Parallel & Concurrent Haskell 7: GPGPU programming with Accelerate

Simon Marlow
What is GPGPU programming?

• General Purpose Graphics Processing Unit
• i.e. using your graphics card to do something other than play games or zoom windows
• GPUs have many more cores than your CPU:

Device 0: GeForce GTX 580 (compute capability 2.0)
16 multiprocessors @ 1.54 GHz (512 cores), 1 GB global memory
• Main difference:
  – All the cores run the same code at the same time
  – (but operate on separate data)
• SIMD (Single Instruction Multiple Data)
  – or just “Data Parallelism”
• We can’t program a GPU in the same way as a CPU – it has a different instruction set, and we can’t run Haskell programs on it directly
• The GPU has its own memory, so data has to be explicitly moved back and forth
Accelerate

Accelerate is a *Domain-specific language* for GPU programming

- Running Haskell/Accelerate program
- CUDA code
- Results of the GPU computation

Compiled by NVidia’s compiler, loaded onto the GPU, and executed.

This process may happen several times during the program’s execution
The CUDA code isn’t compiled every time – code fragments are cached and re-used
• So when you program using Accelerate, you are writing a Haskell program that generates a CUDA program

• But in many respects, it looks just like a Haskell program. (It shares various concepts with Repa too)

• For testing, there is also an interpreter that can run the Accelerate program without using the GPU
  – much more slowly of course
Some practical details

- Hopefully by the time you read this the Accelerate devs will have fixed the bugs that I found while writing this lecture 😊
- Now we’re ready to play with some of the basics.
- Accelerate is a large API, have the docs to hand: http://hackage.haskell.org/packages/archive/accelerate/0.12.0.0/doc/html/Data-Array-Accelerate.html

```bash
$ cabal install accelerate
$ cabal install accelerate-cuda

$ ghci
Prelude> import Data.Array.Accelerate as A
Prelude A> import Data.Array.Accelerate.Interpreter as I
Prelude A I>
```
Arrays and indices

- Accelerate computations take place over arrays

- The Array type has two type parameters:
  
  - **data**
  - **Array shape**

  The *shape* of the array (think: dimensions)

  The *element type* of the array
  
  A fixed set of element types are supported: Int8, Int32, Float, etc., and tuples.
- **Shapes:**

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

- $Z$ stands for zero dimensions (a scalar, with one element)
- $Z :: \text{Int}$ is the shape of a one-dimensional array (a vector) indexed by \text{Int}
- In fact, the only index type allowed is \text{Int}
- $Z :: \text{Int} :: \text{Int}$ is the shape of a two-dimensional array (a matrix) indexed by \text{Int}
- $(::)$ associates left, so $Z :: \text{Int} :: \text{Int}$ is $(Z :: \text{Int}) :: \text{Int}$
  - hence the tail/head naming in the type
- types and values look similar: $Z :: 3 :: Z :: \text{Int}$
Handy type synonyms:

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

type Scalar e = Array DIM0 e
type Vector e = Array DIM1 e
```
Playing with Accelerate arrays

- Accelerate provides some operations for experimenting with arrays, without using the Accelerate DSL itself.

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

```haskell
ghci> fromList (Z:.10) [1..10]
```
Playing with Accelerate arrays

- Accelerate provides some operations for experimenting with arrays, without using the Accelerate DSL itself.

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

```haskell
ghci> fromList (Z:.10) [1..10]
<interactive>:9:1:
  No instance for (Shape (Z ::. head0))
  arising from a use of `fromList'
Possible fix: add an instance declaration for (Shape (Z ::. head0))
In the expression: fromList (Z ::. 10) [1 .. 10]
In an equation for `it': it = fromList (Z ::. 10) [1 .. 10]
```

- Defaulting does not apply, because `Shape` is not a standard class
• Try with a type signature

```ghci
ghci> fromList (Z:.10) [1..10] :: Vector Int
Array (Z :: 10) [1,2,3,4,5,6,7,8,9,10]
```

• Ok, we made a vector from a list. Let’s try a matrix:

```ghci
ghci> fromList (Z:.3:.5) [1..] :: Array DIM2 Int
Array (Z :: 3 :: 5) [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

```
1  2  3  4  5 
6  7  8  9 10 
11 12 13 14 15
```

• fills along the rightmost dimension first.
• Of course, the array is really just a vector internally
  • the shape `(Z :: 3 :: 5)` tells Accelerate how to interpret indices.
Of course, the array is really just a vector internally.

- the shape \((Z :: 3 :: 5)\) tells Accelerate how to interpret indices.

```
ghci> fromList (Z:.3:.5) [1..] :: Array DIM2 Int
Array (Z :: 3 :: 5) [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

```
<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>
```

```
> let arr = fromList (Z:.3:.5) [1..] :: Array DIM2 Int
> indexArray arr (Z:.2:.1)
12
```

```
<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>
```

indices count from zero!
• You can even change the shape of an array without changing its representation – e.g. change a 3x5 array into a 5x3 array
  • but the operation is part of the full accelerate DSL, so we can’t demonstrate it yet
Arrays of tuples

Again this is really just a trick: Accelerate is turning the array of tuples into a tuple of arrays internally.

Note: there are no nested arrays. Array is not an allowable element type. Regular arrays only!
• Now to really run an Accelerate computation

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

• `run` comes from either
  • `Data.Array.Accelerate.Interpreter`
  • `Data.Array.Accelerate.CUDA`
• we’ll use the interpreter for now.
• `Arrays` constrains the result to be an array, or a tuple of arrays
• What is `Acc`?
  • This is the DSL type. `Acc` is really a data structure representing an array computation, that run will interpret (or compile and run on the GPU)
• First example: add 1 to every element

```haskell
> let arr = fromList (Z:.3:.5) [1..] :: Array DIM2 Int
> run $ A.map (+1) (use arr)
Array (Z :: 3 :: 5) [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
```

• We have to get our array into the Acc world:
  • this may involve copying it to the GPU

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

• Next, we use `A.map` to apply a function to every element
  • The A. disambiguates with `Prelude.map`

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

This is the function to apply to every element. But what’s Exp?
A.map ::
(Shape ix, Elt a, Elt b) =>
(Exp a -> Exp b) -> Acc (Array ix a) -> Acc (Array ix b)

- **Acc a**: an array computation delivering an a
  - a is typically an instance of class **Arrays**
- **Exp a**: a scalar computation delivering an a
  - a is typically an instance of class **Elt**

- In Accelerate the world is divided into **Acc** and **Exp**, so that we don’t accidentally use an array operation where an element operation is needed.
- Overloading is used so that numeric Haskell expressions can often be used where an **Exp** is required.
  - e.g. (:+1) :: **Exp Int** -> **Exp Int**
• We can see the data structure that Accelerate compiled our program to, by omitting the `run`:

```haskell
> A.map (+1) (use arr)
map
  (\x0 -> x0 + 1)
(use ((Array (Z :. 3 :. 5) [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]])
```

• One more example:

```haskell
> run $ A.map (^2) (use arr)
Array (Z :. 3 :. 5) [1,4,9,16,25,36,49,64,81,100,121,144,169,196,225]
```
Folds over arrays

- Folding (+) over the array gives the sum
- The result was an array of one element (a scalar). Why?
  - fold has an interesting type:

\[
\text{fold} :: (\text{Shape } \text{ix}, \text{Elt } a) \\
\quad \rightarrow (\text{Exp } a \rightarrow \text{Exp } a \rightarrow \text{Exp } a) \\
\quad \rightarrow \text{Exp } a \rightarrow \text{Acc } (\text{Array } (\text{ix } :: \text{Int}) a) \rightarrow \text{Acc } (\text{Array } \text{ix } a)
\]

- The fold happens over the outer dimension of the array
let arr = fromList (Z:.3:.5) [1..] :: Array DIM2 Int
run $ A.fold (+) 0 (use arr)
Array (Z :: 3) [15,40,65]
fold :: (Shape ix, Elt a)  
=> (Exp a -> Exp a -> Exp a)  
-> Exp a -> Acc (Array (ix ::. Int) a) -> Acc (Array ix a)

• Is it a left or a right fold?
• Neither!
  • the fold happens in parallel, tree-like
  • therefore the function should be associative, otherwise the results will be non-deterministic
  • (we pretend that floating-point operations are associative, even though strictly speaking they aren’t)
Indexing an array

(!) :: (Shape ix, Elt e) => Acc (Array ix e) -> Exp ix -> Exp e

• To try this out we need to make a one-dimensional array from an Exp:

unit :: Exp e -> Acc (Scalar e)

• Try it:

> let arr = fromList (Z:.3:.5) [1..] :: Array DIM2 Int
> run $ unit (use arr ! (Z :: 2 :: 1))

<interactive>:19:24:
  Couldn't match expected type `Exp DIM2'
    with actual type `tail0 :: head0'
  In the second argument of `(!)', namely `(Z :: 2 :: 1)'
  In the first argument of `unit', namely `(use arr ! (Z :: 2 :: 1))'
  In the second argument of `($)', namely
    `unit (use arr ! (Z :: 2 :: 1))'

• Ok, so we can’t just use (Z :: 2 :: 1) as an Exp ix
• Need a way to get from an \((Z :. \text{Int} :. \text{Int})\) to \(\text{Exp} (Z :. \text{Int} :. \text{Int})\)

```haskell
index0 :: \(\text{Exp} \ Z\)
index1 :: \(\text{Exp} \ \text{Int} \to \text{Exp} (Z :. \text{Int})\)
index2 :: \(\text{Exp} \ \text{Int} \to \text{Exp} \ \text{Int} \to \text{Exp} \ \text{DIM2}\)

> let arr = fromList (Z:.3:.5) [1..] :: Array \text{DIM2} \text{Int}
> run $ unit (use arr ! index2 2 1)
Array (Z) [12]
```
• Arrays can be reshaped:

\[
\text{reshape :: (Shape \text{ ix}, \text{ Shape } \text{ ix}', \text{ Elt } \text{ e})} \\
\rightarrow \text{Exp } \text{ ix} \rightarrow \text{Acc (Array } \text{ ix'} \text{ e)} \\
\rightarrow \text{Acc (Array } \text{ ix e)}
\]

\[
> \text{arr} \\
\text{Array (Z :: 3 :: 5) [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]} \\
> \text{run $ reshape (index2 5 3) (use arr)} \\
\text{Array (Z :: 5 :: 3) [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]}
\]

• It’s the same array data, just the shape is different

• Indexing will show the difference:

\[
> \text{run $ unit (use arr ! index2 2 1)} \\
\text{Array (Z) [12]} \\
> \text{run $ unit (reshape (index2 5 3) (use arr) ! index2 2 1)} \\
\text{Array (Z) [8]}
\]
More Array operations

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

> run $ A.zipWith (+) (use arr) (use arr)
Array (Z :: 3 :: 5) [2,4,6,8,10,12,14,16,18,20,22,24,26,28,30]
```
Array creation

• We don’t really want to create all our arrays in Haskell and then move them over with `use`
  – better to create them directly if possible

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

generate :: (Shape ix, Elt a)
    => Exp ix -> (Exp ix -> Exp a)
    -> Acc (Array ix a)
```
• Turn a Haskell value into an Exp:

```haskell
class Elt t => t -> Exp t
```
Boolean operations

- Standard boolean operations are available, but with different names because the standard names are not overloaded in Haskell:

\[
(\text{==*}) \quad :: \quad (\text{Elt } t, \text{IsScalar } t) \Rightarrow \text{Exp } t \rightarrow \text{Exp } t \rightarrow \text{Exp } \text{Bool} \\
\text{also } /\text{*}, <\text{*}, <=\text{*}, >\text{*}, >=\text{*}
\]

\[
(\text{&&*}) \quad :: \quad \text{Exp } \text{Bool} \rightarrow \text{Exp } \text{Bool} \rightarrow \text{Exp } \text{Bool} \\
(\text{||*}) \quad :: \quad \text{Exp } \text{Bool} \rightarrow \text{Exp } \text{Bool} \rightarrow \text{Exp } \text{Bool} \\
\text{not} \quad :: \quad \text{Exp } \text{Bool} \rightarrow \text{Exp } \text{Bool}
\]

- Conditionals (if):

\[
(\text{??}) \quad :: \quad \text{Elt } t \Rightarrow \text{Exp } \text{Bool} \rightarrow (\text{Exp } t, \text{Exp } t) \rightarrow \text{Exp } t
\]

\[
> \text{run } \$ \text{A.map (\text{\lambda } x \rightarrow x \mod 2 \text{==* } 1 \text{? (x } \ast \text{ } 2, x - 3)) } (\text{use arr}) \\
\text{Array (Z :: 3 :: 5) [2,-1,6,1,10,3,14,5,18,7,22,9,26,11,30]}
\]

- Use sparingly! Leads to SIMD divergence.
A Mandelbrot set generator
Basics

- Operation over the *complex plane*
- We pick a window onto the complex plane.
  - only points between -2.0 ... 2.0 on both axes are interesting
  - divide the window into pixels (e.g. 512x512), each pixel has a value c given by its coordinates on the plain
- A point is *in the set* if when iterating this equation, the value of $|Z|$ does not diverge:

  $Z_{n+1} = c + Z_n^2$

- where $|Z|$ is given by $\sqrt{x^2 + y^2}$
- definitely diverges if $|Z| > 2$
  - optimisation: drop the $\sqrt{}$, check for $> 4$
- Fixed number of iterations
- Pretty pictures: colour depends on no. of iterations before divergence
• So the calculation for each pixel is independent: good for SIMD
• Complications:
  • iteration
  • remember the iteration count when divergence occurs
  • there is likely to be *some* conditional somewhere, but we want to minimize this
Getting started

• **first some types:**

```haskell
type F = Float
type Complex = (F,F)
type ComplexPlane = Array DIM2 Complex
```

• **Now let’s define the function we will iterate, `next`:**

```haskell
next :: Exp Complex -> Exp Complex -> Exp Complex
next c z = c `plus` (z `times` z)
```

• **Now we need to define `plus` and `times`**

```haskell
plus :: Exp Complex -> Exp Complex -> Exp Complex
plus a b = ...
```
• **Exp Complex** is **Exp (Float,Float)**
  • How can we deconstruct the pair inside the **Exp**?
  • Accelerate provides these:

```
fst :: (Elt a, Elt b) => Exp (a, b) -> Exp a
snd :: (Elt a, Elt b) => Exp (a, b) -> Exp b
```

• So we can write:

```
plus :: Exp Complex -> Exp Complex -> Exp Complex
plus a b = ...
    where
        ax = A.fst a
        ay = A.snd a
        bx = A.fst b
        by = A.snd b
```

• But we also need to construct the result pair
  • \((ax+bx, ay+by)\) has type \((Exp F, Exp F)\)
  • we want **Exp** \((F,F)\)
  • Fortunately **lift** has this type (amongst many others)
• Fortunately \textbf{lift} has this type (amongst many others)

\begin{verbatim}
lift :: (Exp F, Exp F) -> Exp (F, F)
\end{verbatim}

• So we have:

\begin{verbatim}
plus :: Exp Complex -> Exp Complex -> Exp Complex
plus a b = lift (ax+bx, ay+by)
  where
    ax = A.fst a
    ay = A.snd a
    bx = A.fst b
    by = A.snd b
\end{verbatim}

• In general, \textbf{lift} is for taking ordinary Haskell values into Exp or Acc, and \textbf{unlift} is for the opposite
  • (but it’s “more complicated than that”)
  • in fact, \textbf{fst} and \textbf{snd} are defined in terms of \textbf{unlift}:

\begin{verbatim}
fst :: (Elt a, Elt b) => Exp (a, b) -> Exp a
fst e = let (x, _:: Exp b) = unfold e in x
\end{verbatim}
So we can write plus in a slightly nicer way:

```haskell
plus :: Exp Complex -> Exp Complex -> Exp Complex
plus a b = lift (ax+bx, ay+by)
  where
    (ax, ay) = unpack a :: (Exp F, Exp F)
    (bx, by) = unpack b :: (Exp F, Exp F)
```

Note we had to add some type signatures
- rules of thumb for fixing type errors:
  - add type signatures
  - comment out code until it passes the type checker

There’s one more way to simplify this:
- `lift2` is a function that lifts the result and unlifts the arguments for a 2-ary function:

```haskell
plus :: Exp Complex -> Exp Complex -> Exp Complex
plus = lift2 f
  where f :: (Exp F, Exp F) -> (Exp F, Exp F) -> (Exp F, Exp F)
    f (ax,ay) (bx,by) = (ax+bx,ay+by)
```

`times` is similar to `plus`. 
What about iteration/conditionals?

- Iteration is ok as long as we do the same thing to every element in every iteration.
- So we have to apply the function even to elements that have already diverged.
  - (wasted work is not really an issue, we have lots of cores)
- Key idea: keep a pair \((z_n, i)\) per element.
  - \(z_n\) is the current Z value.
  - \(i\) is the iteration that divergence occurred, or the current iteration otherwise.
- So for each element, our inputs are \((z, i)\) and \(c\).
  - Compute \(z' = \text{next } c \ z\).
  - If \(z'\) diverged, result is \((z, i)\).
  - Else result is \((z', i+1)\).
iter :: Exp Complex -> Exp (F,F,Int) -> Exp (F,F,Int)
iter c z =
  let
    (x,y,i) = unfold z :: (Exp F, Exp F, Exp Int)
    z' = next c (lift (x,y))
  in
  (dot z' >* 4.0) ?
    ( z
      , lift (A.fst z', A.snd z', i+1)
    )

• There are no nested tuples, so instead of (Complex, Int) we must use (F, F, Int)
• First unfold z, and then call next
• Next, check whether z' has diverged
  • dot is just $x^2 + y^2$ (not shown)
  • If it has diverged, then return the old z
  • otherwise, return z' and i+1
• Due to SIMD divergence, the GPU will execute each iteration in two passes: first the true branches, then the false branches
Final pieces

```
genPlane :: F -> F  -- X bounds of the view
    -> F -> F  -- Y bounds of the view
    -> Int     -- X resolution in pixels
    -> Int     -- Y resolution in pixels
    -> Acc ComplexPlane
```

- calls generate to make the initial `ComplexPlane`

```
mkinit :: Acc ComplexPlane -> Acc (Array DIM2 (F,F,Int))
```

- makes the initial array of \((z,i)\) values; the input to the first iteration
mandelbrot :: F -> F -> F -> F -> Int -> Int -> Int
  -> Acc (Array DIM2 (F,F,Int))

mandelbrot x y x' y' screenX screenY depth
  = iterate go zs0 !! depth
  where
    cs  = genPlane x y x' y' screenX screenY
    zs0 = mkinit cs

    go :: Acc (Array DIM2 (F,F,Int))
       -> Acc (Array DIM2 (F,F,Int))
    go = A.zipWith iter cs

iterate :: (a -> a) -> a -> [a] -- in the Prelude

• but... doesn’t that generate a program as large as the number of iterations?
  • Accelerate has some clever caching: it generates the code for one iteration and then re-uses it
  • You can see what it is generating with –ddump-cc
Finally

• The main function calls `run`, and then feeds the output into Gloss to generate a picture
• See the full code in `code/mandel/mandel.hs`
• Run it like this:

```sh
$ ghc -O mandel.hs
$ ./mandel --size=512 --limit=256 --cuda
```
Wrap up

• Hopefully 0.13 will be released by now (0.12 had a couple of bugs that affect us)
• Exercise: crack my password!
• If you struggle with type errors, ask the assistants
  – add type signatures, comment-out code
  – figuring out conversions between types is the most common problem