Paralelismo en Datos

Programación Funcional Avanzada

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

Paralelismo en Haskell

. . . la historia hasta ahora

- Paralelismo explícito usando `forkIO` y `MVars`.
  - Complejidad semántica para controlar la mutación del estado.
  - Difícil razonar sobre las relaciones.
  - Control preciso sobre la granularidad del trabajo.

UNIVERSIDAD SIMÓN BOLÍVAR

Paralelismo en Haskell

. . . la historia hasta ahora

- Paralelismo explícito usando `forkIO` y `MVars`.
  - Complejidad semántica para controlar la mutación del estado.
  - Difícil razonar sobre las relaciones.
  - Control preciso sobre la granularidad del trabajo.
- Paralelismo explícito usando `forkIO` y `STM`.
  - Control automático y riguroso sobre la mutación del estado.
  - Habilidad de componer transacciones.

UNIVERSIDAD SIMÓN BOLÍVAR

Paralelismo en Haskell

. . . la historia hasta ahora

- Paralelismo explícito usando `forkIO` y `MVars`.
  - Complejidad semántica para controlar la mutación del estado.
  - Difícil razonar sobre las relaciones.
  - Control preciso sobre la granularidad del trabajo.
- Paralelismo explícito usando `forkIO` y `STM`.
  - Control automático y riguroso sobre la mutación del estado.
  - Habilidad de componer transacciones.
- Paralelismo implícito usando estrategias o Monad `Par`.
  - Hilos implícitos controlados por el RTS.
  - Operan es espacio puro – transparencia referencial.
  - Difícil asegurar granularidad adecuada de manera consistente.

UNIVERSIDAD SIMÓN BOLÍVAR

Paralelismo en Haskell

. . . la historia hasta ahora

- Paralelismo explícito usando `forkIO` y `MVars`.
  - Complejidad semántica para controlar la mutación del estado.
  - Difícil razonar sobre las relaciones.
  - Control preciso sobre la granularidad del trabajo.
- Paralelismo explícito usando `forkIO` y `STM`.
  - Control automático y riguroso sobre la mutación del estado.
  - Habilidad de componer transacciones.
- Paralelismo implícito usando estrategias o Monad `Par`.
  - Hilos implícitos controlados por el RTS.
  - Operan es espacio puro – transparencia referencial.
  - Difícil asegurar granularidad adecuada de manera consistente.
- Difíciles de implantar en memoria distribuida.
  - Apuntadores libres – pobre localidad espacial.
  - Difícil obtener desempeño *predecible* y *escalable*.

UNIVERSIDAD SIMÓN BOLÍVAR

Paralelismo de Datos

Aplicar la misma operación, en paralelo,
sobre cada elemento de una colección grande.

- Aplicable a problemas que pueden expresarse con:
  - Pureza funcional – semántica transparente.
  - Buena granularidad – un hilo por procesador.
  - Excelente localidad – particionamiento simple de los datos.
- Buen modelo de desempeño con escalabilidad predecible – ha sido objeto de estudio por décadas por los "calculistas".
- Se ha venido usando de forma limitada en otros lenguajes – *High Performance Fortran*, OpenMP, MPI para C, ...

UNIVERSIDAD SIMÓN BOLÍVAR

Paralelismo de Datos

Condiciones diferentes

- Adoptar lo que se ha venido usando en otros lenguajes.
- Aplicar la misma función pura y estricta, pero en paralelo, sobre cada elemento de una colección grande *finita*.
  - Grandes. Muy grandes.
  - Como son finitas, se les llama *vectores*
- La evaluación debe costar más o menos lo mismo por elemento para que la técnica tenga buenos resultados de desempeño.
  - Se necesita *excelente* desempeño secuencial.
  - No sirven las listas estándar – estructuras *unboxed* especializadas.

UNIVERSIDAD SIMÓN BOLÍVAR

Regular Parallel Arrays (REPA)

These are the vectors you're looking for

- REPA – operaciones para crear y operar sobre arreglos finitos.
- Operaciones eficientes paralelizadas por la librería.
- Importar calificado –
  diferenciar de operaciones provistas en `Prelude` y `Data.List`.
- Tipo de datos abstracto opaco para arreglos –
  no confudir con los arreglos estándar de Haskell.

```
data Array r sh e
```

UNIVERSIDAD SIMÓN BOLÍVAR

Arreglos en REPA

La eficiencia está en los detalles

```
data Array r sh e
```

- `r` – **representación** del arreglo en memoria.
  - `U` – *unboxed* vectors.
  - `D` – *delayed* vectors como funciones de índices a valores.
  - Cosas aún más sofisticadas.
- `e` – elementos almacenados en el arreglo (`Int`, `Double`, `Word8`).
- `sh` – la **forma** del arreglo.
  - Número de dimensiones.
  - Condiciones sobre las dimensiones.

UNIVERSIDAD SIMÓN BOLÍVAR

La forma de los arreglos

- Se expresa empleando dos constructores

```
data Z = Z
data tail :. head = tail :. head
```

- `Z` es un arreglo sin dimensiones – un valor escalar.
- `Z :. Int` – agrega una dimensión `Int`.
- Aliases de tipo para simplificar la construcción

```
type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int
```

- Las dimensiones sólo pueden ser `Int`

UNIVERSIDAD SIMÓN BOLÍVAR

Construyendo arreglos REPA

De listas a vectores

- Pasar una lista tradicional a vector unidimensional

```
fromListUnboxed :: (Shape sh, Unbox a)
                => sh -> [a] -> Array U sh a
```

- Siempre se puede probar en GHCi

```
> fromListUnboxed (Z :. 10)
                  [1..10] :: Array U DIM1 Int
AUnboxed (Z :. 10) (fromList [1,2,3,4,5,6,7,8,9,10])
```

- `AUnboxed` – datos *unboxed*
- `Z :. 10` – una dimensión de 10 posiciones.

UNIVERSIDAD SIMÓN BOLÍVAR

Construyendo arreglos REPA

De listas a matrices

- Pasar una lista tradicional a vector bidimensional

```
fromListUnboxed :: (Shape sh, Unbox a)
                => sh -> [a] -> Array U sh a
```

- Basta cambiar la forma del arreglo

```
> fromListUnboxed (Z :. 3 :. 5)
                  [1..15] :: Array U DIM2 Int
AUnboxed ((Z :. 3) :. 5) (fromList [1,...,15])
```

- Matriz de 3 filas y 5 columnas.
  - Conceptualmente

$$\begin{matrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{matrix}$$

  - Internamente es **un** vector contiguo.

UNIVERSIDAD SIMÓN BOLÍVAR

Accediendo a elementos en los vectores

- El operador `(!)`

```
(!) :: (Shape sh, Source r e)
    => Array r sh e -> sh -> e
```

- Se usan `Z` y `(:.)` como *valores* en este caso – no confundirse.

```
> let arr = fromListUnboxed
                (Z :. 3 :. 5)
                [1..15] :: Array U DIM2 Int
> arr ! (Z :. 2 :. 1)
12
```

- Recordando que internamente son vectores

```
toIndex :: Shape sh => sh -> sh -> Int

> toIndex (Z:.5:.3 :: DIM2) (Z:.2:.1 :: DIM2)
11
```

UNIVERSIDAD SIMÓN BOLÍVAR

Operando con la forma del arreglo

Total, es una interpreación de espacio contiguo

- No hace falta copiar el arreglo para reinterpretarlo

```
reshape :: (Shape sh1, Shape sh2, Source r1 e)
        => sh2 -> Array r1 sh1 e -> Array D sh2 e

> reshape (Z:.5:.3 :: DIM2) arr ! (Z:.2:.1 :: DIM2)
8
```

- Obteniendo información sobre la forma de un arreglo

```
extent :: (Shape sh, Source r e)
       => Array r sh e -> sh
rank :: Shape sh => sh -> Int
size :: Shape sh => sh -> Int
```

  - `extent` – la forma completa.
  - `rank` – número de dimensiones.
  - `size` – cantidad total de elementos.

UNIVERSIDAD SIMÓN BOLÍVAR

Operaciones con vectores

Como si fueran listas...

- Aplicar una función pura sobre *todos* los elementos

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

- Vector resultante es Delayed – vector resultado "por demanda".
- Combinar varios map en secuencia no evalúa **nada** hasta que se aplique una función de evaluación definitiva.
  - **Nunca** se calculan los vectores intermedios.
  - Se aplica la **fusión** funcional pura.
  - *Lazy Computation Pipeline*

UNIVERSIDAD SIMÓN BOLÍVAR

Forzando la construcción del resultado

Sin los pasos intermedios

```
computeS :: (Target r2 e, Load r1 sh e)
         => Array r1 sh e -> Array r2 sh e
```

- Calcular de manera secuencial

```
> computeS
    (map (+1)
      (map (^2) arr)) :: Array U DIM2 Int
AUnboxed ((Z :. 3) :. 5) (fromList [2,...,226])
```

- Compila a un ciclo secuencial sobre el vector.
- El cuerpo del ciclo tiene el equivalente a $(\lambda x -> x^2 + 1)$

UNIVERSIDAD SIMÓN BOLÍVAR

Arreglos diferidos

Operación sin datos

- Con una forma y una función de índices a valores

```
fromFunction :: sh -> (sh -> a) -> Array D sh

> let a = fromFunction (Z:.10)
                       (\(Z:.i) -> i :: Int)
```

- Función se aplica sobre los índices – noten el *pattern matching*.
- El vector Diferido no se evalúa inmediatamente.
- Se puede pedir un elemento específico –
  se evalúa la función en ese momento sobre ese índice.

```
> a ! (Z:.5)
5
```

UNIVERSIDAD SIMÓN BOLÍVAR

Caminos más cortos en un grafo

Floyd-Warshall sobre grafos densos

- Representaremos grafos con matrices de adyacencia
  - Arreglo de dos dimensiones indizado por vértices.
  - Cada elemento es la longitud del camino entre ambos vértices.
- Definiremos un par de aliases para los tipos

```
type Weight   = Int
type Graph r = Array r DIM2 Weight
```

UNIVERSIDAD SIMÓN BOLÍVAR

Caminos más cortos en un grafo

Programación dinámica vectorizada

```
shortestPaths :: Graph U -> Graph U
shortestPaths g0 = go g0 0
  where
   Z :. _ :. n = extent g0
   go !g !k | k == n     = g
            | otherwise =
      let g' = computeS $ fromFunction (Z:.n:.n) sp
      in go g' (k+1)
        where
          sp (Z:.i:.j) = min (g ! (Z:.i:.j))
                             (g ! (Z:.i:.k) +
                              g ! (Z:.k:.j))
```

- El número de vértices se determina con *pattern matching*.
- go es recursiva de cola y estricta en ambos argumentos – arreglo acumulador y vértice a procesar.

UNIVERSIDAD SIMÓN BOLÍVAR

Caminos más cortos en un grafo

Fast and curious

- El programa principal construye grafos densos

```
main = do
  [n] <- fmap (fmap read) getArgs
  let g = fromListUnboxed
            (Z:.n:.n)
            [0..n^(2::Int)-1] :: Graph U
  print (sumAllS (shortestPaths g))
```

- Compilamos y corremos

```
$ ghc -O2 --make fw
```

- Si se dispone del *backend* LLVM, tanto mejor

```
$ ghc -O2 --make fw -fllvm
```

UNIVERSIDAD SIMÓN BOLÍVAR

Can I has faster?

Fast and serious

- El tamaño del vector es conocido, por tanto es posible dividir con precisión la carga de cómputo entre varios núcleos.
- El vector está dispuesto linealmente y contiguo en memoria, por tanto se maximiza la coherencia del caché de cada núcleo.
- Sólo hay que cambiar la función de manifestación

```
computeP :: (Monad m, Source r2 e,
             Target r2 e, Load r1 sh e)
         => Array r1 sh e -> m (Array r2 sh e)
```

- *Cualquier* monad – sólo interesa la la garantía de secuenciación.

UNIVERSIDAD SIMÓN BOLÍVAR

Return of the Identity

. . . porque todo este código es puro

- Modificaciones triviales al código
  - `runIdentity` para la llamada auxiliar.
  - `return` para el grafo final.
  - Usar un bloque `do` para el `let/in`.
- Compilar y ejecutar con múltiples núcleos

```
$ ghc -O2 --make -fllvm -fforce-recomp -threaded fw
$ fw +RTS -N -RTS 500
```

UNIVERSIDAD SIMÓN BOLÍVAR

Catamorfismos Vectorizados

Acumulación de resultados

- Cálculo de *folds* homogéneos sobre vectores – secuencial y paralelo.

```
foldS :: (Shape sh, Source r a,
          Unbox a, Elt a)
      => (a -> a -> a) -> a ->
         Array r (sh :. Int) a -> Array U sh a
foldP :: (Monad m,
          Shape sh, Source r a,
          Unbox a, Elt a)
      => (a -> a -> a) -> a ->
         Array r (sh :. Int) a -> m (Array U sh a)
```

- Si los contenidos son numéricos...
  - `sumS` y `sumP` – vector con suma de filas.
  - `sumAllS` y `sumAllP` – suma de todos los elementos.

UNIVERSIDAD SIMÓN BOLÍVAR

Aún más rápido...

Warp speed, Mr. GPU

- Un GPU moderno usualmente es uno o dos órdenes de magnitud más potente para cálculo numérico.
- El GPU emplea un modelo de cómputo notablemente diferente a la combinación CPU/FPU.
  - Muchísimas unidades de procesamiento más lentas que el CPU.
  - Todas ejecutan *exactamente* la misma instrucción simultáneamente.
  - Deben programarse de manera diferente al CPU/FPU.
- CUDA y OpenCL son lenguajes de programación específicos – muy bajo nivel, detalles de operación, compilador especial...

UNIVERSIDAD SIMÓN BOLÍVAR

Accelerate

`Data.Array.Accelerate`

- Accelerate es un lenguaje de dominio específico (EDSL) embebido en Haskell.
  - Comparte conceptos con REPA – arreglos, forma, índices.
  - Folds y operaciones más complejas.
- Se escribe Haskell con instrucciones estilizadas para GPU.
- Compilable e interpretable – probar cosas desde GHCi.
- `Data.Array.Accelerate` – construir cómputos.
- `Data.Array.Accelerate.Interpreter` – interpretarlos cuando no se cuenta con un GPU.
- *Backends* para emitir CUDA y enviarlo al GPU.

UNIVERSIDAD SIMÓN BOLÍVAR

Arreglos e índices

Similares a REPA

- Los arreglos no requieren representación explícita

```
data Array sh e
```

- La forma y los índices se expresan igual que en REPA

```
data Z = Z
data tail :. head = tail :. head
```

- Los vectores se pueden construir de manera similar

```
fromList :: (Shape sh, Elt e)
         => sh -> [e] -> Array sh e
```

UNIVERSIDAD SIMÓN BOLÍVAR

Ejecutando cómputos con Accelerate

Run, Forrest! Run!

- Un cómputo acelerado que produce valores de tipo `a`

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

  - `run` de `Data.Array.Accelerate.Interpreter` – pruebas.
  - `run` de `Data.Array.Accelerate.CUDA` – GPU.

- `Acc` representa el contexto de cómputo

```
use :: Arrays arrays => arrays -> Acc arrays
```

permite "inyectar" vectores desde Haskell hasta el contexto.

UNIVERSIDAD SIMÓN BOLÍVAR

Un cómputo acelerado simple

Para comparar con REPA

```
> let arr = fromList (Z:.3:.5) [1..] :: Array DIM2 Int
> run $ map (+1) (use arr)
Array (Z :. 3 :. 5) [2,...,16]
```

- El `run` particular interpreta o ejecuta en CUDA.
- En el código real, debería calificar `use` y `map`.

Este `map` es curioso

Porque debe llevar cómputo al GPU

```
map :: (Shape ix, Elt a, Elt b)
    => (Exp a -> Exp b)
    -> Acc (Array ix a)
    -> Acc (Array ix b)
```

- `Acc` indica que el vector de entrada y de salida son cómputos en el contexto `Accelerate`.
- `Exp` indica que el valor de entrada y de salida son calculados en el contexto `Accelerate`.
- `Accelerate` provee una instancia `Num (Exp a)` – usar constantes y operaciones numéricas directamente.

UNIVERSIDAD SIMÓN BOLÍVAR

Vectores dentro de `Acc`

Porque `fromList` es muy caro

- Vectores con valor idéntico

```
fill :: (Shape sh, Elt e)
     => Exp sh -> Exp e -> Acc (Array sh e)
```

- Hay que suministrar la firma de tipos como en REPA

```
> run $ fill (index2 3 3) 42 :: Array DIM2 Int
Array (Z :. 3 :. 3) [42,...,42]
```

UNIVERSIDAD SIMÓN BOLÍVAR

Vectores dentro de `Acc`

Porque `fromList` es muy caro

- Vectores desde enumeraciones

```
enumFromN :: (IsNum e, Shape sh, Elt e)
          => Exp sh
          -> Exp e -> Acc (Array sh e)
enumFromStepN :: (IsNum e, Shape sh, Elt e)
              => Exp sh -> Exp e
              -> Exp e -> Acc (Array sh e)
```

- Hay que suministrar la firma de tipos como en REPA

```
> run $ enumFromStepN (index2 3 5)
                      15 (-1) :: Array DIM2 Int
Array (Z :. 3 :. 5) [15,14,...,1]
```

UNIVERSIDAD SIMÓN BOLÍVAR

Vectores dentro de `Acc`

Porque `fromList` es muy caro

- Vectores generados a partir de los índices

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

- Hay que suministrar la firma de tipos como en REPA

```
> run $ generate
          (index2 3 5)
          (\ix -> let Z:.y:.x = unlift ix
                  in x + y) :: Array DIM2 Int
Array (Z :. 3 :. 5) [0,1,2,3,4,1,2,3,4,5,2,3,4,5,6]
```

- `unlift` asiste en tomar valores dentro de un `Exp`.

UNIVERSIDAD SIMÓN BOLÍVAR

Caminos más cortos en un grafo

Ajustando la representación

```
type Weight = Int32
type Graph  = Array DIM2 Weight
```

- `Accelerate` está particularmente optimizado para los tipos nativos – usaremos `Int32`
- Ajustamos para vectores `Accelerate` – eliminar la representación interna.

UNIVERSIDAD SIMÓN BOLÍVAR

Caminos más cortos en un grafo

Paso a paso

```
step :: Acc (Scalar Int) -> Acc Graph -> Acc Graph
step k g = generate (shape g) sp
  where
    k'     = the k
    sp :: Exp DIM2 -> Exp Weight
    sp ix = let (Z:.i:.j) = unlift ix
            in min (g ! (index2 i j))
                   (g ! (index2 i k') +
                    g ! (index2 k' j))
```

- Se procesa el $k-$ésimo nodo – `Scalar Int`
- Usamos `unlift` para extraer de `Exp`.
- Usamos `the` para convertir `Scalar Int` en `Exp e`.
- Con `index2` construimos un subíndice por partes.

UNIVERSIDAD SIMÓN BOLÍVAR

Caminos más cortos en un grafo

Composición de pasos

```
shortestPathsAcc :: Int -> Acc Graph -> Acc Graph
shortestPathsAcc n g0 = foldl1 (>->) steps $ g0
  where
    steps :: [ Acc Graph -> Acc Graph ]
    steps = [step (unit (constant k)) | k <- [0..n-1]]
```

- Lista de pasos
  - Aplicación paracial de `step` a cada nodo – currying.
  - m `constant` y `unit` para construir un `Acc (Scalar e)`
- *Pipeline Operator* `(>->)`

```
(>->) :: (Arrays a, Arrays b, Arrays c)
      => (Acc a -> Acc b) -> (Acc b -> Acc c)
      -> Acc a -> Acc c
```

Conecta cómputos `Acc` indicando que no hay que conservar datos intermedios – consumo de memoria casi constante.

UNIVERSIDAD SIMÓN BOLÍVAR

Caminos más cortos en un grafo

Ejecutando la transformación

```
shortestPaths :: Graph -> Graph
shortestPaths g0 = run (shortestPathsAcc n (use g0))
  where
    Z :. _ :. n = arrayShape g0
```

- `run` – interpretar o enviar a CUDA según la librería en uso.
- `use` para llevar el vector al contexto `Acc`.
- `arrayShape` opera en contexto puro fuera de `Acc`.

UNIVERSIDAD SIMÓN BOLÍVAR

Caminos más cortos en un grafo

El programa principal – el generador

```
main = do
 (n:_) <- fmap (fmap read) getArgs
 print (run
   (let g :: Acc Graph
        g = generate (constant (Z:.n:.n) :: Exp DIM2)
                     f

        f :: Exp DIM2 -> Exp Weight
        f ix = let i,j :: Exp Int
                   Z:.i:.j = unlift ix
               in
                   A.fromIntegral j +
                   A.fromIntegral i *
                   constant (Prelude.fromIntegral n)
```

- g – genera el vector que representa un grafo denso.
- f – genera el costo entre i y j.

UNIVERSIDAD SIMÓN BOLÍVAR

Caminos más cortos en un grafo

El programa principal – el procesador

```
...
in
A.foldAll (+) (constant 0) (shortestPathsAcc n g)))
```

- `foldAll` – catamorfismo en `Accelerate`

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

- g – se ejecuta una vez generada.
- Separación entre la generación de cómputos y la evaluación.

`Accelerate` es un compilador –
`Acc` construye el AST y `run` emite código

UNIVERSIDAD SIMÓN BOLÍVAR

Una comparación apurada...

Porque son las 08:50 y aún estoy haciendo láminas

- REPA secuencial con LLVM

```
$ ghc -O2 --make -fllvm -rtsopts fws
$ fws +RTS -s -RTS 2000
Total   time  102.76s  (102.74s elapsed)
Productivity  99.5% of total user
```

UNIVERSIDAD SIMÓN BOLÍVAR

Una comparación apurada...

Porque son las 08:50 y aún estoy haciendo láminas

- REPA secuencial con LLVM

```
$ ghc -O2 --make -fllvm -rtsopts fws
$ fws +RTS -s -RTS 2000
Total   time  102.76s  (102.74s elapsed)
Productivity  99.5% of total user
```

- REPA paralelo con LLVM sobre cuatro núcleos

```
$ ghc -O2 --make -threaded -fllvm -rtsopts fwp
$ fwp +RTS -N -s -RTS 2000
Total   time  196.83s  ( 53.14s elapsed)
Productivity  98.2% of total user
```

UNIVERSIDAD SIMÓN BOLÍVAR

Una comparación apurada...

Porque son las 08:50 y aún estoy haciendo láminas

- REPA secuencial con LLVM

```
$ ghc -O2 --make -fllvm -rtsopts fws
$ fws +RTS -s -RTS 2000
Total   time  102.76s  (102.74s elapsed)
Productivity  99.5% of total user
```

- REPA paralelo con LLVM sobre cuatro núcleos

```
$ ghc -O2 --make -threaded -fllvm -rtsopts fwp
$ fwp +RTS -N -s -RTS 2000
Total   time  196.83s  ( 53.14s elapsed)
Productivity  98.2% of total user
```

- Accelerate sobre CUDA – los núcleos están libres.

```
$ ghc -O2 --make -threaded -rtsopts fwa
$ fwa +RTS -s -RTS 2000
Total   time   43.59s  ( 43.56s elapsed)
Productivity  65.4% of total user
```

Quiero saber más...

- Documentación de `Data.Array.Repa`
- Documentación de `Data.Array.Accelerate`

UNIVERSIDAD SIMÓN BOLÍVAR