Nested parallelism in Accelerate

Robert Clifton-Everest
University of New South Wales
robertce@cse.unsw.edu.au

GPUs

GPUs

- Lots of raw computing power
  - This one: 2688 cores @ 867 MHz

GPUs

- Lots of raw computing power
  - This one: 2688 cores @ 867 MHz
- Different hardware design
  - Limited instruction set
  - SIMD: Cores run the same program, but on different data

GPUs

- Lots of raw computing power
  - This one: 2688 cores @ 867 MHz
- Different hardware design
  - Limited instruction set
  - SIMD: Cores run the same program, but on different data
- How can we take advantage of this power?

GPUs

- Lots of raw computing power
  - This one: 2688 cores @ 867 MHz
- Different hardware design
  - Limited instruction set
  - SIMD: Cores run the same program, but on different data
- How can we take advantage of this power?

With a high-level embedded language of course!

Accelerate

Accelerate

- Data parallel language operating over multi-dimensional arrays

Accelerate

- Data parallel language operating over multi-dimensional arrays

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

Accelerate

- Data parallel language operating over multi-dimensional arrays

```
type Vector e = Array (Z:.Int) e
```

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

Accelerate

- Data parallel language operating over multi-dimensional arrays

```
type Vector e = Array (Z:.Int) e
```

```
type Scalar e = Array Z e
```

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

Accelerate

- Data parallel language operating over multi-dimensional arrays

```
type Vector e = Array (Z:.Int) e
```

```
type Scalar e = Array Z e
```

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

```
zipWith :: (Exp a -> Exp b -> Exp c)
        -> Acc (Array sh a)
        -> Acc (Array sh b)
        -> Acc (Array sh c)
```

Accelerate

- Data parallel language operating over multi-dimensional arrays

```
type Vector e = Array (Z:.Int) e
```

```
type Scalar e = Array Z e
```

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

```
fold :: (Exp e -> Exp e -> Exp e)
     -> Exp e
     -> Acc (Array (sh:.Int) e)
     -> Acc (Array sh e)
```

```
zipWith :: (Exp a -> Exp b -> Exp c)
        -> Acc (Array sh a)
        -> Acc (Array sh b)
        -> Acc (Array sh c)
```

A (slightly) more complex example

- Matrix-vector multiplication.

A (slightly) more complex example

- Matrix-vector multiplication.

- In terms of `dotp`?

A (slightly) more complex example

- Matrix-vector multiplication.

- In terms of `dotp`?

```
mvm :: Acc (Array (Z:.Int:.Int) Float)
    -> Acc (Vector Float)
    -> Acc (Vector Float)
mvm mat vec = generate (index1 (height mat))
                       (λi -> the (dotp vec (getRow i mat)))
```

A (slightly) more complex example

- Matrix-vector multiplication.

- In terms of `dotp`?

```
index1 :: Exp Int -> Exp (Z:.Int)
```

```
mvm :: Acc (Array (Z:.Int:.Int) Float)
    -> Acc (Vector Float)
    -> Acc (Vector Float)
mvm mat vec = generate (index1 (height mat))
                       (λi -> the (dotp vec (getRow i mat)))
```

A (slightly) more complex example

- Matrix-vector multiplication.

- In terms of `dotp`?

```
index1 :: Exp Int -> Exp (Z:.Int)
```

```
mvm :: Acc (Array (Z:.Int:.Int) Float)
    -> Acc (Vector Float)
    -> Acc (Vector Float)
mvm mat vec = generate (index1 (height mat))
                       (λi -> the (dotp vec (getRow i mat)))
```

```
the :: Acc (Scalar e) -> Exp e
```

A (slightly) more complex example

- Matrix-vector multiplication.
- In terms of `dotp`?

```
index1 :: Exp Int -> Exp (Z:.Int)
```

```
mvm :: Acc (Array (Z:.Int:.Int) Float)
    -> Acc (Vector Float)
    -> Acc (Vector Float)
mvm mat vec = generate (index1 (height mat))
                       (λi -> the (dotp vec (getRow i mat)))
```

```
the :: Acc (Scalar e) -> Exp e
```

```
*** Exception: Cyclic definition of a value of type 'Exp' (sa = 46)
```

A (slightly) more complex example

- Matrix-vector multiplication.

- In terms of `dotp`?

```
index1 :: Exp Int -> Exp (Z:.Int)
```

```
mvm :: Acc (Array (Z:.Int:.Int) Float)
    -> Acc (Vector Float)
    -> Acc (Vector Float)
mvm mat vec = generate (index1 (height mat))
                       (λi -> the (dotp vec (getRow i mat)))
```

```
the :: Acc (Scalar e) -> Exp e
```

```
*** Exception: Cyclic ... type 'Exp' (sa = 46)
```

Nested parallelism

- The “right” way

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

1	2	3	4	5

```
mvm :: Acc (Array (Z:.Int:.Int) Float)
    -> Acc (Vector Float)
    -> Acc (Vector Float)
mvm mat vec
  =
```

- The “right” way

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5

```
mvm :: Acc (Array (Z:.Int:.Int) Float)
    -> Acc (Vector Float)
    -> Acc (Vector Float)
mvm mat vec
  =                             replicateRow (height mat) vec
```

- The “right” way

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

×

1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5

```
mvm :: Acc (Array (Z:.Int:.Int) Float)
    -> Acc (Vector Float)
    -> Acc (Vector Float)
mvm mat vec
  =             zipWith (*) mat (replicateRow (height mat) vec)
```

- The “right” way

1	4	9	16	25
6	14	24	36	50
11	24	29	56	75
16	34	54	76	100

```
mvm :: Acc (Array (Z:.Int:.Int) Float)
    -> Acc (Vector Float)
    -> Acc (Vector Float)
mvm mat vec
  =              zipWith (*) mat (replicateRow (height mat) vec)
```

- The “right” way

1	4	9	16	25
6	14	24	36	50
11	24	29	56	75
16	34	54	76	100

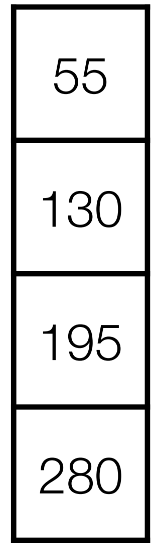
```
mvm :: Acc (Array (Z:.Int:.Int) Float)
    -> Acc (Vector Float)
    -> Acc (Vector Float)
mvm mat vec
  = fold (+) 0 (zipWith (*) mat (replicateRow (height mat) vec))
```

- The “right” way

55
130
195
280

```
mvm :: Acc (Array (Z:.Int:.Int) Float)
    -> Acc (Vector Float)
    -> Acc (Vector Float)
mvm mat vec
  = fold (+) 0 (zipWith (*) mat (replicateRow (height mat) vec))
```

- The “right” way




```
mvm :: Acc (Array (Z:.Int:.Int) Float)
    -> Acc (Vector Float)
    -> Acc (Vector Float)
mvm mat vec
  = fold (+) 0 (zipWith (*) mat (replicateRow (height mat) vec))
```

We want to write the first version

Enabling nested parallelism

Enabling nested parallelism

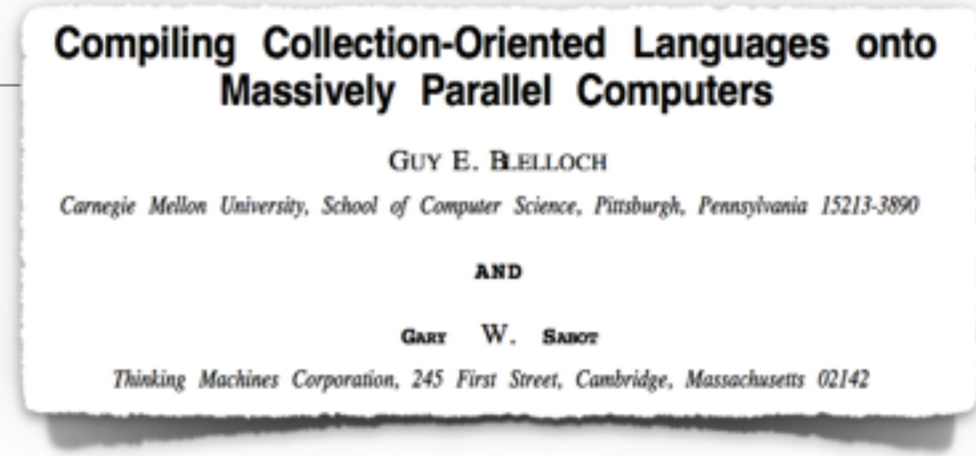
- Vectorisation

Enabling nested parallelism

- Vectorisation
  - First described by Blelloch and Sabot

Enabling nested parallelism

- Vectorisation
  - First described by Blelloch and Sabot



Compiling Collection-Oriented Languages onto Massively Parallel Computers

GUY E. BLELLOCH

Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890

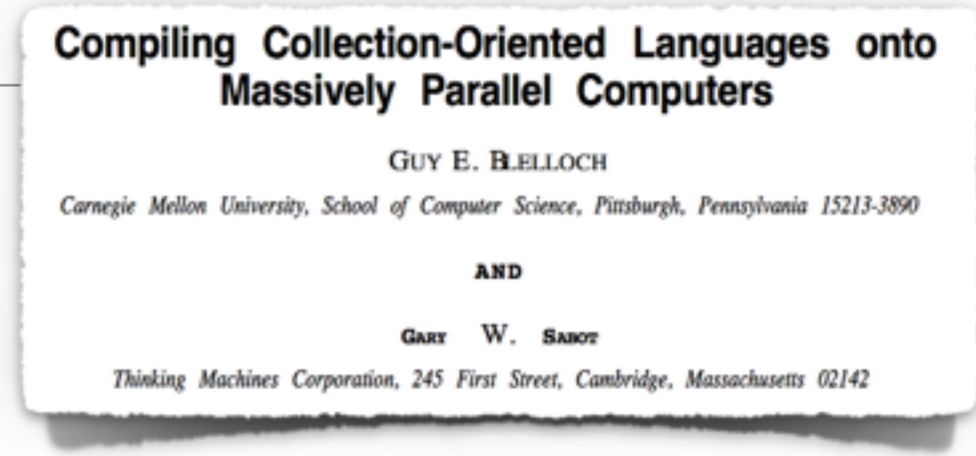
AND

GARY W. SABOT

Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142

Enabling nested parallelism

- Vectorisation
  - First described by Blelloch and Sabot
  - Converts a nested parallel program into a flat parallel program



Compiling Collection-Oriented Languages onto Massively Parallel Computers

GUY E. BLELLOCH

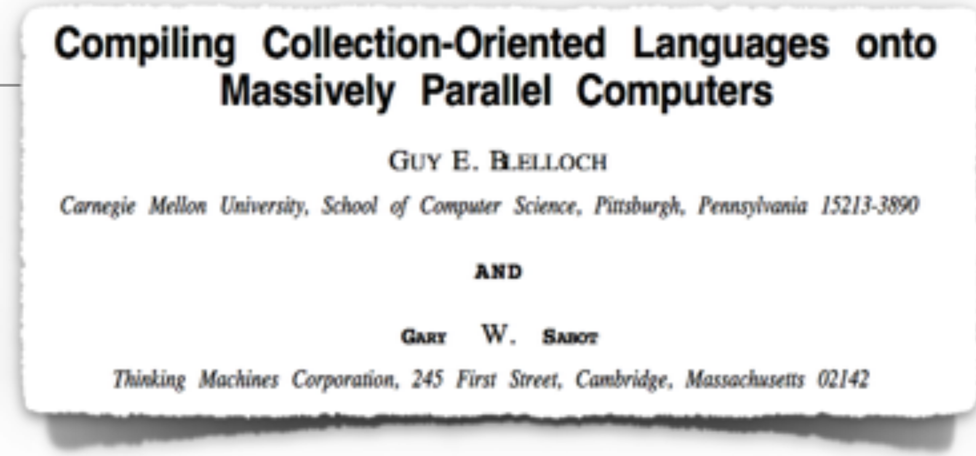
Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890

AND

GARY W. SABOT

Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142

Enabling nested parallelism



Compiling Collection-Oriented Languages onto Massively Parallel Computers

GUY E. BLELLOCH

Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890

AND

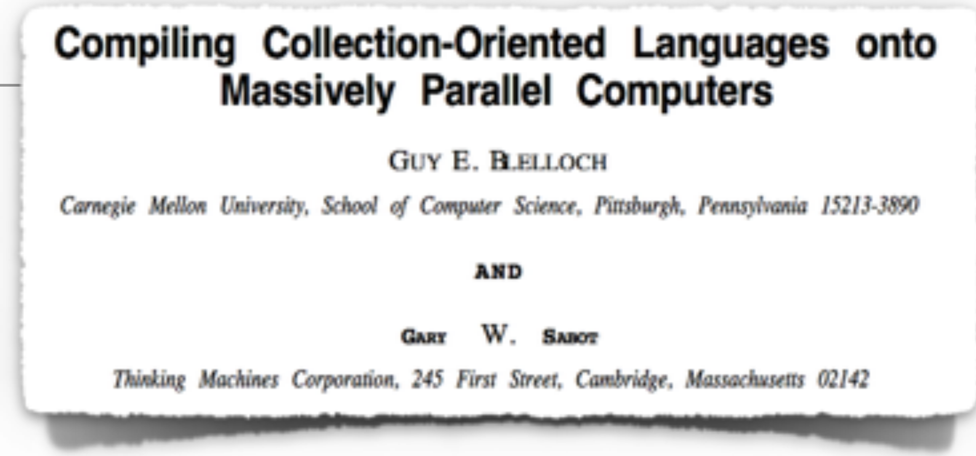
GARY W. SABOT

Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142

- Vectorisation
  - First described by Blelloch and Sabot
  - Converts a nested parallel program into a flat parallel program
  - Programs must be pure, no side effects, no destructive updates, etc.

Enabling nested parallelism

- Vectorisation
  - First described by Blelloch and Sabot
  - Converts a nested parallel program into a flat parallel program
  - Programs must be pure, no side effects, no destructive updates, etc.
  - Simple, but naive



Compiling Collection-Oriented Languages onto Massively Parallel Computers

GUY E. BLELLOCH

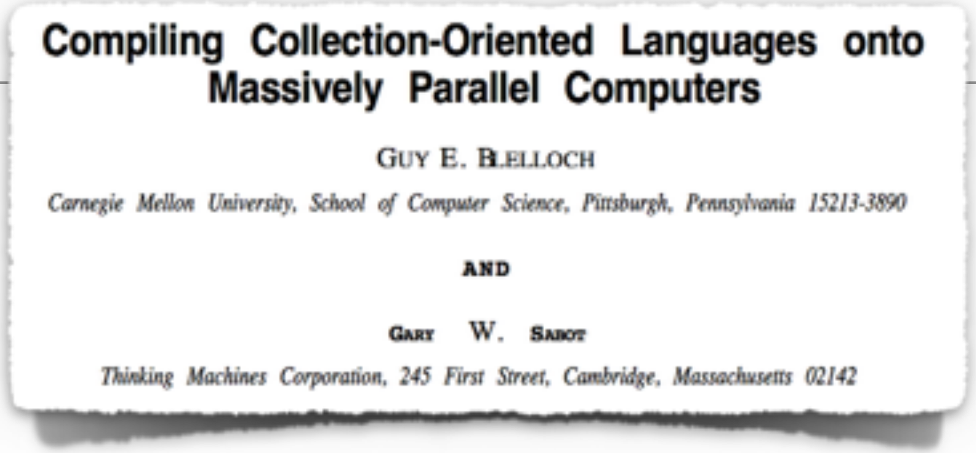
Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890

AND

GARY W. SABOT

Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142

Enabling nested parallelism



Compiling Collection-Oriented Languages onto Massively Parallel Computers

GUY E. BLELLOCH

Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890

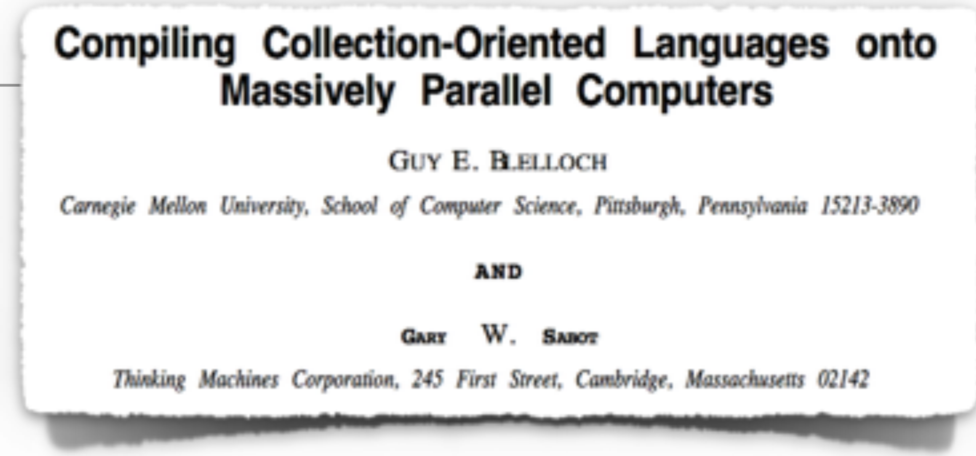
AND

GARY W. SABOT

Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142

- Vectorisation
  - First described by Blelloch and Sabot
  - Converts a nested parallel program into a flat parallel program
  - Programs must be pure, no side effects, no destructive updates, etc.
  - Simple, but naive
  - Relies heavily on subsequent optimisations

Enabling nested parallelism


Compiling Collection-Oriented Languages onto Massively Parallel Computers

GUY E. BLELLOCH

Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890

AND

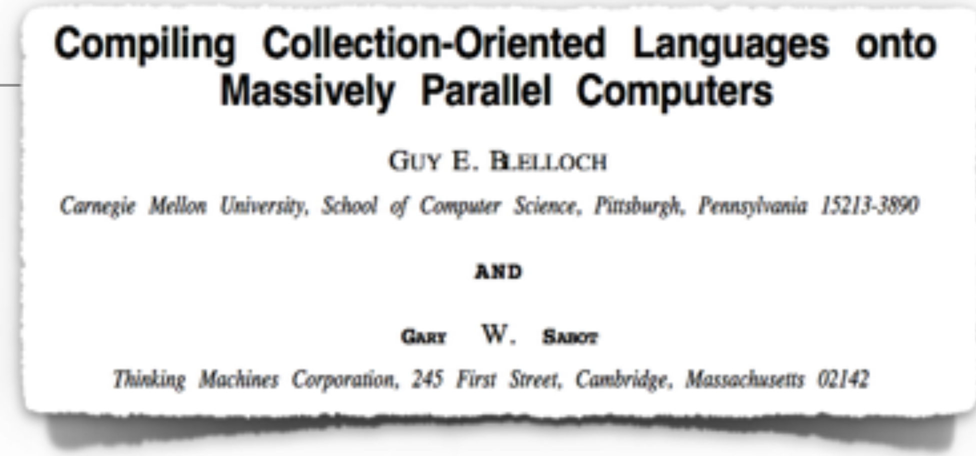
GARY W. SABOT

Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142

- Vectorisation
  - First described by Blelloch and Sabot
  - Converts a nested parallel program into a flat parallel program
  - Programs must be pure, no side effects, no destructive updates, etc.
  - Simple, but naive
  - Relies heavily on subsequent optimisations
  - Lots of work done to improve this

Enabling nested parallelism

- Vectorisation
  - First described by Blelloch and Sabot



Compiling Collection-Oriented Languages onto Massively Parallel Computers

GUY E. BLELLOCH

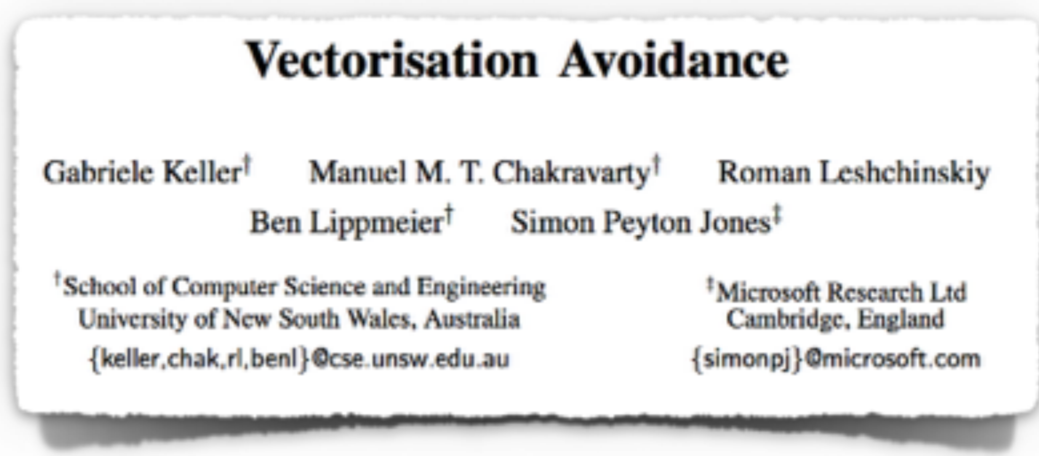
Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890

AND

GARY W. SABOT

Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142

  - Converts a nested parallel program into a flat parallel program
  - Programs must be pure, no side effects, no destructive updates, etc.
  - Simple, but naive
  - Relies heavily on subsequent optimisations
  - Lots of work done to improve this



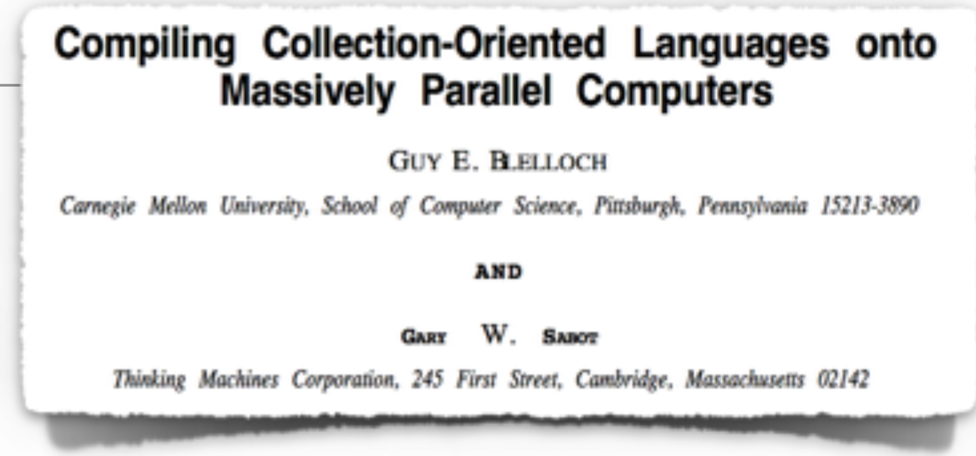
Vectorisation Avoidance

Gabriele Keller[†] Manuel M. T. Chakravarty[†] Roman Leshchinskiy
Ben Lippmeier[†] Simon Peyton Jones[‡]

[†]School of Computer Science and Engineering
University of New South Wales, Australia
{keller,chak,rl,benl}@cse.unsw.edu.au

[‡]Microsoft Research Ltd
Cambridge, England
{simonpj}@microsoft.com

Enabling nested parallelism



Compiling Collection-Oriented Languages onto Massively Parallel Computers

GUY E. BLELLOCH

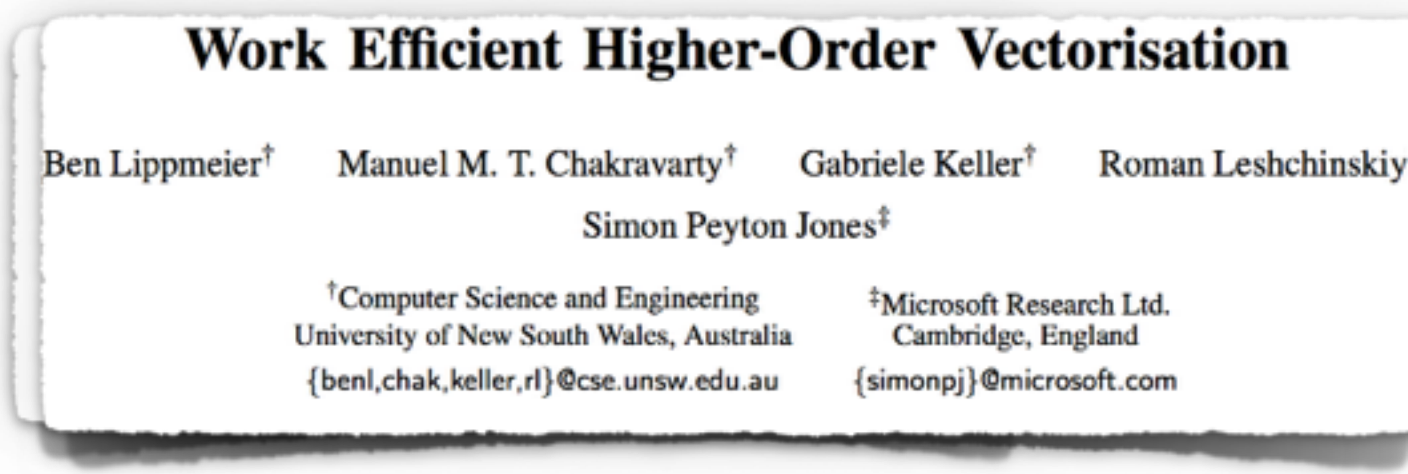
Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890

AND

GARY W. SABOT

Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142

- Vectorisation
  - First described by Blelloch and Sabot
  - Converts a nested parallel program into a flat parallel program
  - Programs must be pure, no side effects, no destructive updates, etc.
  - Simple, but naive
  - Relies heavily on subsequent optimisations
  - Lots of work done to improve this



Work Efficient Higher-Order Vectorisation

Ben Lippmeier[†] Manuel M. T. Chakravarty[†] Gabriele Keller[†] Roman Leshchinskiy

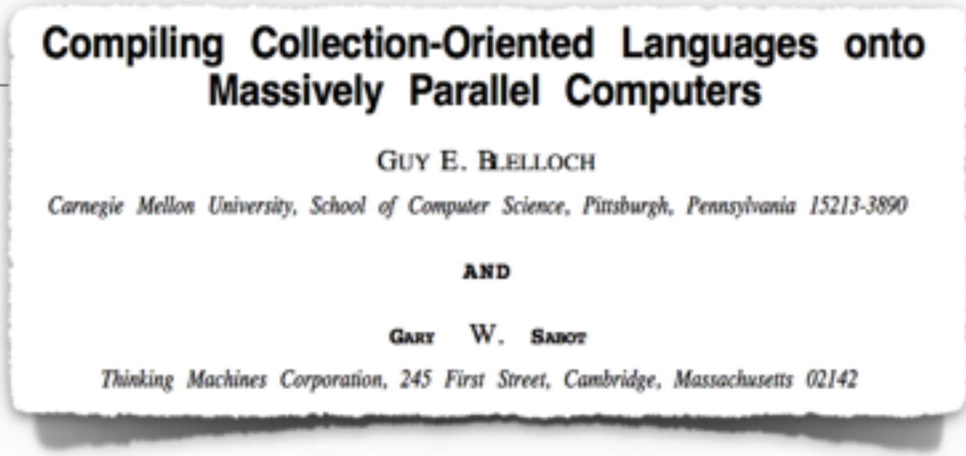
Simon Peyton Jones[‡]

[†]Computer Science and Engineering
University of New South Wales, Australia
{benl,chak,keller,rl}@cse.unsw.edu.au

[‡]Microsoft Research Ltd.
Cambridge, England
{simonpj}@microsoft.com

Enabling nested parallelism

- Vectorisation
  - First described by Blelloch and Sabot
  - Converts a nested parallel program into a flat parallel program
  - Programs must be pure, no side effects, no destructive updates, etc.
  - Simple, but naive
  - Relies heavily on subsequent optimisations
  - Lots of work done to improve this



Compiling Collection-Oriented Languages onto Massively Parallel Computers

GUY E. BLELLOCH

Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania 15213-3890

AND

GARY W. SABOT

Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142

Data Flow Fusion with Series Expressions in Haskell

Ben Lippmeier[†] Manuel M. T. Chakravarty[†] Gabriele Keller[†] Amos Robinson[†]

[†]Computer Science and Engineering
University of New South Wales, Australia
{benl,chak,keller,amosr}@cse.unsw.edu.au

The lifting transformation

```
foo     :: Int -> Float -> Float
```

$\mathcal{L}_n[\![$`foo`$]\!]$ `:: Vector Int -> Vector Float -> Vector Float`

The lifting transformation

```
foo     :: Int -> Float -> Float
```

$\mathcal{L}_n[\![$`foo`$]\!]$ `:: Vector Int -> Vector Float -> Vector Float`

The expression being transformed

The lifting transformation

```
foo        :: Int -> Float -> Float
```

$\mathcal{L}_n$⟦foo⟧ :: Vector Int -> Vector Float -> Vector Float

The size parameter

The expression being transformed

The lifting transformation

```
foo      :: Int -> Float -> Float
```

$\mathcal{L}_n[\![\texttt{foo}]\!]$ `:: Vector Int -> Vector Float -> Vector Float`

The size parameter

The expression being transformed

$\mathcal{L}_n$

(Where c is a constant)

The lifting transformation

```
foo       :: Int -> Float -> Float
```

$\mathcal{L}_n[\![\texttt{foo}]\!]$ `:: Vector Int -> Vector Float -> Vector Float`

The size parameter

The expression being transformed

$\mathcal{L}_n$ (Where c is a constant)

$\mathcal{L}_n$ (Where x is not a lifted variable)

The lifting transformation

```
foo          :: Int -> Float -> Float
```

$\mathcal{L}_n$⟦foo⟧ `:: Vector Int -> Vector Float -> Vector Float`

The size parameter

The expression being transformed

$\mathcal{L}_n$ (Where c is a constant)

$\mathcal{L}_n$ (Where x is not a lifted variable)

$\mathcal{L}_n$ (Where x is a lifted variable)

The lifting transformation

```
foo       :: Int -> Float -> Float
```

$\mathcal{L}_n$⟦foo⟧ `:: Vector Int -> Vector Float -> Vector Float`

The size parameter

The expression being transformed

$\mathcal{L}_n$ (Where c is a constant)

$\mathcal{L}_n$ (Where x is not a lifted variable)

$\mathcal{L}_n$ (Where x is a lifted variable)

$\mathcal{L}_n$

The lifting transformation

```
foo      :: Int -> Float -> Float
```

$\mathcal{L}_n$⟦`foo`⟧ `:: Vector Int -> Vector Float -> Vector Float`

The size parameter

The expression being transformed

$\mathcal{L}_n$ (Where c is a constant)

$\mathcal{L}_n$ (Where x is not a lifted variable)

$\mathcal{L}_n$ (Where x is a lifted variable)

$\mathcal{L}_n$

$\mathcal{L}_n$

The lifting transformation

$$\texttt{foo} \quad \texttt{:: Int -> Float -> Float}$$

$$\mathcal{L}_n[\![\texttt{foo}]\!] \quad \texttt{:: Vector Int -> Vector Float -> Vector Float}$$

The size parameter

The expression being transformed

$\mathcal{L}_n$ (Where c is a constant)

$\mathcal{L}_n$ (Where x is not a lifted variable)

$\mathcal{L}_n$ (Where x is a lifted variable)

$\mathcal{L}_n$

$\mathcal{L}_n$

$\mathcal{L}_n$ (Where
lifted equivalent)

The lifting transformation

```
bar :: Int -> Int
bar = λx. 2*x + 1
```

The lifting transformation

```
bar :: Int -> Int
bar = λx. 2*x + 1
```

$\mathcal{L}_n[\![\texttt{bar}]\!]$ `:: Vector Int -> Vector Int`

$\mathcal{L}_n[\![\texttt{bar}]\!]$ `= λx. (replicate (length x) 2)` $*^{\uparrow}$ `x` $+^{\uparrow}$ `(replicate (length x) 1)`

The lifting transformation

```
bar :: Int -> Int
bar = λx. 2*x + 1
```

$\mathcal{L}_n[\![\texttt{bar}]\!]$ `:: Vector Int -> Vector Int`

$\mathcal{L}_n[\![\texttt{bar}]\!]$ `= λx. (replicate (length x) 2)` $*^{\uparrow}$ `x` $+^{\uparrow}$ `(replicate (length x) 1)`

What about vector functions?

The lifting transformation

```
bar :: Int -> Int
bar = λx. 2*x + 1
```

$\mathcal{L}_n[\![\text{bar}]\!]$ `:: Vector Int -> Vector Int`

$\mathcal{L}_n[\![\text{bar}]\!]$ `= λx. (replicate (length x) 2)` $*^{\uparrow}$ `x` $+^{\uparrow}$ `(replicate (length x) 1)`

What about vector functions?

```
sum :: Vector Int -> Int
```

The lifting transformation

```
bar :: Int -> Int
bar = λx. 2*x + 1
```

$\mathcal{L}_n$⟦bar⟧ `:: Vector Int -> Vector Int`

$\mathcal{L}_n$⟦bar⟧ `= λx. (replicate (length x) 2)` $*^{\uparrow}$ `x` $+^{\uparrow}$ `(replicate (length x) 1)`

What about vector functions?

```
sum :: Vector Int -> Int
```

$\mathcal{L}_n$⟦sum⟧ `:: Vector (Vector Int) -> Vector Int`

The lifting transformation

```
bar :: Int -> Int
bar = λx. 2*x + 1
```

$\mathcal{L}_n[\![$`bar`$]\!]$ `:: Vector Int -> Vector Int`

$\mathcal{L}_n[\![$`bar`$]\!]$ `= λx. (replicate (length x) 2)` $*^{\uparrow}$ `x` $+^{\uparrow}$ `(replicate (length x) 1)`

What about vector functions?

```
sum :: Vector Int -> Int
```

$\mathcal{L}_n[\![$`sum`$]\!]$ `:: Vector (Vector Int) -> Vector Int`

Nested vectors

Nested vectors

Nested vectors

- Vectors of pointers? Grossly inefficient.

Nested vectors

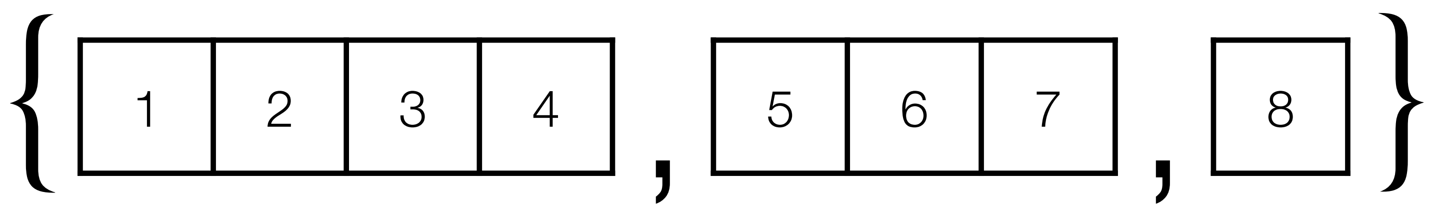
- Vectors of pointers? Grossly inefficient.

- Blelloch's solution

Nested vectors

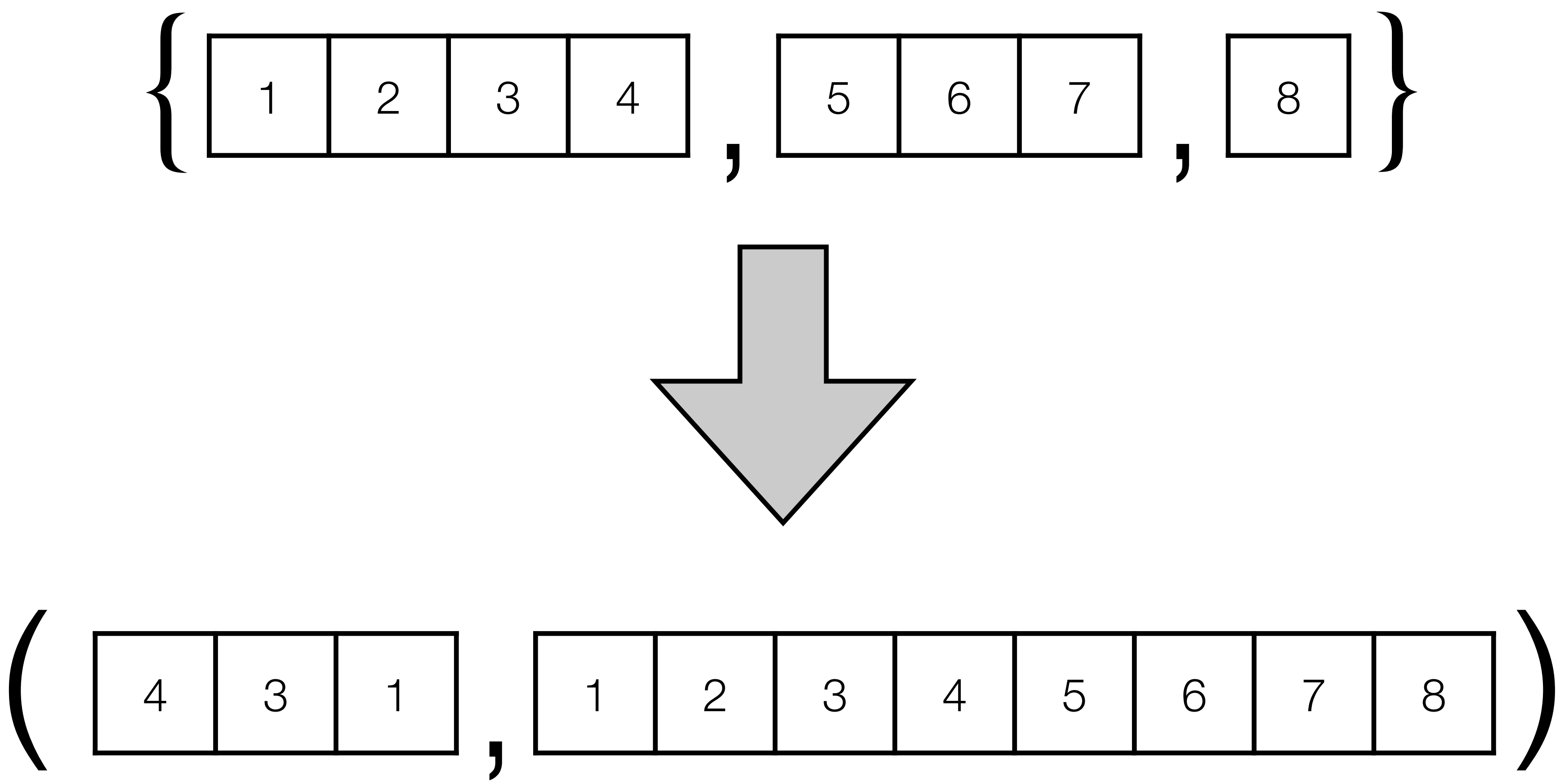
- Vectors of pointers? Grossly inefficient.

- Blelloch's solution



Nested vectors

- Vectors of pointers? Grossly inefficient.
- Blelloch's solution



Vectorisation with multidimensional arrays

- Will this solution work with multidimensional arrays?

Vectorisation with multidimensional arrays

- Will this solution work with multidimensional arrays?

- Does it now require this?

```
foo        :: Int -> Float -> Float
𝓛n⟦foo⟧ :: Array sh Int -> Array sh Float -> Array sh Float
```

Vectorisation with multidimensional arrays

- Will this solution work with multidimensional arrays?

- Does it now require this?

```
foo     :: Int -> Float -> Float
ℒn⟦foo⟧ :: Array sh Int -> Array sh Float -> Array sh Float
```

- No, lifting to vectors is sufficient

Vectorisation with multidimensional arrays

- Will this solution work with multidimensional arrays?

- Does it now require this?

```
foo     :: Int -> Float -> Float
𝓛n⟦foo⟧ :: Array sh Int -> Array sh Float -> Array sh Float
```

- No, lifting to vectors is sufficient
  - At the machine level it's all vectors anyway

Vectorisation with multidimensional arrays

- Will this solution work with multidimensional arrays?
- Does it now require this?

```
foo      :: Int -> Float -> Float
𝓛n⟦foo⟧ :: Array sh Int -> Array sh Float -> Array sh Float
```

- No, lifting to vectors is sufficient
  - At the machine level it's all vectors anyway
- What about nested arrays?

Vectorisation with multidimensional arrays

- Will this solution work with multidimensional arrays?

- Does it now require this?

```
foo       :: Int -> Float -> Float
𝓛n⟦foo⟧ :: Array sh Int -> Array sh Float -> Array sh Float
```

- No, lifting to vectors is sufficient
    - At the machine level it's all vectors anyway

- What about nested arrays?
    - Unsupported by accelerate

Vectorisation with multidimensional arrays

- Will this solution work with multidimensional arrays?

- Does it now require this?

```
foo       :: Int -> Float -> Float
ℒn⟦foo⟧ :: Array sh Int -> Array sh Float -> Array sh Float
```

- No, lifting to vectors is sufficient
    - At the machine level it's all vectors anyway

- What about nested arrays?
    - Unsupported by accelerate
    - But, we only need vectors of arrays

Vectorisation with multidimensional arrays

- Will this solution work with multidimensional arrays?

- Does it now require this?

```
foo      :: Int -> Float -> Float
ℒn⟦foo⟧ :: Array sh Int -> Array sh Float -> Array sh Float
```

- No, lifting to vectors is sufficient
  - At the machine level it's all vectors anyway

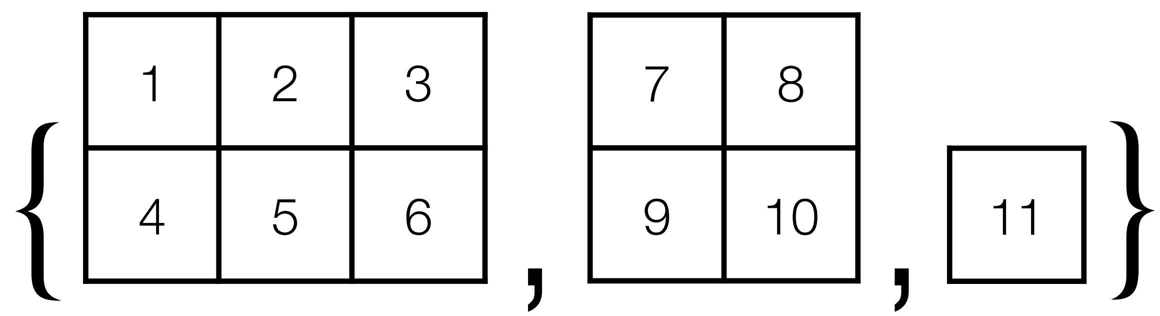
- What about nested arrays?
  - Unsupported by accelerate
  - But, we only need vectors of arrays

Vectors of arrays

```
type Vector' = …a vector of arrays…
```

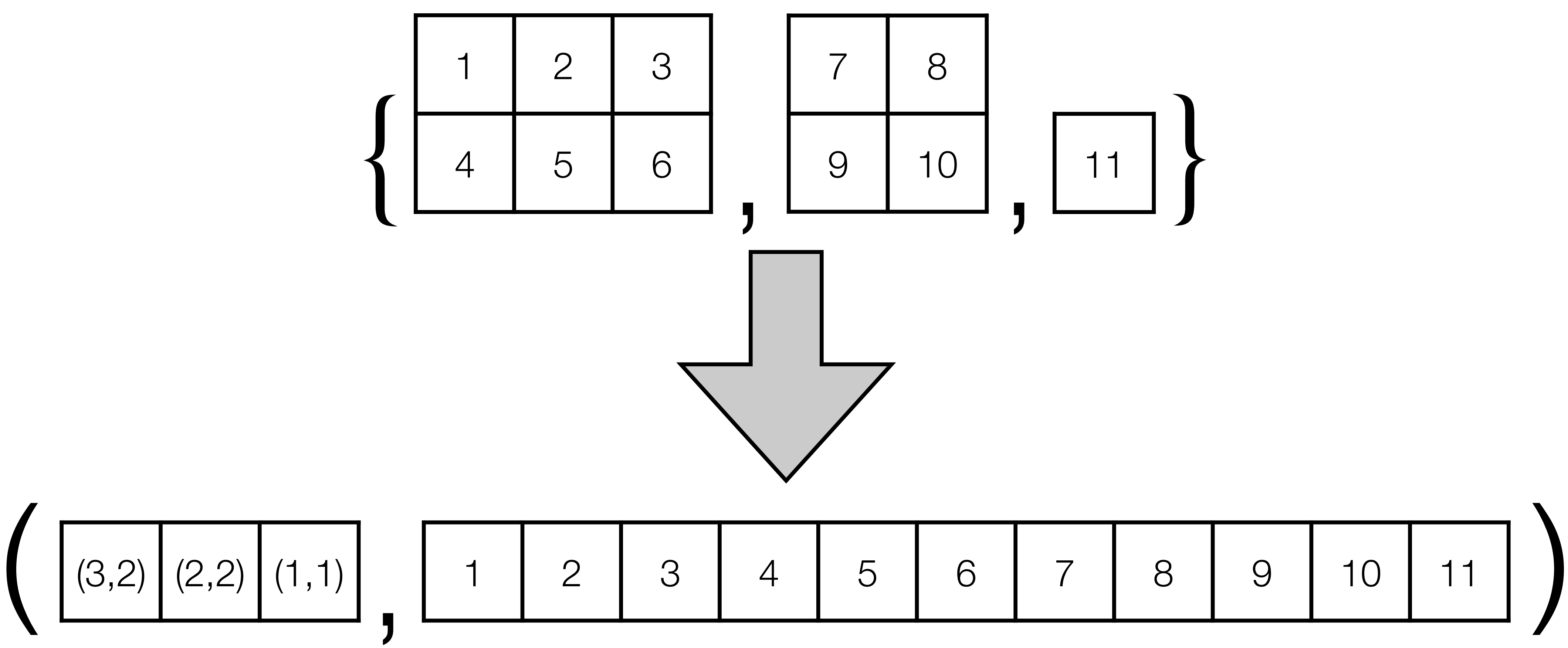
Vectors of arrays

```
type Vector' = …a vector of arrays…
```



Vectors of arrays

```
type Vector' = …a vector of arrays…
```



Back to mvm

```
mvm mat vec = generate (index1 (height mat))
                       (λi -> the (dotp vec (getRow i mat)))
```

Back to mvm

```
mvm mat vec = generate (index1 (height mat))
                       (λi -> the (dotp vec (getRow i mat)))

mvm mat vec
  = ℒn⟦(λi -> the (dotp vec (getRow i mat)))⟧
```

Back to mvm

```
mvm mat vec = generate (index1 (height mat))
                       (λi -> the (dotp vec (getRow i mat)))

mvm mat vec
  = ℒn⟦(λi -> the (dotp vec (getRow i mat)))⟧
    (map index1 [0..height mat])
```

Back to mvm

```
mvm mat vec = generate (index1 (height mat))
                       (λi -> the (dotp vec (getRow i mat)))
mvm mat vec
  = ℒn⟦(λi -> the (dotp vec (getRow i mat)))⟧
      (map index1 [0..height mat])

mvm mat vec
  = (λi -> the↑ (ℒn⟦dotp⟧ (replicate (length i) vec)
                          (getRow↑ i (replicate (length i) mat)))
      (map index1 [0..height mat])
```

```
mvm mat vec
  = (λi -> the↑ (ℒn⟦dotp⟧ (replicate (length i) vec)
                           (getRow↑ i (replicate (length i) mat))))
    (map index1 [0..height mat])
```

```
mvm mat vec
  = (λi -> the↑ (ℒn⟦dotp⟧ (replicate (length i) vec)
                          (getRow↑ i (replicate (length i) mat))))
    (map index1 [0..height mat])
```

Inlining, Fusion, simplification, etc.

```
mvm mat vec
  = (λi -> the↑ (ℒn⟦dotp⟧ (replicate (length i) vec)
                             (getRow↑ i (replicate (length i) mat))))
    (map index1 [0..height mat])
```

Inlining, Fusion, simplification, etc.

```
mvm mat vec = snd $ fold↑ (+) 0 (let h = height mat
                                 in ( replicate h (shape vec)
                                    , zipWith (*)
                                              (replicate↑ h vec)
                                              (flatten mat)))
```

```
mvm mat vec
  = (λi -> the↑ (ℒn⟦dotp⟧ (replicate (length i) vec)
                           (getRow↑ i (replicate (length i) mat))))
    (map index1 [0..height mat])
```

Inlining, Fusion,
simplification, etc.

```
mvm mat vec = snd $ fold↑ (+) 0 (let h = height mat
                                 in ( replicate h (shape vec)
                                    , zipWith (*)
                                              (replicate↑ h vec)
                                              (flatten mat)))
```

```
mvm mat vec
  = (λi -> the↑ (ℒn⟦dotp⟧ (replicate (length i) vec)
                           (getRow↑ i (replicate (length i) mat))))
    (map index1 [0..height mat])
```

Inlining, Fusion,
simplification, etc.

```
mvm mat vec = snd $ fold↑ (+) 0 (let h = height mat
                                 in ( replicate h (shape vec)
                                    , zipWith (*)
                                              (replicate↑ h vec)
                                              (flatten mat)))
```

A vector of arrays

```
mvm mat vec
  = (λi -> the↑ (ℒn⟦dotp⟧ (replicate (length i) vec)
                           (getRow↑ i (replicate (length i) mat))))
    (map index1 [0..height mat])
```

Inlining, Fusion, simplification, etc.

```
mvm mat vec = snd $ fold↑ (+) 0 (let h = height mat
                                  in ( replicate h (shape vec)
                                     , zipWith (*)
                                               (replicate↑ h vec)
                                               (flatten mat)))
```

A vector of arrays
… but they're all the same size?

```
mvm mat vec
  = (λi -> the↑ (ℒn⟦dotp⟧ (replicate (length i) vec)
                          (getRow↑ i (replicate (length i) mat))))
    (map index1 [0..height mat])
```

Inlining, Fusion, simplification, etc.

```
mvm mat vec = snd $ fold↑ (+) 0 (let h = height mat
                                 in ( replicate h (shape vec)
                                    , zipWith (*)
                                              (replicate↑ h vec)
                                              (flatten mat)))
```

Replace with normal fold?

A vector of arrays
… but they're all the same size?

?

```
mvm mat vec
  = fold (+) 0 (zipWith (*) mat (replicateRow (height mat) vec))
```

```
mvm mat vec = snd $ fold↑ (+) 0 (let h = height mat
                                 in ( replicate h (shape vec)
                                    , zipWith (*)
                                              (replicate↑ h vec)
                                              (flatten mat)))
```

?

```
mvm mat vec
  = fold (+) 0 (zipWith (*) mat (replicateRow (height mat) vec))
```

Questions?