SHARED DATA STRUCTURES IN NESTED DATA PARALLELISM

Manuel M T Chakravarty

University of New South Wales

JOINT WORK WITH

Gabriele Keller

Roman Leshchinskiy

Ben Lippmeier

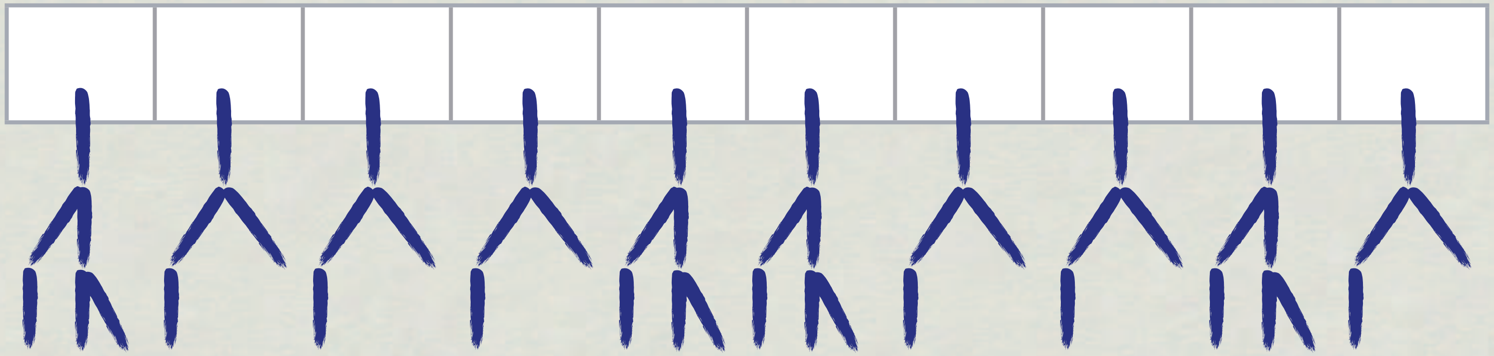
Simon Peyton Jones

Flattening

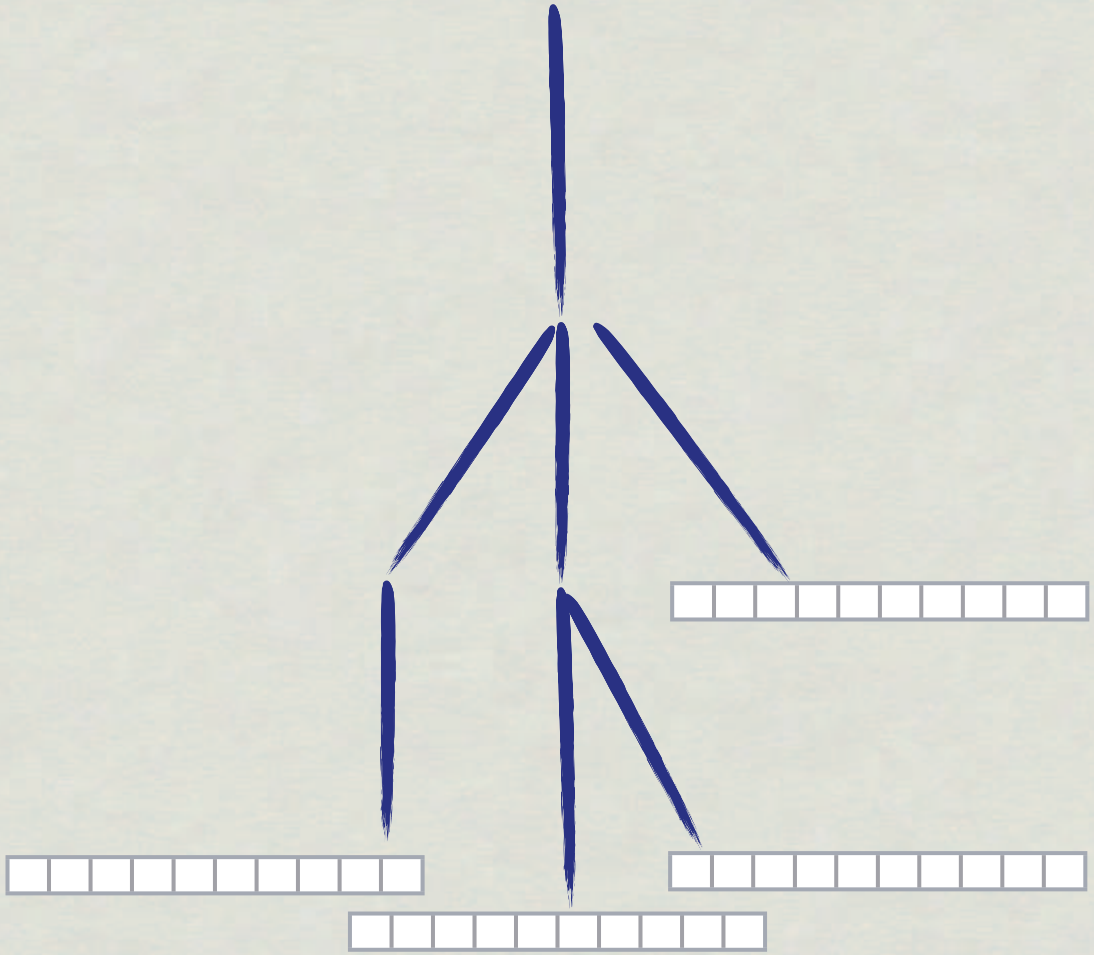
- Flattening is a program transformation
- It transforms both code and data structures
- Scalar computations become array-valued
- We perform it on GHC's Core language (an extended lambda calculus)

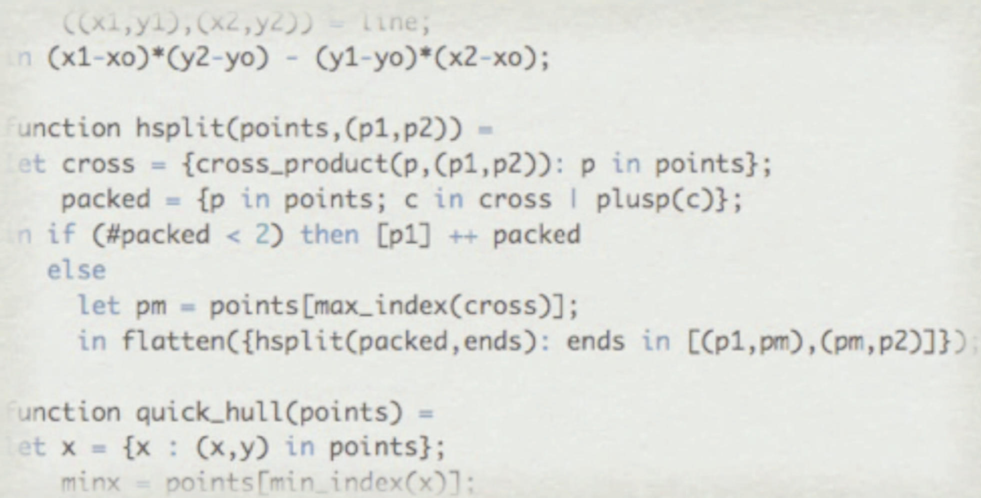
Part of the implementation of Data Parallel Haskell

Flattening forests



Flattening forests





- Introduced by Blelloch & Sabot for NESL

CM-2

http://www.mission-base.com/tamiko/cm/cm-image.html

The age of SIMD machines

CRAY Y-MP

http://en.wikipedia.org/wiki/File:Cray_Y-MP_GSFC.jpg

```
= (x1-xo) * (y2 - yo) - (y1 - yo) * (x2 - xo)


split :: [:Point:] -> Line -> [:Point:]
split points line@(p1, p2)
 | lengthP packed Int.== 0 = [:p1:]
 | otherwise
 = concatP [: hsplit packed ends | ends <- [:(p1, pm), (pm, p2):] :]
 where
   cross  = [: distance p line | p <- points :]
   packed = [: p | (p,c) <- zipP points cross, c > 0.0 :]
   pm     = points !: maxIndexP cross


uickHull :: [:Point:] -> [:Point:]
uickHull points
 | lengthP points Int.== 0 = points
 | otherwise
```

- We extended it to cover

sum types

```
data Either a b = Left a | Right b
```

recursive data types

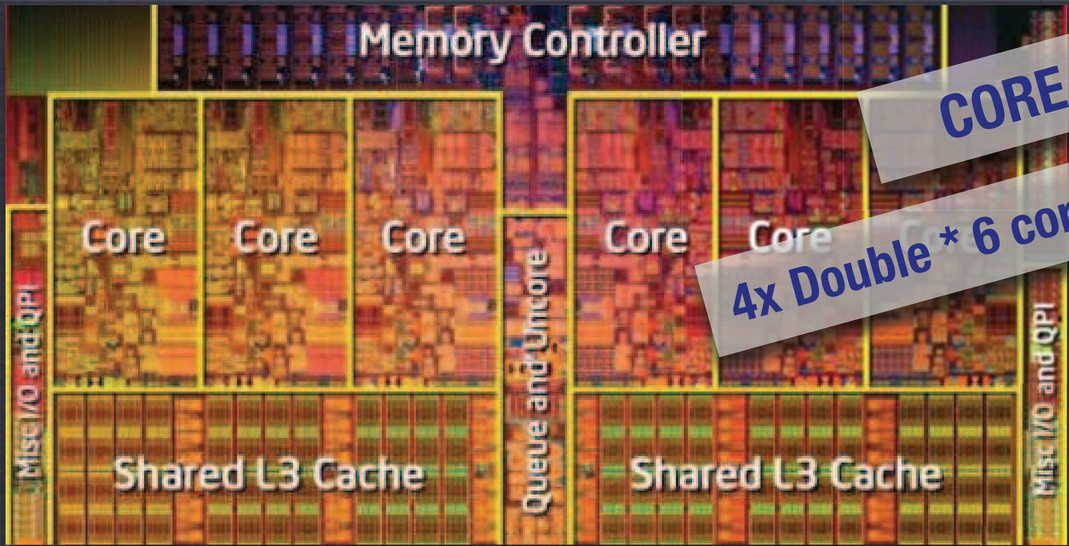
```
data Tree a = Tree a [:Tree a:]
```

higher-order functions

```
mapP :: (a -> b) -> [:a:] -> [:b:]
```

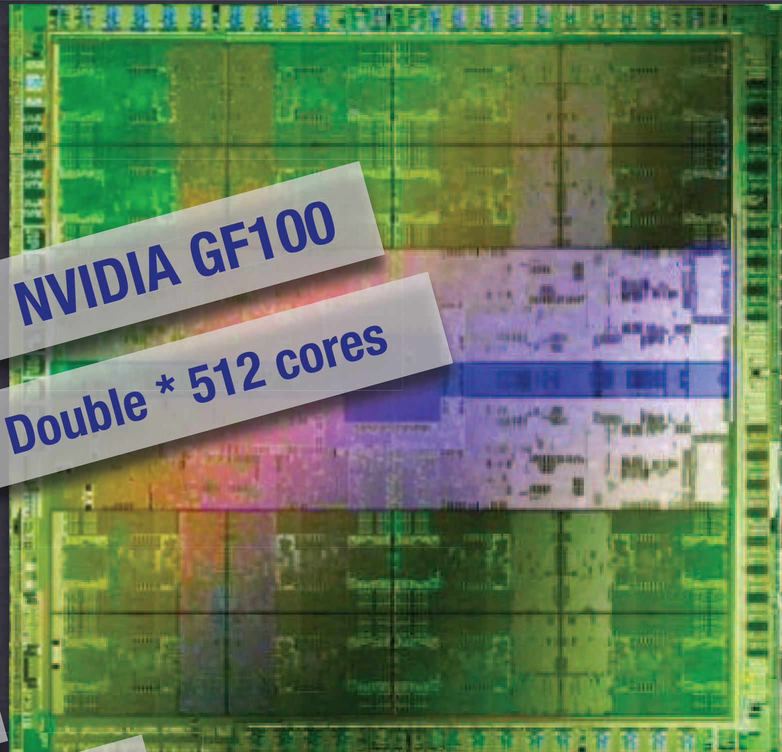
Flattening has a dual purpose

- Produce SIMD-friendly code
- Flatten nested data parallelism



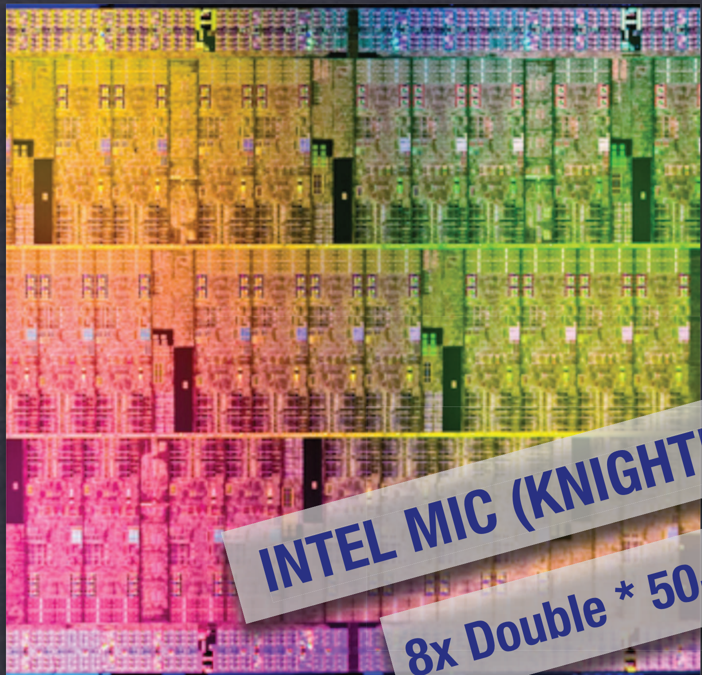
CORE I7 970

4x Double * 6 cores * 2 threads



NVIDIA GF100

32x Double * 512 cores



INTEL MIC (KNIGHT'S CORNER)

8x Double * 50+ cores * 4(?) threads

THE RENAISSANCE OF SIMD

BETTER POWER EFFICIENCY

Nested data parallelism

- Enables sparse structures & irregular parallelism
- Flat data parallelism is not modular!

```
[: myLibraryFun x | x <- xs :]
```

Is this function itself parallel?

With flat parallelism it cannot be parallel!

A simple example of flattening

```
f :: Float -> Float -> Float -> Float
f x y z = x * y + z
```

The lifted version of f

FLATTENING

```
f^ :: [:Float:] -> [:Float:] -> [:Float:]
f^ xs ys zs = xs *^ ys +^ zs
```

We call flattening also vectorisation

```
[: f x y z | x <- xs
           | y <- ys
           | z <- zs :]
```

→

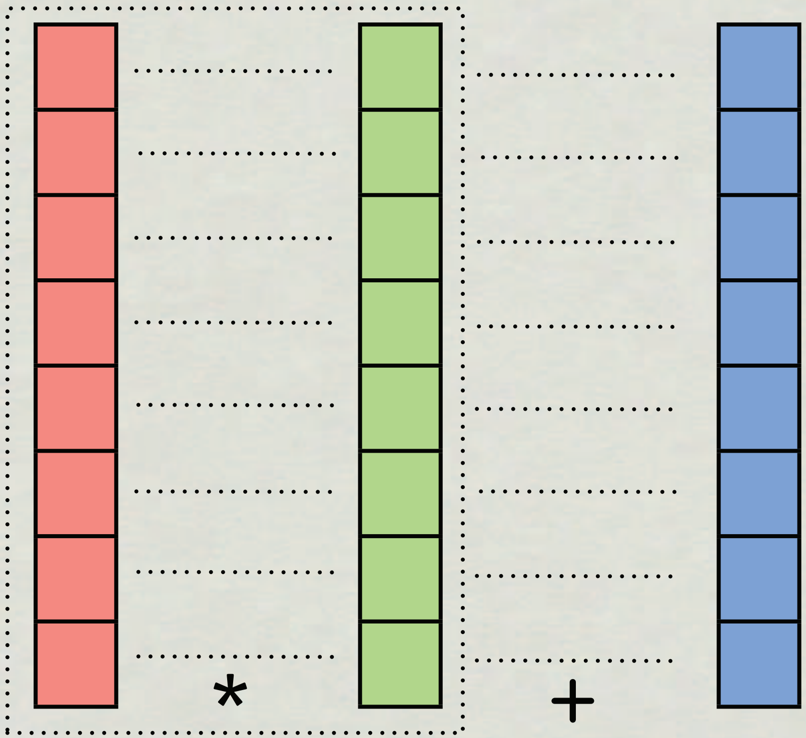
```
f^ xs ys zs
```

```
f x y z = (x * y) + z
```

```
f^ c xs ys zs = (xs *^ ys) +^ zs
```

Lifting context
of type `Int`

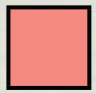
```
c = len xs
  = len ys
  = len zs
```



Shared data structures

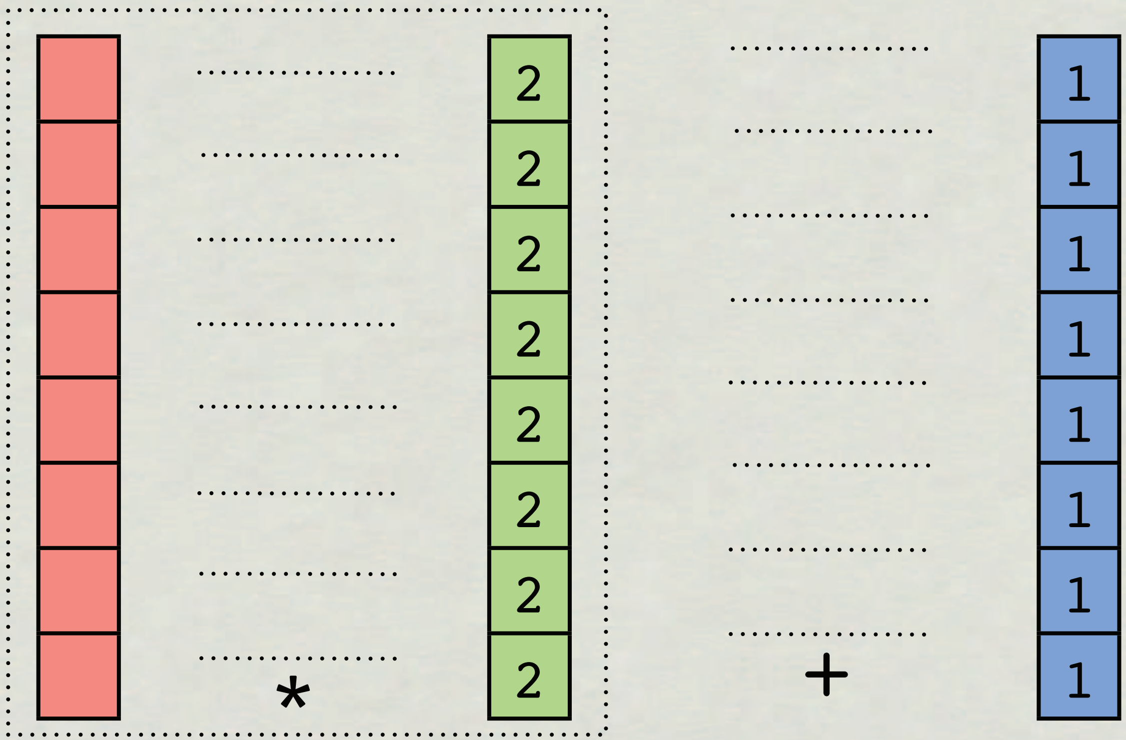
Constants shared across multiple parallel computations

```
f x = (x * 2) + 1
```



rep = replicateP

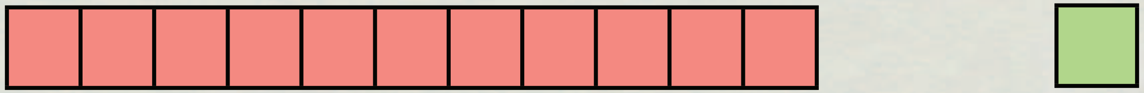
```
f^ c xs = (xs *^ rep c 2) +^ rep c 1
```



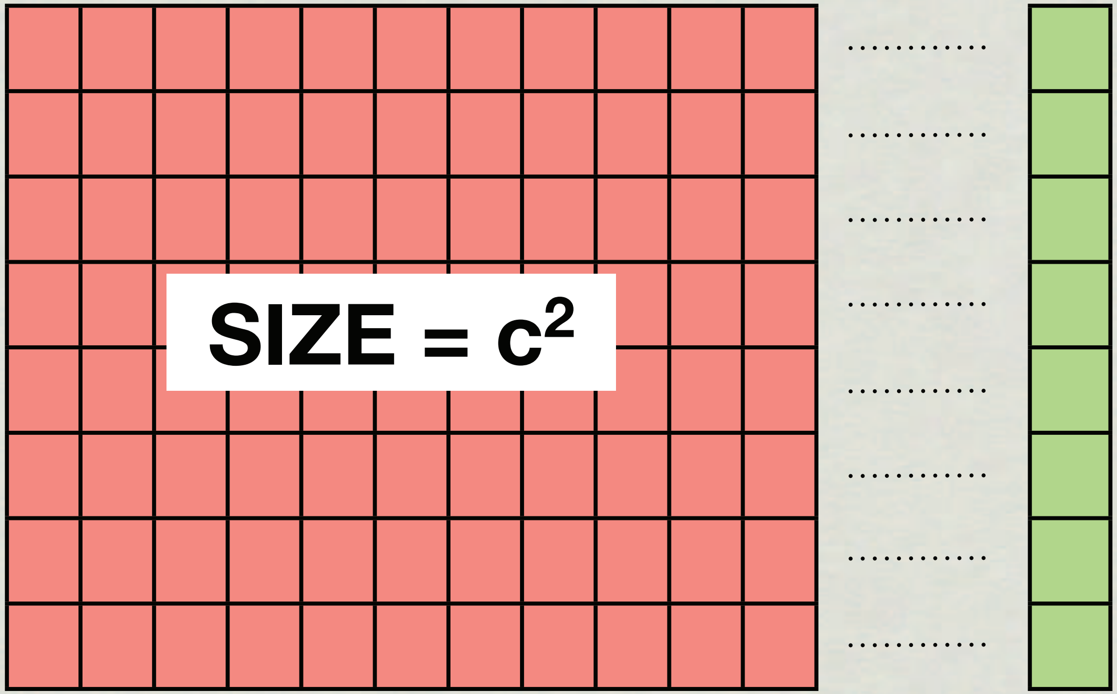
- We need to replicate constants to respect the interface of `(+^)` and `(*^)`
- The same holds for user-defined lifted functions
- Vectorisation of partial application also leads to replication

But this quickly leads to overheads!

```
xs    = [...]
f i   = xs  !  i
```



```
f^ c is = rep c xs !^ is
```



xs is replicated **c** times to extract **one element** from each copy

Sparse-matrix vector multiplication

- Realistic example program
- Suffers from sharing the multiplied vector
- Vector is replicated *n* times (for an *n*x*n* matrix)

RANDOM SPARSE MATRIX

$$
\begin{array}{c}
\begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \quad \text{index} \\
\begin{pmatrix}
0 & 2 & 1.5 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 3 & 0 \\
4 & 0 & 0 & 7 & 6.5 \\
0 & 1 & 0 & 0 & 0
\end{pmatrix}
\end{array}
$$

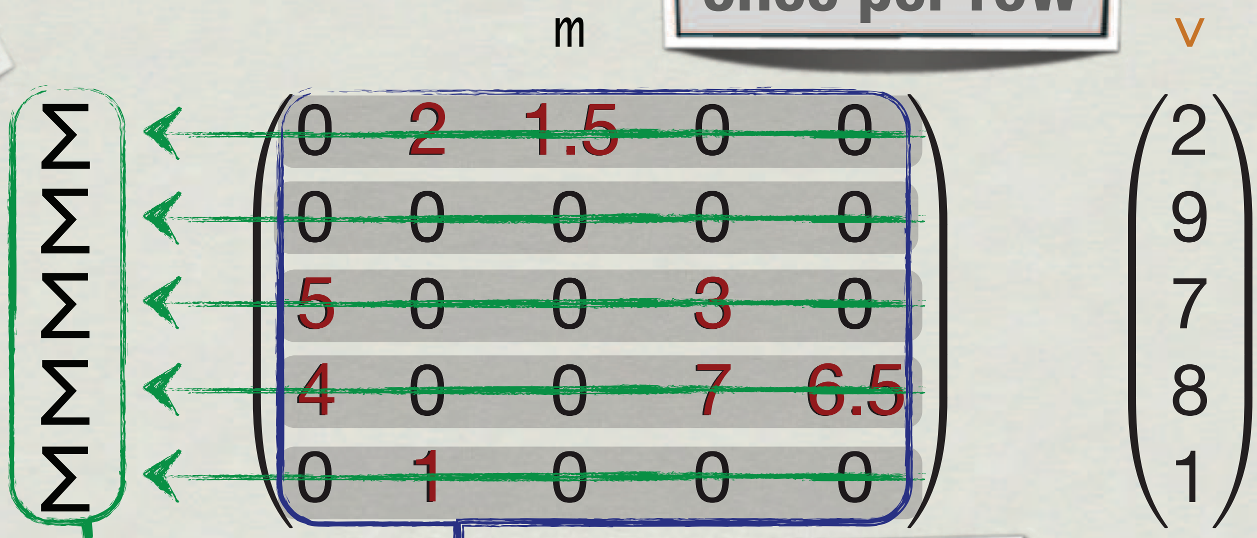
sparse vector

```
[:[:(1 , 2 ), (2, 1.5):]
 ,[::]
 ,[:(0, 5):]
 ,[:(0, 4), (3, 7), (4, 6.5):]
 ,[:(1, 1):]
                              :]
```

compressed sparse row (CSR)

IRREGULAR

Flattening replicates v once per row



all parallel folds computed in parallel

all products computed in parallel

```
smvm m v
  = [: sumP [: x * (v !: i) | (i, x) <- row :]
     | row <- m :]
```

Multiply one row against the vector

A pathological example

Assume length is
a power of 2

```
treeLookup :: [:Int:] -> [:Int:] -> [:Int:]
treeLookup table xs
  | lengthP xs == 1 = [: table !: (xs !: 0) :]
  | otherwise
  = let
      half = lengthP xs `div` 2
      xss  = segmentP [:half, half:] xs
    in
    concatP [: treeLookup table ys | ys <- xss:]
```

Equally
subdivide xs

Shared use in two
parallel invocations

SPACE EXPLOSION

NUMBER OF COPIES OF TABLE IS EXPONENTIAL IN DEPTH OF RECURSION

This problem has been known for a while

- Palmer, Prins & Westfold: *Work-Efficient Nested Data-Parallelism*
- Blelloch & Greiner: *A Provable Time and Space Efficient Implementation of NESL*
- Spoonhower, Blelloch, Harper & Gibbons: *Space Profiling for Parallel Functional Programs*

The issue is work as well as space efficiency!

First-order programs

- Palmer et al. modified the flattening transformation
- That modification doesn't extend to the higher-order case
- It also only deals with the replicate function, but omits the other issues we will identify

Thread-based approaches

- Blelloch & Greiner introduced a thread-based approach later extended by Spoonhower et al.
- Instead of flattening, they use very fine-grained threads
- In that setting, the crucial insight is to use the right scheduling policy (work stealing)

Consumers of replicate

```
f :: Int -> [:Float:]
f i = if p i then a!i else [::]
```

`a :: [:[:Float:]:]`

```
f^ :: [:Int:] -> [:[:Float:]:]
f is = let
        fs             = p^ is
        as             = replicateP^ ... a
        (as_t, as_e)   = splitP fs as
        (is_t, is_e)   = splitP fs is
        r_t            = as_t !^ is_t
      in ...
```

Expensive consumer

Our goal

- Stick with flattening to support SIMD hardware
- Avoid the space explosion
- Avoid work inefficiency
- Prove that our implementation is time and space efficient

we leave this to future work

Delaying index-space transformations

Index-space transformations

- Operations that merely re-arrange array data
- In particular those re-arranging subarrays of nested arrays

These get used in lifted array-processing functions

- replicate (segmented)
- index & slice (segmented)
- split & combine (segmented)
- append (segmented)
- back permutation

Our approach

- Delay index-space transformations
- Don't re-arrange subarrays eagerly
- Instead, keep track of pending re-arrangement

The flattening transformation stays the same!

Scattered and virtual segment descriptors

Library implements arrays generically using a data family

```
data Array (Array a) = Nested Segd (Array a)

data Segd       = Segd (Array Int) (Array Int)
```

```
xs = Array (Array Int)
xs = [[1 2 3] [8 7] [0] [9 3 9 1]]
```

```
seg lens:    [ 3 2 1 4 ]

seg starts: [ 0 3 5 6 ]
```

flat data:

1	2	3	8	7	0	9	3	9	3
0	1	2	3	4	5	6	7	8	9

Library implements arrays generically using a data family

```
data Array (Array a) = Nested Segd (Array a)
data Segd            = Segd (Array Int) (Array Int)
```

```
xs = Array (Array Int)
xs = [[1 2 3] [8 7] [0] [9 3 9 1]]
```

```
seg lens:    [ 3 2 1 4 ]
seg starts:  [ 0 3 5 6 ]

flat data:     1  2  3  8  7  0  9  3  9  3
```

```
data Array (Array a) = Nested Segd (Array a)
data Segd        = Segd (Array Int) (Array Int)
```

```
xs = Array (Array (Array Int))
xs = [[[1 2 3][8 7]] [] [[0][9 3 9 1]]]
```

```
seg lens:   [ 3 2 1 4 ]   seg lens:   [ 2 0 2 ]
seg starts: [ 0 3 5 6 ]   seg starts: [ 0 5 5 ]
```

```
flat data:    1  2  3  8  7  0  9  3  9  3
```

Virtual segments

Basic idea

- Don't copy data, keep track of repetition counts

```
replicateP 80000 [:0..89999:]
     = [:[:0..89999:] [:0..89999:] ...:]

rep count:  80000
seg lens:   [: 90000 :]
seg starts: [: 0 :]

flat data:  1 2 ⋯ 89999
```

Lifted replicate

```
replicateP^ :: [:Int:] -> [:a:] -> [:[:a:]:]

replicateP^ [:2 3 1:] [:xs ys zs:]
   = [:xs xs ys ys ys zs:]
```

```
virt seg ids:     [:0 0 1 1 1 2:]
phys seg lens:    [:5 3 8:]
phys seg starts: [:0 5 8:]
```

used physical segments

flat data:

x1	x2	x3	x4	x5	y1	y2	y3	z1	z2	z3	z4	z5	z6	z7	z8
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Consumers: Packing

```
pack :: [:Bool:] -> [:a:] -> [:a:]
pack [:T F T T F:] [:as bs cs ds es:]
   = [:as cs ds:]
```

```
virt seg ids:     [:0  1  2  3  4:]
phys seg lens:    [:3  2  5  1  5:]
phys seg starts:  [:0  3  5  10 11:]
```

flat data:

a1	a2	a3	b1	b2	c1	c2	c3	c4	c5	d1	e1	e2	e3	e4	e5
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Consumers: Packing

```
pack :: [:Bool:] -> [:a:] -> [:a:]

pack [:T F T T F:] [:as bs cs ds es:]
   = [:as cs ds:]
```

```
                      [:T  T  T:]
virt seg ids:         [:0  2  3:]
phys seg lens:        [:3  2  5  1  5:]
phys seg starts:      [:0  3  5  10 11:]
```

flat data:

a1	a2	a3	b1	b2	c1	c2	c3	c4	c5	d1	e1	e2	e3	e4	e5
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Consumers: Packing

```
pack :: [:Bool:] -> [:a:] -> [:a:]
pack [:T F T T F:] [:as bs cs ds es:]
   = [:as cs ds:]
```

```
                     [:T  T  T:]
virt seg ids:        [:0  1  2:]
phys seg lens:       [:3  5  1:]
phys seg starts:     [:0  5  10:]
```

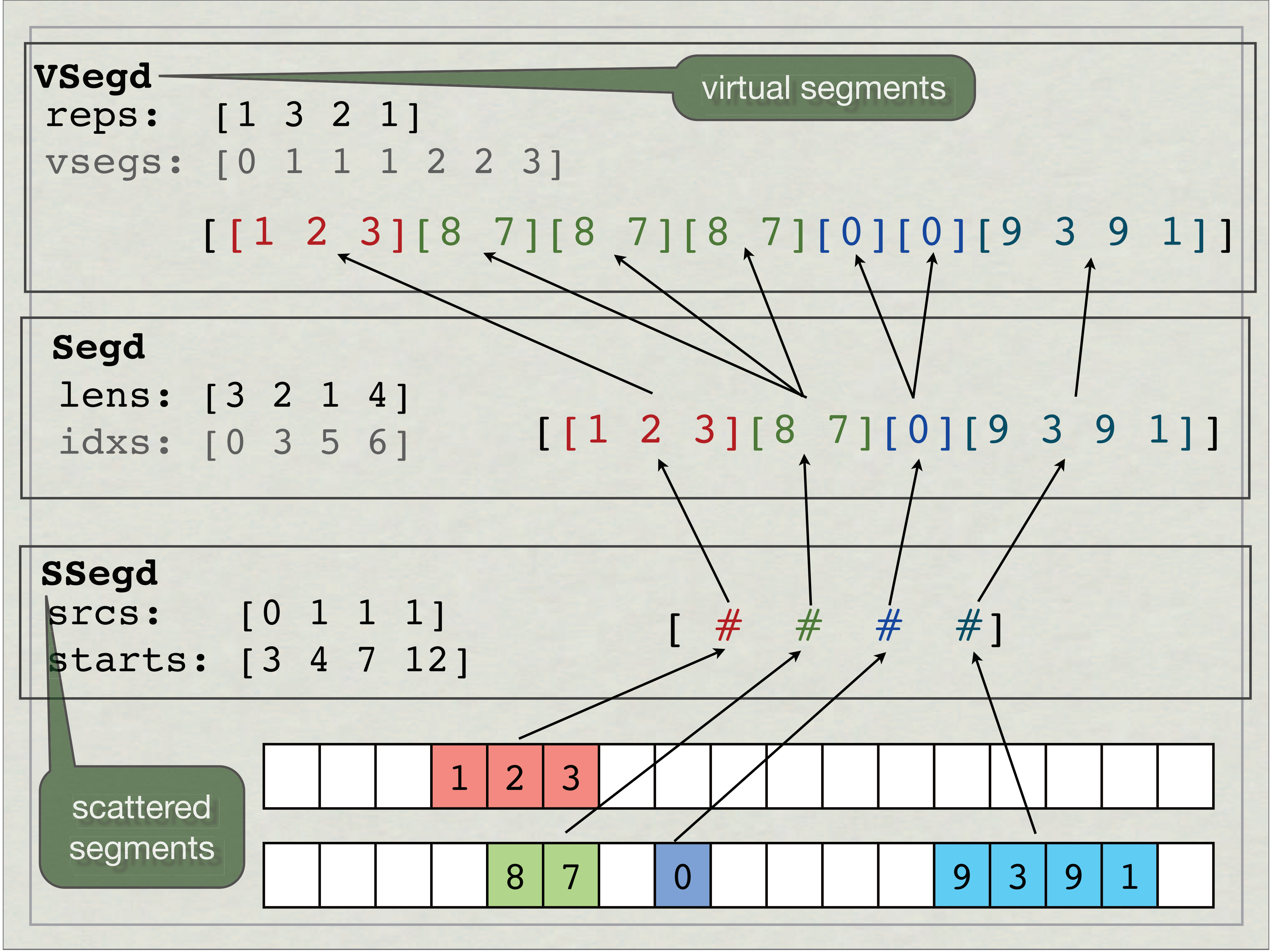
flat data:

a1	a2	a3	b1	b2	c1	c2	c3	c4	c5	d1	e1	e2	e3	e4	e5
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Scattered segments

- replicate (segmented)
- index & slice (segmented)
- split & combine (segmented)
- append (segmented)
- back permutation

assemble a
segmented array from
(two) others


VSegd
virtual segments
reps: [1 3 2 1]
vsegs: [0 1 1 1 2 2 3]
[[1 2 3][8 7][8 7][8 7][0][0][9 3 9 1]]
Segd
lens: [3 2 1 4]
idxs: [0 3 5 6]
[[1 2 3][8 7][0][9 3 9 1]]
SSegd
srcs: [0 1 1 1]
starts: [3 4 7 12]
[# # # #]
scattered segments
1 2 3
8 7 0 9 3 9 1

Summary

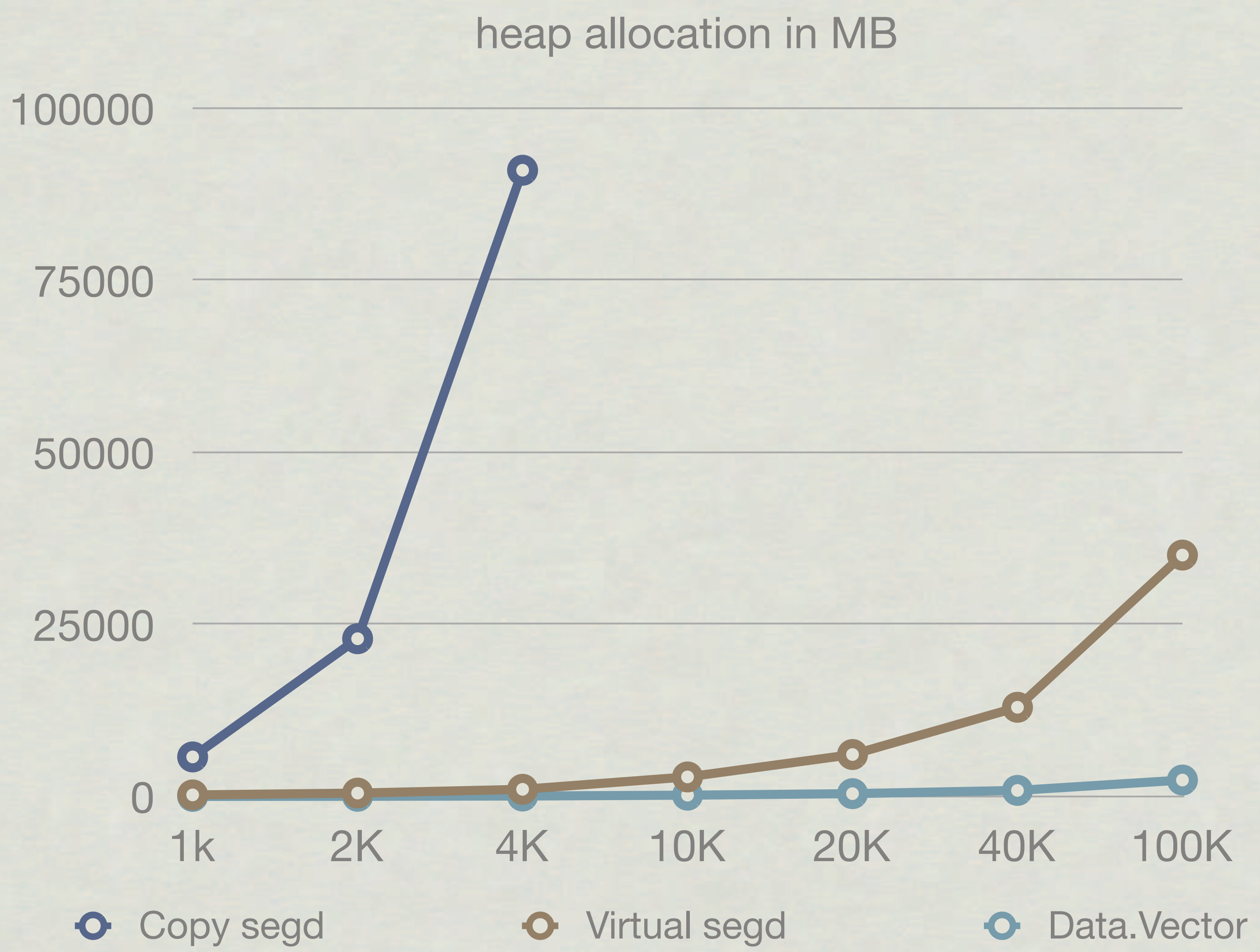
- **Virtual segments:** encoding repetition
- **Sparse virtual segments:** encoding packing
- **Scattered segments:** encoding combinations of multiple subarrays

Benchmarks

Implementation status

- Implemented DPH library with scattered and virtual segment descriptors
- Basic implementation that still misses some important optimisations
- It runs all our test and example programs
- Will be available with GHC 7.4.1

Barnes Hut


heap allocation in MB
100000
75000
50000
25000
0
1k
2K
4K
10K
20K
40K
100K
Copy segd
Virtual segd
Data.Vector

As Ben put it,

"we've made it to the ball park, but haven't yet stepped on the field..."

Conclusions

✓ **preserve work & space complexity**

- With flattening, shared data structures need special treatment
- Delay index-space transformations; leave flattening as it is
- More on Data Parallel Haskell:

`http://haskell.org/haskellwiki/GHC/Data_Parallel_Haskell`