HASKELL ARRAYS, ACCELERATED USING GPUS

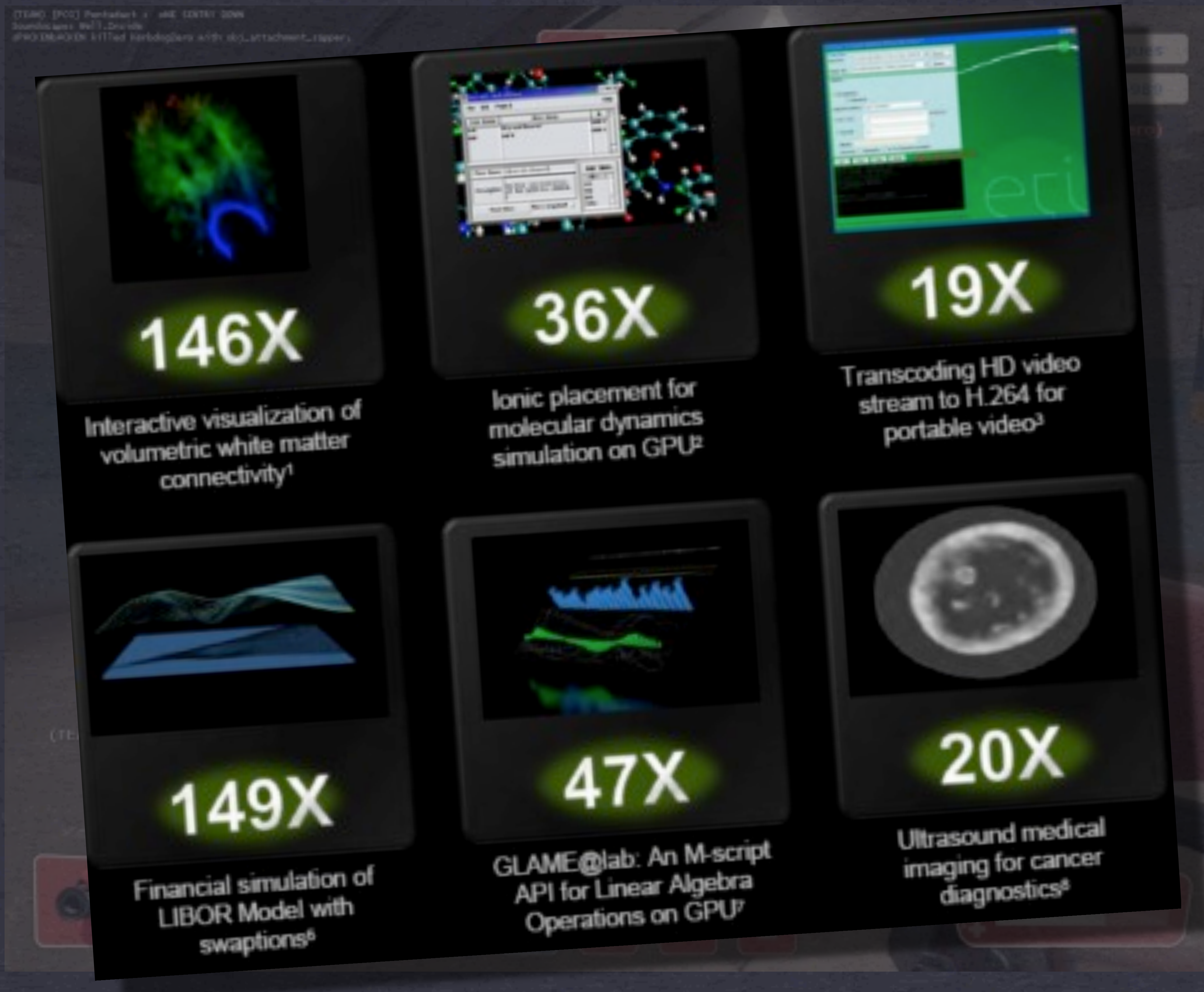
Manuel M. T. Chakravarty
University of New South Wales

JOINT WORK WITH
Gabriele Keller
Sean Lee

General Purpose GPU Programming (GPGPU)

Modern GPUs are Freely Programmable

But no function pointers & limited recursion



Modern GPUs are Freely Programmable

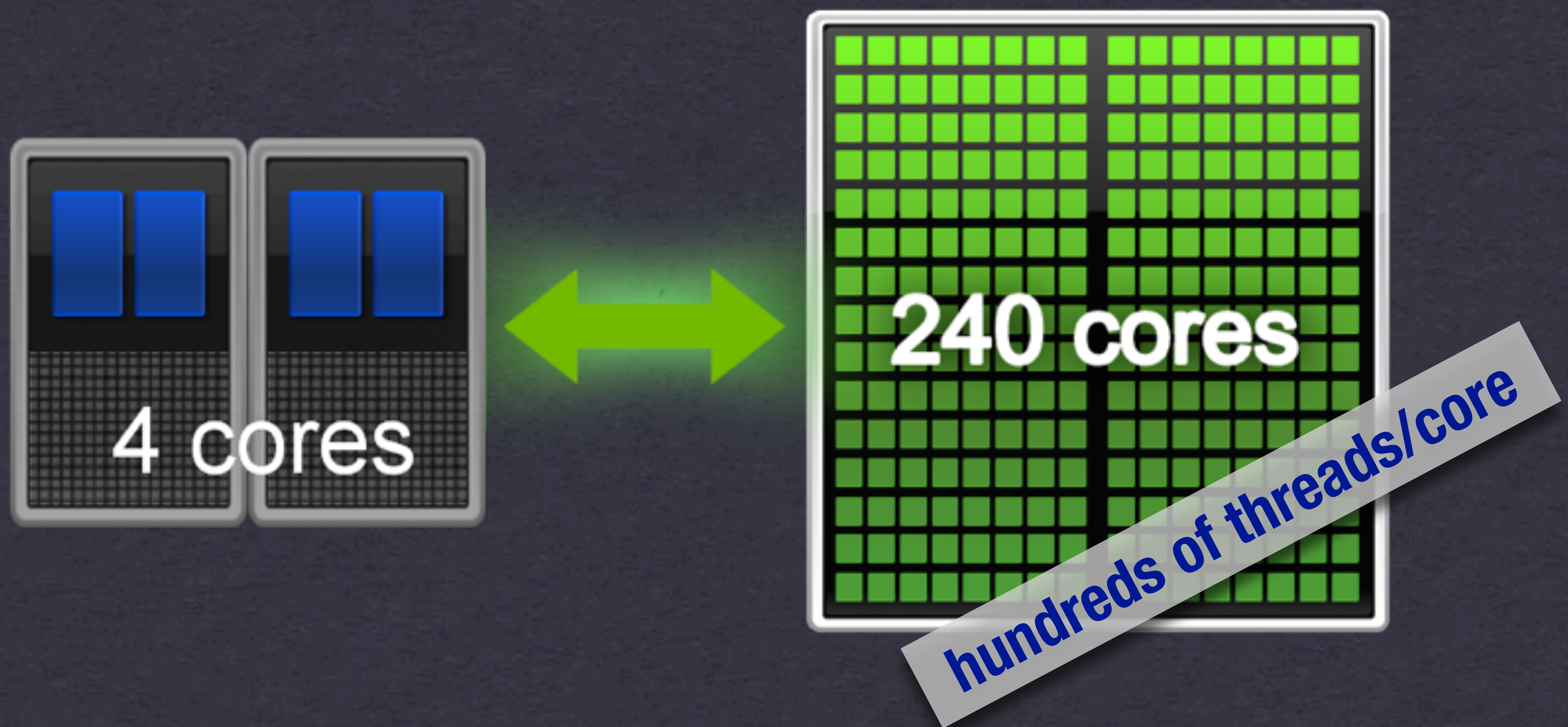
But no function pointers & limited recursion

Very Different Programming Model

(Compared to multicore CPUs)

Quadcore
Xeon CPU

Tesla T10
GPU



Massively Parallel Programs Needed

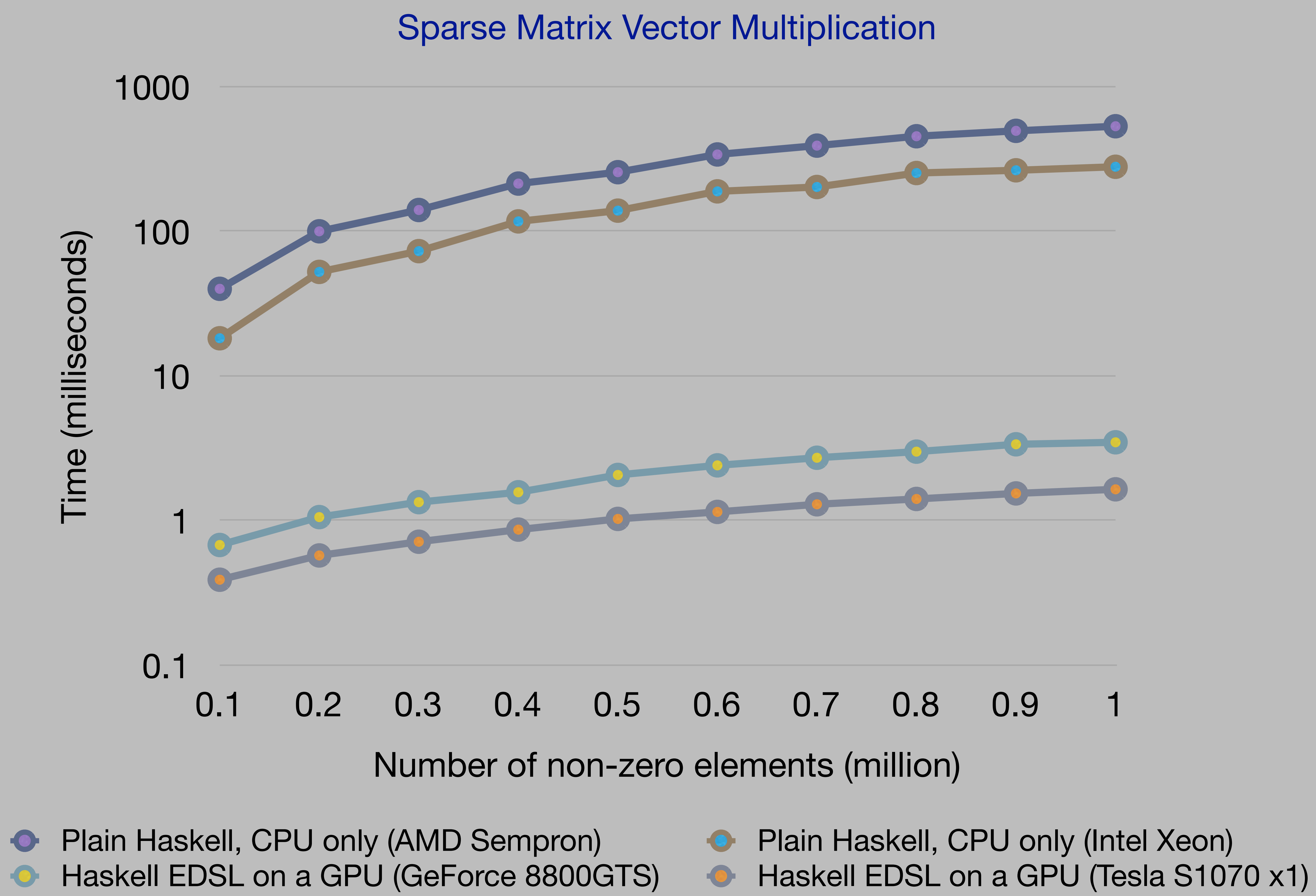
Tens of thousands of dataparallel threads

Programming GPUs is hard! Why bother?

PLS

Reduce power consumption!

- GPU achieves 20x better performance/Watt (judging by peak performance)
- Speedups between 20x to 150x have been observed in real applications



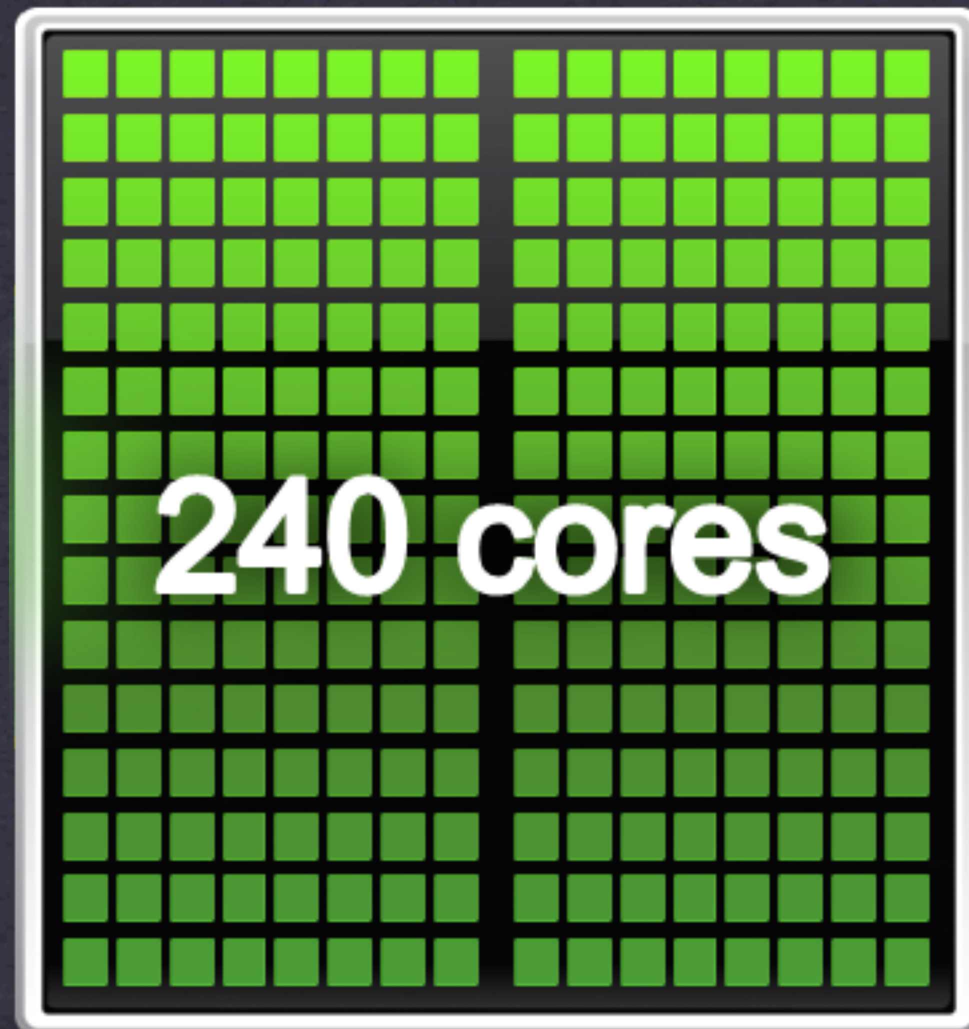
Prototype of code generator targeting GPUs

Computation only, without CPU ⇄ GPU transfer

Challenges

- Code must be massively dataparallel
- Control structures are limited
  - No function pointers
  - Very limited recursion
- Software-managed cache, memory-access patterns, etc.
- Portability...

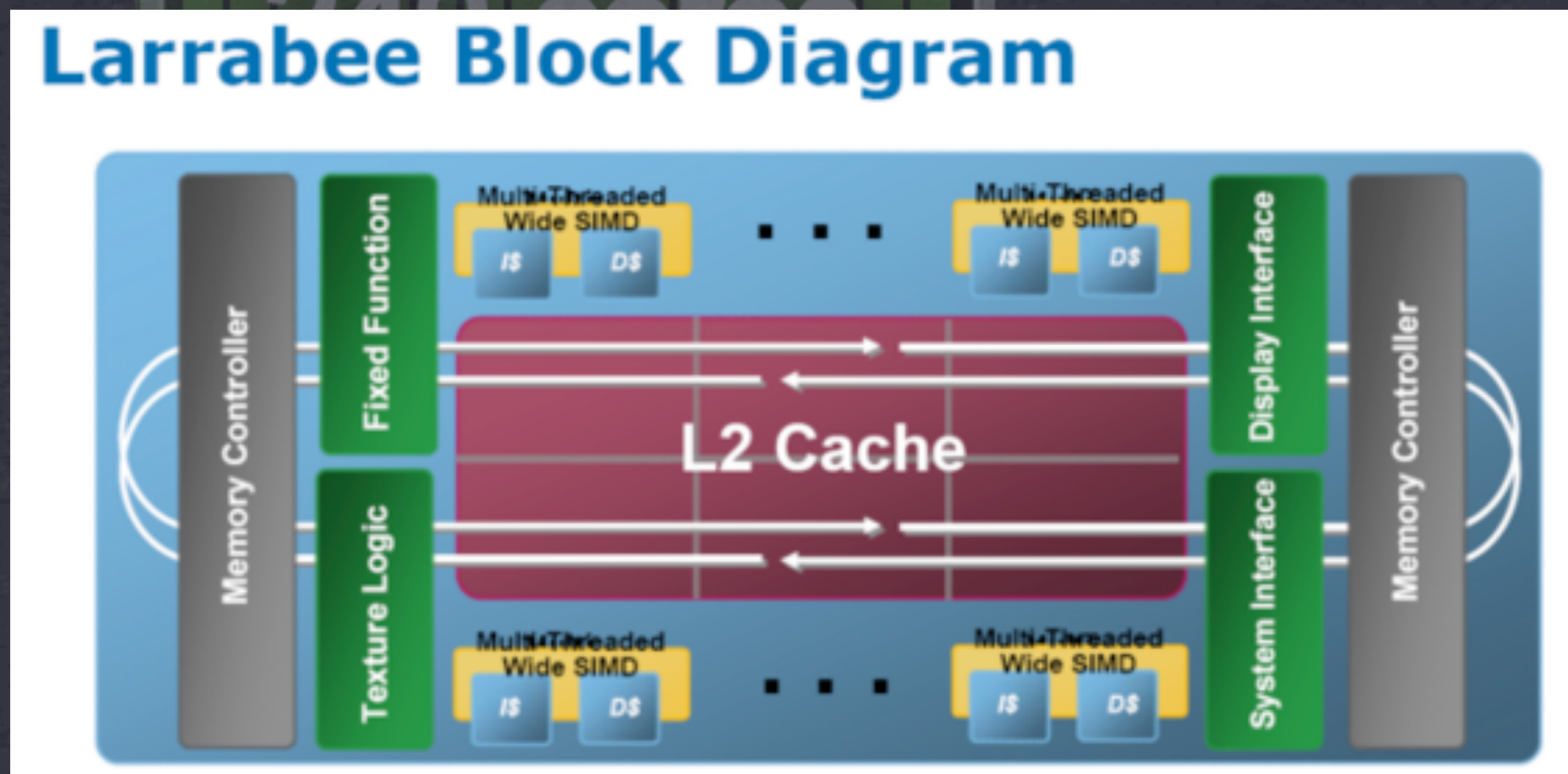
PLS



Tesla T10
GPU

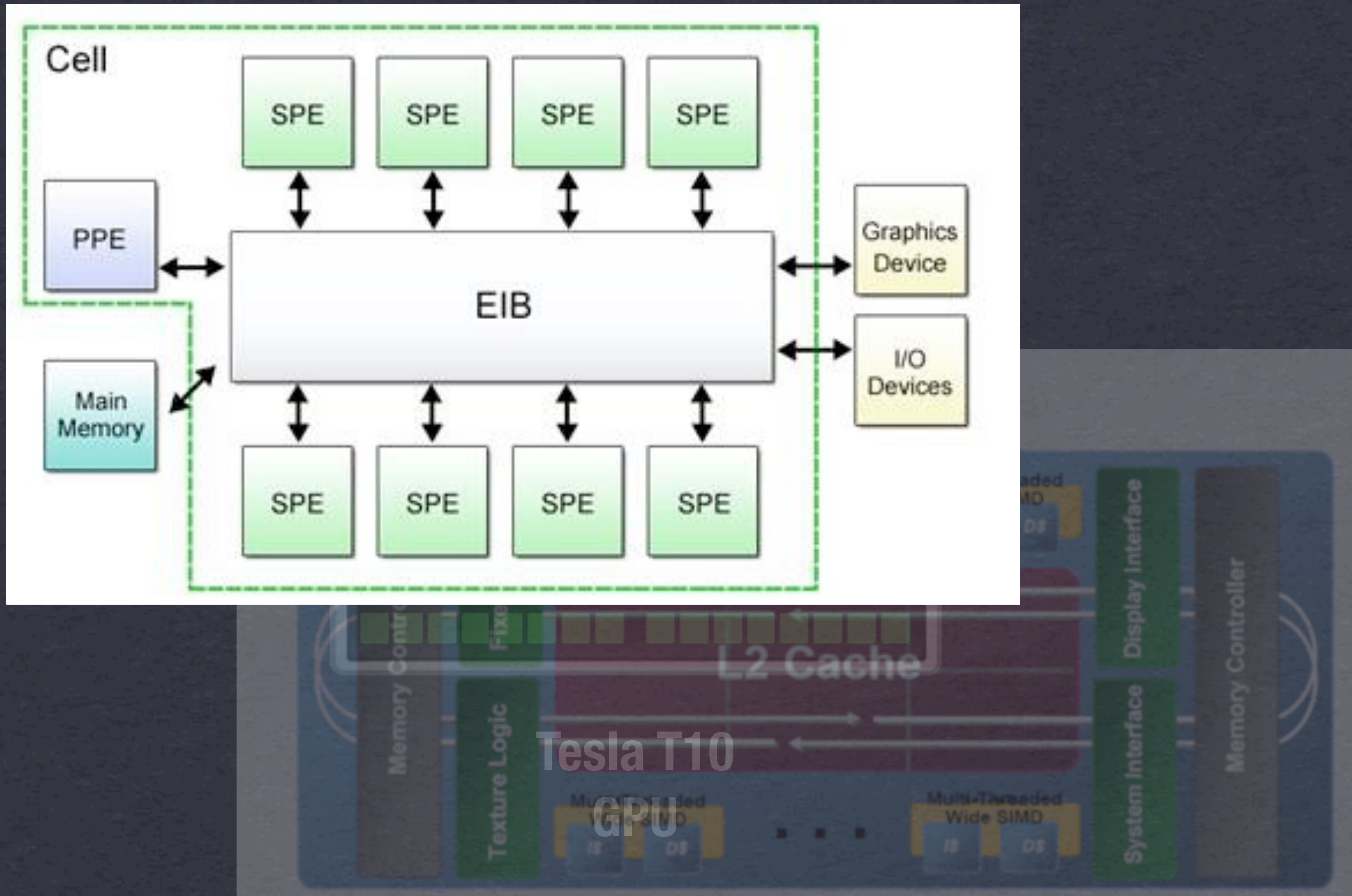
Other Compute Accelerator Architectures

Goal: portable data parallelism



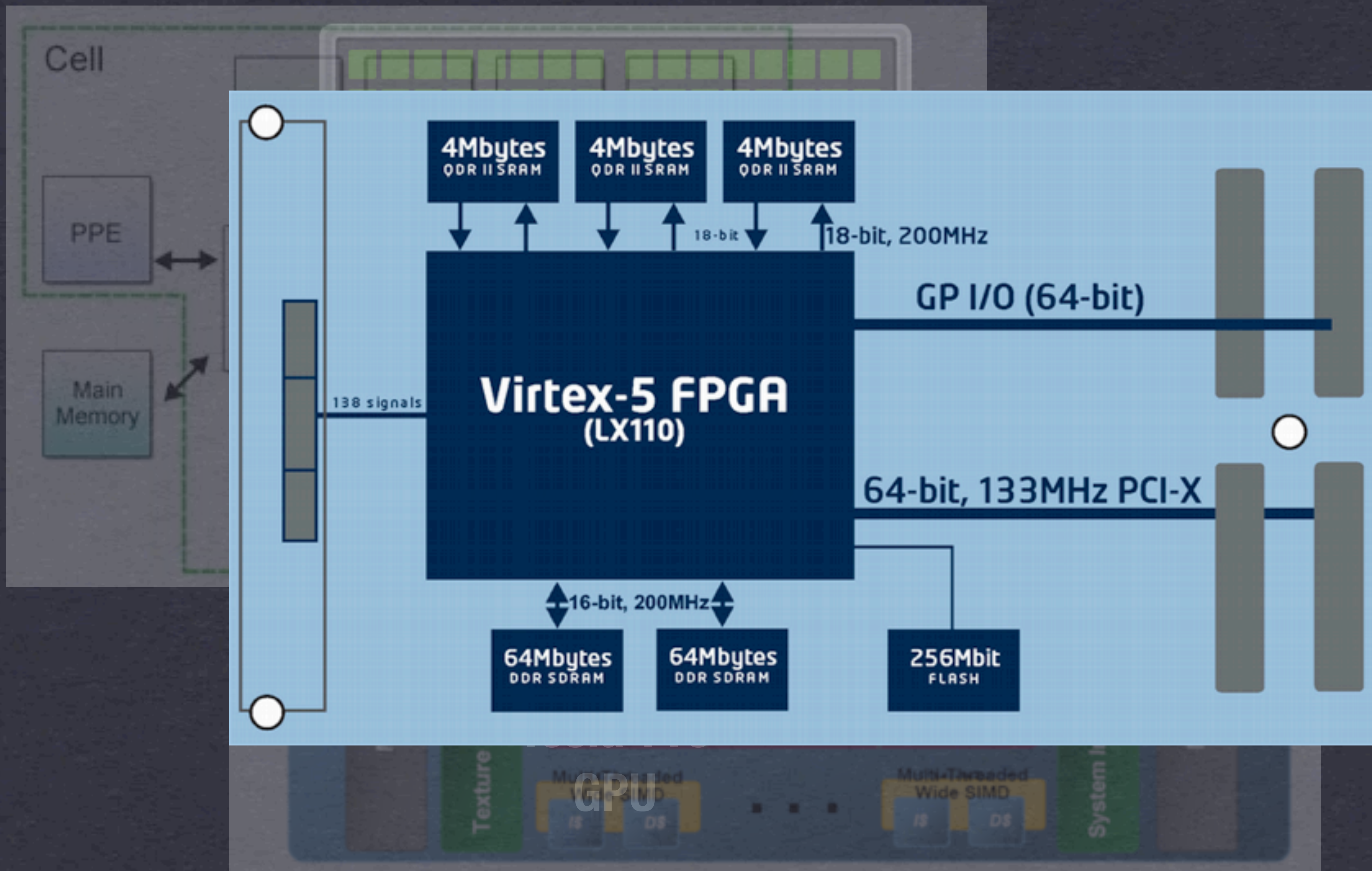
Other Compute Accelerator Architectures

Goal: portable data parallelism



Other Compute Accelerator Architectures

Goal: portable data parallelism



Other Compute Accelerator Architectures

Goal: portable data parallelism

Data.Array.Accelerate

- Collective operations on multi-dimensional regular arrays
- Embedded DSL
  - Restricted control flow
  - First-order GPU code
- Generative approach based on combinator templates
- Multiple backends

PLS

Data.Array.Accelerate

- Collective operations on multi-dimensional regular arrays

✓ massive data parallelism

- Embedded DSL
  - Restricted control flow
  - First-order GPU code
- Generative approach based on combinator templates
- Multiple backends

PLS

Data.Array.Accelerate

- Collective operations on multi-dimensional regular arrays
- Embedded DSL
  - Restricted control flow
  - First-order GPU code
- Generative approach based on combinator templates
- Multiple backends

✓ massive data parallelism

✓ limited control structures

PLS

Data.Array.Accelerate

- Collective operations on multi-dimensional regular arrays
- Embedded DSL
  - Restricted control flow
  - First-order GPU code
- Generative approach based on combinator templates
- Multiple backends

✓ massive data parallelism

✓ limited control structures

✓ hand-tuned access patterns

PLS

Data.Array.Accelerate

- Collective operations on multi-dimensional regular arrays
- Embedded DSL
  - Restricted control flow
  - First-order GPU code
- Generative approach based on combinator templates
- Multiple backends

✓ massive data parallelism

✓ limited control structures

✓ hand-tuned access patterns

✓ portability

import Data.Array.Accelerate

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')
```

PLS

import Data.Array.Accelerate

Dot product

Haskell array

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

PLS

import Data.Array.Accelerate

Dot product

Haskell array

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

EDSL array =
desc. of array comps

import Data.Array.Accelerate

Dot product

Haskell array

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

EDSL array = desc. of array comps

Lift Haskell arrays into EDSL — may trigger host→device transfer

PLS

import Data.Array.Accelerate

Dot product

Haskell array

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

EDSL array = desc. of array comps

Lift Haskell arrays into EDSL — may trigger host→device transfer

EDSL array computations

PLS

import Data.Array.Accelerate

Sparse-matrix vector multiplication

```
type SparseVector a = Vector (Int, a)
type SparseMatrix a = (Segments, SparseVector a)

smvm :: Acc (SparseMatrix Float)
     -> Acc (Vector Float)
     -> Acc (Vector Float)
smvm (segd, smat) vec
  = let
      (inds, vals) = unzip smat
      vecVals      = backpermute (shape inds)
                                 (\i -> inds!i) vec
      products     = zipWith (*) vecVals vals
    in
    foldSeg (+) 0 products segd
```

import Data.Array.Accelerate

[0, 0, 6.0, 0, 7.0] ≈ [(2, 6.0), (4, 7.0)]

Sparse-matrix vector multiplication

```
type SparseVector a = Vector (Int, a)
type SparseMatrix a = (Segments, SparseVector a)

smvm :: Acc (SparseMatrix Float)
     -> Acc (Vector Float)
     -> Acc (Vector Float)
smvm (segd, smat) vec
  = let
      (inds, vals) = unzip smat
      vecVals      = backpermute (shape inds)
                                 (\i -> inds!i) vec
      products     = zipWith (*) vecVals vals
    in
    foldSeg (+) 0 products segd
```

import Data.Array.Accelerate

[0, 0, 6.0, 0, 7.0] ≈ [(2, 6.0), (4, 7.0)]

Sparse-matrix vector multiplication

```
type SparseVector a = Vector (Int, a)
type SparseMatrix a = (Segments, SparseVector a)

smvm :: Acc (SparseMatrix Float)
     -> Acc (Vector Float)
     -> Acc (Vector Float)
smvm (segd, smat) vec
  = let
      (inds, vals) = unzip smat
      vecVals      = backpermute (shape inds)
                                 (\i -> inds!i) vec
      products     = zipWith (*) vecVals vals
    in
    foldSeg (+) 0 products segd
```

[[10, 20], [], [30]] ≈ ([2, 0, 1], [10, 20, 30])

Architecture of Data.Array.Accelerate

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

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

Reify & HOAS -> de Bruijn

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

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

Reify & HOAS -> de Bruijn

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

Recover sharing
(CSE or Observe)

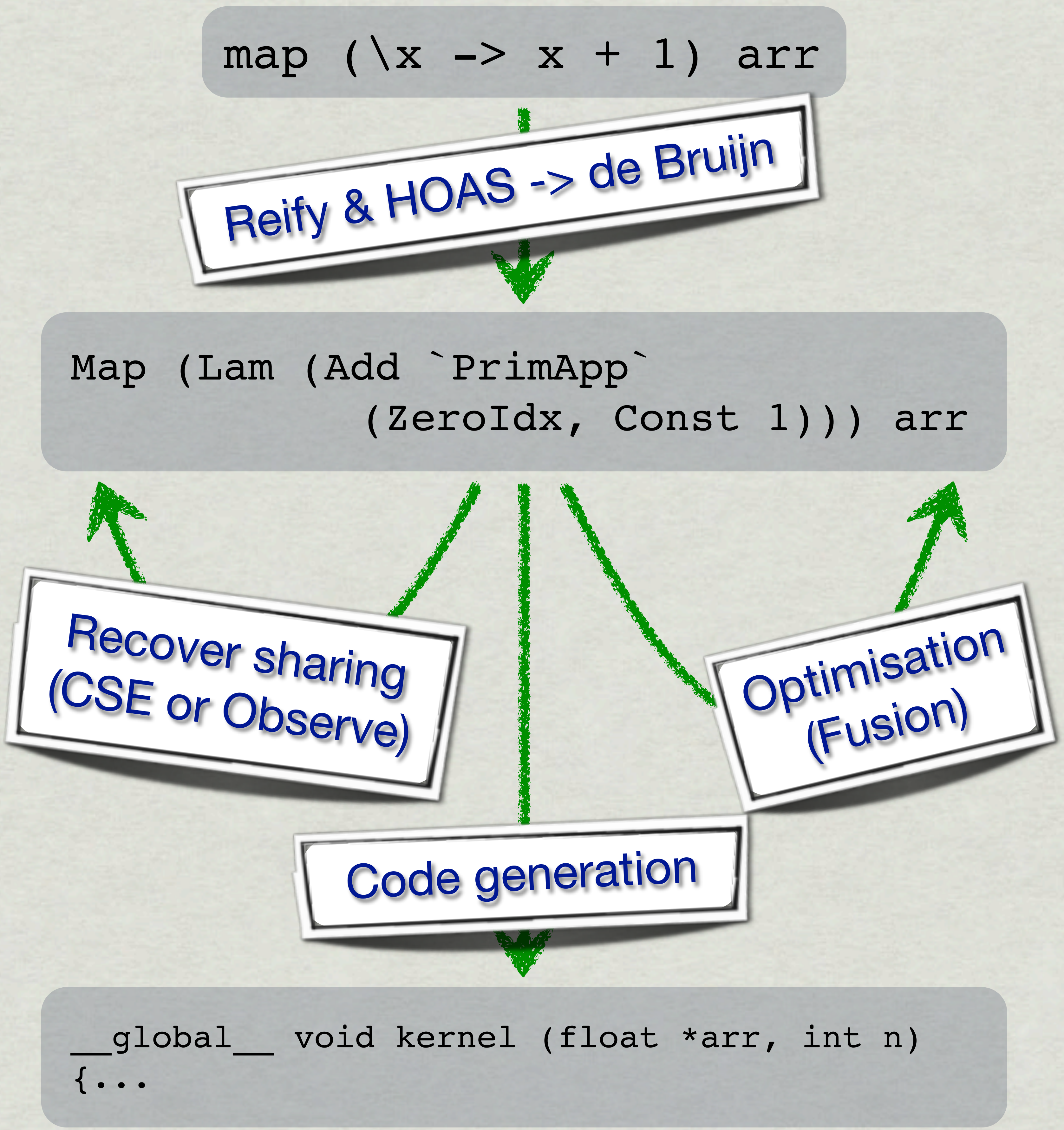
```
map (\x -> x + 1) arr
```

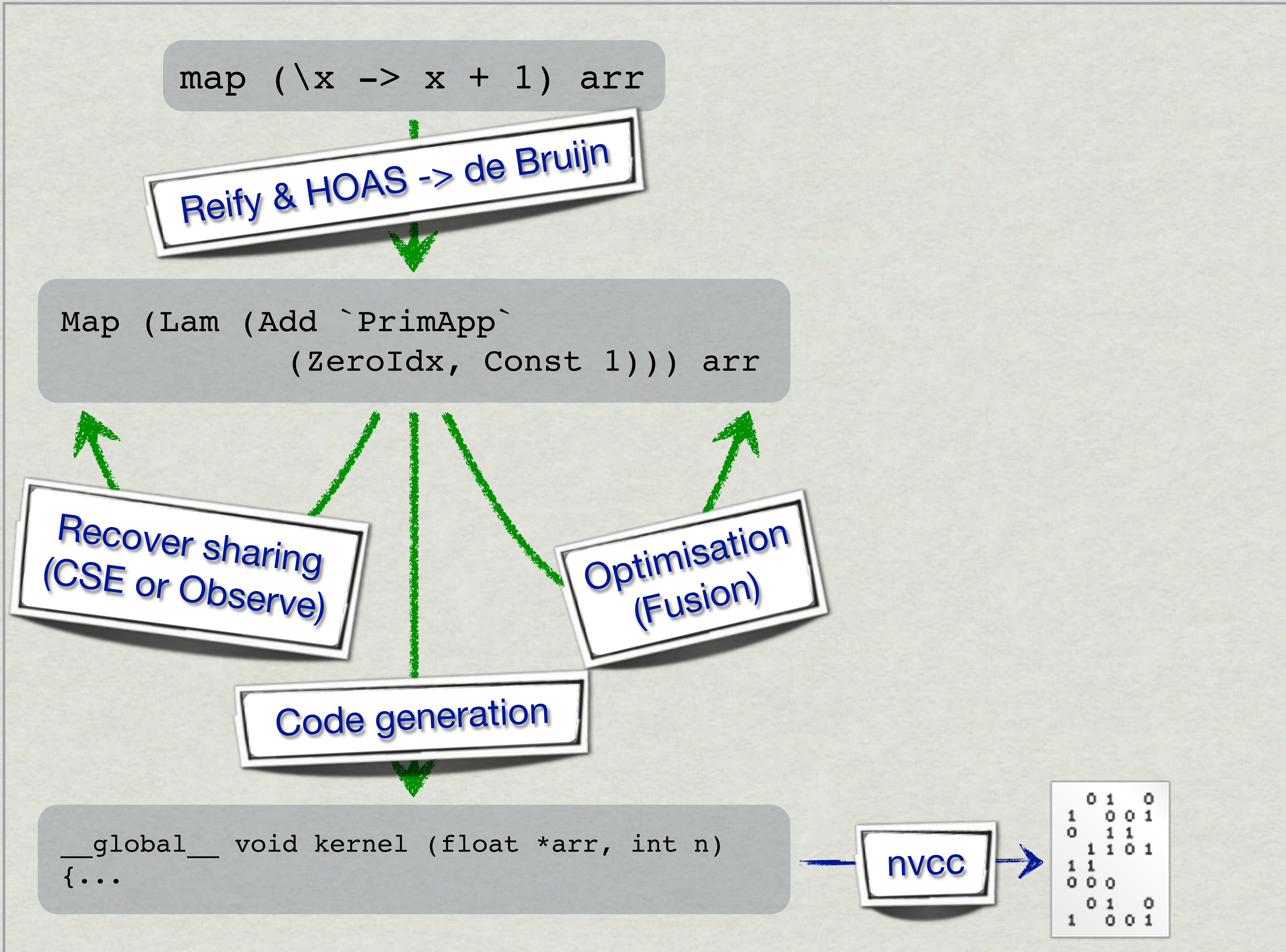
Reify & HOAS -> de Bruijn

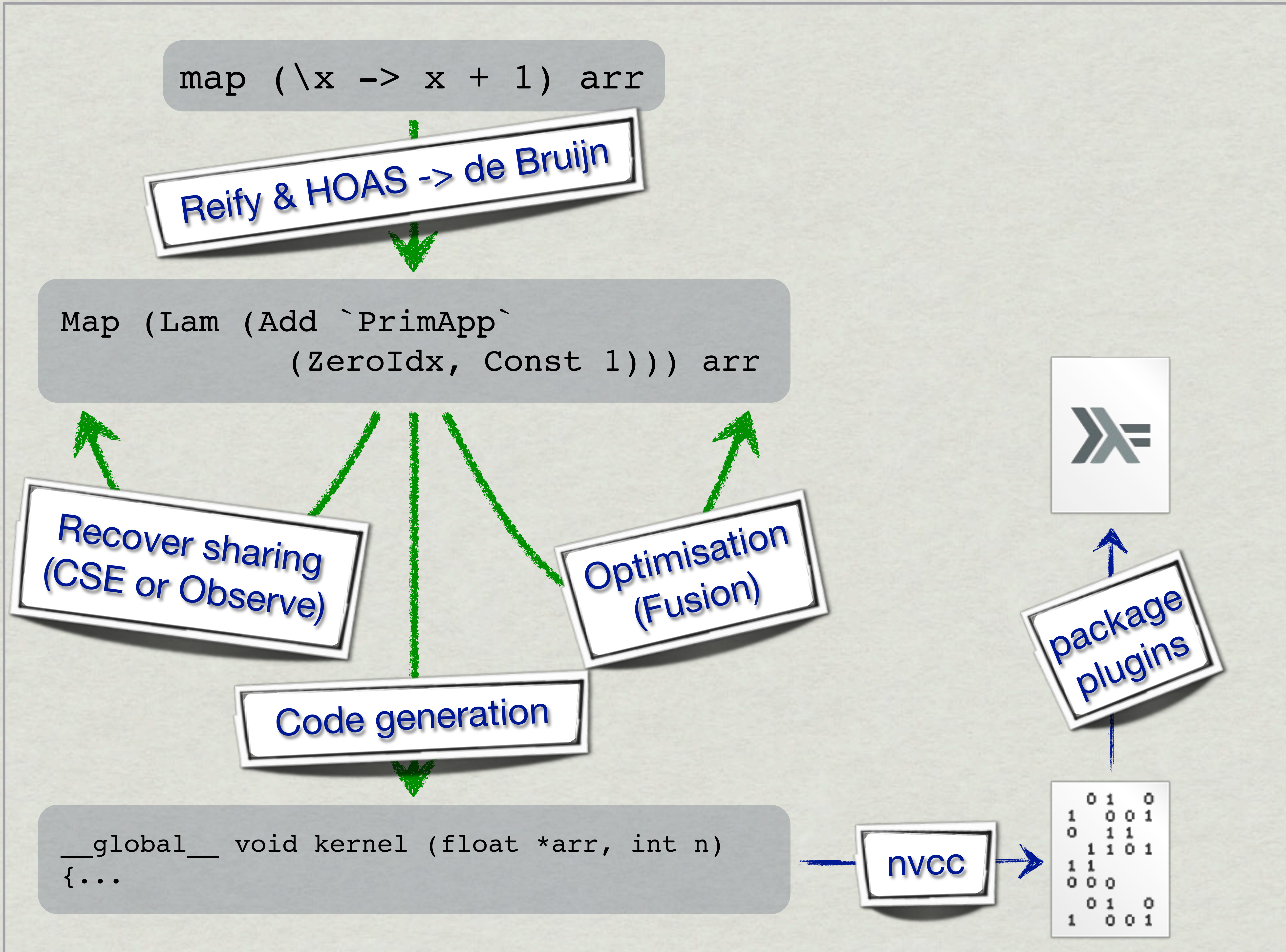
```
Map (Lam (Add `PrimApp`
            (ZeroIdx, Const 1))) arr
```

Recover sharing
(CSE or Observe)

Optimisation
(Fusion)


map (\x -> x + 1) arr
Reify & HOAS -> de Bruijn
Map (Lam (Add `PrimApp`
(ZeroIdx, Const 1))) arr
Recover sharing
(CSE or Observe)
Optimisation
(Fusion)
Code generation
__global__ void kernel (float *arr, int n)
{...


map (\x -> x + 1) arr
Reify & HOAS -> de Bruijn
Map (Lam (Add `PrimApp`
(ZeroIdx, Const 1))) arr
Recover sharing
(CSE or Observe)
Optimisation
(Fusion)
Code generation
__global__ void kernel (float *arr, int n)
{...
nvcc


map (\x -> x + 1) arr
Reify & HOAS -> de Bruijn
Map (Lam (Add `PrimApp`
(ZeroIdx, Const 1))) arr
Recover sharing
(CSE or Observe)
Optimisation
(Fusion)
Code generation
__global__ void kernel (float *arr, int n)
{...
nvcc
package
plugins

The API of Data.Array.Accelerate

(The main bits)

PLS

Array types

```
data Array dim e
```
— regular, multi-dimensional arrays

```
type DIM0 = ()
type DIM1 = Int
type DIM2 = (Int, Int)
```
⟨and so on⟩

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

Array types

```
data Array dim e
```
— regular, multi-dimensional arrays

```
type DIM0 = ()
type DIM1 = Int
type DIM2 = (Int, Int)
```
⟨and so on⟩

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

EDSL forms

```
data Exp e
```
— scalar expression form

```
data Acc a
```
— array expression form

Array types

```
data Array dim e
```
— regular, multi-dimensional arrays

```
type DIM0 = ()
type DIM1 = Int
type DIM2 = (Int, Int)
```
⟨and so on⟩

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

EDSL forms

```
data Exp e
```
— scalar expression form

```
data Acc a
```
— array expression form

Classes

```
class Elem e
```
— scalar and tuples types

```
class Elem ix => Ix ix
```
— unit and integer tuples

Scalar operations

```
instance Num (Exp e)                 — overloaded arithmetic
instance Integral (Exp e)
⟨and so on⟩

(==*), (/=*), (<*), (<=*),           — comparisons
  (>*), (>=*), min, max
(&&*), (||*), 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 array shape
      => Acc (Array dim e) -> Exp dim
```

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

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

— multi-dimensional replication

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

— Example: `replicate (All, 10, All) twoDimArr`

Collective array operations — slicing

— slice extraction

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

— Example: `slice (5, All, 7) threeDimArr`

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)
```

PLS

Collective array operations — reductions

```
fold :: (Ix dim, Elem a)
     => (Exp a -> Exp a -> Exp a)   — associative
     -> Exp a
     -> Acc (Array dim a)
     -> Acc (Scalar a)

scan :: Elem a
     => (Exp a -> Exp a -> Exp a)   — associative
     -> Exp a
     -> Acc (Vector a)
     -> (Acc (Vector a), Acc (Scalar a))
```

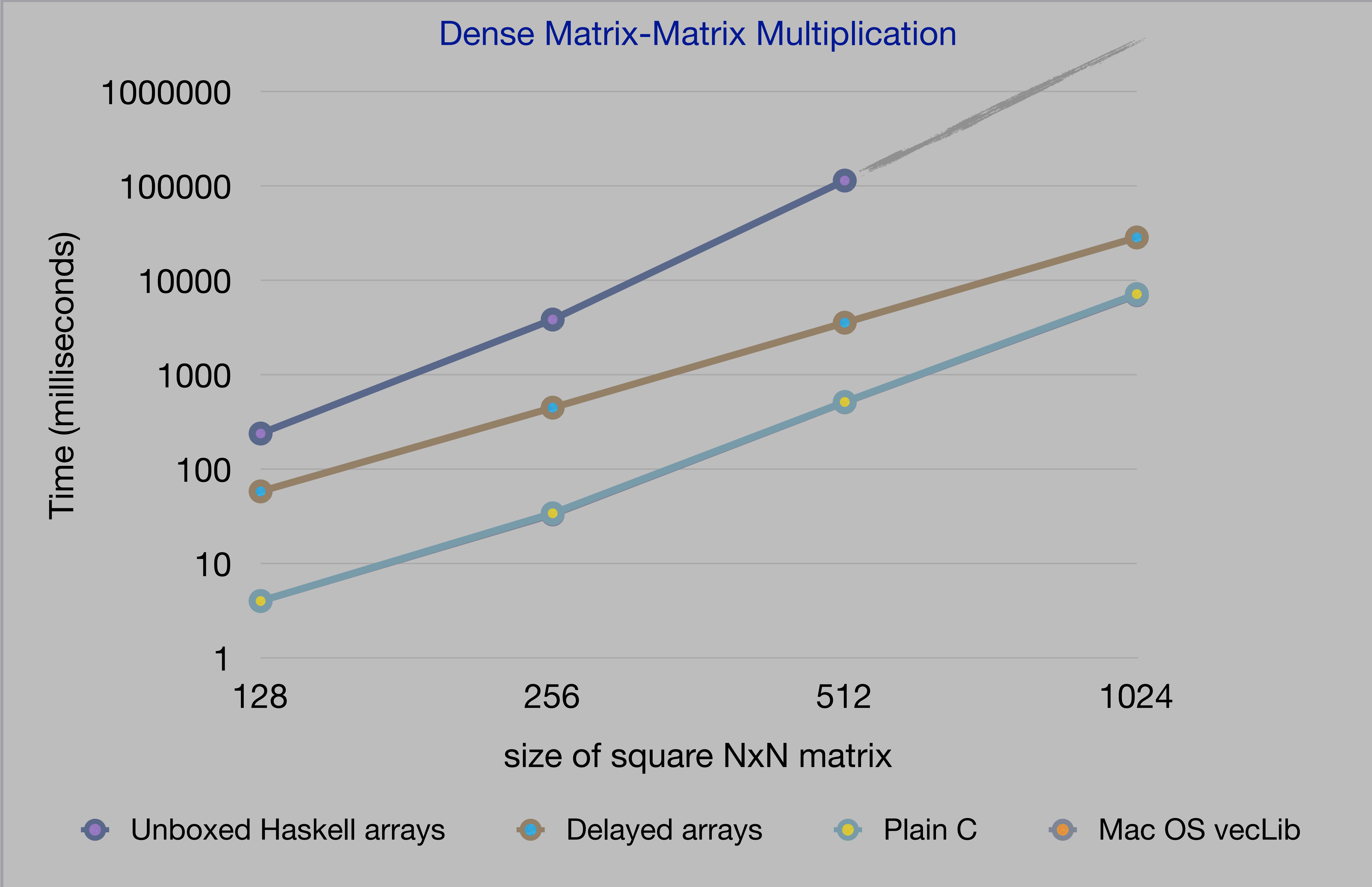
PLS

Collective array operations — permutations

```
permute :: (Ix dim, Ix dim', Elem a)
        => (Exp a -> Exp a -> Exp a)
        -> Acc (Array dim' a)
        -> (Exp dim -> Exp dim')
        -> Acc (Array dim  a)
        -> Acc (Array dim' a)

backpermute :: (Ix dim, Ix dim', Elem a)
            => Exp dim'
            -> (Exp dim' -> Exp dim)
            -> Acc (Array dim  a)
            -> Acc (Array dim' a)
```

PLS



Regular arrays in package dph-seq

@ 3.06 GHz Core 2 Duo (with GHC 6.11)

Conclusion

* EDSL for processing multi-dimensional arrays
* Collective array operations (highly data parallel)
* Support for multiple backends
* Status:
    * Very early version on Hackage (only interpreter)

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

    * Currently porting GPU backend over
    * Looking for backend contributors

PLS