Better fusion for filters

Amos Robinson
PhD student at UNSW

February 26, 2014

I want my programs to be fast

But I don't want to have to write this kind of rubbish.

```
int  *out     = malloc(in_len);
int   out_ix = 0;
int   fold_m = 0;
int   in_ix  = 0;

for (; in_ix != in_len; ++in_ix) {
  int elm = in[in_ix] + 1;
  fold_m  = max(fold_m, elm);
  if (elm > 0) {
      out[out_ix++] = elm;
  }
}

*out_ptr = out;
*out_len = out_ix;
*fold_res= fold_m;
```

Haskell is alright for expressing high-level programs

But the high-level version isn't necessarily the fastest.

```
filterMax (vs : Vector Int) =
 let vs' = map    (+1)    vs
     m   = fold      0 max vs'
     vs''= filter (>0)    vs'
 in (m, vs'')
```

How many loops in this program?

About 3.

```
filterMax (vs : Vector Int) =
 let vs' = map    (+1)    vs
     m   = fold     0 max vs'
     vs''= filter (>0)    vs'
 in (m, vs'')
```

So, we need to merge the loops together

- This is called *fusion*
- But we must be careful not to modify the program's behaviour

There are restrictions

Only loops of the same size can be fused

```
different (as bs : Vector Int) =
 let as' = map (+1) as
     bs' = map (-1) bs
 in (as', bs')
```

No fusion here: `as` and `bs` may be different lengths.

A fold can't be fused with its output

Because a fold consumes the entire input before producing anything.

```
normalise (as : Vector Int) =
 let sum = fold (+) 0  as
     as' = map  (/sum) as
 in  as'
```

Two loops: `sum`; `as'`

What of filters?

Can we fuse operations on filtered data with the original?

```
sum2       (as : Vector Int) =
 let filt  = filter (>0)  as
     s1    = fold   (+) 0 filt
     s2    = fold   (+) 0 as


 in (s1, s2)
```

One loop: `filt s1 s2`

What of filters?

Only if they're in the generator loop

```
normalise2 (as : Vector Int) =
 let filt  = filter (>0)   as
     s1    = fold   (+) 0 filt
     s2    = fold   (+) 0 as
     filt' = map    (/s1) filt
     as'   = map    (/s2) as
 in (filt', as')
```

Three loops: `filt s1 s2`; `filt'`; `as'`

How do we do it?

Let's try to actually perform fusion on this program.

```
normalise2 as =
 let filt  = filter (>0)   as
     s1    = fold   (+) 0 filt
     s2    = fold   (+) 0 as
     filt' = map    (/s1) filt
     as'   = map    (/s2) as
 in (filt', as')
```

Integer linear programming

Find a variable assignment that minimises the goal, satisfying constraints.

```
Minimise    2x +  y
Subject to   x +  y   >= 2
             x + 2y   >= 4


==>


x = 0,
y = 2
```

A variable for each pair of nodes

For each pair of distinct nodes i and j, we have:

- boolean variable `C(i,j)`, 0 if fused together
- constant weight `W(i,j)`, the benefit of fusion

```
Minimise   Sum W(i,j) * C(i,j)
Subject to ...
```

Fold constraints

folds cannot be fused with their outputs

```
 let filt  = filter (>0)   as
     s1    = fold   (+) 0 filt
     s2    = fold   (+) 0 as
     filt' = map    (/s1) filt
     as'   = map    (/s2) as
```

```
Minimise   Sum W(i,j) * C(i,j)
Subject to
           C(s1,filt') = 1
           C(s2,as')   = 1
```

Filter constraints

Filtered data, despite having different sizes, *may* be fused:

```
let filt  = filter (>0)   as
    s1    = fold   (+) 0 filt
    s2    = fold   (+) 0 as
    filt' = map    (/s1) filt
    as'   = map    (/s2) as
```

```
Minimise   Sum W(i,j) * C(i,j)
Subject to ...
           C(filt,s1)     = 1 ==> C(s1,s2)      = 1
           C(filt,s2)     = 1 ==> C(s1,s2)      = 1

           C(filt,filt') = 1 ==> C(filt',as') = 1
           C(filt,as')   = 1 ==> C(filt',as') = 1
```

Acyclic constraint

The last thing we need is to ensure the clustering is acyclic; otherwise we'd need to execute two clusters at once.

- Give each node a number `pi(i)`
- If `C(i,j) = 0` then `pi(i) = pi(j)`
- If j mentions i and `C(i,j) = 1` then `pi(i) < pi(j)`
- Otherwise, if j doesn't mention i then they can be anything

Acyclic constraint - ILP

```
PiMentions(i,j) =
      C(i,j) <= pi(j) - pi(i) <= 100*C(i,j)

PiNoMention(i,j)
 -100*C(i,j) <= pi(j) - pi(i) <= 100*C(i,j)
```

Put it all together

```
Minimise   Sum W(i,j) * C(i,j)
Subject to
     C(s1,filt')    = 1
     C(s2,as')      = 1

     C(filt,s1)     = 1 ==> C(s1,s2)      = 1
     C(filt,s2)     = 1 ==> C(s1,s2)      = 1

     C(filt,filt') = 1 ==> C(filt',as') = 1
     C(filt,as')    = 1 ==> C(filt',as') = 1

     PiMentions  (filt, s1)
     PiNoMention (filt, s2)
     PiMentions  (filt, filt')
     PiNoMention (filt, as')
     ...
```

Put it all together

```
Minimise   Sum W(i,j) * C(i,j)
Subject to
     C(s1,filt')    = 1
     C(s2,as')      = 1

     C(filt,s1)     = 1 ==> C(s1,s2)      = 1
     C(filt,s2)     = 1 ==> C(s1,s2)      = 1

     C(filt,filt') = 1 ==> C(filt',as') = 1
     C(filt,as')    = 1 ==> C(filt',as') = 1

==>
C(filt, s1)     = 0
C(filt, s2)     = 0
C(s1,   s2)     = 0

C(filt, s1)     = 0
C(filt, s2)     = 0
```

prabob

Definition of `W(i,j)`

Just a heuristic:

```
W(i,j) = 100 if j mentions i
             (then fusing them together means
             i will be in cache when j is computed)

W(i,j) = 1   otherwise, the only benefit is fewer loops
```

There may be multiple ways of fusing a program

How do we choose which one?

```
multiple (us : Vector Int) =
 let xs = map  (+1)  us
     y  = fold (+) 0 us
     ys = map  (+y)  xs
 in  ys
```

Choice 1

3 loops, 2 manifest arrays

```
multiple (us : Vector Int) =
 let xs      = loop #1: map  xs
     y       = loop #2: fold y
     ys      = loop #3: map  ys
 in  ys
```

Choice 2

2 loops, 2 manifest arrays

```
multiple (us : Vector Int) =
 let (xs, y) = loop #1: map  xs, fold y

     ys      = loop #2: map  ys
 in  ys
```

Choice 3 - ding!

2 loops, 1 manifest array

```
multiple (us : Vector Int) =
 let y        = loop #1: fold y

     ys       = loop #2: map  xs, map  ys
 in  ys
```

My rough definition of ‘best’ is

Minimise manifest arrays, then minimise number of loops.

Fusion is NP-hard

See Alain Darte 1998 for a proof by reduction from vertex cover.