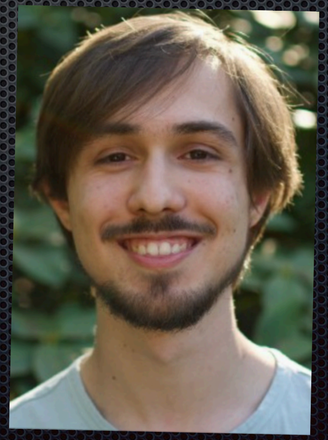
Accelerate

High Performance Computing in Haskell

Gabriele Keller

Trevor McDonell, Ivo Gabe de Wolff, David van Balen, Josh Meredith



many others: Robbert van der Helm, Tom Smeding, Bart Wijgers, Nara Prasetya, Rick van Hoef, Hugo Peters, Tijn Janssen, Jason van den Hurk

What is Accelerate?

What's currently happening

Where we want to take it

Accelerate

- Started as a spin-off project of DPH (Data Parallel Haskell) at UNSW, Sydney with Manuel Chakravarty, Sean Lee,Trevor McDonell
- Now most team members based in Utrecht
- DPH:
  - based on a generalisation of NESL's irregular, nested model, trying to map it efficiently to concrete architectures via GHC

Accelerate

- Accelerate supports a fairly simple model (nested regular data parallelism)
- Idea is to generalise it for certain forms of irregular parallelism
- Initial aim:
  - give Haskell programmers a *low overhead way* to exploit the available hardware (GPU, multiple cores on the their desktop/laptop) to achieve performance
- Don’t want to depend on GHC for the performance critical part
  - deeply embedded in Haskell

Accelerate

Haskell/Accelerate program

GHC

Haskell/Accelerate executable

Reify and optimise Accelerate program

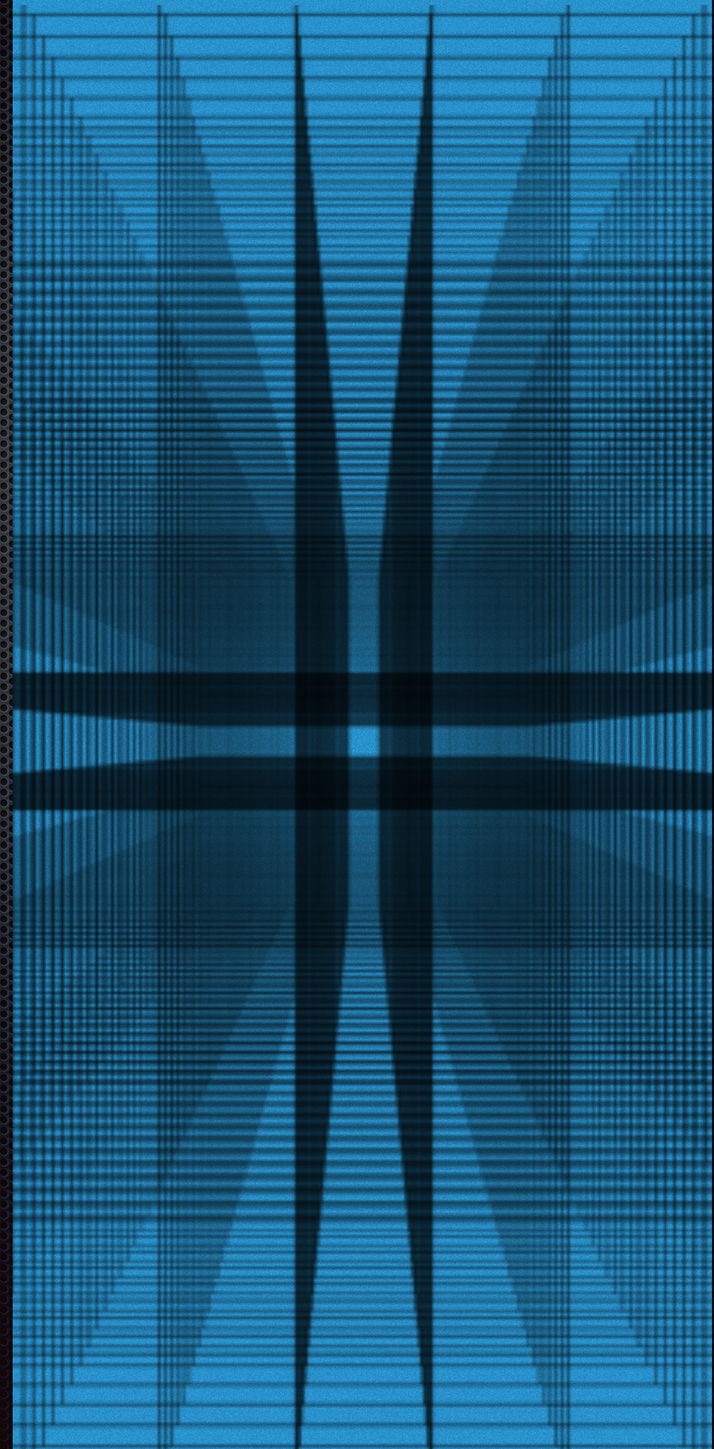
Target code

Compile and run on the CPU/GPU

Copy result back to Haskell

Accelerate

- Data-parallel computations are expressed via operations on multi-dimensonal arrays



Continuing Adam's Syntax benchmark:

NumPy

```
matmul = lambda x, y: np.einsum ('ik,kj->ij', x,y)
```

SaC

```
{ [i,j] -> sum ({ [k] -> A[i,k] * B[k,j] }) }
```

Dex

```
for i j. sum for k. x.i.k * y.k.j
```

Futhark

```
def matmul_i32 [n] [p] (A: [n][m] i32)
                       (B: [m][p] i32) =
    map (\A_row ->
      map (\B_col -> reduce (+) 0 (map2 (*) A_row B_col))
          (transpose B))
      A)
```

Matrix multiplication in Accelerate

```
mmx1  ::  Num a
      =>  Acc  (Matrix a) ->
          Acc  (Matrix a) ->  Acc (Matrix a)
mmx1 arr brr
  = sum (generate (I3 cA rB rA)
          (\(I3 i j k) -> arr!(I2 i k) * brr!(I2 k j)))
    where
        (I2 cA rA) = shape arr
        (I2 cB rB) = shape brr

mmx2 arr brr
  = sum (zipWith (*) arrRepl brrRepl)
    where
        arrRepl = replicate (S3 All cB  All) arr
        brrRepl = replicate (S3 rA  All All) trr
        trr     = transpose brr
        (I2 cA rA) = shape arr
        (I2 cB rB) = shape brr

mmx3 arr brr = arr <> brr
```

Index-based vs point-free

- Accelerate offers a set of second order array operations
  - maps, zipWiths
  - stencil (convolution matrix) operations
  - scans, folds
  - permutations and backpermutations/scatter and gathers
  - conditionals, while loops and such on the array and scalar level

Index-based vs point-free

- Many of the specialised functions in Accelerate can be expressed via generate (`zipWith`, `replicate`, `transpose`)

```
generate shapeOfResult indexToValueFn
```

- if possible, specialised functions should be used
  - Accelerate doesn't try to detect access patterns in `generate`

memory access patterns are more explicit

intermediate structures are ‘fused away’

```
mmx2 arr brr
  = sum (zipWith (*) arrRepl brrRepl)
    where
      trr     = transpose brr
      arrRepl = replicate (S3 All cB  All) arr
      brrRepl = replicate (S3 rA  All All) trr
      (I2 cA rA) = shape arr
      (I2 cB rB) = shape brr
```

as long as the data of an array is only used once
work is never copied as a result of fusion

A closer look at the types

```
mmx2 arr brr
  = sum (zipWith (*) arrRepl brrRepl)
    …


sum = fold (+) 0


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

Shape (Rank) polymorphism

- The Shape type describes shapes of and indices to multi-dimensional arrays:

```
data Z = Z

infixl :.
data tail :. head = tail :. head
```

- Members of type class Shape have the form `Z :. Int :. Int :. …`

```
type DIM0      = Z
type DIM1      = DIM0 :. Int
…
type Scalar a = Array DIM0 a
type Vector a = Array DIM1 a
type Matrix a = Array DIM2 a
```

- In the matrix multiplication program: `(I3 i j k)`

Slices

- Similar to Shape, but components include `Any`, `All`

```
slice :: (Slice slix, Elt e)
      => Acc (Array (FullShape slix) e)
      -> Exp slix
      -> Acc (Array (SliceShape slix) e)

replicate :: (Slice slix, Elt e)
          => Exp slix
          -> Acc (Array (SliceShape slix) e)
          -> Acc (Array (FullShape slix) e)
```

```
arrRepl = replicate (S3 All cB  All) arr
brrRepl = replicate (S3 rA  All All) trr
```

The `Elt` type class

```
sum = fold (+) 0

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

The `Elt` type class

- All the usual base types are members of this class
  - floating points, ints of various sizes, char, boolean, and so on
  - shapes and indices
  - n-tuples
  - sum-types, like the `Maybe` type:

```
data Maybe a = Nothing | Just a

Elt a => Elt (Maybe a)
```

The `Elt` type class

- User-defined data types:
  - Accelerate compiler needs to know
    - how to represent these types efficiently as matrix elements

```
class Elt a where
  type EltR a
   fromElt :: a -> EltR a
   toElt   :: EltR a -> a
```

The `Elt` type class

- Default implementations for non-recursive data types are provided
  - product types (tuples, records)
    - arrays of tuples represented as tuples of arrays
  - sum types (alternatives)
    - tuples of arrays of data, flags

The `Elt` type class

- Arrays are not in the Elt type class
- Ragged/irregular matrices are not directly supported
- Segmented operations can be used to express manually flattened computations

```
foldSeg :: forall sh e i. (Shape sh, Elt e, IsIntegral i)
        => (Exp e -> Exp e -> Exp e)
        -> Exp e
        -> Acc (Array (sh :. Int) e)
        -> Acc (Segments i)
        -> Acc (Array (sh :. Int) e)
```

The `Elt` type class

```
smvm :: Num a =>
        Acc (SparseMatrix a) -> Acc (Vector a) -> Acc (Vector a)
smvm smat vec =
  let
    (T2 segd (T2 inds vals)) = smat
    vecVals  = backpermute inds vec
    products = zipWith (*) vecVals vals
  in
    foldSeg (+) 0 products segd
```

The `Exp` and `Acc` type constructors

```
sum = fold (+) 0

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

`Exp a` sequential calculation resulting in value of scalar type `a`

`Acc arr` parallel calculation resulting in value of array type `arr`

both just represent abstract syntax trees of the computations, not actual values of that type

The `Exp` and `Acc` type constructors

- To generate code and execute it on the CPUs or GPU, we need to call a version of `run`

  - `run :: Acc a -> a`
  - `runQ :: (Acc a -> Acc b) -> a -> b`
  - `runN :: (Acc a -> Acc b) -> a -> b`

types simplified

The `Exp` and `Acc` type constructors

- Lifting into the embedded language can be done automatically for overloaded values:

```
0     :: Num a => a
(+)   :: Num a => a -> a -> a

instance Num a => Num (Exp a) where
    (+) expr1 expr2 = …
```

The `Exp` and `Acc` type constructors

- Not so easy for non-overloaded values

```
not :: Exp Bool -> Exp Bool
not False = True
not True  = False
```

```
lift :: a -> Exp a
```

(simplified)

```
not :: Exp Bool -> Exp Bool
not False = lift True
not True  = lift False
```

```
(?) :: Elt t => Exp Bool -> (Exp t, Exp t) -> Exp t
```

```
not :: Exp Bool -> Exp Bool
not b = b ? (lift False, lift True)
```

How can we pattern match on embedded values?

Pattern matching on embedded expressions

- Unlifting is not generally possible without evaluation

```
Exp a -> a
```

- only possible it topmost constructor of a type is unique

```
Exp (a, b) -> (Exp a, Exp b)
```

```
swap:: Exp (a, b) -> Exp (b, a)
swap xy =
  let
    (x, y) = unlift xy
  in lift (y, x)
```

Pattern matching on embedded expressions

- Haskell pattern synonyms and view patterns

```
swap :: Exp (a, b) -> Exp (b, a)
swap (T2 x y) = T2 y x
```

```
sum (generate (I3 cA rB rA)
              (\(I3 i j k) -> arr!(I2 i k) * brr!(I2 k j)))
```

but this does not for types with multiple constructors - e.g. `Bool`

Pattern matching on embedded expressions

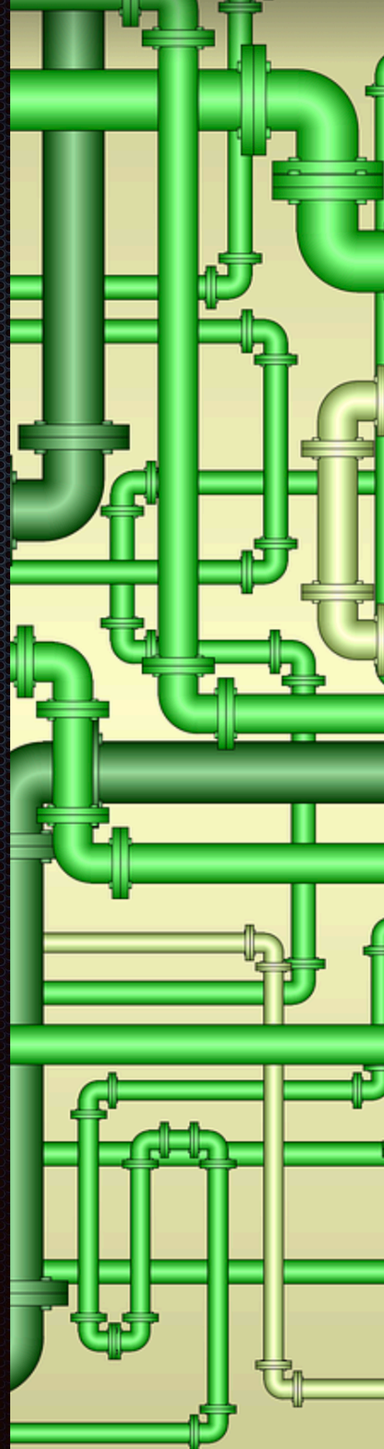
```
data Either a b = Left a | Right b
  deriving (Generic, Elt)

mkPattern ''Either


swapE :: Exp (Either a b) -> Exp (Either b a)
swapE (Left_  a)  = Right_ a
swapE (Right_ b)  = Left_  b

match swapE
```

Current work

Rewrite of the compiler pipeline

Ivo Gabe de Wolff & David van Balen

- Improved fusion
- Destructive updates
- Specification of different schedules on the user level



Improved Fusion

- Currently
  - intermediate arrays whose data is used only once are reliably fused
  - work is never duplicated (even if that might be the right thing to do)
- In the new pipeline:
  - horizontal, vertical and diagonal fusion
  - takes estimated cost of operations into account
  - using ILP solver to find more efficient schedules

Destructive updates

- When iterating over the whole array, Accelerate is pretty good at handling memory efficiently
- However, destructive updates are important for some classes of algorithms

```
permute
    :: forall sh sh' a. (Shape sh, Shape sh', Elt a)
    => (Exp a -> Exp a -> Exp a)          -- combination function
    -> Acc (Array sh' a)                 -- default values
    -> (Exp sh -> Exp (Maybe sh'))       -- index permutation function
    -> Acc (Array sh  a)                 -- source values to be permuted
    -> Acc (Array sh' a)
```

- In the new pipeline, we can identify situations where arrays can and should be updated destructively

Specifying schedules on the user-level

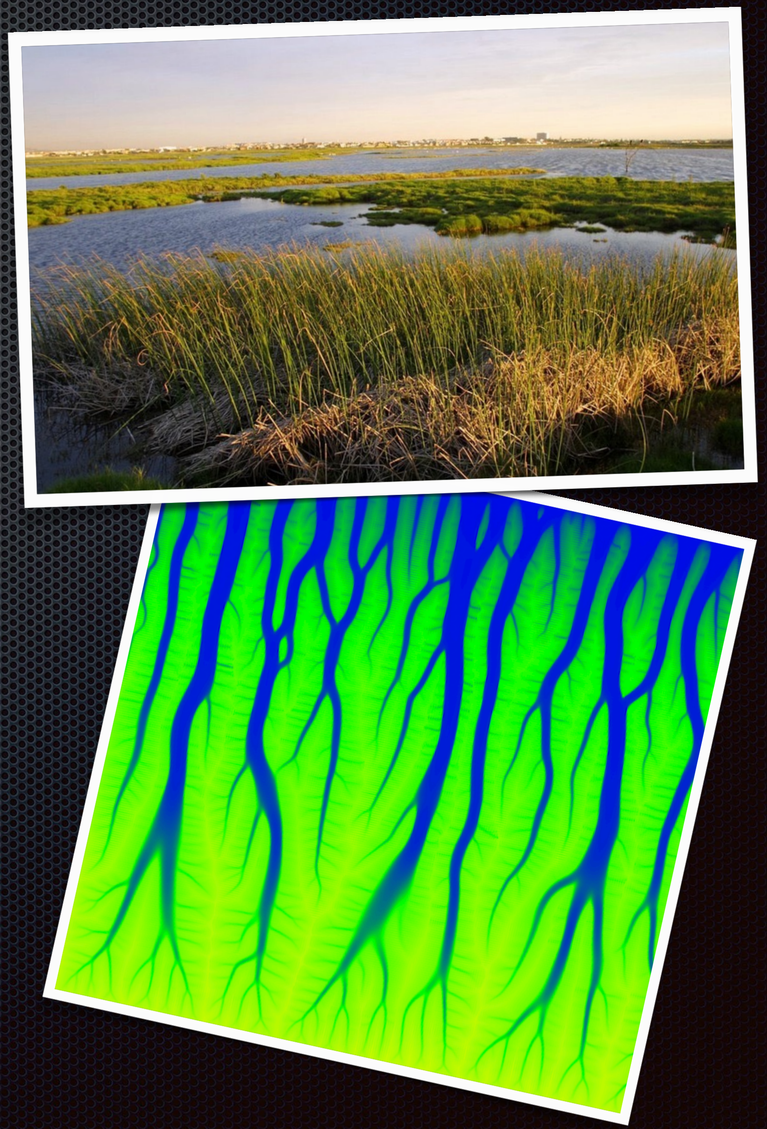
- Currently, user has to rely on Accelerate to pick the schedule
- No explicit representation of the schedule in the AST
- New pipeline will provide high-level constructs for the user to specify schedules and enable optimisations


Future Work

More sophisticated stencil computations

- We're working with researchers from Geoscience at UU and NIOZ on simulations of coastal developments
- Essentially, stencil computations:

```
stencil:: (Stencil sh a stencil, Elt b)
       => (stencil -> Exp b)
       -> Boundary (Array sh a)
       -> Acc (Array sh a)
       -> Acc (Array sh b)

stencilFn ((_,t,_),
           (l,c,r),
           (_,b,_)) = t + l + c + r + b
```



- More efficient scheduling on GPUs
- Language support for dynamic adaption of simulation speed
- Better interface for easy visualisation of data

Automatic Differentiation

- Tom Smeding's work on reverse automatic differentiation for Accelerate identified some performance pain points
  - generated code nothing like hand-written code Accelerate optimises for
  - more sophisticated fusion required
  - memory management issues need to be addressed
- Size inference?

Thank you!

acceleratehs.org

https://github.com/AccelerateHS/