Lightweight Polytypic Staging: a new approach to Nested Data Parallelism in Scala

Alexander Slesarenko
Keldysh Institute of Applied Mathematics, 2012

The Domain - Nested Data Parallelism

- The original idea
  - Guy Blelloch, Gary Sabot: in the early 90's ([1] is a good starting point)
  - NESL – proof of concept (first order, interpreted language)
- Generalizations (90's – 00's)
  - Chakravarty, Keller, S. P. Jones et al. (5 or 6 papers)
  - Data Parallel Haskell – higher-order, compiled language [2]
  - Language extension with special syntax
- A big promise but still in research

[1] Guy E. Blelloch. *Vector models for data-parallel computing.* MIT Press, Cambridge, MA, USA, 1990
[2] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. *Harnessing the Multicores: Nested Data Parallelism in Haskell*, 2008

Motivation: NDP as an embedded DSL

- NDP is not a “silver bullet”
- Some applications **fit** to the model but others **don’t**
- For those that **fit** we want high-level declarative language
- IDEALLY: **If it is expressible then it is automatically vectorizable** (with asymptotic work-efficiency)
- Should interact with other DSLs and the host language
- Yet another tool in the Scala toolbox

Polytypic DSL

DSL	Monotypic (traditional)	Polytypic (data type generic)
Shallow embedding	✓ Ordinary types and functions ✓ Execution on the JVM	
Deep embedding		

Polytypic DSL

DSL	Monotypic (traditional)	Polytypic (data type generic)
Shallow embedding	✓ Ordinary types and functions ✓ Execution on the JVM	
Deep embedding	✓ Ordinary types and functions ✓ Staging + transform ✓ Execution on XXX by code generation	

Polytypic DSL

DSL	Monotypic (traditional)	Polytypic (data type generic)
Shallow embedding	✓ Ordinary types and functions ✓ Execution on the JVM	✓ Type-indexed types and functions ✓ Execution on the JVM
Deep embedding	✓ Ordinary types and functions ✓ Staging + transform ✓ Execution on XXX by code generation	

Polytypic DSL

DSL	Monotypic (traditional)	Polytypic (data type generic)
Shallow embedding	✓ Ordinary types and functions ✓ Execution on the JVM	✓ Type-indexed types and functions ✓ Execution on the JVM
Deep embedding	✓ Ordinary types and functions ✓ Staging + transform ✓ Execution on XXX by code generation	✓ Type-indexed types and functions ✓ Staging + transform ✓ Execution on XXX by code generation

- In the implementation we need "the best" of the two worlds
  - Type-indexed types from generic programming
  - Staged execution from deep embedding

Polytypic DSL

DSL	Monotypic (traditional)	Polytypic (data type generic)
Shallow embedding	✓ Ordinary types and functions ✓ Execution on the JVM	✓ Type-indexed types and functions ✓ Execution on the JVM
Deep embedding	✓ Ordinary types and functions ✓ Staging + transform ✓ Execution on XXX by code generation	✓ Type-indexed types and f… Polytypic Staging ✓ Stag… …nsform ✓ Execution … generation Nested Data Parallelism

- In the implementation we need “the best” of the two worlds
  - Type-indexed types from generic programming
  - Staged execution from deep embedding

Framework: Polymorphic Embedding of DSLs

```
type Rep[A]    // abstract type constructor of representations
type PA[A] = Rep[PArray[A]]
trait PArray[A] {    // parallel array (to express parallelism)
  def length: Rep[Int]
  def map[R](f: Rep[A] => Rep[R]): PA[R]
  def zip[B](b: PA[B]): PA[(A,B)]
       …
}
```

```
type Vector = PArray[Float]  // parallel array
def dotProduct(vec1: Rep[Vector], vec2: Rep[Vector]): Rep[Float] =
  sum((vec1 zip vec2) map { case Pair(v1,v2) => v1 * v2 })
```

Framework: Polymorphic Embedding of DSLs

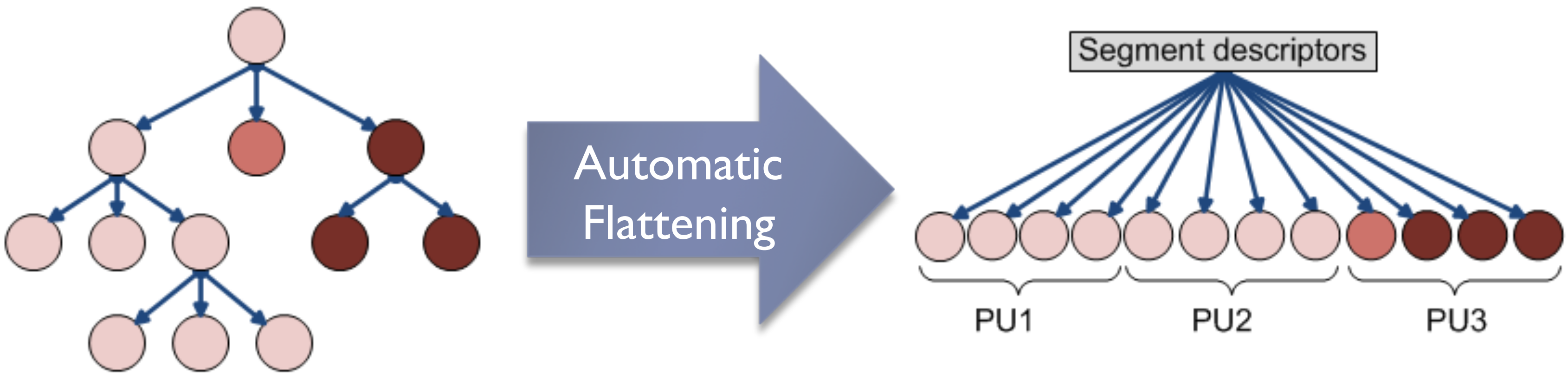
```
type Rep[A]    // abstract type constructor of representations
type PA[A] = Rep[PArray[A]]
trait PArray[A] {    // parallel array (to express parallelism)
  def length: Rep[Int]
  def map[R](f: Rep[A] => Rep[R]): PA[R]
  def zip[B](b: PA[B]): PA[(A,B)]
      …
}
```

- The same code
- Two implementations
- Equivalent semantics

```
type Vector = PArray[Float]  // parallel array
def dotProduct(vec1: Rep[Vector], vec2: Rep[Vector]): Rep[Float] =
  sum((vec1 zip vec2) map { case Pair(v1,v2) => v1 * v2 })
```

Shallow Embedding	Deep Embedding
`type Rep[A] = A`	`type Rep[A] = Exp[A]`
✓Should be simple, good for testing and debugging	✓Should generate an efficient code

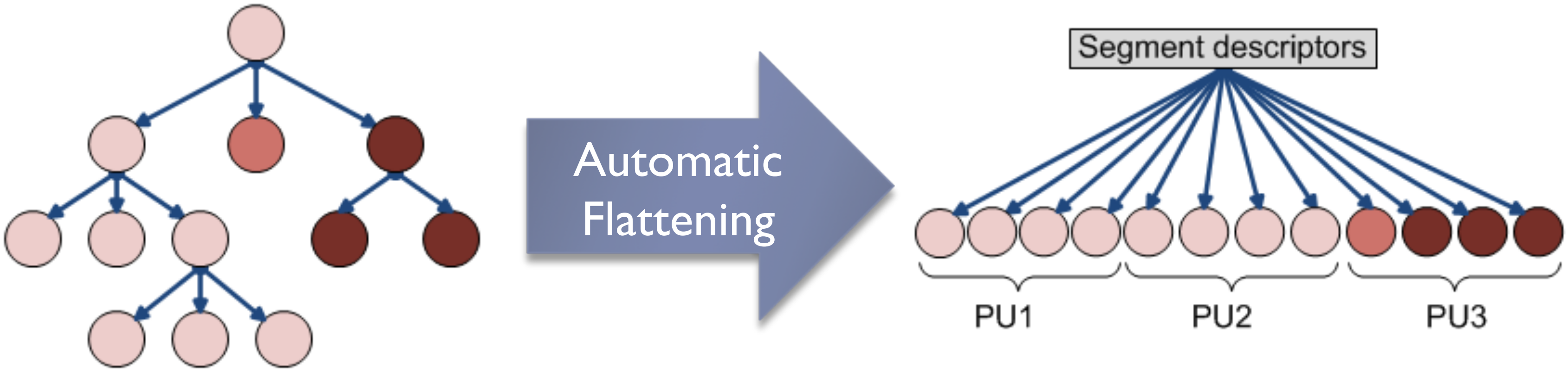
The Key Idea – flattening transformation



- The data might be irregular
- ill-balanced and not very parallel at top level
- The one we want to write

- regular after flattening
- Balanced chunking
- The one we want to run

The Key Idea – flattening transformation



- The data might be irregular
- ill-balanced and not very parallel at top level
- The one we want to write

- regular after flattening
- Balanced chunking

- The one we want to run

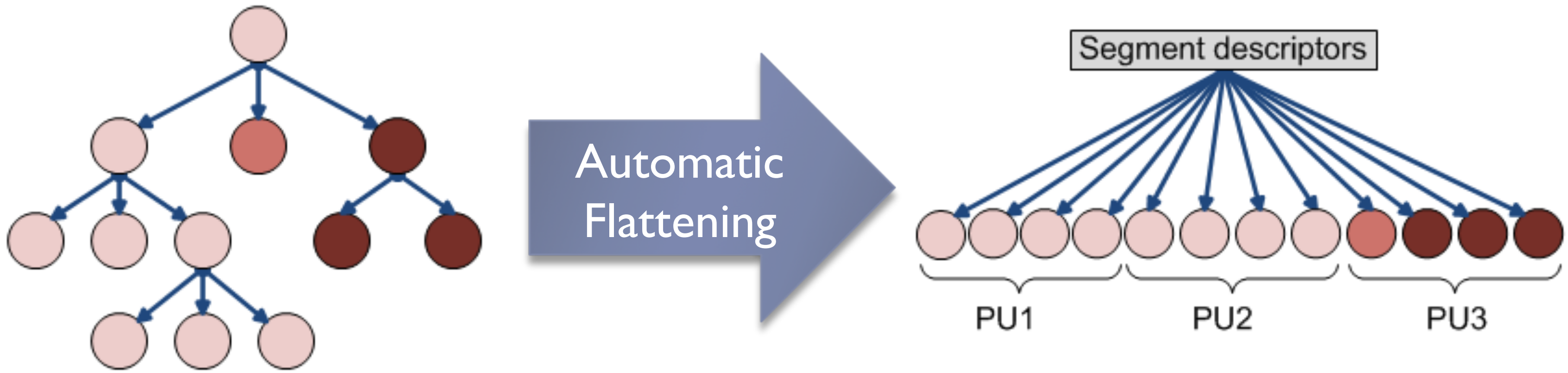
Sparse matrix

$$\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 1.0 & 0 & 2.0 & 0 \\ 3.0 & 4.0 & 5.0 & 0 \\ 0 & 0 & 0 & 6.0 \end{pmatrix} \end{array}$$

Compressed row format

(0, 1.0)	(2, 2.0)	
(0, 3.0)	(1, 4.0)	(2, 5.0)
(3, 6.0)		

The Key Idea – flattening transformation



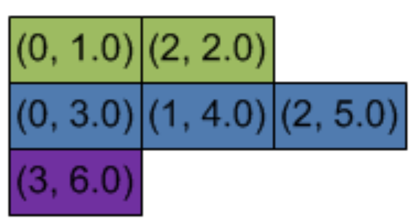
- The data might be irregular
- ill-balanced and not very parallel at top level
- regular after flattening
- Balanced chunking

- The one we want to write

Sparse matrix

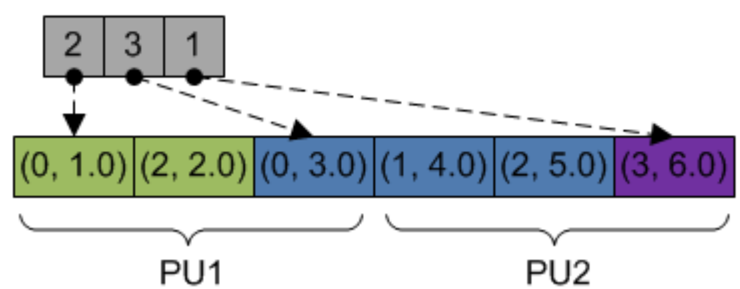
$$\begin{array}{cccc} 0 & 1 & 2 & 3 \end{array}$$
$$\begin{pmatrix} 1.0 & 0 & 2.0 & 0 \\ 3.0 & 4.0 & 5.0 & 0 \\ 0 & 0 & 0 & 6.0 \end{pmatrix}$$

Compressed row format



- The one we want to run

Flat representation



How flattening happens

Nested Code	Flattened Code
`p: A => B    // primitive`	`type PA[A] = PArray[A]` `p^: PA[A] => PA[B] // p-lifted`
`def g(as: PA[A]) = as map p`	`def g(as: PA[A]) = p^(as)`

`def h(m: PA[PA[A]])`
`    = m map g`
	`def h(m: PA[PA[A]])`
	`    = m map g`
	`    = m map p^  // inline g`
	`    = p^^(m)     // ???`

How flattening happens

Nested Code	Flattened Code
`p: A => B     // primitive`	`type PA[A] = PArray[A]` `p^: PA[A] => PA[B] // p-lifted`
`def g(as: PA[A]) = as map p`	`def g(as: PA[A]) = p^(as)`

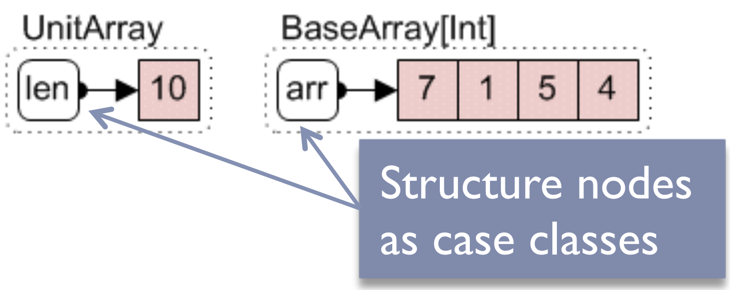
`def h(m: PA[PA[A]])`
`    = m map g`
	`def h(m: PA[PA[A]])`
	`    = m map g`
	`    = m map p^   // inline g`
	`    = p^^(m)     // ???`

The key insight (we don't need p^^)

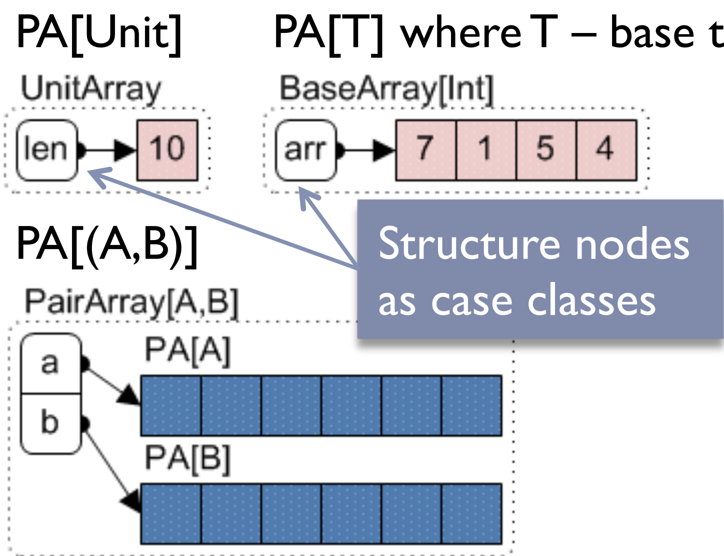
```
def p^^(m: PA[PA[A]]): PA[PA[A]] =
      unconcat(m, p^(concat(m)))

def concat[A](nested: PA[PA[A]]): PA[A]
def unconcat[A,B](shape: PA[PA[A]], values: PA[B]): PA[PA[B]]
```

Data structures that support flattening

PA[Unit] PA[T] where T – base type



Data structures that support flattening

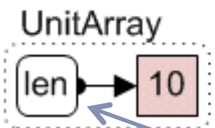


Constant time operations

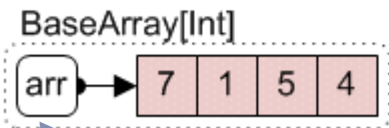
```
def zip[A,B](a:PA[A], b:PA[B]):PA[(A,B)]
    = PairArray(a, b)
```

Data structures that support flattening

PA[Unit]

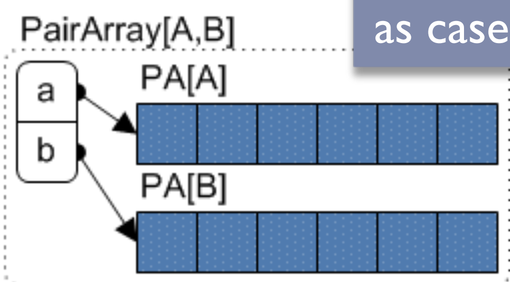


PA[T] where T – base type

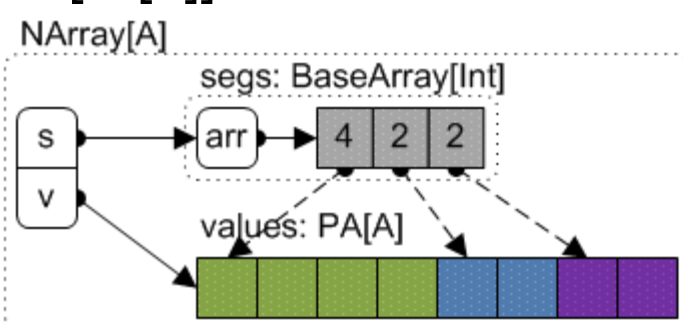


Structure nodes as case classes

PA[(A,B)]



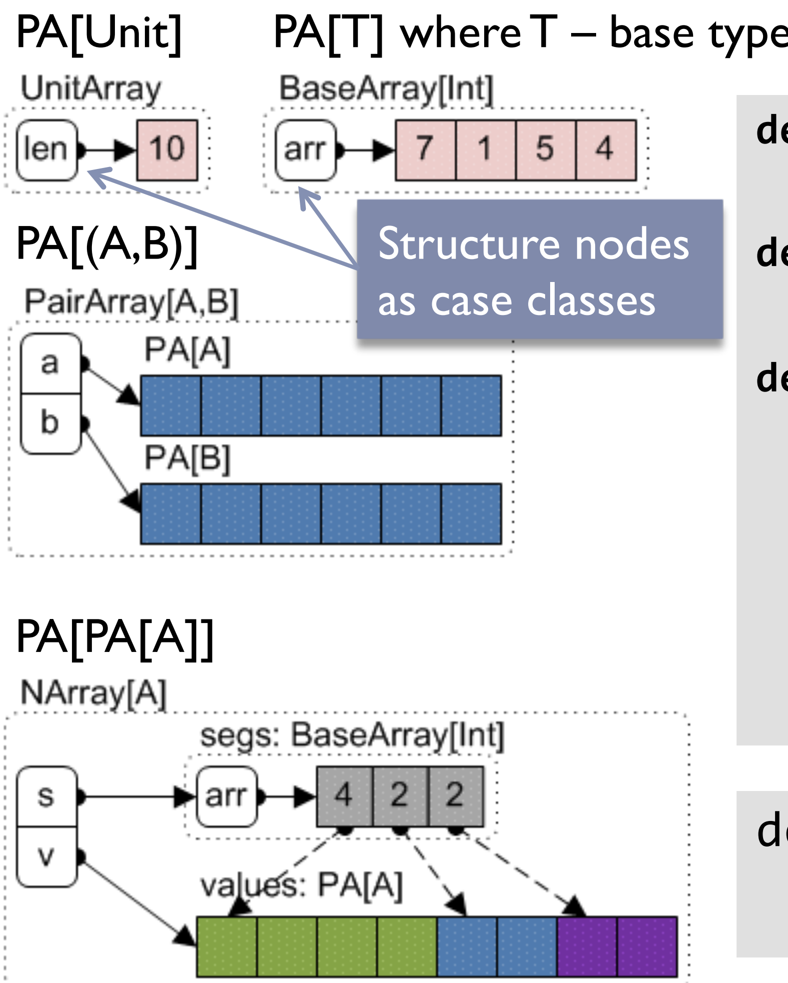
PA[PA[A]]



Constant time operations

```
def zip[A,B](a:PA[A], b:PA[B]):PA[(A,B)]
    = PairArray(a, b)
```

Data structures that support flattening



Constant time operations

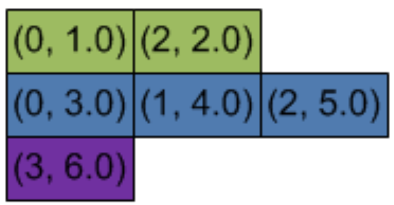
```
def zip[A,B](a:PA[A], b:PA[B]):PA[(A,B)]
    = PairArray(a, b)
def concat[A](na: PA[PA[A]]): PA[A] =
    na match { case NArray(vs,_) => vs }
def unconcat[A,B](shape:PA[PA[A]],
                       vs:PA[B]): PA[PA[B]] =
  shape match {
   case NArray(_,segs) =>
  NArray(vs,segs)
  }
```

```
def p^^(m: PA[PA[A]]): PA[PA[A]] =
  unconcat(m, p^(concat(m)))
```

Example (application specific types)

In the DSL we can construct types

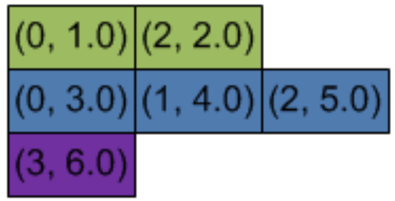
```
T = Unit | Int | Float | Boolean // base types
  | (T1,T2)                      // product of types
  | (T1+T2)                      // sum of types
  | PArray[T]                    // nested array
```



$$\overset{M}{\begin{pmatrix} 1.0 & 0 & 2.0 & 0 \\ 3.0 & 4.0 & 5.0 & 0 \\ 0 & 0 & 0 & 6.0 \end{pmatrix}} * \overset{V}{\begin{pmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{pmatrix}} = \overset{R}{\begin{pmatrix} 7.0 \\ 26 \\ 24 \end{pmatrix}}$$

```
// Matrix in compressed row format
type SVector = PArray[(Int,Float)] // parallel array of products
type SMatrix = PArray[SVector]     // nested array of rows
type Vector = PArray[Float]        // dense vector
```

Example (Sparse Matrix Vector Multiplication)



$$\underset{M}{\begin{pmatrix} 1.0 & 0 & 2.0 & 0 \\ 3.0 & 4.0 & 5.0 & 0 \\ 0 & 0 & 0 & 6.0 \end{pmatrix}} * \underset{V}{\begin{pmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{pmatrix}} = \underset{R}{\begin{pmatrix} 7.0 \\ 26 \\ 24 \end{pmatrix}}$$

```
// Matrix in compressed row format
type SVector = PArray[(Int,Float)] // parallel array of products
type SMatrix = PArray[SVector]     // nested array of rows
type Vector = PArray[Float]        // dense vector
```

```
def sparseVectorMul(sv: SVector, vec: Vector): Float =
      sum(sv map { case (i,v) => vec(i) * v })

def smvm(matr: SMatrix, vec: Vector): Vector =
      for (row <- matr)
          yield sparseVectorMul(row, vec)
```

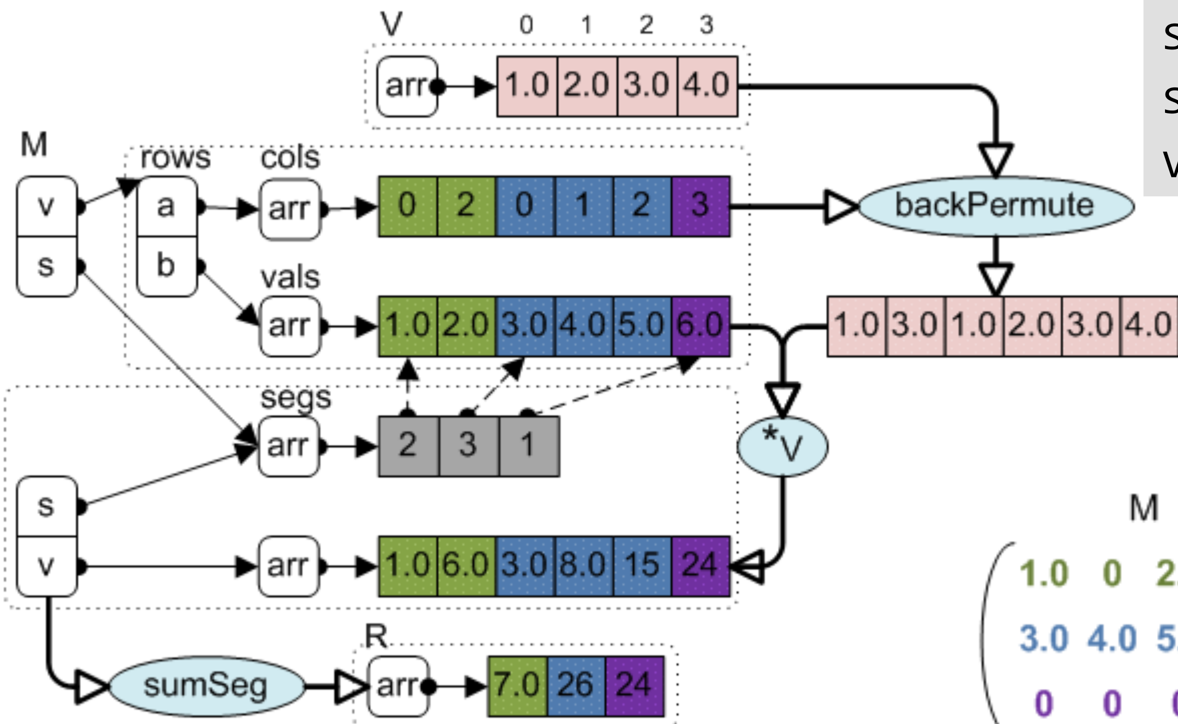
Inner parallelism

Outer parallelism

SMVM vectorized

```
def sparseVectorMul(sv: SVector, vec: Vector): Float =
       sum(sv map { case (i,v) =>  vec(i) * v })
def matrixVectorMul(matr: SMatrix, vec: Vector) =
       for (row <- matr) yield sparseVectorMul(row, vec)
```

```
SVector = PA[(Int,Float)]
SMatrix = PA[SVector]
Vector = PA[Float]
```



$$
\begin{array}{c}M\\ \begin{pmatrix}1.0 & 0 & 2.0 & 0\\ 3.0 & 4.0 & 5.0 & 0\\ 0 & 0 & 0 & 6.0\end{pmatrix}\end{array}
*
\begin{array}{c}V\\ \begin{pmatrix}1.0\\ 2.0\\ 3.0\\ 4.0\end{pmatrix}\end{array}
=
\begin{array}{c}R\\ \begin{pmatrix}7.0\\ 26\\ 24\end{pmatrix}\end{array}
$$

Polytypic Staging

- Uses generic programming to capture domain semantics
- Allows the flattening of the DSL code by staged execution
- Is based on practical approaches: Scala-virtualized compiler, Polymorphic Embedding and LMS
- Not limitied to NDP domain

Conclusions

- Nested data parallelism can be implemented in Scala as an embedded polytypic DSL
- To support flattening we need both staging and type-indexed data types
- Lightweight Polytypic Staging (LPS) is a framework for embedding of polytypic DSLs
- Nested Data Parallelism is a "killer app" for LPS
- WANTED: other polytypic domains?

References

[1] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. *Polymorphic embedding of DSLs*. GPCE ’08

[2] Tiark Rompf, Martin Odersky. *Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs*. GPCE’10

[3] Bruno C.d.S. Oliveira, Jeremy Gibbons. *Scala for generic programmers.* WGP'08

Q&A

???