Parallel Functional Programming
Repa

Mary Sheeran

http://www.cse.chalmers.se/edu/course/pfp
Slide borrowed from G. Keller’s lecture
Parallel arrays [: e :] (which can contain arrays)
Expressing parallelism = applying collective operations to parallel arrays

Note: demand for any element in a parallel array results in eval of all elements
DPH array operations

(!:) :: [:a:] -> Int -> a
sliceP :: [:a:] -> (Int,Int) -> [:a:]
replicateP :: Int -> a -> [:a:]
mapP :: (a->b) -> [:a:] -> [:b:]
zipP :: [:a:] -> [:b:] -> [::(a,b):]
zipWithP :: (a->b->c) -> [:a:] -> [:b:] -> [:c:]
filterP :: (a->Bool) -> [:a:] -> [:a:]
concatP :: [:[:a:]:] -> [:a:]
concatMapP :: (a -> [:b:]) -> [:a:] -> [:b:]
unconcatP :: [:[:a:]:] -> [:b:] -> [:[:b:]:]
transposeP :: [:[:a:]:] -> [:[:a:]:]
expandP :: [:[:a:]:] -> [:b:] -> [:b:]
combineP :: [:Bool:] -> [:a:] -> [:a:] -> [:a:]
splitP :: [:Bool:] -> [:a:] -> ([:a:], [:a:])
Examples

\[ \text{svMul} :: [: (Int, Float) :] \rightarrow [: \text{Float} :] \rightarrow \text{Float} \]
\[ \text{svMul} \ sv \ v = \text{sumP} [: f*(v !: i) \mid (i,f) \leftarrow \text{sv} :] \]

\[ \text{smMul} :: [:[: (Int, Float) :][:] :] \rightarrow [: \text{Float} :] \rightarrow [: \text{Float} :] \]
\[ \text{smMul} \ sm \ v = [: \text{svMul} \ \text{row} \ v \mid \text{row} \leftarrow \text{sm} :] \]

Nested data parallelism
Parallel op (svMul) on each row
Data parallelism

Perform *same* computation on a collection of *differing* data values

examples: HPF (High Performance Fortran)  
CUDA

Both support only *flat data parallelism*

Flat : each of the individual computations on (array) elements is sequential  
those computations don’t need to communicate  
parallel computations don’t spark further parallel computations
API for purely functional, collective operations over dense, rectangular, multi-dimensional arrays supporting shape polymorphism

ICFP 2010
Ideas

Purely functional array interface using collective (whole array) operations like map, fold and permutations can
- combine efficiency and clarity
- focus attention on structure of algorithm, away from low level details

Influenced by work on algorithmic skeletons based on Bird Meertens formalism (look for PRG-56)

Provides shape polymorphism not in a standalone specialist compiler like SAC, but using the Haskell type system
terminology

Regular arrays
dense, rectangular, most elements non-zero

shape polymorphic
functions work over arrays of arbitrary dimension
Regulärer Arrays
Dichte, rechteckig, meisten Elementen nicht Null

Shape polymorphism
Funktionen arbeiten über Arrays von arbiträrer Dimension

Anmerkung: die Arrays sind rein funktionell und unveränderlich
Alle Elemente eines Arrays werden gleichzeitig benötigt -> Parallelität

P Verarbeitungselemente, n Array-Elemente => n/P folgende Elemente auf jedem Prozess-Element
data Array sh e = Manifest sh (Vector e) |
                Delayed sh (sh -> e)
data Array sh e = Manifest sh (Vector e)  
  | Delayed sh (sh -> e)

class Shape sh where
  toIndex :: sh -> sh -> Int
  fromIndex :: sh -> Int -> sh
  size :: sh -> Int
  ...more operations...
data DIM1 = DIM1 !Int
data DIM2 = DIM2 !Int !Int
...more dimensions...
\[
\text{index} :: \text{Shape} \; \text{sh} \Rightarrow \text{Array} \; \text{sh} \; \text{e} \Rightarrow \text{sh} \Rightarrow \text{e}
\]
\[
\text{index} \; (\text{Delayed} \; \text{sh} \; f) \; \text{ix} = f \; \text{ix}
\]
\[
\text{index} \; (\text{Manifest} \; \text{sh} \; \text{vec}) \; \text{ix} = \text{indexV} \; \text{vec} \; (\text{toIndex} \; \text{sh} \; \text{ix})
\]
delay :: Shape sh => Array sh e -> (sh, sh -> e)
delay (Delayed sh f) = (sh, f)
delay (Manifest sh vec)
    = (sh, \ix -> indexV vec (toListIndex sh ix))
map :: Shape sh => (a -> b) -> Array sh a -> Array sh b
map f arr = let (sh, g) = delay arr
            in Delayed sh (f . g)
zipWith :: Shape sh => (a -> b -> c) -> Array sh a -> Array sh b -> Array sh c

zipWith f arr1 arr2
  = let (sh1, f1) = delay arr1
      (_sh2, f2) = delay arr2
      get ix = f (f1 ix) (f2 ix)
  in Delayed sh1 get
force :: Shape sh => Array sh e -> Array sh e
force arr
  = unsafePerformIO
    $ case arr of
      Manifest sh vec
        -> return $ Manifest sh vec
      Delayed sh f
        -> do mvec <- unsafeNew (size sh)
           fill (size sh) mvec (f . fromIndex sh)
           vec <- unsafeFreeze mvec
           return $ Manifest sh vec
Delayed (or pull) arrays  great idea!

Represent array as function from index to value

Not a new idea
Originated in Pan in the functional world I think

See also
Compiling Embedded Languages
But this is 100* slower than expected

```haskell
doubleZip :: Array DIM2 Int -> Array DIM2 Int
           -> Array DIM2 Int
doubleZip arr1 arr2
    = map (* 2) $ zipWith (+) arr1 arr2
```
Fast but cluttered

doubleZip arr1@(Manifest !_ !_) arr2@(Manifest !_ !_) = force $ map (* 2) $ zipWith (+) arr1 arr2
Things moved on!

Repa from ICFP 2010 had ONE type of array (that could be either delayed or manifest, like in many EDSLs)

A paper from Haskell’11 showed efficient parallel stencil convolution

http://www.cse.unsw.edu.au/~keller/Papers/stencil.pdf
Fancier array type (Repa 2)

```
data Array sh a
  = Array
      { arrayExtent :: sh
        , arrayRegions :: [Region sh a] }

data Region sh a
  = Region
      { regionRange :: Range sh
        , regionGen :: Generator sh a }

data Range sh
  = RangeAll
    | RangeRects
        { rangeMatch :: sh -> Bool
          , rangeRects :: [Rect sh] }

data Rect sh
  = Rect sh sh

data Generator sh a
  = GenManifest
      { genVector :: Vector a }
    | forall cursor.
      GenCursored
          { genMake :: sh -> cursor
            , genShift :: sh -> cursor -> cursor
            , genLoad :: cursor -> a }
```

Figure 5. New Repa Array Types
Fancier array type

But you need to be a guru to get good performance!
Put Array representation into the type!

The fundamental problem with Repa 1 & 2 is the following: at a particular point in the code, the programmer typically has a clear idea of the array representation they desire. For example, it may consist of three regions, left edge, middle, right edge, each of which is a delayed array. Although this knowledge is statically known to the programmer, it is invisible in the types and only exposed to the compiler if very aggressive value inlining is used. Moreover, the programmer’s typeless reasoning can easily fail, leading to massive performance degradation.

The solution is to expose static information about array representation to Haskell’s main static reasoning system; its type system.
Repá 3 (Haskell’12)

Guiding Parallel Array Fusion with Indexed Types

Ben Lippmeier†  Manuel M. T. Chakravarty†  Gabriele Keller†  Simon Peyton Jones‡

†Computer Science and Engineering
University of New South Wales, Australia
{benl,chak,keller}@cse.unsw.edu.au
‡Microsoft Research Ltd
Cambridge, England
{simonpj}@microsoft.com

Abstract
We present a refined approach to parallel array fusion that uses indexed types to specify the internal representation of each array. Our approach aids the client programmer in reasoning about the performance of their program in terms of the source code. It also makes the intermediate code easier to transform at compile-time, resulting in faster compilation and more reliable runtimes. We demonstrate how our new approach improves both the clarity and performance of several end-user written programs, including a fluid flow solver and an interpolator for volumetric data.

This second version of doubleZip runs as fast as a hand-written imperative loop. Unfortunately, it is cluttered with explicit pattern matching, bang patterns, and use of the force function. This clutter is needed to guide the compiler towards efficient code, but it obscures the algorithmic meaning of the source program. It also demands a deeper understanding of the compilation method than most users will have, and in the next section, we will see that these changes add an implicit precondition that is not captured in the function signature. The second major version of the library, Repa 2, added support for efficient parallel stencil convolution, but at the same time also increased the level of clutter needed to achieve efficient code.

http://www.youtube.com/watch?v=YmZtP11mBho

quote on previous slide was from this paper
version

I use the most recent Repa (with recent Haskell platform)
cabal update
cabal install repa

There is also repa-examples, which pulls in all Repa libraries

http://repa.ouroborus.net/

(I installed llvm and this gives some speedup, though not in my case 40% as mentioned in PCPH.)
Repa Arrays

Repa arrays are wrappers around a linear structure that holds the element data.

The representation tag determines what structure holds the data.

**Delayed Representations (functions that compute elements)**
D -- Functions from indices to elements.
C -- Cursor functions.

**Manifest Representations (real data)**
U -- Adaptive unboxed vectors.
V -- Boxed vectors.
B -- Strict ByteStrings.
F -- Foreign memory buffers.

**Meta Representations**
P -- Arrays that are partitioned into several representations.
S -- Hints that computing this array is a small amount of work, so computation should be sequential rather than parallel to avoid scheduling overheads.
I -- Hints that computing this array will be an unbalanced workload, so computation of successive elements should be interleaved between the processors.
X -- Arrays whose elements are all undefined.
10 Array representations!

- D – Delayed arrays (delayed) §3.1
- C – Cursored arrays (delayed) §4.4
- U – Adaptive unboxed vectors (manifest) §3.1
- V – Boxed vectors (manifest) §4.1
- B – Strict byte arrays (manifest) §4.1
- F – Foreign memory buffers (manifest) §4.1
- P – Partitioned arrays (meta) §4.2
- S – Smallness hints (meta) §5.1.1
- I – Interleave hints (meta) §5.2.1
- X – Undefined arrays (meta) §4.2
10 Array representations!

- D – Delayed arrays (delayed) §3.1
- C – Cursored arrays (delayed) §4.4
- U – Adaptive unboxed vectors (manifest) §3.1
- V – Boxed vectors (manifest) §4.1
- B – Strict byte arrays (manifest) §4.1
- F – Foreign memory buffers (manifest) §4.1
- P – Partitioned arrays (meta) §4.2
- S – Smallness hints (meta) §5.1.1
- I – Interleave hints (meta) §5.2.1
- X – Undefined arrays (meta) §4.2

But the 18 minute presentation at Haskell’12 makes it all make sense!! Watch it!

http://www.youtube.com/watch?v=YmZtP11mBho
Type Indexing

data family Array rep sh e

[Block diagram]

type index giving representation
Type Indexing

data family Array rep shape
Type Indexing

data family Array rep sh e

element type
map

:: (Shape sh, Source r a) =>
  (a -> b) -> Array r sh a -> Array D sh b
map

:: (Shape sh, Source r a) =>
   (a -> b) -> Array r sh a -> Array D sh b

map f arr = case delay arr of ADelayed sh g ->
   ADelayed sh (f . g)
Fusion

Delayed (and cursored) arrays enable fusion that avoids intermediate arrays

User-defined worker functions can be fused

This is what gives tight loops in the final code
Parallel computation of array elements

```
computeP :: (Load r1 sh e, Target r2 e, Source r2 e, Monad m)
=> Array r1 sh e -> m (Array r2 sh e)
```
import Data.Array.Repa as R

transpose2P :: Monad m => Array U DIM2 Double -> m (Array U DIM2 Double)
import Data.Array.Repa as R

transpose2P :: Monad m => Array U DIM2 Double -> m (Array U DIM2 Double)
example

import Data.Array.Repa as R

transpose2P :: Monad m => Array U DIM2 Double -> m (Array U DIM2 Double)
snoc lists

Haskell lists are cons lists
1:2:3:[] is the same as [1,2,3]

Repa uses snoc lists at type level for shape types
and at value level for shapes

DIM2 = Z :: Int :: Int is a shape type

Z :: i :: j read as (i,j) is an index into a two dim. array
transpose 2D array in parallel

```haskell
transpose2P :: Monad m => Array U DIM2 Double -> m (Array U DIM2 Double)
transpose2P arr = arr `deepSeqArray`
  do computeUnboxedP
    $ unsafeBackpermute new_extent swap arr
  where swap (Z :: i :: j) = Z :: j :: i
    new_extent = swap (extent arr)
```
more general transpose
(on inner two dimensions)

```haskell
transpose :: (Shape sh, Source r e) =>
    Array r ((sh :: Int) :: Int) e
            -> Array D ((sh :: Int) :: Int) e
```
more general transpose (on inner two dimensions) is provided

```haskell
transpose :: (Shape sh, Source r e) =>
    Array r ((sh :: Int) :: Int) e
-> Array D ((sh :: Int) :: Int) e
```

This type says an array with at least 2 dimensions. The function is shape polymorphic.
more general transpose (on inner two dimensions) is provided

```
transpose
:: (Shape sh, Source r e) =>
   Array r ((sh :: Int) :: Int) e
-> Array D ((sh :: Int) :: Int) e
```

Functions with at-least constraints become a parallel map over the unspecified dimensions (called rank generalisation)

Important way to express parallel patterns
Remember

Arrays of type (Array D sh a) or (Array C sh a) are *not real arrays*. They are represented as functions that compute each element on demand. You need to use `computeS`, `computeP`, `computeUnboxedP` and so on to actually evaluate the elements.

(quote from http://hackage.haskell.org/package/repa-3.4.0.1/docs/Data-Array-Repa.html which has lots more good advice, including about compiler flags)
Example: sorting

Batcher’s bitonic sort
(see lecture from last week)

“hardware-like” data-independent

bitonic sequence

inc (not decreasing)
then
dec (not increasing)
or a cyclic shift of such a sequence
Swap!
bitonic $\leq$ bitonic
Butterfly

bitonic
Butterfly

bitonic

bitonic

bitonic

>=
bitonic merger
Question

What are the work and depth (or span) of bitonic merger?
Making a recursive sorter (D&C)

Make a bitonic sequence using two half-size sorters
Batcher’s sorter (bitonic)
Let’s try to write this sorter down in Repa
bitonic merger
bitonic merger

whole array operation
`dee :: (Shape sh, Monad m) => (Int -> Int -> Int) -> (Int -> Int -> Int) -> Int
-> Array U (sh :: Int) Int
-> m (Array U (sh :: Int) Int)`

deef f g s arr = let sh = extent arr in computeUnboxedP $ fromFunction sh ixf
where
  ixf (sh :: i) = if (testBit i s) then (g a b) else (f a b)
  where
    a = arr ! (sh :: i)
    b = arr ! (sh :: (i `xor` s2))
    s2 = (1::Int) `shiftL` s

assume input array has length a power of 2, s > 0 in this and later functions
```haskell
dee :: (Shape sh, Monad m) => (Int -> Int -> Int) -> (Int -> Int -> Int)
    -> Int
    -> Array U (sh :: Int) Int
    -> m (Array U (sh :: Int) Int)
dee f g s arr = let sh = extent arr in computeUnboxedP $ fromFunction sh ixf
    where
        ixf (sh :: i) = if (testBit i s) then (g a b) else (f a b)
            where
                a = arr ! (sh :: i)
                b = arr ! (sh :: (i `xor` s2))
                s2 = (1::Int) `shiftL` s
```

dee f g 3 gives index i matched with index (i xor 8)
bitonicMerge n = compose [dee min max (n-i) | i <- [1..n]]
tmerge
vee :: (Shape sh, Monad m) => (Int -> Int -> Int) -> (Int -> Int -> Int) -> Int
    -> Array U (sh ::. Int) Int
    -> m (Array U (sh ::. Int) Int)
vee f g s arr = let sh = extent arr in computeUnboxedP $ fromFunction sh ixf
    where
        ixf (sh ::. ix) = if (testBit ix s) then (g a b) else (f a b)
            where
                a = arr ! (sh ::. ix)
                b = arr ! (sh ::. newix)
                newix = flipLSBsTo s ix
vee :: (Shape sh, Monad m) => (Int -> Int -> Int) -> (Int -> Int -> Int) -> m (Array U (sh :: Int) Int)

vee f g s arr = let sh = extent arr in computeUnboxedP $ fromFunction sh ixf
  where
    ixf (sh :: ix) = if (testBit ix s) then (g a b) else (f a b)
      where
        a = arr ! (sh :: ix)
        b = arr ! (sh :: newix)
        newix = flipLSBSTo s ix

vee f g 3
  out(0) -> f  a(0)  a(7)
  out(7) -> g  a(7)  a(0)
  out(1) -> f  a(1)  a(6)
  out(6) -> g  a(6)  a(1)
tmerge

tmerge n = compose $ vee min max (n-1) : [dee min max (n-i) | i <- [2..n]]
tsort n = compose [tmerge i | i <- [1..n]]
Question

What are work and depth of this sorter??
Performance is decent!

Initial benchmarking for $2^{20}$ Ints

Around 800ms on 4 cores on this laptop

Compared to around 1.6 seconds for Data.List.sort (which is sequential)

Still slower than Persson’s non-entry from the sorting competition in the 2012 course (which was at 400ms) -- a factor of a bit under 2, which is about what you would expect when comparing Batcher’s bitonic sort to quicksort.
Comments

Should be very scalable

Can probably be sped up! Need to add sequentialness 😊

Similar approach might greatly speed up the FFT in repa-examples
(and I found a guy running an FFT in Haskell competition)

Note that this approach turned a nested algorithm into a flat one

Idiomatic Repa (written by experts) is about 3 times slower.
Genericity costs here!

Message: map, fold and scan are not enough. We need to think more about higher order functions on arrays (e.g. with binary operators)
Repa’s real strength

Stencil computations!

```haskell

[stencil2| 0 1 0
  1 0 1
  1 0 1
  0 1 0 |]
```

do
(r, g, b) <- liftM (either (error . show) R.unzip3) readImageFromBMP "in.bmp"
[r', g', b'] <- mapM (applyStencil simpleStencil) [r, g, b]
writeImageToBMP "out.bmp" (U.zip3 r' g' b')
```
Repa’s real strength

Nice success story at NYT

Haskell in the Newsroom

Haskell in Industry
is your friend

See for example

Conclusions

Based on DPH technology

Good speedups!

Neat programs

Good control of Parallelism

BUT  CACHE AWARENESS needs to be tackled
Conclusions

Development seems to be happening in Accelerate, which now works for both multicore and GPU (work ongoing)

Array representations for parallel functional programming is an important, fun and frustrating research topic 😊
Questions to think about

What is the right set of whole array operations?

(remember Backus from the first lecture)