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
[‡] NVIDIA Corporation, USA

Challenge

- **We all know that…**
  – parallel hardware is becoming more widespread
  – shared state combined with concurrency leads to suffering

- **Data parallelism is successful in the large**
  – on server farms: CGI rendering, MapReduce…
  – well understood: Fortran and OpenMP for high performance computing

- **Increasing core counts of CPUs and GPUs (graphics processing unit)**
  – 8-core CPU versus 512-core GPU are the current extreme

- **GPU-like architectures require data parallelism**
  – achieve 20x better performance per watt (judging by peak performance)
  – speedups of 20-150x have been observed in real applications

Challenge

- **Code must be massively data parallel**
  - real SIMD architecture, 32-threads per instruction dispatch unit
  - threads must execute the same instruction to keep the ALUs busy

- **Control structures are limited**
  - no [function] pointers and very limited recursion
  - not suitable for executing arbitrary code from high-level languages

- **Strict memory access patterns, software managed cache**

- **Portability…**

Data.Array.Accelerate

- **Embedded domain-specific language (eDSL)**
  - restricted set of operations
  - individual computations don't need to communicate
  - parallel computations don't spark further parallel computations

✓ limited control structures

- **Collective operations on multi-dimensional regular arrays**
  - arrays are dense and rectangular

✓ massive data parallelism

- **Generative approach based on skeleton templates**

✓ hand-tuned access patterns

- **Multiple backends**

✓ portability

Example

- **Vector dot-product**

Haskell array

EDSL array = description of array computations

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

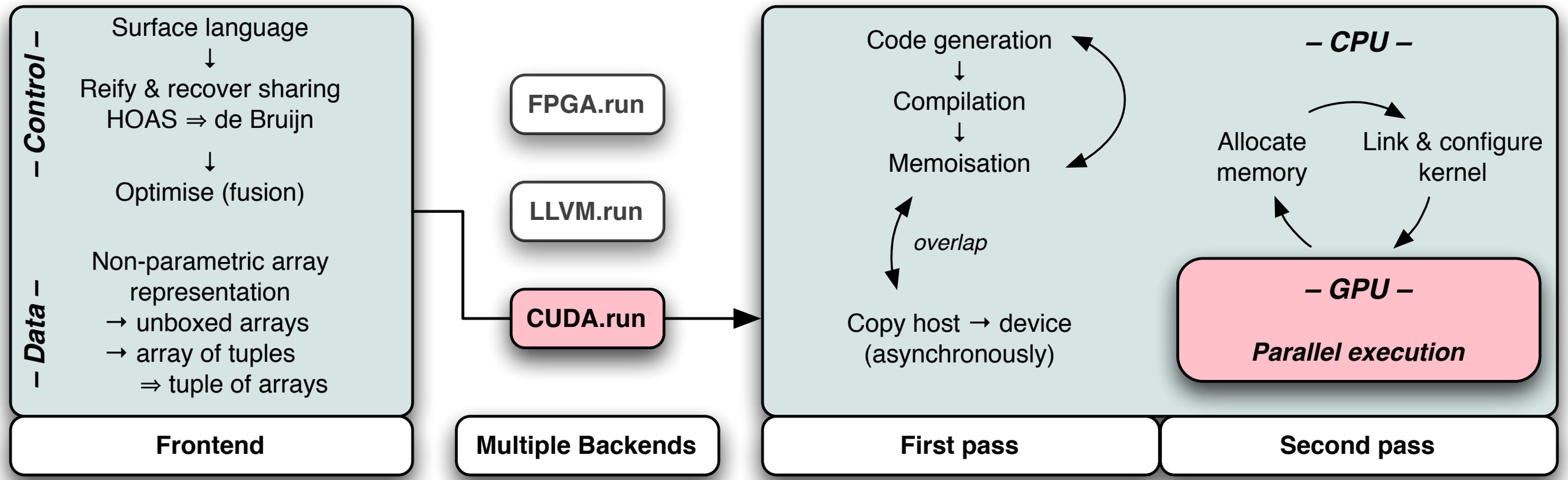
Lift Haskell array into EDSL

EDSL array computations

The Architecture of Data.Array.Accelerate

(The main bits)

Overview


– Control –
Surface language
↓
Reify & recover sharing
HOAS ⇒ de Bruijn
↓
Optimise (fusion)
– Data –
Non-parametric array representation
→ unboxed arrays
→ array of tuples
⇒ tuple of arrays
FPGA.run
LLVM.run
CUDA.run
Code generation
↓
Compilation
↓
Memoisation
overlap
Copy host → device
(asynchronously)
– CPU –
Allocate memory
Link & configure kernel
– GPU –
Parallel execution
Frontend
Multiple Backends
First pass
Second pass

The surface language

- **Regular, multi-dimensional arrays**
  – non-parametric representation for array elements (type family)

```
data Array dim e

type DIM0 = Z
type DIM1 = Z :. Int
type DIM2 = Z :. Int :. Int
  <and so on>

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

- **Stratified language**
  – collective operations comprise many scalar computations
  – the range of `Exp` computations is intentionally restricted

```
data Exp e              -- embedded scalar computation form
data Acc a              -- array expression form
```

- **Scalar operations**
  – overloaded arithmetic, comparisons, relations, scalar indexing, array shape

Frontend

- **Reify HOAS (higher-order abstract syntax)**
  – operations do not directly issue computations

```
zipWith :: (Ix dim, Elem a, Elem b, Elem c)
        => (Exp a -> Exp b -> Exp c)
        -> Acc (Array dim a)
        -> Acc (Array dim b)
        -> Acc (Array dim c)
zipWith = ZipWith
```

– evaluation reflects the computation into a term tree
– `Acc` is a GADT whose constructors represent collective operations

```
data Acc a where
   ...
   ZipWith :: (Elem e1, Elem e2, Elem e3)
           => (Exp e1 -> Exp e2 -> Exp e3)
           -> Acc (Array dim e1)
           -> Acc (Array dim e2)
           -> Acc (Array dim e3)
  <and so on>
```

Frontend

- **Recover sharing and convert HOAS → de Bruijn**
  – nameless de Bruijn representation necessary for “looking under lambdas”
  – type preserving transformation

- **Optimisation (not currently done)**
  – all operations create real arrays & associated memory traffic
  – index transformations generally want to be pushed into the consumer
  – no array fusion: `map f . map g → map (f . g)`

CUDA backend

- **Overall: map a nameless (de Bruijn) AST to low-level code**
  – together with architecture specific optimisations (also not done)

- **Parallel behaviour is encapsulated by specific algorithmic skeletons**
  – one for each collective operation in `Acc`
  – parameterised by type and scalar operations injected at predefined points

```
__global__ void zipWith
(
    TyOut *         d_out,
    const TyIn1 *   d_in1,
    const TyIn0 *   d_in0,
    const int       length
){
    int ix   = blockDim.x * blockIdx.x + threadIdx.x;
    int grid = blockDim.x * gridDim.x;

    for (; ix < length; ix += grid) {
        d_out[ix] = apply(d_in1[ix], d_in0[ix]);
    }
}
```

Code generation

- **Generally straightforward translation from de Bruijn to C AST forms**
  - currently throws out all type information (oops…)
  - lambdas are easy because `Exp` contains no general application form

```
Fold add (Const 0) (ZipWith mul (Use xs) (Use ys))
  where
    mul = Lam (Lam (Body (PrimMul (FloatingType ...)
                         `PrimApp`
                         Tuple (NilTup `SnocTup` (Var (SuccIdx ZeroIdx))
                                               `SnocTup` (Var ZeroIdx))))
    add = ...
```

```
static inline __device__
TyOut apply(const TyIn1 x1, const TyIn0 x0)
{
    return x1 * x0;
}
```

  - finally, add appropriate `typedef`s for element types & `#include` the skeleton

Code generation (2)

- **Multidimensional indices are mapped to structs**

```
typedef int32_t                Ix;
typedef Ix                     DIM1;
typedef struct { Ix a1,a0; } DIM2;
   <and so on>
```

- **Skeletons need to convert to/from multidimensional & row-major form**
  - family of function for the various index types
  - CUDA supports function overloading → skeletons are ad-hoc polymorphic

```
int  dim(DIMn sh);
int  size(DIMn sh);
int  toIndex(DIMn sh, DIMn ix);     // index into row-major representation
DIMn fromIndex(DIMn sh, Ix ix);     // invert ‘toIndex’
```

Code generation (3)

- **Scalar expression forms with array valued arguments**

```
IndexScalar :: Acc (Array dim t) -> Exp dim  -> Exp t
Shape       :: Acc (Array dim e)             -> Exp dim
```

- **Execution**
  – first lift the array computation out of the scalar expression and let-bind it

- **Code generation**
  – parameter list to a skeleton function is fixed
  – extra inputs are introduced through global variables
  – arrays are read through texture references (cached read-only arrays)
  – shape information stored in `__constant__` memory

Code generation (4)

- **Arrays of tuples**
  – Accelerate supports arbitrary tuple types, something not available in CUDA

```
typedef struct { int a; float b; } arrayIntFloat[];
```

- **Non-parametric representation**
  – arrays of tuples → tuple of arrays

```
typedef struct { int a[]; float b[]; } arrayIntFloat;
```

- **Getters and Setters**
  – we would also like the convenience of working with tuples in scalar code

```
typedef struct { int  a1; float  a0; } TyIn0;
typedef struct { int* a1; float* a0; } ArrIn0

static inline __device__
TyIn0 get0(const ArrIn0 d_in0, const Ix idx)
{
    TyIn1 r = { d_in0.a1[idx], d_in0.a0[idx] };
    return r;
}
```

Code generation (5)

- **Tuples in scalar expressions**
  - nested tuple expressions provide additional pain
  - internally flattened; tuple projection must be wary of indexing past groups

```
foo :: Acc (Vector (Int,(Float,Float)) -> Acc (Vector Int)
foo = map fst
```

```
foo = Map (Lam (Body (Prj (SuccTupIdx ZeroTupIdx) (Var ZeroIdx))))
```

```
typedef int TyOut;
typedef struct { int a2; float a1; float a0; } TyIn0;

static inline __device__
TyOut apply(const TyIn0 x0)
{
    return x0.a2;
}
```

∴ 1 = 2 QED

Execution

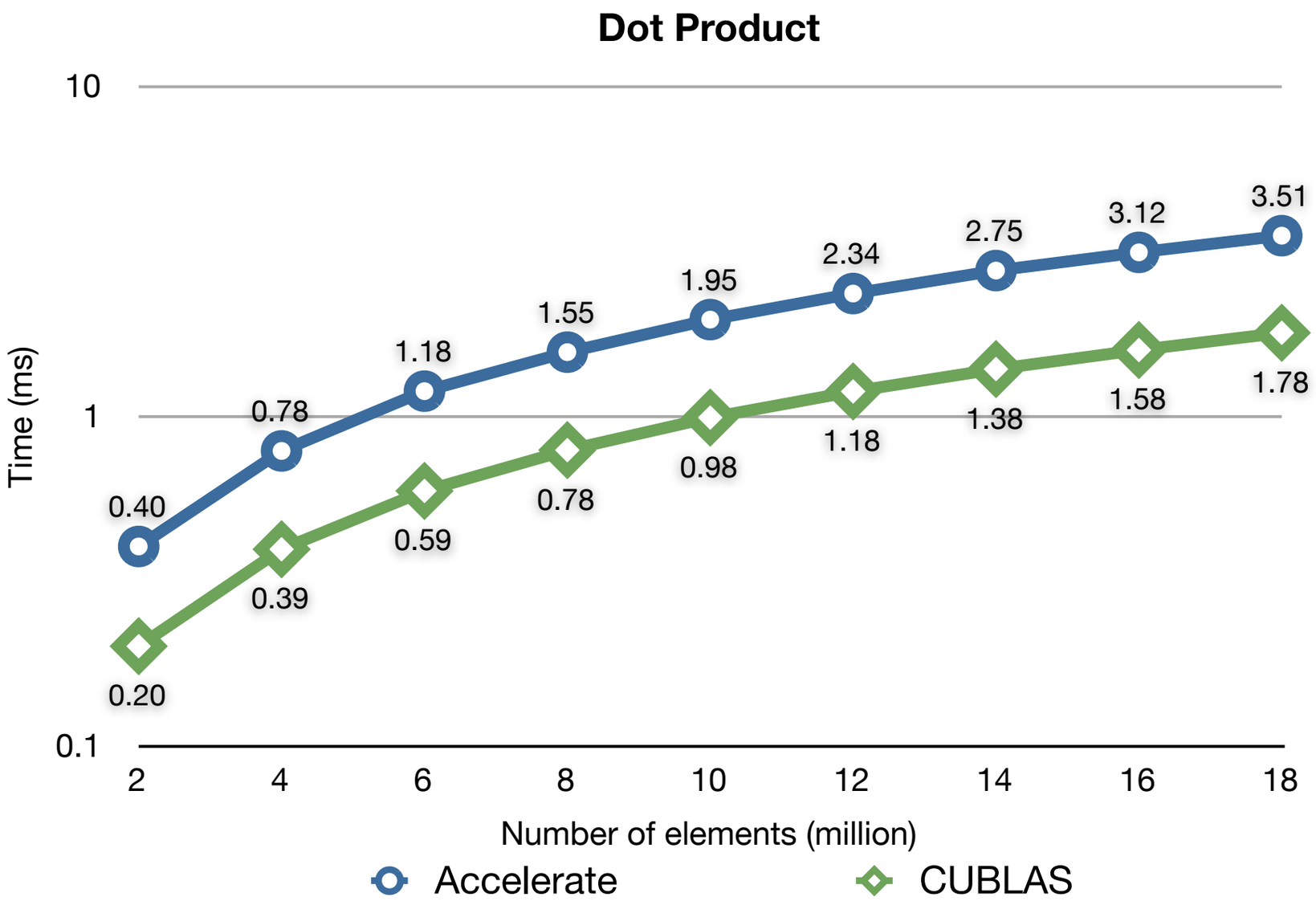
- **Second traversal of the nameless AST**
  1. Use nodes return a reference to the associated device memory
  2. non-skeleton nodes (let bindings, shape manipulation) executed directly
  3. skeleton nodes link & execute the appropriate binary

- **Kernel execution**
  – interleave host-side control and device execution; FFI bindings
  – efforts made for optimal thread/block launch configuration
  – expected vs. provided function argument types?

- **Single point of entry: CUDA.run**
  – hides some unsafe use of `unsafePerformIO`

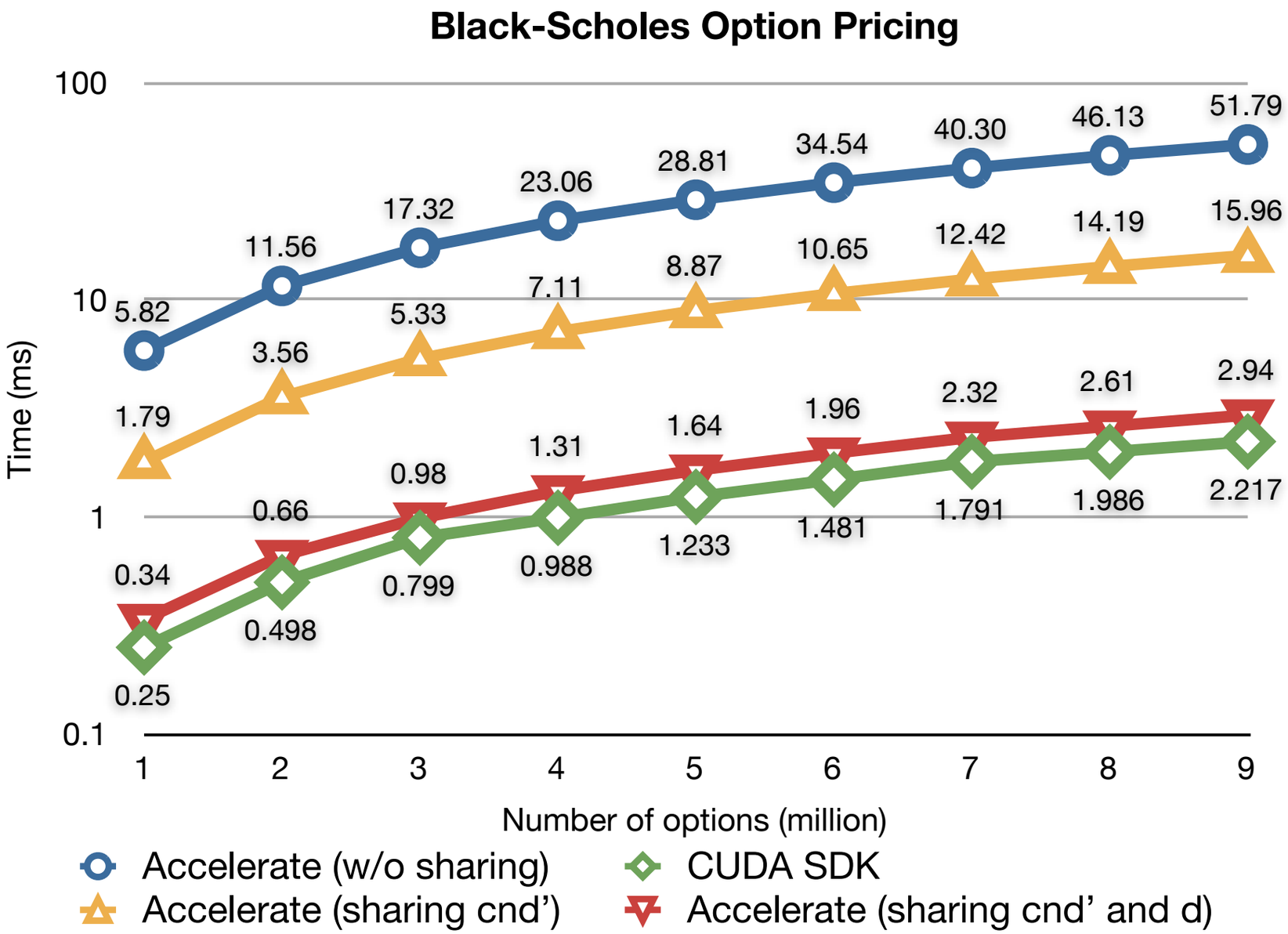
Results

- **Vector dot-product**
  - lack of array fusion results in extra memory traffic
  - Data.Vector computes 18M in 71ms



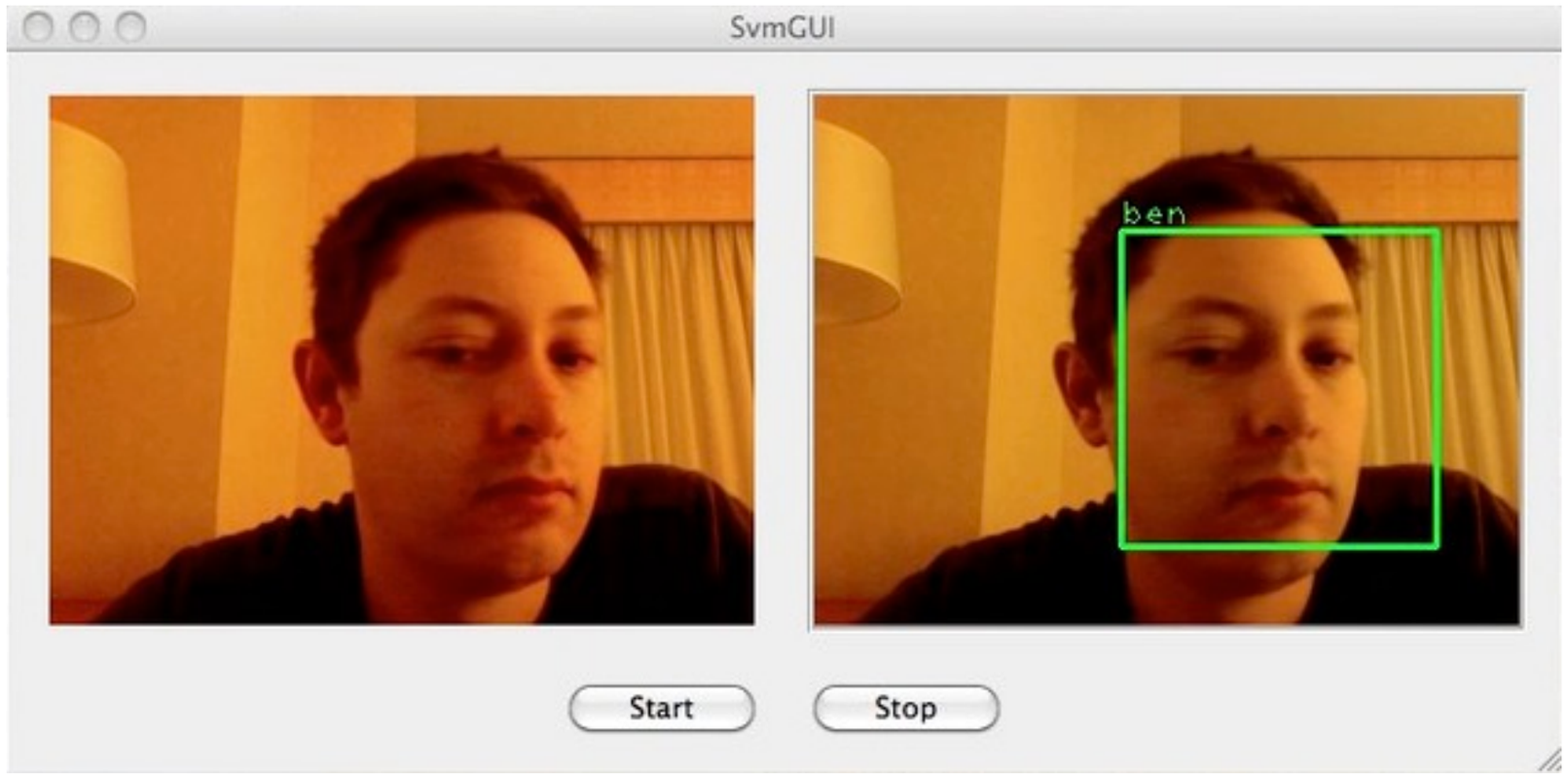
Results

- **Black-Scholes option pricing**
  – lack of scalar subexpression sharing results in SIMD divergence



~~Drunk~~ Face Detection

- **Feature matching algorithm, written in Accelerate**
  – courtesy Ben Lever, NICTA



Love & bunnies?

- **Lack of type information**
- **Supported collective operations**
- **Optimisations**
  – array fusion
  – subexpression sharing (array valued arguments; scalar expressions)
- **Code generation driven by device capabilities**
  – double only supported on compute 1.3 devices and above
  – failures occur very late in the pipeline

Break it:

http://trac.haskell.org/accelerate
http://hackage.haskell.org/package/accelerate

- Early version on Hackage, CUDA backend requires a moderately recent NVIDIA GPU

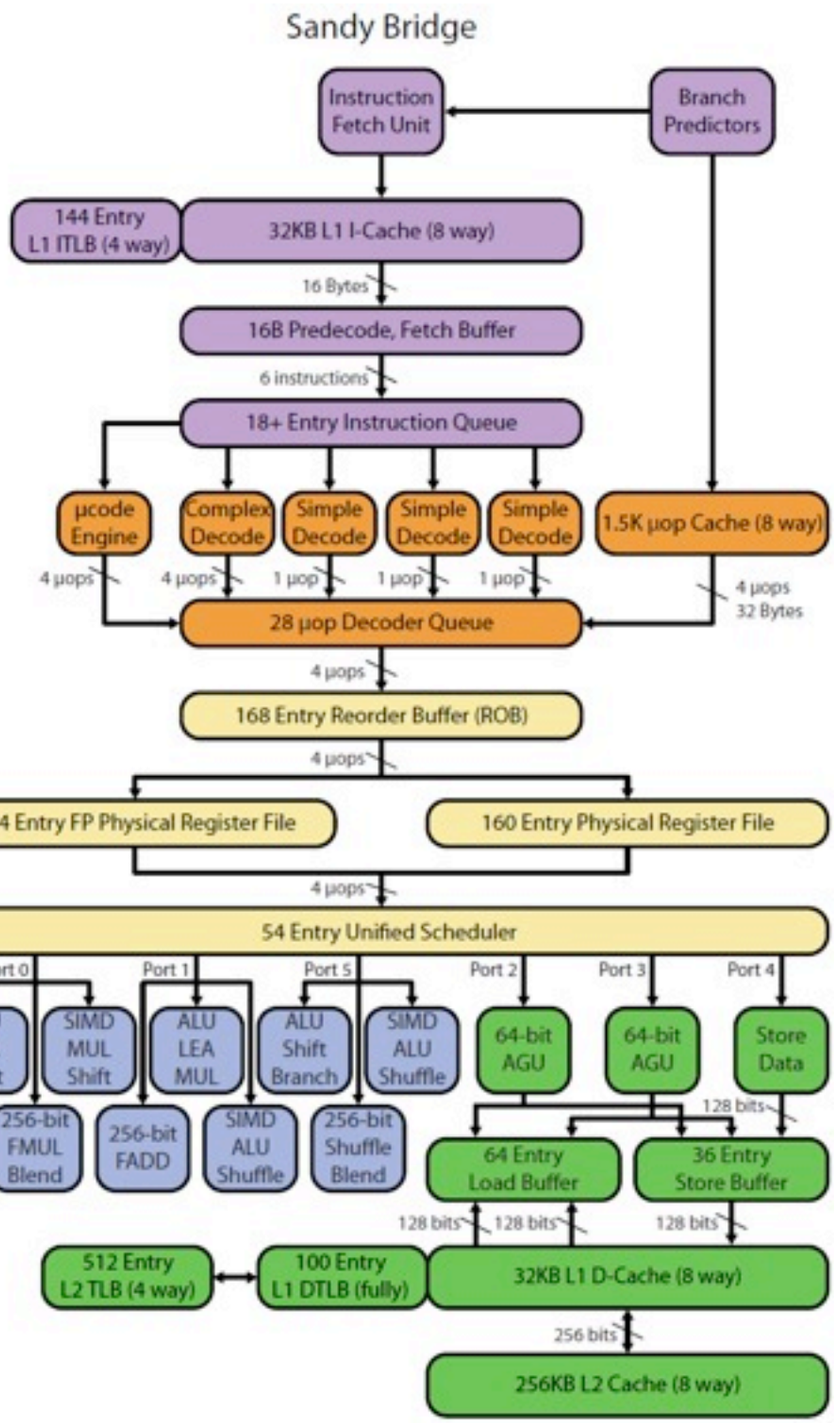
- Send me example programs (or patches!) and I will add them to the repository

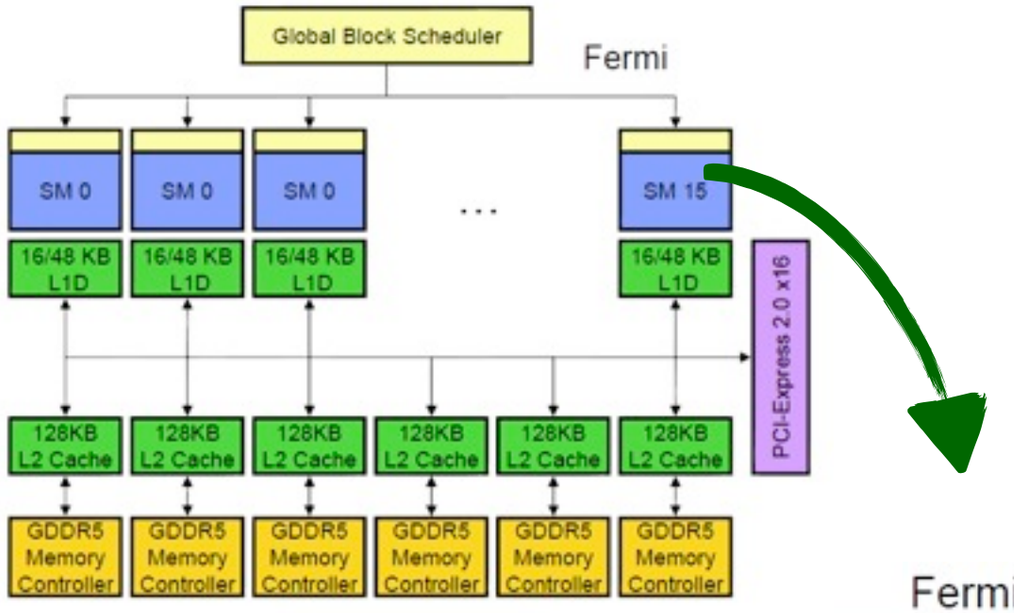
- Looking for backend contributors

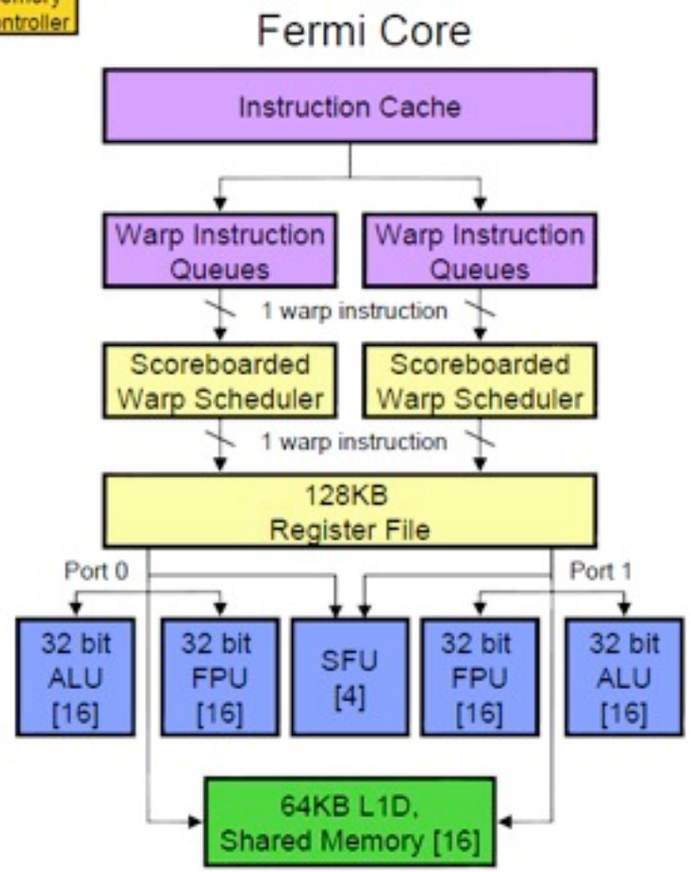
Questions?

Extra Slides…

Architecture-101







David Kanter, http://realworldtech.com

The API

- **Scalar operations**

```
instance Num (Exp e)                                        -- overloaded arithmetic
instance Integral (Exp e)
   <and so on>

(==*), (/=*), (<*), (<=*), (>*), (>=*), min, max            -- comparisons
(&&*), (||*), not                                           -- logical operators

(?) :: Elem t                                               -- conditional expression
    => Exp Bool -> (Exp t, Exp t) -> Exp t

(!) :: (Ix dim, Elem e)                                     -- scalar indexing
    => Acc (Array dim e) -> Exp dim -> Exp e

shape :: Ix dim                                             -- yield an array shape
      => Acc (Array dim e) -> Exp dim
```

The API

- **Collective array operations: creation**

```
-- use an array from Haskell land
use :: (Ix dim, Elem e)
    => Array dim e -> Acc (Array dim e)

-- create a singleton array
unit :: Elem e
     => Exp e -> Acc (Scalar e)

-- multidimensional slice and replicate
-- changing array dimensionality is tracked through type hackery
replicate :: (SliceIx slix, Elem e)
          => Exp slix
          -> Acc (Array (Slice    slix) e)
          -> Acc (Array (SliceDim slix) e)

slice :: (SliceIx slix, Elem e)
      => Acc (Array (SliceDim slix) e)
      -> Exp slix
      -> Acc (Array (Slice slix) e)

-- Example: replicate (Z :. 2 :. All :. 3) arr
-- yields a three-dimensional array where arr is replicated twice across the first
-- and three times across the second dimension
```

The API

- **Collective array operations: mapping**

```
map :: (Ix dim, Elem a, Elem b)
    => (Exp a -> Exp b)
    -> Acc (Array dim a)
    -> Acc (Array dim b)

zipWith :: (Ix dim, Elem a, Elem b, Elem c)
        => (Exp a -> Exp b -> Exp c)
        -> Acc (Array dim a)
        -> Acc (Array dim b)
        -> Acc (Array dim c)

-- a map with local context; also stencil2 variant analogous to zipWith
stencil :: (Ix dim, Elem a, Elem b, Stencil dim a stencil)
        => (stencil -> Exp b)                       -- stencil function
        -> Boundary a                               -- boundary condition
        -> Acc (Array dim a)                        -- source array
        -> Acc (Array dim b)                        -- destination array
```

The API

- **Collective array operations: reductions**

```
-- also segmented and inclusive (fold1) versions of both
fold :: (Ix dim, Elem a)                                    -- over innermost dimension
     => (Exp a -> Exp a -> Exp a)                           -- associative
     -> Exp a
     -> Acc (Array (dim:.Int) a)
     -> Acc (Array dim a)

scan :: Elem a                                              -- left and right variants
     => (Exp a -> Exp a -> Exp a)                           -- associative
     -> Exp a
     -> Acc (Vector a)
     -> Acc (Vector a)
```

The API

- **Collective array operations: permutations**

```
permute :: (Ix dim, Ix dim', Elem a)
        => (Exp a -> Exp a -> Exp a)                    -- combination function
        -> Acc (Array dim' a)                           -- array of default values
        -> (Exp dim -> Exp dim')                        -- permutation
        -> Acc (Array dim  a)                           -- permuted array
        -> Acc (Array dim' a)

backpermute :: (Ix dim, Ix dim', Elem a)
            => Exp dim'                                 -- shape of the result array
            -> (Exp dim' -> Exp dim)                    -- permutation
            -> Acc (Array dim  a)                       -- permuted array
            -> Acc (Array dim' a)
```

The API

- **Collective array operations: construction**

```
-- construct a new array by applying a function at each index
-- large potential for misuse
generate :: (Ix dim, Elem a)
         => Exp dim
         -> (Exp dim -> Exp a)
         -> Acc (Array dim a)
```