The Implementation of

PARALLEL FUNCTIONAL ARRAY PROGRAMMING

Gabriele Keller
(currently) UNSW Sydney
(very soon) Utrecht University
Parallel Programming performance!

Functional Languages
abstract higher order functions
controlled side effects...

PARALLEL FUNCTIONAL PROGRAMMING
MAKING FP (AND TYPES) WORK FOR US

• Abstraction also means the compiler has more information
  - controlled side effects, no user-level pointers,…
• Collection oriented versus explicit loops/recursion
• Expressive type systems help to
  - guide the user
  - guide the compiler write
Composite data structures

Immutable structures

Expressive type system & inference

Strong static typing

Haskell

Strictly isolating side-effects

Polymorphism & generics

Boxed values

Higher-order functions & closures

Principled, pure, functional programming
How about domain specific languages with specialised code generation?
DOMAIN SPECIFIC LANGUAGES

➤ Are **restricted** languages
  - Generally have specialised features to a particular application domain
  - HTML, Matlab, SQL, postscript, LaTeX ...

➤ **Embedded** domain specific languages
  - Implemented as libraries in the host language, so can integrate with the host language
  - Reuse the syntax of the host language (as well as parser, type checker…)
  - The host language can generate embedded code
  - Functional languages are great as host languages
• An **embedded domain-specific language** for high-performance computing in Haskell
ACCELERATE

- Array computations
  - Ray tracing
  - Mandelbrot fractal
  - n-body gravitational simulation

- Everything else
There are two ways to embed a language

- shallow embedding
- deep embedding
SHALLOW EMBEDDING

➤ Shallow embedding provides fixed interpretation
➤ Semantics captured in the type
➤ **Example**: arithmetic expression language

```haskell
type Expr = Float
```
**DEEP EMBEDDING**

- Captures DSL expression as abstract syntax tree (AST), allowing multiple interpretations

```haskell
data Expr
  = Add  Expr  Expr
  |  Mult Expr  Expr
  |  Neg  Expr
  |  Const Float

data Expr where
  Add    :: Expr -> Expr -> Expr
  Mult   :: Expr -> Expr -> Expr
  Neg    :: Expr -> Expr -> Expr
  Const  :: Float -> Expr
```

```haskell
sampleExpr
  = Mult (Add (Const 1) (Const 3)) (Const 5)

add  :: Expr -> Expr -> Expr
add  = Add
```

Often, constructors are not exposed, but wrapped in regular functions.
DEEP EMBEDDING

➤ Captures DSL expression as AST, allowing multiple interpretations

```haskell
data Expr where
  Add :: Expr -> Expr -> Expr
  Mult :: Expr -> Expr -> Expr
  Neg  :: Expr -> Expr -> Expr
  Const :: Float -> Expr

eval :: Expr -> Float
eval (Const x) =  
 eval (Add e1 e2) = 
 eval (Mult e1 e2) = 
 eval (Neg e) = 
```
DEEP EMBEDDING

data Expr where
    Add :: Expr -> Expr -> Expr
    Mult :: Expr -> Expr -> Expr
    Neg  :: Expr -> Expr -> Expr
    Const :: Float -> Expr

simplify :: Expr -> Expr

execute :: Expr -> IO Float
The expression representation is untyped:

data Expr where
  Add :: Expr → Expr → Expr
  Mult :: Expr → Expr → Expr
  Neg  :: Expr → Expr

eval :: Expr → Float

could be Float or Bool!
The expression representation is untyped:

```haskell
data Expr where
  Add  :: Expr -> Expr -> Expr
  Mult :: Expr -> Expr -> Expr
  Neg  :: Expr -> Expr
  If   :: Expr -> Expr -> Expr -> Expr
  Less :: Expr -> Expr -> Expr
  NConst :: Float -> Expr
  BConst :: Bool -> Expr

data Result where
  FRes :: Float -> Result
  BRes :: Bool -> Result
```

```haskell
eval :: Expr -> Result
```
ASIDE: PARAMETRISED ALGEBRAIC DATA TYPES

```
data Tree a
  = Leaf
  | Node a (Tree a) (Tree a)
```

```
data Tree a where
  Leaf :: a
  Node :: a -> Tree a -> Tree a -> Tree a
```

ASIDE: PARAMETRISED ALGEBRAIC DATA TYPES

```
data Tree a = Leaf
             | Node a (Tree a) (Tree a)
```

```
data Tree a where
  Leaf :: Tree b
  Node :: c -> Tree c -> Tree c -> Tree c
```
DEEP EMBEDDING

- **Generalised Algebraic Data Types (GADTs)**

```haskell
data Expr where
  Add :: Expr -> Expr -> Expr
```

```haskell
eval :: Expr a -> a
eval (Const c) = c
eval (If cond e1 e2) =
  if (eval cond)
    then eval e1
    else eval e2
```
DEEP EMBEDDING

➤ Generalised Algebraic Data Types (GADTs)

```haskell
data Expr a where
  Add :: Expr Float -> Expr Float -> Expr Float
  Mult :: Expr Float -> Expr Float -> Expr Float
  Neg :: Expr Float -> Expr Float
  Less :: Expr Float -> Expr Float -> Expr Bool
  Const :: a -> Expr a
  If :: Expr Bool -> Expr a -> Expr a -> Expr a

simplify :: Expr a -> Expr a
simplify (Const n) = Const n
simplify (Neg (Neg e)) = simplify e
simplify (Add e1 e2) = Add (simplify e1) (simplify e2)
simplify (Mult e1 e2) = Mult (simplify e1) (simplify e2)
```
LET’S LOOK AT ACCELERATE NOW!
ACCELERATE

- Computations take place on dense, multidimensional arrays
- Parallelism is introduced in the form of collective operations on arrays

- The usual suspects: maps, zipWiths, folds, generators, permutes and backpermutes, stencil operations
FIRST EXAMPLE

- dot-product in Haskell (on lists):

```haskell
import Prelude

dotp :: Num a => [a] -> [a] -> a

dotp xs ys = foldl (+) 0 (zipWith (*) xs ys)
```

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

foldl :: (b -> a -> b) -> b -> [a] -> b
FIRST EXAMPLE

- dot-product in Haskell (on vectors):

```haskell
import Data.Vector.Unboxed

dotp :: (Num a, Unbox a)
  => Vector a
  -> Vector a
  -> a

dotp xs ys = foldl (+) 0 (zipWith (*) xs ys)
```

zipWith :: (a -> b -> c) -> Vector a -> Vector b -> Vector c

foldl :: (b -> a -> b) -> b -> Vector a -> b
FIRST EXAMPLE

➤ dot-product in Haskell (using Accelerate):

```haskell
import Data.Array.Accelerate

dotp :: (Num a, Elt a)
    => Acc (Vector a)
    -> Acc (Vector a)
    -> Acc (Scalar a)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

`zipWith : (Elt a, Elt b, Elt c)
    => (Exp a -> Exp b -> Exp c) -> Acc (Vector a) -> Acc (Vector b)
    -> Acc (Vector c)`

`fold :: Elt a =>
    (Exp a -> Exp a -> Exp a) -> Exp a -> Acc (Vector a) -> Acc (Scalar a)`
DETOUR: TYPE CLASSES IN HASKELL

instance Num Int where
 (+) = ...
 (*) = ...
 ...

foo :: Num a => a -> a
foo x = x * x + x

foo' :: NumDict a -> a -> a
foo' dict x = (getAdd dict)((getMult dict) x x) x

type annotations may become necessary
DIFFERENT RUN FUNCTIONS

Running an accelerate program:

```haskell
dotp :: (Num a, Elt a) =>
    Acc (Vector a) -> Acc (Vector a) -> Acc (Scalar a)

vec1, vec2 :: Acc (Vector Float)
vec1 = ...;
vec2 = ...

accPrg :: Acc (Scalar Float)
accPrg = dotp vec1 vec2
```

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

putStrLn $ show $ run (dotp vec1 vec2)
```
RUNNING AN ACCELERATE PROGRAM

➤ Plugging it all together:

```haskell
dotp :: (Num a, Elt a) =>
    Acc (Vector a) -> Acc (Vector a) -> Acc (Scalar a)
dotp vec1 vec2 = …

vec1, vec2 :: Vector Float
vec1 = …
vec2 = …

main = P.putStrLn $ P.show $
    run1 (uncurry dotp) (vec1, vec2)
```

run1 :: Arrays a => (Acc a -> Acc b) -> a -> b
DIFFERENT RUN FUNCTIONS

➤ Compiling Accelerate programs at Haskell compile time:

runQ
Accelerate expressions can be of two distinct types:

- Embedded *sequential, scalar* expression:
  
  \[ \text{Exp} \ a \]

- Embedded *array* computations:
  
  \[ \text{Acc} \ a \]

What is the difference between these two?

\[ \text{Exp} \ \text{Int} \]

\[ \text{Acc} \ (\text{Scalar} \ \text{Int}) \]
Nested parallel computations can’t be expressed:

\[
\text{map} :: (\text{Elt} \ a, \text{Elt} \ b) \Rightarrow
(\text{Exp} \ a \rightarrow \text{Exp} \ b) \rightarrow \text{Acc} \ (\text{Vector} \ a) \rightarrow \text{Acc} \ (\text{Vector} \ b)
\]
DEFINITION OF EXP

- Exp is a GADT whose constructors represent scalar operations

```haskell
data Exp a where
  Const :: Elt c
          => c
          -> Exp c

  PrimApp :: (Elt a, Elt r)
            => PrimFun (a -> r)
            -> Exp a
            -> Exp r
```

Apply primitive scalar function: (+), (*) ...
AD-HOC POLYMORPHISM FOR EXP

➤ Overloaded the standard type classes to reflect arithmetic expressions

➤ The Num instance for Exp terms allows us to reuse standard operators like (+) and (*)

instance Num (Exp Int) where
   x + y = PrimAdd numType `PrimApp` tup2 (x, y)
   ...

AD-HOC POLYMORPHISM FOR EXP

- Use explicit dictionary passing to support ad-hoc polymorphism
  
  - Type checker chooses the correct instance when creating the dictionary
  
  - Pattern matching on the dictionary constructor makes the class constraints available
How does the dictionary trick work?

With a standard algebraic data type the following are equivalent:

\[
\begin{align*}
\text{foo} :: & \text{ Foo } a \rightarrow a \rightarrow a \\
\text{foo } _ & \ x = x+1 \\
\text{bar} :: & \text{ Foo } a \rightarrow a \rightarrow a \\
\text{bar (Foo } _) & \ x = x+1
\end{align*}
\]

But, with GADTs this is not the case

\[
\begin{aligned}
\text{data Foo a where} \\
\text{Foo :: Num a } & \Rightarrow a \rightarrow \text{ Foo } a
\end{aligned}
\]
ACCELERATE TYPES

➤ We encountered two different Accelerate array types:

\[
\text{dotp} :: (\text{Num \ a}, \text{Elt \ a}) \Rightarrow \\
\quad \text{Acc (Vector \ a)} \rightarrow \text{Acc (Vector \ a)} \rightarrow \text{Acc (Scalar \ a)}
\]

➤ These are just two special cases of Accelerate’s Array types

➤ parametrised with the shape type \text{sh}

➤ element type \text{a}

\[
\text{Array \ sh \ a}
\]
ARRAY SHAPES

The shape of an array determines its dimensionality and extent

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

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

type Scalar a = Array DIM0 a
type Vector a = Array DIM1 a
ARRAY SHAPES

Operations are shape polymorphic:

\[
\text{map} :: (\text{Shape } \text{sh}, \text{Elt } a, \text{Elt } b) \Rightarrow \\
(\text{Exp } a \to \text{Exp } b) \to \text{Acc } (\text{Array } \text{sh } a) \to \text{Acc } (\text{Array } \text{sh } b)
\]

\[
\text{zipWith} :: (\text{Shape } \text{sh}, \text{Elt } a, \text{Elt } b, \text{Elt } c) \Rightarrow \\
(\text{Exp } a \to \text{Exp } b \to \text{Exp } c) \to \\
\text{Acc } (\text{Array } \text{sh } a) \to \text{Acc } (\text{Array } \text{sh } b) \to \text{Acc } (\text{Array } \text{sh } c)
\]

\[
\text{fold} :: (\text{Shape } \text{sh}, \text{Elt } a) \Rightarrow \\
(\text{Exp } a \to \text{Exp } a \to \text{Exp } a) \to \text{Exp } a \to \\
\text{Acc } (\text{Array } (\text{sh :: Int}) a) \to \text{Acc } (\text{Array } \text{sh } a)
\]

\[
\text{generate} :: (\text{Shape } \text{sh}, \text{Elt } a) \Rightarrow \\
\text{Exp } \text{sh} \to (\text{Exp } \text{sh} \to \text{Exp } a) \to \text{Acc } (\text{Array } \text{sh } a)
\]
This means that our dot-product has actually a more general type:

\[
\text{dotp} :: (\text{Num } a, \text{ Elt } a) \\
\Rightarrow \text{Acc } (\text{Vector } a) \\
\rightarrow \text{Acc } (\text{Vector } a) \\
\rightarrow \text{Acc } (\text{Scalar } a) \\
\text{dotp } xs \text{ ys } = \text{fold } (+) \text{ 0 } (\text{zipWith } (*) \text{ xs } \text{ ys})
\]

\[
\text{dotp} :: (\text{Num } a, \text{ Elt } a, \text{ Shape } \text{sh}) \Rightarrow \\
\text{Acc } (\text{Array } (\text{sh :: } \text{Int}) a) \rightarrow \\
\text{Acc } (\text{Array } (\text{sh :: } \text{Int}) a) \rightarrow \\
\text{Acc } (\text{Array } \text{sh } a)
\]
SUMMARY SO FAR

- Looked at deep and shallow embedding
  - GADTs to maintain types in the AST
- Programming model of Accelerate
  - writing simple Accelerate programs
  - fromList :: (Elt e, Shape t) => t -> [e] -> Array t e
  - use :: Arrays arrays => arrays -> Acc arrays
Members of the Elt class contain admissible surface types for array elements:

- `()`
- `Int`, `Int32`, `Int64`, `Word`, `Word32`, `Word64`
- `Float`, `Double`
- `Char`
- `Bool`
- Array indices formed from `Z` and `(:.)`
- Tuples of all of these, e.g. `(Bool, Int, (Float, Float))`

To meet hardware restrictions, there are no nested arrays in Accelerate.
- GPUs are efficient processing arrays of elementary type
- not so much for aggregate types, pointers
- similarly CPU when using SIMD vector instructions
- set of types LLVM supports is fixed
- We map the user-friendly surface types to efficient representations
ELT CLASS IS USER EXTENSIBLE

- Using type families (i.e., functions from type to type)

```haskell
type family EltRepr t
type instance EltRepr Int = Int
type instance EltRepr Float = Float
type instance EltRepr (a,b) = ProdRepr (EltRepr a, EltRepr b)

type family ProdRepr t
type instance ProdRepr (a,b) = (((), a), b)
type instance ProdRepr (a,b,c) = (((), a), b), c)
```

- To extend the class, define

```haskell
fromElt :: a -> EltRepr a
toElt :: EltRepr a -> a
```
LIFTING

➤ How can we construct values of Exp type?

- Accelerate supplies Exp versions of Haskell Prelude ops, some constant values via overloading

➤ lift/unlift to switch to and (sometimes) back

```
trueExp :: Exp Bool
trueExp = lift True

true :: Bool
true =.unlift trueExp

swap :: Exp (Int, Int) -> Exp (Int, Int)
swap pairExp =
  let (x, y) = unlift pairExp :: (Exp Int, Exp Int)
  in lift (y, x)
```
LIFTING

c is a type constructor (e.g., Exp, Acc)

Plain is an associated type (function from type e to some other type)
Strips away the surface type constructors

```haskell
class Lift c e where
  type Plain e
  lift :: e -> c (Plain e)
```

```haskell
class Lift c e => Unlift c e where
  unlift :: c (Plain e) -> e
```
instance Lift Exp Int where
  type Plain Int = Int
  lift = Exp . Const

instance Lift Exp (Exp e) where
  type Plain (Exp e) = e
  lift = id

Plain (Exp Int, Int) ~ (Int, Int) ~ Plain (Int, Exp Int)
BACK TO THE IMPLEMENTATION OF EDSLs
EXECUTION OF AN ACCELERATE PROGRAM

➤ What happens when we compile & run a regular Haskell program?
EXECUTION OF AN ACCELERATE PROGRAM

What happens when we compile & run an Accelerate program?

Compile time

GHC

Run time

Acc

reify/compile/execute
WHY A TYPED AST?

➤ We compile the AST during application runtime
  - embedded compile time errors become application runtime errors

➤ Source of the type error can be
  - Accelerate user error
  - bugs in the Accelerate compiler

➤ Type checking the intermediate representation during Accelerate compilation
  - only shows this particular program is correct
  - transformation
WHY A TYPED AST?

➤ Applies to all runtime compiled EDSLs

➤ but particularly important for high-performance DSL

  - compilation is more challenging

➤ We learned the hard way

  - original CUDA backend was untyped

  - many bugs we found could have been avoided with a typed backend
Let us look at a slightly more interesting EDSL:

- values of the source language can be lifted into the DSL like the `Const` constructor in the arithmetic DSL
- function application
- lambda-abstraction

How do we model variables?
We can use the variables and abstraction mechanism of the host language:

```haskell
data HOExpr a where
\HConst :: a \rightarrow HOExpr a
\HApp :: HOExpr (a \rightarrow b) \rightarrow HOExpr a \rightarrow HOExpr b
\HLambda :: (HOExpr a \rightarrow HOExpr b) \rightarrow HOExpr (a \rightarrow b)

let f = \e a \rightarrow (\HApp (\HApp (\HConst (+)) e) e) e

eval (\HApp (\HLambda f) (\HConst 5))
```

```haskell
\evalHO :: HOExpr a \rightarrow a
\evalHO (\HConst c) = c
\evalHO (\HApp e1 e2) = (\evalHO e1) (\evalHO e2)
\evalHO (\HLambda f) = \a \rightarrow (\evalHO (f (\HConst a))
```
HIGHER-ORDER AST

➤ Convenient to write and evaluate:
  - abstraction
  - application of the host language

➤ Not suitable for transformation & analysis of the AST
  - can’t see inside functions

➤ Summary: good for surface syntax, not great for internal representation
FIRST ORDER AST

Variables as regular terms of the language

type VarId = String

data FOExpr a where
  FConst :: a -> FOExpr a
  FApp :: FOExpr (a -> b) -> FOExpr a -> FOExpr b
  FVar :: VarId -> FOExpr a
  FLambda :: VarId -> FOExpr b -> FOExpr (a -> b)

evalFO :: FOExpr a -> a
evalFO (FVar varId) = ???

we need an environment of some sort
but what is its type??
Alternative representation of lambda-terms eliminating names:

\[ \lambda x. \lambda y. (x + y) \]

\[ \lambda x. (x + (\lambda f. f x)(+1)) \]

Names are replaced with indices encoding the nesting depth of the binder

\[ \lambda \lambda (i_1 + i_0) \]

\[ \lambda (i_0 + (\lambda i_0 i_1)(+1)) \]
First Order Syntax with de Bruijn

**Idea:**

- A environment is either empty, or a tuple of value and rest environment.
- The type of the environment describes the type of all the values it contains.
- The $i$th entry is the value of variable bound at nesting level $n$.

```haskell
data Val env where
  Empty :: Val ()
  Push :: Val env -> t -> Val (env, t)

Push (Push Empty 5) True :: Val (((()), Int), Bool)
```
FIRST ORDER SYNTAX WITH DE BRUIJN

➤ Idea:

➤ a variable is a typed index

➤ the type encodes the type of the value the variable it represents, as well as the type of the environment is needs

```haskell
data Idx env t where
  ZeroIdx :: Idx (env, t) t
  SuccIdx :: Idx env t -> Idx (env, s) t
```

```haskell
prj :: Idx env t -> Val env -> t
prj ZeroIdx (Push val v) = v
prj (SuccIdx idx) (Push val _) = prj idx val
```
A term type in our language is parametrised with two types:

- the result type $t$
- the environment type $env$

```haskell
data Term env t where
  Var :: Idx env t                -> Term env t
  Con :: t                        -> Term env t
  Lam :: Term (env, s) t          -> Term env (s -> t)
  App :: Term env (s -> t) -> Term env s -> Term env t
```
The type-safe evaluator is now pretty straightforward:

```haskell
eval :: Term env t -> Val env -> t
eval (Var ix)       val = prj ix val
eval (Con v)        val = v
eval (Lam body)     val = eval body . (val `Push``)
eval (App fun arg)  val = (eval fun val) (eval arg val)
```
Typed higher-order abstract syntax is
- convenient as surface syntax
- not suitable for program analysis, program transformations

Typed De Bruijn first order abstract syntax
- impractical to use as surface syntax
- well suited as internal representation

Solution:
- user writes program in HO-syntax
- we convert it to De Bruijn representation
PROBLEM SOLVED NOW, RIGHT?

UHM, NO...
SUMMARY

➤ Surface types and representation types
➤ First-order and Higher-order abstract syntax
➤ Typed De Bruijn representation
What does the AST look like for this expression?
let
  inc = (+) 1 :: Exp Int -> Exp Int
  three = inc 2 :: Exp Int
  nine = (*) three three :: Exp Int
in (-) (inc nine) nine
PROBLEM

➤ **Sharing** in the host language internal representation

➤ Not readily observable

➤ Processing the tree means we are loosing the sharing information
  - often results in **large ASTs**
  - expressions are evaluated **multiple times**
  - really, really inefficient
PROBLEM

➤ Black-Scholes option pricing
  - Accelerate without sharing 20 times slower than CUDA implementation on GPU

```haskell
blackscholes :: Vector (Float, Float, Float)
  -> Acc (Vector (Float, Float))
blackscholes = map callput . use
  where
callput x =
    let (price, strike, years) = unlift x
    r = constant riskfree
    v = constant volatility
    v_sqrtT = v * sqrt years
    d1 = (log (price / strike) +
          (r + 0.5 * v * v) * years) / v_sqrtT
    d2 = d1 - v_sqrtT
    cnd d = let c = cnd' d in d >* 0 ? (1.0 - c, c)
    cndD1 = cnd d1
    cndD2 = cnd d2
    x_expRT = strike * exp (-r * years)
    in
      lift (price * cndD1 - x_expRT * cndD2
           , x_expRT * (1.0 - cndD2) - price * (1.0 - cndD1))
        rsqrt2pi * exp (-0.5*d*d) * poly k
```
PROBLEM

➤ Including ‘let’ in the surface language would make it extremely awkward to use

➤ Can we have ‘let’ in the internal representation, and convert without losing sharing?
We need to be able to observe an implementation detail pure functional languages abstract over referential equality:

- not enough to know two values are the same, we need to check if they share a location
- language level reference equality clashes with referential transparency, garbage collection, compiler optimisations

Luckily, the need for referential equality pops up in other contexts as well

- memoization, $O(1)$ comparison of large objects, ...
STABLE NAMES

➤ Idea:

- associate values with an address-like stable name

```haskell
data StableName a

mkStableName :: a -> IO (StableName a)
hashStableName :: StableName a -> Int

instance Eq (StableName a)
instance Ord (StableName a)
```
**STABLE NAMES**

\[
\text{mkStableName } x = \text{mkStableName } y \\
\implies \\
x = y
\]

\[
x \neq y \\
\implies \\
\text{mkStableName } x \neq \text{mkStableName } y
\]

\[
x = y \\
\not\implies \\
\text{mkStableName } x = \text{mkStableName } y
\]
SHARING RECOVERY

- Traverse the ADT structure and identify the shared nodes with the help of stable names

- insert let-bindings in the de Bruijn internal representation at the right positions
SHARING RECOVERY
let x =

```
let x =
  8
  @
  @
  (-)
  @
  x
  (*)
  @
  x
  2
  11
  @
  11
  (+)
```

SHARING RECOVERY
let x =

\[
\begin{align*}
\text{(-)} & \quad \text{(*)} \\
 x & \quad 8 \\
 & \quad 11 \\
 & \quad 11 \\
 & \quad 2
\end{align*}
\]
let x =

let y =

(-)

(*)

(+)

1

2

11

11

x

y

x

y
let x = @
let y = @ (+) 1
let z = @ x 2

@ (-) @ x y
@ (*) @ z
The two transformation are fused into a single one
- applying De Bruijn conversion first would destroy sharing

We lose type information during conversion from HOAS to De Bruijn
- type checked dynamically
PERFORMANCE

➤ Impact of sharing recovery*:

➤ Black Scholes, 20M elements:
  - CUDA, handwritten: 6.7ms
  - Accelerate, w/o sharing: 116ms
  - Accelerate w. sharing: 6.12ms

➤ Canny edge detection, 16M pixel:
  - OpenCV: 50.6ms
  - Accelerate, w/o sharing: 82.7ms
  - Accelerate w. sharing: 78.4ms

➤ Fluid-flow simulation, 2M particles
  - Accelerate, w/o sharing: 107ms
  - Accelerate w. sharing: 119ms

*Tesla T10 processor (compute capability 1.3, 30 multiprocessors = 240 cores at 1.3GHz, 4GB RAM) backed by two quad-core Xenon E5405 CPUs (64-bit, 2GHz, 8GB RAM), running GNU/Linux (Ubuntu 12.04 LTS). The reported GPU runtimes are averages of 100 runs.
FUSION

➤ Well-known problem of collection-oriented programming:

```haskell
map f $ map g xs

fold (+) $ enumFromThenTo 0 1 1000000000

let
    xs = map sqrt $ enumFromThenTo 0 1 1000000000
    in fold (+) $ zipWith (*) $ map (+ (fold (+) xs)) xs
```

➤ Unnecessary intermediate structures, traversals
Like sharing recovery, **fusion is essential** if we care about performance

- Mandelbrot: speed up of 1000%
- typically, at least 50% faster
Many of the classical techniques don’t work in this context:

- e.g., `build/fold` like fusion approaches destroy the parallel pattern

```haskell
build ::
  (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)

foldr k z (build g) = g k z

map f xs = build (\c n -> foldr
  (\a b -> c (f a) b) n xs)
...
```
FUSION

➤ Fusion often relies on inlining for producer/consumer pairs to be detected, only done conservatively

➤ Accelerate fusion happens at run time, need to be aware of the costs (but also: more information available)

➤ Result of fusion transformation needs to fit in to our code generation templates
FUSION

➤ Producers:
- each element of the result depends on at most one element of input array (e.g., map, backpermute, generate)

➤ Consumers:
- each element of result depends on multiple elements of input array (e.g., folds, scans, stencil operations)

➤ We treat them separately
- Producer/Producer fused via program transformation
- Producer/Consumer during code generation
Arrays represented as delayed computations:

data DelayedAcc a where
  Done :: Acc a
          -> DelayedAcc a
  Yield :: (Shape sh, Elt e)
          => Exp sh
          -> Fun (sh -> e)
          -> DelayedAcc (Array sh e)
  Step :: (Shape sh, Shape sh', Elt e, Elt e')
         => Exp sh'
         -> Fun (sh' -> sh)
         -> Fun (e -> e')
         -> Idx (Array sh e)
         -> DelayedAcc (Array sh' e')
Example: map

```haskell
mapD :: (Shape sh, Elt a, Elt b)
  => Fun (a -> b)
  -> DelayedAcc (Array sh a)
  -> DelayedAcc (Array sh b)
mapDf (Step shpgv)
  = Step shp (f . g) v
mapD f (Yield sh g)
  = Yield sh (f . g)
```

To prevent fusion, arrays can be made manifest

```haskell
compute :: Arrays a => Acc a -> Acc a
```
PRODUCER/CONSUMER FUSION

- Producer/consumer fusion is done during code generation
- Producer operations are inserted in the consumer code templates
- No support for consumer/consumer fusion yet
TYPE SAFE CODE GENERATION

- LLVM IR represents types as value-level data structure
- We track them as Haskell types in the LLVM binding
- Guarantees we only generate type correct LLVM programs
- GADT to define LLVM instruction set:

```haskell
data Instruction a where
  Add :: NumType a
  -> Operand a
  -> Operand a
  -> Instruction a

... |
```

- We translate the well-typed Accelerate AST into a well-typed LLVM AST
LLVM BACKEND FRAMEWORK

- LLVM is a reusable framework, portable across diverse architectures
- Accelerate LLVM backend framework
  - a set of re-usable components
  - reduces the cost of implementing future backends
- Existing backends:
  - vectorising multicore CPU
  - GPU backend
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input Size</th>
<th>Contender (ms)</th>
<th>Accelerate full (ms)</th>
<th>Accelerate no fusion (ms)</th>
<th>Accelerate no sharing (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black Scholes</td>
<td>20M</td>
<td>6.70 (CUDA)</td>
<td>6.19 (92%)</td>
<td>(not needed)</td>
<td>116 (1731%)</td>
</tr>
<tr>
<td>Canny</td>
<td>16M</td>
<td>50.6 (OpenCV)</td>
<td>78.4 (155%)</td>
<td>(not needed)</td>
<td>82.7 (164%)</td>
</tr>
<tr>
<td>Dot Product</td>
<td>20M</td>
<td>1.88 (CUBLAS)</td>
<td>2.35 (125%)</td>
<td>3.90 (207%)</td>
<td>(not needed)</td>
</tr>
<tr>
<td>Fluid Flow</td>
<td>2M</td>
<td>5461 (Repa-N7)</td>
<td>107 (1.96%)</td>
<td>(not needed)</td>
<td>119 (2.18%)</td>
</tr>
<tr>
<td>Mandelbrot (limit)</td>
<td>2M</td>
<td>14.0 (CUDA)</td>
<td>24.0 (171%)</td>
<td>245 (1750%)</td>
<td>245 (1750%)</td>
</tr>
<tr>
<td>N-Body</td>
<td>32k</td>
<td>54.4 (CUDA)</td>
<td>607 (1116%)</td>
<td>(out of memory)</td>
<td>(out of memory)</td>
</tr>
<tr>
<td>Radix sort</td>
<td>4M</td>
<td>780 (Nikola)</td>
<td>442 (56%)</td>
<td>657 (84%)</td>
<td>657 (84%)</td>
</tr>
<tr>
<td>SMVM (protein)</td>
<td>4M</td>
<td>0.641 (CUSP)</td>
<td>0.637 (99%)</td>
<td>32.8 (5115%)</td>
<td>(not needed)</td>
</tr>
<tr>
<td>Name</td>
<td>Non-zeros (nnz/row)</td>
<td>CUSP</td>
<td>Accelerate</td>
<td>Accelerate no fusion</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------</td>
<td>------</td>
<td>------------</td>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>Dense</td>
<td>4M (2K)</td>
<td>14.48</td>
<td>14.62</td>
<td>3.41</td>
<td></td>
</tr>
<tr>
<td>Protein</td>
<td>4.3M (119)</td>
<td>13.55</td>
<td>13.65</td>
<td>0.26</td>
<td></td>
</tr>
<tr>
<td>FEM/Spheres</td>
<td>6M (72)</td>
<td>12.63</td>
<td>9.03</td>
<td>4.70</td>
<td></td>
</tr>
<tr>
<td>FEM/Cantilever</td>
<td>4M (65)</td>
<td>11.98</td>
<td>7.96</td>
<td>4.41</td>
<td></td>
</tr>
<tr>
<td>Wind Tunnel</td>
<td>11.6M (53)</td>
<td>11.98</td>
<td>7.33</td>
<td>4.62</td>
<td></td>
</tr>
<tr>
<td>FEM/Harbour</td>
<td>2.37M (50)</td>
<td>9.42</td>
<td>6.14</td>
<td>0.13</td>
<td></td>
</tr>
<tr>
<td>QCD</td>
<td>1.9M (39)</td>
<td>7.79</td>
<td>4.66</td>
<td>0.13</td>
<td></td>
</tr>
<tr>
<td>FEM/Ship</td>
<td>3.98 (28)</td>
<td>12.28</td>
<td>6.60</td>
<td>4.47</td>
<td></td>
</tr>
<tr>
<td>Economics</td>
<td>1.27M (6)</td>
<td>4.59</td>
<td>0.90</td>
<td>1.06</td>
<td></td>
</tr>
<tr>
<td>Epidemiology</td>
<td>1.27M (4)</td>
<td>6.42</td>
<td>0.59</td>
<td>0.91</td>
<td></td>
</tr>
<tr>
<td>FEM/Accelerator</td>
<td>2.62M (22)</td>
<td>5.41</td>
<td>3.08</td>
<td>2.92</td>
<td></td>
</tr>
<tr>
<td>Circuit</td>
<td>959k (6)</td>
<td>3.56</td>
<td>0.82</td>
<td>1.08</td>
<td></td>
</tr>
<tr>
<td>Webbase</td>
<td>3.1M (3)</td>
<td>2.11</td>
<td>0.47</td>
<td>0.74</td>
<td></td>
</tr>
<tr>
<td>LP</td>
<td>11.3M (2825)</td>
<td>5.22</td>
<td>5.04</td>
<td>2.41</td>
<td></td>
</tr>
</tbody>
</table>

GFLOPS/s (higher is better)
THE ACCELERATE PROJECT

➤ Open source project
  - https://github.com/AccelerateHS/

➤ Current project members
  - Trevor McDonell
  - Rob Everest
  - Josh Meredith
  - Manuel Chakrabarty