Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Fusion in Accelerate

Ivo Gabe de Wolff

Includes material from Accelerate source code and slides of Gabriele Keller

June 22, 2021

1. Accelerate

Accelerate


```
dotp :: Acc (Vector Float)
     -> Acc (Vector Float)
     -> Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

Outline

- Introduction to Accelerate
- How fusion currently works
- Current research: extend fusion & in-place updates

Arrays

- Array programming
- `Acc`: array computation
- Multi-dimensional
- Shapes: `Z` or `sh :. Int`
- `type Scalar a = Array Z a`
- `type Vector a = Array (Z :. Int) a`
- `type Matrix a = Array (Z :. Int :. Int) a`

Combinators

Language only includes parallelizable constructs.

- Map
- Folds and scans
- Permute (scatter, random writes)
- Backpermute (gather, random reads)
- Stencil (map with neighbourhood)

And control flow to compose those parallel operations: conditionals, loops, let-bindings, tuples.

Domain specific language

- Combinators imply the parallel structure.
- No need for (fragile) analyses to recover parallel structure.

Embedded DSL

- Haskell gives polymorphism and support for higher order functions.
- In our internal AST, programs are monomorphic and first order.

☞ Analyses are simpler in this monomorphic combinator-based language.

Map


```
map :: (Shape sh, Elt a, Elt b)
    => (Exp a -> Exp b)
    -> Acc (Array sh a)
    -> Acc (Array sh b)
plusOne = map (+1)
```

- Each thread calculates one element of the array

Backpermute


```
backpermute :: (Shape sh, Shape sh', Elt a)
            => Exp sh'
            -> (Exp sh' -> Exp sh)
            -> Acc (Array sh  a)
            -> Acc (Array sh' a)

reverse xs = backpermute (shape xs)
    (\(I1 i) -> I1 (size xs - 1 - i)) xs
```

- Random reads

Transform


```
transform :: (Shape sh, Shape sh'
             , Elt a, Elt b)
          => Exp sh'
          -> (Exp sh' -> Exp sh)
          -> (Exp a -> Exp b)
          -> Acc (Array sh a)
          -> Acc (Array sh' b)

foo = reverse . plusOne
```

- Composition of `backpermute` and `map`.
- Not exposed to the user.
- Result of fusion.

Generate


```
generate :: (Shape sh, Elt a)
         => Exp sh
         -> (Exp sh -> Exp a)
         -> Acc (Array sh a)
generate (constant (I1 100))
         (\(I1 i) -> i * 2)
-- [0, 2, 4, ...]
```

- Each thread computes the value of one index.
- Could be used to implement `map`, `backpermute` and `transform`,
- but then we would lose some structure.

Folds


```
fold :: (Shape sh, Elt a)
     => (Exp a -> Exp a -> Exp a)
     -> Exp a
     -> Acc (Array (sh :. Int) a)
     -> Acc (Array sh a)
```

- Associative operator to enable parallel reduction.
- Other variants: fold without initial value, segmented folds, scans.

More combinators

- Permute: random writes
- Conditional
- While
- Zip, unzip
- Pairs
- Slice, replicate
- Stencil

AST

- Generalized algebraic datatypes (GADT).
- Typed environment and result.
- Pattern functor: type argument acc for recursive positions.

```
data PreOpenAcc acc aenv a where
  Avar  :: ArrayVar           aenv (Array sh e)
        -> PreOpenAcc acc aenv (Array sh e)
  Map   :: TupleType e'
        -> Fun                aenv (e -> e')
        -> acc                aenv (Array sh e)
        -> PreOpenAcc acc aenv (Array sh e')
  ...
newtype OpenAcc aenv t =
  OpenAcc (PreOpenAcc OpenAcc aenv t)
```

Current pipeline

- Sharing recovery
- Simplify tuples
- Fusion
- Code generation with LLVM

Runtime compiler

- Accelerate operates at Haskell runtime
- We compile and run the program at Haskell runtime
- Allows meta programming
- Compilation at Haskell compile time is possible with Template Haskell

Sharing recovery

- The combinators construct an AST representing the computation
- Some nodes may be used multiple times in the tree
- Make this explicit by adding let bindings

```
let
  xs = generate ..
in
  T2 xs (fold (+) 0 xs)
```

2. Fusion in the current pipeline

Fusion

- Programming model advocates splitting the program into many kernels.
- A naive implementation would result in the creation of many intermediate arrays.
- Fusion: combine multiple kernels into one.
- Mandelbrot had a speed up of 1000%, typically 50%.

Minimize:

- Number of kernels
- Number of (intermediate) arrays
- Number of memory operations

Example

$y = \textbf{map}\ f\ \$\ \textbf{map}\ g\ x$

- Why create the intermediate array for the result of the right `map`?
- Fusion will prevent the creation of that array.
- Result: `y = map (f . g) x`

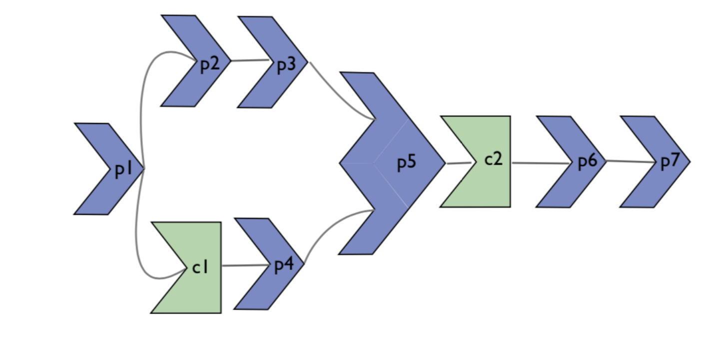
Example

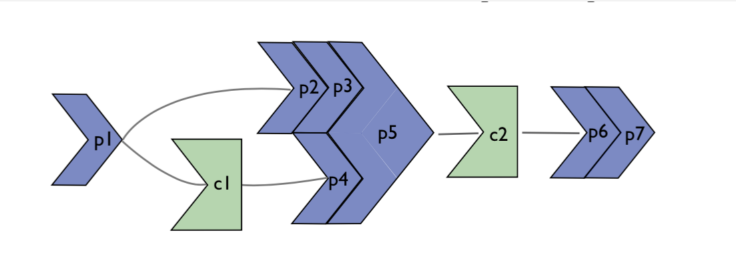

```
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

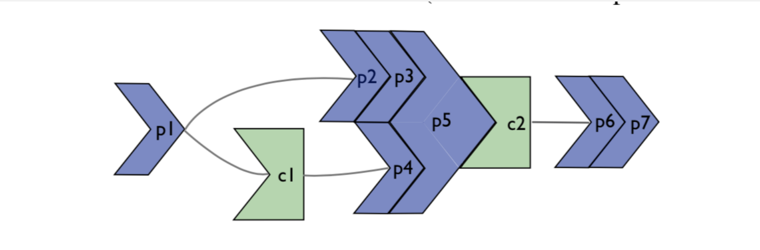
- Why create the intermediate array for the result of `zipWith`?
- Fusion will prevent the creation of that array.
- We cannot represent the result in the same AST data type.

Elementwise and collective operations

- *Elementwise*: Each element of the result depends on at most one element of input array (e.g, map, backpermute, generate).
- *Collective*: Each element of result depends on multiple elements of input array (e.g., folds, scans, stencil operations).
- Fusion treats them separately:
  - Elementwise/Elementwise fused via program transformation.
  - Elementwise/Collective during code generation.


p2
p3
p1
p5
c2
p6
p7
c1
p4


p1
p2
p3
c1
p4
p5
c2
p6
p7


p1
c1
p2
p3
p4
p5
c2
p6
p7

Elementwise operations as functions

- Represent elementwise operations as a function from index to value.
- In the dot product example:
  `\ix -> (xs ! ix) * (ys ! ix)`.
- We also need an expression of the size (shape) of the array.
- We use this for elementwise/collective fusion.
- However, after elementwise/elementwise fusion we still want to know which elementwise operation we have.

Cunctation

We convert an array computation into a cunctation, which is either:

Done : Terms which we cannot fuse.

Yield : An expression of its size and a function from index to value. Similar to `generate`.

Step : An array variable, an index transform and a value transform. Similar to `transform`.

☞ Step $<$ Yield $<$ Done

Cunctation of `map`

- Map of a Step becomes a Step
- Map of a Yield becomes a Yield
- Map of a Done: manifest the argument array, then return a Step

Elementwise/Elementwise fusion

- Convert cunctations back to elementwise operations.
- `Yield` becomes a `generate`.
- `Step` becomes:
  - a `backpermute` if the value transform was an identity function.
  - a `map` if the index transform was an identity function (and the size is preserved).
  - a `transform` otherwise.

☞ A `map` may allow more other optimisations than a `transform`, like in-place updates.

Pattern functors

AST is defined as a pattern functor, closed by OpenAcc:

```
newtype OpenAcc aenv t =
  OpenAcc (PreOpenAcc OpenAcc aenv t)
```

Use a different data type to close it:

```
data DelayedOpenAcc aenv a where
  Manifest :: PreOpenAcc DelayedOpenAcc aenv a
           -> DelayedOpenAcc aenv a
  Delayed  :: ArrayR (Array sh e)
           -> Exp aenv sh
           -> Fun aenv (sh  -> e)
           -> Fun aenv (Int -> e)
           -> DelayedOpenAcc aenv (Array sh e)
```

Pattern functors

- Allows to store information anywhere in the tree.
- Type and effects: you could store variables of the proof tree this way.
- Sharing recovery does that in multiple steps: `UnscopedAcc`, `ScopedAcc`, before going to the resulting `Acc`.
- Disadvantage: We could store a `Delayed` at locations where we really expect a manifest array.

Code generation

- Delayed terms stay present in the AST until code generation.
- Embed the code of elementwise function into the code of the collective operation.
- Example of dot product: instead of performing indexing in the code of the fold, add code of
  `\ix -> (xs ! ix) * (ys ! ix)`.

After the break

What is the best way to compile this to imperative code?

```
as = map f1 x
bs = map f2 x

cs = map (\y -> bs !! y) as
ds = map (\y -> as !! y) bs
```

Note that (!!) is indexing in arrays, so constant time.

3. Current research

Current research

- Horizontal and diagonal fusion like (map f xs, map g xs).
- Collective/Collective fusion
- In-place updates in more cases

This results in many options how a program can be transformed. Greedy approach doesn't work anymore.

An example


```
as = map f1 xs
bs = map f2 xs

cs = map (\y -> bs !! y) as
ds = map (\y -> as !! y) bs
```

Fusion becomes a clustering problem. Options include:

- $\{as, bs\}$, $\{cs, ds\}$ (horizontal fusion)
- $\{as\}$, $\{bs, ds\}$, $\{cs\}$ (diagonal fusion)
- $\{bs\}$, $\{as, cs\}$, $\{ds\}$ (diagonal fusion)

Horizontal fusion

$\{as, bs\}, \{cs, ds\}$

```
as, bs, cs, ds = new arrays;
parallel for i in 0..n
  { x = xs[i]; as[i] = f1(x); bs[i] = f2(x); }
parallel for i in 0..n
  { cs[i] = bs[as[i]]; ds[i] = as[bs[i]]; }
```

Diagonal fusion

$\{as\}$, $\{bs, ds\}$, $\{cs\}$

```
as, bs, cs, ds = new arrays;
parallel for i in 0..n
  { as[i] = f1(xs[i]); }
parallel for i in 0..n
  { b = f2(xs[i]; bs[i] = b; ds[i] = as[b]; }

parallel for i in 0..n
  { cs[i] = bs[as[i]]; }
```

In-place updates

- Reuse an input array instead of allocating a new array.
- The input should be a unique reference.
- A map can reuse the input array if it has the same element type (or more general, the same element size).
- A permute in general has to copy the defaults array. If we can perform in-place updates, we don't need to copy that.
- For permute, in-place updates can result in a lower time complexity.

Uniqueness?

We can safely perform in-place updates on unique references.

- Uniqueness: there is only one reference to a variable.
- Temporal uniqueness: there is only one reference to a variable *at the time we perform the in-place update*.

Thus, we can perform in-place updates if we can reorder the program such that other uses of the array happen earlier.

In-place updates in the example


```
as = map f1 xs
bs = map f2 xs

cs = map (\y -> bs !! y) as
ds = map (\y -> as !! y) bs
```

The map of c may perform an in-place update on a if the map of d is executed earlier.

Diagonal fusion

$\{as\}$, $\{bs, ds\}$, $\{cs\}$

```
as, bs, cs, ds = new arrays;
parallel for i in 0..n
  { as[i] = f1(xs[i]); }
parallel for i in 0..n
  { b = f2(xs[i]; bs[i] = b; ds[i] = as[b]; }

parallel for i in 0..n
  { cs[i] = bs[as[i]]; }
```

Diagonal fusion, with in-place updates

$\{as\}, \{bs, ds\}, \{cs\}$

```
as, bs, ds = new arrays;
parallel for i in 0..n
  { as[i] = f1(xs[i]); }
parallel for i in 0..n
  { b = f2(xs[i]; bs[i] = b; ds[i] = as[b]; }
cs = as;
parallel for i in 0..n
  { cs[i] = bs[as[i]]; }
```

What is the best option?

- First option: 2 loops, 4 arrays
- Second option: 3 loops, 3 arrays
- And 9 other options, of which 3 are maximal

Interaction with in-place updates

- The best option for fusion prevents in-place updates
- The best option for in-place updates prevents fusion
- Can we decide on them in one analysis?

Analysis

- Fusion becomes a clustering problem
- Modelled as an ILP (Integer Linear Program, not instruction level parallellism!)
- Extend the ILP with in-place updates

Objective function

Minimize:

- Number of kernels
- Number of (intermediate) arrays
- Number of memory operations

Decision variables

- For each combinator, at which point in time it should be executed
- For an input of map or permute, whether we perform in-place updates

Constraints

- Fusion: enforce dependencies in order
- In-place: if we perform in-place updates, ensure that other uses of the array happen earlier

Implementation work

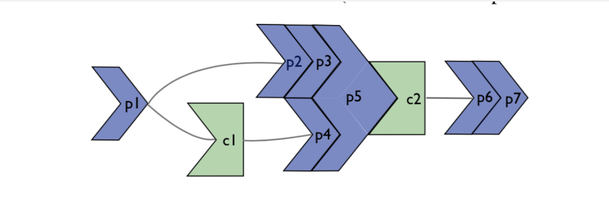
- After solving the ILP, we must transform the program accordingly
- Add synchronisation points to ensure temporal uniqueness
- Task parallelism: execute multiple kernels at the same time
- Typed environment makes large program transformations difficult

Other goals of the new pipeline

- Lower runtime overhead
- More backend specific optimisations

Let bindings

- Array computations can be bound to variables.
- Fusing those may duplicate work.
- Currently: only fuse those if the variable is used once.
- Desired: fuse those in many cases, as the cost of memory is usually higher than the computational cost.



Conclusion

- Program analyses are easier on the DSL
- Some analyses are not needed at all: the combinators imply the parallel structure
- Being an EDSL we also need to perform other analysis than a classic compiler (sharing recovery)
- Many ways to transform a program with fusion and in-place updates
- Use an ILP to make the decision