = Array.get arr2 (! lv_8) in
```

```
          incr lv_8; lv_10 := el_25; lv_12 := 0; lv_9 := true);
```

```
      if ! lv_9 then
```

```
        (if (! lv_12) <= lv_11 then
```

```
          let el_23 = Array.get arr1 (! lv_12) in
```

```
          let lv_24 = (! lv_10) - el_23 in
```

```
          incr lv_12; lv_22 := false; incr lv_13;
```

```
          lv_7 := ((! lv_7) + (lv_21 + lv_24))
```

```
        else lv_9 := false)
```

```
      done
```

```
    done
```

```
  done;
```

```
  ! lv_7
```

自由変数の中身が大域的に保存
= クロージャ変換

クロージャ
の生成

クロージャ
の呼び出し

問題解決後の生成コード: option型の値

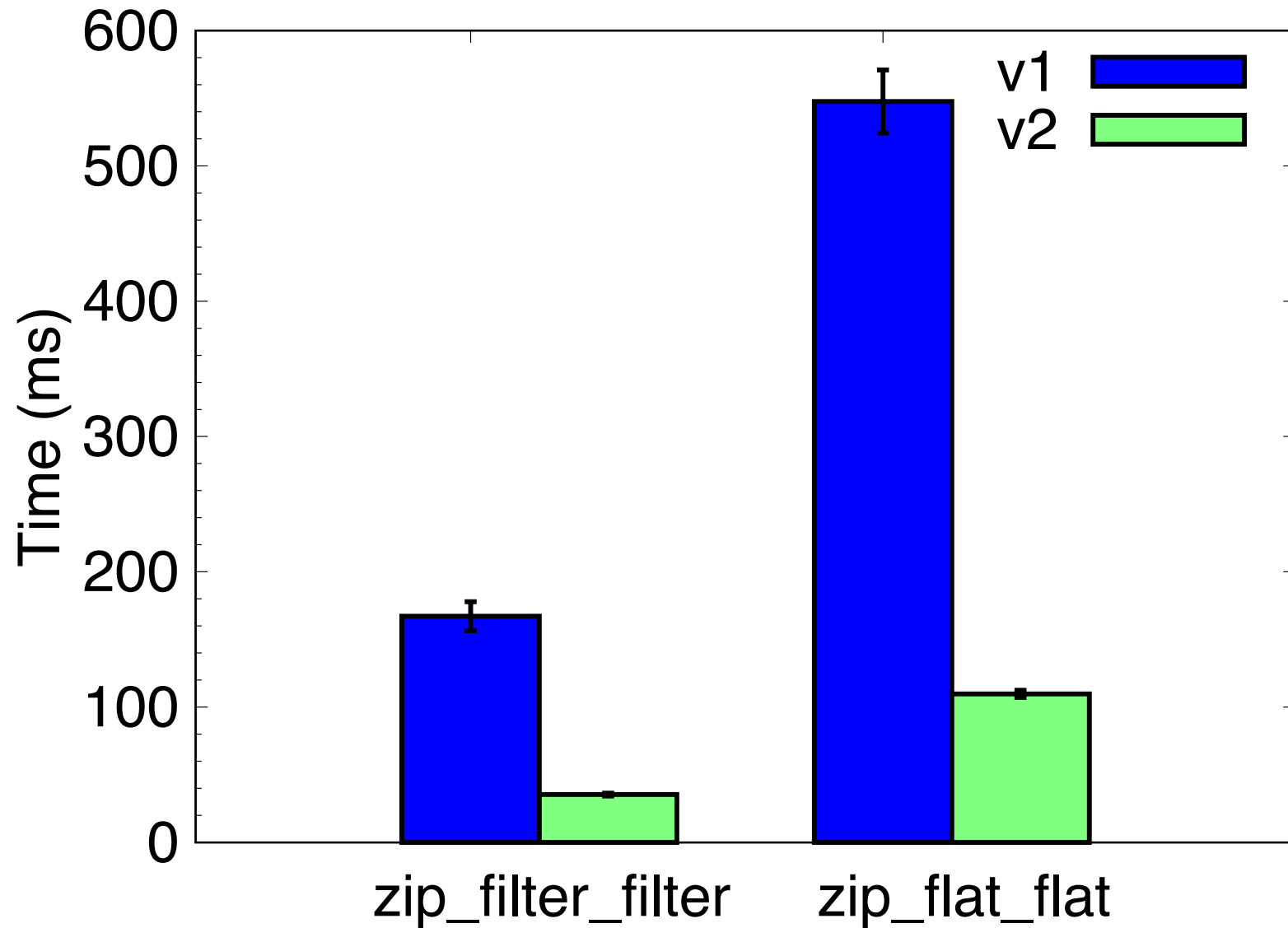
```
let ... in
let lv_10 = ref (Obj.obj (Obj.new_block 0 0)) in
let lv_12 = ref (Obj.obj (Obj.new_block 0 0)) in
let ... in
while ... do
  ...
  while ... do
    ...
    while ... do
      if not (! lv_9) then
        (let el_25 = Array.get arr2 (! lv_8) in
         incr lv_8; lv_10 := el_25; lv_12 := 0; lv_9 := true);
      if ! lv_9 then
        (if (! lv_12) <= lv_11 then
          let el_23 = Array.get arr1 (! lv_12) in
          let lv_24 = (! lv_10) - el_23 in
          incr lv_12; lv_22 := false; incr lv_13;
          lv_7 := ((! lv_7) + (lv_21 + lv_24))
          else lv_9 := false)
        done
      done
    done;
    ! lv_7
```

Objによる未初期化の
ミュータブル変数

目次

- ・ 背景
- ・ 提案手法
- ・ **結果**
- ・ まとめ

ベンチマーク1

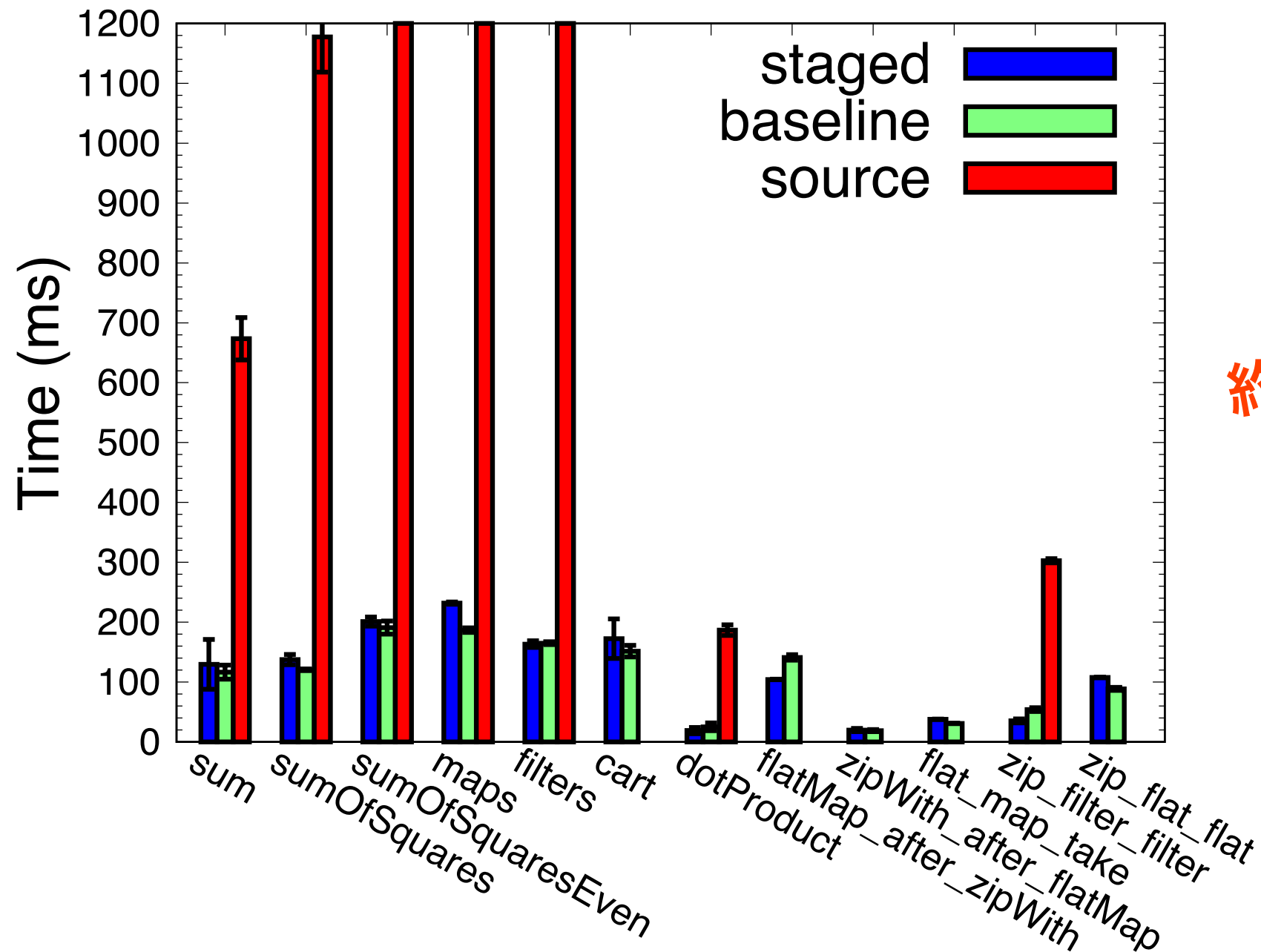


約5倍高性能

strymonas v2
→ > strymonas v1

- 当初のstrymonas (v1)
- 改善後のstrymonas (v2)

ベンチマーク2



約5 ~ 25倍高性能

strymonas v2

≈ 手書き

≫ streaming

- 改善後のstrymonas (v2)
- 手書きの命令的なコード
- streaming [\[odis-labs.github.io/streaming\]](https://github.com/odis-labs/streaming)

処理の記述方法の変化: 二乗和のパイプライン

strymonas v1

```
of_arr .<arr>.  
|> map (fun x -> .<~x * ~x>.)  
|> fold (fun z x -> .<~z + ~x>.) .<0>.
```

- ・ OCamlのコード生成: **ステージ化注釈**

処理の記述方法の変化: 二乗和のパイプライン

strymonas v1

```
of_arr .<arr>.  
|> map (fun x -> .<~x * ~x>.)  
|> fold (fun z x -> .<~z + ~x>.) .<0>.
```

ユーザーから隠蔽

- OCamlのコード生成: **ステージ化注釈**
- 多言語のコード生成: **専用のDSL**

strymonas v2

```
of_arr (int_array arr)  
|> map (fun x -> x * x)  
|> fold (fun z x -> z + x) (int 0)
```

生成コードの品質の変化: 二乗和のパイプライン

strymonas v1

```
let lv_1 = arr in
let lv_3 = ref 0 in
for i_4 = 0 to (Array.length lv_1) - 1 do
  let el_5 = Array.get lv_1 i_4 in
  let lv_6 = el_5 * el_5 in lv_3 := !lv_3 + lv_6
done;
!lv_3
```

生成コードの品質の変化: 二乗和のパイプライン

strymonas v1

```
let lv_1 = arr in  
let lv_3 = ref 0 in  
for i_4 = 0 to (Array.length lv_1) - 1 do  
— let el_5 = Array.get lv_1 i_4 in  
— let lv_6 = el_5 * el_5 in lv_3 := !lv_3 + lv_6  
done;  
!lv_3
```

strymonas v2

```
let lv_1 = arr in  
let lv_2 = (Array.length lv_1) - 1 in  
let lv_3 = ref 0 in  
for i_4 = 0 to lv_2 do  
  let el_5 = Array.get lv_1 i_4 in  
  let lv_6 = el_5 * el_5 in lv_3 := !lv_3 + lv_6  
done;  
!lv_3
```

効率性の保証範囲の拡大

let挿入

目次

- ・ 背景
- ・ 提案手法
- ・ 結果
- ・ **まとめ**

まとめ

- ・ strymonasの問題の解決
 - => クロージャと中間データ構造を例外なく完全に除去
 - => より清浄なstream fusion
- ・ その他の改善
 - ・ 実装の簡単化
 - ・ 多言語のコード生成に対応
 - ・ 効率性の保証範囲の拡大
- ・ 今後の課題
 - ・ 中間言語を低水準言語(CやLLVM)向けに最適化してコンパイル
 - ・ window処理の実現
 - ・ TensorFlow等へstrymonasのアプローチを応用