GPU Programming

With thanks to Manuel Chakravarty
for some borrowed slides

GPUs change the game

Gaming drives development

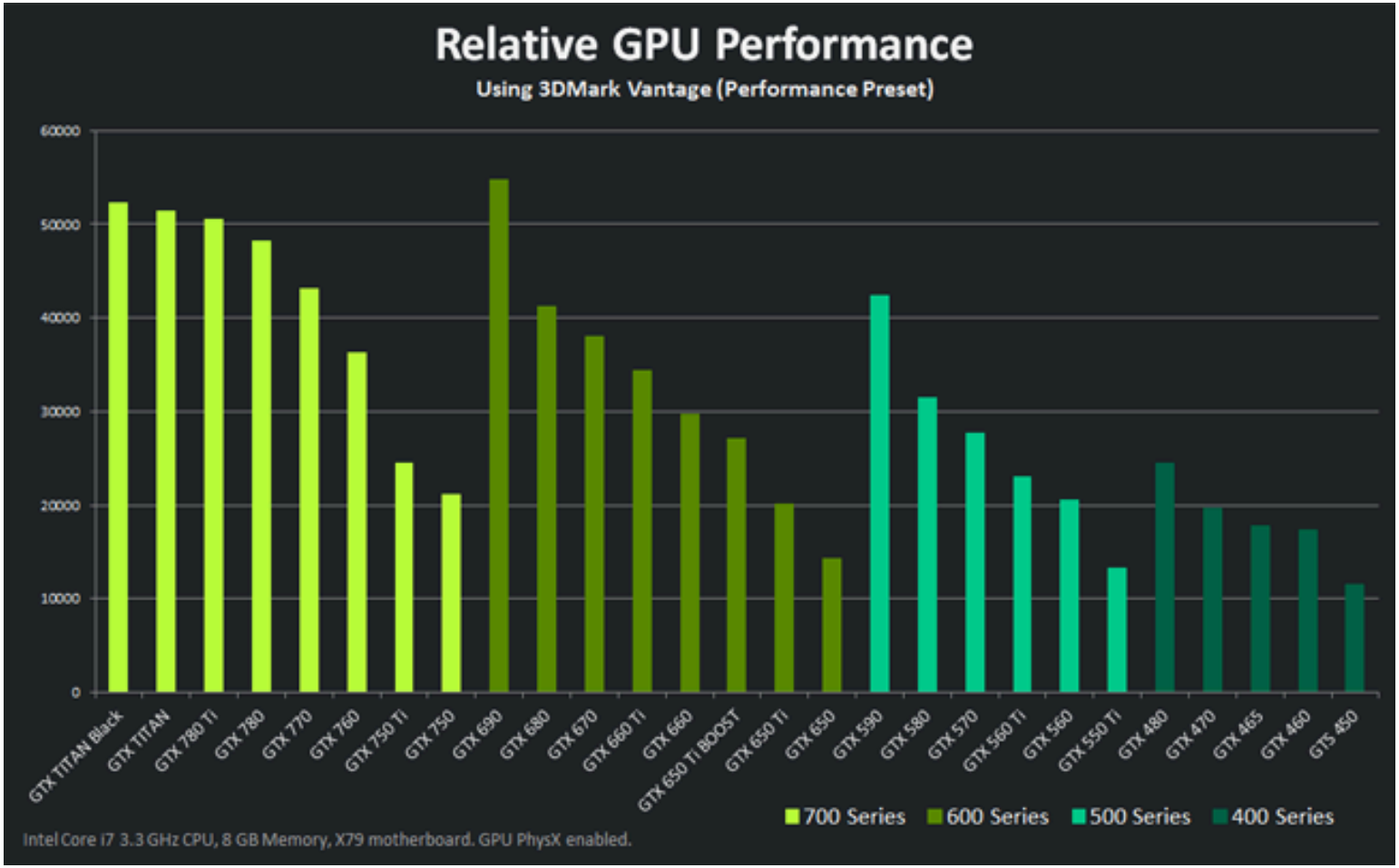


Image from http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-black/performance

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

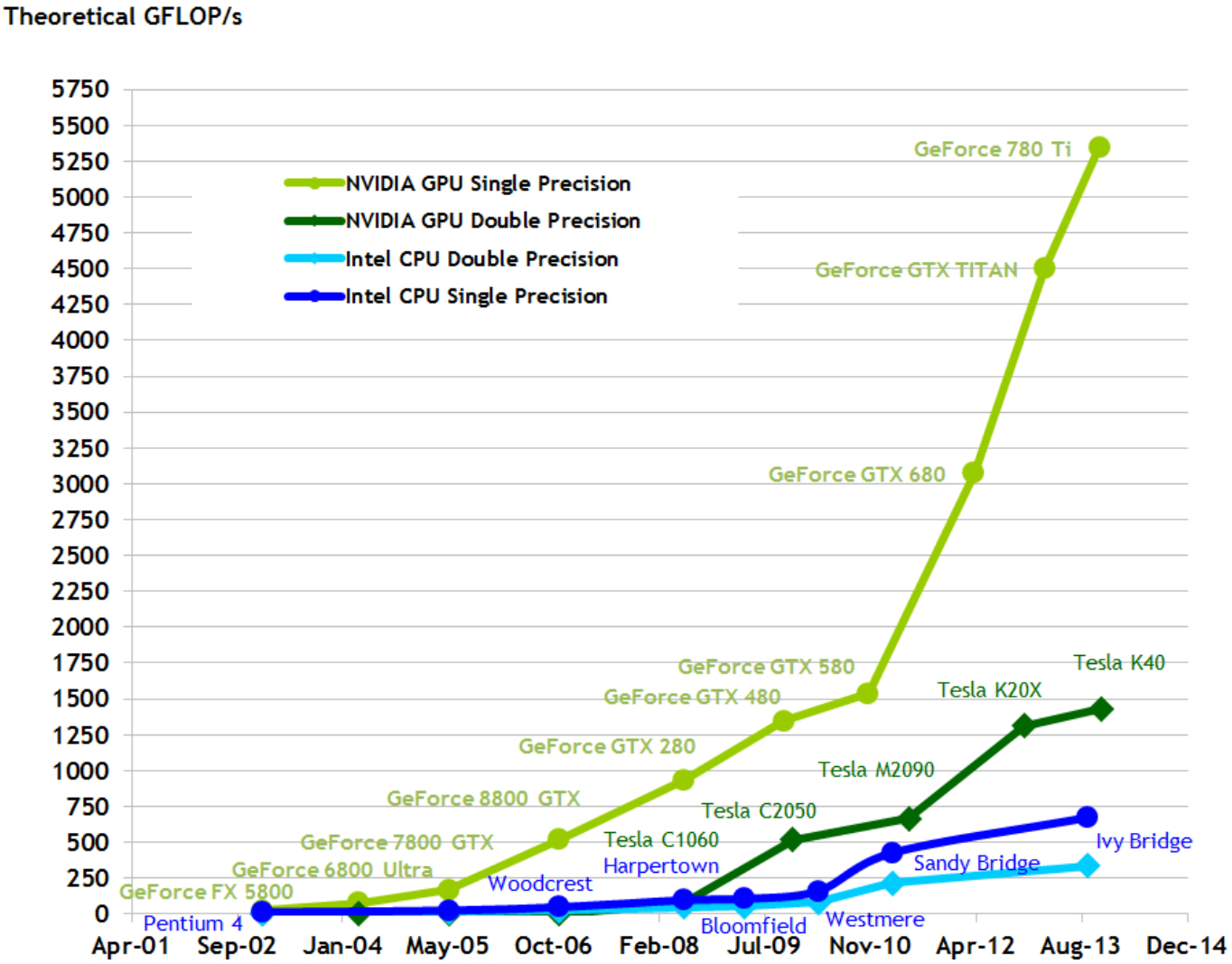
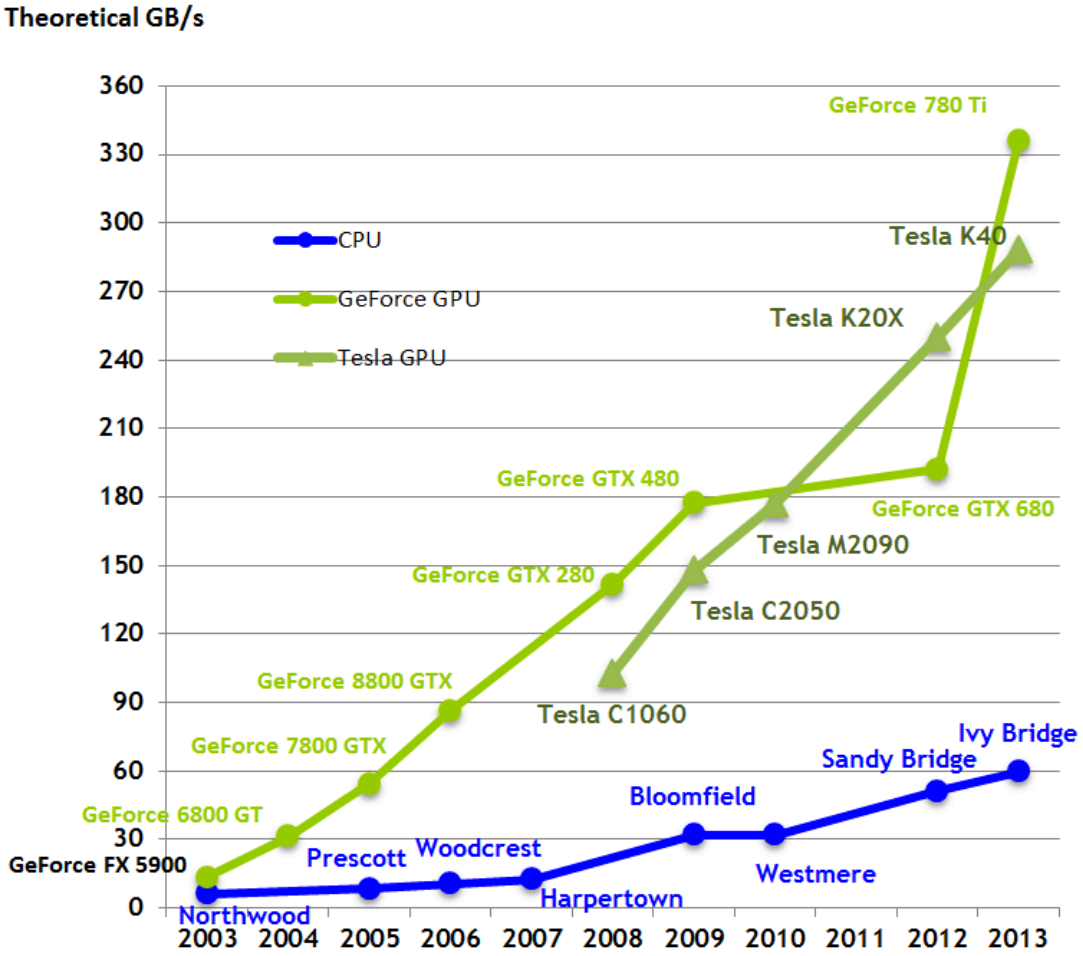


Image from http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#abstract

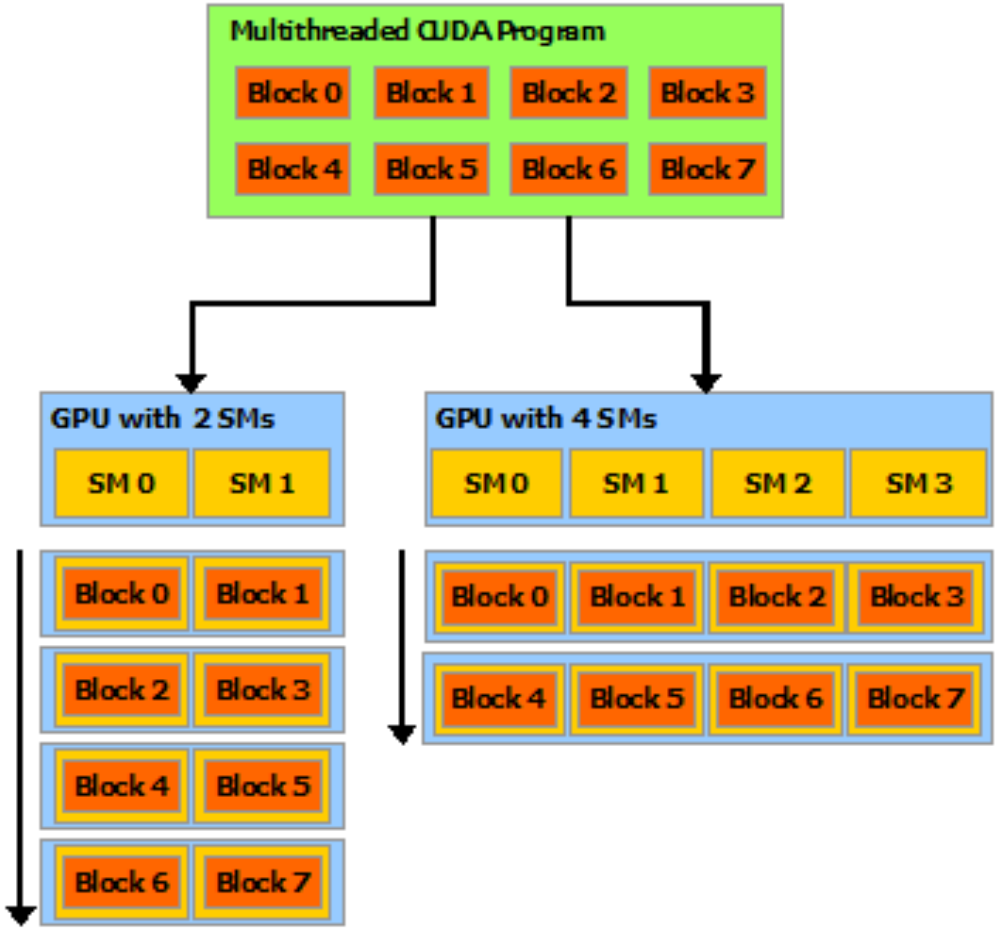
Bandwidth to memory



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)
  ...

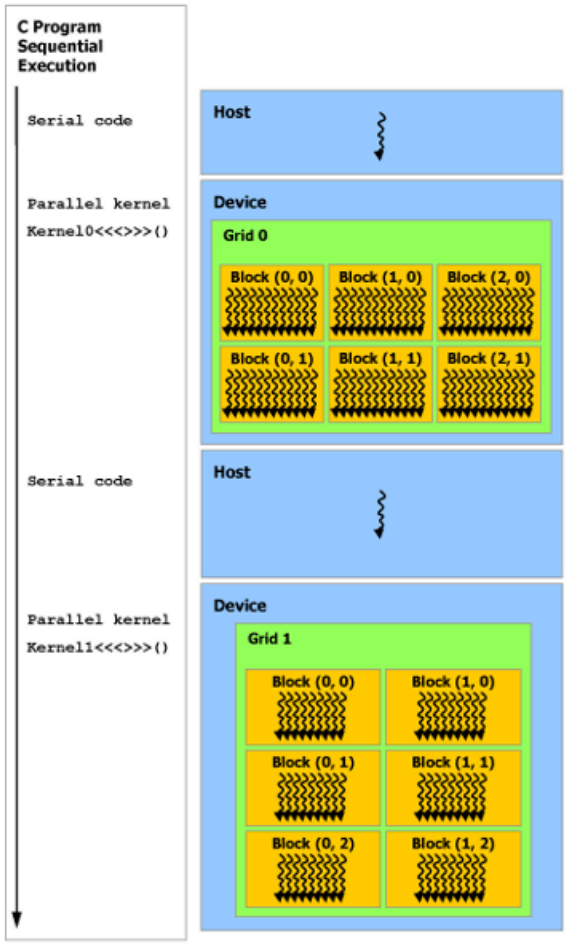


Image from http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#heterogeneous-programming

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

Sestoft mentioned a commercial F# to CUDA compiler (from QuantAlea)!

Loo.py is seriously cool!

Raising the level of abstraction

Functional

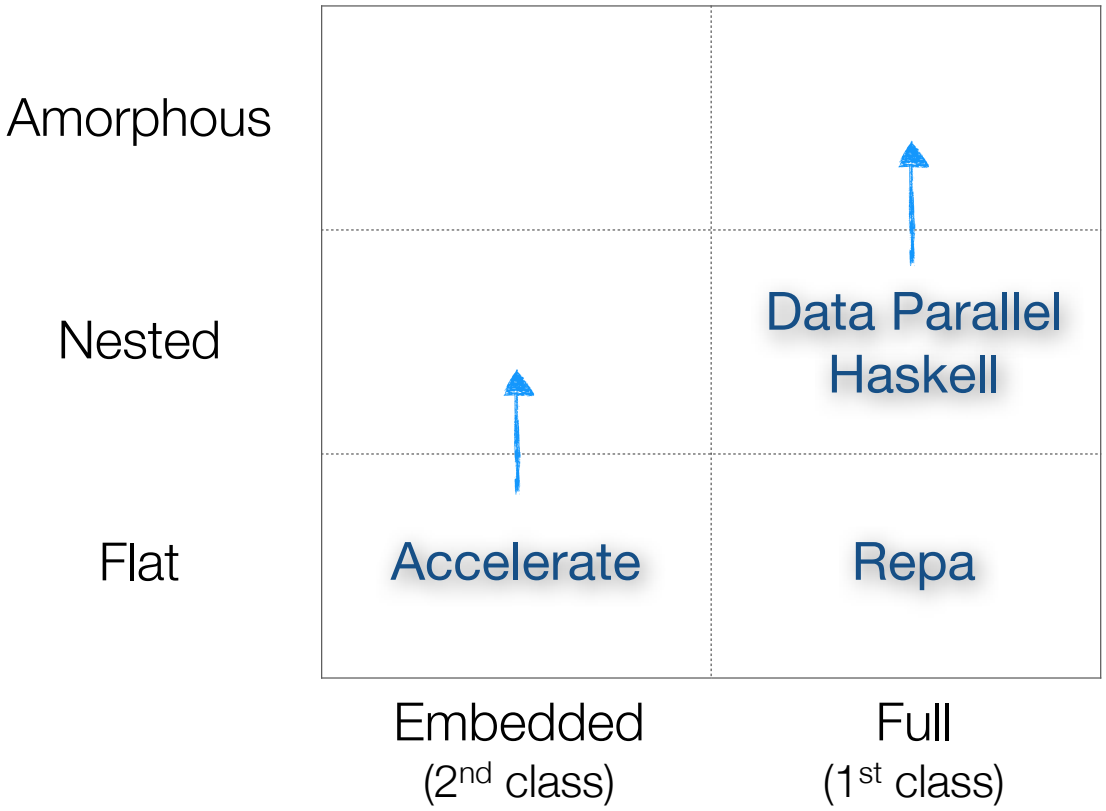
- Accelerate
- Obsidian

(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

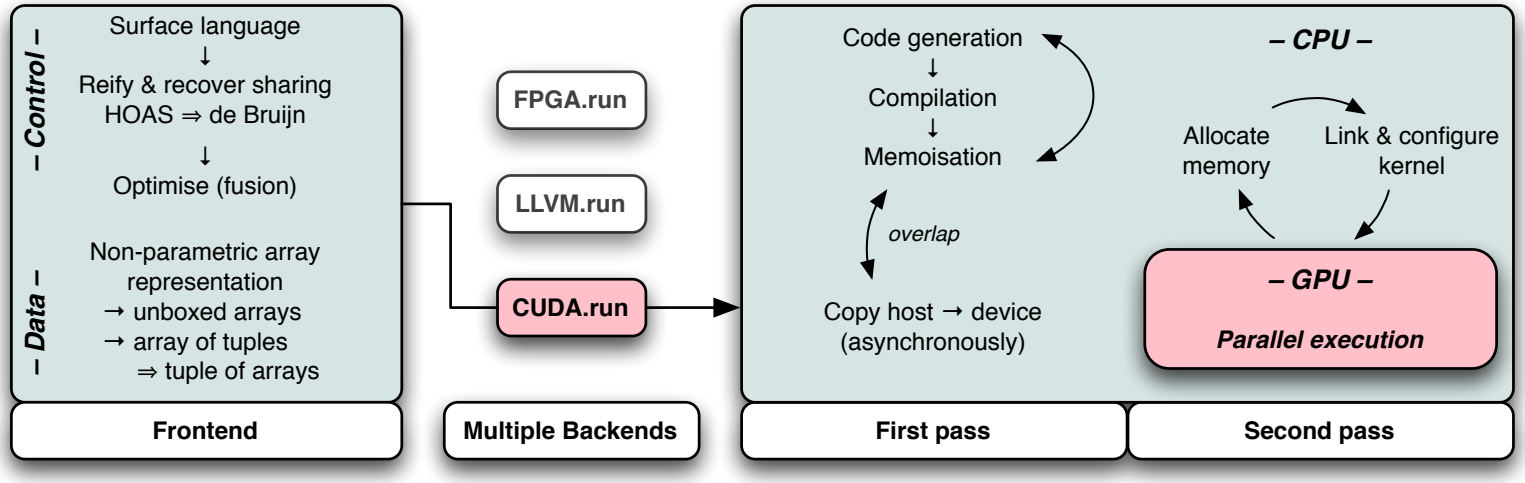


Figure 2. Overall structure of `Data.Array.Accelerate`.

(from the DAMP'11 paper)

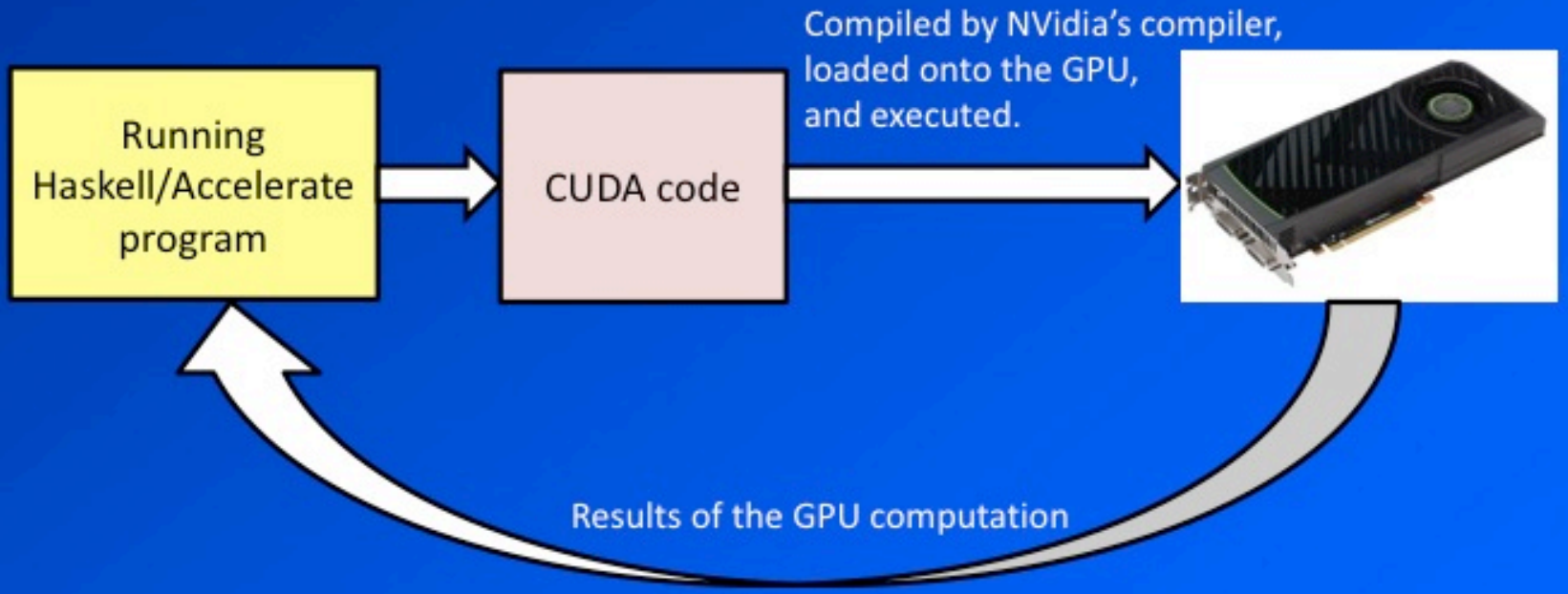
Accelerate back-ends

back-end	addresses	state
Interpreter	testing	works
CUDA	Nvidia graphic cards	works
CL	any graphic card through OpenCL	prototype
LLVM	any processor through LLVM	prototype
Repa	any processor in plain Haskell	stalled
FPGA	programmable hardware	fictional

Thielemann's view

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)

(see Lava)

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)
```

Assume an associative operator that is folded in a tree shape

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)
```

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

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]
```

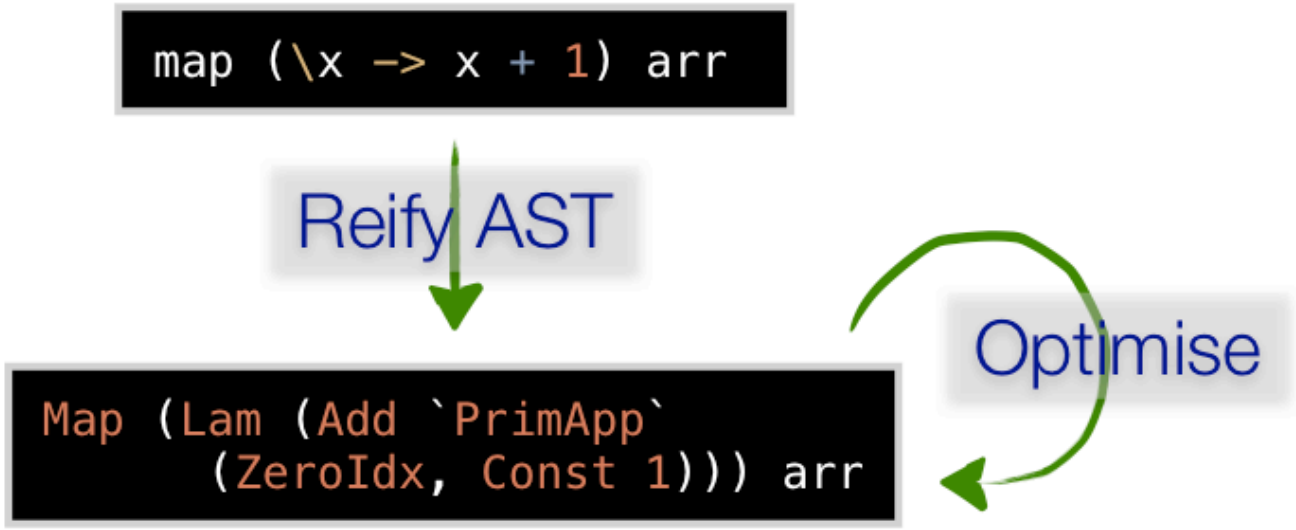
Second attempt much faster. Kernels are memoised.

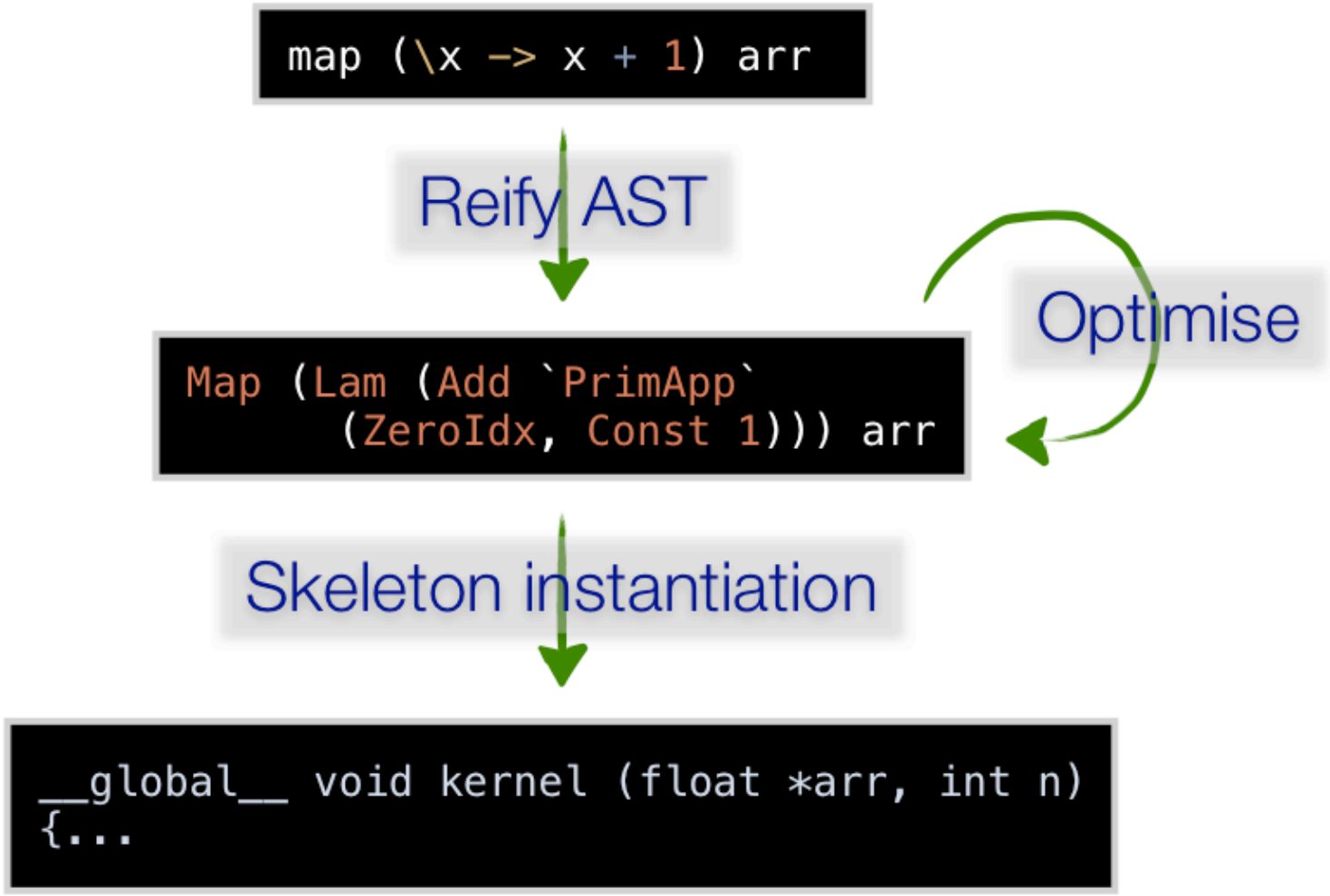
```
map (\x -> x + 1) arr
```

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


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

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

Reify AST

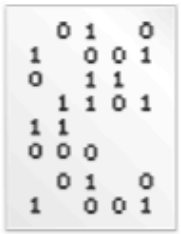
```
Map (Lam (Add `PrimApp`
      (ZeroIdx, Const 1))) arr
```

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)

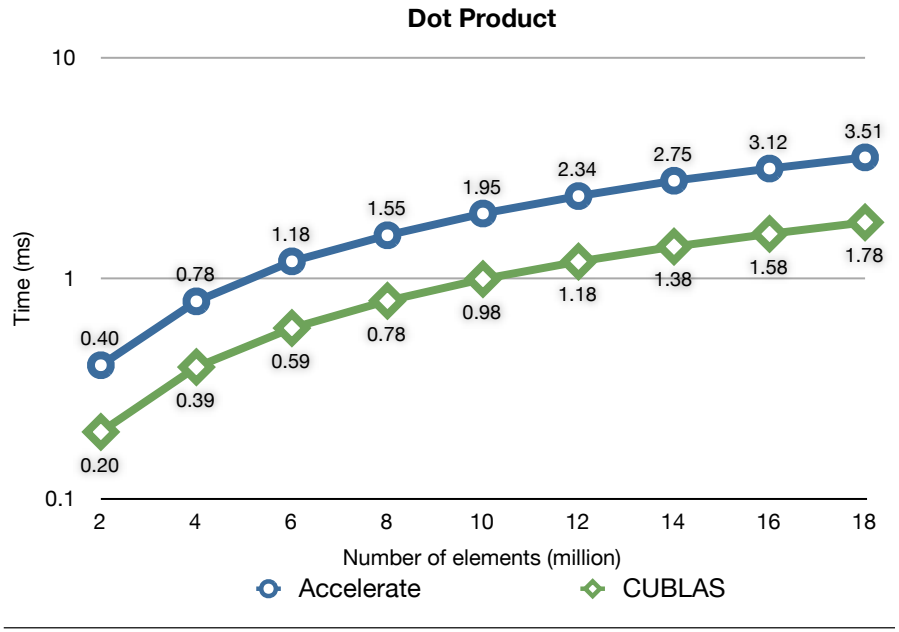


Figure 3. Kernel execution time for a dot product.

Performance (DAMP'11 paper)

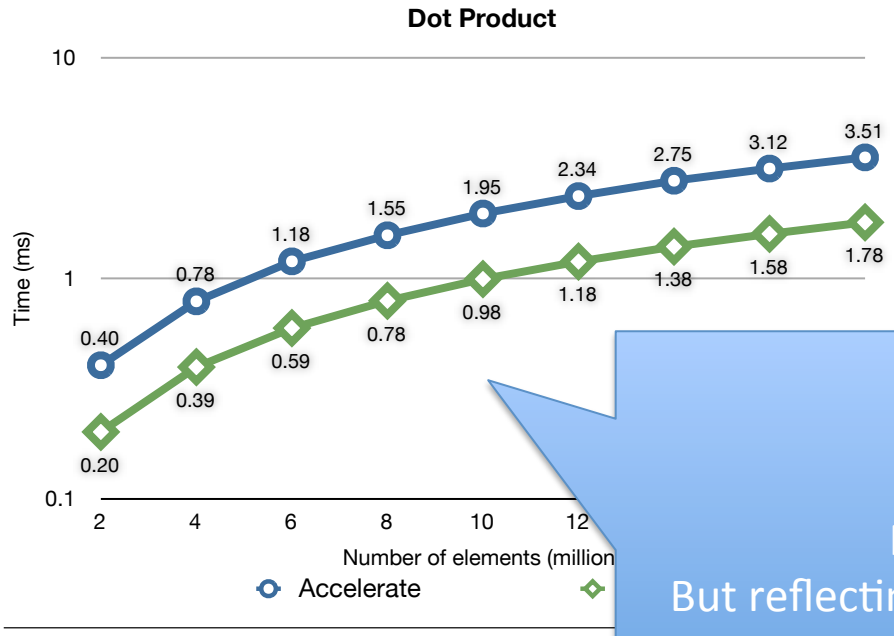


Figure 3. Kernel execution time for a

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.

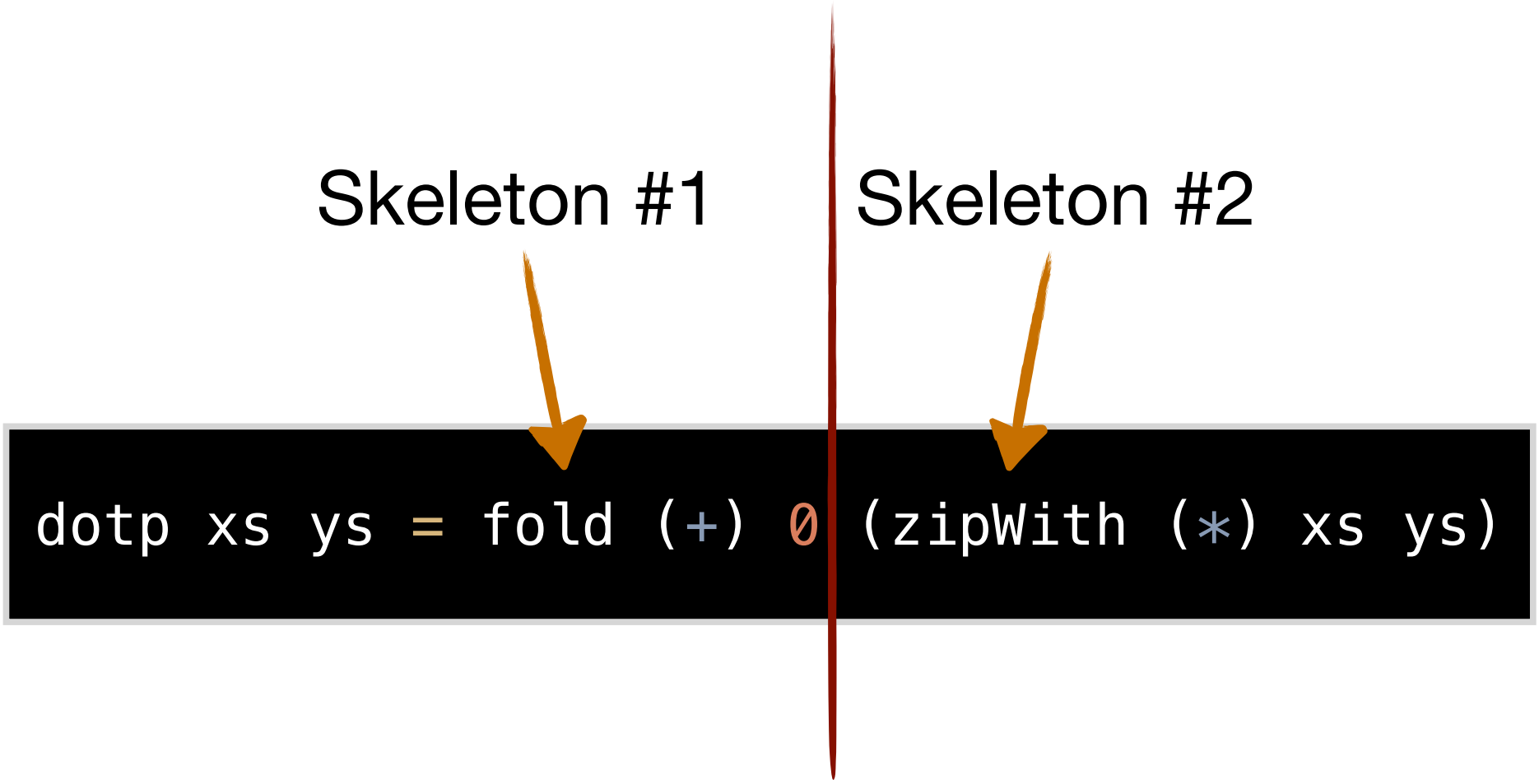
We recently presented *Accelerate*, an EDSL and skeleton-based code generator targeting the CUDA GPU development environment [8]. In the present paper, we present novel methods for optimising the code using *sharing recovery* and *array fusion*.

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

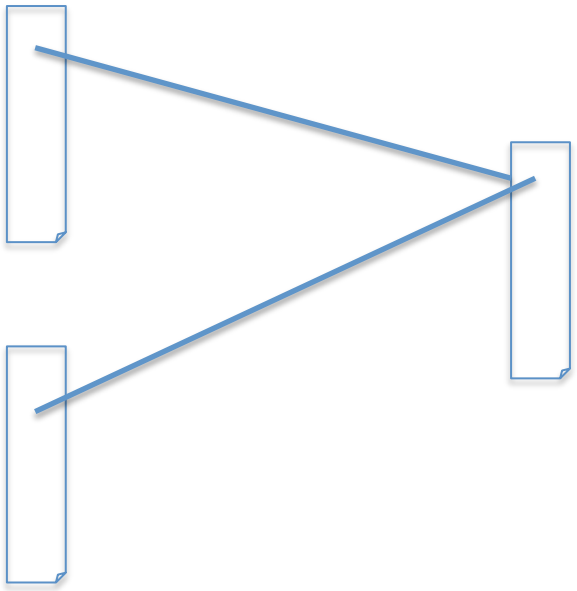
Extra traversal

Combined skeleton

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

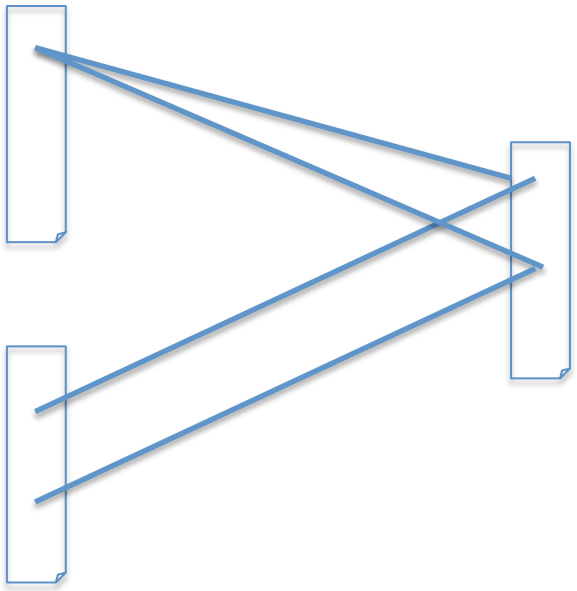
Producers

"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."



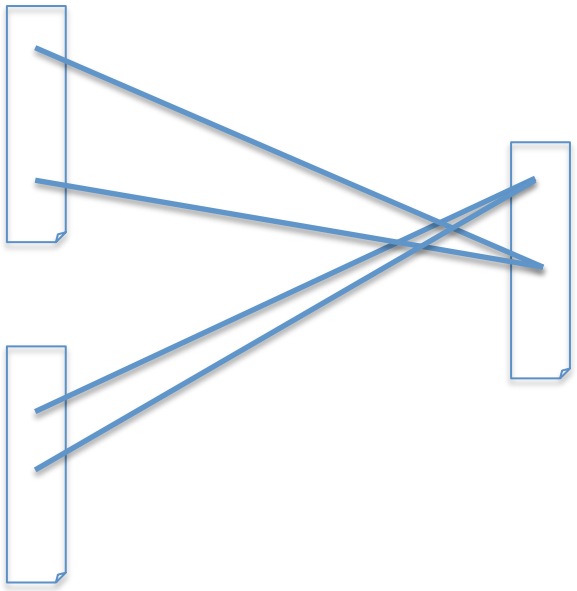
Producers

"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."



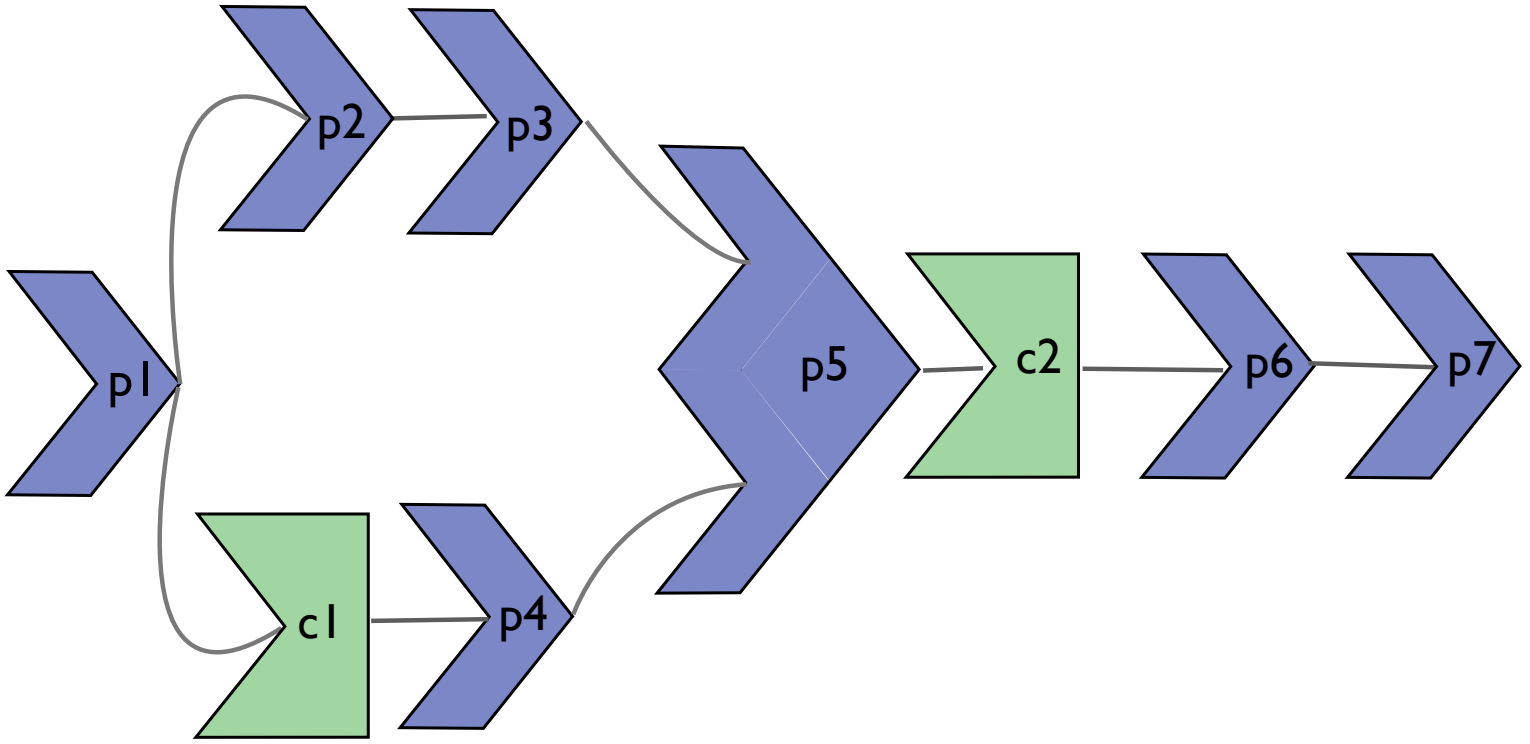
Producers

```
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)
            -> Acc (Array (SliceShape slix)
generate    :: Exp sh -> (Exp sh -> Exp a) ->


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

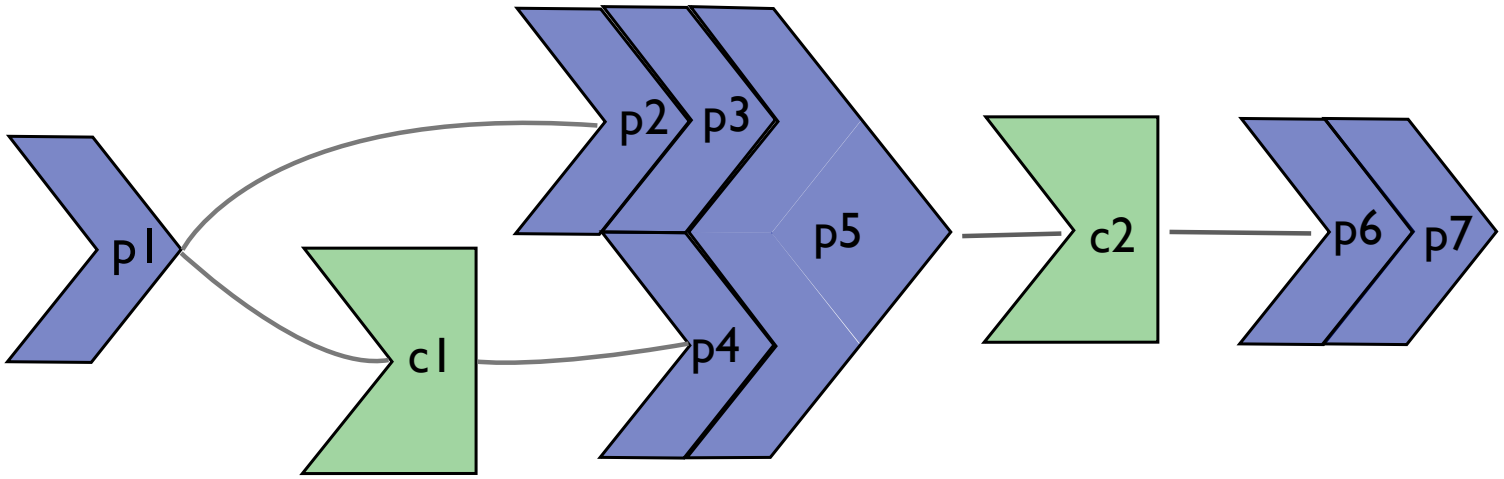
Note the NESL influence on programming idioms!!

Fusing networks of skeletons



Fusing networks of skeletons

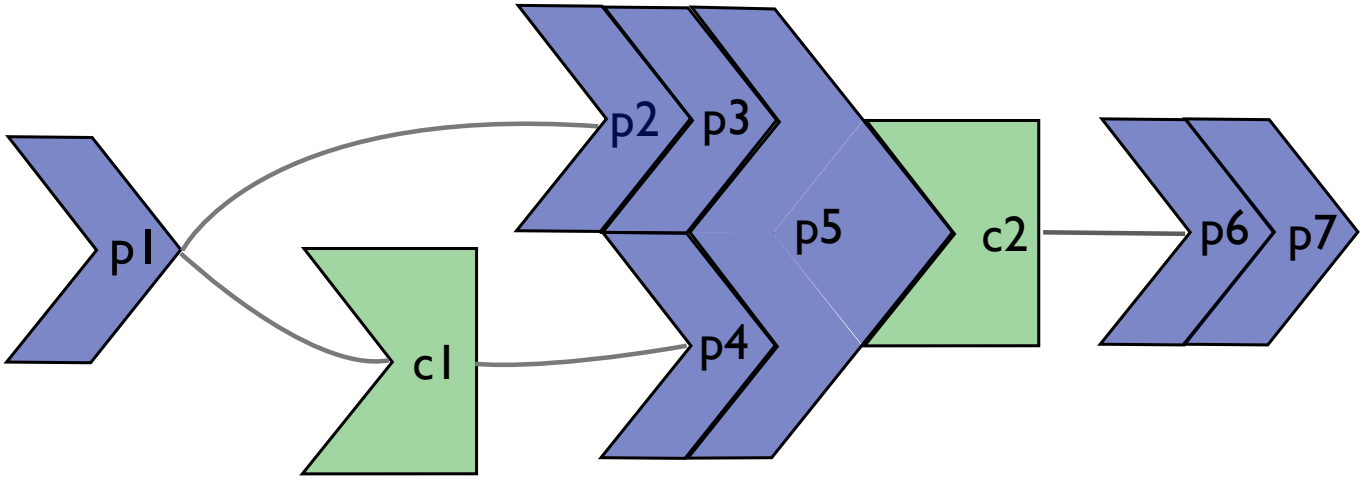
Phase 1: producer/producer fusion



This is the easy case

Fusing networks of skeletons

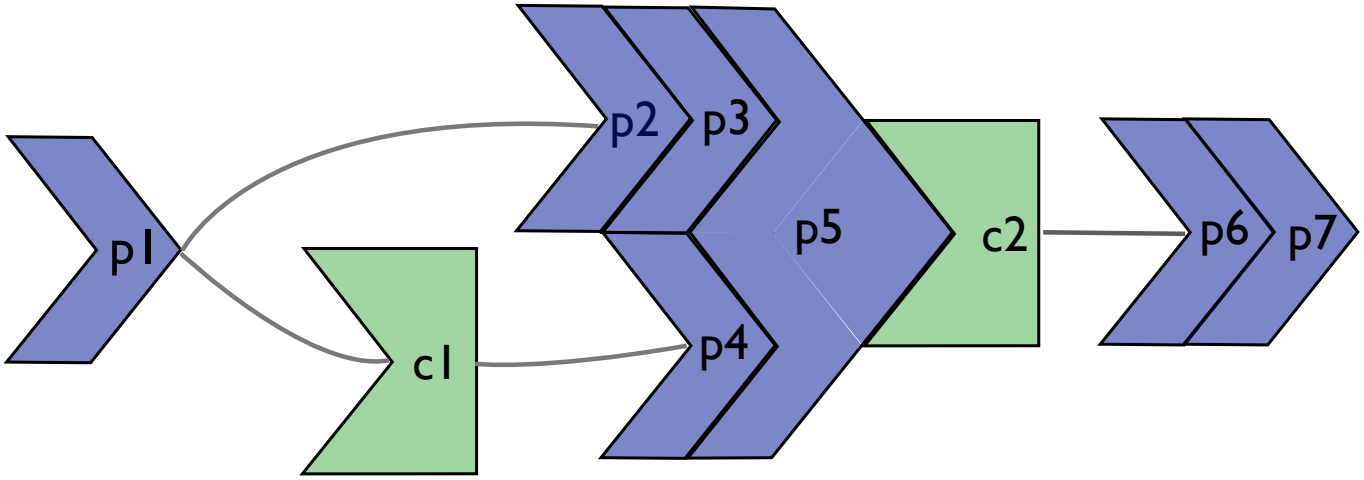
Phase 2: consumer/producer fusion



Fuse a producer followed by a consumer into the consumer
Happens during code generation. Speciailise 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 of skeletons

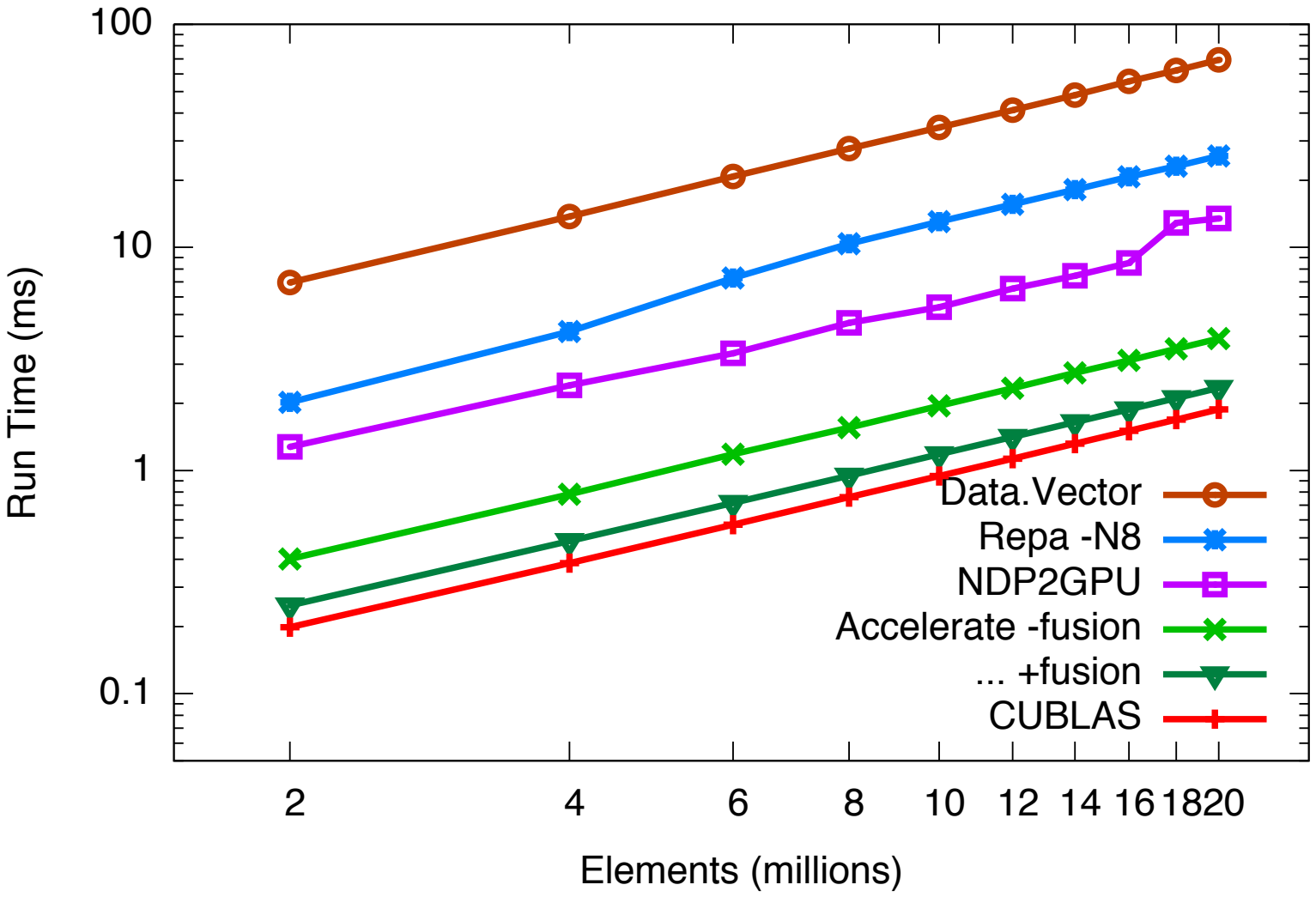
…reduces the abstraction penalty

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

Read Chap. 6 of Marlow book

Look at accelerate-examples

Break?

GPU programming in Obsidian

Ack: Obsidian is developed by Joel Svensson.

github.com/svenssonjoel/obsidian
checkout master-dev for latest version

Accelerate

Get acceleration from your GPU by writing familiar combinators

Hand tuned skeleton templates

Compiler cleverness to fuse and memoise the resulting kernels

Leaves a gap between the programmer and the GPU (which most people want)

Obsidian

Can we bring FP benefits to GPU programming, without giving up control of low level details?

This is an instance of the research questions in our big SSF project called Resource Aware Functional Programming

(You might have seen a lecture about Feldspar in some other course.)

Obsidian

- mid-level programming of CUDA, OpenCL and sequential C on CPU
- explicit control of parallelism arrangement in Threads, Thread blocks, Grid
- supports batched monadic/imperative programming

my applications:

- Cholesky decomposition for band-matrices:
  based on `mapAccum` (not available in Accelerate)
- pivot vector to permutation array conversion:
  requires mutable manipulation (not complete in Obsidian)
- call Obsidian code from Accelerate

Thielemann's view again

Assumptions

To get really good performance from a GPU, one must control

- use of memory
- memory access patterns
- synchronisation points
- where the boundaries of kernels are
- patterns of sequential code (control of task size)

Vital to be able to experiment with variants on a kernel easily

Assumptions

To get really good performance from a GPU, one must control

- use
- me
- wh
- pa

Vital t
kerne

We aim to give the programmer this control

We avoid compiler cleverness!

Cost model should be entirely transparent

Building blocks

Embedded DSL in Haskell

Pull and push arrays

Use of types to allow “hierarchy-polymorphic” functions (Thread, Warp, Block, Grid)

A form of virtualisation to remove arbitrary limits like max #threads per block

Memory layout is taken care of (statically)

Building blocks

Embedded DSL in Haskell

Pull and push arrays

Use
functio

A form
like ma

Delayed arrays
See Pan by Elliot http://conal.net/pan/
Or even
Compilation and Delayed Evaluation in APL, Guibas and Wyatt, POPL'78

Building blocks

Embedded DSL in Haskell

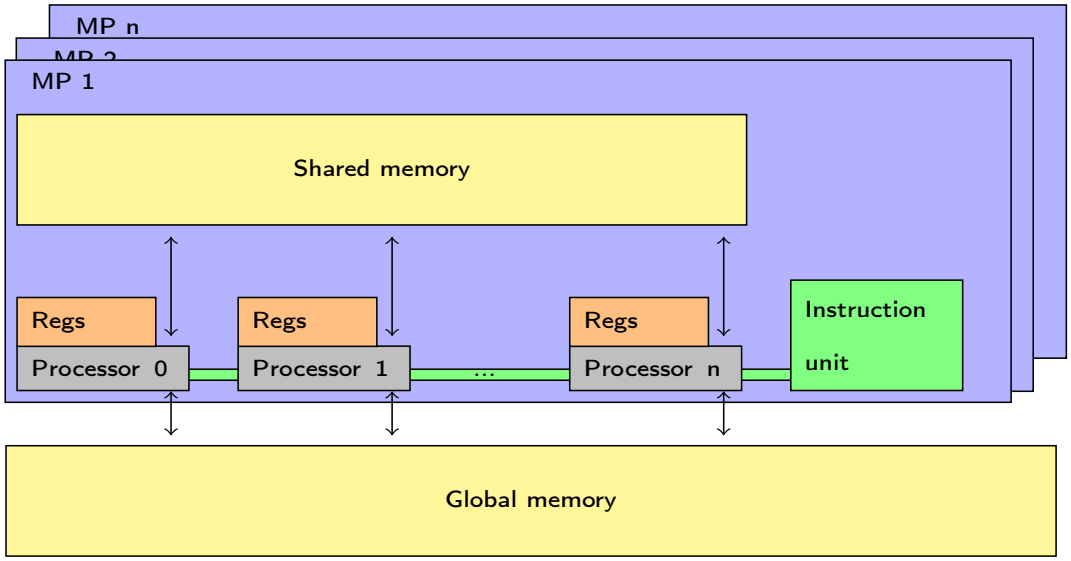
Pull and push arrays

Use of types
functions (Tl

A form of vi
like max #th

A new array representation due to Claessen
will come back to this

GPU


MP n
MP 1
Shared memory
Regs
Regs
Regs
Instruction
unit
Processor 0
Processor 1
...
Processor n
Global memory

CUDA programming model

Single Program Multiple Threads

Kernel = Function run N times by N threads

Hierarchical thread groups

Associated memory hierarchy

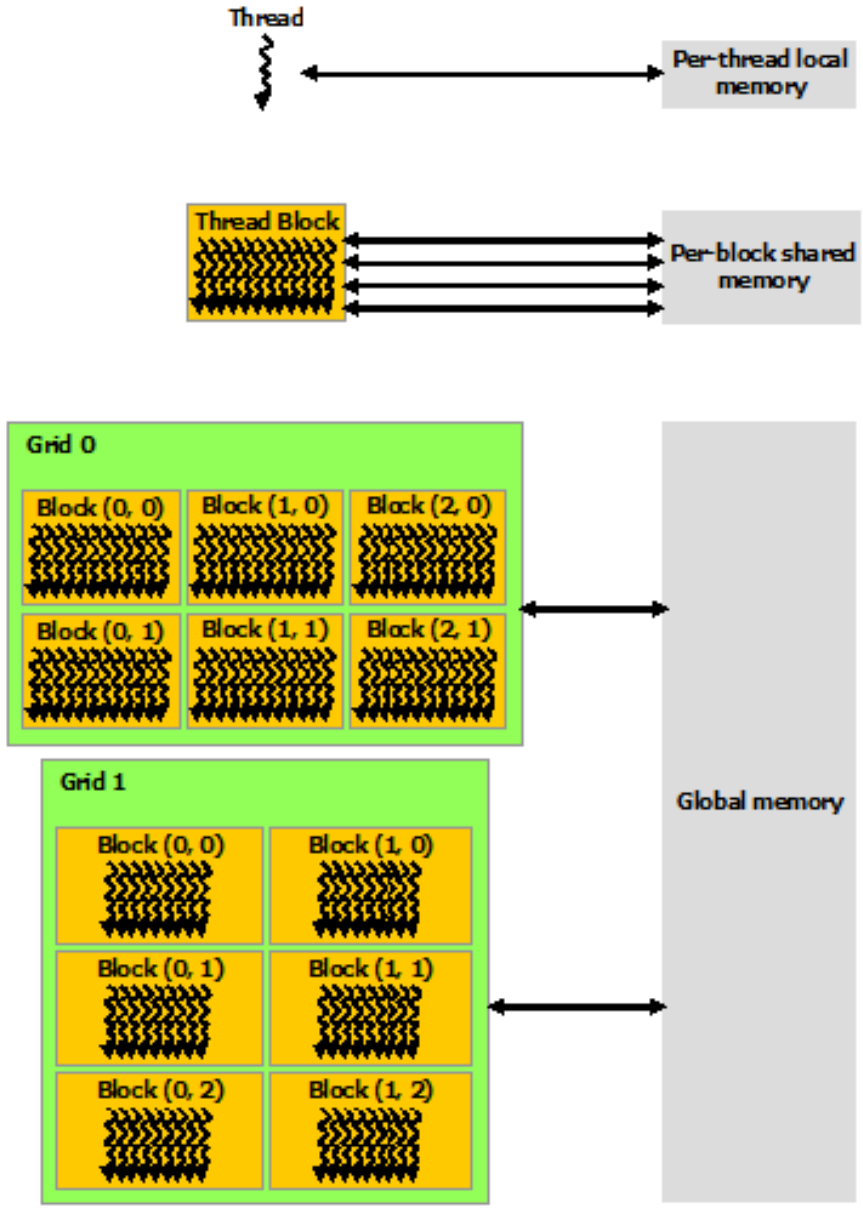


Image from http://docs.nvidia.com/cuda/cuda-c-programming-guide/#memory-hierarchy

The flow of kernel execution

Initialize/acquire the device (GPU)

Allocate memory on the device (GPU)

Copy data from host (CPU) to device (GPU)

Execute the kernel on the device (GPU)

Copy result from device (GPU) to host (CPU)

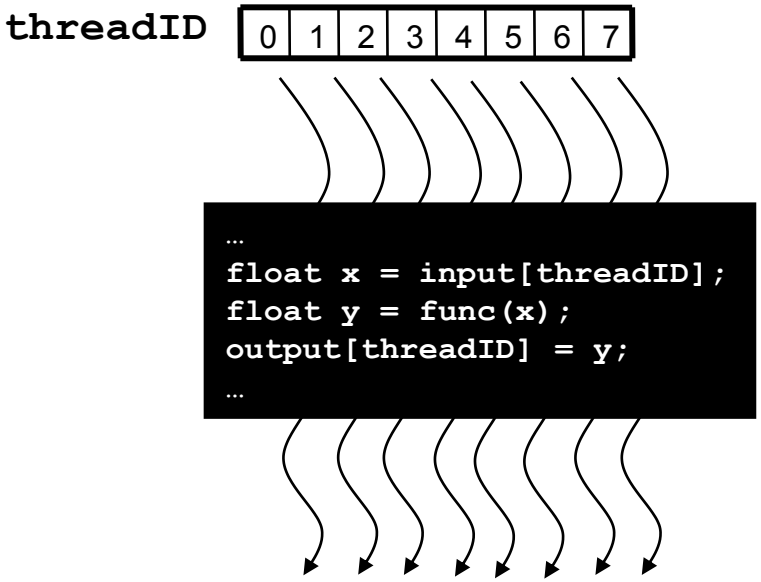
Deallocate memory on device (GPU)

Release device (GPU)

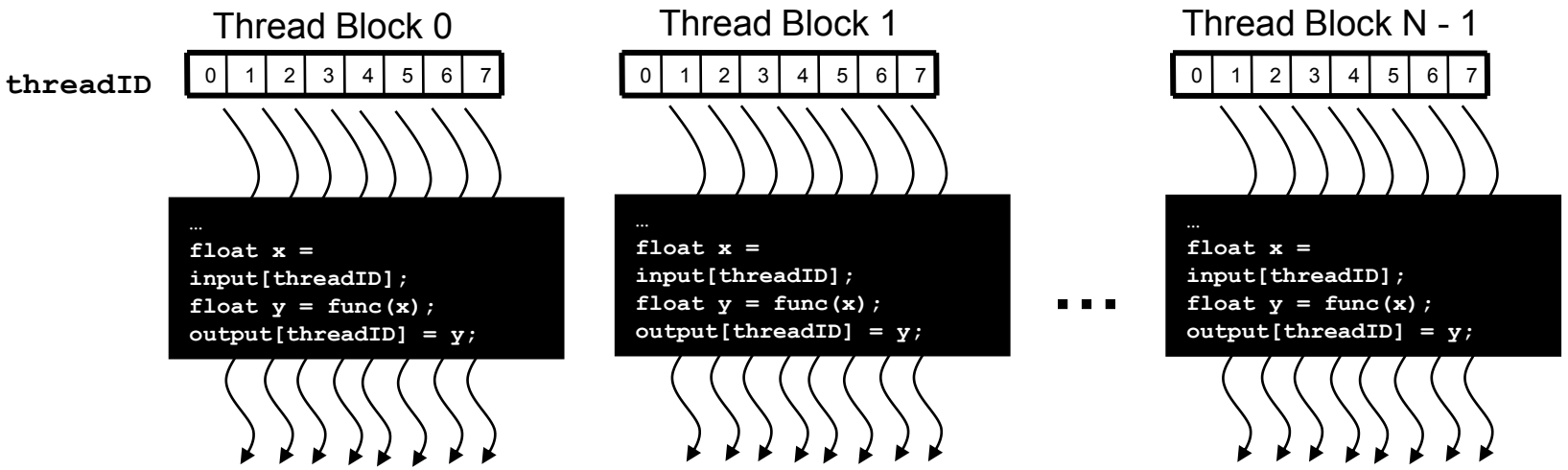
CUDA kernel

Executed by an array of Threads

Each thread has an ID that is used to compute memory addresses and make control decisions



Blocks



Threads within a block communicate via shared memory and barrier synchronisation (__ syncthreads();)

Threads in different blocks cannot cooperate

Hierarchy

Level	Parallelism	Shared Memory	Thread synchronisation
Thread	No	Yes	No
Warp	Yes	Yes	Lock-step execution
Block	Yes	Yes	Yes
Grid	Yes	No	No

Memory access patterns

Some patterns of global memory access can be coalesced. Others cannot. Missing out on coalescing ruins performance!

Global memory works best when adjacent threads access a contiguous block

For shared memory, successive 32 bit words are in different banks. Multiple simultaneous access to a bank = bank conflict = another way to ruin performance. Conflicting accesses are serialised.

Thread ID is usually built from

blockIdx	Block index within a grid	uint3
blockDim	Dimension of the block	dim3
threadIdx	Thread index within a block	uint3

gridDim gives the dimensions of the grid (the number of blocks in each dimension)

We'll use linear blocks and grids (easier to think about)

For more info about CUDA see https://developer.nvidia.com/gpu-computing-webinars
esp. the 2010 intro webinars

First CUDA kernel

```
__global__ void inc(float *i, float *r){
  unsigned int ix = blockIdx.x * blockDim.x + threadIdx.x;
  r[ix] = i[ix]+1;
}
```

Host code

```
#include <stdio.h>
#include <cuda.h>
#define BLOCK_SIZE 256
#define BLOCKS 1024
#define N (BLOCKS * BLOCK_SIZE)

int main(){
  float *v, *r;
  float *dv, *dr;

  v = (float*)malloc(N*sizeof(float));
  r = (float*)malloc(N*sizeof(float));

  //generate input data
  for (int i = 0; i < N; ++i) {
     v[i] = (float)(rand () % 1000) / 1000.0; }

  /* Continues on next slide */
```

Host code

```
  cudaMalloc((void**)&dv, sizeof(float) * N );
  cudaMalloc((void**)&dr, sizeof(float) * N );

  cudaMemcpy(dv, v, sizeof(float) * N,cudaMemcpyHostToDevice);

  inc<<<BLOCKS, BLOCK_SIZE,0>>>(dv,dr);

  cudaMemcpy(r, dr, sizeof(float) * N, cudaMemcpyDeviceToHost);

  cudaFree(dv);
  cudaFree(dr);

  for (int i = 0; i < N; ++i) {
      printf("%f ", r[i]); }
  printf("\n");

  free(v);
  free(r);
}
```

Obsidian

```
incLocal arr = fmap (+1) arr
```

Building an AST just like in Accelerate

Obsidian Pull arrays

```
incLocal :: Pull Word32 EWord32 -> Pull Word32 EWord32
incLocal arr = fmap (+1) arr
```

Pull size element-type

Static Word32 = Haskell value known at compile time
Dynamic EWord32 = Exp Word32 (an expression tree)

Immutable

Obsidian Pull arrays

```
data Pull s a = Pull {pullLen :: s,
                      pullFun :: EWord32 -> a}
```

(length and function from index to value, the *read-function*, see Elliott's Pan, also called delayed arrays)

```
type SPull = Pull Word32
type DPull = Pull EWord32
```

A consumer of a pull array needs to iterate over those indices of the array it is interested in and apply the pull array function at each of them.

Fusion for free

```
fmap f (Pull n ixf) = Pull n (f . ixf)
```

Example

```
incLocal arr = fmap (+1) arr
```

This says what the computation should do

How do we lay it out on the GPU??

```
incPar :: Pull EWord32 EWord32 -> Push Block EWord32 EWord32
incPar = push . incLocal
```

push converts a pull array to a push array and pins it to a particular part of the GPU hierarchy

No cost associated with pull to push conv.

Key to getting fine control over generated code

GPU Hierarchy in types

```
data Thread
data Step t

type Warp  = Step Thread
type Block = Step Warp
type Grid  = Step Block
```

GPU Hierarchy in types

```
-- | Type level less-than-or-equal test.
type family LessThanOrEqual a b where
   LessThanOrEqual Thread   Thread   = True
   LessThanOrEqual Thread   (Step m) = True
   LessThanOrEqual (Step n) (Step m) = LessThanOrEqual n m
   LessThanOrEqual x y               = False


type a *<=* b = (LessThanOrEqual a b ~ True)
```

Program data type

```
data Program t a where

  Identifier :: Program t Identifier

  Assign :: Scalar a
            => Name
            -> [Exp Word32]
            -> (Exp a)
            -> Program Thread ()

. . .

  -- use threads along one level
  -- Thread, Warp, Block.
ForAll :: (t *<=* Block) => EWord32
            -> (EWord32 -> Program Thread ())
            -> Program t ()

. . .
```

Program data type

```
seqFor :: EWord32 -> (EWord32 -> Program t ()) -> Program t ()

. . .

Sync     :: (t *<=* Block) => Program t ()

. . .
```

Program data type

```
. . .


Return :: a -> Program t a
Bind   :: Program t a -> (a -> Program t b) -> Program t b
```

```
instance Monad (Program t) where
  return = Return
  (>>=) = Bind
```

See

Svenningsson, Josef, & Svensson, Bo Joel. (2013). Simple and Compositional Reification of Monadic Embedded Languages. ICFP 2013.

Obsidian push arrays

```
data Push t s a = Push s (PushFun t a)
```

The general idea of push arrays is due to Koen Claessen

Obsidian push arrays

```
-- | Push array. Parameterised over Program type and size type.

data Push t s a = Push s (PushFun t a)

type PushFun t a = Writer a -> Program t ()
```

Push array only allows bulk request to push ALL elements via a writer function

The general idea of push arrays is due to Koen Claessen

Obsidian push arrays

```
-- | Push array. Parameterised over Program type and size type.

data Push t s a = Push s (PushFun t a)

type PushFun t a = Writer a -> Program t ()
type Writer    a = a -> EWord32 -> TProgram ()
```

consumer of a push array needs to apply the push-function to a suitable writer

Often the push-function is applied to a writer that stores its input value at the provided input index into memory. This is what the compute function does when applied to a push array.

The general idea of push arrays is due to Koen Claessen

Obsidian push arrays

The function push converts a pull array to a push array:

```
push ::  (t *<=* Block) => ASize s => Pull s e -> Push t s e
push (Pull n ixf) =
  mkPush n $ \wf ->
    forAll (sizeConv n) $ \i -> wf (ixf i) i
```

Obsidian push arrays

The function push converts a pull array to a push array:

```
push ::  (t *<=* Block) => ASize s => Pull s e -> Push t s e
push (Pull n ixf) =
  mkPush n $ \wf ->
    forAll (sizeConv n) $ \i -> wf (ixf i) i
```

This function sets up an iteration schema over the elements as a forAll loop. It is not until the t parameter is fixed in the hierarchy that it is decided exactly how that loop is to be executed. All iterations of the forAll loop are independent, so it is open for computation in series or in parallel.

```
forAll :: (t *<=* Block) => EWord32
          -> (EWord32 -> Program Thread ())
          -> Program t ()
forAll n f = ForAll n f
```

```
forAll :: (t *<=* Block) => EWord32
          -> (EWord32 -> Program Thread ())
          -> Program t ()
forAll n f = ForAll n f
```

Type says that forAll can't be applied at the Grid level (because that would involve dreaming up #blocks and #threads per block)

```
forAll :: (t *<=* Block) => EWord32
          -> (EWord32 -> Program Thread ())
          -> Program t ()
forAll n f = ForAll n f
```

ForAll iterates a body (described by higher order abstract syntax) a given number of times over the resources at level t

iterations independent of each other

```
forAll :: (t *<=* Block) => EWord32
          -> (EWord32 -> Program Thread ())
          -> Program t ()
forAll n f = ForAll n f
```

ForAll iterates a body (described by higher order abstract syntax) a given number of times over the resources at level t

iterations independent of each other

t = Thread => sequential

T = Warp, Block => parallel

Obsidian push array

A push array is a length and a filler function

Filler function encodes a loop at level t in the hierarchy

Its argument is a writer function

Push array allows only a bulk request to push all elements via a writer function

When invoked, the filler function creates the loop structure, but it
inlines the code for the writer inside the loop.

A push array with elements computed by f and writer wf corresponds to a loop
for (i in [1,N]) {wf(i,f(i));}

When forced to memory, each invocation of wf would write one memory location
A[i] = f(i)

Push and pull arrays

Neither pull nor push arrays are manifest

Both fuse by default.

Both immutable.

Don’t appear in Expression or Program datatypes
Shallow Embedding

See
Svenningsson and Axelsson on combining deep and shallow embeddings

Argh. Why two types of array??

Concatenation of pull arrays is inefficient.
Introduces conditionals (which can ruin performance)

Concatenation of Push arrays is efficient.
No conditionals.

splitting arrays up and using parts of them is easy using pull arrays.

Push and Pull arrays seem to have strengths and weaknesses that complement each other.

Pull good for reading. Push good for writing. Pull -> Push functions common

Back to example

```
incGrid1 :: Word32 -> DPull EWord32 -> DPush Grid EWord32
incGrid1 n arr = asGridMap (push . fmap (+1))  (splitUp n arr)
```

```
perform :: IO ()
perform =
  withCUDA $ do
    kern <- capture 512 (incGrid1 512)

    useVector (V.fromList [0..1023 :: Word32]) $ \ i ->
      withVector 1024 $ \ o ->
      do o <== (1,kern) <> i
         r <-  peekCUDAVector o
         lift $ putStrLn $ show r
```

```
perform :: IO ()
perform =
  withCUDA $ do
    kern <- capture 512 (incGrid1 512)

    useVector (V.fromList [0..1023 :: Word32]) $ \ i ->
      withVector 1024 $ \ o ->
      do o <== (1,kern) <> i
         r <-  peekCUDAVector o
         lift $ putStrLn $ show r
```

```
perform :: IO ()
perform =
  withCUDA $ do
    kern <- capture 512 (incGrid1 512)

    useVector (V.fromList [0..1023 :: Word32]) $ \ i ->
      withVector 1024 $ \ o ->
      do o <== (1,kern) <> i
         r <-  peekCUDAVector o
         lift $ putStrLn $ show r

*Reduction> perform
[1,2,3,4,5,6,7 …
```

gen0.cu

```
#include <stdint.h>
extern "C" __global__ void gen0(uint32_t* input0, uint32_t n0,
                                uint32_t* output1)
{
    uint32_t bid = blockIdx.x;
    uint32_t tid = threadIdx.x;

    for (int b = 0; b < n0 / 512U / gridDim.x; ++b) {
        bid = blockIdx.x * (n0 / 512U / gridDim.x) + b;
        output1[bid * 512U + tid] = input0[bid * 512U + tid] + 1U;
        bid = blockIdx.x;
        __syncthreads();
    }
    bid = gridDim.x * (n0 / 512U / gridDim.x) + blockIdx.x;
    if (blockIdx.x < n0 / 512U % gridDim.x) {
        output1[bid * 512U + tid] = input0[bid * 512U + tid] + 1U;
    }
    bid = blockIdx.x;
    __syncthreads();
}
```

gen0.cu

```
#include <stdint.h>
extern "C" __global__ void gen0(uint32_t* input0, uint32_t n0,
                                uint32_t* output1)
{
    uint32_t bid = blockIdx.x;
    uint32_t tid = threadIdx.x;

    for (int b = 0; b < n0 / 512U / gridDim.x; ++b) {
        bid = blockIdx.x * (n0 / 512U / gridDim.x) + b;
        output1[bid * 512U + tid] = input0[bid * 512U + tid] + 1U;
        bid = blockIdx.x;
        __syncthreads();
    }
    bid = gridDim.x
    if (blockIdx.x < n0
        output1[bid * 512U
    }
    bid = blockIdx.x;
    __syncthreads();
}
```

Will go around the first loop twice
(an example of block virtualisation)

And zero times through the second loop

```
withCUDA $ do
    kern <- capture 128 (incGrid1 512)

    useVector (V.fromList [0..1023 :: Word32]) $ \ i ->
      withVector 1024 $ \ o ->
      do o <== (1,kern) <> i
         r <-  peekCUDAVector o
         lift $ putStrLn $ show r
```

```
#include <stdint.h>
extern "C" __global__ void gen0(uint32_t* input0, uint32_t n0,
                                uint32_t* output1)
{
    uint32_t bid = blockIdx.x;
    uint32_t tid = threadIdx.x;

    for (int b = 0; b < n0 / 512U / gridDim.x; ++b) {
        bid = blockIdx.x * (n0 / 512U / gridDim.x) + b;
        for (int i = 0; i < 4; ++i) {
            tid = i * 128 + threadIdx.x;
            output1[bid * 512U + tid] = input0[bid * 512U + tid] + 1U;
        }
        tid = threadIdx.x;
        bid = blockIdx.x;
        __syncthreads();
    }
    bid = gridDim.x * (n0 / 512U / gridDim.x) + blockIdx.x;
    if (blockIdx.x < n0 / 512U % gridDim.x) {
        for (int i = 0; i < 4; ++i) {
            tid = i * 128 + threadIdx.x;
            output1[bid * 512U + tid] = input0[bid * 512U + tid] + 1U;
        }
        tid = threadIdx.x;
    }
    bid = blockIdx.x;
    __syncthreads();
}
```

compute instead of push

```
#include <stdint.h>
extern "C" __global__ void gen0(uint32_t* input0, uint32_t n0,
                        uint32_t* output1)
{
  __shared__ uint8_t sbase[2048U];
  uint32_t bid = blockIdx.x;
  uint32_t tid = threadIdx.x;
  uint32_t* arr0 = (uint32_t*) (sbase + 0);

  for (int b = 0; b < n0 / 512U / gridDim.x; ++b) {
    bid = blockIdx.x * (n0 / 512U / gridDim.x) + b;
    arr0[tid] = input0[bid * 512U + tid] + 1U;
    __syncthreads();
    output1[bid * 512U + tid] = arr0[tid];
    bid = blockIdx.x;
    __syncthreads();
  }
  bid = gridDim.x * (n0 / 512U / gridDim.x) + blockIdx.x;
  if (blockIdx.x < n0 / 512U % gridDim.x) {
    arr0[tid] = input0[bid * 512U + tid] + 1U;
    __syncthreads();
    output1[bid * 512U + tid] = arr0[tid];
  }
  bid = blockIdx.x;
  __syncthreads();
}
```

Doesn't make sense in this kernel but does in multistage (ie most) kernels
Point is to have control of memory use

Reduction

```
-- generic parallel or sequential reduction
reduce :: (Compute t, Data a)
       => (a -> a -> a)
       -> SPull a
       -> Program t (SPush t a)
reduce f arr
  | len arr == 1 = return $ push arr
  | otherwise    =
    do let (a1,a2) = halve arr
       arr' <- compute $ push $ zipWith f a1 a2
       reduce f arr'
```

Reduction

```
-- generic parallel or sequ
reduce :: (Compute t, Data
       => (a -> a -> a)
       -> SPull a
       -> Program t (SPush
reduce f arr
  | len arr == 1 = return $ pus
  | otherwise    =
    do let (a1,a2) = halve arr
       arr' <- compute $ push $ zipWith f a1 a2
       reduce f arr'
```

fine for a commutative operator

```
reduce2stage :: Data a
             => Word32
             -> (a -> a -> a)
             -> SPull a -> Program Block (SPush Block a)
reduce2stage m f arr = do
  arr' <- compute $ asBlock (fmap body (splitUp m arr))
  reduce f arr'
  where body a = execWarp (reduce f a)

reduceGrid :: Data a
           => Word32
           -> Word32
           -> (a -> a -> a)
           -> DPull a -> DPush Grid a
reduceGrid m n f arr = asGrid $ fmap body (splitUp m arr)
    where
      body a = execBlock (reduce2stage n f a)
```

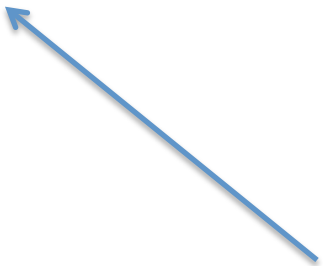
```
coalesce :: ASize l
         => Word32 -> Pull l a -> Pull l (Pull Word32 a)
coalesce n arr =
  mkPull s $ \i ->
    mkPull n $ \j -> arr ! (i + (sizeConv s) * j)
  where s = len arr `div` fromIntegral n
```

Access data by splitting up but also permuting the array (to give good memory access pattern)

```
red3 :: Data a
     => Word32
     -> (a -> a -> a)
     -> Pull Word32 a
     -> Program Block (SPush Block a)
red3 cutoff f  arr
  | len arr == cutoff =
    return $ push $ fold1 f arr
  | otherwise =
    do
      let (a1,a2) = halve arr
      arr' <- compute (zipWith f a1 a2)
      red3 cutoff f arr'
```

```
red5' :: Data a
           => Word32
           -> (a -> a -> a)
           -> Pull Word32 a
           -> Program Block (SPush Block a)
red5' n f arr =
  do arr' <- compute $ asBlockMap (execThread' . seqReduce f)
                                  (coalesce n arr)
     red3 2 f arr'
```

```
red5' :: Data a
           => Word32
           -> (a -> a -> a)
           -> Pull Word32 a
           -> Program Block (SPush Block a)
red5' n f arr =
  do arr' <- compute $ asBlockMap (execThread' . seqReduce f)
                                  (coalesce n arr)
     red3 2 f arr'
```



Reuse!!

A lot of index manipulation tedium is relieved!

Autotuning springs to mind!!

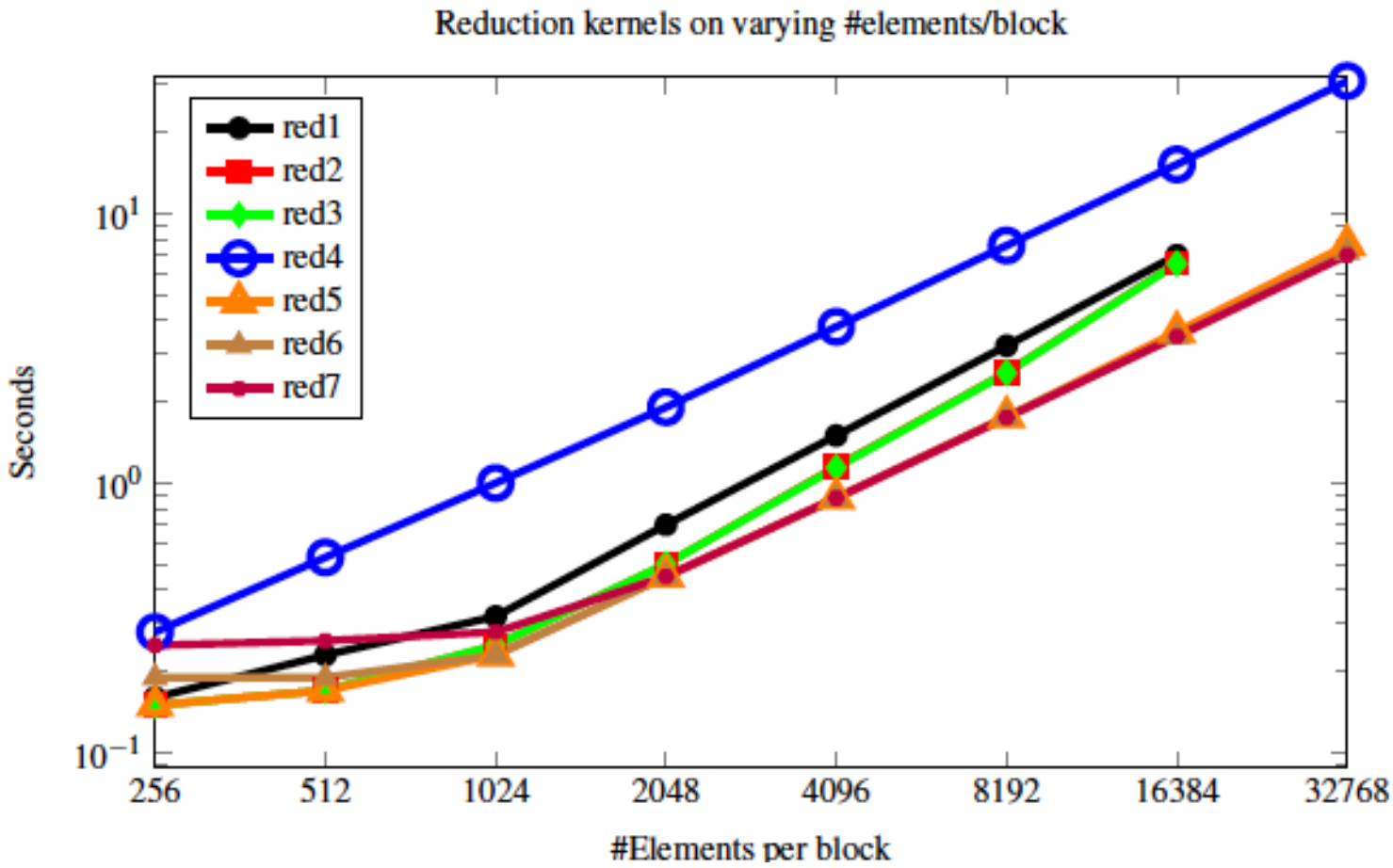

Reduction kernels on varying #elements/block
red1
red2
red3
red4
red5
red6
red7
Seconds
10^1
10^0
10^{-1}
256
512
1024
2048
4096
8192
16384
32768
#Elements per block

Fig. 11. The **threads-per-block** setting that achieved the best time shown in Figure 10. These settings are difficult to predict in advance. Kernels that use virtualized threads are highlighted, note that there are many of these amongst the best selection. Again, **elements-per-block** varies over the X axis.

Kernel	256	512	1024	2048	4096	8192	16384	32768
red1	**64**	**128**	**128**	**256**	**256**	**512**	**512**	n/a
red2	**64**	**128**	**64**	**128**	**256**	**512**	**512**	n/a
red3	**64**	**128**	**64**	**128**	**256**	**512**	**512**	n/a
red4	64	64	128	**64**	**64**	**64**	**128**	**512**
red5	32	64	64	**64**	**128**	**256**	**256**	**512**
red6	32	32	64	64	**128**	**128**	**256**	**256**
red7	32	32	32	64	128	**128**	512	**128**

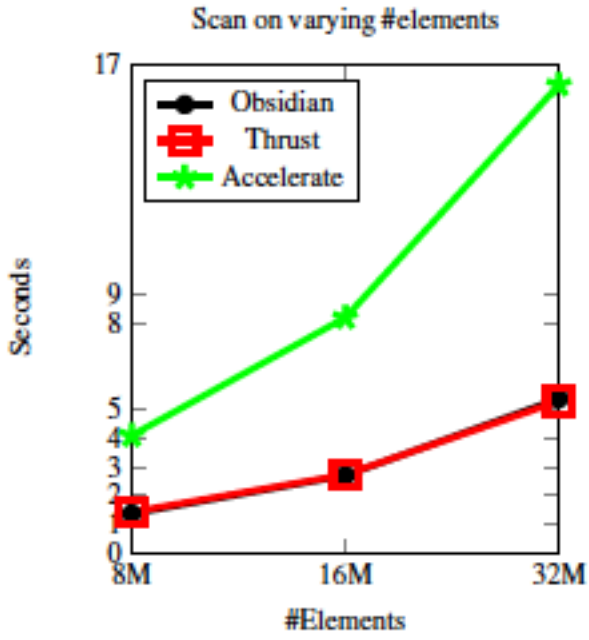


Fig. 18. The running time of scan algorithms for larger data sizes. The time reported is the sum of 1000 executions, excluding data transfer to and from the GPU memory. These number are collected on an NVIDIA GTX680. The presented Accelerate numbers are estimates based on a lower number of iterations as explained in Section 7.3.

Compilation to CUDA (overview)

1. Reification Produce a Program AST
2. Convert Program level datatype to list of statements
3. Liveness analysis for arrays in memory
4. Memory mapping
5. CUDA code generation (including virtualisation of threads, warps and blocks)

Compilation to CUDA (overview)

1. Reification — produce a Program AST
2. Convert Program datatype to list of statements
3. Liveness ana
4. Memory ma
5. CUDA code g
   virtualisation ... ks)

Obsidian is quite small
Could be a good EDSL to study!!

Summary I

Key benefit of EDSL is ease of design exploration

Performance is very satisfactory (after parameter exploration)
comparable to Thrust

“Ordinary” benefits of FP are worth a lot here
(parameterisation, reuse, higher order functions etc)

Pull and push arrays a powerful combination

In reality, also need mutable arrays (which are there but need further development, see Thielemann’s experience with Obsidian and Accelerate)

Providing a warp abstraction is good. CUDA doesn’t do it. But super GPU programmers are entirely warp oriented!!

Summary II

Flexibility to add and control sequential behaviour is vital to performance
(Thielemann)

Use of types to model the GPU hierarchy interesting!
gives something in between flat and nested data parallelism

constrains the user to programming idioms appropriate to the GPU
similar ideas could be used in other NUMA architectures

Need to adapt to changes in GPUs (becoming more and more general, e.g. communication between threads in warps via “shuffles”)

What we REALLY need is a layer above Obsidian (plus autotuning)
see spiral.net for inspiring related work

Summary III

I want a set of combinators with strong algebraic properties (e.g. for data-independent algorithms like sorting and scan).

Need something simpler and more restrictive than push arrays

Array combinators have not been sufficiently studied.

A community is forming See Array'15 with PLDI

The bigger picture

Obsidian is a good (backend) tool for exploring what is really the heart of the matter:

Understanding how to provide nice abstractions to the programmer while still gaining performance from parallel machines (which are only going to get more and more parallel)

This is compatible with Blelloch's vision too

We would be happy if any of you wanted to work on using or developing Obsidian ☺

Joel Svensson will be around soon for the second half of the year

CUDA programming is fun, but Obsidian programming is even more fun!