Fizz and Fuse Your Arrays

Mike Vollmer

Indiana University

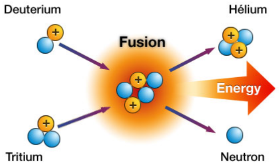
PL Wonks, 23 October 2015

Fission and Fusion

- **Fusion** combines operations on arrays.
- **Fission** splits up operations on arrays.

What is Fusion?

Functional programs often allocate lots of intermediary data structures.
Fusion is an optimization that addresses this problem.



Removing Intermediary Structures

For example, these two map calls:

```
(map f (map g ls))
```

The inner (map g ls) is executed first, and it allocates a list of g applied to every element of ls.
If we compose f and g, we get:

```
(let ([f (lambda (x) (f (g x)))])
  (map f ls))
```

An Array at a Time

Programming with arrays is convenient.

```
(define scalar-product
  (lambda (a b)
    (vector-sum (vector-map * a b))))
```

But it can be troublesome. What is the problem with this function?

Pull Arrays

What if we **change the representation** of our arrays?
Instead of representing arrays directly as Scheme vectors, we can use a function from indices to values.

```
(define pull
  (lambda (f len)
    `(pull ,f ,len)))
```

- Each element of an array is computed independently, allowing for easy parallelization.
- The consumer of a `pull array` can decide how to schedule the computations.

Pull Arrays

Converting pull arrays to scheme vectors is easy, using `build-vector`.

```
(define pull->vector
  (lambda (arr)
    (match arr
      [`(pull ,f ,len)
       (build-vector len f)])))
```

We can also go the other way:

```
(define vector->pull
  (lambda (vect)
    (let ([f (lambda (i) (vector-ref vect i))]
          [len (vector-length vect)])
      (pull f len))))
```

Pull Arrays

Now, map is very simple! Maybe.

```
(define map
  (lambda (f arr)
    (match arr
      [‘(pull ,g ,len)
       (let ([h (lambda (i) (f (g i)))])
         (pull h len))])))
```

And it looks more than a bit like our early example of fusion.

Pull Arrays

Like map, but with two array arguments.

```
(define zipwith
  (lambda (f arr1 arr2)
    (match-let
        ([`(pull ,f1 ,len1) arr1]
         [`(pull ,f2 ,len2) arr2])
      (let ([g (lambda (i) (f (f1 i) (f2 i)))]
            [len (min len1 len2)])
        (pull g len)))))
```

The resulting array length is the minimum of the two input lengths.

Pull Arrays

To fold/reduce an array, we just loop over the length of the array.

```
(define fold
  (lambda (f init arr)
    (match arr
      [`(pull ,g ,len)
        (let loop ([acc init] [i 0])
          (if (= i len) acc
              (loop (f acc (g i)) (add1 i))))])))
```

Scalar Product

```
(let ([a (pull f1 len1)] [b (pull f2 len2)])
  (fold + 0 (zipwith * a b)))
```

Let's see fusion in action!

```
= (fold + 0 (zipwith * (pull f1 len1)
                       (pull f2 len2)))
= (fold + 0 (pull (lambda (i) (* (f1 i) (f2 i)))
                  (min len1 len2)))
= (let loop ([acc 0] [i 0])
    (if (= i (min len1 len2)) acc
        (loop (+ acc (* (f1 i) (f2 i)))
              (add1 i))))
```

Generate and Fuse

If we specify our arrays directly as functions, an interesting opportunity opens up.

```
(let ([a (pull f1 len1)] [b (pull f2 len2)])
  (fold + 0 (zipwith * a b)))
```

The arrays a and b *could* be generated by `vector->pull`, but if they are written as functions we don't allocate an array. The code that generates the array is *fused* into the loop that consumes it.

Duplicating Work

Any time we read from the same array many times, we are in danger of duplicating work. This may be desirable in some situations, but if we want to avoid it we can `compute` the array.

```
(define compute
  (lambda (arr)
    (match arr
      [`(pull ,f ,len)
       (let ([vec (pull->vector arr)])
         `(pull ,(lambda (i) (vector-ref vec i))
             ,len))])))
```

Staged Pull Arrays

We can also use pull arrays to generate code. First we change our operators to return code.

```
(define s*
  (lambda (a b) `(* ,a ,b)))
(define s+
  (lambda (a b) `(+ ,a ,b)))
(define sadd1
  (lambda (a) `(add1 ,a)))
...
```

Staged Pull Arrays

We can generate code that calls build-vector.

```
(define scompute
  (lambda (arr)
    (match arr
      [`(pull ,f ,len)
       (let ([i (gensym 'i)])
         `(build-vector
           ,len
           (lambda (,i) ,(f i))))])))
```

Staged Pull Arrays

We can also generate code for a reduction.

```
(define sfold
  (lambda (f init arr)
    (match arr
      [`(pull ,g ,len)
        (let ([acc (gensym 'acc)]
              [i (gensym 'i)]
              [loop (gensym 'loop)])
          `(let ,loop ([,acc ,init] [,i 0])
             (if (= ,i ,len) ,acc
                 (,loop ,(f acc (g i))
                        (add1 ,i)))))])))
```

Staged Scalar Product

```
(define sprod
  (lambda (a b) (sfold s+ 0 (szipwith s* a b))))

> (sprod (pull f 5) (pull g 5))
'(let loop3786 ((acc3784 0) (i3785 0))
   (if (= i3785 5)
     acc3784
     (loop3786 (+ acc3784 (* (f i3785)
                              (g i3785)))
               (add1 i3785))))
```

Concat

We can define array concatenation over pull arrays like this:

```
(define concat
  (lambda (arr1 arr2)
    (match-let
        ([`(pull ,f1 ,len1) arr1]
         [`(pull ,f2 ,len2) arr2])
      (let ([g (lambda (i)
                 (if (< i len1) (f1 i)
                     (f2 (- i len1))))]
            [len (+ len1 len2)])
        (pull g len)))))
```

Is this efficient? Every element access now requires a conditional!

Limitations of Pull Arrays

Benefits of pull arrays

- Fusion
- Allows compositional programming

Limitations of pull arrays

- Concatenation (conditional in inner loop)
- Producing more than one element at a time

Push Arrays

Push arrays are parameterized by a function that *knows how to write an array*.

```
(define push
  (lambda (f len)
    `(push ,f ,len)))
(define pcompute
  (lambda (arr)
    (match arr
      [`(push ,f ,len)
        (let ([v (make-vector len)])
          (f (lambda (i a) (vector-set! v i a)))
          v)])))
```

Push Arrays

We can convert pull arrays to push arrays.

```
(define pull->push
  (lambda (arr)
    (match arr
     [`(pull ,f ,len)
       (let ([r (lambda (w)
                  (let loop ([i 0])
                    (if (= i len) (void)
                        (begin
                          (w i (f i))
                          (loop (add1 i))))))])
         (push r len))])))
```

Push Arrays

Now concat is efficient.

```
(define concat
  (lambda (arr1 arr2)
    (match-let
        ([`(push ,f1 ,len1) arr1]
         [`(push ,f2 ,len2) arr2])
      (let ([r (lambda (w)
                 (f1 w)
                 (f2 (lambda (i a)
                       (w (+ i len1) a))))])
        (push r (+ len1 len2))))))
```

Push Arrays

We can also write more than one element at a time.

```
(define dup
  (lambda (arr)
    (match arr
      [`(push ,f ,len)
        (let ([r (lambda (w)
                   (f (lambda (i a)
                        (w (* 2 i) a)
                        (w (+ (* 2 i) 1) a))))])
          (push r (* 2 len)))])))
```

Push Arrays

We can structure array computations like this:

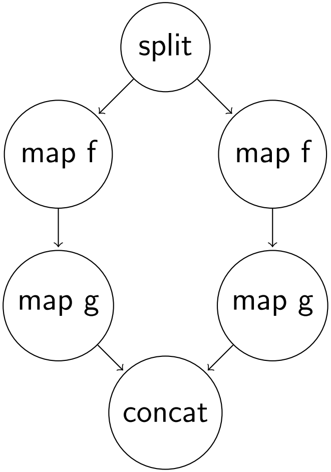
- Start with pull array.
- Convert to push array when we encounter an operation which cannot be done efficiently with pull arrays.
- Continue using push array, and write to memory if we encounter an operation which cannot be done efficiently with push arrays.

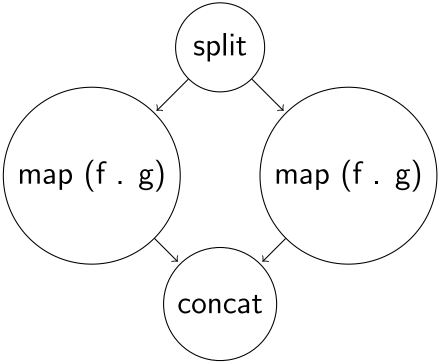
Array Fission

Why would we ever want to go the other way?

- Putting all of our work in a single loop makes it difficult to distribute across multiple devices (GPUs, CPUs).
- Splitting loops over arrays into smaller "chunks" that fit in cache is a well-known optimization.

Array Fission





Array Fission

Wrapper functions like `fission-map` can either take a plain array or two arrays that will eventually be concatenated.

```
(define fission-map
  (lambda (f arr)
    (match arr
      [`(pull ,g ,len)
       (match-let ([`(,a ,b) (split arr)])
         `(concat ,(map f a) ,(map f b)))]
      [`(concat ,a ,b)
       `(concat ,(map f a) ,(map f b))])))
```

Array Fission

`fission-compute` takes either a pull array or our `concat` structure.

```
(define fission-compute
  (lambda (arr)
    (match arr
      [`(pull ,f ,len)
       (compute arr)]
      [`(concat ,a ,b)
       (compute (concat a b))])))
```

We know how to efficiently concatenate arrays now, right?

Array Fission

fission-fold is similar, but it can also split the work of a fold on a pull array.

```
(define fission-fold
  (lambda (f init arr)
    (match arr
      [`(pull ,g ,len)
       (match-let ([`(,a ,b) (split arr)])
         (f (fold f init a) (fold f init b)))]
      [`(concat ,a ,b)
       (f (fold f init a) (fold f init b))])))
```

Ongoing Work

We are still experimenting with fissioning in Accelerate.
See: *Converting Data-Parallelism to Task-Parallelism by Rewrites* in FHPC 2015.