ACCELERATE AND THE OUTSIDE WORLD: INTERFACING ACCELERATE WITH LOW-LEVEL CUDA

Robert Clifton-Everest
UNSW
robertce@cse.unsw.edu.au

ACCELERATE IN A NUTSHELL

- Embedded language for high-performance array computations
- Well known for its CUDA backend
- Array computations are of type **`Acc`** and scalar computations are of type **`Exp`**

- This function signature tells you most of what you need to know

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

DOES IT WORK WITH OTHER GPGPU FRAMEWORKS?

- Up till now, no.
- Now, kinda.

TWO DISTINCT PROBLEMS

Problem 1

The direction of the arrow corresponds to function calls

TWO DISTINCT PROBLEMS

Problem 2

The direction of the arrow corresponds to function calls

EXAMPLE FOR PROBLEM I

- Smoothlife

- Conway's game of life generalised to a continuous domain

- Relies on a Fast Fourier Transform
- We could have written our own FFT (actually, Trevor did), or...
- We take advantage of the cuFFT library

CUFFT gives us C functions like this

```
cufftResult cufftExecC2C(cufftHandle *plan, cufftComplex *idata,
                         cufftComplex *odata, int direction);
```

- We can import this in to Haskell easily enough

```
foreign import cufftExecC2C ...
```

- But how can we call this function from within an accelerate computation and give it the device pointers it needs?

We add this operation

```
foreignAcc :: (Arrays arr, Arrays res, Foreign ff)
           => ff arr res                -- The foreign function
           -> (Acc arr -> Acc res) -- The pure equivalent
           -> Acc arr
           -> Acc res
```

We add this operation

```
foreignAcc :: (Arrays arr, Arrays res, Foreign ff)
           => ff arr res                -- The foreign function
           -> (Acc arr -> Acc res) -- The pure equivalent
           -> Acc arr
           -> Acc res
```

Backends provide implementations of this class

```
class Typeable2 f => Foreign f where
  ...
```

So for the CUDA backend

```
newtype CuForeignAcc args results
  = CuForeignAcc (args -> CIO results)
  deriving (Typeable)

instance Foreign CuForeignAcc where
  ...
```

`CIO` is an abstract monad giving us access to functions like these

```
-- Allocate a new array
allocateArray :: (Shape dim, Elt e) => dim -> CIO (Array dim e)

-- Get the device pointers associated with an array
devicePtrsOfArray :: Array sh e -> CIO (DevicePtrs (EltRepr e))

-- Push and pull data from the device
peekArray, pokeArray :: (Shape dim, Elt e) => Array dim e -> CIO ()
```

Putting it all together

```
doFFT :: Acc (Array DIM2 Complex)
      -> Acc (Array DIM2 Complex)
doFFT arr = foreignAcc (CuForeign foreignFFT)
                       pureFFT
                       arr
  where
    pureFFT = ... a slow but pure Accelerate FFT ...


foreignFFT :: Array DIM2 Complex -> CIO (Array DIM2 Complex)
foreignFFT arr = do
  hndl <- ... do some initialisation of cufft ...
  out <- allocateArray (shape arr)
  ((), DevicePtr idata) <- devicePtrsOfArray arr
  ((), DevicePtr odata) <- devicePtrsOfArray out
  liftIO $ cufftExecC2C hndl idata odata 1
  return out
```

PROBLEM 2

- What if we have an existing CUDA C/C++ application and we want to replace parts of it with Accelerate?

A SIMPLE EXAMPLE

- Vector dot product

- Accelerate code looks like this

```
dotp :: Acc (Vector Float)
     -> Acc (Vector Float)
     -> Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

- How can we call this from C?

- First we do this

```
{-# LANGUAGE TemplateHaskell #-}
module Dotp where

foreignAccModule

dotp :: Acc (Vector Float)
     -> Acc (Vector Float)
     -> Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)

exportAfun1 'dotp
```

- When compiled this will generate **`Dotp.h`**

- Now somewhere in our C program

- First we compile the accelerate program

```
#include “Dotp.h”

AccHandle hndl;
Program p_dotp;

void init() {
  CUcontext ctx;
  CUdevice dev;
  cuCtxGetCurrent(&ctx);
  cuCtxGetDevice(&dev)

  hndl = accelerateCreate(ctx, dev);
  p_dotp = dotp_compile(hndl);
}
```

- Then we can call it

```
float dotp(float *x, float* y, int n) {
  int sh[] = { n };
  ResultArray res;
  dotp_run(p_dotp, &a, sh, &b, sh, &res);

  float* out;
  float ret;
  getDevicePtrs(res, &out);
  cudaMemcpy(&ret, out, sizeof(float), cudaMemcpyDeviceToHost);
  return ret;
}
```

QUESTIONS?