Pull and Push arrays, Effects and Array Fusion

Josef Svenningsson
Chalmers University of Technology

Edinburgh University 2014-11-18

Parallel Functional Array Representations

Efficiency

This talk will center around efficient array representations.

An array operation is *efficient* if it supports:

- Fusion
- No memory allocation
- Parallellizeable
- No conditionals inside generated loops

Additional bonus points for:

- No use of `div` or `mod`

Pull arrays

```
data Pull a = Pull (Int -> a) Int
```

- Each element is computed independently
  - The consumer decide how to schedule the computations
  - Makes it easy to parallelize
- Well known, used in many different libraries and formalisms

Pull arrays

Some example functions

```
map :: (a -> b) -> Pull a -> Pull b
map f (Pull ixf l) = Pull (f. ixf) l

sum :: Num a => Pull a -> a
sum (Pull ixf l)
    = forLoop l 0 (\i sum ->
          sum + ixf i
      )

zipWith :: (a -> b -> c) -> Pull a -> Pull b -> Pull c
zipWith f (Pull ixfa la) (Pull ixfb lb)
  = Pull (\i -> f (ixfa i) (ixfb i))
          (min la lb)
```

Pull array fusion

Scalar product:

```
scProd :: Num a => Pull a -> Pull a -> a
scProd a b = sum (zipWith (*) a b)
```

Pull array fusion

Scalar product:

```
scProd :: Num a => Pull a -> Pull a -> a
scProd a b = sum (zipWith (*) a b)
```

Fusion:

```
sum (zipWith (*) (Pull ixfa la) (Pull ixfb lb)) =

sum (Pull (\i -> ixfa i * ixf b) (min la lb)) =

forLoop (min la lb) 0 (\i sum ->
  sum + (ixfa i * ixfb b)
)
```

Pull arrays in depth

Representing arrays as functions from index to elements has some advantages:

- Fusion
- Compositional style of programming
- Many efficient functions

Pull concatenation

```
(++) :: Pull a -> Pull a -> Pull a
Pull ixf1 l1 ++ Pull ixf2 l2 = Pull ixf (l1+l2)
  where ixf i = if i < l1
                then ixf1 i
                else ixf2 (i - l1)
```

This definition is not *efficient* because it contains a conditional which will be tested each iteration in the inner loop.

Problems

Not all functions are efficient:

- Concatenation
  - Results in a conditional in the inner loop
- Producing more than one element at a time
  - Impossible since each element is computed independently

Push arrays

```
data Push a = Push ((Int -> a -> M ()) -> M ()) Int
```

- `M` is some monad which provides:
    - mutable updates
    - parallelism
- Intuition:
    - Push arrays are programs which write an array to memory,
    - Parameterized by how to write each element to memory
- The producer decides the scheduling order, consumers must be able to handle any order

Push arrays

`Push` arrays solve the shortcomings of `Pull` arrays

- Provides efficient concatenation

```
Push p l ++ Push q m = Push r (l+m)
  where r w = do p w
                 q (\i a -> w (i+l) a)
```

- Can write several elements at once

```
dup (Push p l) = Push q (2*l)
  where q w = p (\i a -> w (2*i) a >> w (2*i+1) a)
```

- Have the same strong fusion guarantees

Duality of Push and Pull

	Pull	Push
Permutations	Ok	Ok
Functor	Ok	Ok
Splitting	Ok	Nope
Zipping	Ok	Nope
Concatenating	Nope	Ok
Multiple writes in loop body	Nope	Ok

Ok	Means operations are *efficiently* implementable.
Nope	Means I'm fairly sure there are no *efficient* implementations.

Combining Pull and Push

- It is efficient to convert from Pull arrays to Push arrays
  It means coming up with a schedule for the Push array.

```
pullToPush :: Pull a -> Push a
pullToPush (Pull ixf l) = Push f l
  where f k =  parM l $ \i ->
                 k (ixf i) i
```

- Converting from Push array to Pull arrays requires memory
  Goes from a completely scheduled representation to a random access representation.

```
force :: Push a -> M (Pull a)
force (Push f l) =
  do marr <- newArray_ (0,l-1)
     f (\a i -> writeArray marr i a)
     arr <- freeze marr
     return (Pull (\i -> arr!i) l)
```

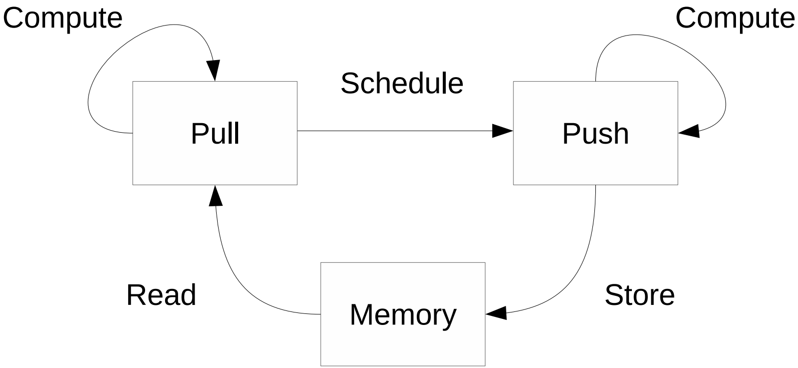
Array transformations

It's cheap to convert `Pull -> Push` but expensive to convert `Push -> Pull`

This suggests a way to structure array computations.

- Start with Pull. Compute, compute, compute.
- Stop when we encounter an operation which cannot be done efficiently with Pull
- Convert to Push. Compute, compute, compute.
- At some point, we can't keep it as Push any longer and store to memory.

Array transformations



Array transformations

```
type ArrTrans a b = Pull a -> Push b
```

- Guarantee: All array transformations are *efficient* when built from operations on `Pull` and `Push` arrays.
- Composing transformations seem to require storing to memory

Example: Stencil computations

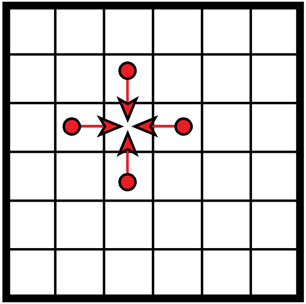
Previously, stencil computations has been modelled by Pull arrays.

- Repa has a very intricate version of pull arrays to accomodate efficient stencil computations.
  However, not all stencil computations can be efficiently implemented using Repa. Only *gather* stencils can be implemented.
- Efficient stencil computations are implemented like this:
  - Divide up input array in sections, perform stencil computations, concatenate all sections.
  - Corresponds very well to array transformations:
    Start with Pull arrays which can be efficiently split
    End with Push arrays which can be efficiently concatenated

Stencil computations

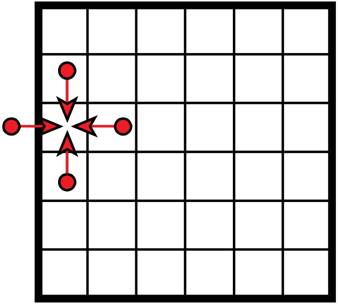
Stencil example (gather stencil):

- Each element of the new array is computed from four elements of the old array



Stencil

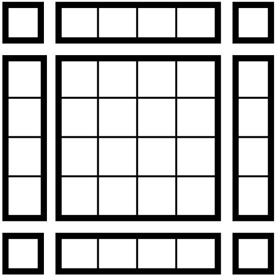
Stencil computations: boundary



Stencil boundary

Stencil computations: split

Using different computations for different regions allows for computing the large center region without testing for the edge cases.



Splitting arrays for efficiency

Stencil computations: split

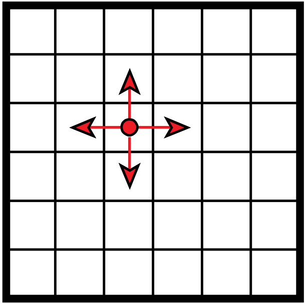
Splitting arrays fits array transformations perfectly

- Splitting is efficiently done using `Pull` arrays
- Reading the individual elements is done from a `Pull` array
- Caching previous reads can be achieved using `Push` arrays
- Reassembling the array corresponds to concatenation, which is done efficiently using `Push` arrays

Scatter stencils

Accomodating scatter stencils is easy

- `Push` arrays can be used to write several array elements at once.



Scatter stencil

Combining gather and scatter

Array transformations can even deal with combined gather and scatter stencils

- Example: image scaling

Image scaling

Stencil computations functionally

Stencil computations have been thoroughly explored using Pull arrays.

- Repa
- Ypnos

Two downsides of these approaches

- Only deals with *gather* stencils.
- Efficient implementations needs much more complicated version of Pull arrays

An advantage array transformations is that they can handle both *gather* and *scatter*.

Scatter stencils are important when the size of the array change.

Summary: Pull and Push

- Pull and Push are two complementary array representations
  - Parallelizeable
  - Supports fusion
- Array transformations
  - `type ArrTrans a b = Pull a -> Push b`
  - Very general and useful notion of array computations
    - Stencil
    - FFT
  - Composing array transformations means allocating memory because we're going from `Push` to `Pull`

Arrays as (co)monads

Pull arrays are comonads

```
class Comonad c where
  counit :: c a -> a
  cobind :: (c a -> b) -> c a -> c b

instance Comonad Pull where
  counit (Pull ixf l) = ixf 0
  cobind f vec@(Pull _ l) = Pull (\ix -> f (rotateL ix vec)) l

rotateL :: Int -> Pull a -> Pull a
rotateL i (Pull ixf l) = Pull (\ix -> ixf ((ix + i) `mod` l)) l
```

Definition is *Efficient*.

Cobind

1	2	3	4

is transformed into

1	2	3	4
2	3	4	1
3	4	1	2
4	1	2	3

A more efficient Pull array comonad

```
data Pull a = Pull (Int -> a) Int Int
```

The extra `Int` is a cursor pointing into the array.

```
instance Comonad Pull where
  counit (Pull ixf _ focus) = ixf focus
  cobind f (Pull ixf l focus)
    = Pull (\ix -> f (Pull ixf l ix)) l focus
```

- The `focus` is only relevant for the `Comonad` instance
- Other operations just ignore the `focus` (or pass it along).

Push arrays are monads

```
instance Monad Push where
  return a = Push (\k -> k a 0) 1
  p >>= f = bindP p f

bindP :: Push a -> (a -> Push b) -> Push b
bindP (Push ixf l) k = Push ixf' l'
  where ixf' k' = do r <- newIORef 0
                     ixf (\ a i ->
                           do n <- readIORef r
                              let (Push ixf'' l'') = k a
                              ixf'' (\b j -> k' b (j+n))
                              modifyIORef r (+l'')
                         )
        l' = unsafePerformIO $
             do r <- newIORef 0
                ixf (\ a _ ->
                      do let (Push _ l'') = k a
                         modifyIORef r (+l'')
                    )
                readIORef r
```

Push arrays are monads

The definition of `bindP` is only semi-efficient.

- In order to parallelize it, requires a parallel prefix sum
- Requires allocating an intermediate array, the size of the outer array.

Can be made efficient by using static sizes.

- Requires indexed monads and comonads

Nesting:

- It requires nested parallel loops - problematic on some target architectures (e.g. GPU)

(Co)Monads and stencils

People have previously noted the relationship between:

Comonads	Gather stencils
Monads	Scatter stencils

Composing array transformations

What Would Category Theorists Do?

We know that Pull arrays are Comonads and Push arrays are Monads.

Can we use these facts somehow?

Combining Monads and Comonads

Category

```
(.) :: (c b -> m d) -> (c a -> m b)
    -> (c a -> m d)
```

Can this operator be implemented *efficiently*?

It would be a way to compose array transformations.

Combining Monads and Comonads

How category theorist implement composition

```
(.) :: (Monad m, Comonad c) =>
       (c b -> m d) -> (c a -> m b) -> (c a -> m d)
f . g = join . fmap f . distribute . fmap g . cojoin
```

The crucial bit is the `distribute` function.

```
distribute :: Pull (Push a) -> Push (Pull a)
```

Can we implement this function?

Distribute

Two lines of attack for implementing `distribute`.

```
distPull :: C1 m => Pull (m a) -> m (Pull a)

distPush :: C2 c => c (Push a) -> Push (c a)
```

Distribute II

```
distPull :: ParM m => Pull (m a) -> m (Pull a)
distPull (Pull ixf l) = parM l $ \i -> do
                                   a <- ixf i
                                   return (Pull (\ix -> a) 1)
```

This definiton is *efficient*.

Gives rise to nested parallel loops.

Distribute laws

Power and Watanabe list the following laws for `distribute`.

1. distribute . fmap join = join . fmap distribute . distribute
2. distribute . fmap return = return
3. distribute . fmap distribute . cojoin = fmap cojoin . distribute
4. counit = fmap counit . distribute

- The laws hold if we flatten nested arrays and compare them element wise
- Different parallelization choices

Composing Array Transformation

Given that `Pull` and `Push` distribute we can compose array transformations (semi-) *efficiently*.

```
(.) :: ArrTrans b c -> ArrTrans a b -> ArrTrans a c
```

- Stencil fusion!

Conclusions

- Push and Pull are dual; Monads and Comonads
- Push and Pull distribute
- Array transformations `Pull a -> Push b` are *efficient*
- Array transformations can be (semi-) efficiently composed

Future work:

- Explore the full consequence of distributivity
- Pull and Push arrays generalize to multi-dimensional arrays. How do array transformations and distribution handle higher dimensions?
- Proofs

Thanks

Some pictures from Wikipedia, CC-BY-SA 3.0