Accelerate

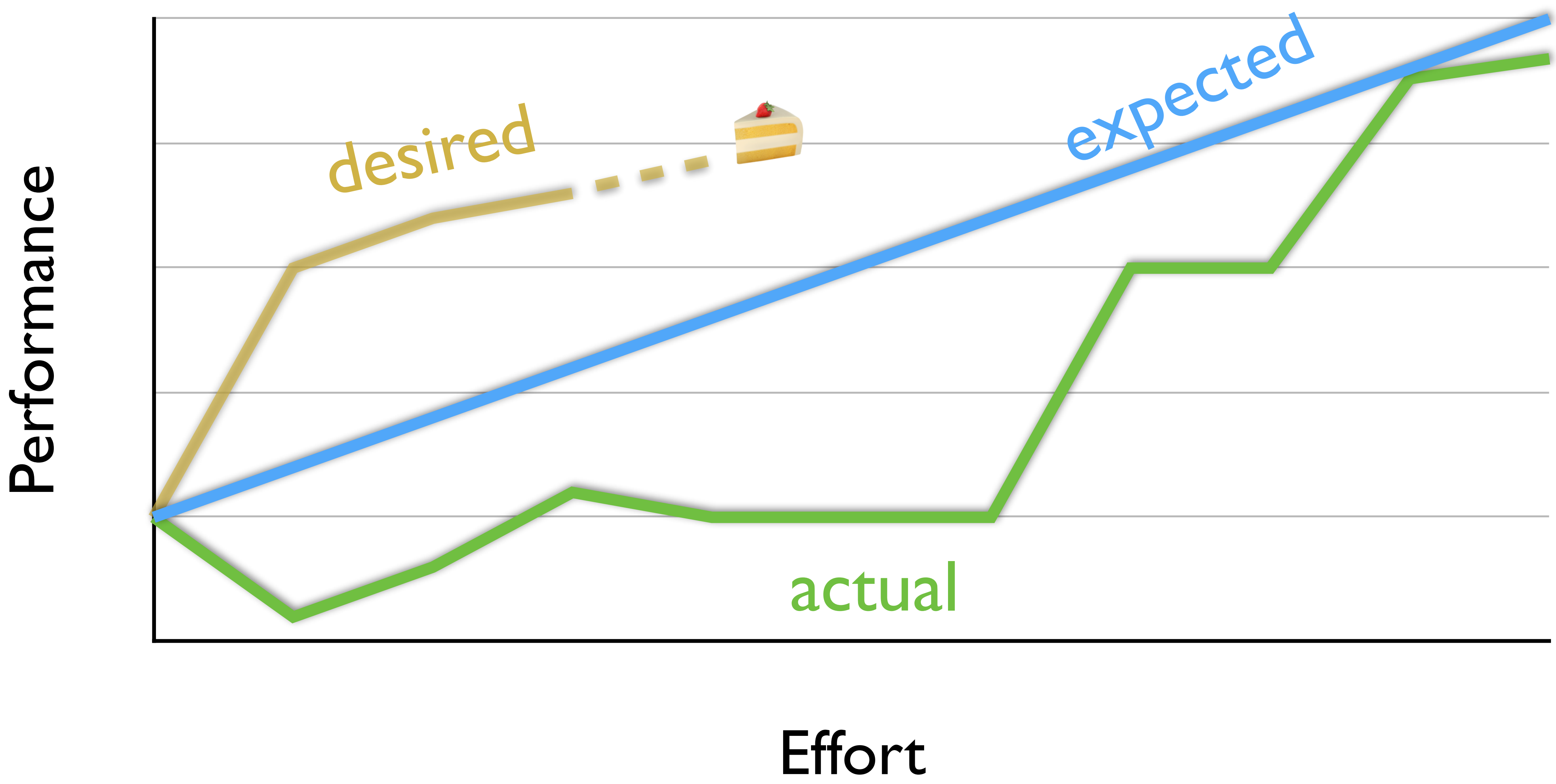
High Performance Simulations in Haskell

Gabriele Keller, Utrecht University

Trevor McDonell (UU) Josh Meredith

acceleratehs.org

AccelerateHS

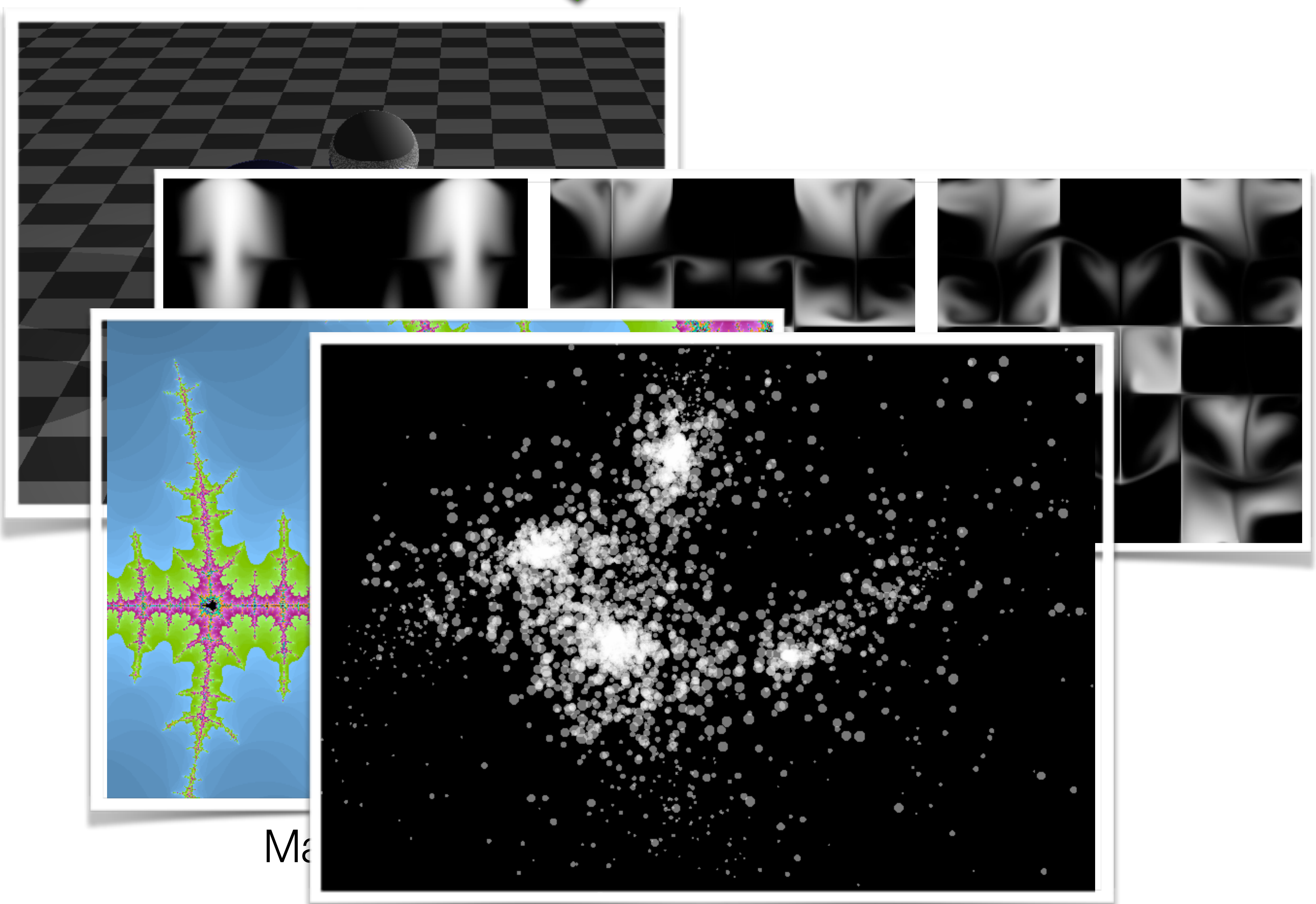

desired
expected
actual
Performance
Effort

Accelerate

- Accelerate supports array based, regular data parallelism
  - Aim: easier to write, while being as fast/faster than hand coded CUDA/ OpenCL
  - multi-dimensional arrays of fixed sized element types
  - no nested arrays
  - element type user extensible

Accelerate

Array/matrix computations



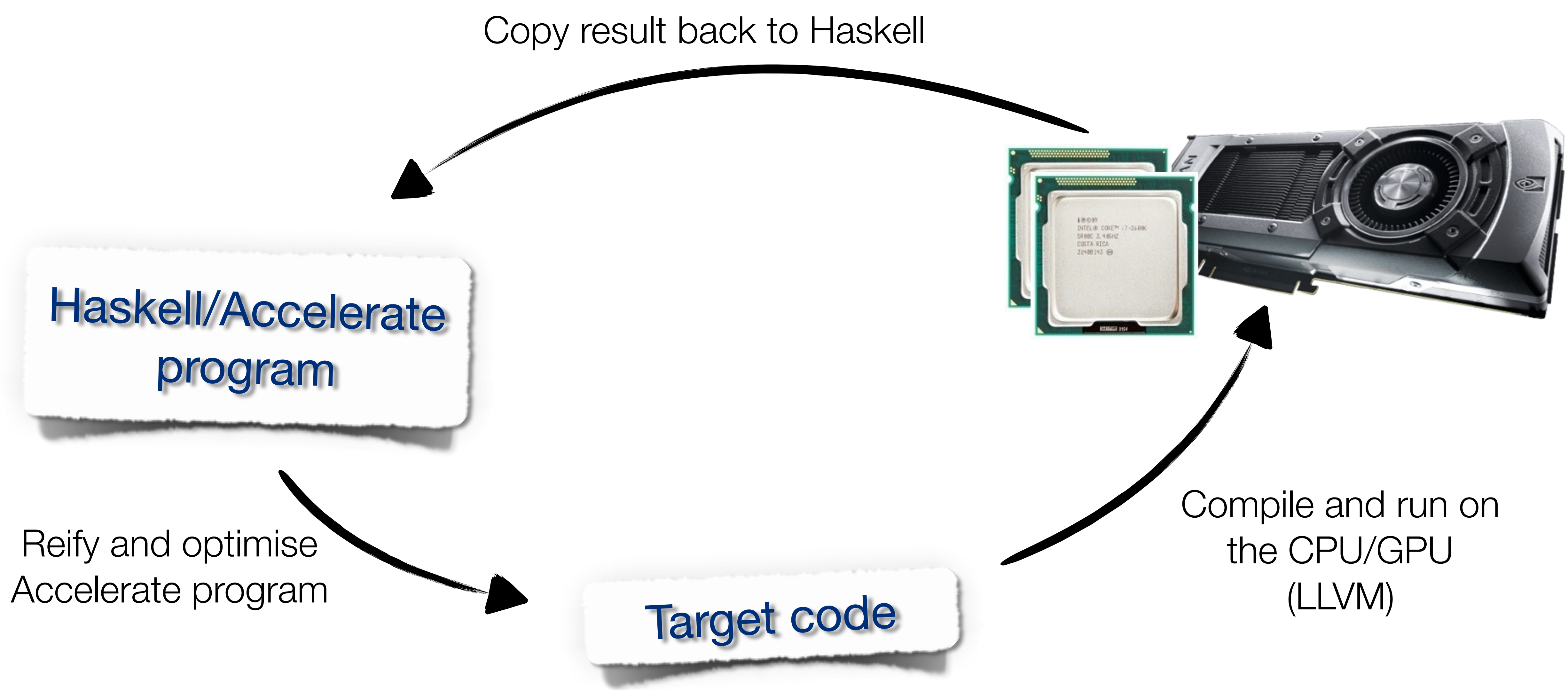
n-body gravitational simulation

Everything else

but, we working on this!

Accelerate

An deeply embedded language for data-parallel arrays



Example: dot product

```
dotp :: Num a
     => [a] -> [a] -> a
dotp xs ys = foldl (+) 0 (zipWith (*) xs ys)
```

Accelerate

Collective operations which compile to parallel code

```
import Data.Array.Accelerate
dotp :: (Elt a, Num a)
     => Acc (Vector a)
     -> Acc (Vector a)
     -> Acc (Scalar a)
```

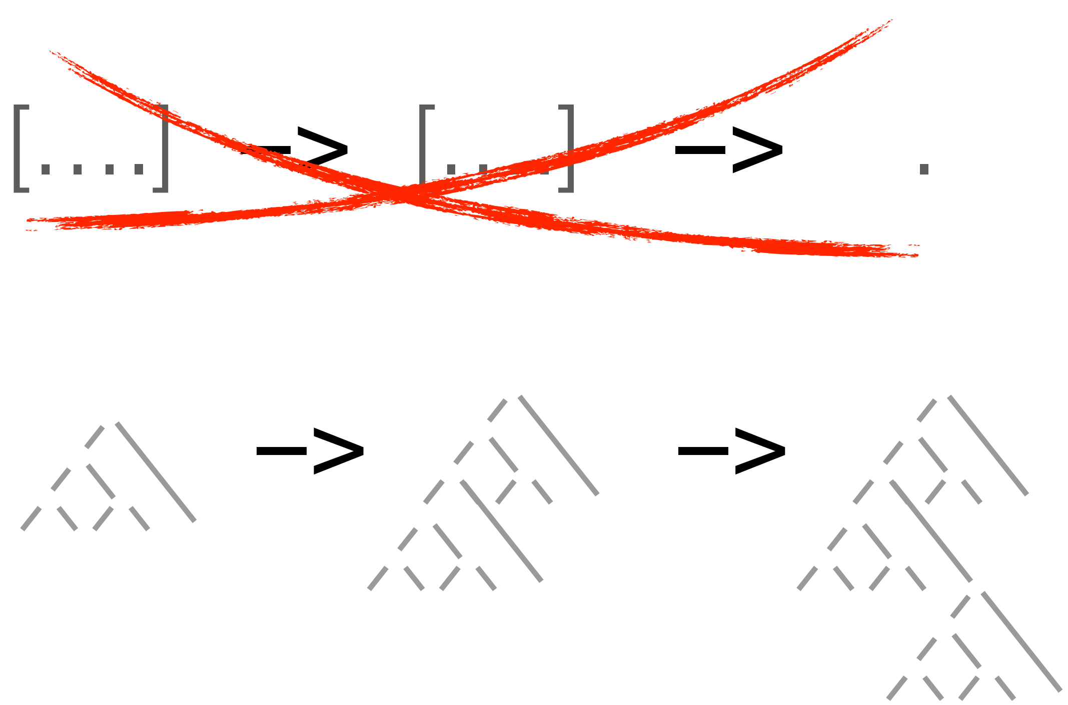
```
dotp xs ys = fold  (+) 0 (zipWith (*) xs ys)
```

Accelerate

Collective operations compiled to parallel code

```haskell
import Data.Array.Accelerate
dotp :: (Elt a, Num a)
     => Acc (Vector a)
     -> Acc (Vector a)
     -> Acc (Scalar a)
dotp xs ys = fold  (+) 0 (zipWith (*) xs ys)
```

language of collective, parallel operations



Accelerate

Collective operations which compile to parallel code

language of sequential, scalar expressions

```
fold (+) 0
```

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

language of collective, parallel operations

rank-polymorphic

To enforce hardware restrictions, nested parallel computation can't be expressed

almost

Accelerate

Collective operations which compile to parallel code

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

shape `sh` of the form `Z :. Int :. Int :. …`

```
type DIM0     = Z
type Scalar a = Array DIM0 a
```

```
type DIM1     = DIM0 :. Int
type Vector a = Array DIM1 a
```

Executing an Accelerate Program

```
run :: Arrays a => Acc a -> a
```

```
import Data.Array.Accelerate
import Data.Array.Accelerate.LLVM.Native -- CPU
```

```
vec1, vec2  :: Acc (Array DIM1 Float)

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

main =
  putStrLn $ show $ run (dotp vec1 vec2)
```

Executing an Accelerate Program

- In general, you don’t want to the system to generate new code for every input

```
run1 :: Arrays a => (Acc a -> Acc b) -> a -> b

vec1, vec2  :: Array DIM1 Float

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

main =
  putStrLn $ show $ run1 (uncurry dotp) (vec1, vec2)
```

Lifting values into the language

`Plain (Exp Int, Int) ~ (Int,Int) ~ Plain (Int, Exp Int)`

- lifting (non-overloaded) values to the expression language and back
  - `Lift e   => lift   :: e -> Exp (Plain e)`
  - `Unlift e => unlift :: Exp e -> e`
- …g, confusing

```
• Couldn't match type ‘Plain t0’ with ‘Double’
  Expected type: Exp (Plain (e0, t0))
    Actual type: Exp (Int, Double)
  The type variable ‘t0’ is ambiguous
• In the first argument of ‘unlift’, namely ‘expr’
  In the expression: unlift expr
  In a pattern binding: (a, b) = unlift expr
• Relevant bindings include
    b :: t0
```

```
erst expr = lift a
  where
    (a, b) = unlift expr :: (Exp Int, Exp Double)
```

Supported data types - the `Elt` class

- GPUs are efficient processing arrays of elementary type
- not so much for aggregate types, pointers
- similarly CPU when using SIMD vector instructions
- set of types LLVM supports is fixed
- We map the user-friendly surface types to efficient representations

Supported data types - the `Elt` class

- Using type families(i.e., functions from type to type)

```
type family EltRepr t
type instance EltRepr Int   = Int
type instance EltRepr Float = Float
type instance EltRepr (a,b) =
       ProdRepr ( EltRepr a, EltRepr b )

type family ProdRepr t
type instance ProdRepr (a,b)   = (((), a), b)
type instance ProdRepr (a,b,c) = ((((), a), b), c)
```

- Extensible: user-defined types need instances for EltRepr

Supported data types - pattern synonyms

- Predefined pattern synonyms T2, T3, … to match tuples of different arity:

```
eFst :: Exp (Int, Double) -> Exp Int
eFst (T2 a _) = a
```

Supported data types

```
data MyT a = MyT Int a
  deriving (Show, Generic)

instance Elt a => Elt (MyT a)
instance Elt a => IsProduct Elt (MyT a)

pattern MyT' :: Elt a => Exp Int -> Exp a -> Exp (MyT a)
pattern MyT' i v = Pattern (i, v)

ex1 :: Exp (MyT Int) -> Exp Int
ex1 (MyT' i v) = i * v
```

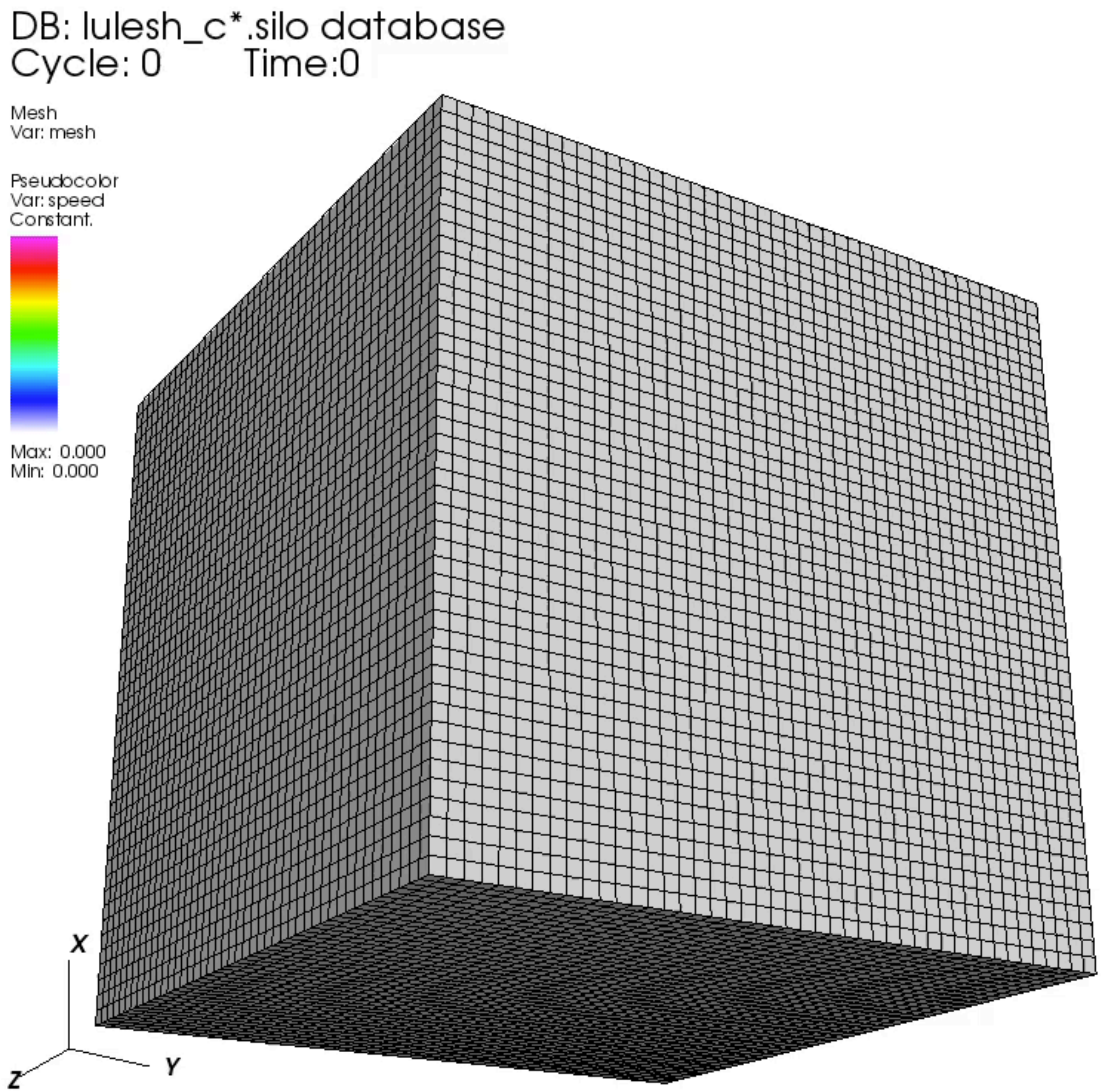
```haskell
instance Elt a => Arrays (SparseMatrix a)
instance Elt a => IsProduct Arrays (SparseMatrix a)

pattern SM' :: Elt a => Acc (Vector (Int,a))
                     -> Acc (Segments Int)
                     -> Acc (SparseMatrix a)
pattern SM' { nonzeros, segd } = Pattern (nonzeros, segd)

smvm :: A.Num a => Acc (SparseMatrix a)
                -> Acc (Vector a)
                -> Acc (Vector a)
smvm sm vec =
  let (ind, nz) = A.unzip (nonzeros sm)
  in
  foldSeg (+) 0
    (A.zipWith (*) nz (gather ind vec))
    (segd sm)
```

LULESH

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics



LULESH

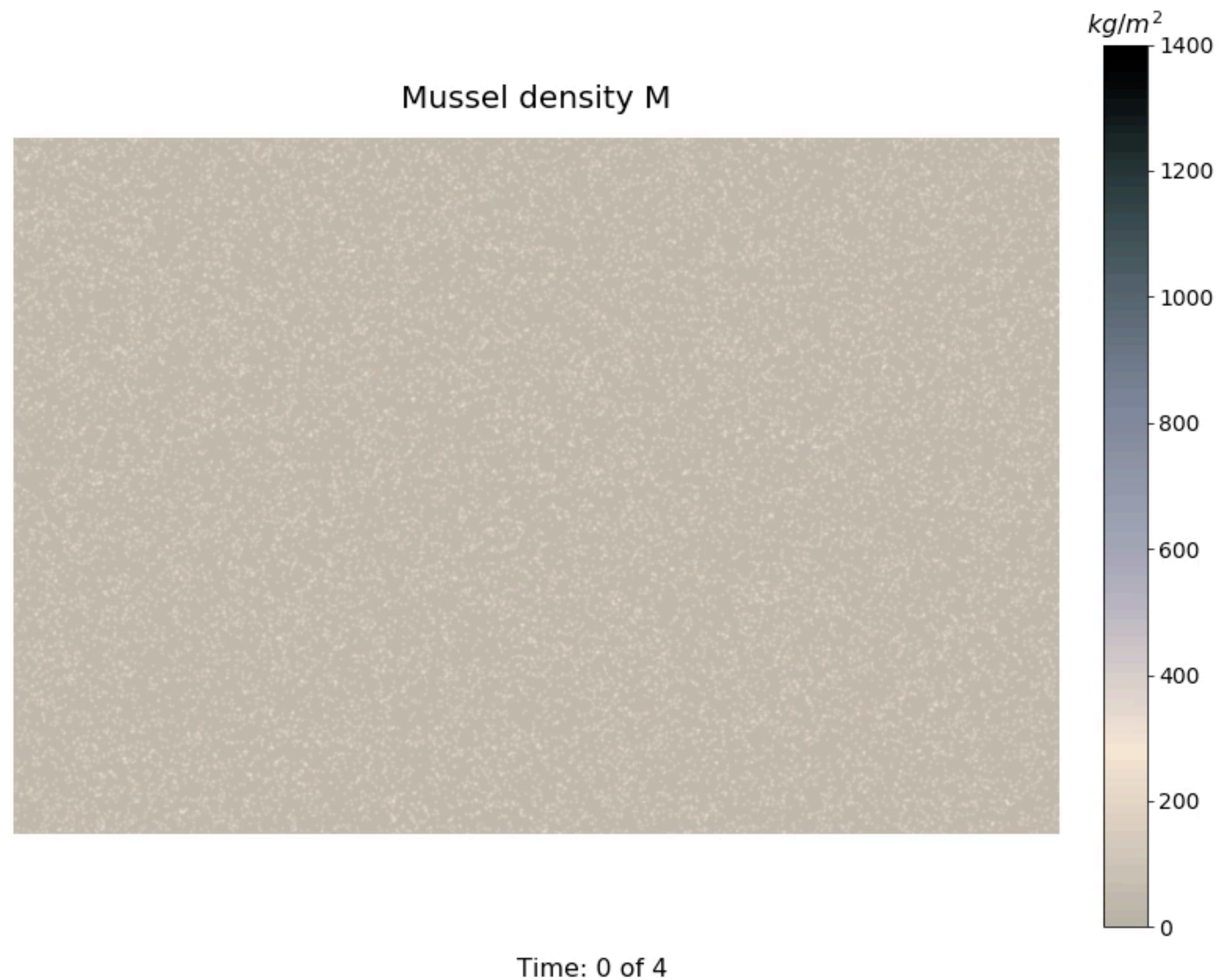
- **Implementation**
  - reference CUDA implementation: 3000 loc
  - reference OpenMP implementation: 2400 loc
  - Accelerate: 1200 loc
- **Performance**
  - reference CUDA implementation, hand optimised: 5.2s
  - Accelerate (GPU): 4.1s
  - reference OpenMP, hand optimised: 64s
  - Accelerate (CPU): 38s

Simulating the formation of spatial patterns in ecosystems

- With Johan van de Koppel, Royal Netherlands Institute for Sea Research (NIOZ)
- Formation of structures like mussel beds, salt marshes, arid bush land follows certain computational patterns
- Problems:
  - Simulation of these processes is extremely time consuming
  - Writing the simulation code is painful

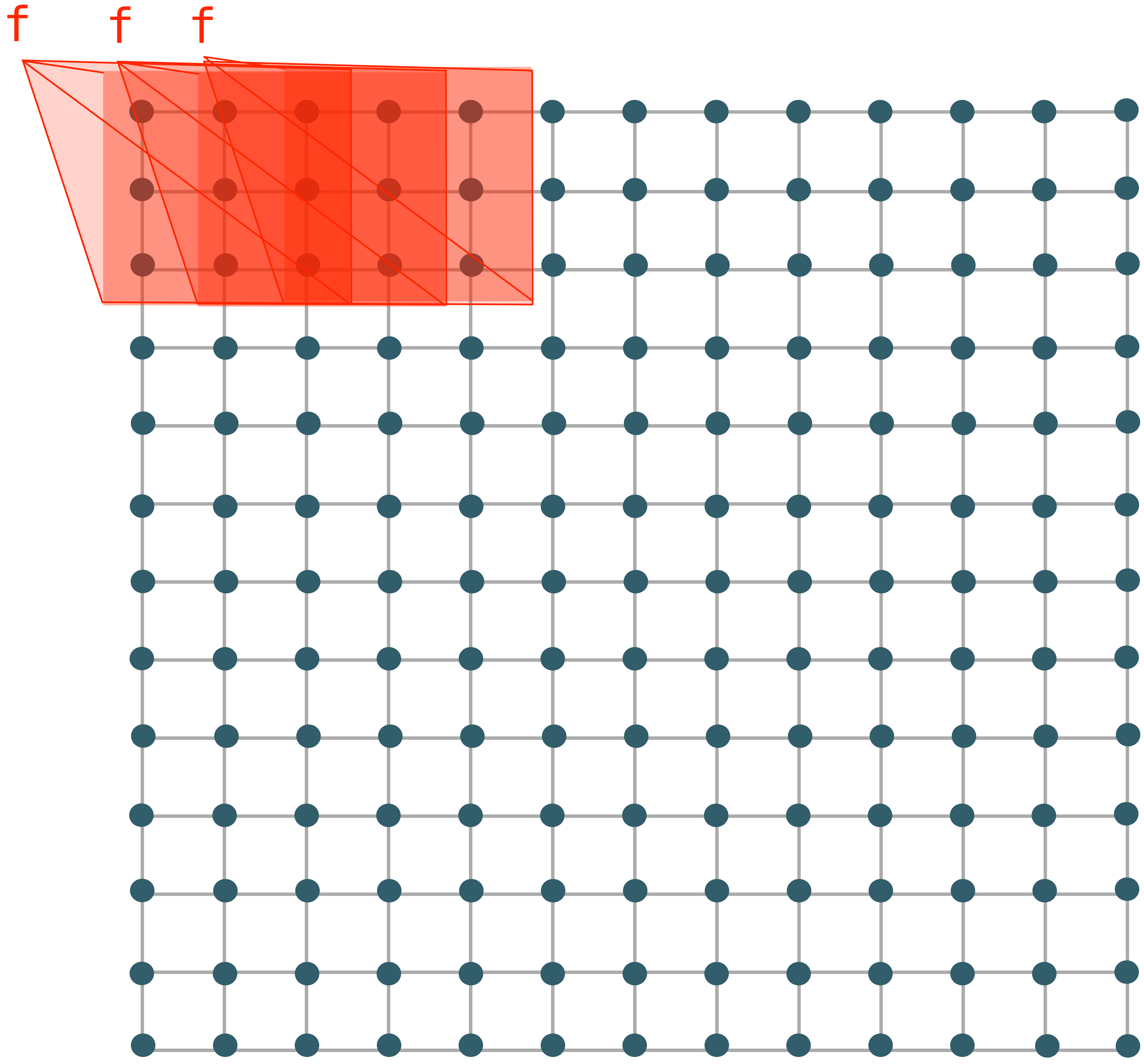
Simulating the formation of spatial patterns in ecosystems

- Combination of system like fluid-flow simulation and **Turing* pattern** computations

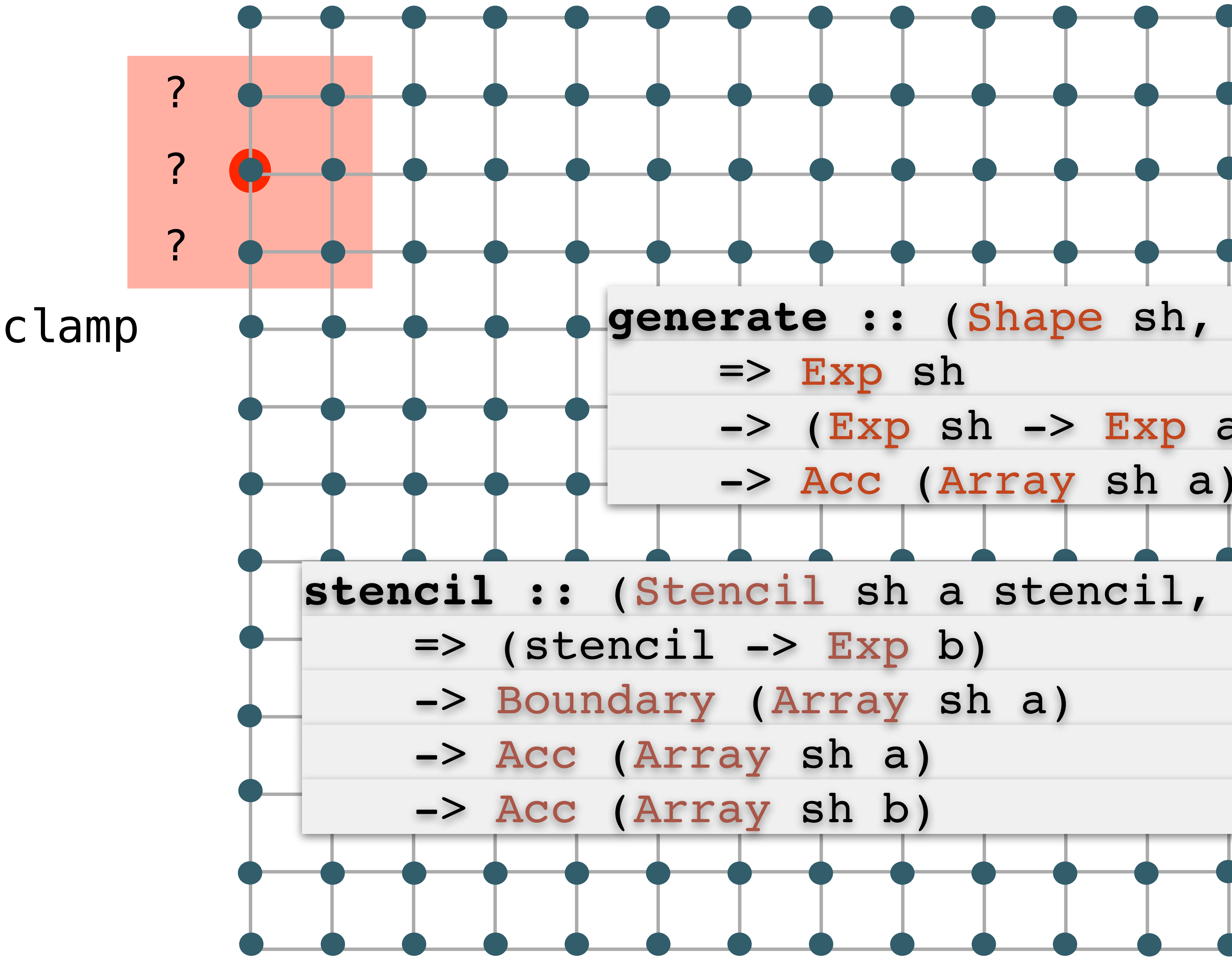


*The Chemical Basis of Morphogenesis

Stencil (convolution matrix) computations

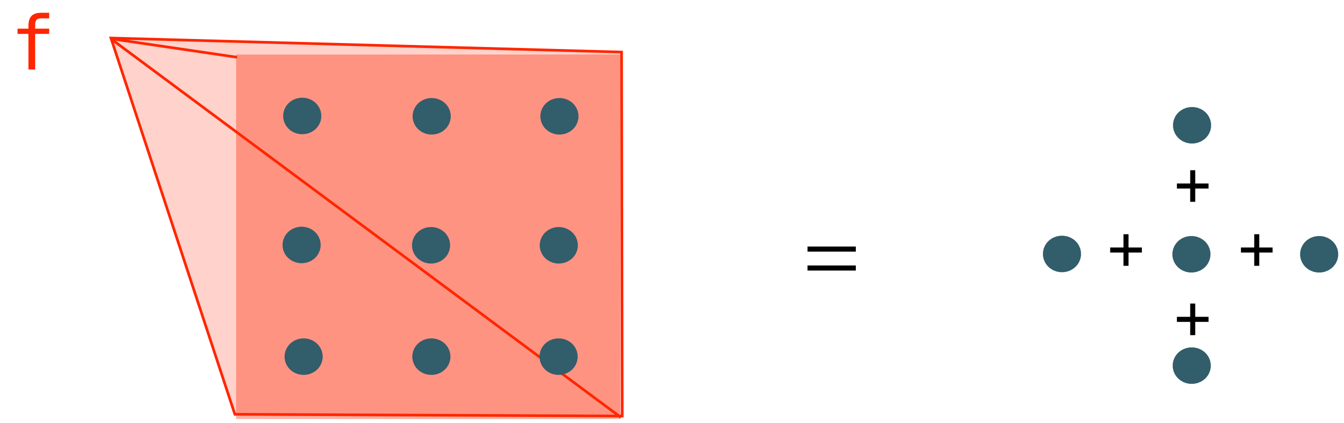


Stencil computations - boundaries




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

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



OpenCL

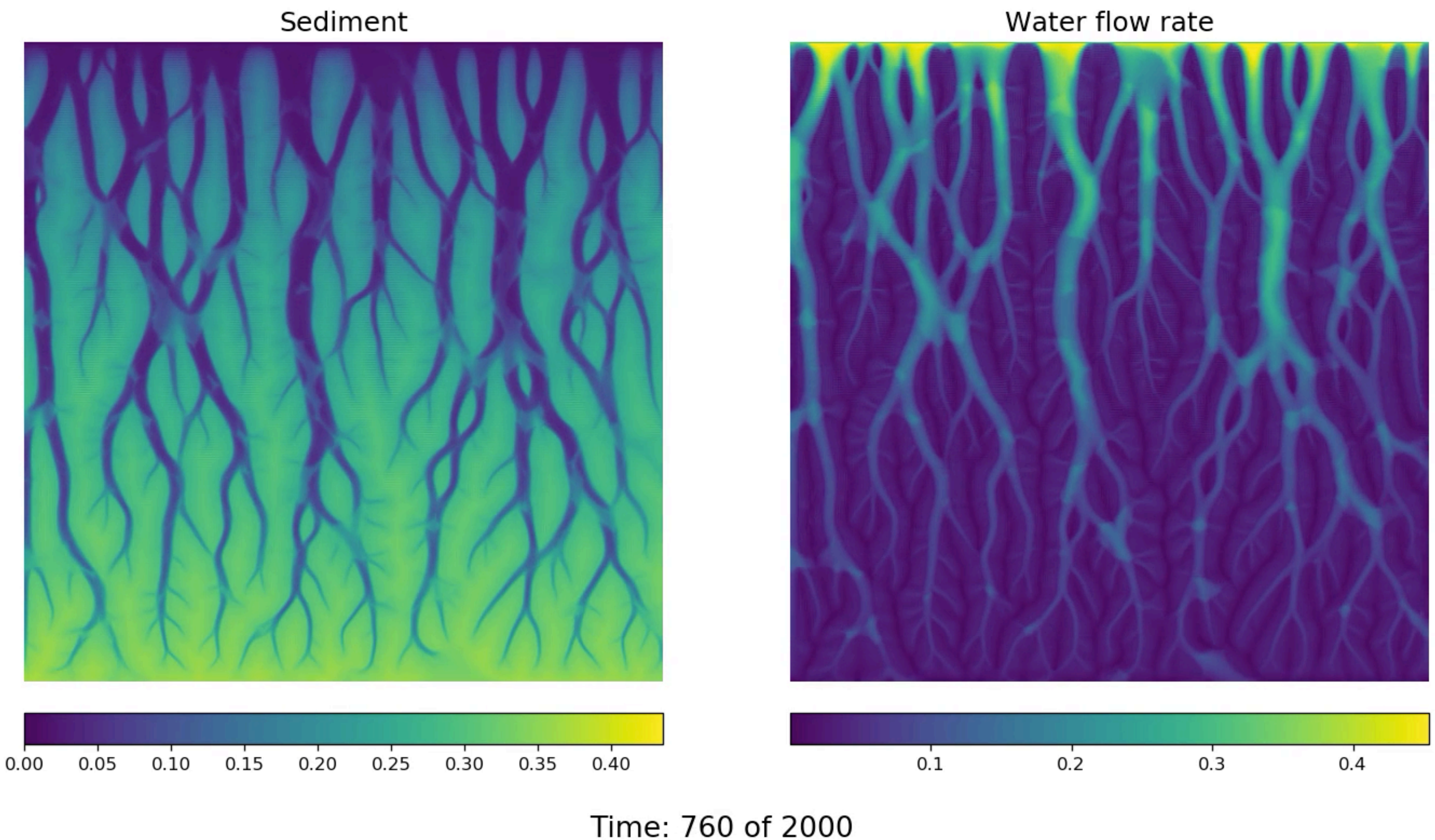
Accelerate

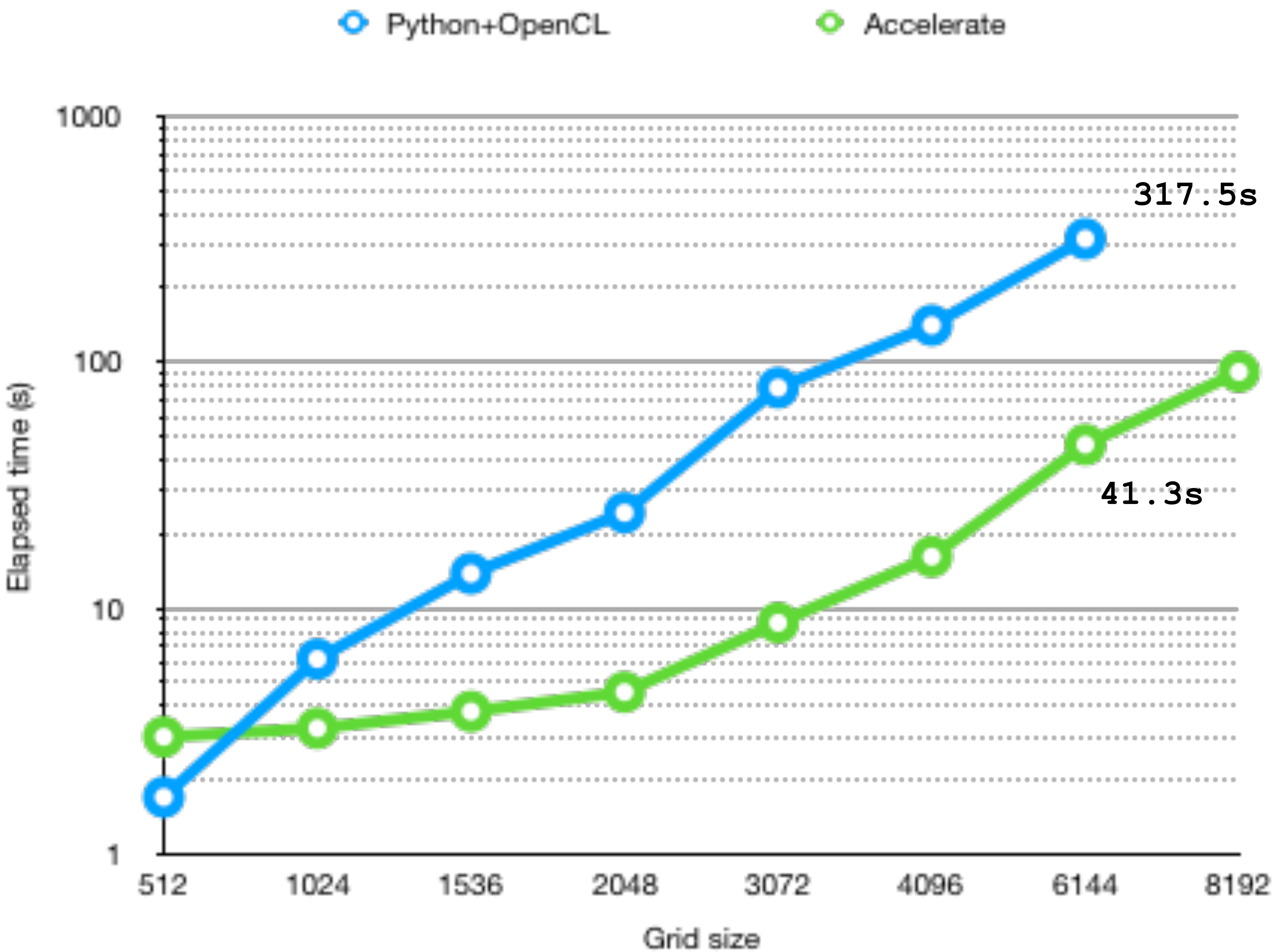
```
__kernel void simulate (__global float* arr
                       ,__global float* new_arr)
{
 const size_t cur = get_global_id(0);
 const size_t row = (size_t)cur/(size_t)Width;
 const size_t col = (size_t)cur%(size_t)Height;

 if (   row > 0 && row     < height-1
    && col > 0 && col < width-1) {
    new_arr[curr] =
      arr[cur] + arr[row * Width + col-1] … ;
 } else if (row == 0 && col < width-1) {
     …
 } else if …
```

```
simulate :: Stencil3x3 Float-> Exp Float
simulate ((_,      top,   _      ),
          (left,    curr, right   ),
          (_,       bot,   _      )) =
top + left + curr + right + bot


new_matrix
  = stencil simulate clamp matrix
```


Sediment
Water flow rate
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0.1
0.2
0.3
0.4
Time: 760 of 2000



GeForce GTX 1080 Ti

Challenges and next steps

- Our **boundary abstraction** not suitable for these applications:
  - set of predefined patterns where to source the arguments for stencil operation from (clap, wrap,…)
  - programmer can define their own pattern
  - **but**: not possible to apply different operation at the boundaries
    - has to be fixed in separate step
  - impacts performance as well as code conciseness/readability
- Even efficient simulations take long - **multi-GPU, other architectures**
- Some support for **irregular computations** is necessary to increase efficiency
- Some simulations require are based on **very large convolution matrices**