Regular Array Computation in Haskell

Geoffrey Mainland

Microsoft Research Ltd

WG 2.8, November 2012

Regular Array Computation in Haskell on GPUs

Geoffrey Mainland

Microsoft Research Ltd

WG 2.8, November 2012

Regular Array Computation in Haskell (on CPUs)

- Repa a fantastic library for writing regular array computations in Haskell.
- The type of an array tells the programmer about its representation—easier reasoning about cost.
- Automatic parallelization
- Produces very efficient code.

Regular Array Computation in Haskell (on CPUs)

- Repa a fantastic library for writing regular array computations in Haskell.
- The type of an array tells the programmer about its representation—easier reasoning about cost.
- Automatic parallelization
- Produces very efficient code.

Can we export this programming model to GPUs?

Computing the Mandelbrot Set

$$z_0 = 0$$
$$z_{i+1} = z_i^2 + c$$

- The point c is a member of the Mandelbrot set iff the z_i's are bounded.
- If $z_i > 2$ for some i, then the z_i's are not bounded, i.e., the point c escapes.
- Iterate the computation of z_i until it escapes or until we reach a fixed limit, in which case we declare that the point is an ostensible member of the Mandelbrot set

$$
\begin{aligned}
&\textbf{type}\ \mathsf{R} &&= \mathsf{Double}\\
&\textbf{type}\ \mathsf{Complex} &&= (\mathsf{R}, \mathsf{R})\\
&\textbf{type}\ \mathsf{ComplexPlane\ r} &&= \mathsf{Array\ r\ DIM2\ Complex}\\
&\textbf{type}\ \mathsf{StepPlane\ r} &&= \mathsf{Array\ r\ DIM2\ (Complex, Int)}
\end{aligned}
$$

type R = Double
type Complex = (R, R)
type ComplexPlane r = Array r DIM2 Complex
type StepPlane r = Array r DIM2 (Complex, Int)

Representation

type R $=$ Double
type Complex $=$ (R, R)
type ComplexPlane r $=$ Array r DIM2 Complex
type StepPlane r $=$ Array r DIM2 (Complex, Int)

Representation → r
Shape → DIM2

Computing c and z_1

$$
\begin{array}{l}
\mathsf{genPlane} :: \mathsf{R} \rightarrow \mathsf{R} \rightarrow \mathsf{R} \rightarrow \mathsf{R} \rightarrow \mathsf{Int} \rightarrow \mathsf{Int} \rightarrow \mathsf{ComplexPlane\ D} \\
\mathsf{genPlane\ lowx\ lowy\ highx\ highy\ viewx\ viewy} = \\
\quad \mathsf{fromFunction}\ (\mathsf{Z} :.\ \mathsf{viewy} :.\ \mathsf{viewx})\ \$\ \lambda(\mathsf{Z} :.\ (!\mathsf{y}) :.\ (!\mathsf{x})) \rightarrow \\
\qquad (\mathsf{lowx} + (\mathsf{fromIntegral\ x} * \mathsf{xsize})\ /\ \mathsf{fromIntegral\ viewx}, \\
\qquad\quad \mathsf{lowy} + (\mathsf{fromIntegral\ y} * \mathsf{ysize})\ /\ \mathsf{fromIntegral\ viewy}) \\
\quad \textbf{where} \\
\qquad \mathsf{xsize}, \mathsf{ysize} :: \mathsf{R} \\
\qquad \mathsf{xsize} = \mathsf{highx} - \mathsf{lowx} \\
\qquad \mathsf{ysize} = \mathsf{highy} - \mathsf{lowy}
\end{array}
$$

Computing c and z_1

```
genPlane :: R → R → R → R → Int → Int → ComplexPlane D
genPlane lowx lowy highx highy viewx viewy =
  fromFunction (Z :. viewy :. viewx) $ λ(Z :. (!y) :. (!x)) →
    (lowx + (fromIntegral x ∗ xsize) / fromIntegral viewx,
      lowy + (fromIntegral y ∗ ysize) / fromIntegral viewy)
  where
    xsize, ysize :: R
    xsize = highx − lowx
    ysize = highy − lowy

mkinit :: ComplexPlane U → StepPlane D
mkinit cs = map f cs
  where
    f :: Complex → (Complex, Int)
    {-# INLINE f #-}
    f z = (z, 0)
```

Computing $z_{i+1} = z_i^2 + c$

```
step :: ComplexPlane U → StepPlane U → IO (StepPlane U)
step cs zs = computeP $ zipWith stepPoint cs zs
  where
    stepPoint :: Complex → (Complex, Int) → (Complex, Int)
    {-# INLINE stepPoint #-}
    stepPoint !c (!z, !i) =
      if magnitude z' > 4.0 then (z, i) else (z', i + 1)
      where
        z' = next c z

    next :: Complex → Complex → Complex
    {-# INLINE next #-}
    next !c !z = c + (z * z)
```

Computing $z_{i+1} = z_i^2 + c$

```
step :: ComplexPlane U → StepPlane U → IO (StepPlane U)
step cs zs = computeP $ zipWith stepPoint cs zs
  where
    stepPoint :: Complex → (Complex, Int) → (Complex, Int)
    {-# INLINE stepPoint #-}
    stepPoint !c (!z, !i) =
      if magnitude z' > 4.0 then (z, i) else (z', i + 1)
      where
        z' = next c z

    next :: Complex → Complex → Complex
    {-# INLINE next #-}
    next !c !z = c + (z * z)

zipWith :: (Shape sh, Source r1 a, Source r2 b)
        ⇒ (a → b → c) → Array r1 sh a → Array r2 sh b
        → Array D sh c
```

Computing $z_{i+1} = z_i^2 + c$

```
step :: ComplexPlane U → StepPlane U → IO (StepPlane U)
step cs zs = computeP $ zipWith stepPoint cs zs
  where
    stepPoint :: Complex → (Complex, Int) → (Complex, Int)
    {-# INLINE stepPoint #-}
    stepPoint !c (!z, !i) =
      if magnitude z' > 4.0 then (z, i) else (z', i + 1)
      where
        z' = next c z

    next :: Complex → Complex → Complex
    {-# INLINE next #-}
    next !c !z = c + (z * z)
```

```
computeP :: (Load r1 sh e, Target r2 e, Source r2 e, Monad m)
         ⇒ Array r1 sh e → m (Array r2 sh e)
```

Putting it all together

```
mandelbrot :: R → R → R → R → Int → Int → Int
           → IO (StepPlane U)
mandelbrot lowx lowy highx highy viewx viewy depth = do
    cs  ← computeP $ genPlane lowx lowy highx highy viewx viewy
    zs1 ← computeP $ mkinit cs
    iterateM (step cs) depth zs1

iterateM :: Monad m ⇒ (a → m a) → Int → a → m a
iterateM f = loop
  where
    loop 0 x = return x
    loop n x = f x >>= loop (n − 1)
```

Nikola switcheroo

- Repa

```
import qualified Prelude as P
import Prelude hiding (map, zipWith)

import Data.Array.Repa
```

Nikola switcheroo

- Repa

```
import qualified Prelude as P
import Prelude hiding (map, zipWith)
import Data.Array.Repa
```

- Nikola

```
import qualified Prelude as P
import Prelude hiding (map, zipWith)
import Data.Array.Nikola.Backend.CUDA
import Data.Array.Nikola.Eval
```

Nikola switcheroo

$$
\begin{aligned}
&\textbf{type } \mathsf{R} &&= \mathsf{Double} \\
&\textbf{type } \mathsf{Complex} &&= (\mathsf{Exp\ R}, \mathsf{Exp\ R}) \\
&\textbf{type } \mathsf{ComplexPlane\ r} &&= \mathsf{Array\ r\ DIM2\ Complex} \\
&\textbf{type } \mathsf{StepPlane\ r} &&= \mathsf{Array\ r\ DIM2\ (Complex, Exp\ Int32)}
\end{aligned}
$$

Computing $z_{i+1} = z_i^2 + c$ in Nikola

```
step :: ComplexPlane G → StepPlane G → P (StepPlane G)
step cs zs = computeP $ zipWith stepPoint cs zs
  where
    stepPoint  ::  Complex
               → (Complex, Exp Int32)
               → (Complex, Exp Int32)
    stepPoint c (z, i) =
      if magnitude z' >* 4.0 then (z, i) else (z', i + 1)
      where
        z' = next c z

    next :: Complex → Complex → Complex
    next c z = c + (z * z)
```

Computing $z_{i+1} = z_i^2 + c$ in Nikola

```
step :: ComplexPlane G → StepPlane G → P (StepPlane G)
step cs zs = computeP $ zipWith stepPoint cs zs
  where
    stepPoint :: Complex
              → (Complex, Exp Int32)
              → (Complex, Exp Int32)
    stepPoint c (z, i) =
      if magnitude z' >* 4.0 then (z, i) else (z', i + 1)
      where
        z' = next c z

    next :: Complex → Complex → Complex
    next c z = c + (z * z)
```

- New representation, G.

Computing $z_{i+1} = z_i^2 + c$ in Nikola

```
step :: ComplexPlane G → StepPlane G → P (StepPlane G)
step cs zs = computeP $ zipWith stepPoint cs zs
  where
    stepPoint :: Complex
              → (Complex, Exp Int32)
              → (Complex, Exp Int32)
    stepPoint c (z, i) =
      if magnitude z' >* 4.0 then (z, i) else (z', i + 1)
      where
        z' = next c z

    next :: Complex → Complex → Complex
    next c z = c + (z * z)
```

- New representation, G.
- New monad, P.

Computing $z_{i+1} = z_i^2 + c$ in Nikola

```
step :: ComplexPlane G → StepPlane G → P (StepPlane G)
step cs zs = computeP $ zipWith stepPoint cs zs
  where
    stepPoint  ::  Complex
               → (Complex, Exp Int32)
               → (Complex, Exp Int32)
    stepPoint c (z, i) =
      if magnitude z' >* 4.0 then (z, i) else (z', i + 1)
      where
        z' = next c z

    next :: Complex → Complex → Complex
    next c z = c + (z * z)
```

- New representation, G.
- New monad, P.
- New operator, >*.

Computing c and z_1 in Nikola

```
genPlane :: Exp R → Exp R → Exp R → Exp R
         → Exp Int32 → Exp Int32
         → P (ComplexPlane G)
genPlane lowx lowy highx highy viewx viewy = computeP $
  fromFunction (Z :. viewy :. viewx) $ λ(Z :. y :. x) →
    (lowx + (fromInt x ∗ xsize) / fromInt viewx,
       lowy + (fromInt y ∗ ysize) / fromInt viewy)
  where
    xsize, ysize :: Exp R
    xsize = highx − lowx
    ysize = highy − lowy
```

Computing c and z_1 in Nikola

```
genPlane :: Exp R → Exp R → Exp R → Exp R
         → Exp Int32 → Exp Int32
         → P (ComplexPlane G)
genPlane lowx lowy highx highy viewx viewy = computeP $
  fromFunction (Z :. viewy :. viewx) $ λ(Z :. y :. x) →
    (lowx + (fromInt x ∗ xsize) / fromInt viewx,
      lowy + (fromInt y ∗ ysize) / fromInt viewy)
  where
    xsize, ysize :: Exp R
    xsize = highx − lowx
    ysize = highy − lowy

mkinit :: ComplexPlane G → P (StepPlane G)
mkinit cs = computeP $ map f cs
  where
    f :: Complex → (Complex, Exp Int32)
    f z = (z, 0)
```

Calling Nikola functions

```
import qualified Mandelbrot.NikolaV1.Implementation as I

step :: ComplexPlane CUF
     → StepPlane CUF
     → IO (StepPlane CUF)
step = $(compile I.step)

genPlane :: R → R → R → R → Int32 → Int32
         → IO (ComplexPlane CUF)
genPlane = $(compile I.genPlane)

mkinit :: ComplexPlane CUF → IO (StepPlane CUF)
mkinit = $(compile I.mkinit)
```

In-place update

```
step :: ComplexPlane G → MStepPlane G → P ()
step cs mzs = do
    zs ← unsafeFreezeMArray mzs
    loadP (zipWith stepPoint cs zs) mzs
  where
    stepPoint  ::  Complex
               → (Complex, Exp Int32)
               → (Complex, Exp Int32)
    stepPoint c (z, i) =
        if magnitude z′ >* 4.0 then (z, i) else (z′, i + 1)
        where
            z′ = next c z

    next :: Complex → Complex → Complex
    next c z = c + (z * z)
```

Iterating on the GPU

```
stepN :: Exp Int32 → ComplexPlane G → MStepPlane G → P ()
stepN n cs mzs = do
    zs ← unsafeFreezeMArray mzs
    loadP (zipWith stepPoint cs zs) mzs
  where
    stepPoint c (z, i) = iterateWhile n go (z, i)
      where
        go (z, i) = if magnitude z' >* 4.0
                    then (lift False, (z, i))
                    else (lift True, (z', i + 1))
          where
            z' = next c z

    next :: Complex → Complex → Complex
    next c z = c + (z * z)
```

Dealing with irregular workloads

```
stepN :: Exp Int32 → ComplexPlane G → MStepPlane G → P ()
stepN n cs mzs = do
    zs ← unsafeFreezeMArray mzs
    loadP (hintIrregular (zipWith stepPoint cs zs)) mzs
  where
    stepPoint c (z, i) = iterateWhile n go (z, i)
      where
        go (z, i) =
          if magnitude z′ >* 4.0
          then (lift False, (z, i))
          else (lift True, (z′, i + 1))
          where
            z′ = next c z

    next :: Complex → Complex → Complex
    next c z = c + (z ∗ z)
```

Demo

Nikola

- "Looks" like Repa, but so what?
- Allows programmer to re-use *reasoning* tools for GPU code.
- Easy interfacing to Haskell.
- Automatic partitioning of loops into separate GPU kernels.
- GPU binary code generated at compile time—no caching or run-time code generation.

Differences wrt Accelerate

- Skeletons vs. intermediate language.
- Static compilation vs. code cache.
- The P monad vs. Acc.
- Indexed types for reasoning about space/time costs.