GPU Programming I

With thanks to Manuel Chakravarty
for some borrowed slides

GPUs change the game

Gaming drives development

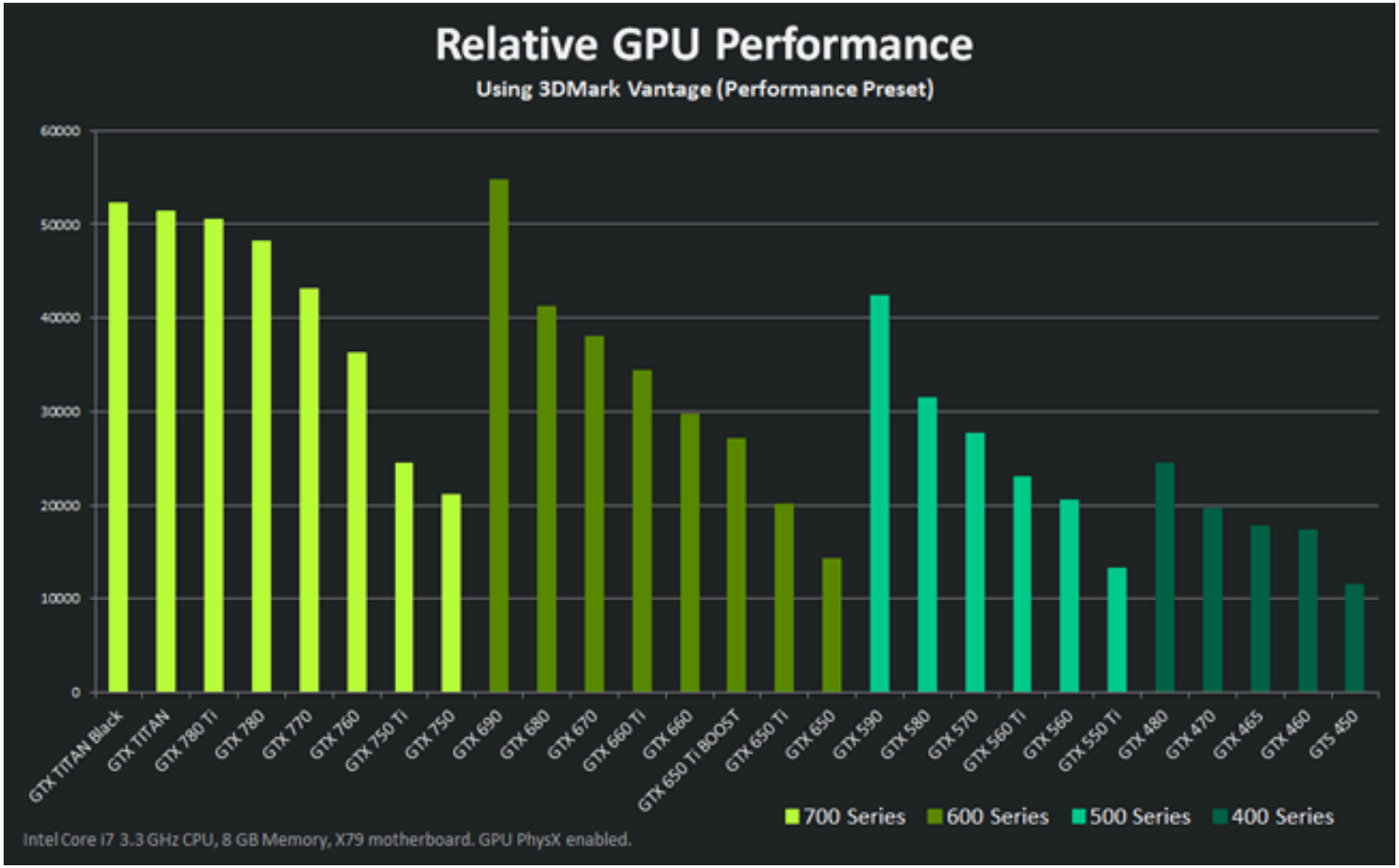


Image from http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-black/performance

GPGPU benefits

General Purpose programming on GPU

GPUs used to be very graphics-specific (shaders and all that)

The pipeline is becoming more and more general purpose even in the gaming GPUs, and there are also special GPUs for GPGPU (more expensive, double precision).

Typical GPGPU users are from finance, sciences needing simulation, bioinformatics etc.

See http://gpgpu.org/

Processing power

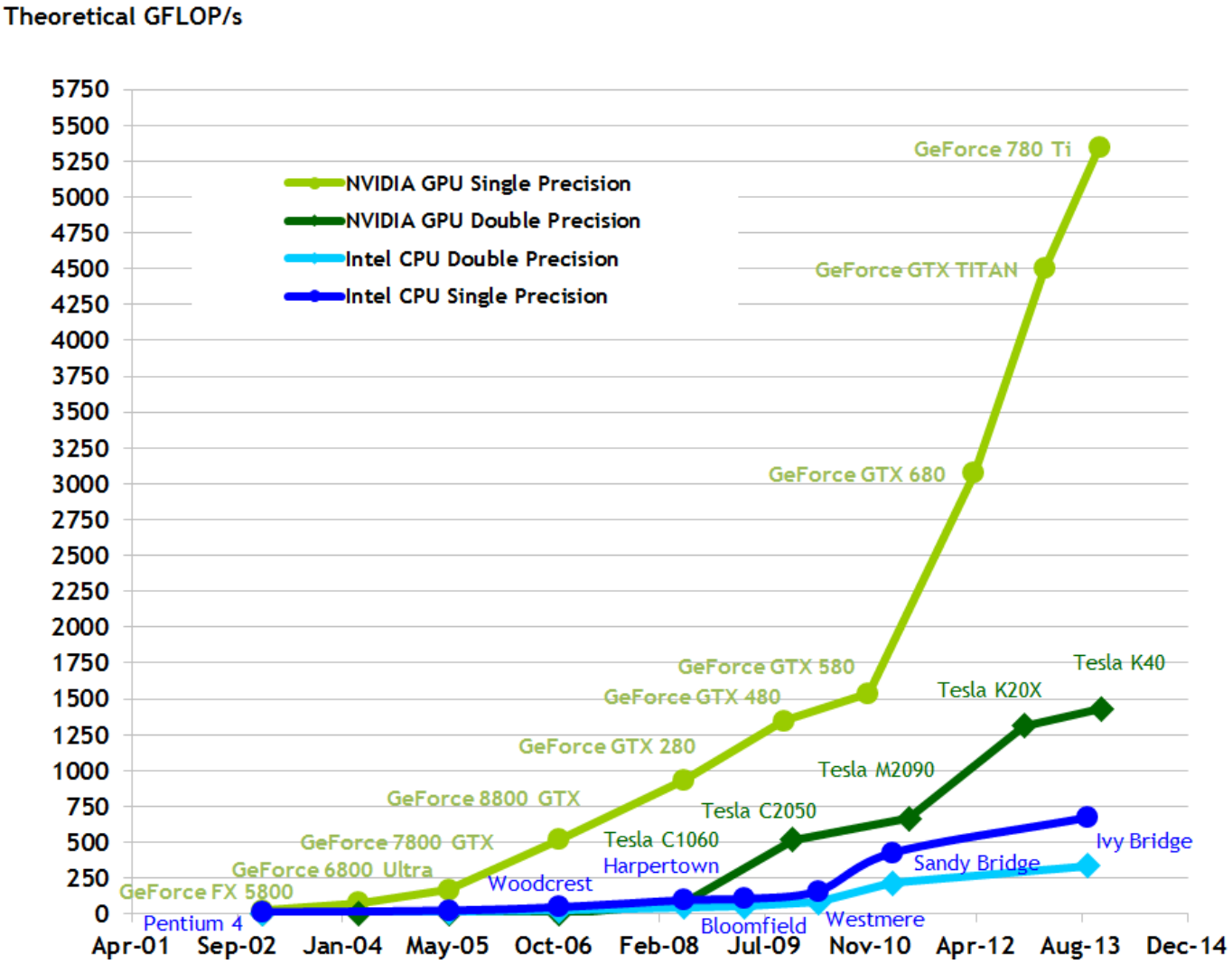
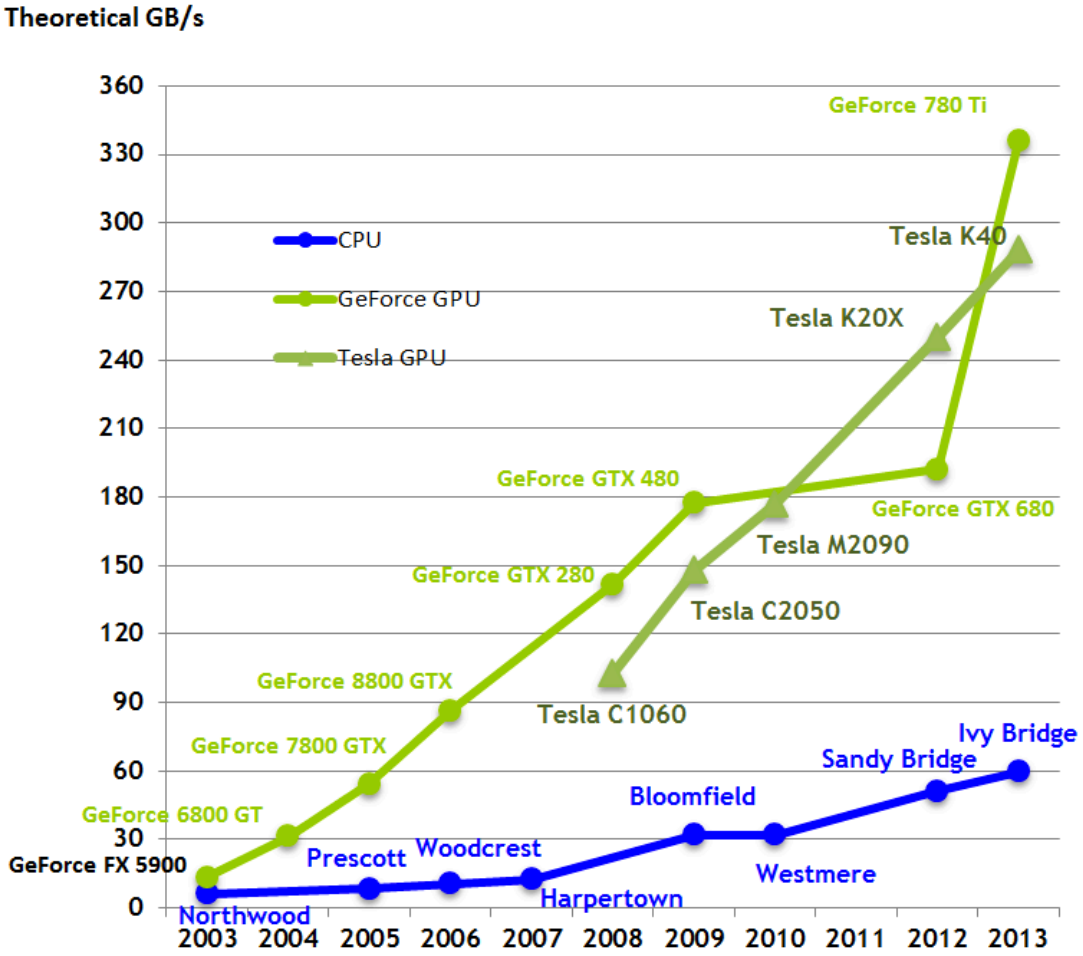


Image from http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#abstract

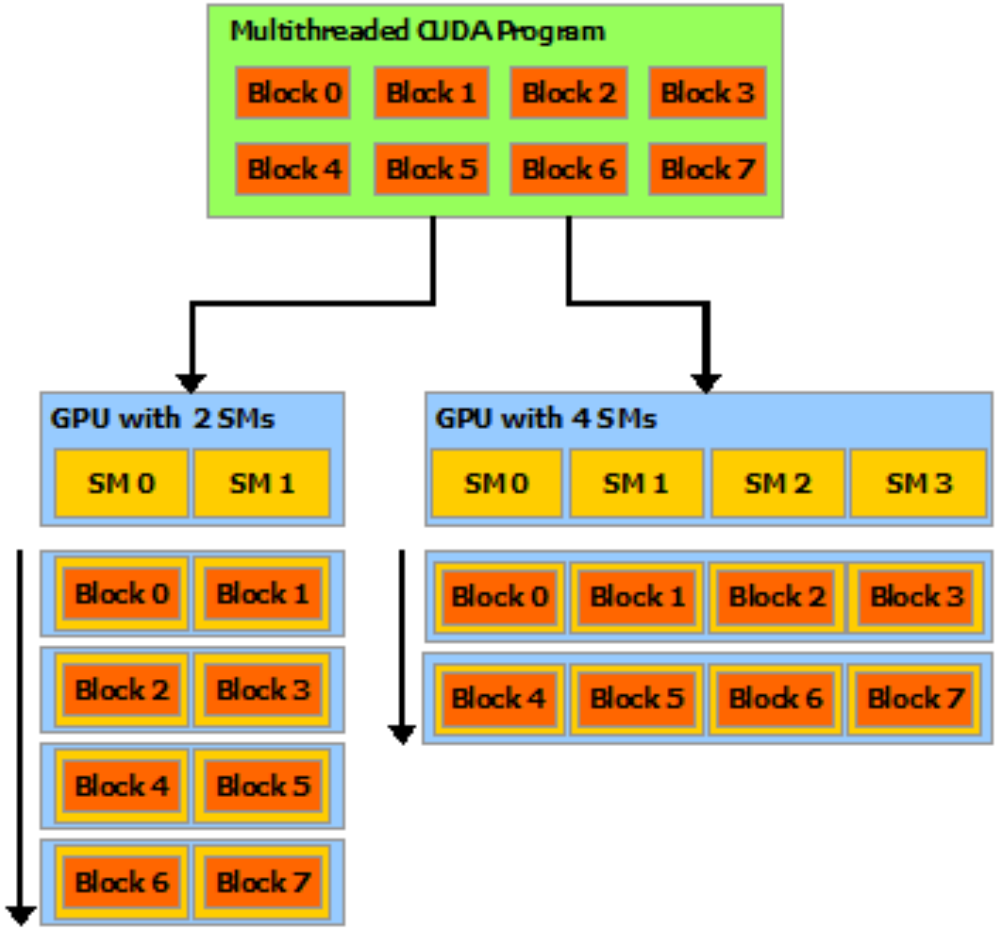
Bandwidth to memory



Transistors used differently

Image from http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#abstract

Need a new programming model



SM = multiprocessor with many small cores/ALUs. Program should run both on wimpy GPU and on a hefty one. MANY threads need to be launched onto the GPU.

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GT 650M"
  CUDA Driver Version / Runtime Version 5.5 / 5.5
  CUDA Capability Major/Minor version number: 3.0
  Total amount of global memory: 1024 MBytes (1073414144 bytes)
  (2) Multiprocessors, (192) CUDA Cores/MP: 384 CUDA Cores
  GPU Clock rate: 900 MHz (0.90 GHz)
  Memory Clock rate: 2508 Mhz
  Memory Bus Width: 128-bit
  ...
  Total amount of constant memory: 65536 bytes
  Total amount of shared memory per block: 49152 bytes
  Total number of registers available per block: 65536
  Warp size: 32
  Maximum number of threads per multiprocessor: 2048
  Maximum number of threads per block: 1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
  ...

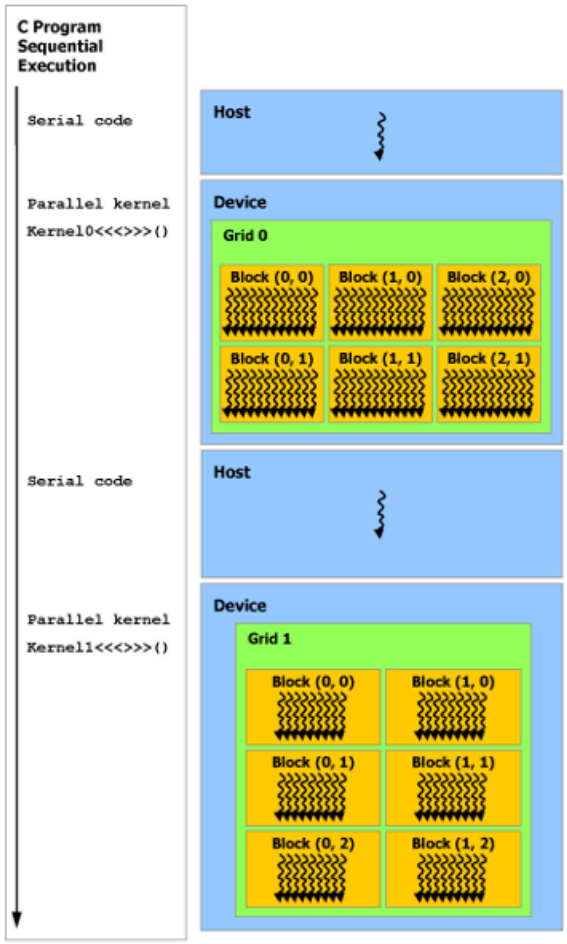


Image from http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#heterogeneous-programming

CUDA C

Gives the user fine control over all of this

User must be aware of the memory hierarchy and of costs of memory access patterns

CUDA programming is great fun (but not the subject of this course) !

OpenCL is a sort of platform-independent CUDA

Raising the level of abstraction

Imperative

Thrust library (C++ template lib. Similar to STL)

CUB library (reusable software components for every layer of the CUDA hierarchy. Very cool!)

PyCUDA, Copperhead and many more

Raising the level of abstraction

Functional

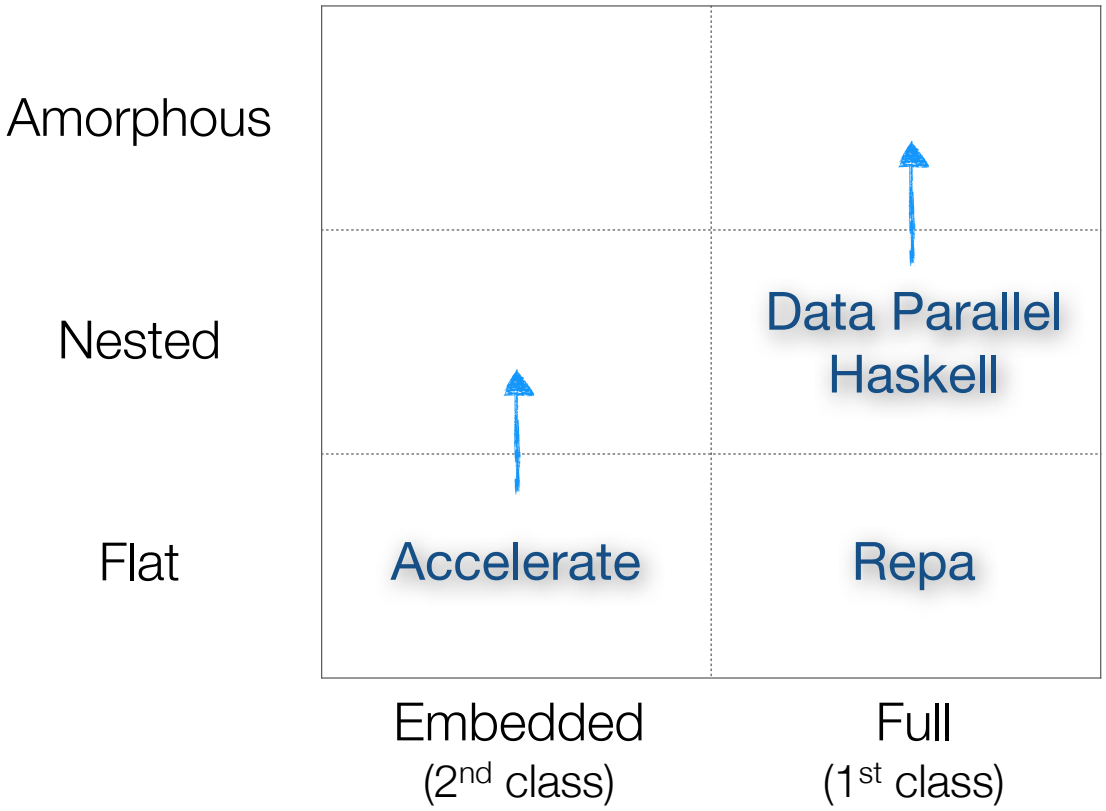
- Accelerate (this lecture)
- Obsidian (next lecture)

(both EDSLs in Haskell generating CUDA)

Nova (U. Edinburgh and NVIDIA, skeleton-based like Accelerate, IR looks generally interesting)

and more

Accelerate


Amorphous
Nested
Flat
Data Parallel Haskell
Accelerate
Repa
Embedded
(2nd class)
Full
(1st class)

Accelerating Haskell Array Codes with Multicore GPUs

Manuel M. T. Chakravarty[†] Gabriele Keller[†] Sean Lee[‡†] Trevor L. McDonell[†] Vinod Grover[‡]

[†]University of New South Wales, Australia
{chak,keller,seanl,tmcdonell}@cse.unsw.edu.au

[‡]NVIDIA Corporation, USA
{selee,vgrover}@nvidia.com

Abstract

Current GPUs are massively parallel multicore processors optimised for workloads with a large degree of SIMD parallelism. Good performance requires highly idiomatic programs, whose development is work intensive and requires expert knowledge.

To raise the level of abstraction, we propose a domain-specific high-level language of array computations that captures appropriate idioms in the form of collective array operations. We embed this purely functional array language in Haskell with an online code generator for NVIDIA's CUDA GPGPU programming environment. We regard the embedded language's collective array operations as algorithmic skeletons; our code generator instantiates CUDA implementations of those skeletons to execute embedded array programs.

This paper outlines our embedding in Haskell, details the design and implementation of the dynamic code generator, and reports on initial benchmark results. These results suggest that we can compete with moderately optimised native CUDA code, while enabling much simpler source programs.

25]. Our work is in that same spirit: we propose a domain-specific high-level language of array computations, called *Accelerate*, that captures appropriate idioms in the form of parameterised, collective array operations. Our choice of operations was informed by the *scan-vector model* [11], which is suitable for a wide range of algorithms, and of which Sengupta et al. demonstrated that these operations can be efficiently implemented on modern GPUs [30].

We regard Accelerate's collective array operations as algorithmic skeletons that capture a range of GPU programming idioms. Our dynamic code generator instantiates CUDA implementations of these skeletons to implement embedded array programs. Dynamic code generation can exploit runtime information to optimise GPU code and enables on-the-fly generation of embedded array programs by the host program. Our code generator minimises the overhead of dynamic code generation by caching binaries of previously compiled skeleton instantiations and by parallelising code generation, host-to-device data transfers, and GPU kernel loading and configuration.

In contrast to our earlier prototype of an embedded language

DAMP'11

Accelerate overall structure

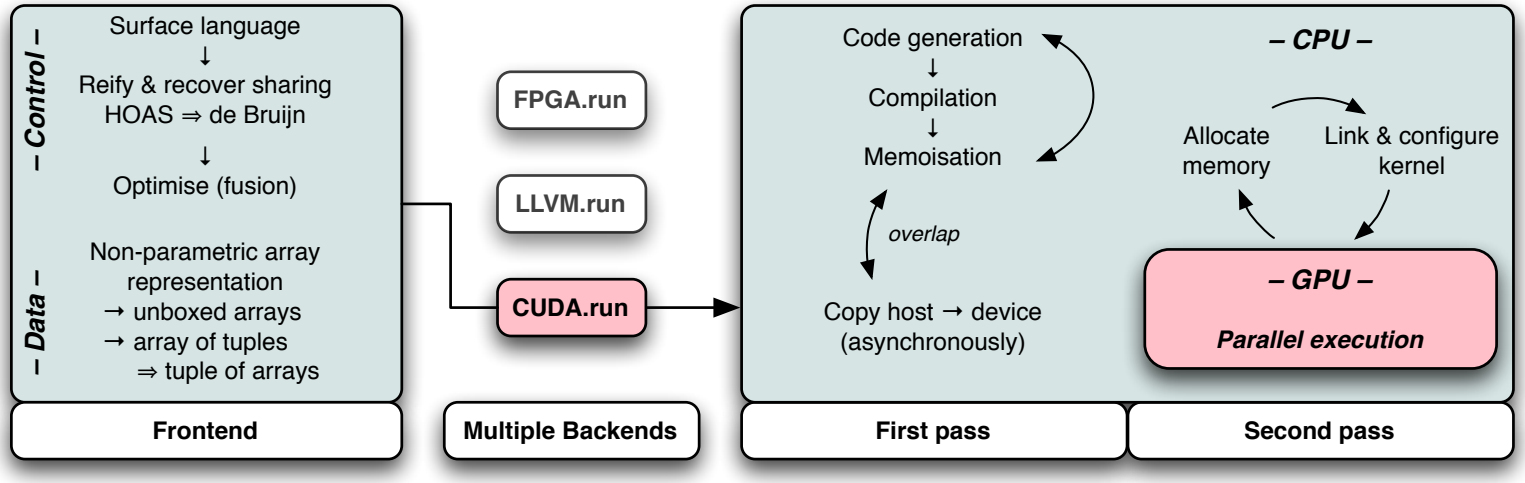
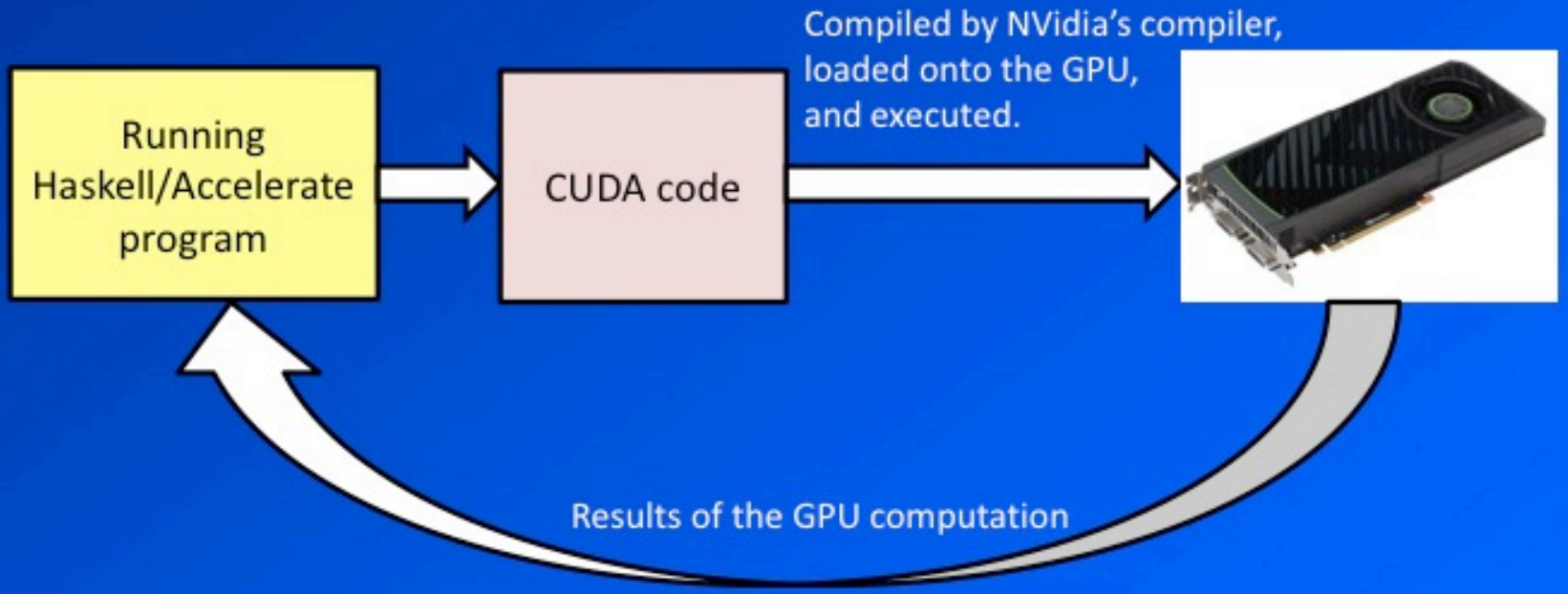


Figure 2. Overall structure of `Data.Array.Accelerate`.

(from the DAMP'11 paper)

Accelerate

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



- 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

User's view (slide by S. Marlow, with thanks)

Embedded code-generating DSL

You write a Haskell program that generates CUDA programs

But the program should look very like a Haskell program (even though it is actually producing ASTs)!

Repa shape-polymorphic arrays reappear

```
data Z = Z — rank-0

data tail :. head = tail :. head — increase rank by 1

type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int
type DIM3 = DIM2 :. Int 〈and so on〉

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

Dot product in Haskell

```
dotp_list :: [Float] -> [Float] -> Float
dotp_list xs ys = foldl (+) 0 (zipWith (*) xs ys)
```

Dot product in Accelerate

```
dotp :: Acc (Vector Float) -> Acc (Vector Float)
        -> Acc (Scalar Float)

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

Dot product in Accelerate

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

Moving an array (literally)

from the Haskell world to the Accelerate world

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

Implies a host to device transfer

Moving an array (literally)

from the Haskell world to the Accelerate world

use :: (S

Implies

Computations in Acc are run on the device

They work on arrays and tuples of arrays.

Remember we are talking about FLAT data parallelism

However, arrays of tuples are allowed (and get converted to tuples of arrays internally)

Plain Haskell code is run on the host

What happens with dot product?

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

This results (in the original Accelerate) in 2 kernels, one for fold and one for zipWith

Collective array operations = kernels

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

Collective array operations = kernels

```
zipWit
 :: (Sh
    (Exp
  -> A
```

- 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

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

Experimenting

```
Prelude> import Data.Array.Accelerate as A
Prelude A> import Data.Array.Accelerate.Interpreter as I
```

Using the interpreter (on the host)

```
Prelude> import Data.Array.Accelerate as A
Prelude A> import Data.Array.Accelerate.Interpreter as I

Prelude A I> let arr = fromList (Z:.3:.5) [1..] :: Array DIM2 Int
```

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

Using the interpreter (on the host)

```
Prelude> import Data.Array.Accelerate as A
Prelude A> import Data.Array.Accelerate.Interpreter as I

Prelude A I> let arr = fromList (Z:.3:.5) [1..] :: Array DIM2 Int
Prelude A I> 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]
```

Using the interpreter (on the host)

```
Prelude> import Data.Array.Accelerate as A
Prelude A> import Data.Array.Accelerate.Interpreter as I

Prelude A I> let arr = fromList (Z:.3:.5) [1..] :: Array DIM2 Int
Prelude A I> 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]
```


run from the interpreter :: Arrays a => Acc a -> a

Even scalars get turned into arrays

```
Prelude> import Data.Array.Accelerate as A
Prelude A> import Data.Array.Accelerate.Interpreter as I

Prelude A I> :t unit
unit :: Elt e => Exp e -> Acc (Scalar e)
```

Collective array operations = kernels

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

Reduces the shape by one dimension

```
Prelude A I> run $ A.fold (+) 0 (use arr)
Array (Z :. 3) [15,40,65]
```

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

15

40

65

to run on the GPU

```
Prelude A I> import Data.Array.Accelerate.CUDA as C

Prelude A I C> C.run $ A.map (+1) (use arr)
Loading package syb-0.4.0 ... linking ... done.
Loading package filepath-1.3.0.1 ... linking ... done.
Loading package old-locale-1.0.0.5 ... linking ... done.
Loading package time-1.4.0.1 ... linking ... done.
Loading package unix-2.6.0.1 ... linking ... done.
…

Array (Z :. 3 :. 5)
[2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
Prelude A I C> C.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]
```

Second attempt much faster. Kernels are memoised.

```
Prelude A I C> let arr1 = fromList (Z:.1000:.1000) [1..] :: Array DIM2 Int

Prelude A I C> I.run $ A.fold (+) 0 (use arr1)
```

This provokes a bug ☹

*** Exception:
*** Internal error in package accelerate ***
*** Please submit a bug report at https://github.com/AccelerateHS/accelerate/issues
./Data/Array/Accelerate/CUDA/State.hs:86 (unhandled): CUDA Exception: out of memory

Though the 500 by 500 case works fine.

Making an array on the device

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

Reshaping arrays

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

Reshaping arrays

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

Prelude A I C> let arr2 = fromList (Z:.4:.5) [1..] :: Array DIM2 Int

 Prelude A I C> I.run $ fold (+) 0 (reshape (index2 (5 :: Exp Int) 4) (use arr2))
Array (Z :. 5) [10,26,42,58,74]
```

Omit run to see datatype

```
Prelude A I C> fold (+) 0 (reshape (index2 (5 :: Exp Int) 4) (use arr2))


let a0 = use (Array (Z :. 4 :. 5) [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20])
in
let a1 = reshape (Z :. 5 :. 4) a0
in fold (\x0 x1 -> x0 + x1) 0 a1
```

Similarly for map

```
Prelude A I C> A.map (+1) (reshape (index2 (5 :: Exp Int) 4) (use arr2))
let a0 = use (Array (Z :. 4 :. 5) [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20])
in
let a1 = reshape (Z :. 5 :. 4) a0
in map (\x0 -> 1 + x0) a1
```

Similarly for map

```
Prelude A I C> A.map (+1) (reshape (index2 (5 :: Exp Int) 4) (use arr2))
let a0 = use (Array (Z :. 4 :. 5) [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20])
in
let a1 = reshape (Z :. 5 :. 4) a0
in map (\x0 -> 1 + x0) a1
```

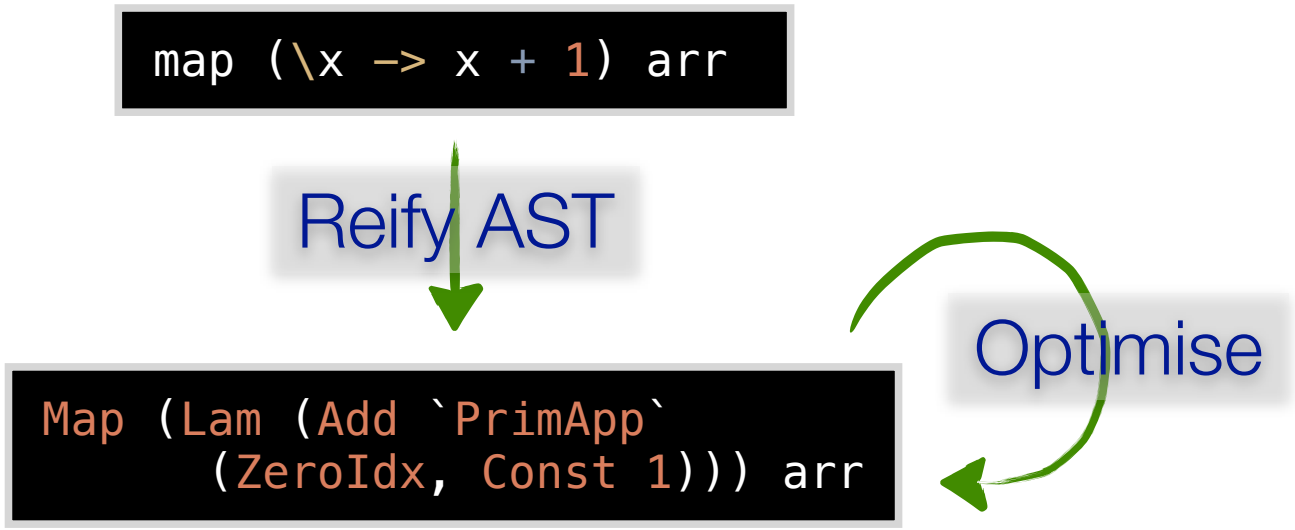
What happens behind the scenes??

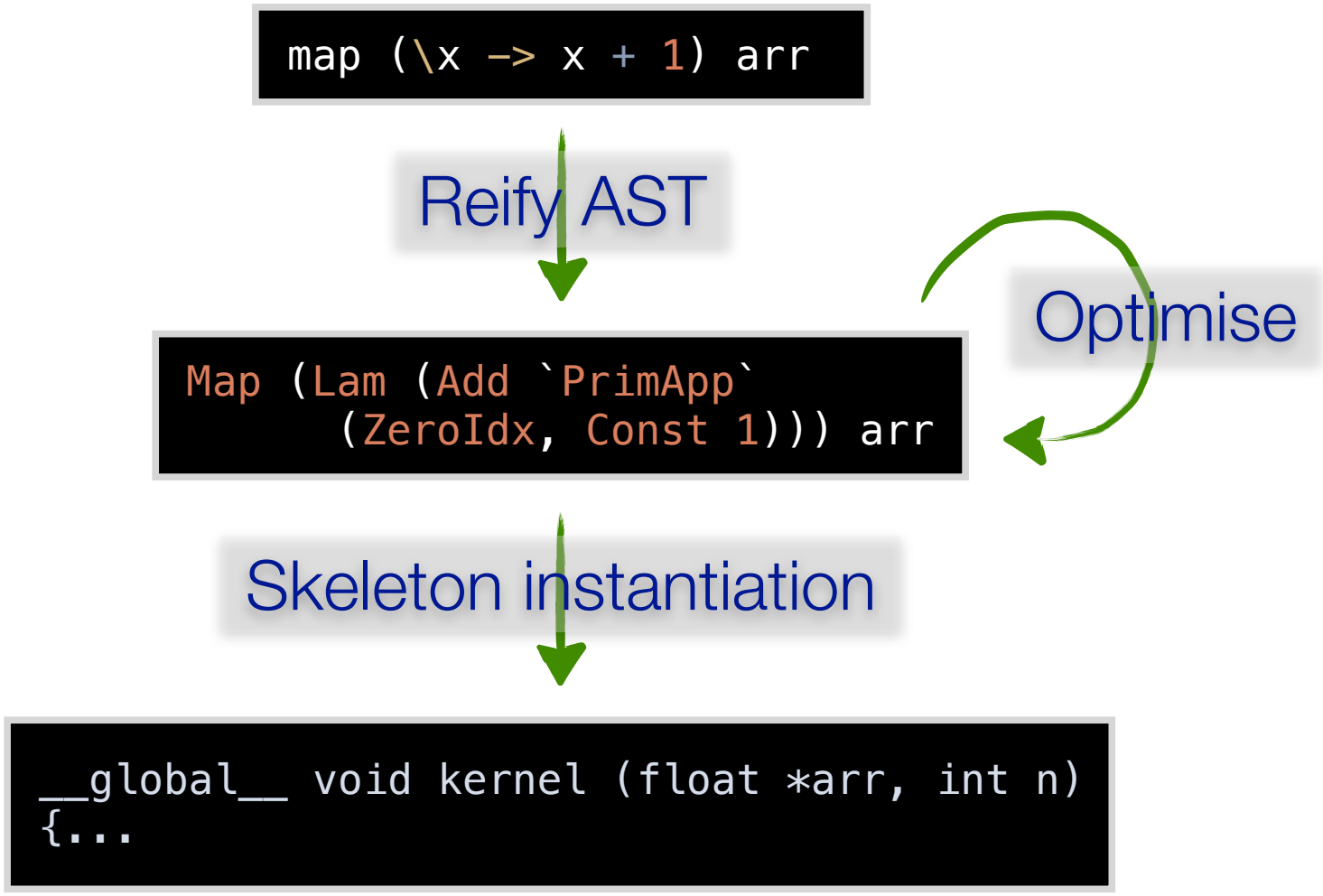
```
map (\x -> x + 1) arr
```

Slides by M. Chakravarty, with thanks

```
map (\x -> x + 1) arr
```


```
Map (Lam (Add `PrimApp`
      (ZeroIdx, Const 1))) arr
```


map (\x -> x + 1) arr
Reify AST
Map (Lam (Add `PrimApp`
(ZeroIdx, Const 1))) arr
Optimise


map (\x -> x + 1) arr
Reify AST
Map (Lam (Add `PrimApp`
(ZeroIdx, Const 1))) arr
Optimise
Skeleton instantiation
__global__ void kernel (float *arr, int n)
{...

```
map (\x -> x + 1) arr
```

Reify AST

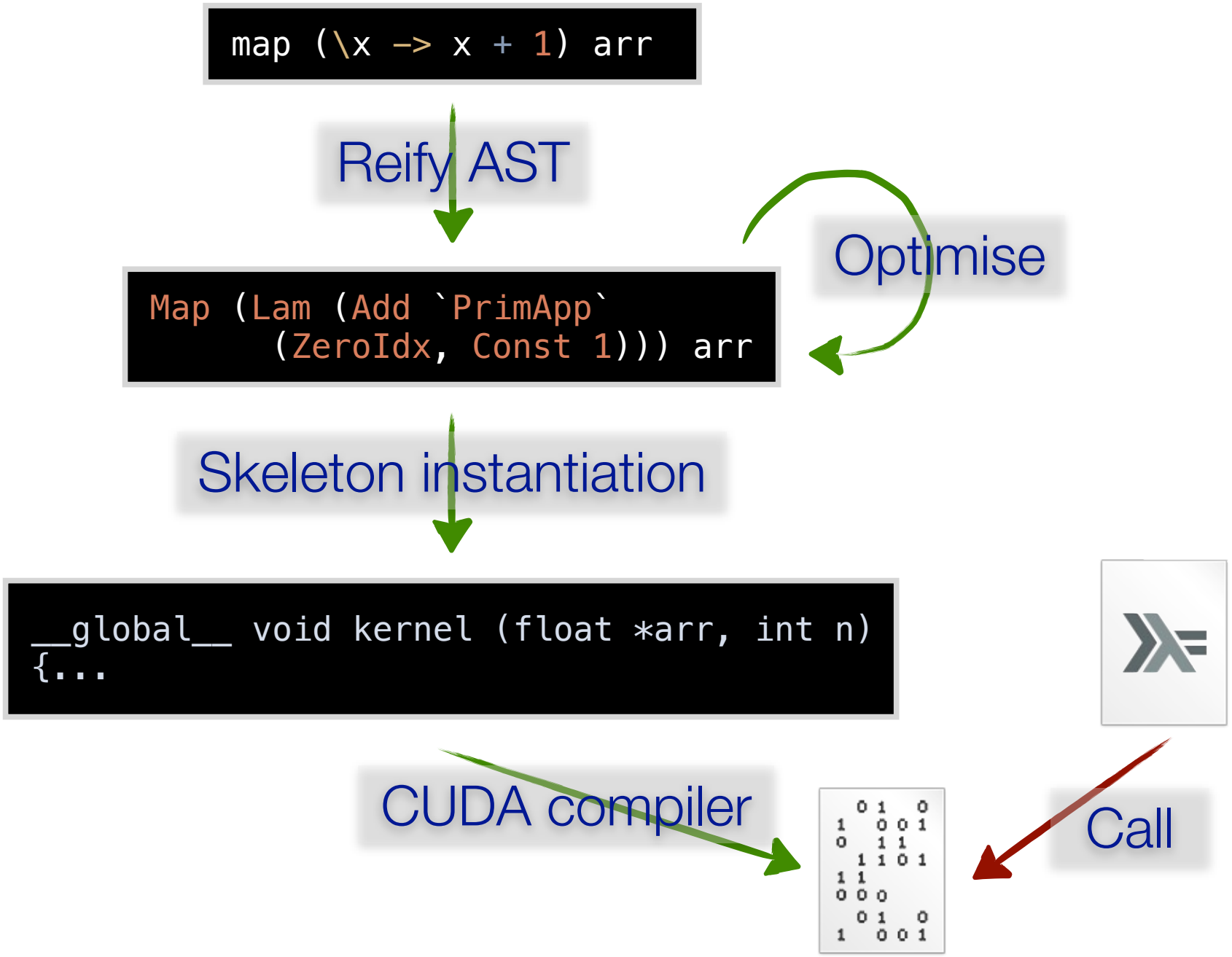
```
Map (Lam (Add `PrimApp`
      (ZeroIdx, Const 1))) arr
```

Optimise

Skeleton instantiation

```
__global__ void kernel (float *arr, int n)
{...
```

CUDA compiler


map (\x -> x + 1) arr
Reify AST
Map (Lam (Add `PrimApp`
(ZeroIdx, Const 1))) arr
Optimise
Skeleton instantiation
__global__ void kernel (float *arr, int n)
{...
CUDA compiler
Call

```
mkMap dev aenv fun arr = return $
  CUTranslSkel "map" [cunit|

  $esc:("#include <accelerate_cuda.h>")
  extern "C" __global__ void
  map ($params:argIn, $params:argOut) {
    const int shapeSize = size(shOut);
    const int gridSize  = $exp:(gridSize dev);
    int ix;

    for ( ix =  $exp:(threadIdx dev)
        ; ix <  shapeSize
        ; ix += gridSize ) {
          $items:(dce x       .=. get ix)
          $items:(setOut "ix" .=. f x)
        }
  } |]
  where ...
```

Combinators as skeletons

Skeleton = code template with holes

Hand tuned

Uses Mainland's CUDA quasi-quoter

Antiquotes such as $params: are the holes

Performance (DAMP'11 paper)

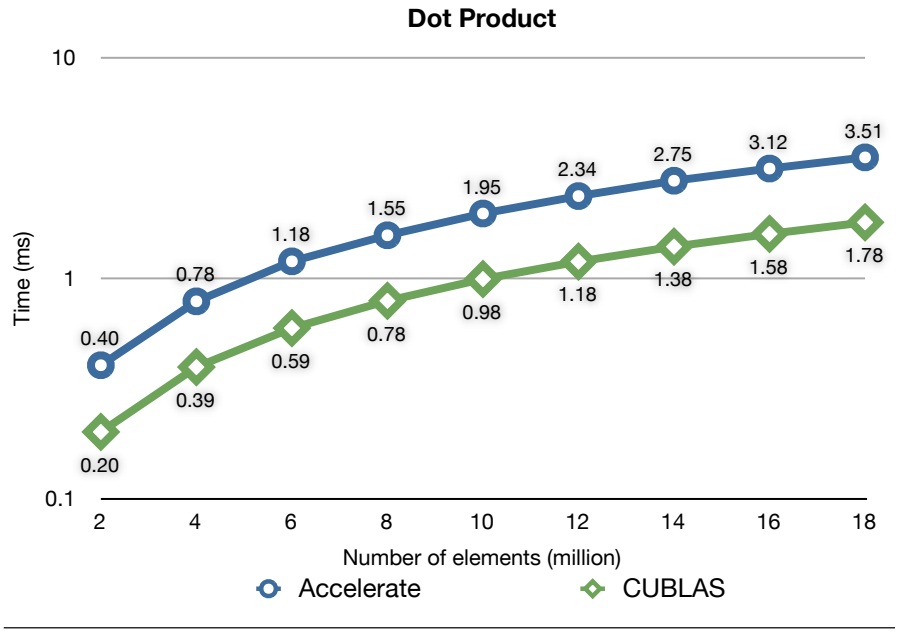


Figure 3. Kernel execution time for a dot product.

Performance (DAMP'11 paper)

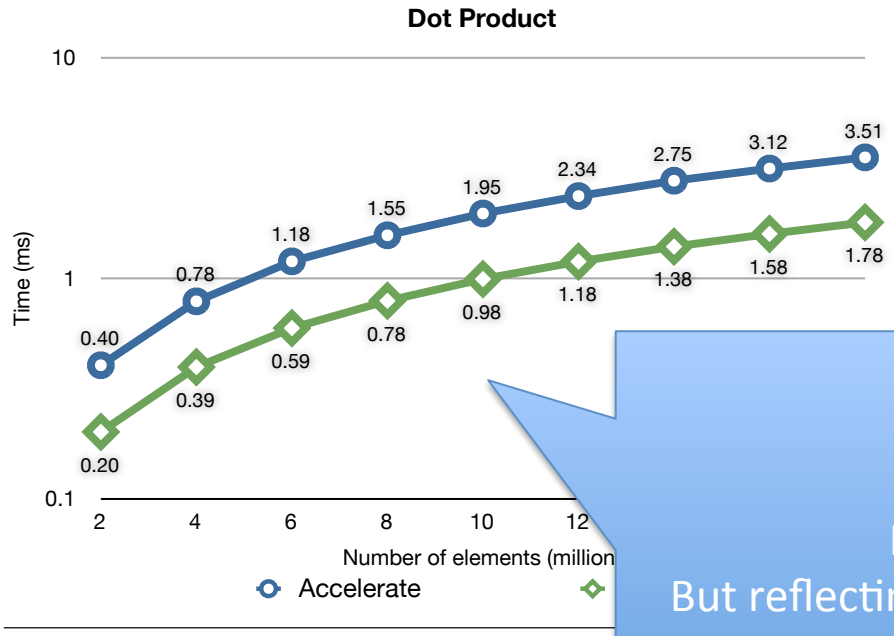


Figure 3. Kernel execution time for a

Pretty good
But reflecting the fact that dotp in Accelerate needs 2 kernels launches

Conclusion (DAMP'11 paper)

Need to tackle fusion of adjacent kernels

Sharing is also an issue

One should write programs to take advantage of kernel memoisation (to reduce kernel generation time)

Optimising Purely Functional GPU Programs

Trevor L. McDonell Manuel M. T. Chakravarty Gabriele Keller Ben Lippmeier
University of New South Wales, Australia
{tmcdonell,chak,keller,benl}@cse.unsw.edu.au

Abstract

Purely functional, embedded array programs are a good match for SIMD hardware, such as GPUs. However, the naive compilation of such programs quickly leads to both code explosion and an excessive use of intermediate data structures. The resulting slowdown is not acceptable on target hardware that is usually chosen to achieve high performance.

In this paper, we discuss two optimisation techniques, *sharing recovery* and *array fusion*, that tackle code explosion and eliminate superfluous intermediate structures. Both techniques are well known from other contexts, but they present unique challenges for an embedded language compiled for execution on a GPU. We present novel methods for implementing sharing recovery and array fusion, and demonstrate their effectiveness on a set of benchmarks.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classification—Applicative (functional) languages; Concurrent, distributed, and parallel languages

Keywords Arrays; Data parallelism; Embedded language; Dynamic compilation; GPGPU; Haskell; Sharing recovery; Array fusion

1. Introduction

Recent work on stream fusion [12], the `vector` package [23], and the parallel array library Repa [17, 19, 20] has demonstrated that (1) the performance of purely functional array code in Haskell can be competitive with that of imperative programs and that (2) purely functional array code lends itself to an efficient parallel implementation on control-parallel multicore CPUs.

programs consisting of multiple kernels the intermediate data structures must be shuffled back and forth across the CPU-GPU bus.

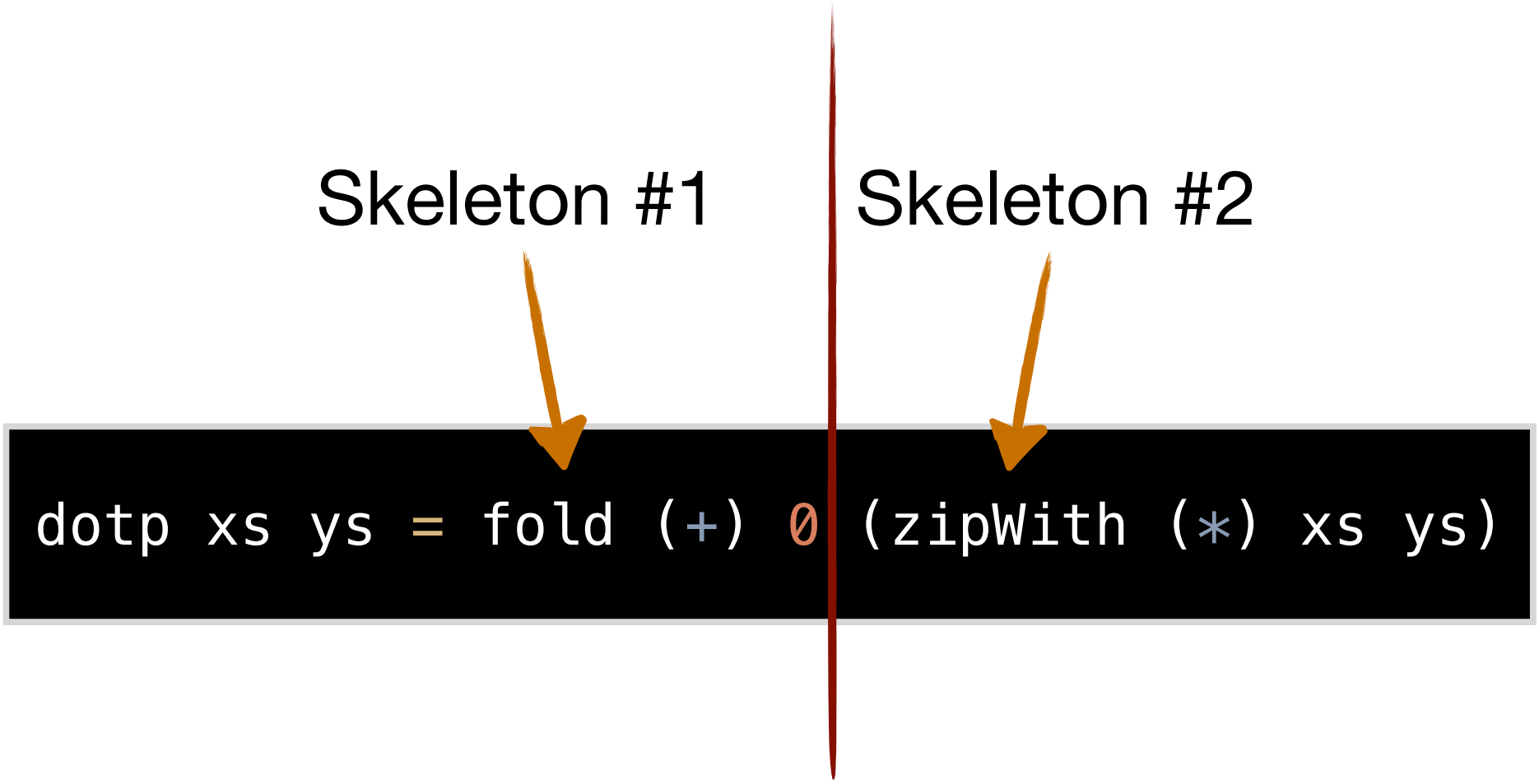
We recently presented *Accelerate*, an EDSL and skeleton-based code generator targeting the CUDA GPU development environment [8]. In the present paper, we present novel methods for optimising the code using *sharing recovery* and *array fusion*.

Sharing recovery for embedded languages recovers the sharing of let-bound expressions that would otherwise be lost due to the embedding. Without sharing recovery, the value of a let-bound expression is recomputed for every use of the bound variable. In contrast to prior work [14] that decomposes expression trees into graphs and fails to be type preserving, our novel algorithm preserves both the tree structure and typing of a deeply embedded language. This enables our runtime compiler to be similarly type preserving and simplifies the backend by operating on a tree-based intermediate language.

Array fusion eliminates the intermediate values and additional GPU kernels that would otherwise be needed when successive bulk operators are applied to an array. Existing methods such as `foldr/build` fusion [15] and stream fusion [12] are not applicable to our setting as they produce tail-recursive loops, rather than the GPU kernels we need for Accelerate. The NDP2GPU system of [4] *does* produce fused GPU kernels, but is limited to simple map/map fusion. We present a fusion method partly inspired by Repa's *delayed arrays* [17] that fuses more general producers and consumers, while retaining the combinator based program representation that is essential for GPU code generation using skeletons.

With these techniques, we provide a high-level programming model that supports shape-polymorphic maps, generators, reductions, permutation and stencil-based operations, while maintaining performance that often approaches hand-written CUDA code.

ICFP'13



Intermediate array

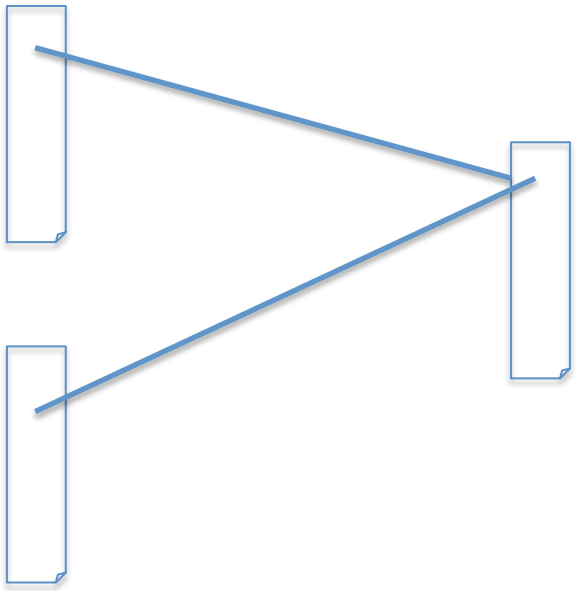
Extra traversal

Combined skeleton

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

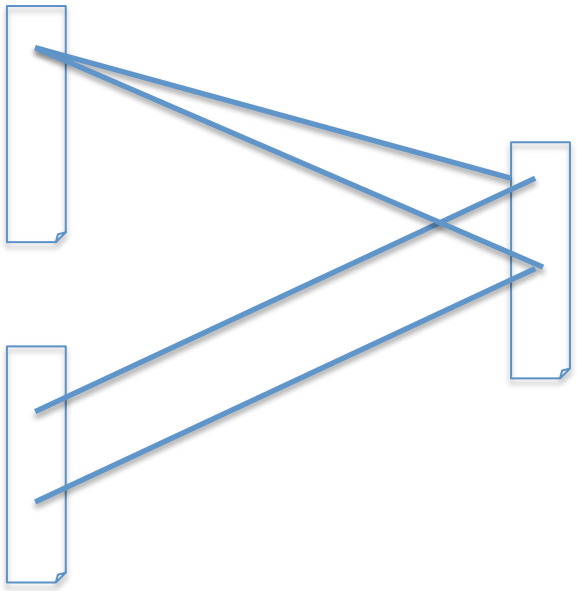
Producers

"Operations where each element of the result array depends on at most one element of each input array. Multiple elements of the output array may depend on a single input array element, but all output elements can be computed independently. We refer to these operations as producers."



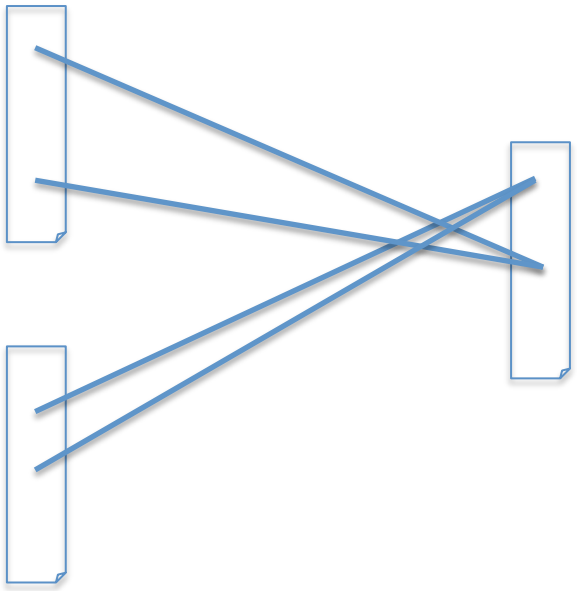
Producers

"Operations where each element of the result array depends on at most one element of each input array. Multiple elements of the output array may depend on a single input array element, but all output elements can be computed independently. We refer to these operations as producers."



Consumers

"Operations where each element of the result array depends on multiple elements of the input array. We call these functions consumers, in spite of the fact that they also produce an array."



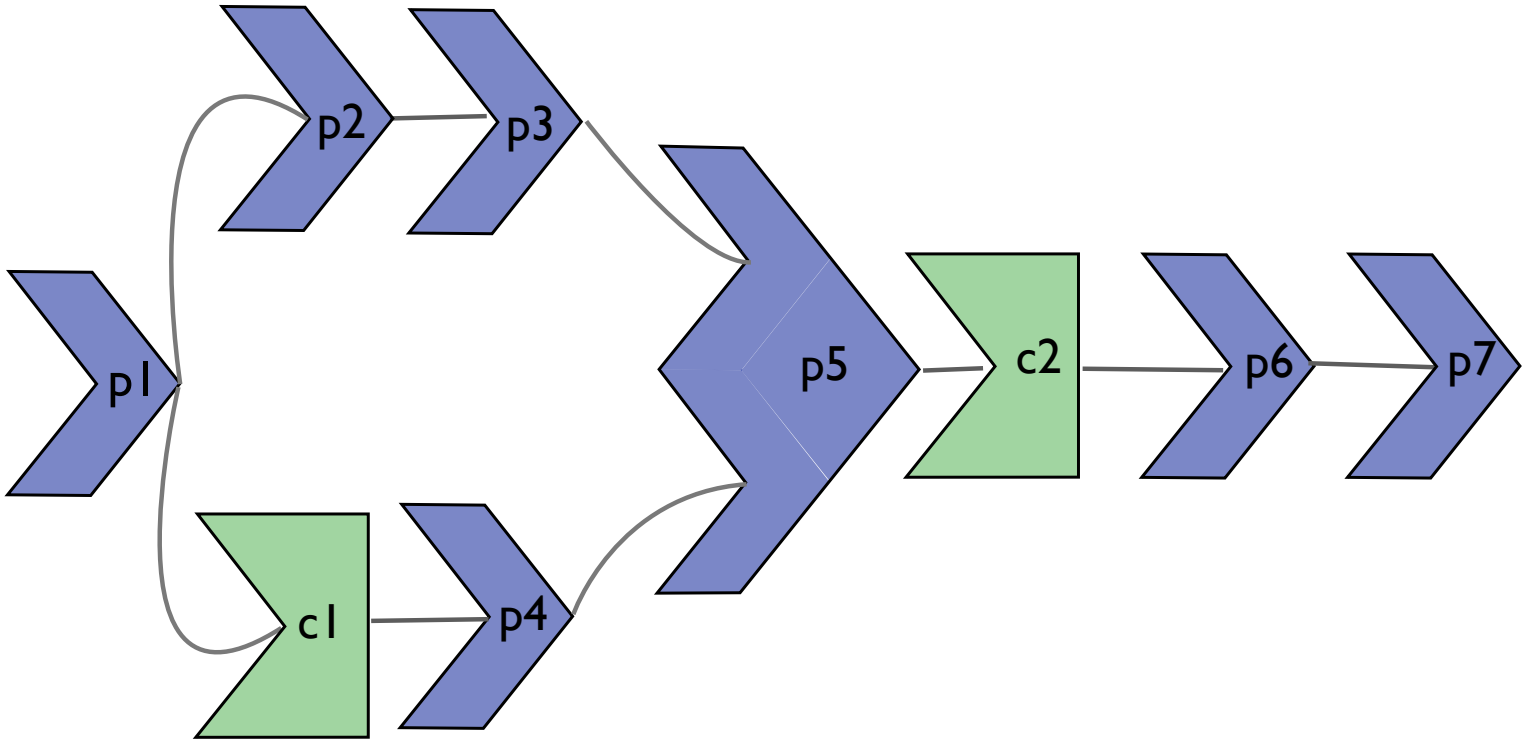
Producers

```
map         :: (Exp a -> Exp b) -> Acc (Array sh a) -> Acc (Array sh b)
zipWith     :: (Exp a -> Exp b -> Exp c) -> Acc (Array sh a) -> Acc (Array sh b)
            -> Acc (Array sh c)
backpermute :: Exp sh' -> (Exp sh' -> Exp sh) -> Acc (Array sh a)
            -> Acc (Array sh' e)
replicate   :: Slice slix => Exp slix
            -> Acc (Array (SliceShape slix) e)
            -> Acc (Array (FullShape  slix) e)
slice       :: Slice slix
            => Acc (Array (FullShape  slix) e) -> Exp slix
            -> Acc (Array (SliceShape slix) e)
generate    :: Exp sh -> (Exp sh -> Exp a) -> Acc (Array sh a)
```

Consumers

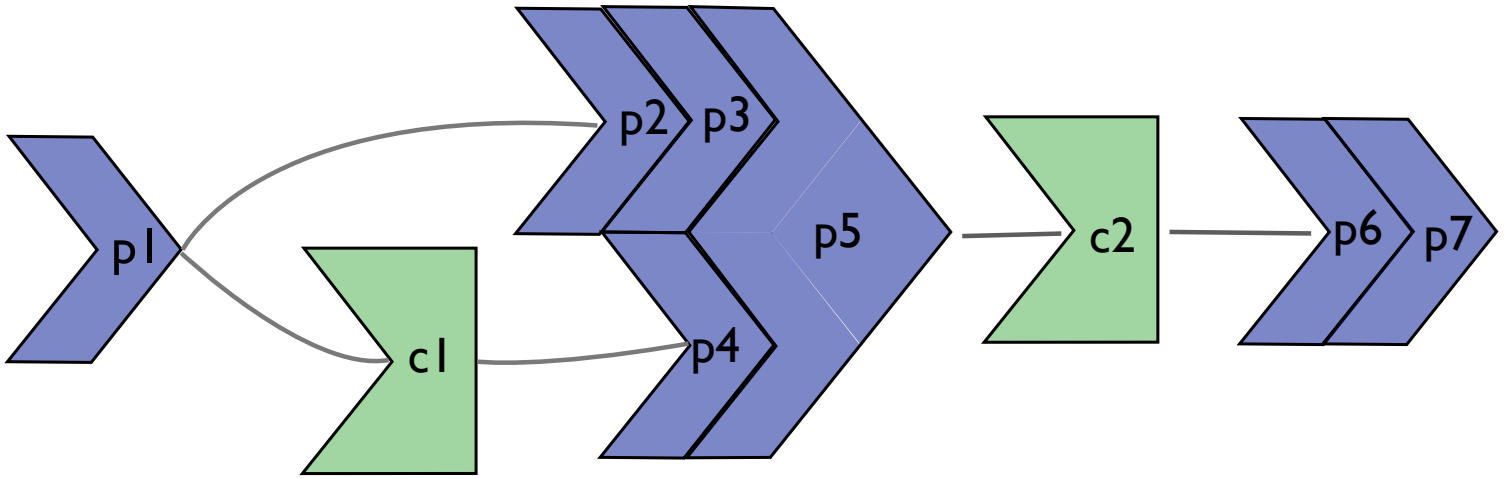
```
fold        :: (Exp a -> Exp a -> Exp a) -> Exp a -> Acc (Array (sh:.Int) a)
            -> Acc (Array sh a)
scan{l,r}   :: (Exp a -> Exp a -> Exp a) -> Exp a -> Acc (Vector a)
            -> Acc (Vector a)
permute     :: (Exp a -> Exp a -> Exp a) -> Acc (Array sh' a)
            -> (Exp sh -> Exp sh') -> Acc (Array sh a) -> Acc (Array sh' a)
stencil     :: Stencil sh a stencil => (stencil -> Exp b) -> Boundary a
            -> Acc (Array sh a) -> Acc (Array sh b)
```

Fusing networks of skeletons



Fusing networks of skeletons

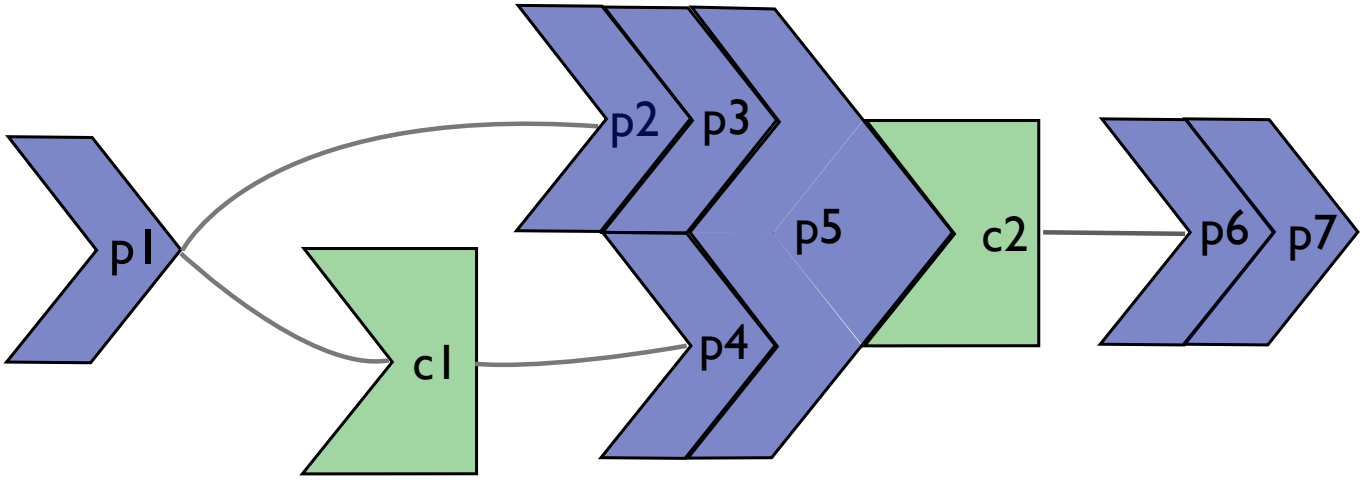
Phase 1: producer/producer fusion



This is the easy case

Fusing networks of skeletons

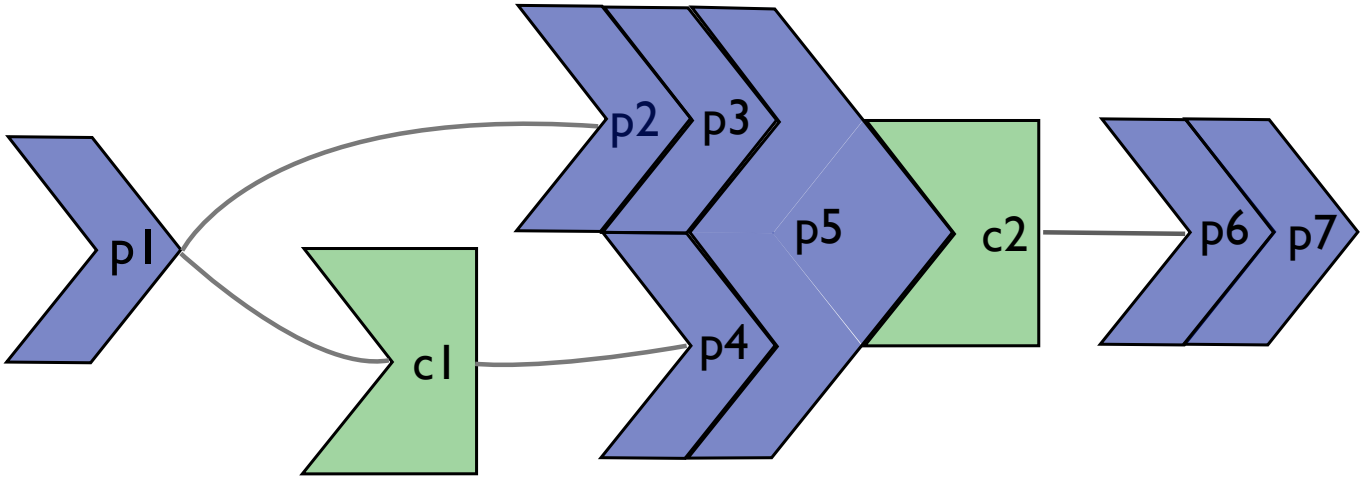
Phase 2: consumer/producer fusion



Fuse a producer followed by a consumer into the consumer
Happens during code generation. Speciaiise consumer skeleton with producer code

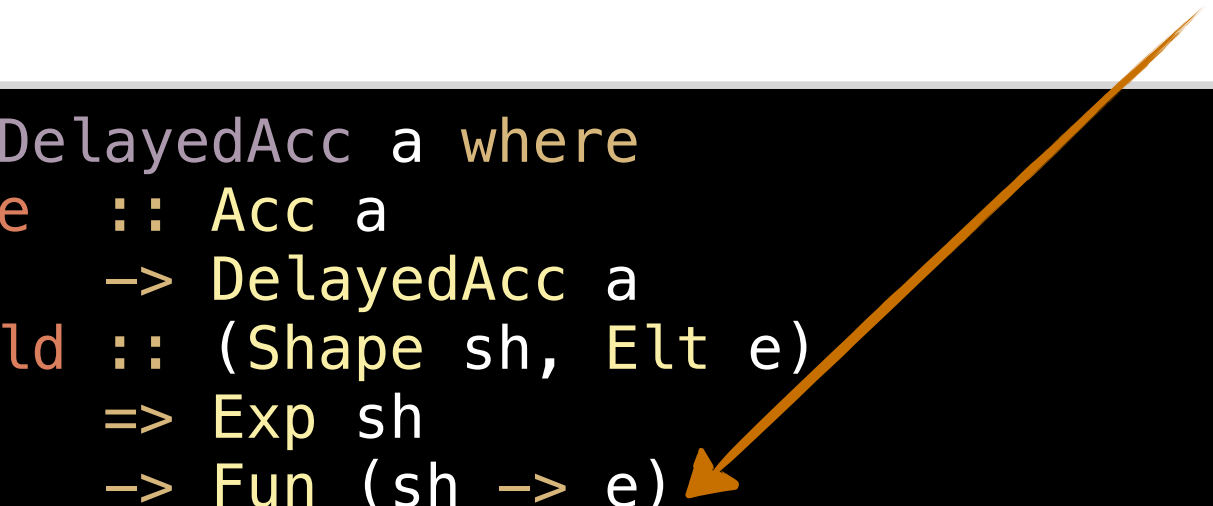
Fusing networks of skeletons

Phase 2: consumer/producer fusion



Producer consumer pairs were not fused at time of writing of the ICFP'13 paper

Fusion friendly

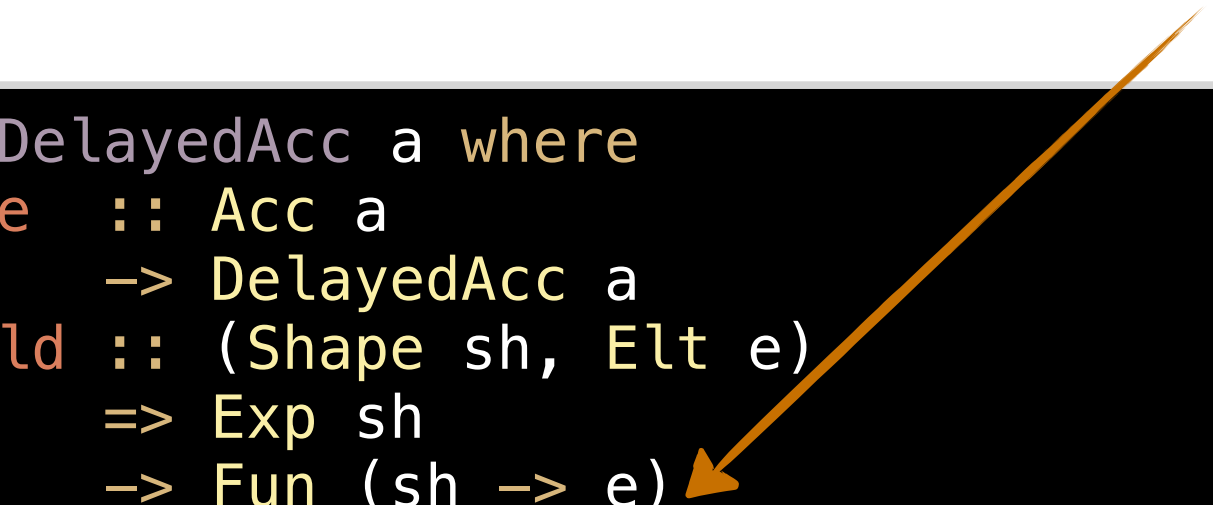


```
data DelayedAcc a where
  Done  :: Acc a
        -> DelayedAcc a
  Yield :: (Shape sh, Elt e)
        => Exp sh
        -> Fun (sh -> e)
        -> DelayedAcc (Array sh e)
  Step  :: …
```

```
mapD f (Yield sh g) = Yield sh (f . g)
```

For Producer Producer fusion, use delayed arrays (like we saw in Repa)

Fusion friendly

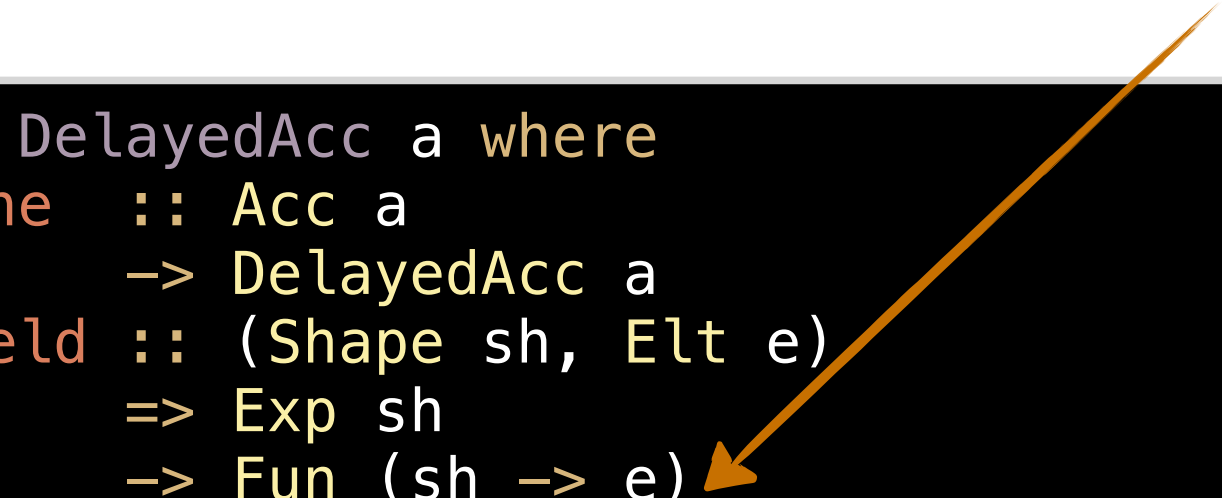


```
data DelayedAcc a where
  Done  :: Acc a
        -> DelayedAcc a
  Yield :: (Shape sh, Elt e)
        => Exp sh
        -> Fun (sh -> e)
        -> DelayedAcc (Array sh e)
  Step  :: …
```

```
mapD f (Yield sh g) = Yield sh (f . g)
```

The third constructor, Step, encodes a special case of the more general Yield that represents the application of an index and/or value space transformation to the argument array.

Fusion friendly



```
data DelayedAcc a where
  Done  :: Acc a
        -> DelayedAcc a
  Yield :: (Shape sh, Elt e)
        => Exp sh
        -> Fun (sh -> e)
        -> DelayedAcc (Array sh e)
  Step  :: …
```

```
mapD f (Yield sh g) = Yield sh (f . g)
```

```
codeGenAcc … (Fold f z arr)
  = mkFold … (codeGenFun f) (codeGenExp z)
             (codeGenEmbeddedAcc arr)
```

Fusion of skeletons

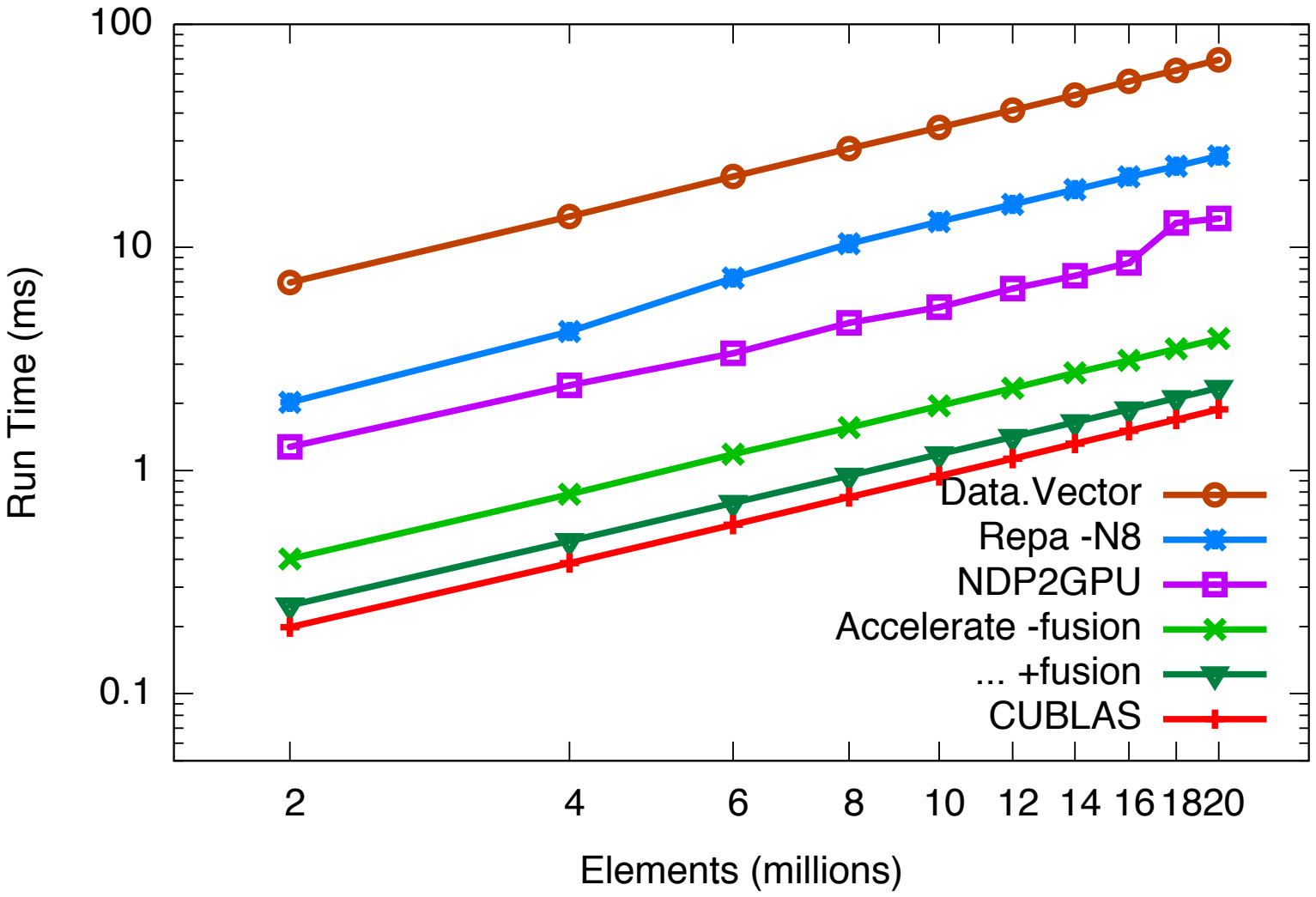
…reduces the abstraction penalty

Code generation idioms vary from high-level combinators

Smart constructors combine producers

Instantiate consumer skeletons with producer code

Dot Product


100
10
1
0.1
Run Time (ms)
2
4
6
8
10
12
14
16
18
20
Elements (millions)
Data.Vector
Repa -N8
NDP2GPU
Accelerate -fusion
... +fusion
CUBLAS

Sharing recovery

```
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))
```

```
riskfree, volatility :: Float
riskfree   = 0.02
volatility = 0.30

horner :: Num a => [a] -> a -> a
horner coeff x = x * foldr1 madd coeff
  where
    madd a b = a + x*b

cnd' :: Floating a => a -> a
cnd' d =
  let poly     = horner coeff
      coeff    = [0.31938153, -0.356563782,
                  1.781477937, -1.821255978,
                  1.330274429]
      rsqrt2pi = 0.39894228040143267793994605993438
      k        = 1.0 / (1.0 + 0.2316419 * abs d)
  in
  rsqrt2pi * exp (-0.5*d*d) * poly k
```

"The function callput includes a significant amount of sharing: the helper functions cnd', and hence also horner, are used twice —for d1 and d2— and its argument d is used multiple times in the body. Our embedded implementation of Accelerate reifies the abstract syntax of the (deeply) embedded language in Haskell. Consequently, each occurrence of a let-bound variable in the source program creates a separate unfolding of the bound expression in the compiled code."

Summary

ICFP'13 paper introduces a new way of doing sharing recovery (a perennial problem in EDSLs)

It also introduces novel ways to fuse functions on arrays

Performance is considerably improved

This is a great way to do GPU programming without bothering too much about how GPUs make life difficult