GPGPU Programming in Haskell with Accelerate

Trevor L. McDonell

@tlmcdonell
tmcdonell@cse.unsw.edu.au

https://github.com/AccelerateHS

Context

- YOW LambdaJam is coming up: May 16-17, Brisbane
- buuut... I need to prepare a talk for it:
  - Split into 30 min + 1.5 hr session
  - Looking for feedback on topics to include / focus on / skip / etc.
  - Particularly if you have tried to use Accelerate before
  - *Especially* if you have tried and failed

What is GPGPU Programming?

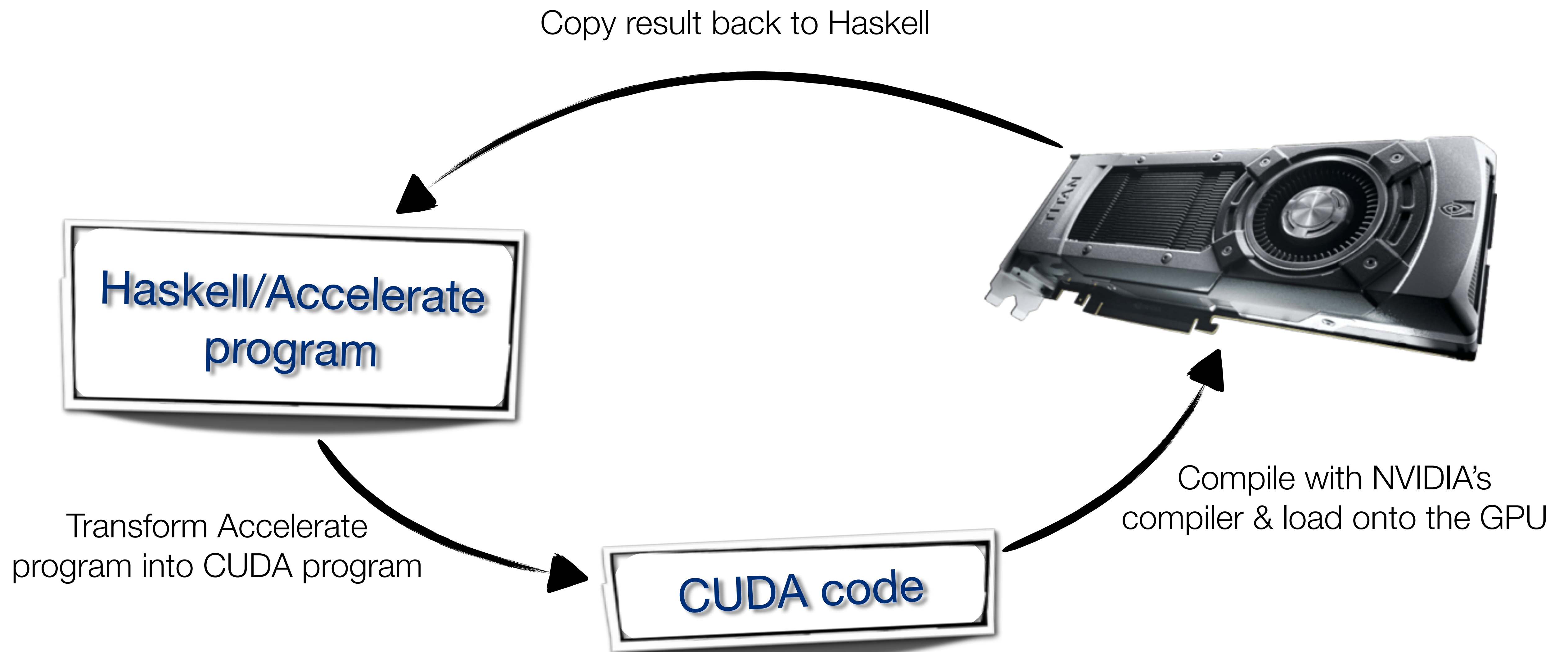
- General Purpose Programming on Graphics Processing Units (GPUs)
- Using your graphics card for something other than playing games
- GPUs have many more cores than a CPU
  - GeForce GTX Titan
  - 2688 cores @ 837 MHz
  - 6 GB memory @ 288 GB/s

What is GPGPU Programming?

- Main differences:
  - Single program multiple data (SPMD / SIMD), or just data-parallelism
  - All the cores run the same program, but on different data
- We can't program these in the same way as a CPU
  - Different instruction sets: can't run a Haskell program directly
  - More restrictive hardware designs, limited control structures
- GPUs have their own memory
  - Data has to be explicitly moved back and forth

Accelerate

- Accelerate is a Domain-Specific Language for GPU programming



Accelerate

- Accelerate is a Domain-Specific Language for GPU programming
  - This process may happen several times during program execution
  - Code and data fragments get cached and reused
- An Accelerate program is a Haskell program the generates a CUDA program
  - However, in many respects this still looks like a Haskell program
  - Shares various concepts with *Repa*, a Haskell array library for CPUs

Accelerate

- Accelerate computations take place on arrays
  - Parallelism is introduced in the form of collective operations over arrays

- Arrays have two type parameters
  - The shape of the array, or dimensionality
  - The element type of the array: Int, Float, etc.

```
data Array sh e
```

Shapes

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

- Shapes determine the dimensions of the array and the type of the index
  - Z represents a rank-zero array (singleton array with one element)
  - (:.) increases the rank by adding a new dimension on the *right*
- Examples:
  - One-dimensional array (vector) indexed by Int: (Z :. Int)
  - Two-dimensional array, indexed by Int: (Z :. Int :. Int)
- This style is used at both the *type* and *value* level:

```
sh :: Z :. Int
sh =  Z :. 10
```

Shapes

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

- We have some handy synonyms as well:

```
type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int
type DIM3 = DIM2 :. Int
  -- and so on...

type Scalar e = Array DIM0 e
type Vector e = Array DIM1 e
```

Arrays

```
data Array sh e
```

- Supported array element types are members of the `Elt` class:
  - ()
  - Int, Int32, Int64, Word, Word32, Word64...
  - Float, Double
  - Char
  - Bool
  - Tuples up to 9-tuples of these, including nested tuples
- Note that `Array` itself is not an allowable element type. There are no nested arrays in Accelerate, regular arrays only!

Arrays

```
data Array sh e
```

- Create an array from a list:

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

  - Generates a multidimensional array by consuming elements from the list and adding them to the array in row-major order

- Example:

```
ghci> fromList (Z:.10) [1..10]
```

  - Defaulting does not apply, because `Shape` is not a standard class

```
<interactive>:3:1:
    No instance for (Shape (Z :. head0))
      arising from a use of `fromList'
    The type variable `head0' is ambiguous
    Possible fix: add a type signature that fixes these type
    Note: there is a potential instance available:
      instance Shape sh => Shape (sh :. Int)
        -- Defined in `Data.Array.Accelerate.Array.Sugar'
    Possible fix: add an instance
                                                             .. 10
<
                              arising from the literal `10
    The type variable `head0' is ambiguous
    Possible fix: add a type signature that fixes these type
    Note: there are several potential instances:
      instance Num Double -- Defined in `GHC.Float'
      instance Num Float -- Defined in `GHC.Float'
      instance Integral a => Num (GHC.Real.Ratio a)
        -- Defined in `GHC.Real'
      ...plus 12 others
    In the second argument of `(:.)', namely `10'
```

Number 1 tip:
Add type signatures

Arrays

```
data Array sh e
```

- Create an array from a list:

```
> fromList (Z:.10) [1..10] :: Vector Float
Array (Z :. 10) [1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0]
```

- Multidimensional arrays are similar:
  - Elements are filled along the right-most dimension first

```
> 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

Arrays

```
data Array sh e
```

- Array indices start counting from zero

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

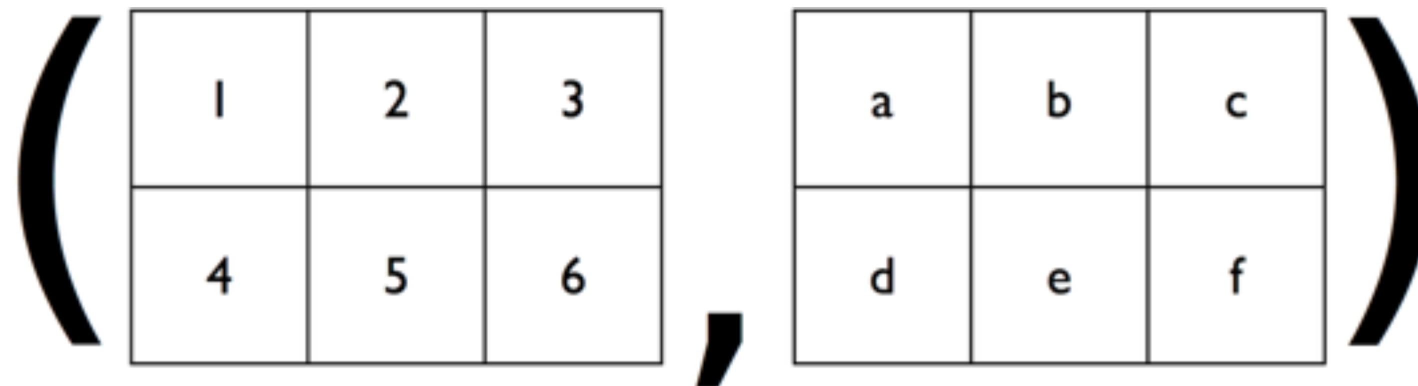
1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

Arrays

```
data Array sh e
```

- Similarly, an array of (possibly nested) tuples:
    - This is just a trick: internally converted into a tuple of arrays

```
> fromList (Z:.2:.3) $ P.zip [1..] ['a'..] :: Array DIM2 (Int,Char)
Array (Z :. 2 :. 3) [(1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e'),(6,'f')]
```



Data.Array.Accelerate

- Need to import both the base library as well as a backend
  - There is also an interpreter available for testing
  - Runs without using the GPU (much more slowly of course)

```
import Prelude                         as P
import Data.Array.Accelerate           as A
import Data.Array.Accelerate.CUDA as CUDA
```

Data.Array.Accelerate

- To actually run an Accelerate computation:

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

  - Run comes from whichever backend we have chosen (CUDA)
  - `Arrays` constrains the result to be an `Array`, or tuple thereof

- What is `Acc`?
  - This is our DSL type
  - A data structure representing a computation that once executed will yield a result of type 'a'

Accelerate is a library of collective operations over arrays of type `Acc a`

Data.Array.Accelerate

- To get arrays into the Acc world:

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

  - This may involve copying data to the GPU

- `use` injects arrays into our DSL

- `run` executes the computation to get arrays out

- Using Accelerate focuses on everything in between

Collective Operations

- Example: add one to each element of an array

```
> 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]
```

- What is the type of `map`?

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

Supported shape & element types

Function to apply at every element. But what is Exp?

DSL array

A Stratified Language

- Accelerate is split into two worlds: `Acc` and `Exp`
  - `Acc` represents collective operations over instances of `Arrays`
  - `Exp` is a scalar computation on things of type `Elt`
- Collective operations in `Acc` comprise many scalar operations in `Exp`, executed in parallel over `Arrays`
- Scalar operations can not contain collective operations
  - This excludes nested data parallelism

Scalar Expressions

- The type class overloading trick is used for standard Haskell classes

```
(+1) :: (Elt a, IsNum a) => Exp a -> Exp a
```

- Standard boolean operations are available with slightly different names
  - The standard names can not be overloaded

```
(==*) :: (Elt t, IsScalar t) => Exp t -> Exp t -> Exp Bool
(/=*), (<*), (>*), min, max, (||*), (&&*)   -- and so on...
```

- Conditionals
  - Use sparingly: leads to SIMD divergence

```
(?) :: Elt t => Exp Bool -> (Exp t, Exp t) -> Exp t
```

Scalar Expressions

- Bring a Haskell value into Exp land

```
constant :: Elt e -> e -> Exp e
```

- Lift an expression into a singleton array

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

Reductions

- Folding `(+)` over a vector produces a sum
  - The result is a one-element array (scalar). Why?

```
> let xs = fromList (Z:.10) [1..] :: Vector Int
> run $ A.fold (+) 0 (use xs)
Array (Z) [55]
```

- Fold has an interesting type:

```
fold :: (Shape sh, Elt a)
     => (Exp a -> Exp a -> Exp a)
     -> Exp a
     -> Acc (Array (sh:.Int) a)
     -> Acc (Array sh        a)
```

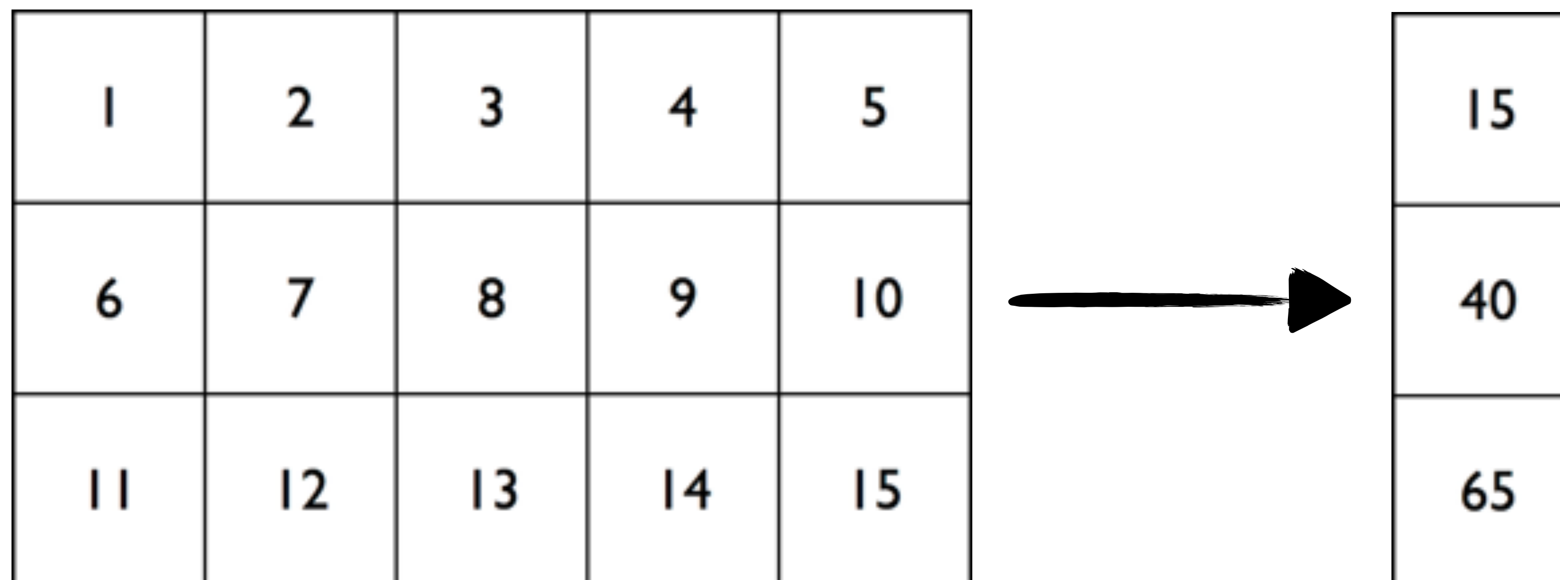
input array

outer dimension removed

Reductions

- Fold occurs over the outer dimension of the array

```
> let mat = fromList (Z:.3:.5) [1..] :: Array DIM2 Int
> run $ A.fold (+) 0 (use mat)
Array (Z :. 3) [15,40,65]
```

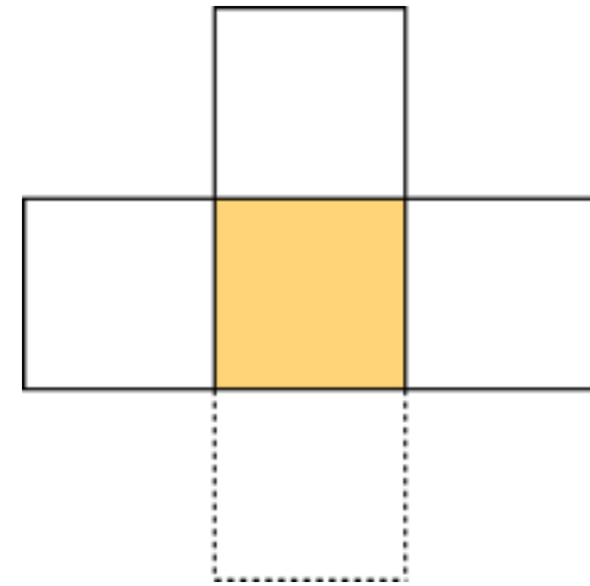


Reductions

- Is this a left-fold or a right-fold?
  - Neither! The fold happens in parallel, tree-like
  - Therefore the function must be associative: `(Exp a -> Exp a -> Exp a)`
  - (We pretend that floating point operations are associative, though strictly speaking they are not)

Stencils

- A 2D convolution...



Permutations

- Forward / backward permutations ...

- Replicate / slice ...

Problem Solving with Accelerate

Example: password "recovery"

MD5 Algorithm

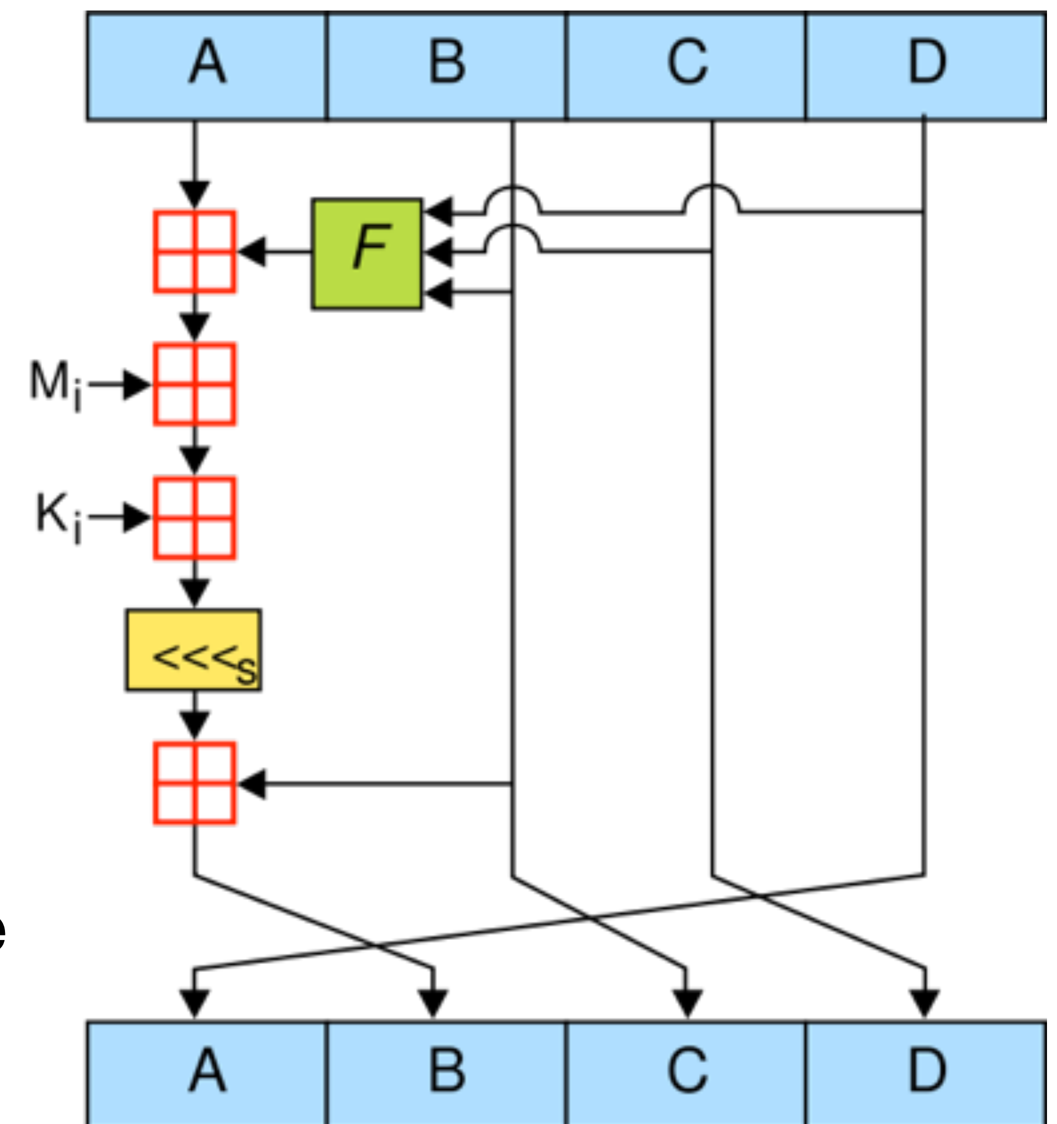
- Aim:
  - Implement one round of MD5: unsalted, single 512-bit block
  - Apply to an array of words
  - Compare hashes to some unknown hash
  - i.e. standard dictionary attack

- For fun: send me your (low quality) password hashes
  - tmcdonell@cse.unsw.edu.au

```
$ md5 -q -s password
5f4dcc3b5aa765d61d8327deb882cf99
```

MD5 Algorithm

- Algorithm operates on a 4-word state, A, B, C, and D
- There are 4 x 16 rounds: F, G, H, and I
  - M_i is a word from the input message
  - K_i is a constant
  - $<<<_s$ is left rotate, by some constant r_i
- Each round operates on the 512-bit message block, modifying the state



http://en.wikipedia.org/wiki/Md5

MD5 Algorithm in Accelerate

- Accelerate is a meta programming language
  - Use regular Haskell to generate the expression for each step of the round
  - Produces an unrolled loop

```
type ABCD = (Exp Word32, Exp Word32, ... )

md5round :: Acc (Vector Word32) -> ABCD
md5round msg
  = P.foldl round (a0,b0,c0,d0) [0..64]
  where
    round :: ABDC -> Int -> ABCD
    round (a,b,c,d) i = ...
```

MD5 Algorithm in Accelerate

- The constants `kᵢ` and `rᵢ` can be embedded directly
  - The simple list lookup would be death in standard Haskell
  - Generating the expression need not be performant, only executing it

```
k :: Int -> Exp Word32
k i = constant (ks P.!! i)
  where
    ks = [ 0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee
         , ...
```

MD5 Algorithm in Accelerate

- The message `M` is stored as an array, so we need array indexing
  - Be wary, arbitrary array indexing can kill performance...

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

- Get the right word of the message for the given round

```
m :: Int -> Exp Word32
m i
  | i < 16 = msg A.! index1 (constant i)
  | i < 32 = msg A.! index1 (constant ((5*i + 1) `rem` 16))
  | ...
```

MD5 Algorithm in Accelerate

- Finally, the non-linear functions F, G, H, and I

```
round :: ABDC -> Int -> ABCD
round (a,b,c,d) i
  | i < 16 = shfl (f b c d)
  | ...
  where
    shfl x  = (d, b + ((a + x + k i + m i) `rotateL` r i), b, c)

    f x y z = (x .&. y) .|. ((complement x) .&. z)
    ...
```

MD5 Algorithm in Accelerate

- MD5 applied to a single 16-word vector: no parallelism here
- Lift this operation to an array of `n` words: stored as a `(Z:.16:.n)` array
  - Store one word per ~~row~~ column, process many words in parallel
  - Need to use `generate`, the most general form of array construction. Equivalently, the most easily misused (as we ~~will~~ won't see)

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

MD5 Algorithm in Accelerate

- As always, data layout is important
  - Accelerate arrays are stored in row-major order

- CUDA threads work together
  - `generate` uses one thread per element
  - For best performance threads need to index adjacent elements of an array
  - This only works for m_i if all the first letters are adjacent in memory, etc.

STOP: demo time

Problem Solving Accelerate

Hints for when things don't work as you expect

Executing Computations

- run vs. run1

Acc Inside

- We can see the data structure Accelerate generates by omitting ‘run’
  - Useful to check if intermediates have been fused away

```
> let xs = fromList (Z:.10) [1..] :: Vector Int
> A.map (*2) $ A.map (\x -> x `mod` 2 ==* 0 ? (x-2, x+3)) (use xs)
let a0 = use (Array (Z :. 10) [1,2,3,4,5,6,7,8,9,10])
in
map (\x0 -> 2 * (0 ==* (mod (x0, 2)) ? (-2 + x0, 3 + x0))) a0
```

Embedded Scalars

- Check what & how often code is getting compiled: -ddump-cc

Nested Data-Parallelism

- matrix-vector multiply: using replicate & a higher-dimensional fold

Iterations

- Running out of memory: Floyd-Warshall
- Using the `(>->)` operator