Efficient Monadic Streams

Josef Svenningsson, Anders Persson, Emil Axelsson,
and Peter Jonsson

Chalmers University of Technology
Ericsson
SICS Swedish ICT AB

FELDSPAR

FELDSPAR

Functional Embedded Language for
Digital Signal Processing and Parallelism

FELDSPAR

- Embedded in Haskell
- Generates C
- Aims to raise the level of abstraction for the programming

- Streams for signal processing

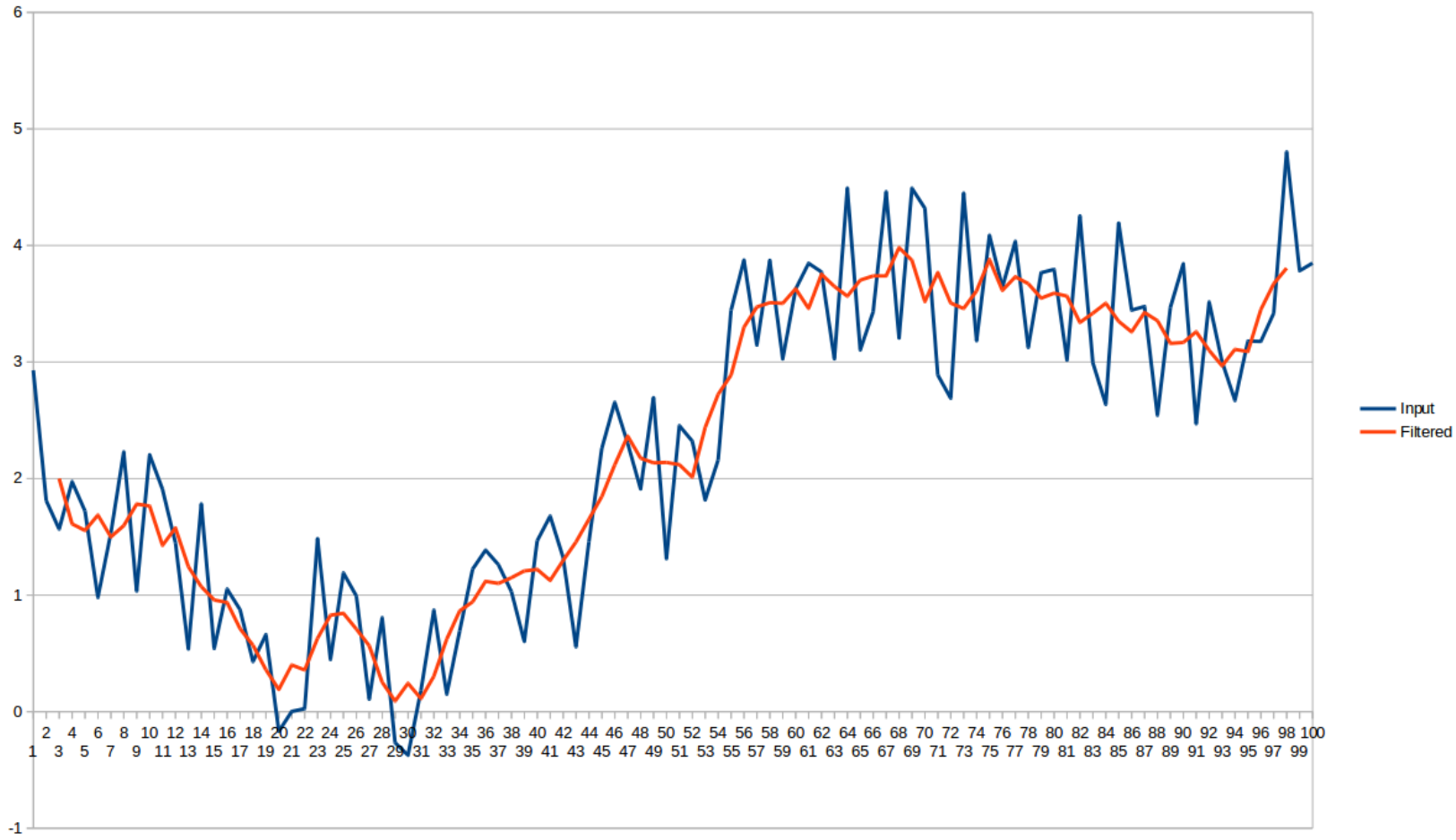
```
data Stream a where
   Stream :: (s -> (a,s)) -> s -> Stream a
```

Compositional list-like interface

```
mapStream :: (a -> b) -> Stream a -> Stream b
mapStream f (Stream next i) = Stream next' i
  where next' s = case next s of
                    (a,s') -> (f a,s')

zip :: Stream a -> Stream b -> Stream (a,b)
zip (Stream next1 i1) (Stream next2 i2) = Stream next (i1,i2)
  where next (s1,s2) = case (next1 s1,next2 s2) of
                         ((a,s1'),(b,s2')) -> ((a,b),(s1',s'2))
```

- The functional stream representation supports *fusion*
- Simply unfold function definitions
- Allows for compositional programming without performance loss
- This transformation happens as a side-effect of how Feldspar is embedded in Haskell.
  No extra implementation effort needed.

```
mapStream f (mapStream g (Stream next init))

==> { unfolding mapStream }

mapStream f (Stream next' init)
  where next' s = case next s of (a,s') -> (g a,s')

==> { unfolding mapStream }

Stream next'' init
  where next' s = case next s of (a,s') -> (g a,s')
        next'' s = case next' s of (b,s') -> (f b,s')

==> { unfolding next' and evaluation }

Stream next'' init
  where next'' s = case next s of (a,s') -> (f (g a),s')
```


6
5
4
3
2
1
0
-1
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
Input
Filtered

```haskell
movingAvg :: Fractional a => Data Int -> Stream a -> Stream a
movingAvg n (Stream step init) = Stream step' init'
  where
    init' = (init,replicate n 0)
    step' (s,window) =
      let (a,s')  = step s
          window' = init window ++ singleton a
      in (avg window, (s',window'))
    avg w = sum (elems w) / fromIntegral n
```

```haskell
movingAvg :: Fractional a => Data Int -> Stream a -> Stream a
movingAvg n (Stream step init) = Stream step' init'
  where
    init' = (init,replicate n 0)
    step' (s,window) =
      let (a,s')  = step s
          window' = init window ++ singleton a
      in (avg window, (s',window'))
    avg w = sum (elems w) / fromIntegral n
```

- The sliding `window` copied each iteration
- Costly when `window` grow large

```
1     for (uint32_t v47 = 0; v47 < 32; v47 += 1)
2     {
3       //Code for computing the average of the window elided
4       for (uint32_t v89 = 0; v89 < v184; v89 += 1)
5       {
6         ((v46).member2).member2[(v63 + 1)] =
7           ((v48).member2).member2[v63];
8       }
9       //Code updating the struct which holds the window elided
10    }
```

```
1   for (uint32_t v47 = 0; v47 < 32; v47 += 1)
2   {
3     //Code for computing the average of the window elided
4     for (uint32_t v89 = 0; v89 < v184; v89 += 1)
5     {
6       ((v46).member2).member2[(v63 + 1)] =
7         ((v48).member2).member2[v63];
8     }
9     //Code updating the struct which holds the window elided
10  }
```

- Line 4-8 is a loop which copies the window

We would like a representation where we can avoid copying the buffer.

Monads to the rescue

```haskell
data Stream a = Stream (M (M a))
```

```
data Stream a = Stream (M (M a))
```

- The monad `M` in Feldspar is similar to `IO` in Haskell
- Provides mutable references and arrays

```haskell
enumFrom :: Data Int -> Stream Int
enumFrom n = Stream $ do
  r <- newRef n
  loop $ do
    i <- readRef r
    writeRef r (i+1)
    return i
```

```
1  enumFrom :: Data Int -> Stream Int
2  enumFrom n = Stream $ do
3    r <- newRef n
4    loop $ do
5      i <- readRef r
6      writeRef r (i+1)
7      return i
```

- Line 3: Initialization
- Lines 5-7: Loop body
- `loop` = `return`

```haskell
data Stream a = Stream (M (M a))
```

- The outer monad performs initialization
- The inner monad is the loop body.

```haskell
mapStream :: (a -> b) -> Stream a -> Stream b
mapStream f (Stream init) = Stream $ do
  body <- init
  loop $ do
    a <- body
    return (f a)

zip :: Stream a -> Stream b -> Stream (a,b)
zip (Stream init1) (Stream init2) = Stream $ do
  next1 <- init1
  next2 <- init2
  loop $ do
    a <- next1
    b <- next2
    return (a,b)
```

Fusion

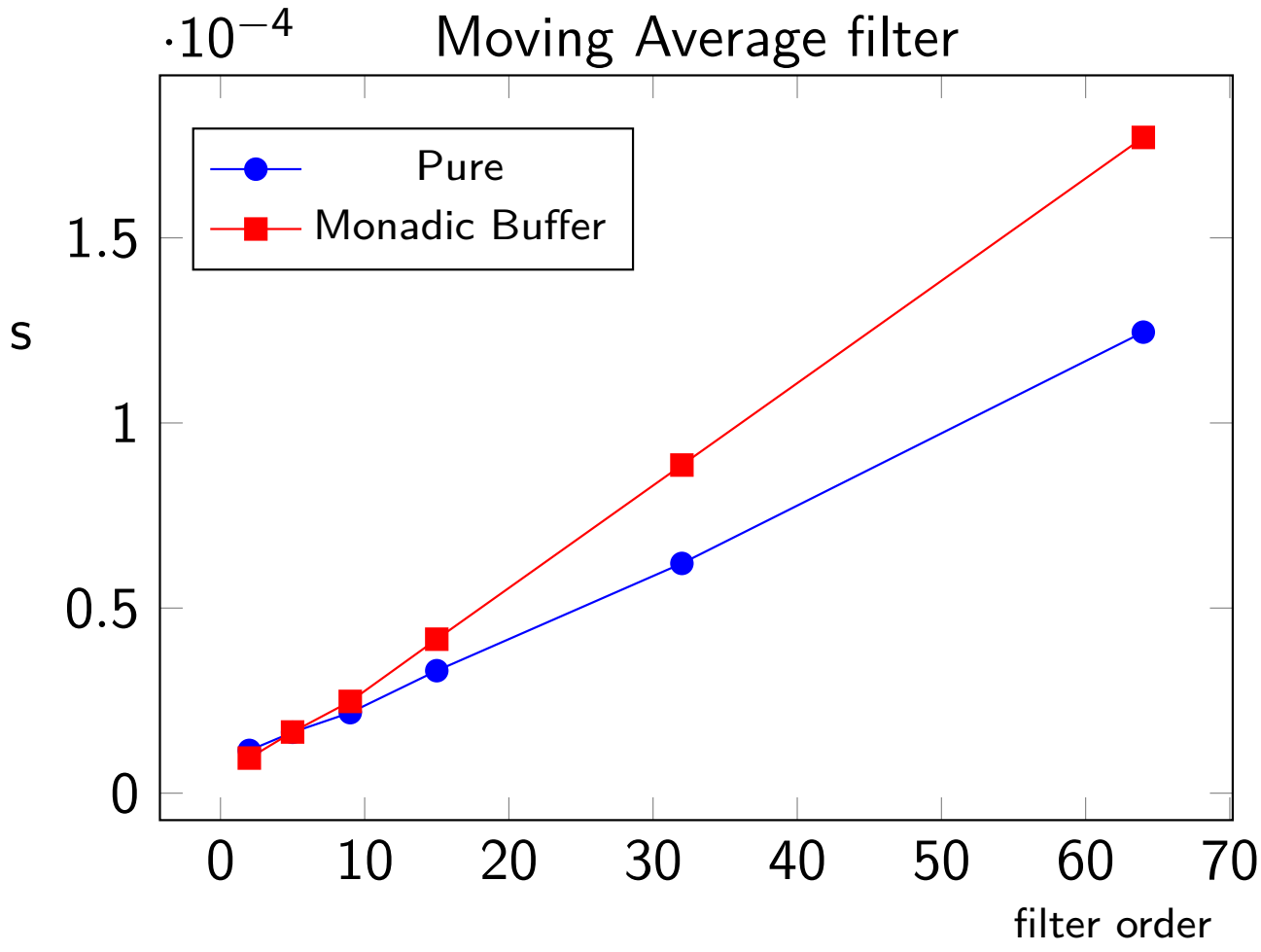
- The new stream type also supports fusion
  - Unfolding of definitions, just like in the functional representation
  - Two of the monad laws: associativity and left identity
- In Feldspar, the two monad laws are also applied as a side-effect of the way it is embedded in Haskell

```haskell
movingAvg :: Int -> Stream Double -> Stream Double
movingAvg n s =
  recurrence (listArray (0,n-1) (replicate n 0.0)) s
             (\input -> sum (elems input) / fromIntegral n)
```

```
recurrence :: Array Int a -> Stream a ->
              (Array Int a -> b) ->
              Stream b
recurrence ii (Stream init) mkExpr = Stream $ do
    next <- init
    ibuf <- initBuffer ii
    loop $ do
      a <- next
      putBuf ibuf a
      b <- withBuf ibuf $ \ib ->
             mkExpr ib
      return b
```

```
1     v14 = 0;
2     copy(v7, zeros);
3     for (uint32_t v24 = 0; v24 < 32; v24 += 1) {
4       v48 = v0[v25];
5       v27 = v14;
6       v14 = ((v27 + 1) % 8);
7       v7[v27] = v48;
8       // Code computing the average elided
9     }
```

- No copying
- Line 7 updates the window


Moving Average filter
$\cdot 10^{-4}$
Pure
Monadic Buffer
s
1.5
1
0.5
0
0
10
20
30
40
50
60
70
filter order

- The cyclic buffer uses **modulus** for each element
- More expensive than copying

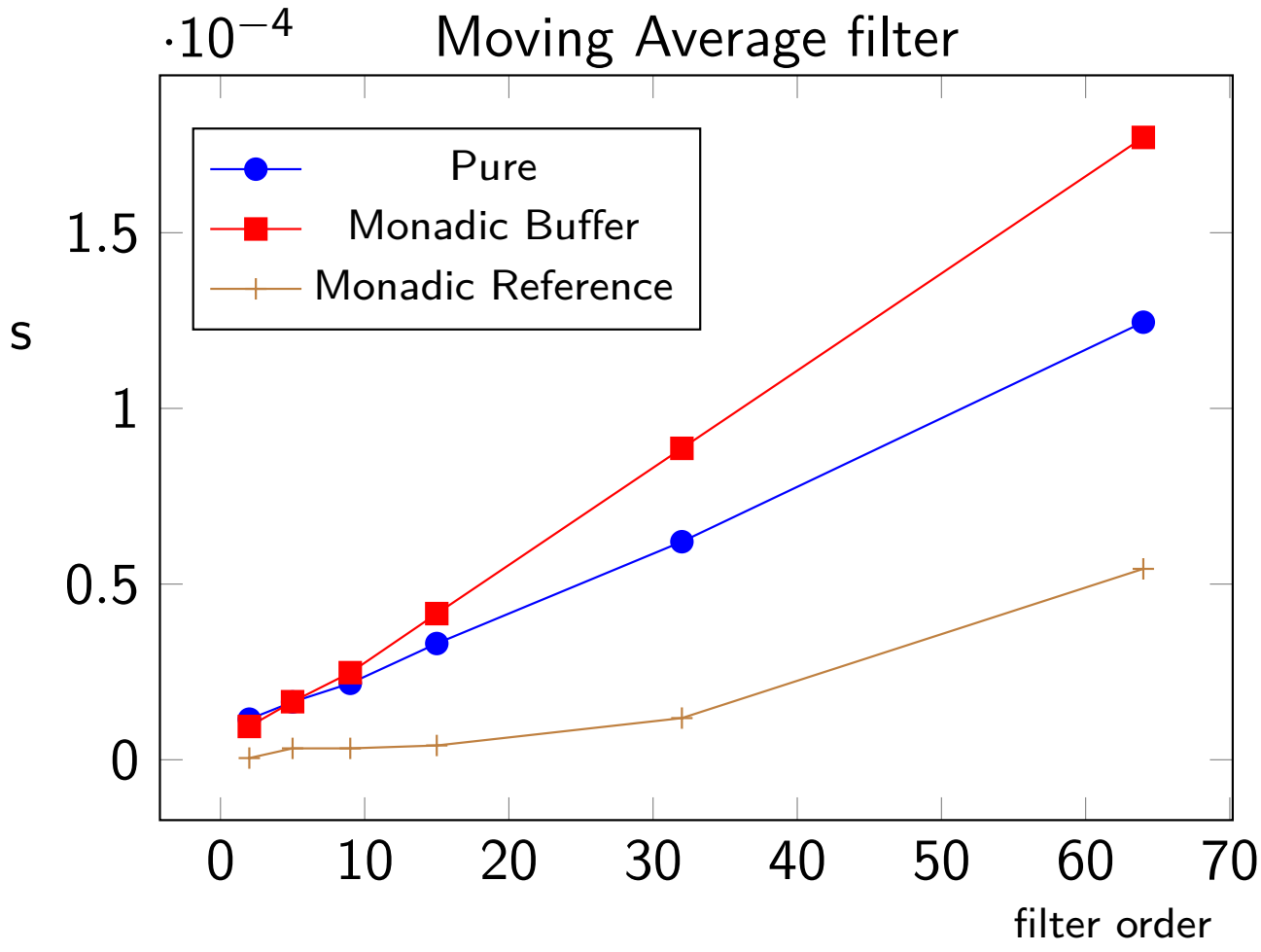
Making a faster filter

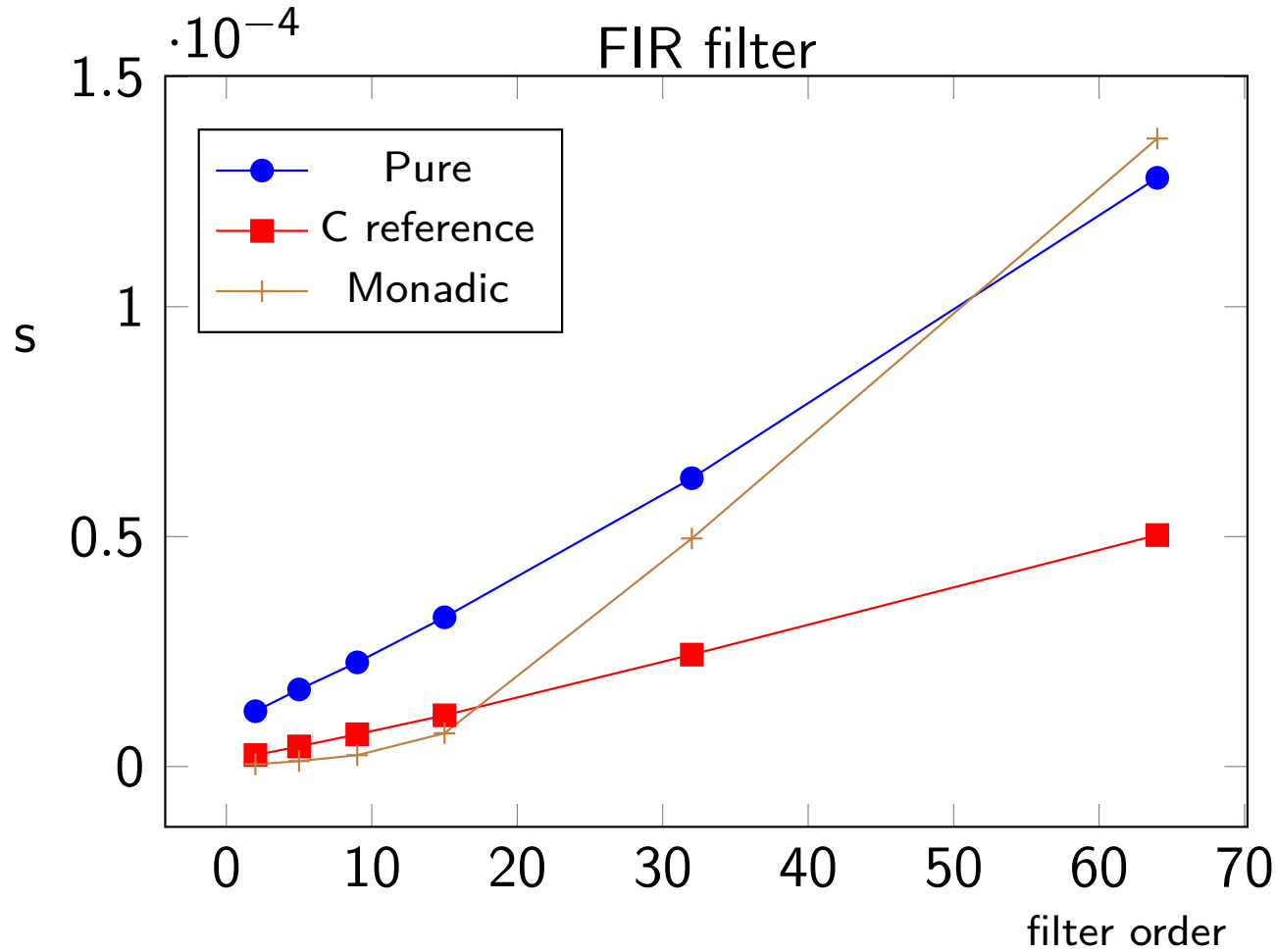
Although the mutable buffer was slower, mutable streams allow for many variations.

For instance:

- Unroll the buffer to keep each element in a reference
- Effectively means that each element is kept in a register

```
recurrenceS :: (Type a, Type b) =>
               [Data a] -> Stream (Data a) ->
               ([Data a] -> Data b) ->
               Stream (Data b)
recurrenceS ii (Stream init) mkExpr = Stream $ do
    next <- init
    ris  <- mapM newRef ii
    loop $ do
      a <- next
      if (not $ null ii) then pBuf ris a else return ()
      b <- wBuf ris $ \ib ->
             return $ mkExpr ib
      return b
  where
    pBuf rs a = zipWithM (\r1 r2 -> getRef r1 >>= setRef r2)
                         (tail $ reverse rs) (reverse rs)
             >> setRef (head rs) a
    wBuf rs f = mapM getRef rs >>= f
```


Moving Average filter
$\cdot 10^{-4}$
Pure
Monadic Buffer
Monadic Reference
s
1.5
1
0.5
0
0
10
20
30
40
50
60
70
filter order


FIR filter
$\cdot 10^{-4}$
1.5
1
0.5
0
s
Pure
C reference
Monadic
0
10
20
30
40
50
60
70
filter order

Conclusion

- A simple, versatile stream representation
- Enables mutation for efficiency
- A declarative, compositional interface
- Well suited for embedded languages
- Fusion can happen for free with the right embedding

Unexpectedly

- Mutation is not always faster

In the paper

- Avoiding multiple loop variables