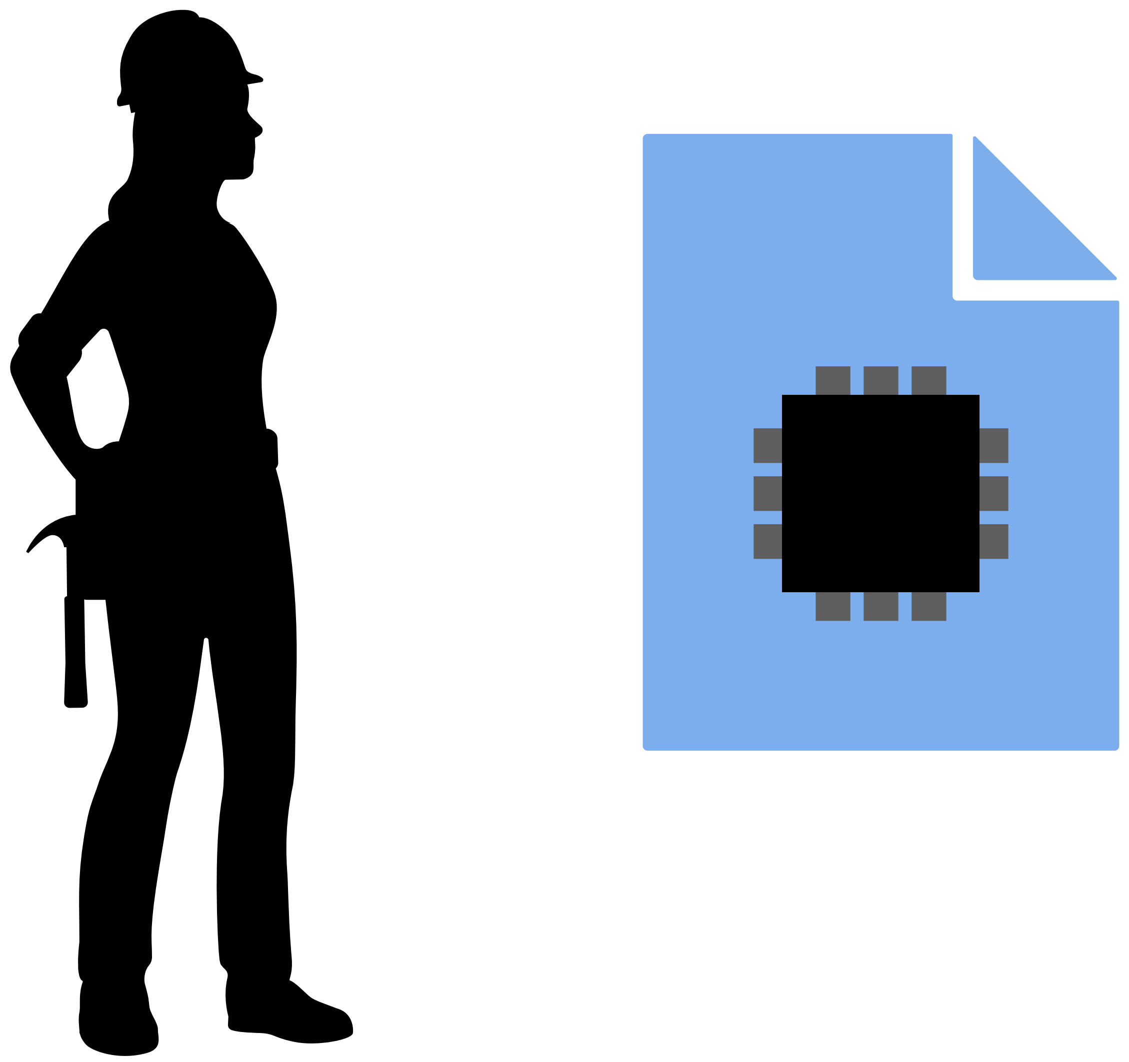
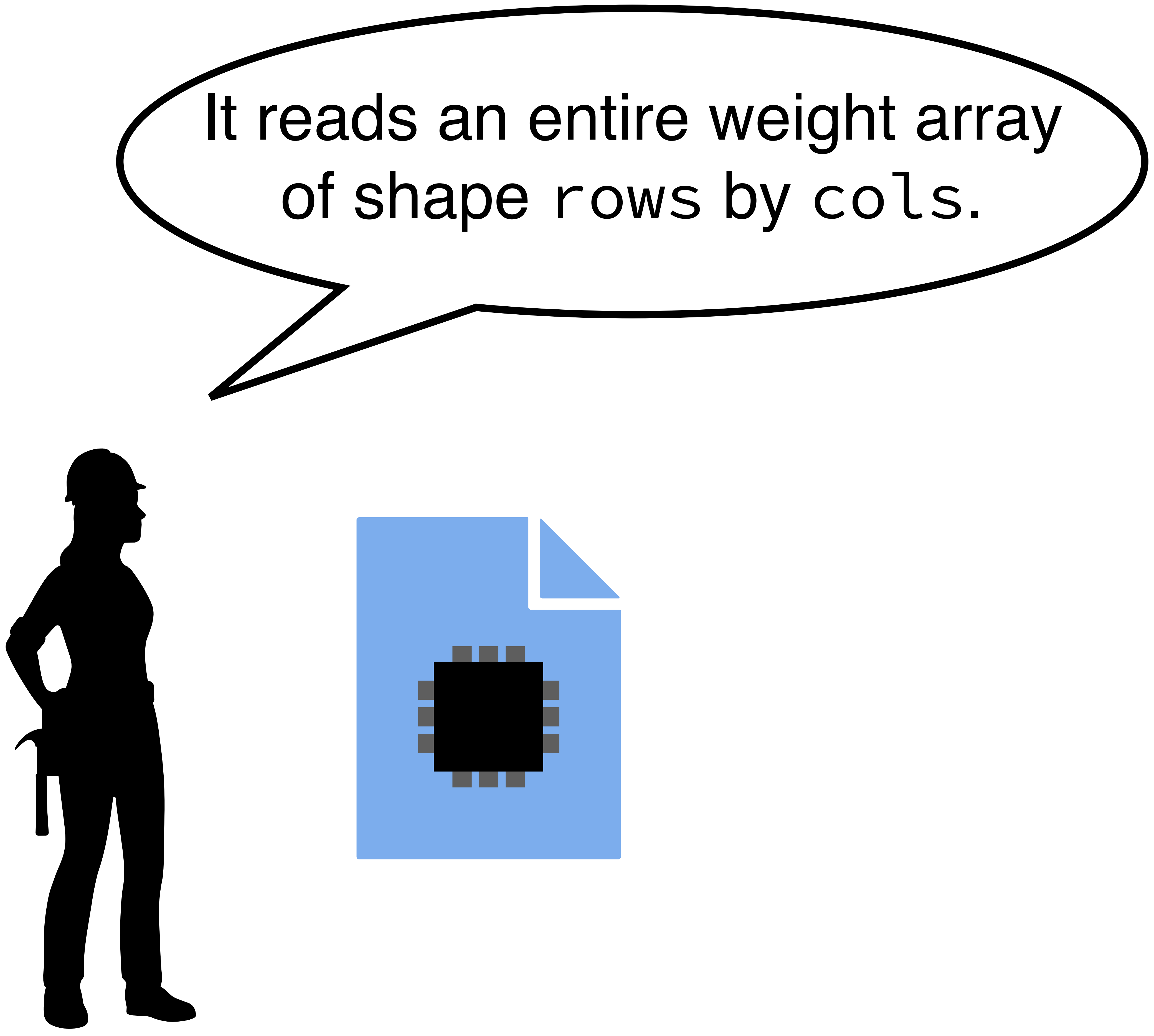
Pure Tensor Program Rewriting via Access Patterns

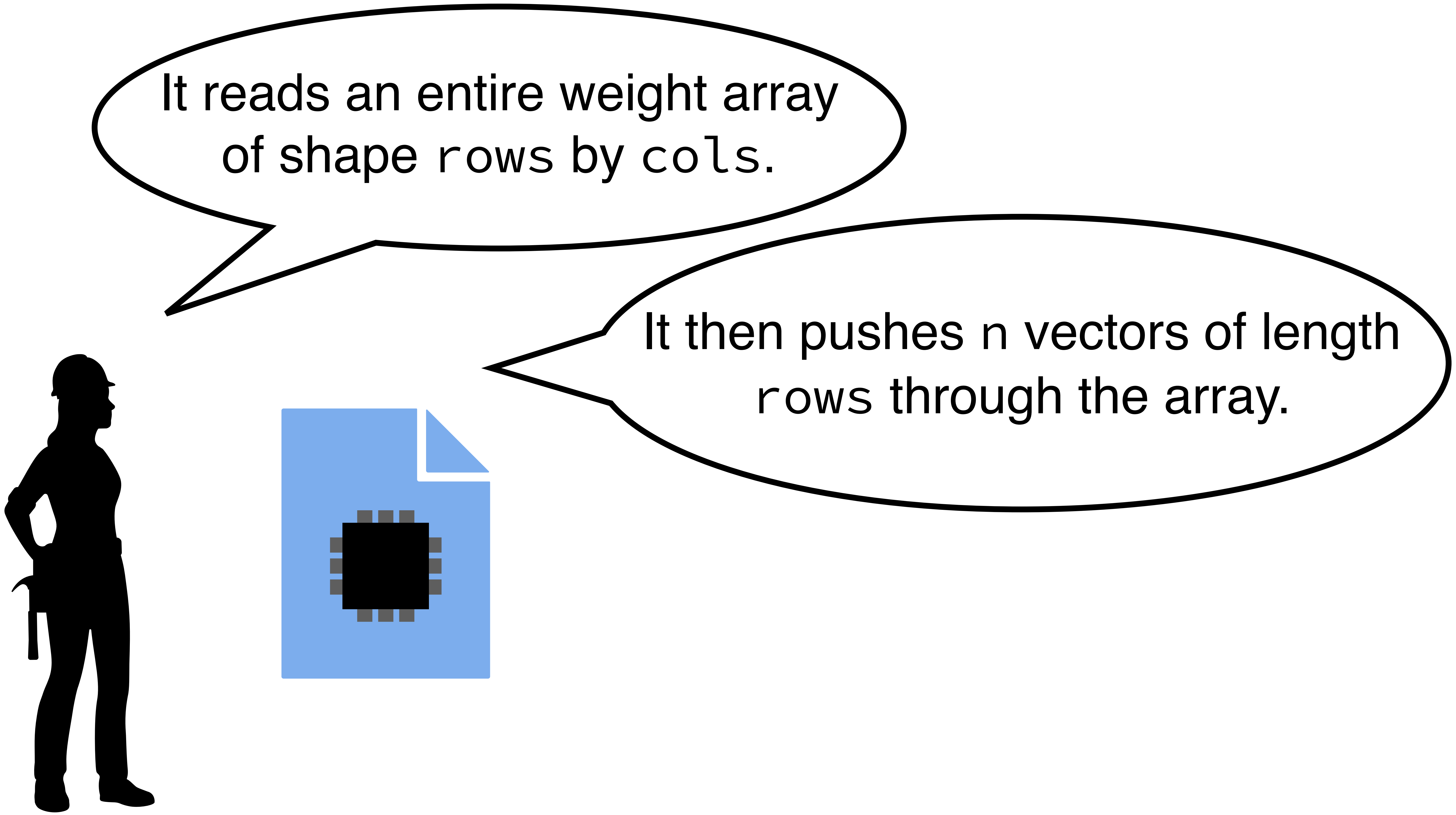
Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, Zachary Tatlock

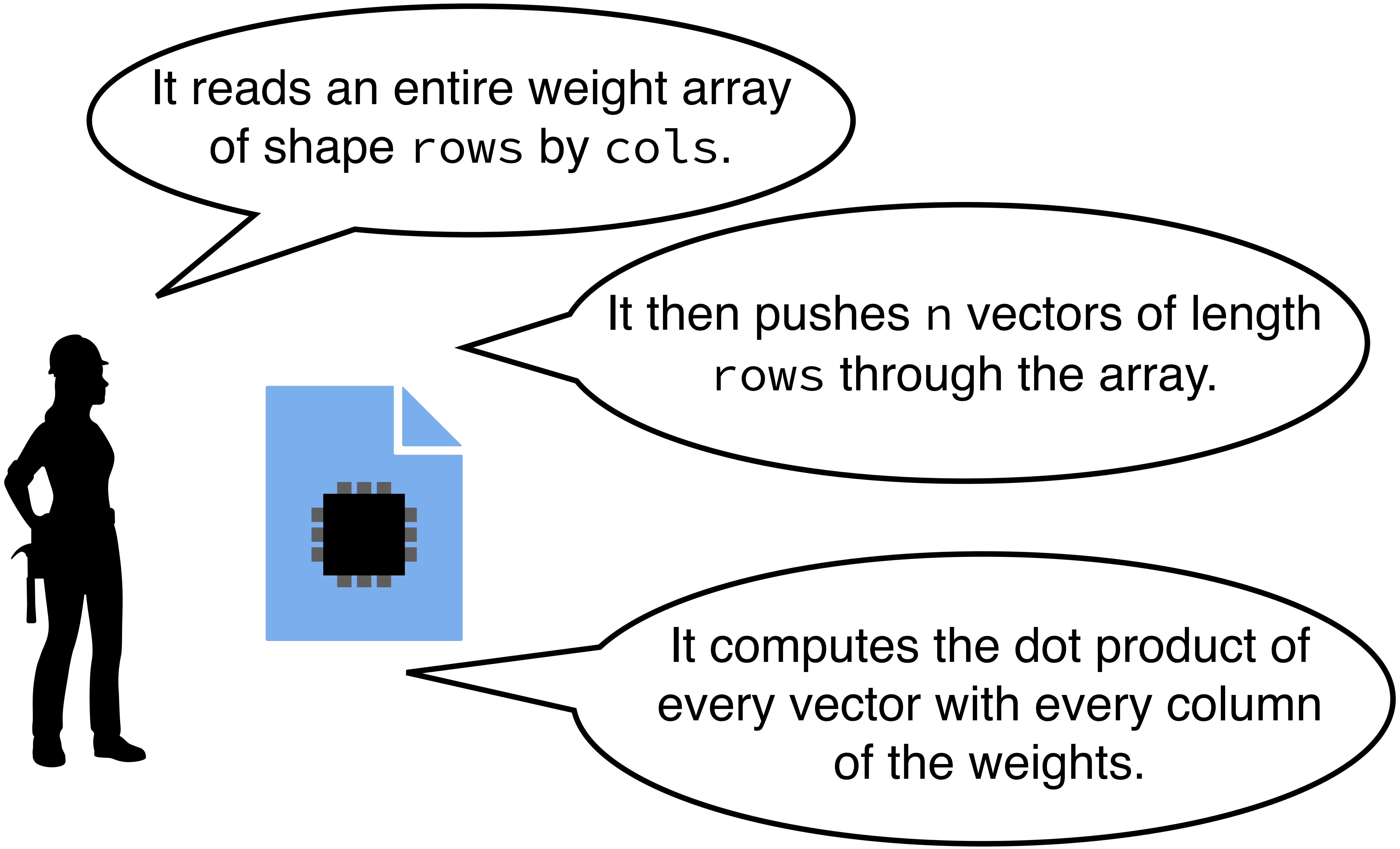
University of Washington

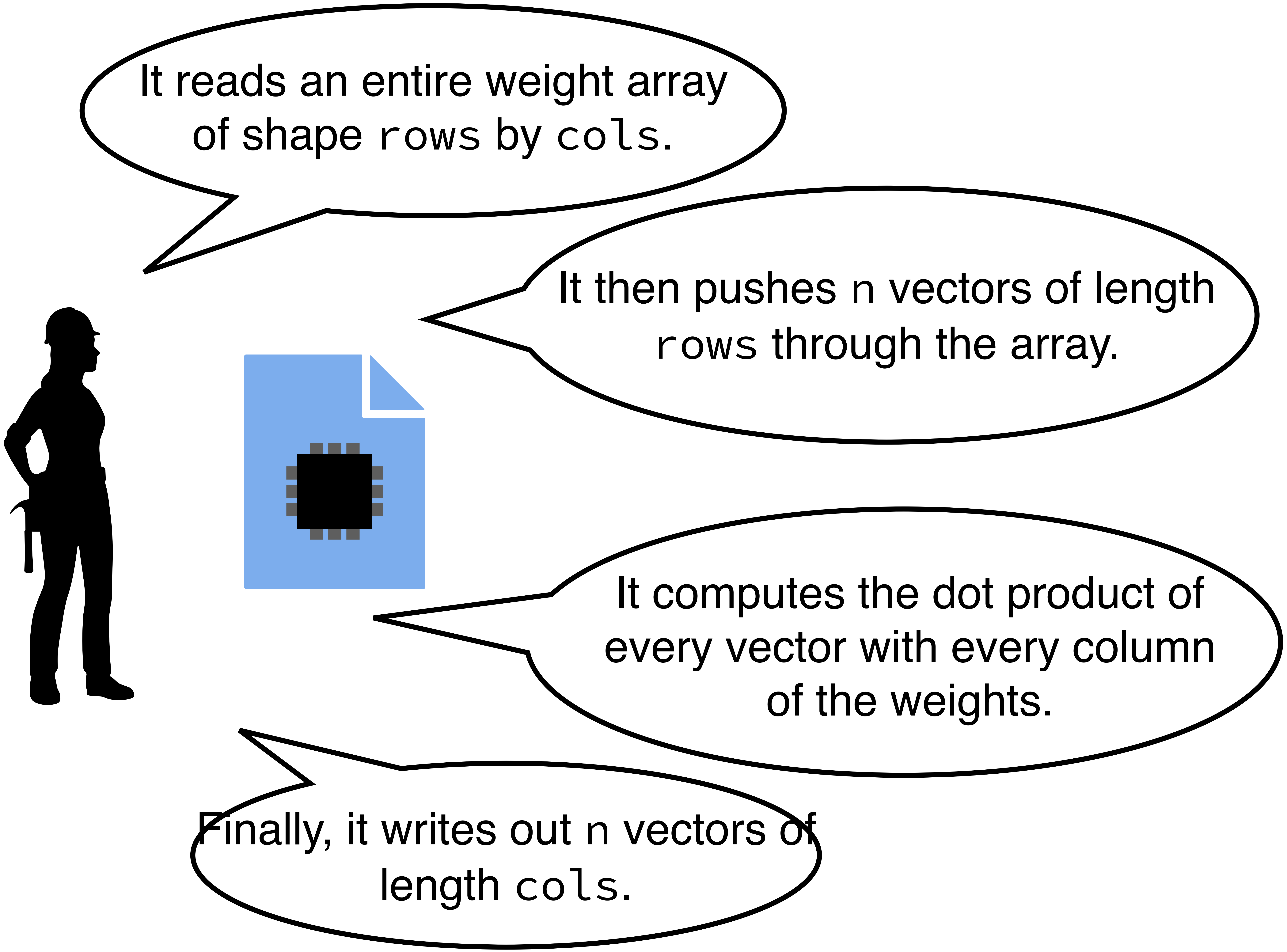
PAUL G. ALLEN SCHOOL

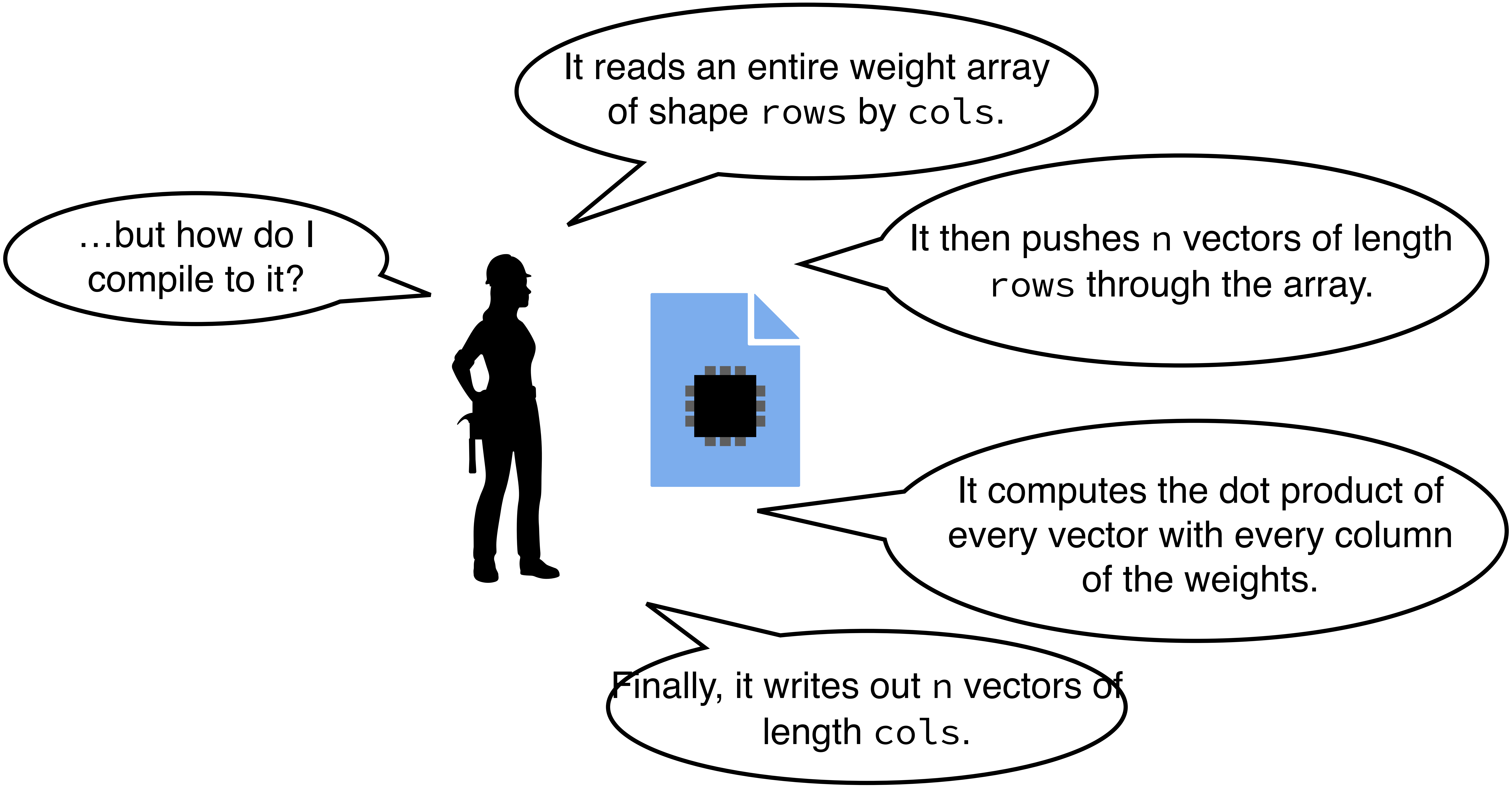


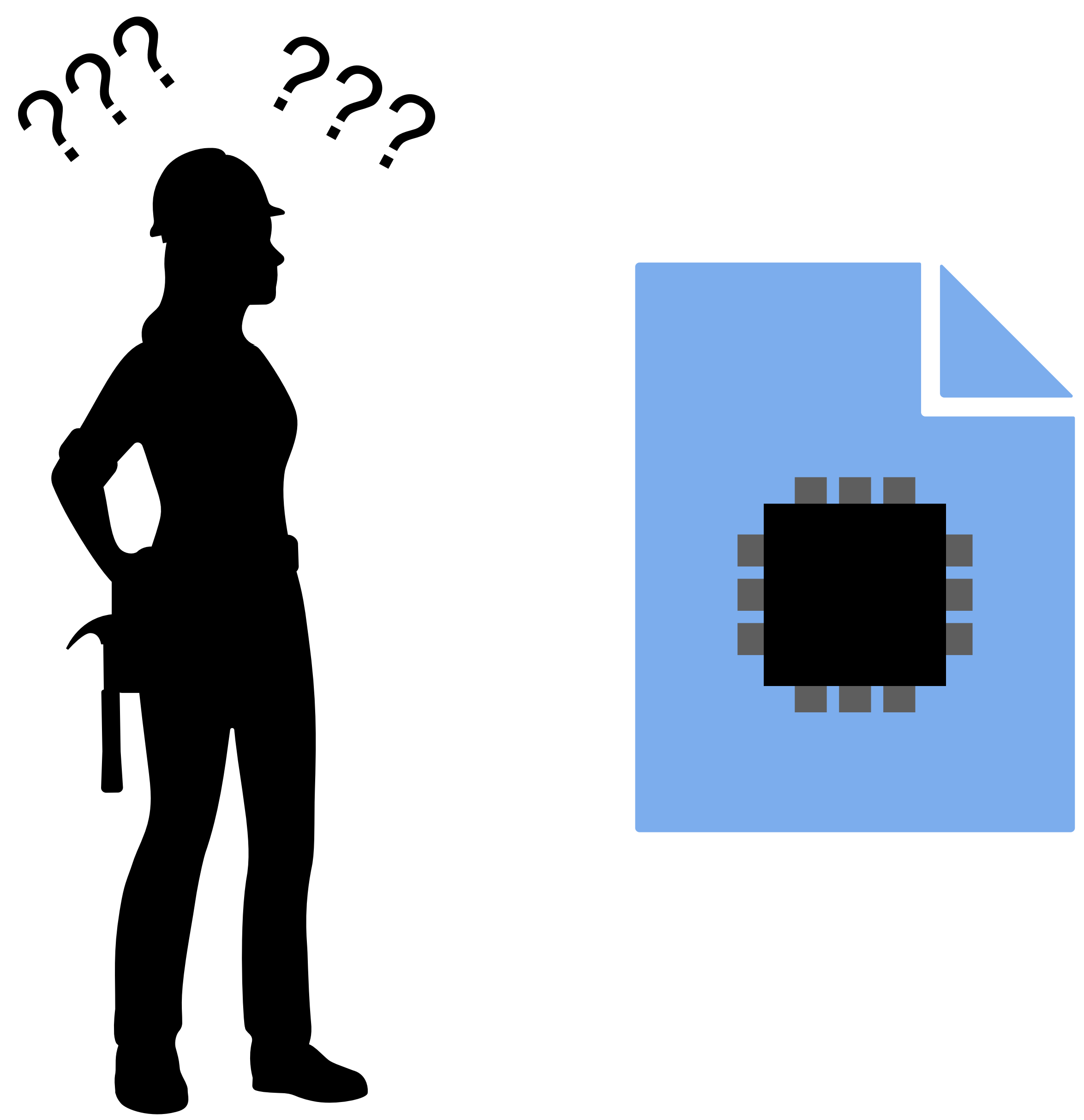

It reads an entire weight array of shape rows by cols.


It reads an entire weight array of shape `rows` by `cols`.
It then pushes `n` vectors of length `rows` through the array.


It reads an entire weight array of shape rows by cols.
It then pushes n vectors of length rows through the array.
It computes the dot product of every vector with every column of the weights.


It reads an entire weight array of shape `rows` by `cols`.
It then pushes `n` vectors of length `rows` through the array.
It computes the dot product of every vector with every column of the weights.
Finally, it writes out `n` vectors of length `cols`.

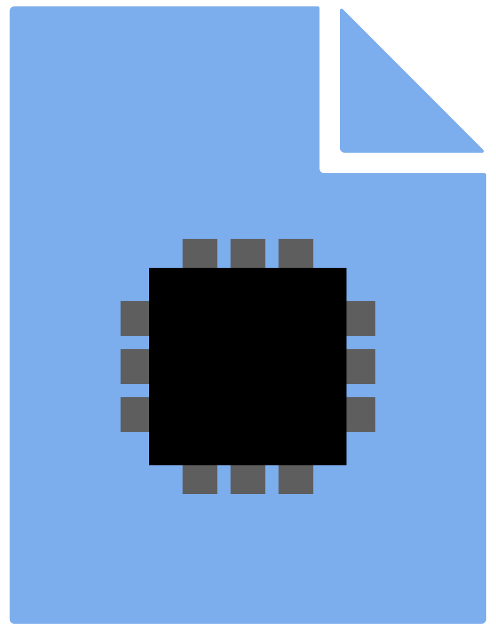

It reads an entire weight array of shape rows by cols.
It then pushes n vectors of length rows through the array.
It computes the dot product of every vector with every column of the weights.
Finally, it writes out n vectors of length cols.
…but how do I compile to it?



<custom compiler>

tvm

PyTorch

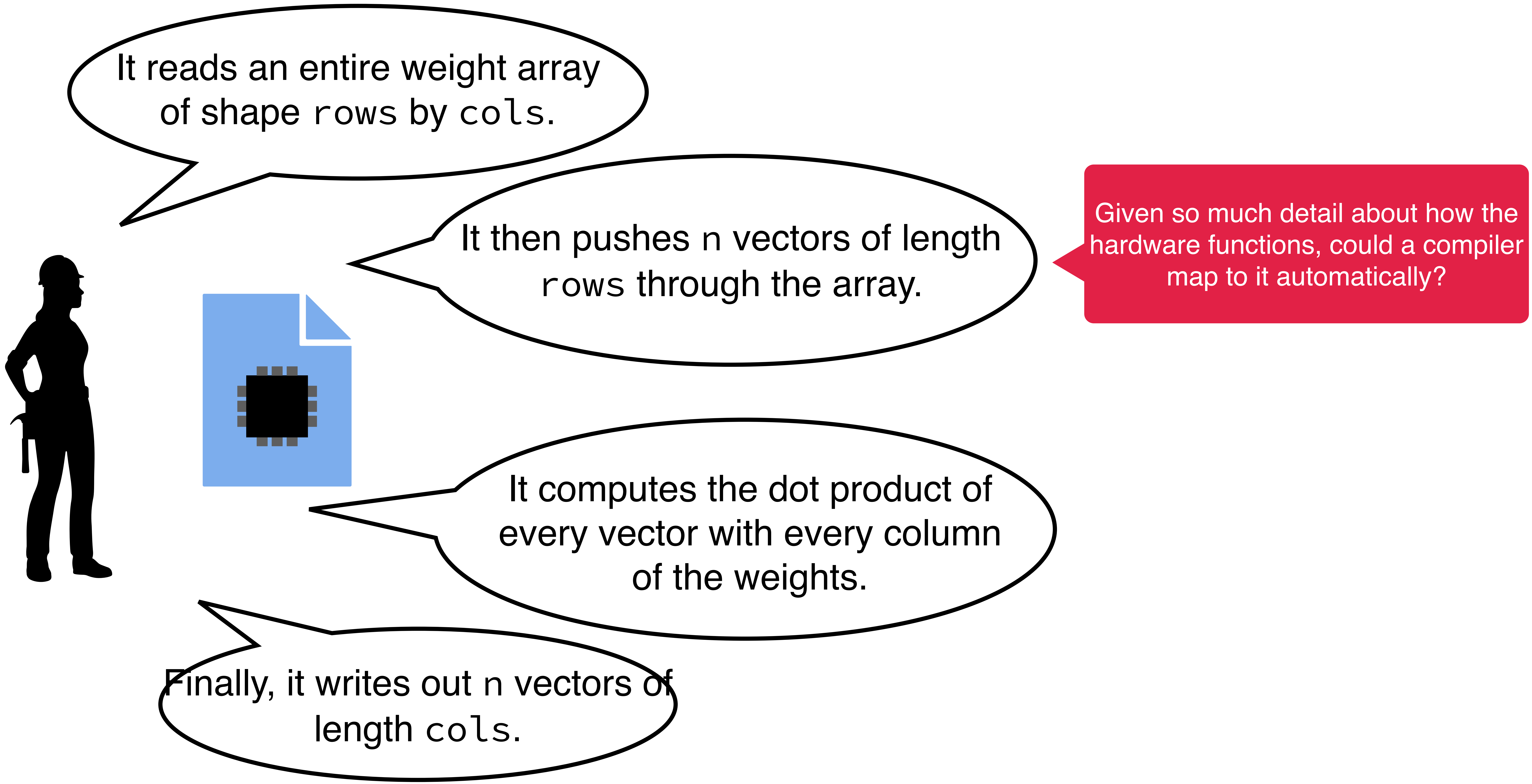


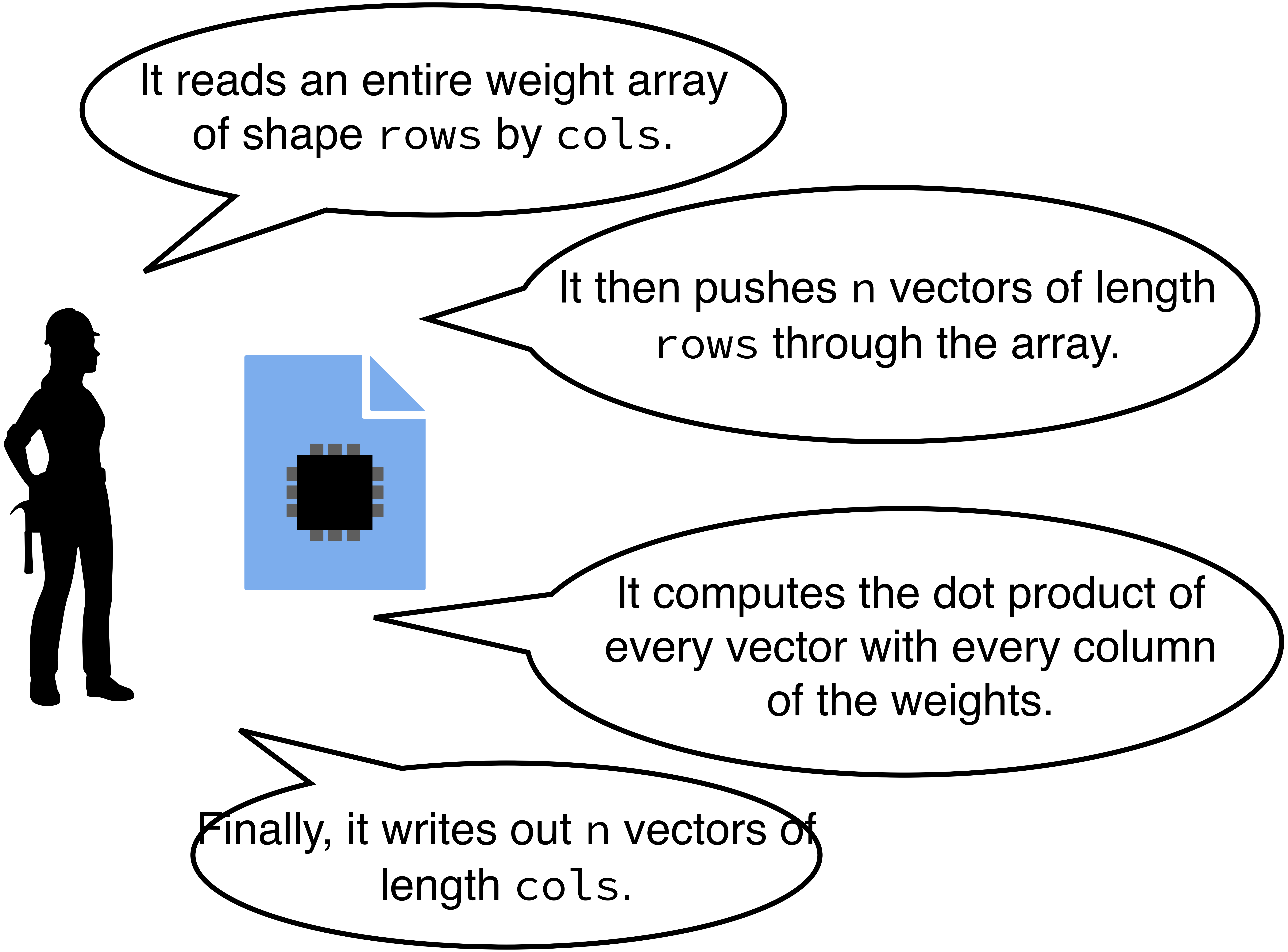
<custom compiler>

tvm

Building backends is hard, even for compiler engineers!

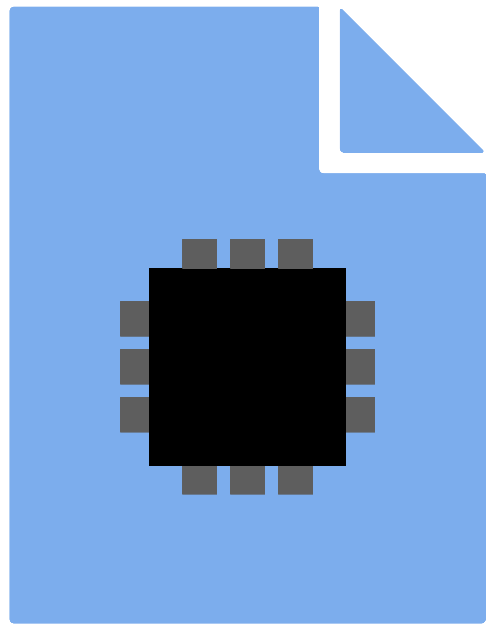
PyTorch


It reads an entire weight array of shape rows by cols.
It then pushes n vectors of length rows through the array.
Given so much detail about how the hardware functions, could a compiler map to it automatically?
It computes the dot product of every vector with every column of the weights.
Finally, it writes out n vectors of length cols.


It reads an entire weight array of shape rows by cols.
It then pushes n vectors of length rows through the array.
It computes the dot product of every vector with every column of the weights.
Finally, it writes out n vectors of length cols.

Can we compile her description of the hardware into a pattern, and search the workload for this pattern?

It reads an entire weight array of shape `rows` by `cols`.

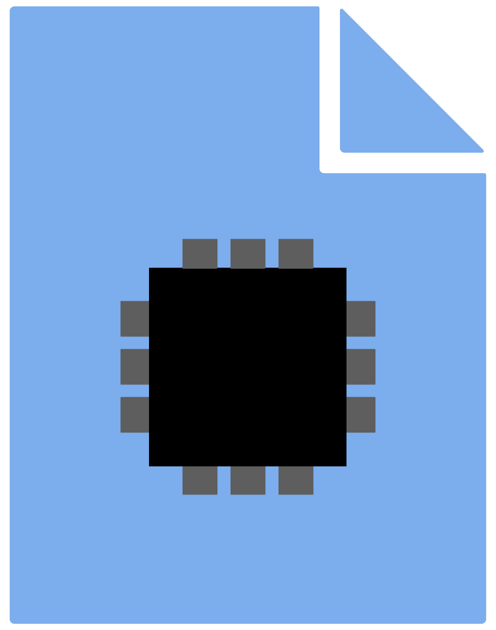


It then pushes `n` vectors of length `rows` through the array.

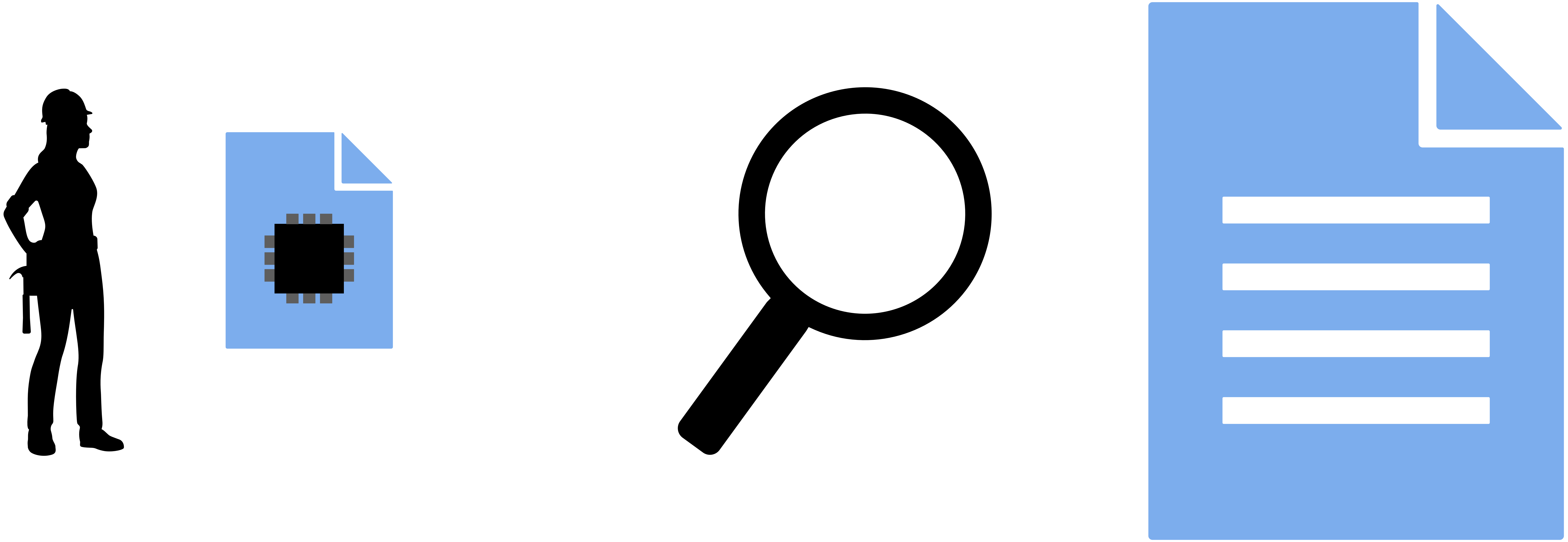
It computes the dot product of every vector with every column of the weights.

Finally, it writes out `n` vectors of length `cols`.

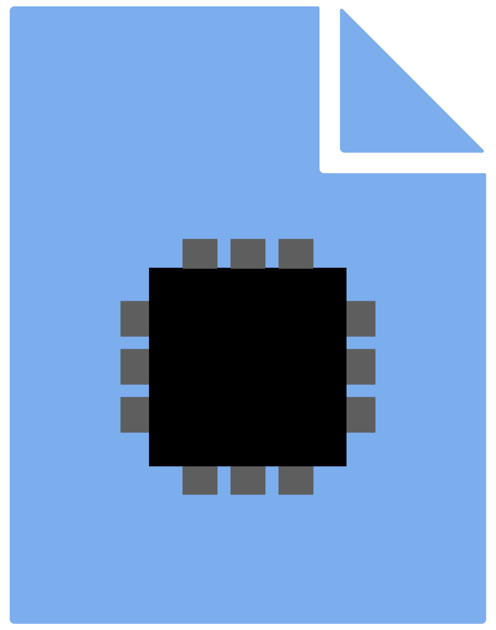
Can we compile her description of the hardware into a pattern, and search the workload for this pattern?



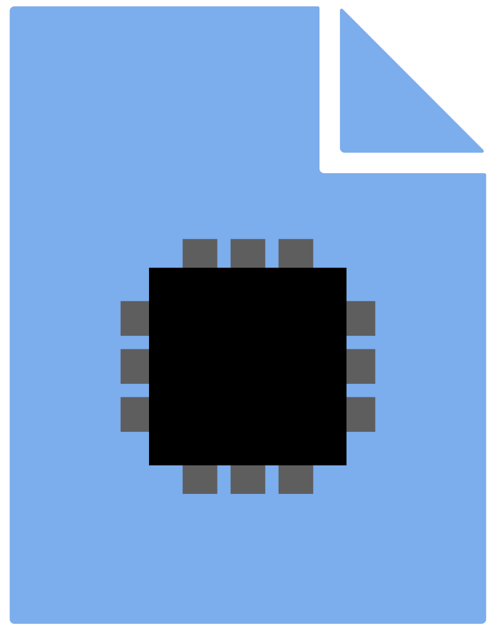
Can we compile her description of the hardware into a pattern, and search the workload for this pattern?



Can we compile her description of the hardware into a pattern, and search the workload for this pattern?



Can we compile her description of the hardware into a pattern, and search the workload for this pattern?



Hardware mapping is a program rewriting problem!

…but current IRs are not up to the task.

Three requirements for a hardware mapping IR:

Three requirements for a hardware mapping IR:

1. The language must be **pure,** enabling equational reasoning in term rewriting.

Three requirements for a hardware mapping IR:

1. The language must be **pure,** enabling equational reasoning in term rewriting.

2. The language must be **low-level,** letting us reason about hardware.

Three requirements for a hardware mapping IR:

1. The language must be **pure,** enabling equational reasoning in term rewriting.

2. The language must be **low-level,** letting us reason about hardware.

3. The language must **not use binding,** making term rewriting much easier.

Three requirements for a hardware mapping IR:

Binding structures—such as lambdas—provide expressiveness.

1. The language ing.

2. The language

3. The language must **not use binding,** making term rewriting much easier.

Three requirements for a hardware mapping IR:

1. The language ing.

Binding structures—such as lambdas—provide expressiveness.

However, they are difficult to deal with in term rewriting: for example, rewrites must explicitly ensure that they do not introduce name conflicts.

2. The language

3. The language must **not use binding,** making term rewriting much easier.

Three requirements for a hardware mapping IR:

1. The language ing.

2. The language

3. The language must **not use binding,** making term rewriting much easier.

Binding structures—such as lambdas—provide expressiveness.

However, they are difficult to deal with in term rewriting: for example, rewrites must explicitly ensure that they do not introduce name conflicts.

Thus, we seek to avoid using binding altogether!

Three examples of IRs from TVM:

Pure?	Low-level?	Can avoid binding?

Three examples of IRs from TVM:

	Pure?	Low-level?	Can avoid binding?
Relay	✅	❌	✅

Three examples of IRs from TVM:

	Pure?	Low-level?	Can avoid binding?
Relay	✅	❌	✅
TE	✅	✅	❌

Three examples of IRs from TVM:

	Pure?	Low-level?	Can avoid binding?
Relay	✅	❌	✅
TE	✅	✅	❌
TIR	❌	✅	❌

Three examples of IRs from TVM:

	Pure?	Low-level?	Can avoid binding?
Relay	✅	❌	✅
TE	✅	✅	❌
TIR	❌	✅	❌

Three examples of IRs from TVM:

	Pure?	Low-level?	Can avoid binding?
Relay	✅		
TE	✅	✅	❌
TIR	❌	✅	❌

Current tensor IRs fall short on our requirements!

PyTorch

mxnet

We present our core abstraction, **access patterns.**

We present our core abstraction, **access patterns.**

Around them, we design **Glenside**, a pure, low-level, binder-free tensor IR.

We present our core abstraction, **access patterns.**

Around them, we design **Glenside**, a pure, low-level, binder-free tensor IR.

Finally, we demonstrate how Glenside **enables low-level tensor program rewriting**.

Outline

- Motivating Example: Matrix Multiplication
- Access Pattern Definition
- Case Studies
  - Reimplementing Matrix Multiplication with Access Patterns
  - Implementing 2D Convolution with Access Patterns
  - Hardware Mapping as Program Rewriting
  - Flexible Hardware Mapping with Equality Saturation

Outline

- Motivating Example: Matrix Multiplication
- Access Pattern Definition
- Case Studies
    - Reimplementing Matrix Multiplication with Access Patterns
    - Implementing 2D Convolution with Access Patterns
    - Hardware Mapping as Program Rewriting
    - Flexible Hardware Mapping with Equality Saturation

We want to represent matrix multiplication in a way that

We want to represent matrix multiplication in a way that

1. is pure,

We want to represent matrix multiplication in a way that

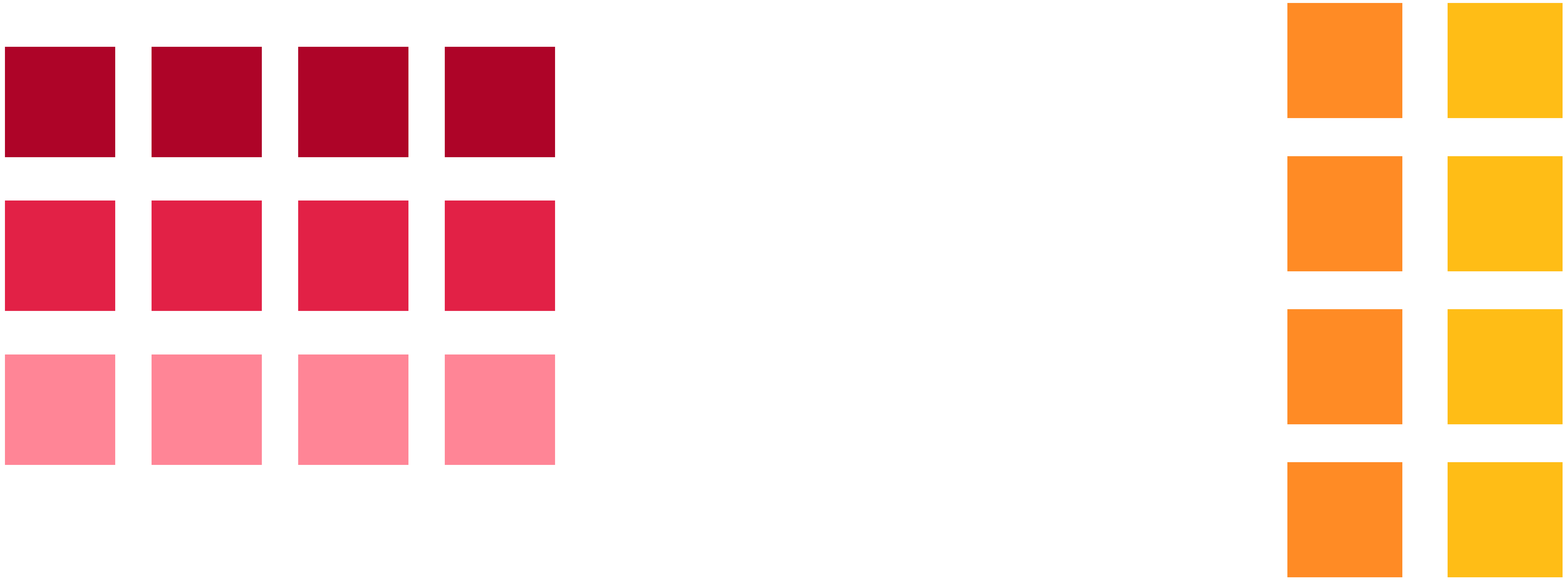
1. is pure,

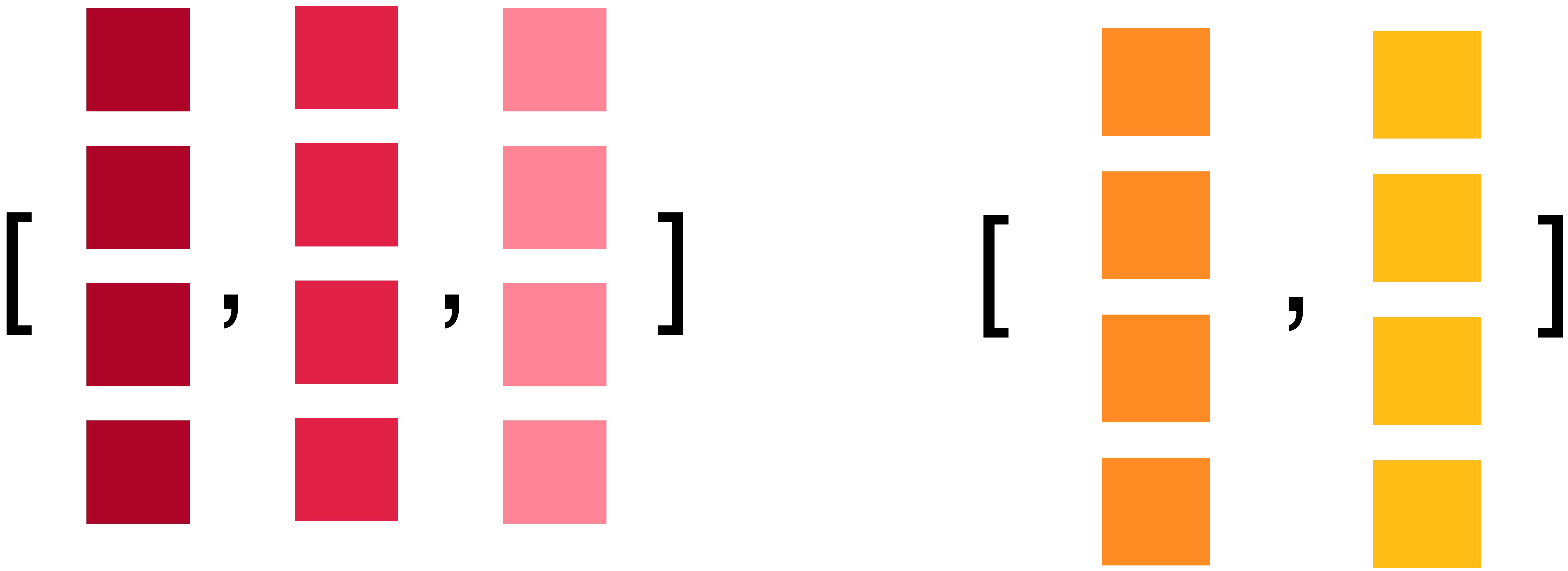
2. is low-level, and

We want to represent matrix multiplication in a way that

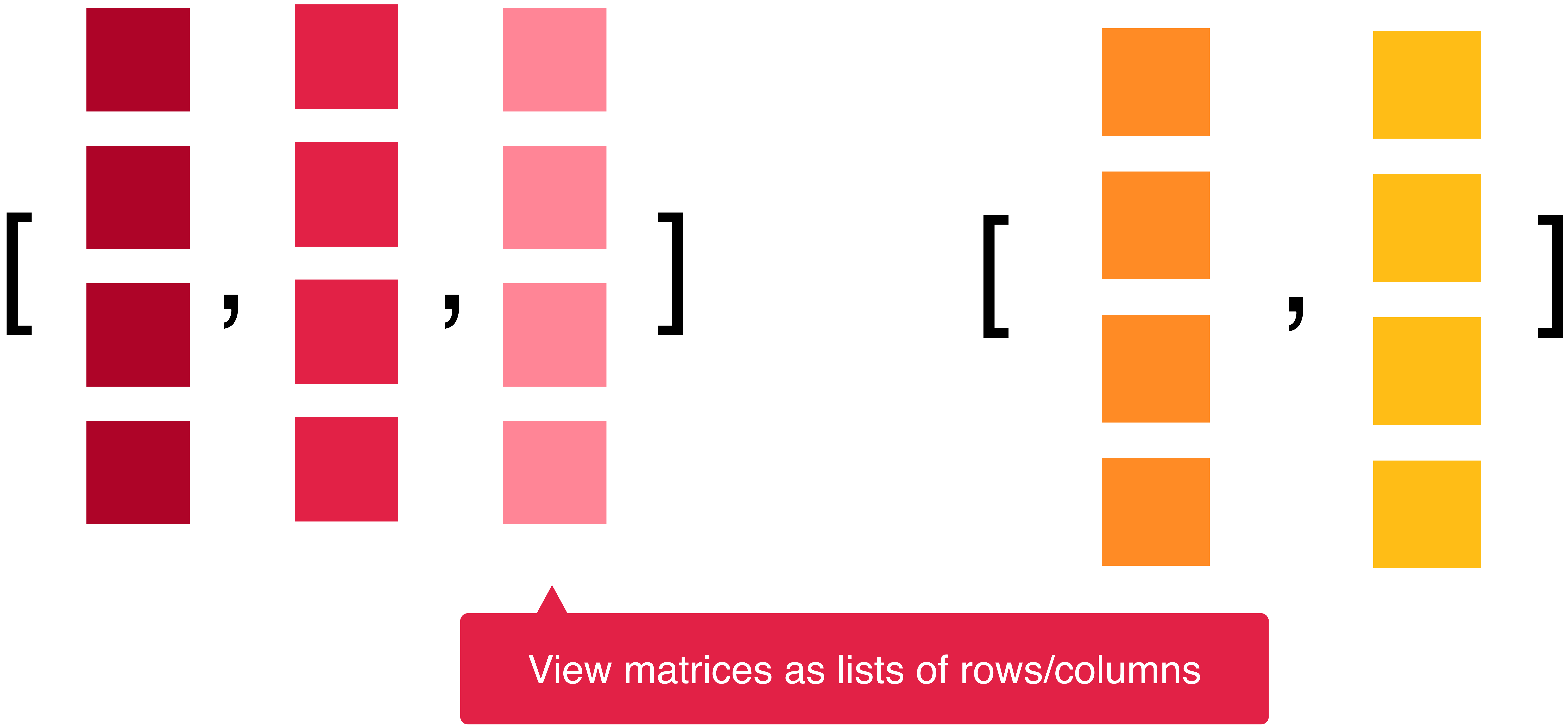
1. is pure,
2. is low-level, and
3. avoids binding.

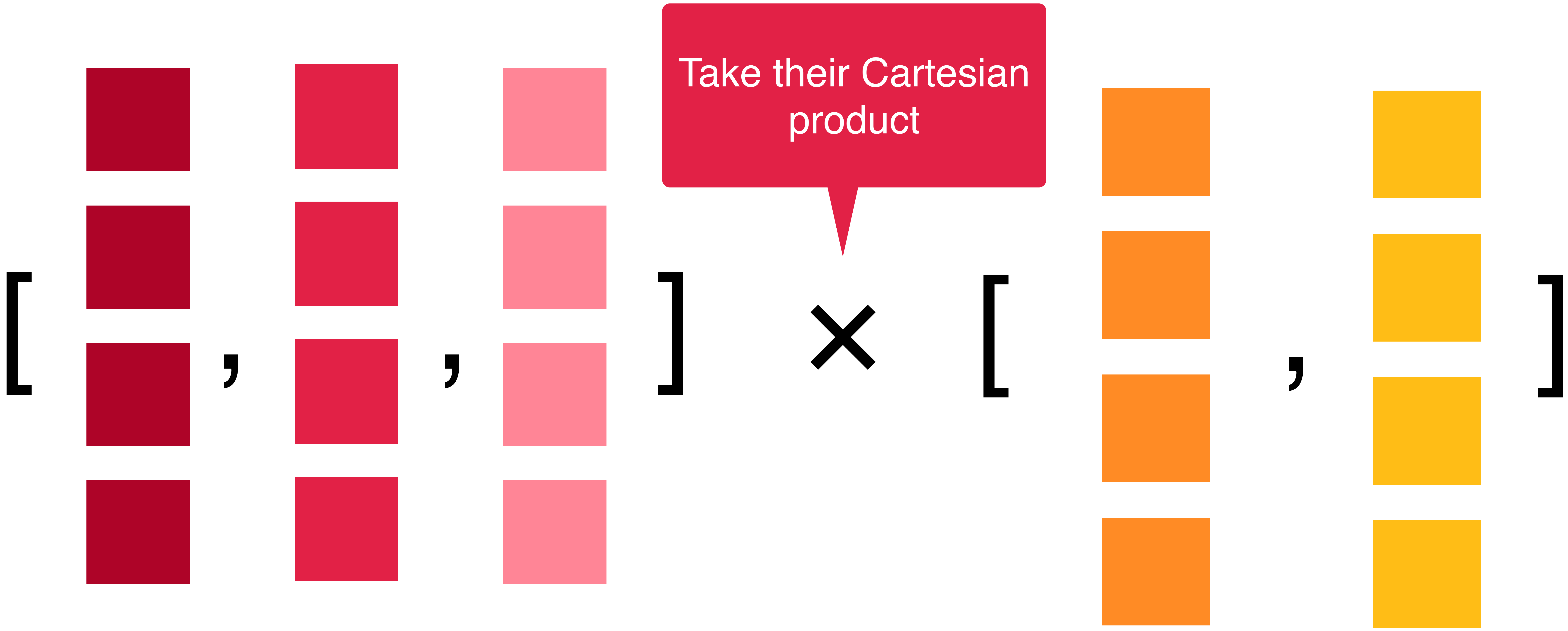
Given matrices A and B, pair each row of A with each column of B, compute their dot products, and arrange the results back into a matrix.

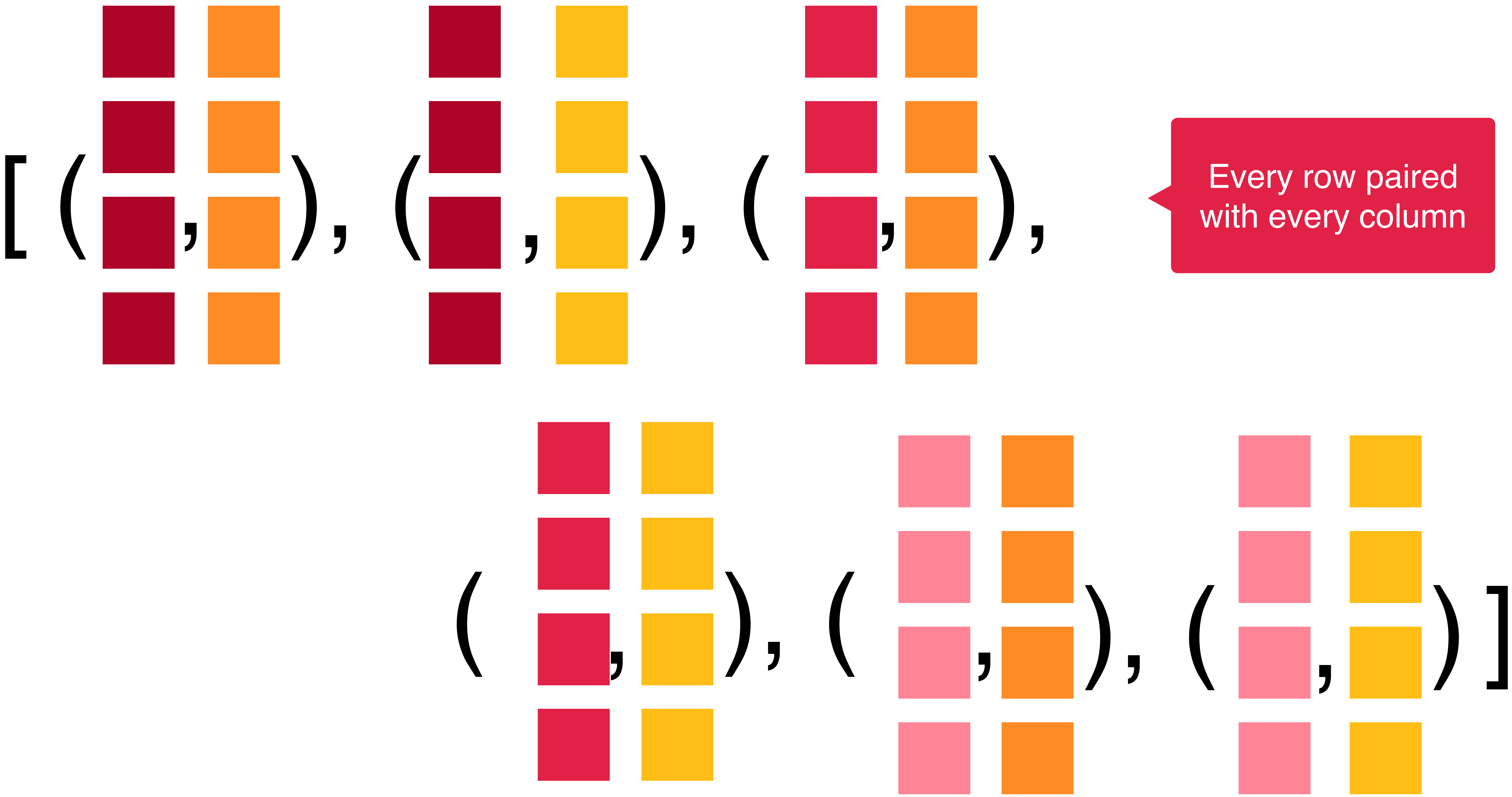




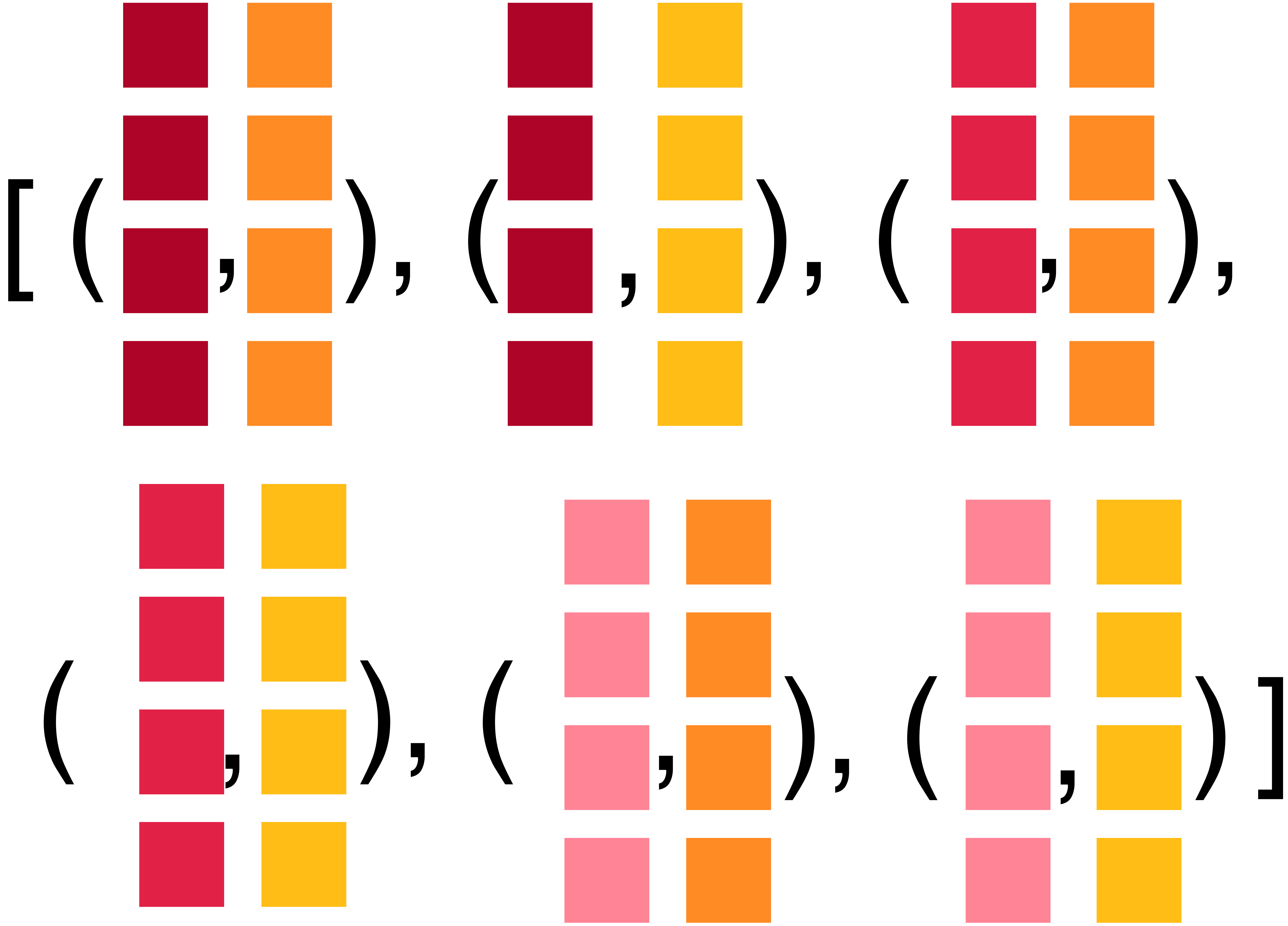
View matrices as lists of rows/columns


View matrices as lists of rows/columns

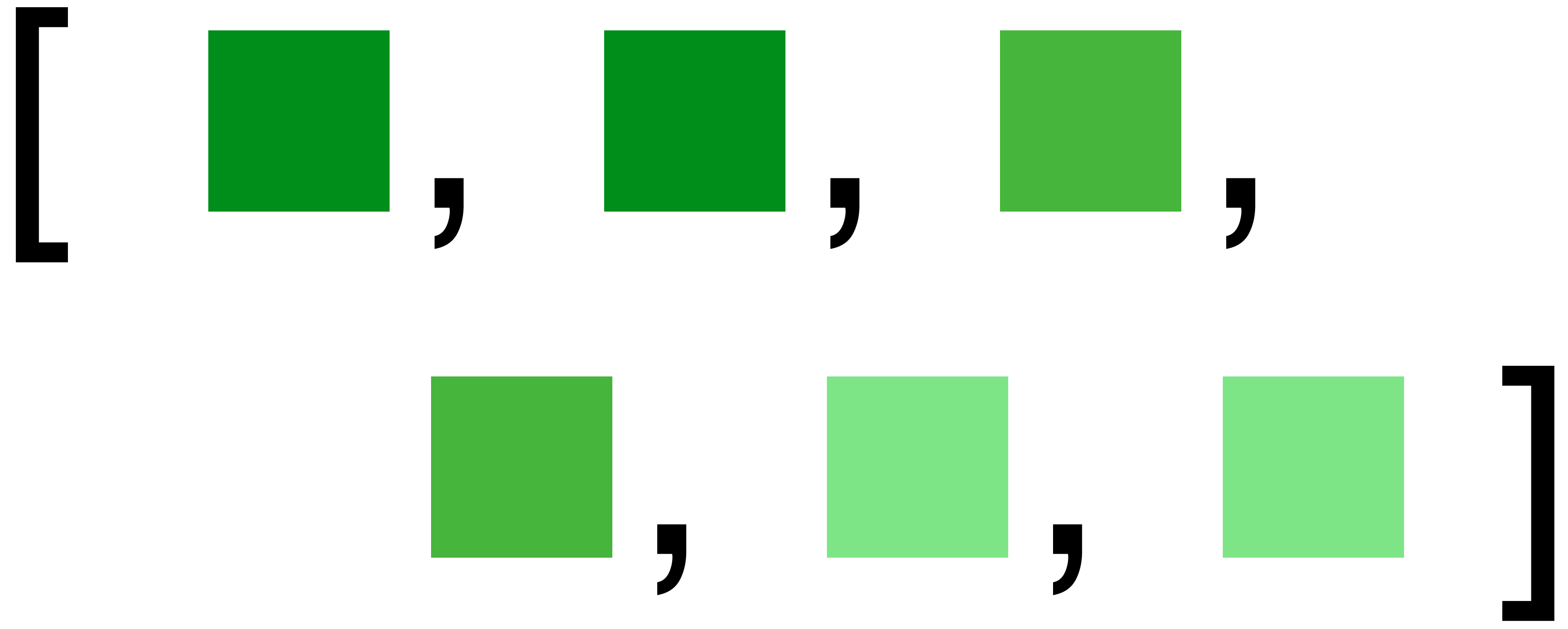

Take their Cartesian product


Every row paired
with every column

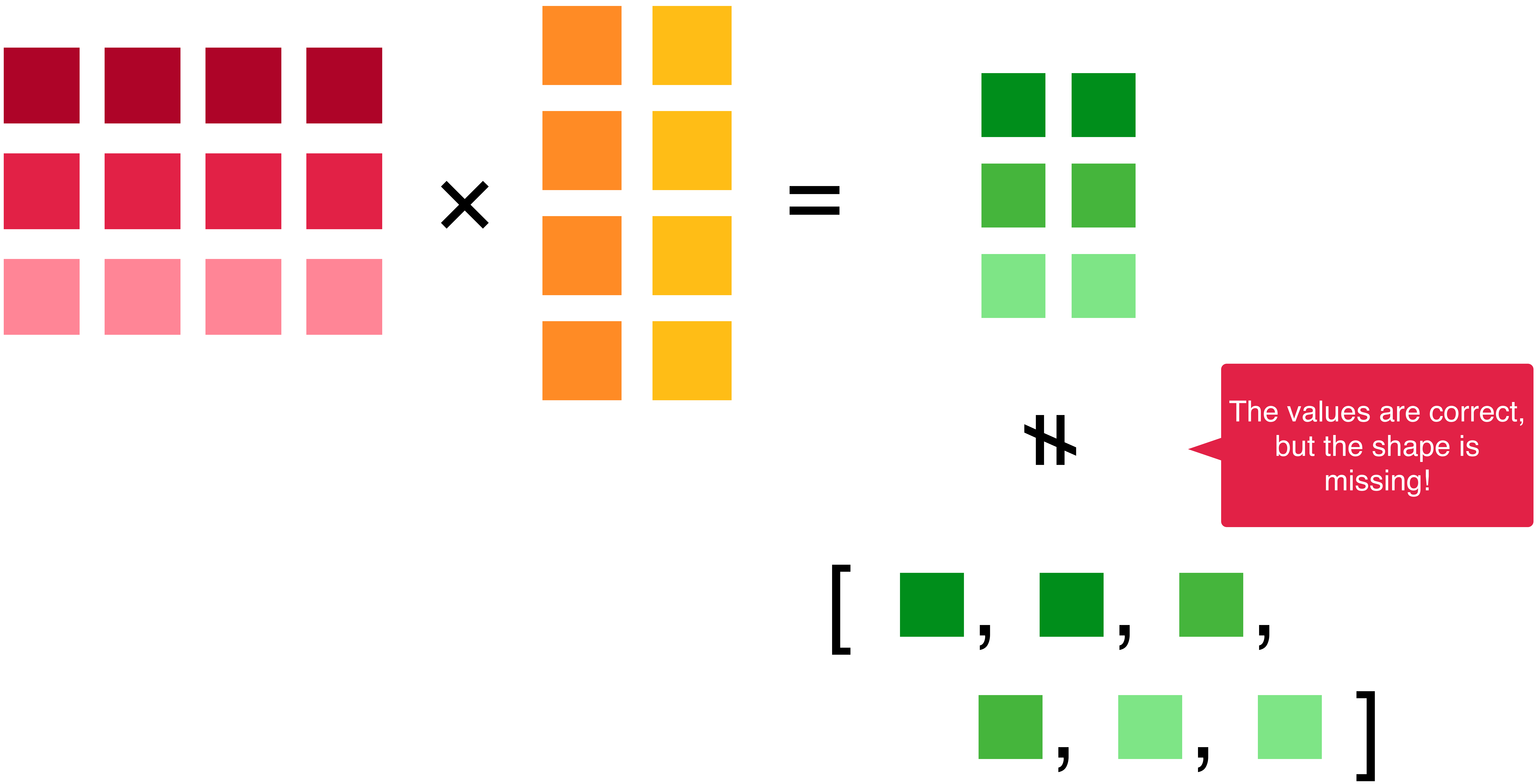
```
map dotProd
```

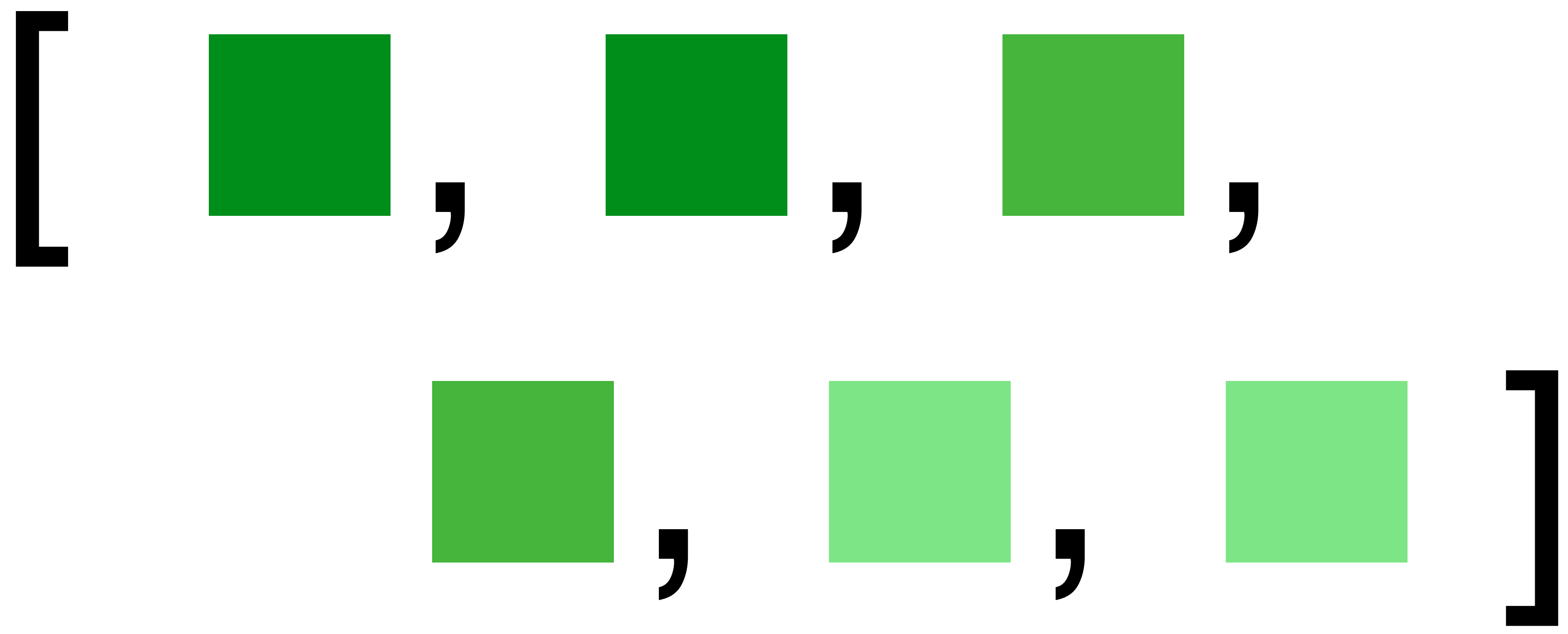


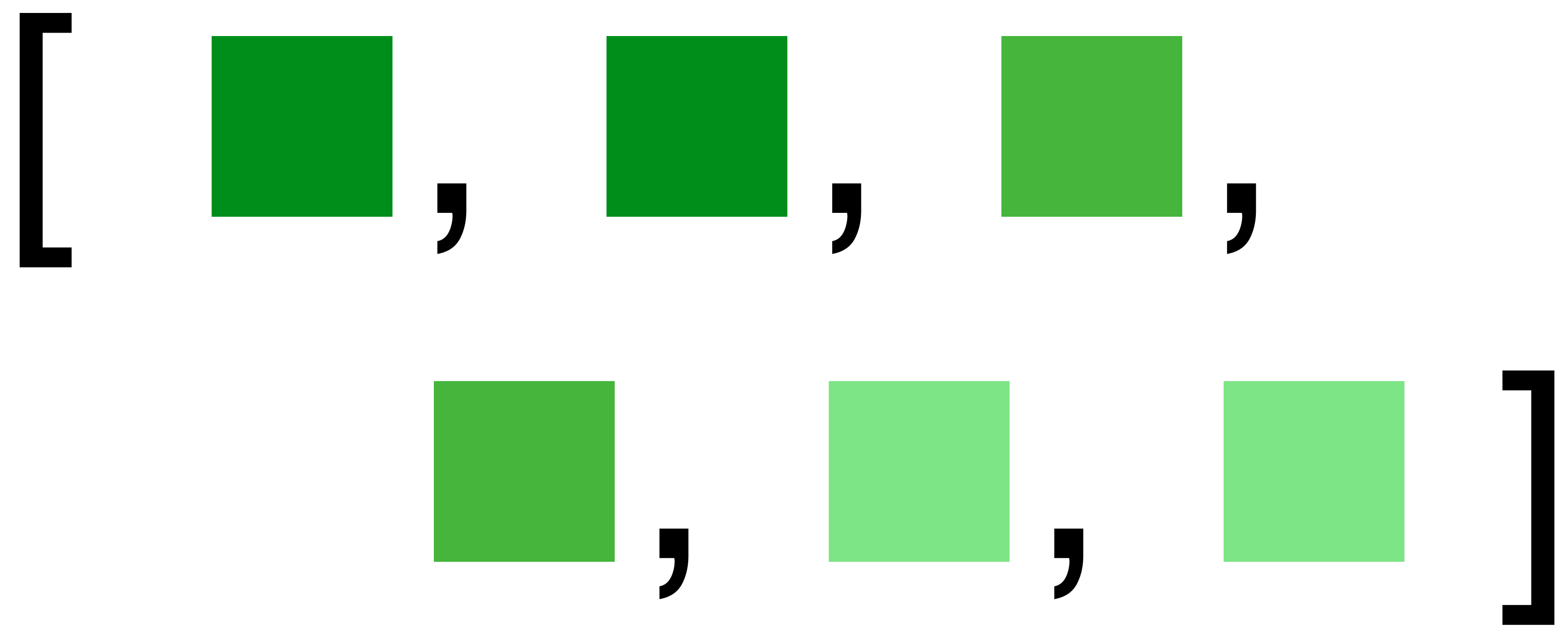
Map dot product operator over every row–column pair



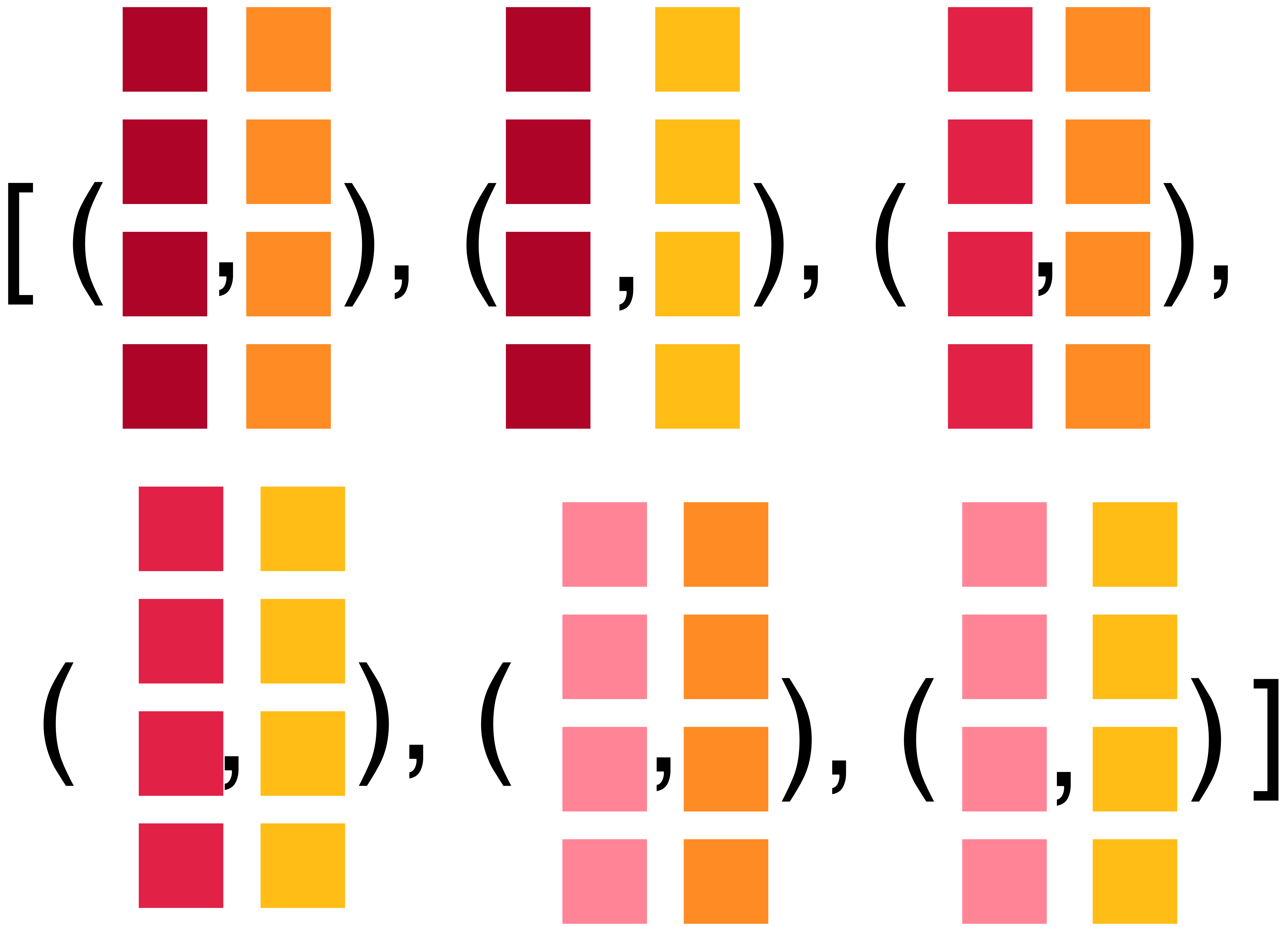
But there’s a problem!

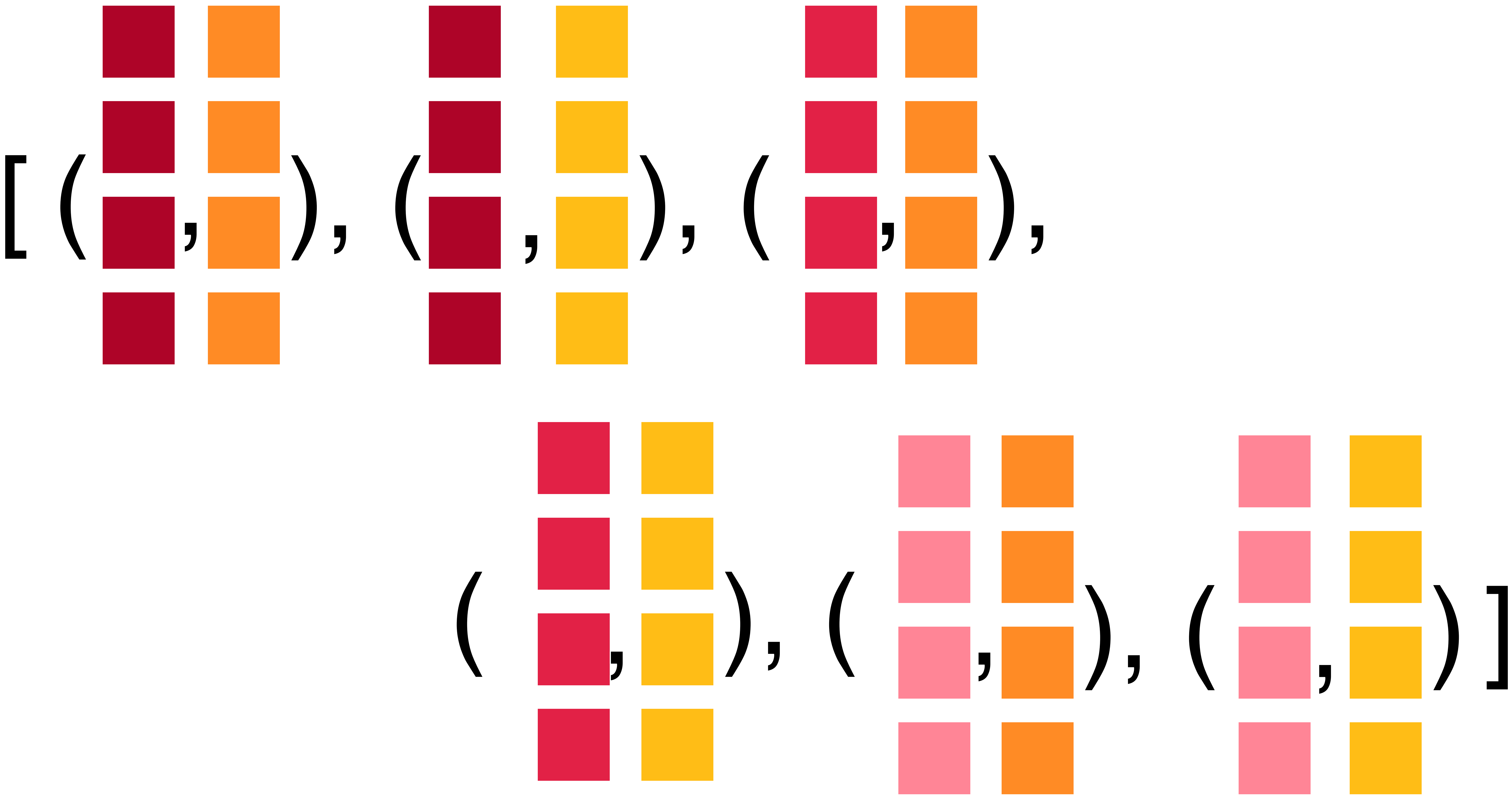

The values are correct,
but the shape is
missing!

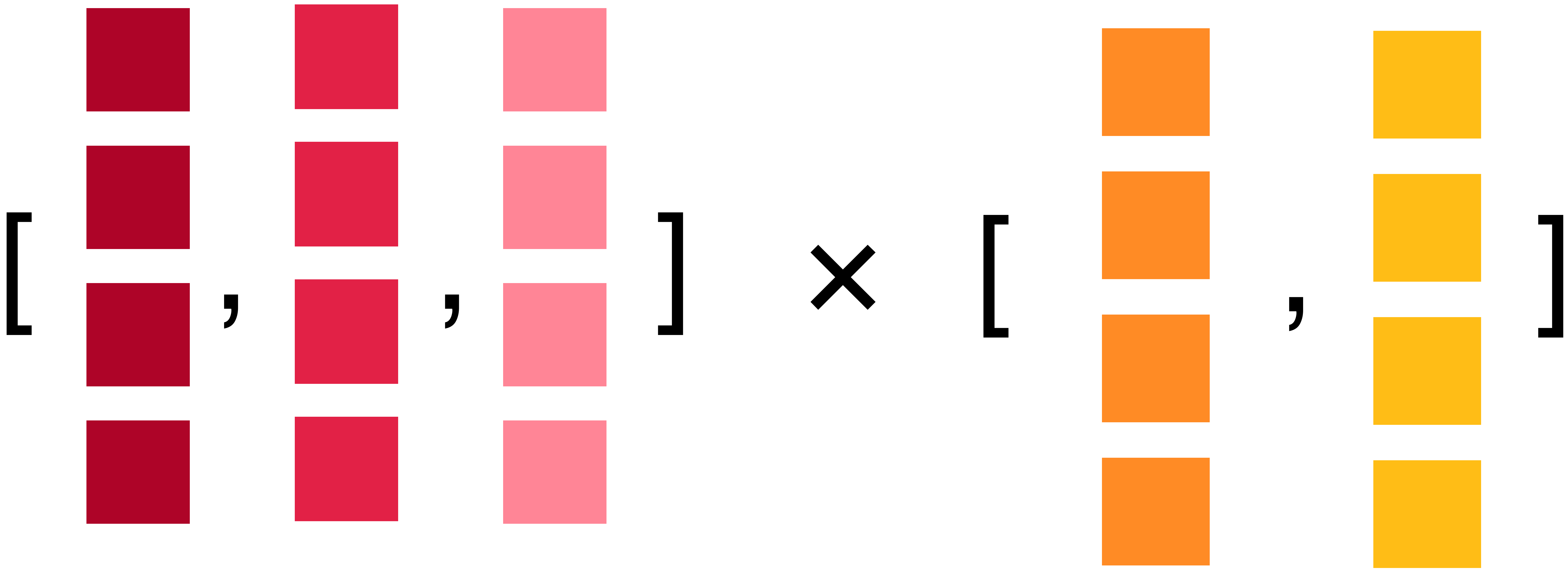


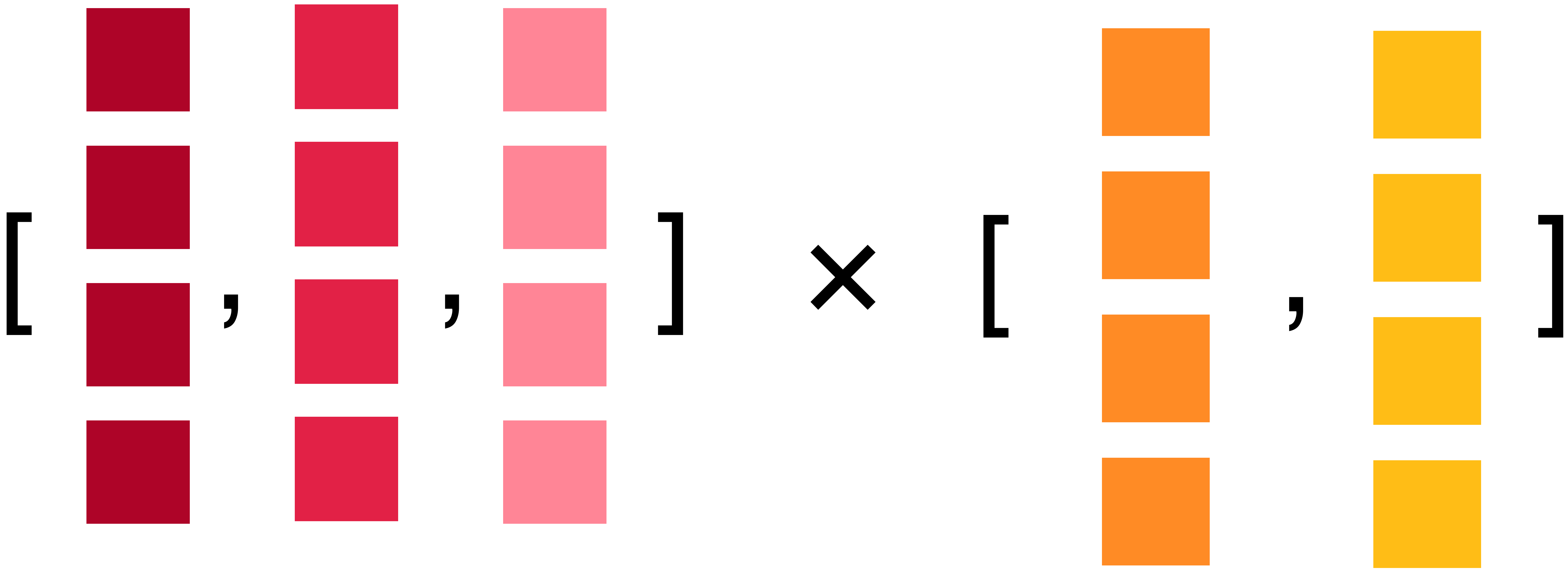


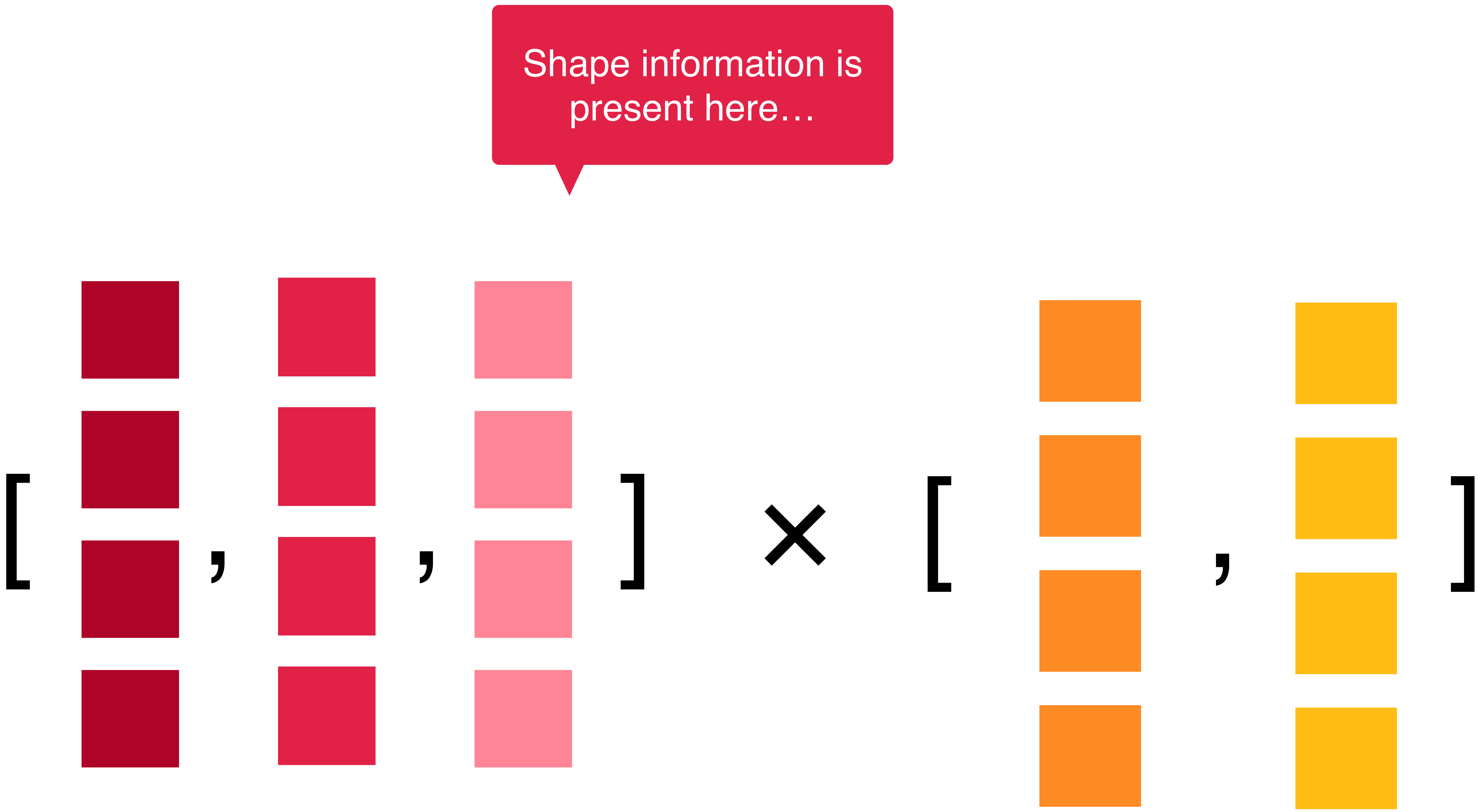
map dot-product

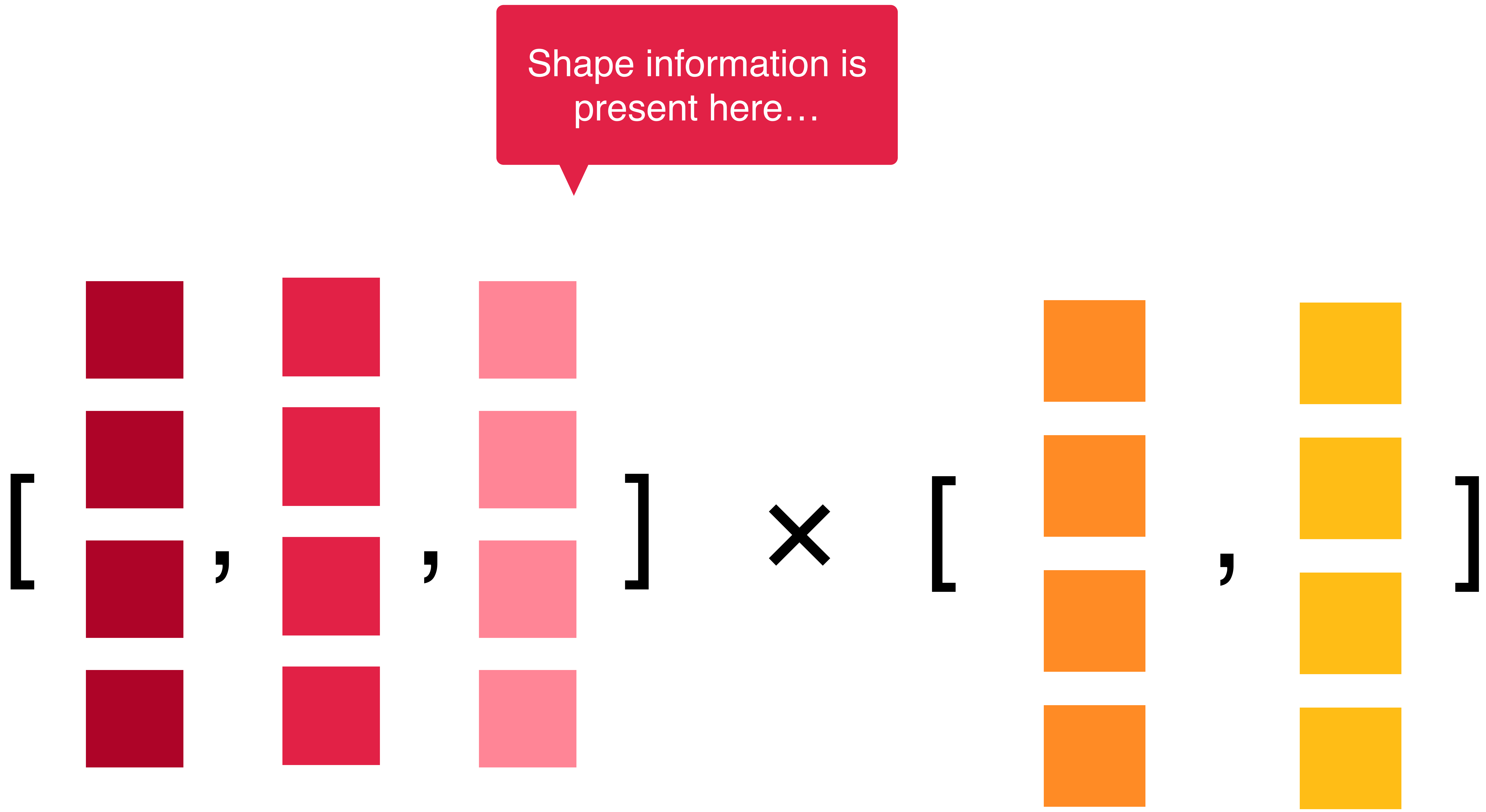


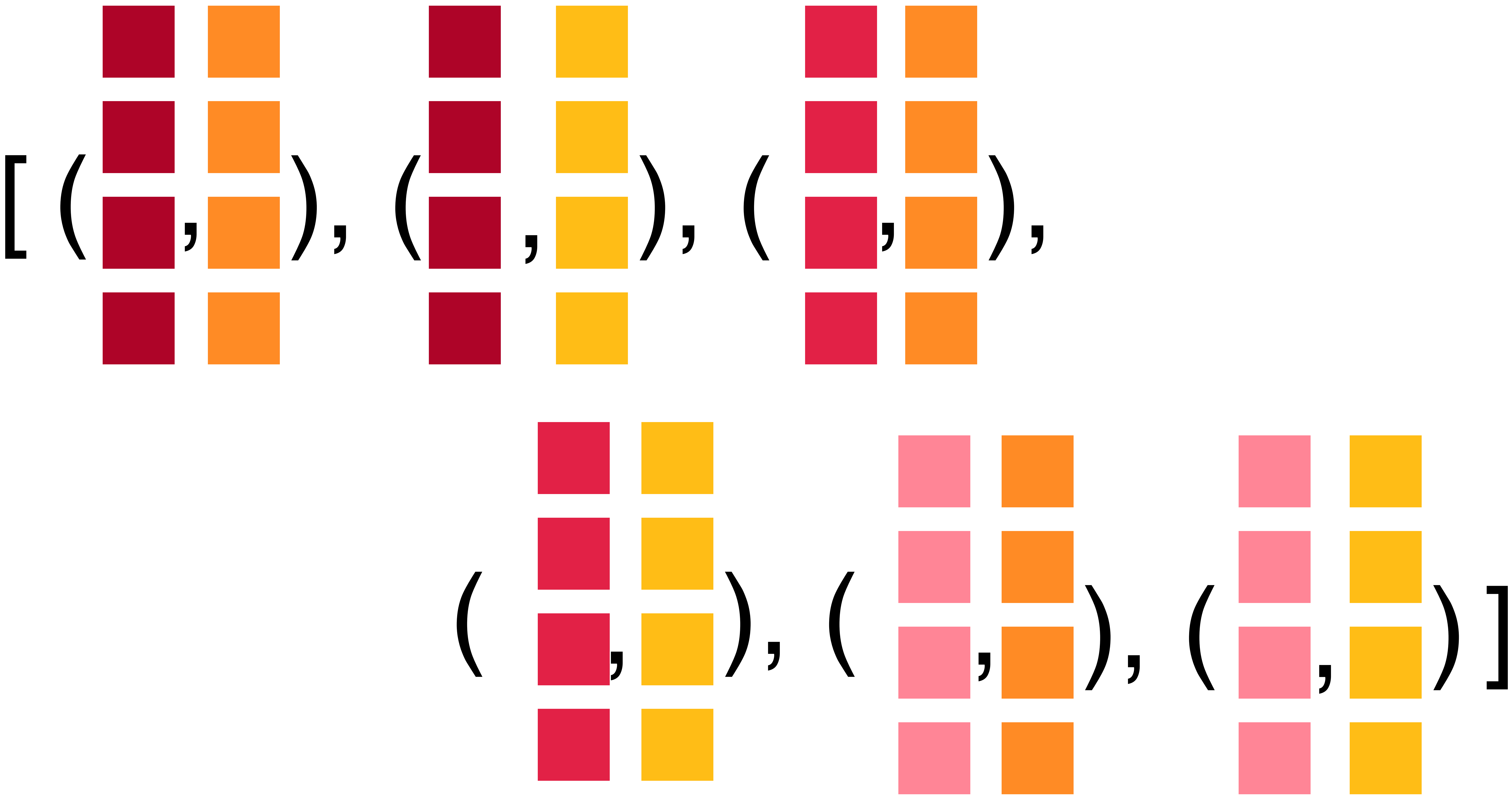


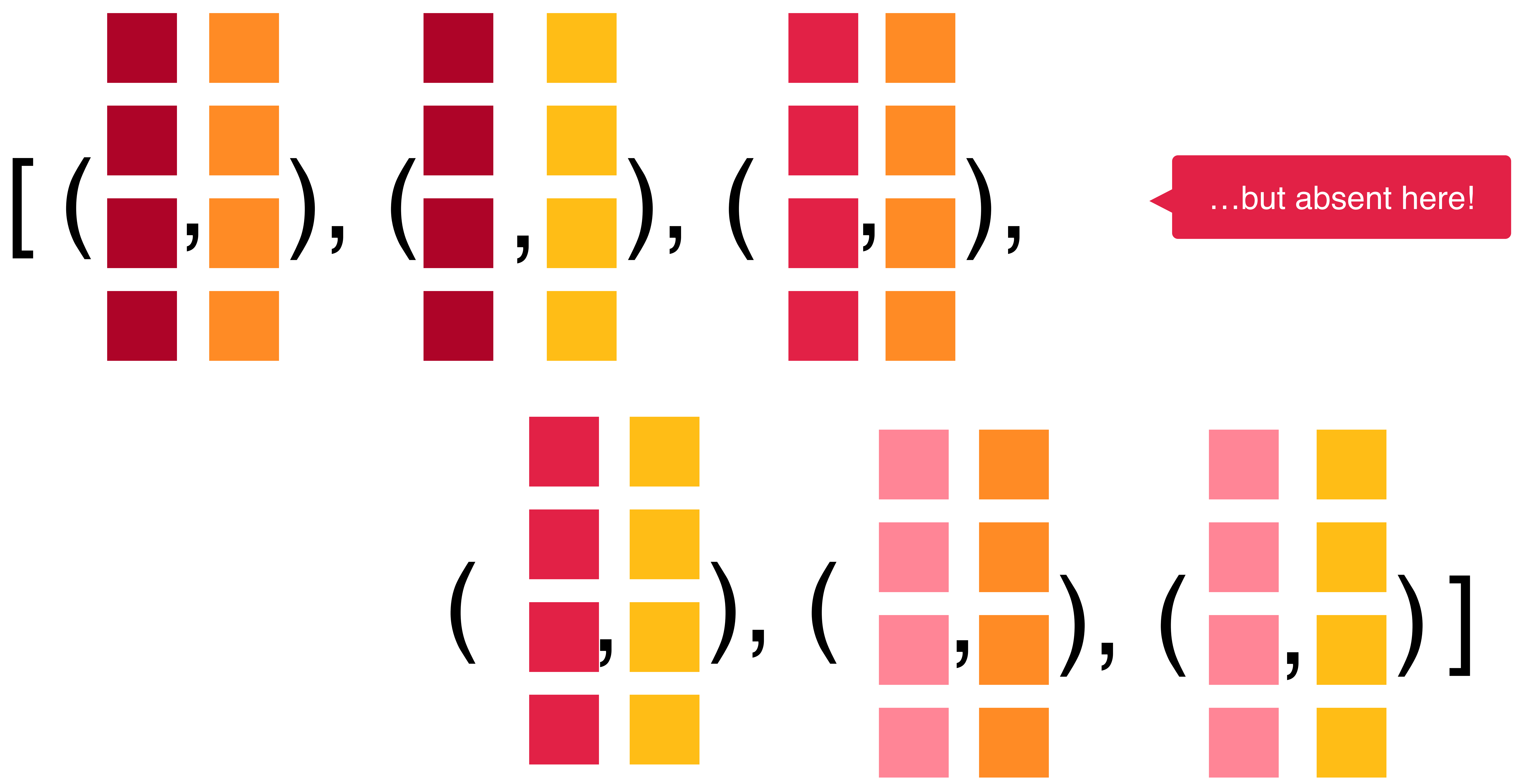




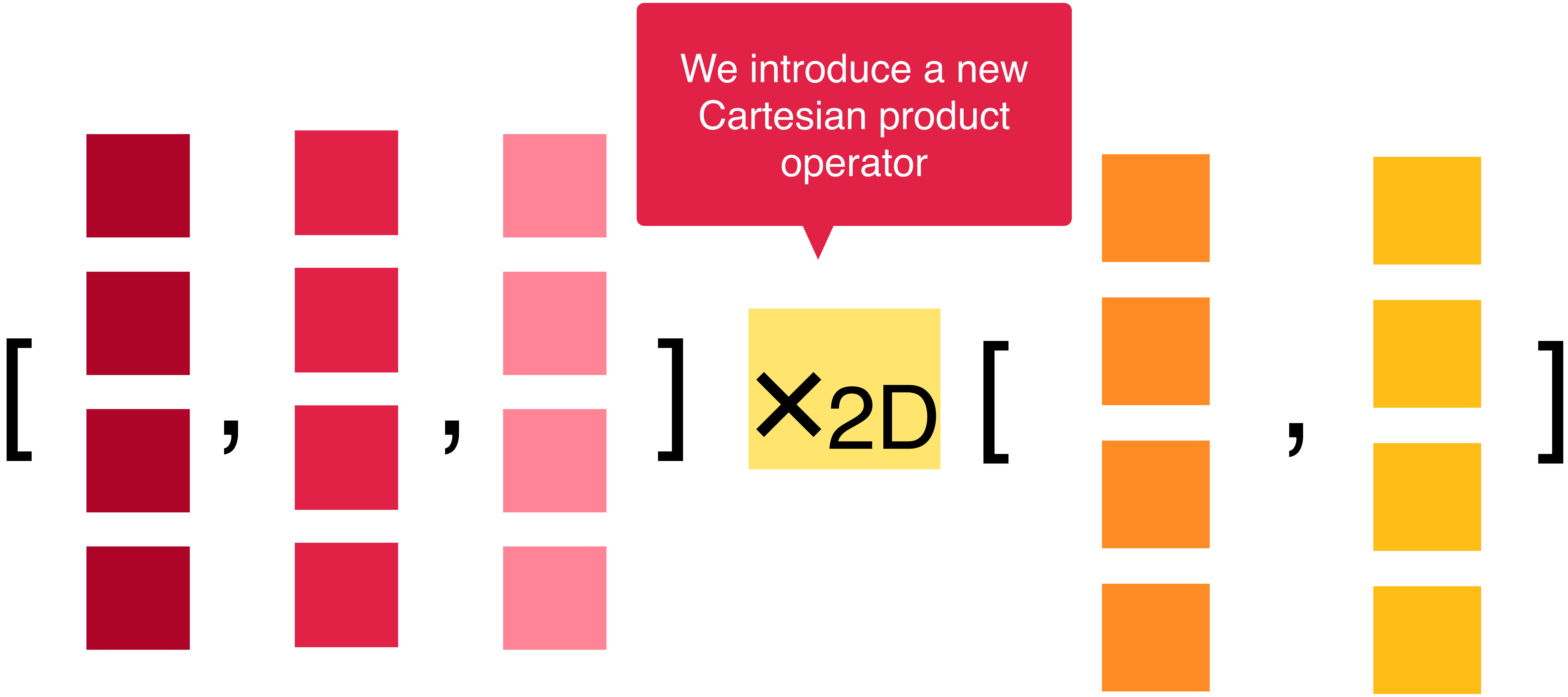

Shape information is present here…

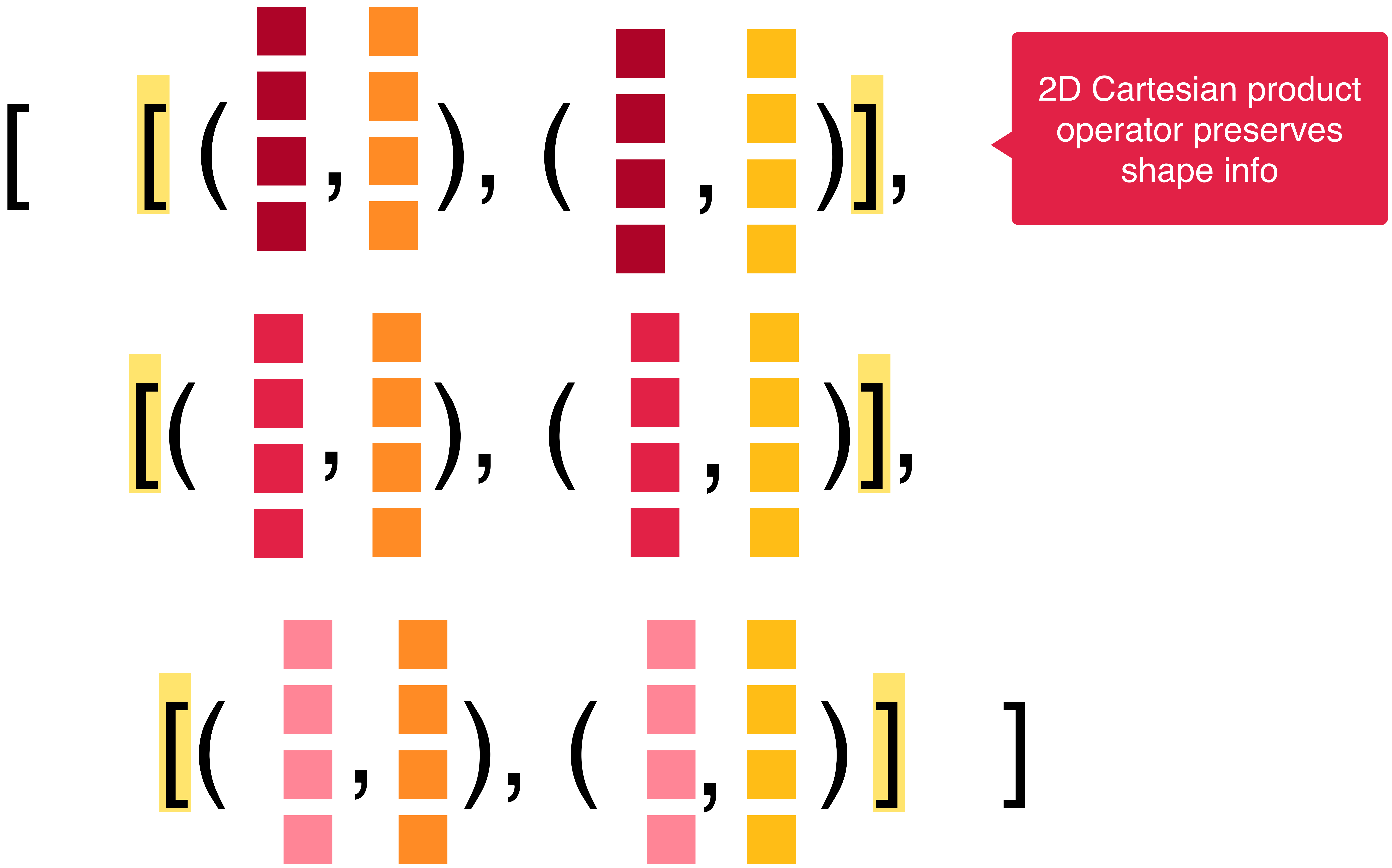

Shape information is present here…

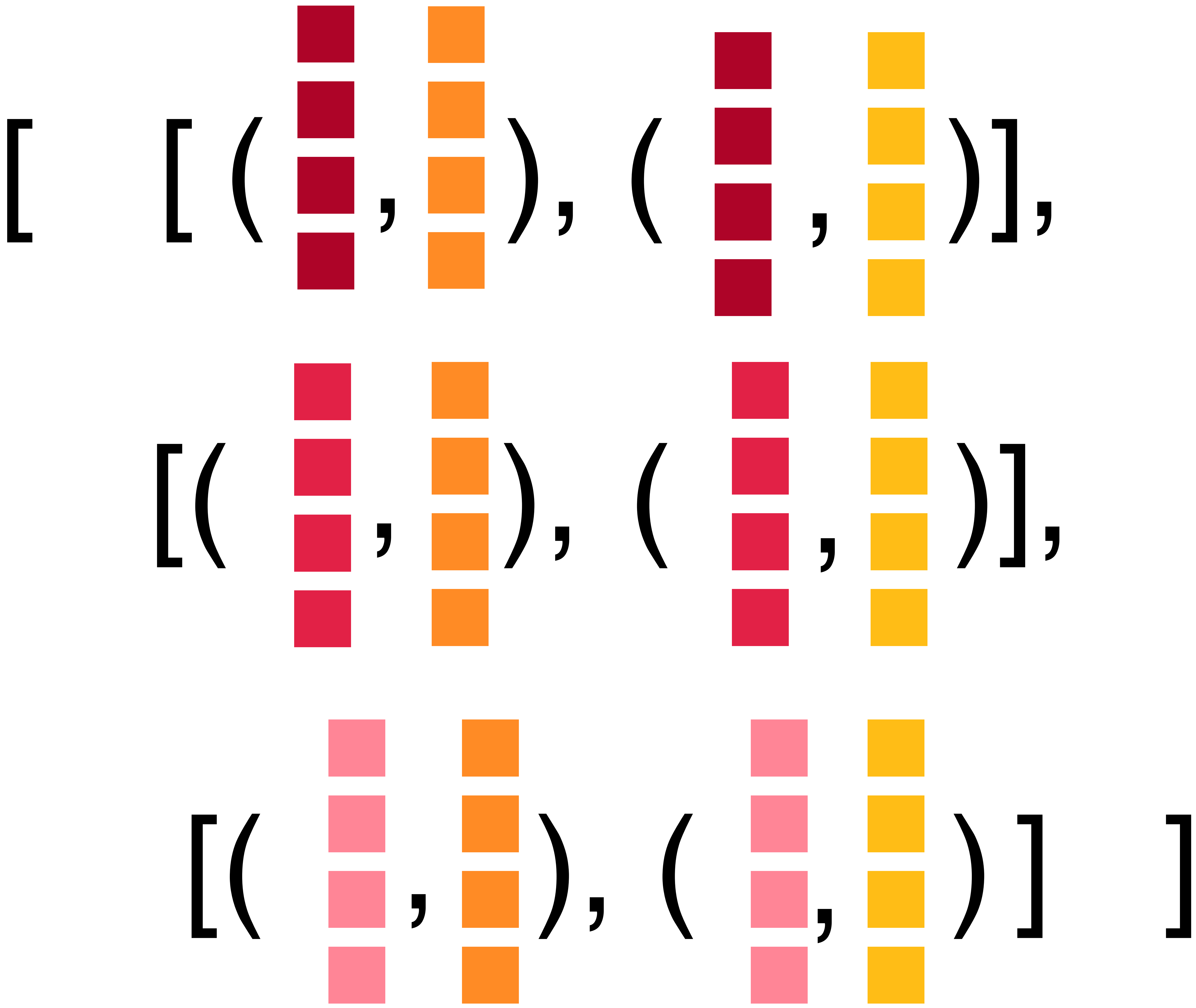


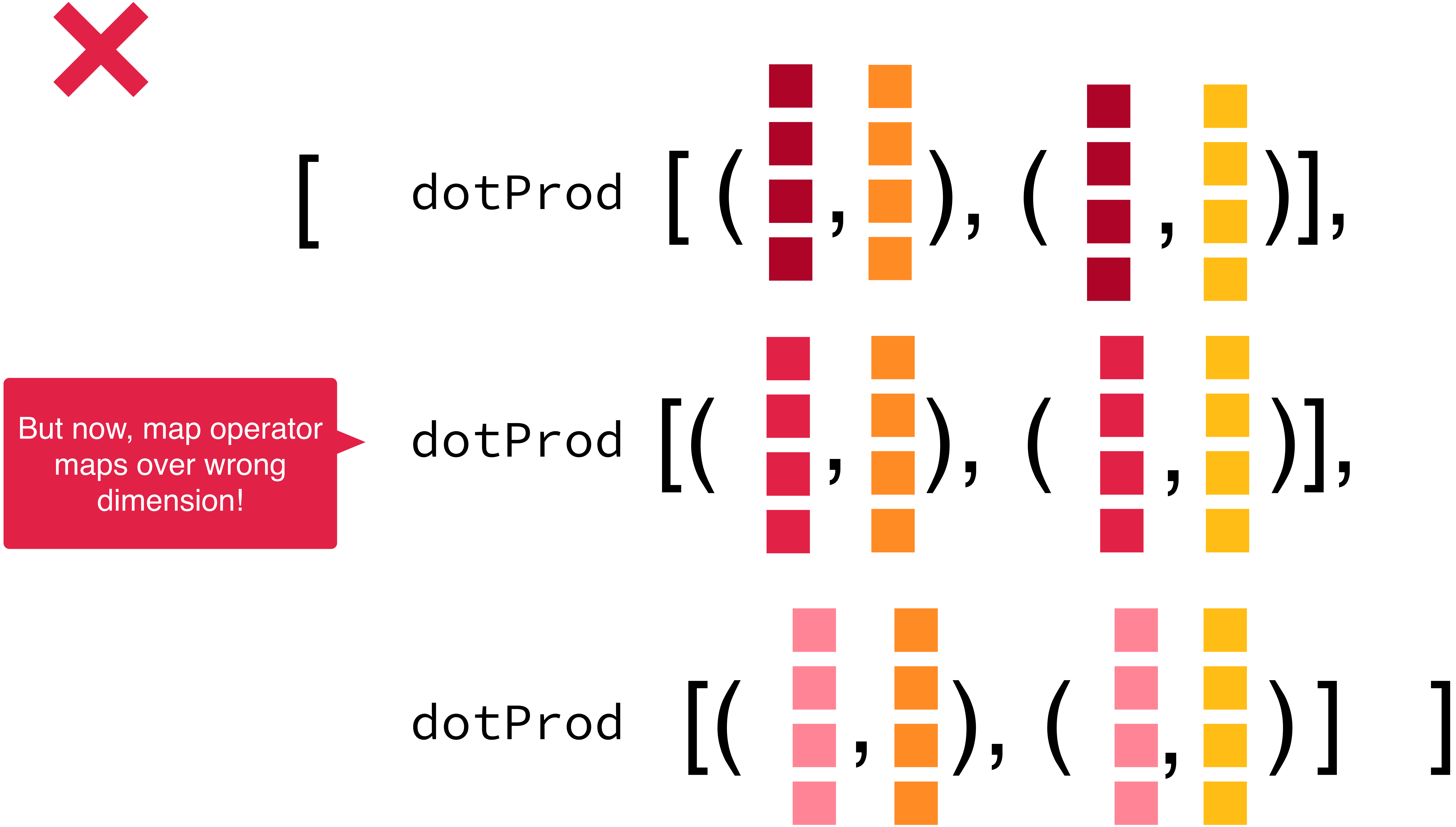

…but absent here!

Cartesian product destroys our shape information!


We introduce a new Cartesian product operator
$\times_{2D}$

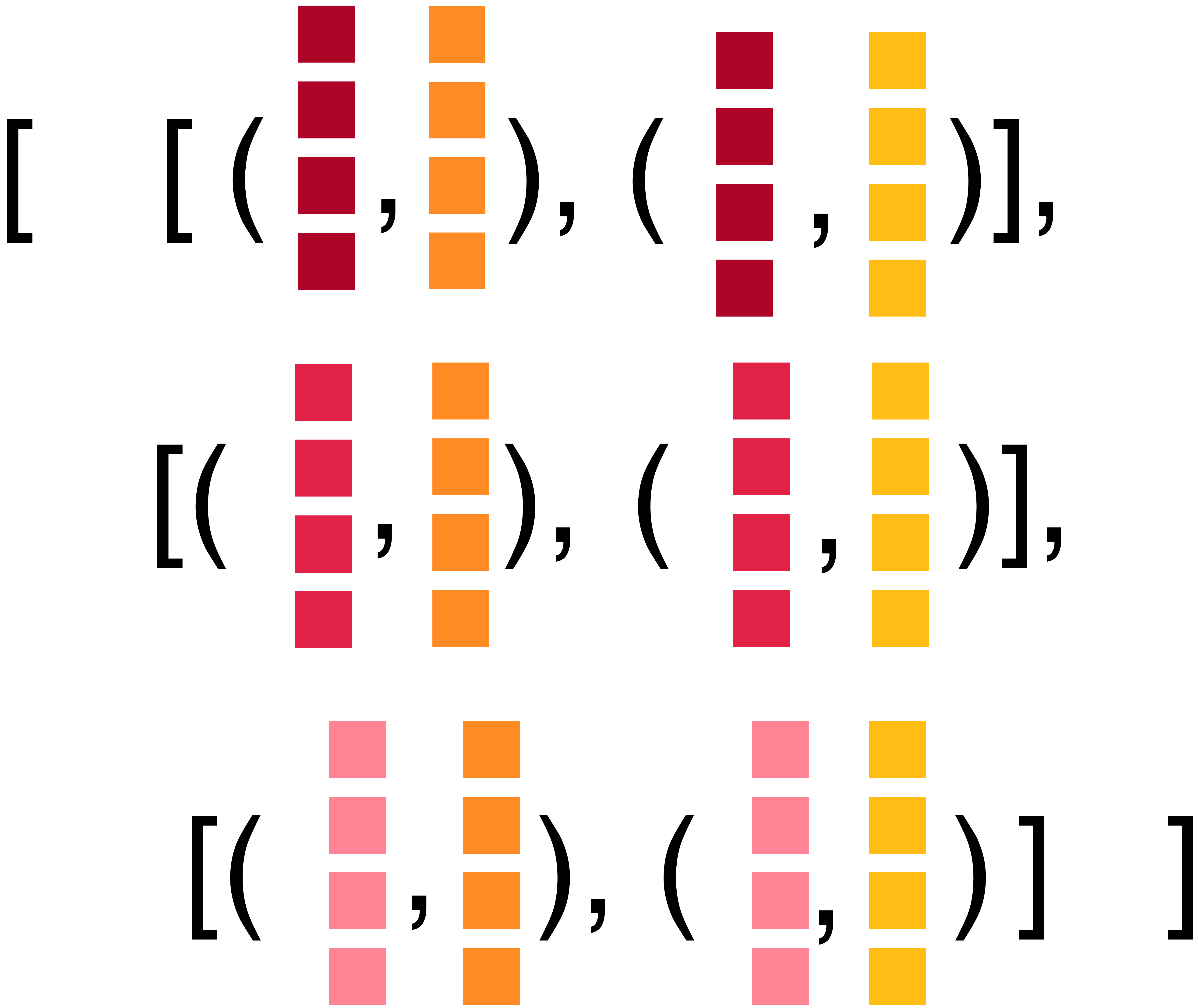

2D Cartesian product
operator preserves
shape info

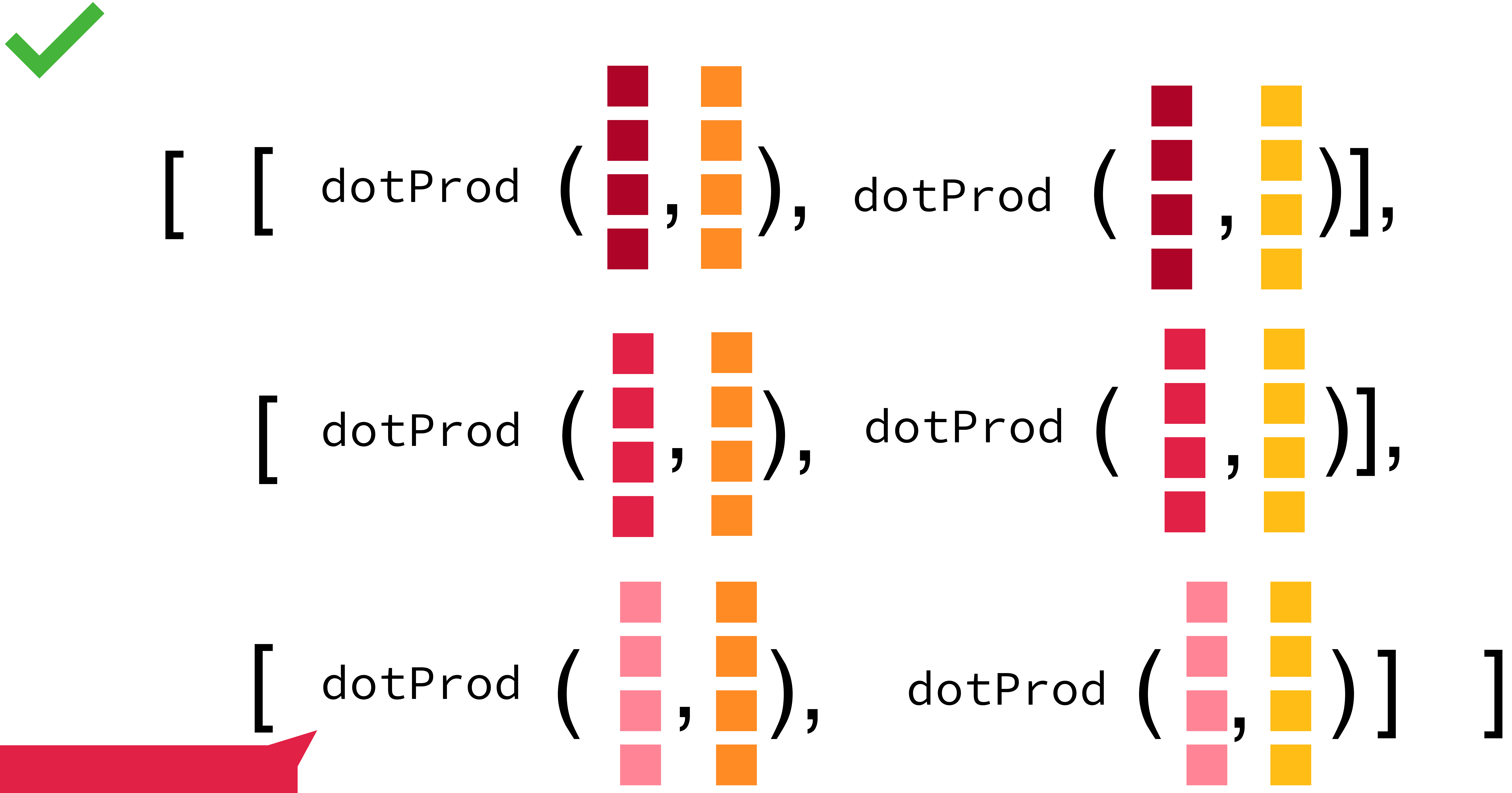
map dotProd 


But now, map operator maps over wrong dimension!
[dotProd [(,), (,)],
dotProd [(,), (,)],
dotProd [(,), (,)]]

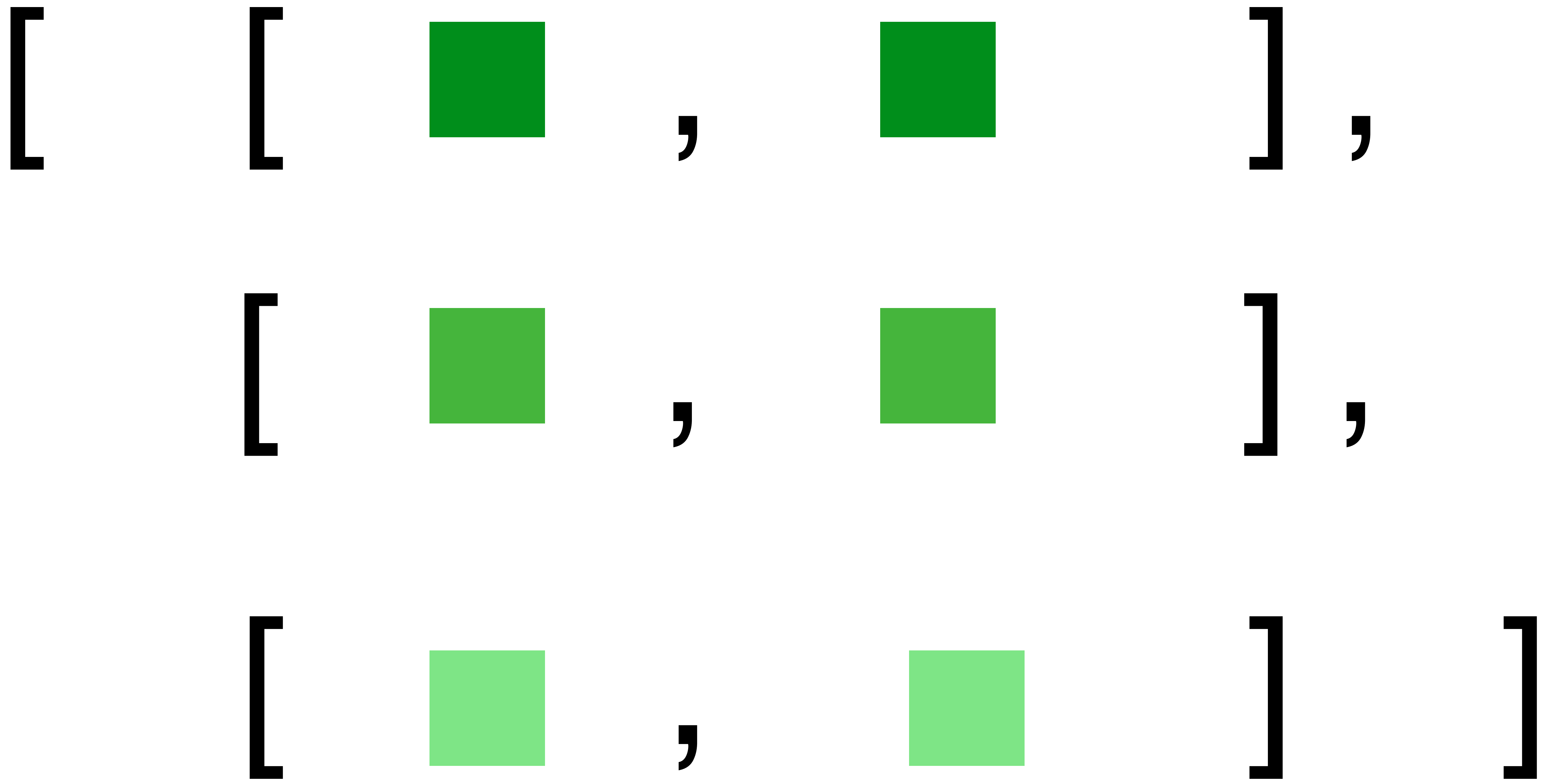
`map2D dotProd`

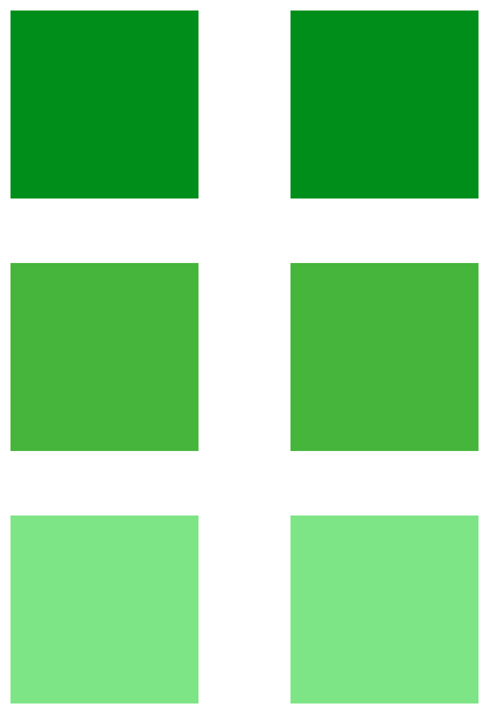
We also need a new map operator

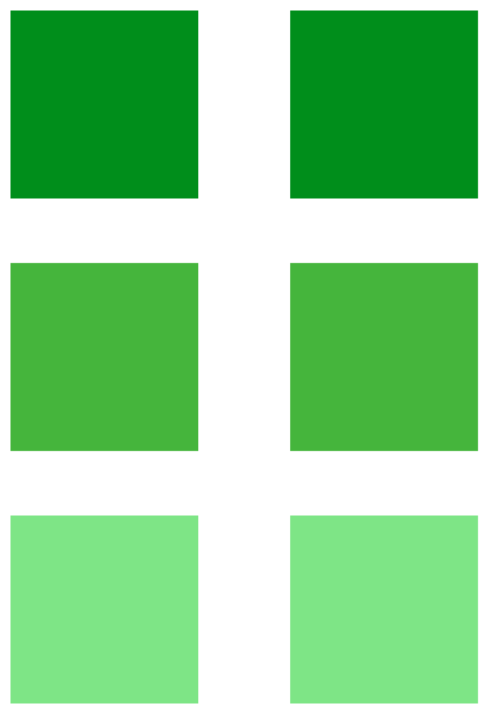




2D map operator maps over correct dimension







Shape information is preserved!

$\times_{2D}$ and `map2D` hard-code which dimensions are **iterated over** and which dimensions are **computed on**…

$\times_{2D}$ and `map2D` hard-code which dimensions are **iterated over** and which dimensions are **computed on**…

…but if tensor shapes change, we'll need entirely new operators!

$\times_{2D}$ and `map2D` hard-code which dimensions are **iterated over** and which dimensions are **computed on**…

…but if tensor shapes change, we'll need entirely new operators!

Can we encode this in the tensor itself?

$\times_{2D}$ and `map2D` hard-code which dimensions are **iterated over** and which dimensions are **computed on**…

…but if tensor shapes change, we'll need entirely new operators!

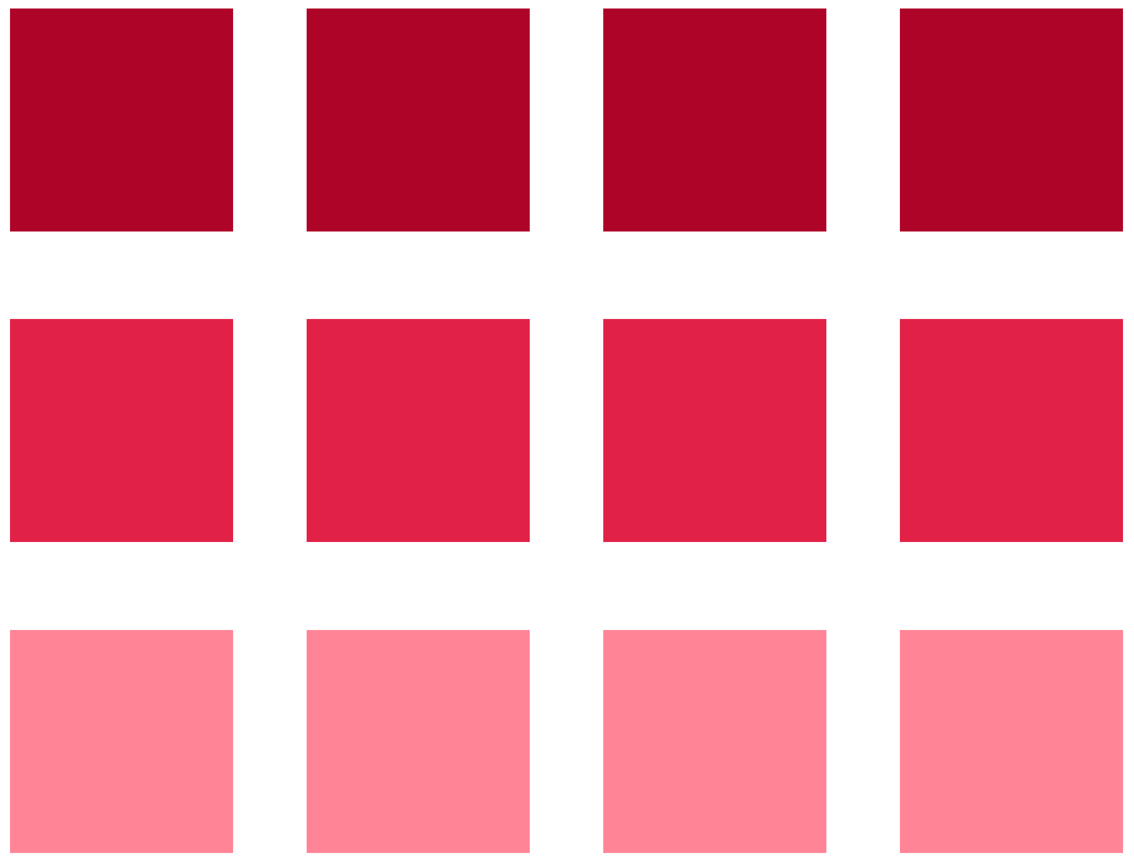
Can we encode this in the tensor itself?

(Yes! This is what access patterns do!)

Outline

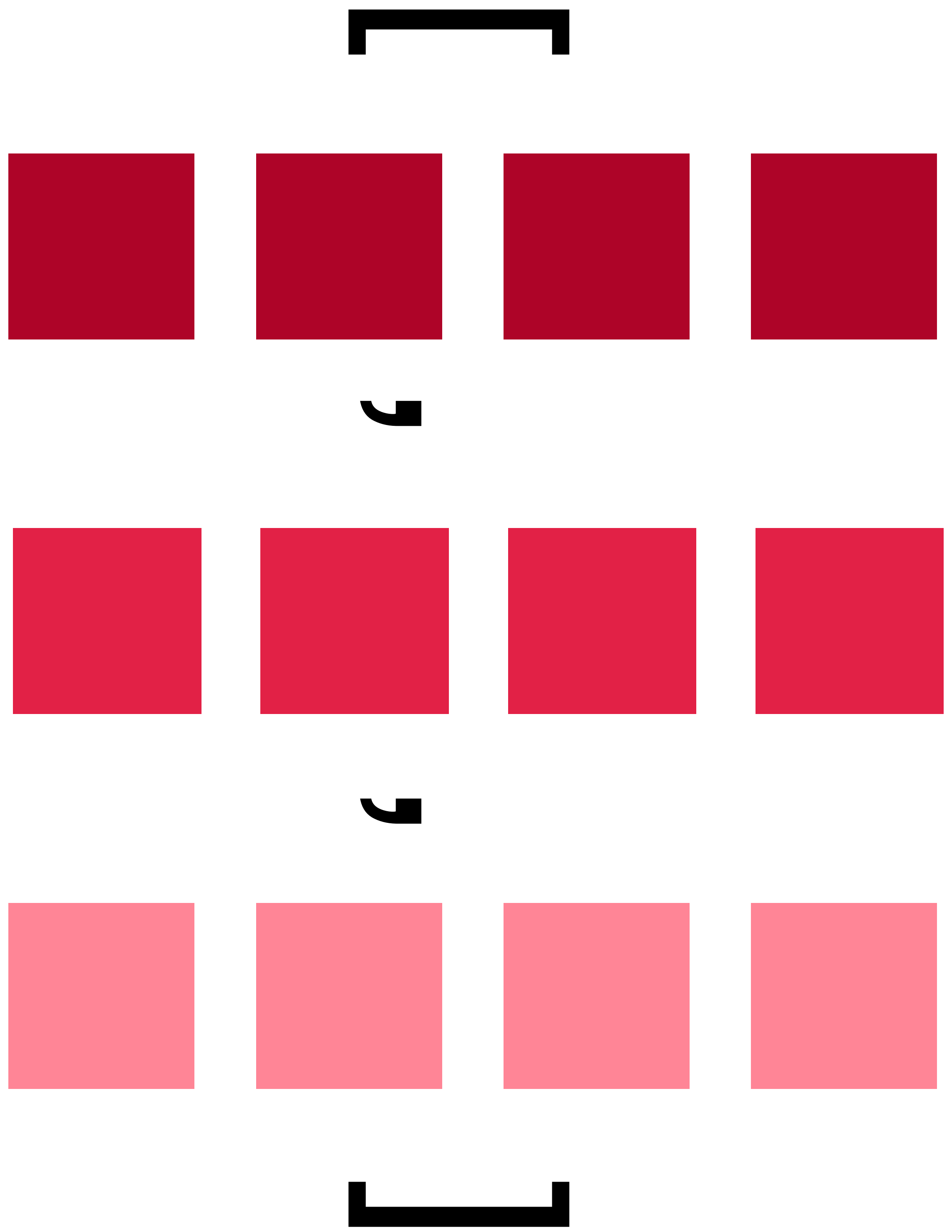
- Motivating Example: Matrix Multiplication
- Access Pattern Definition
- Case Studies
    - Reimplementing Matrix Multiplication with Access Patterns
    - Implementing 2D Convolution with Access Patterns
    - Hardware Mapping as Program Rewriting
    - Flexible Hardware Mapping with Equality Saturation

A **tensor** looks like…



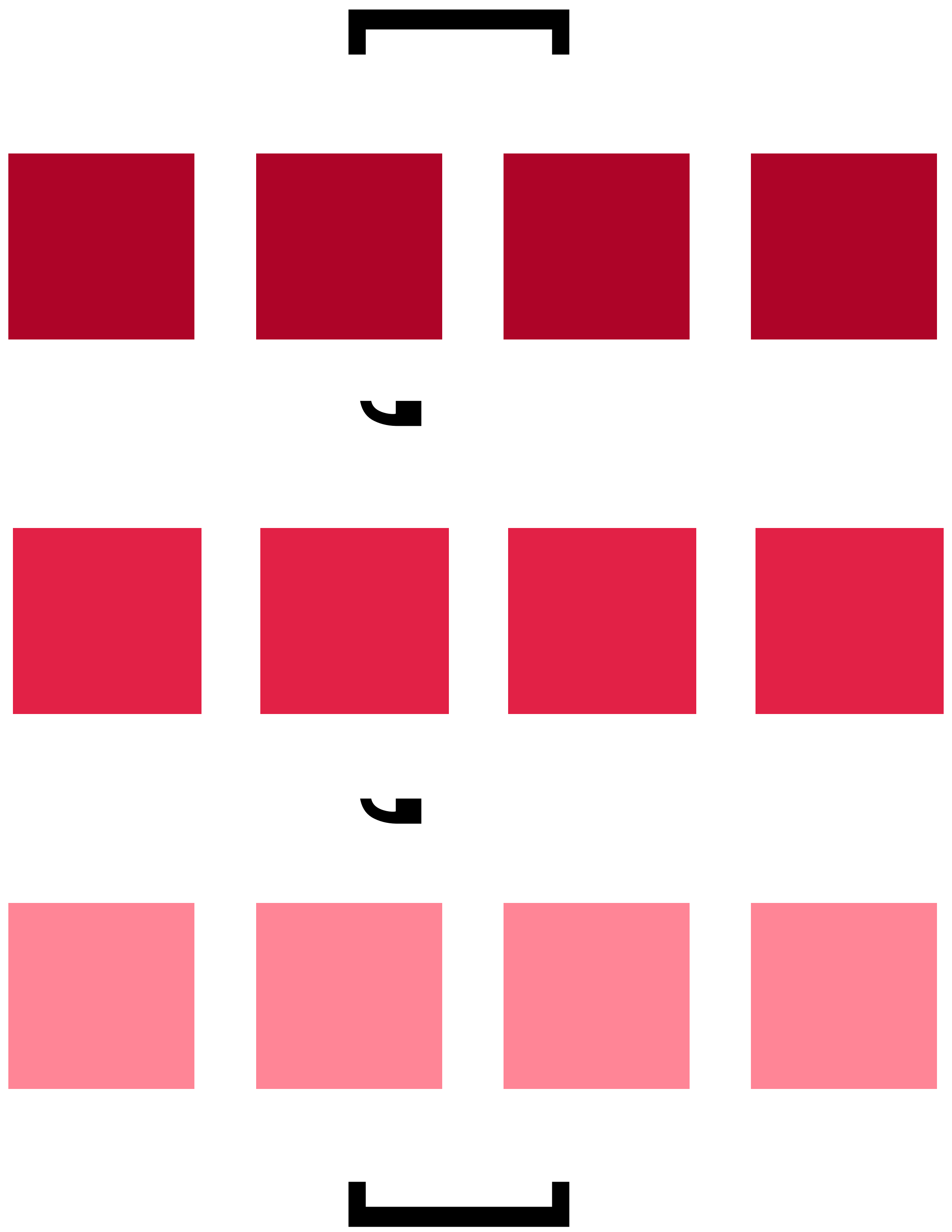
(3, 4)

An **access pattern** looks like…



((3), (4))

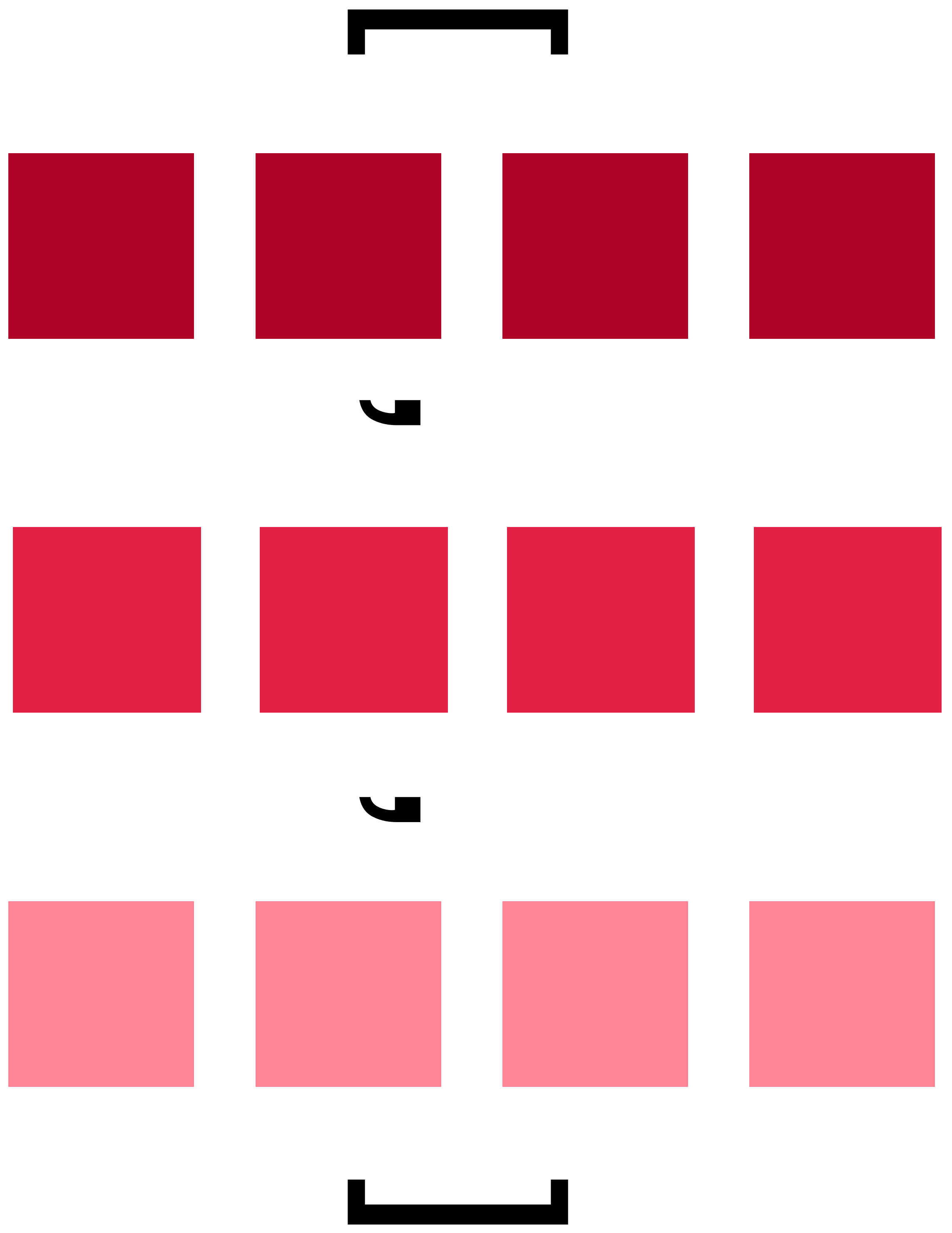
An **access pattern** looks like…



access dimensions
(iterated over)

((3), (4))

An **access pattern** looks like…

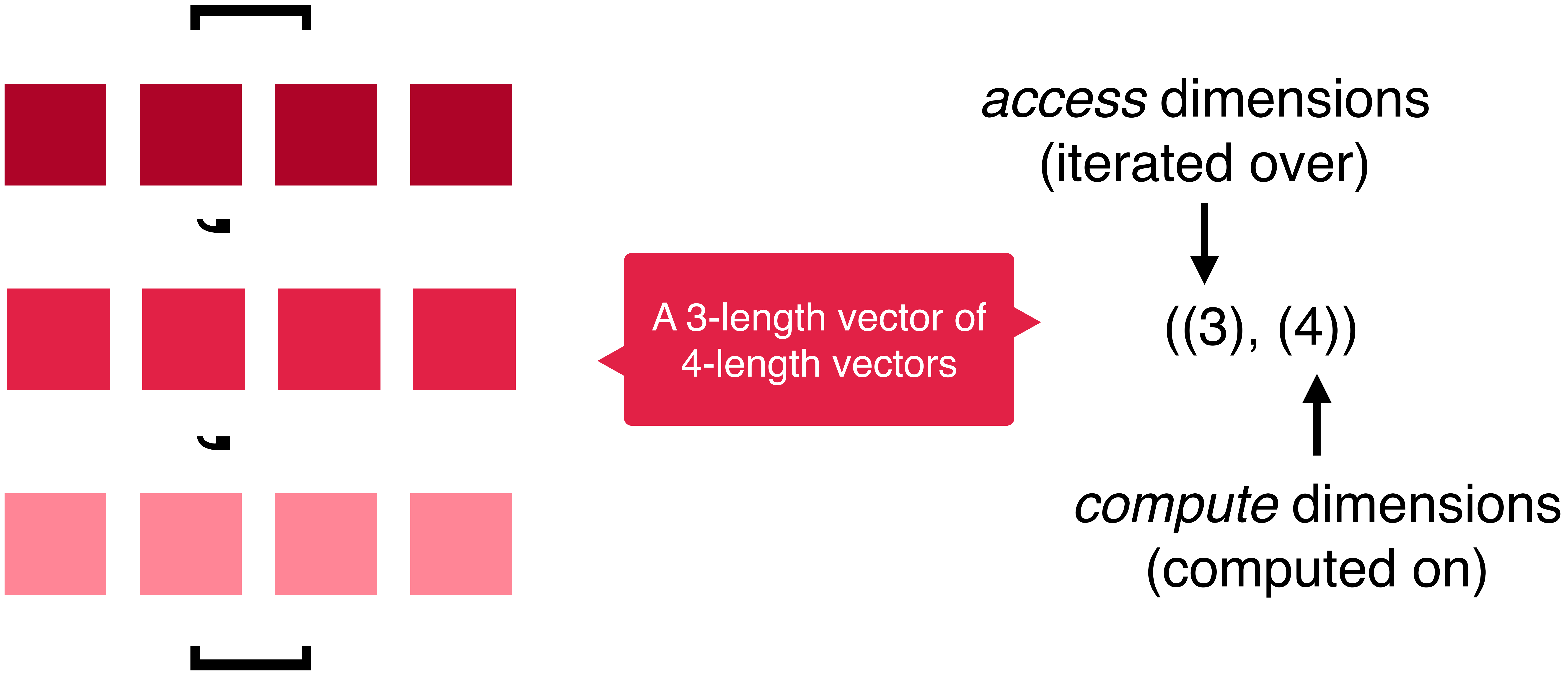


access dimensions
(iterated over)

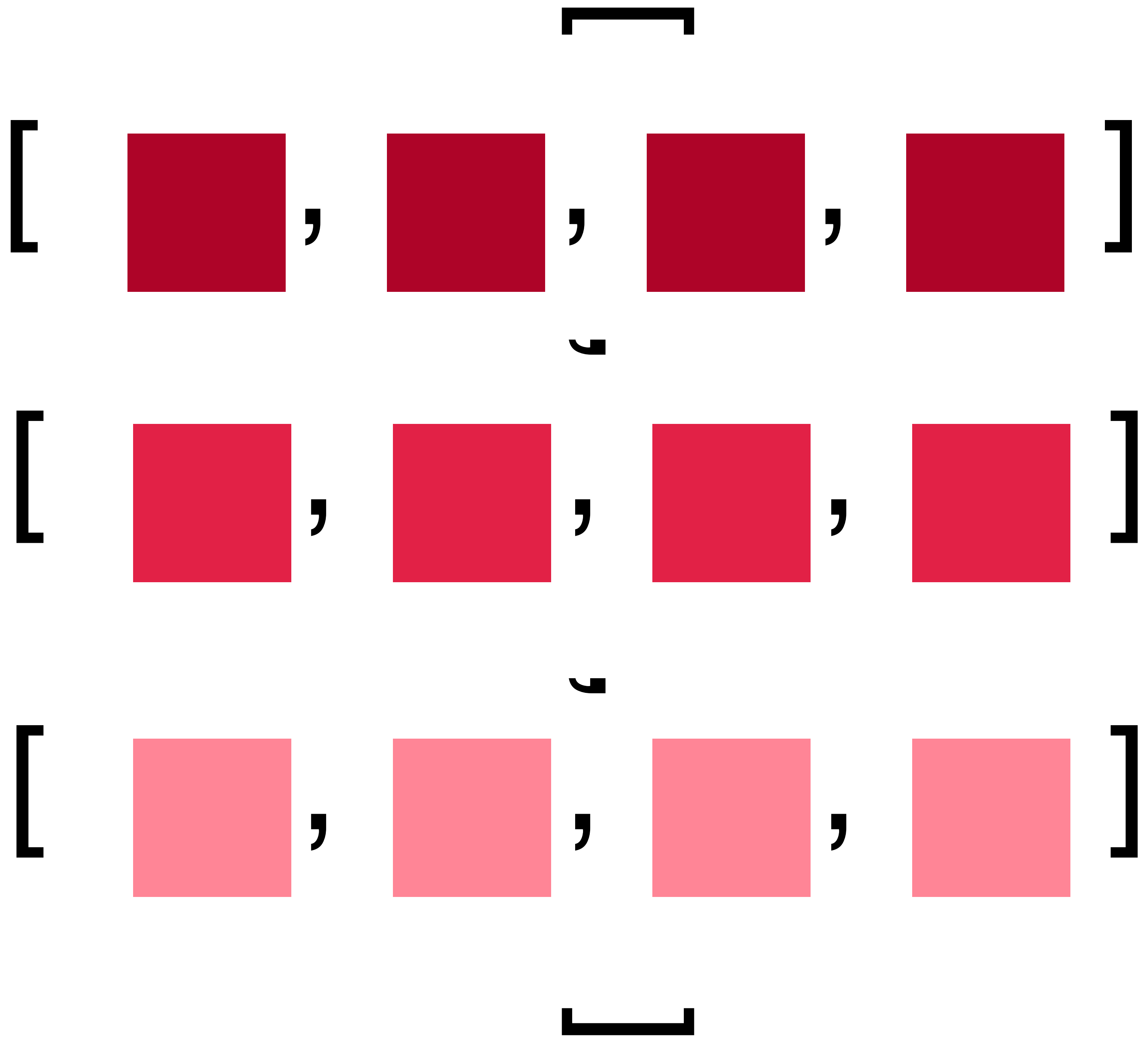
((3), (4))

compute dimensions
(computed on)

An **access pattern** looks like…



An **access pattern** looks like…

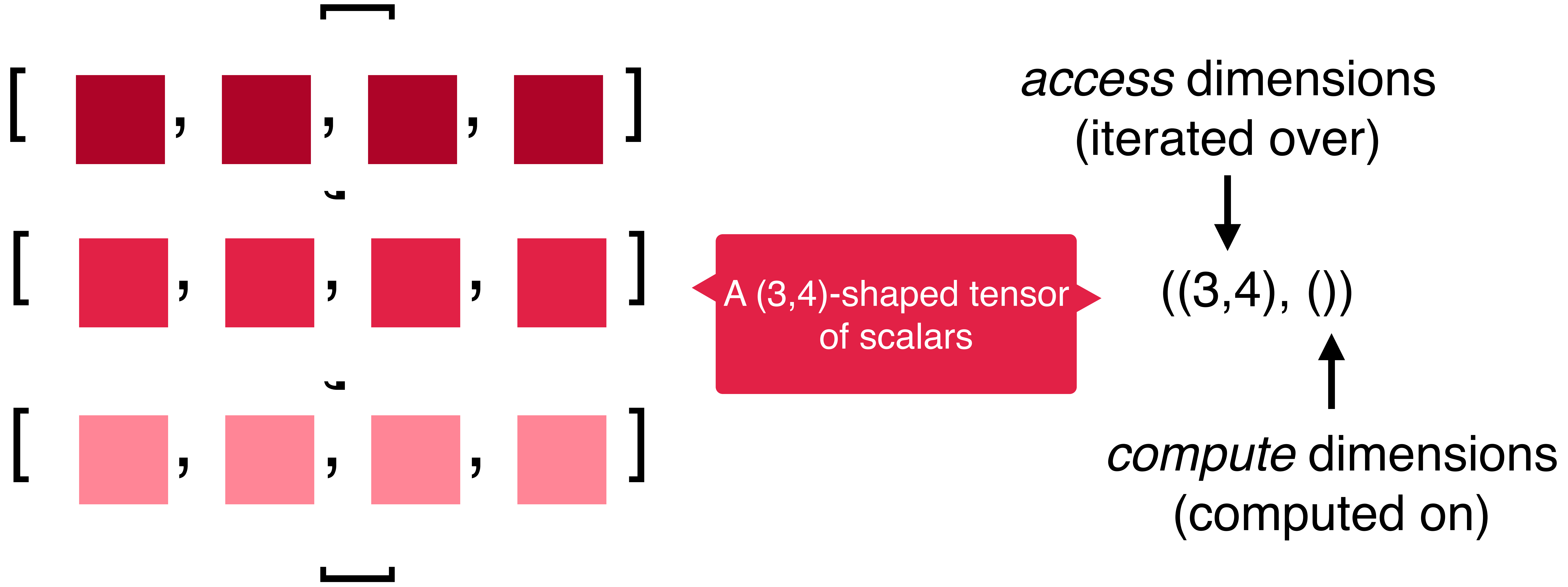


access dimensions
(iterated over)

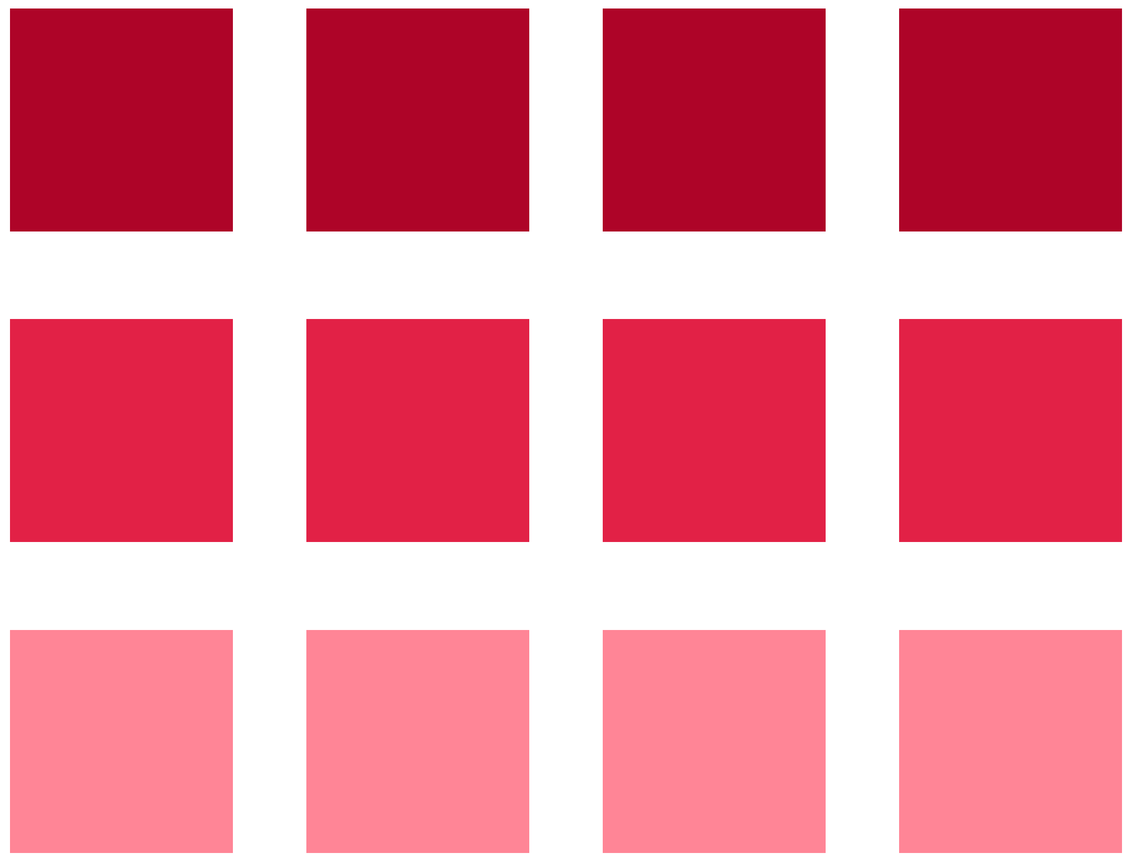
((3,4), ())

compute dimensions
(computed on)

An **access pattern** looks like…



An **access pattern** looks like…

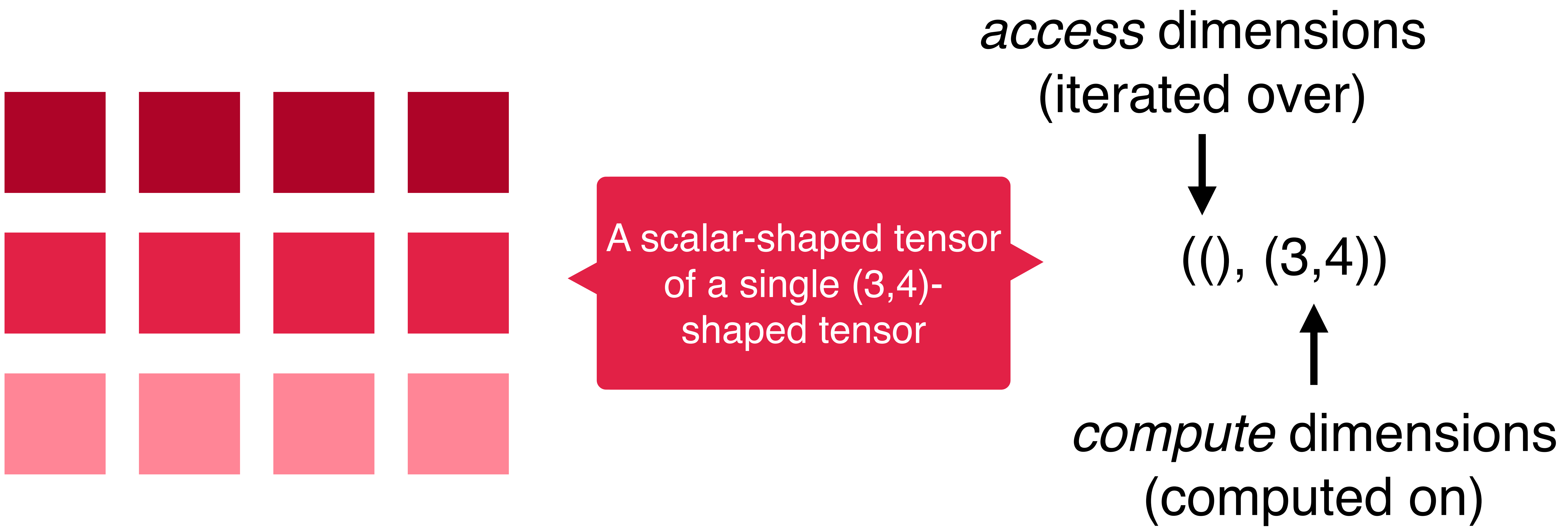


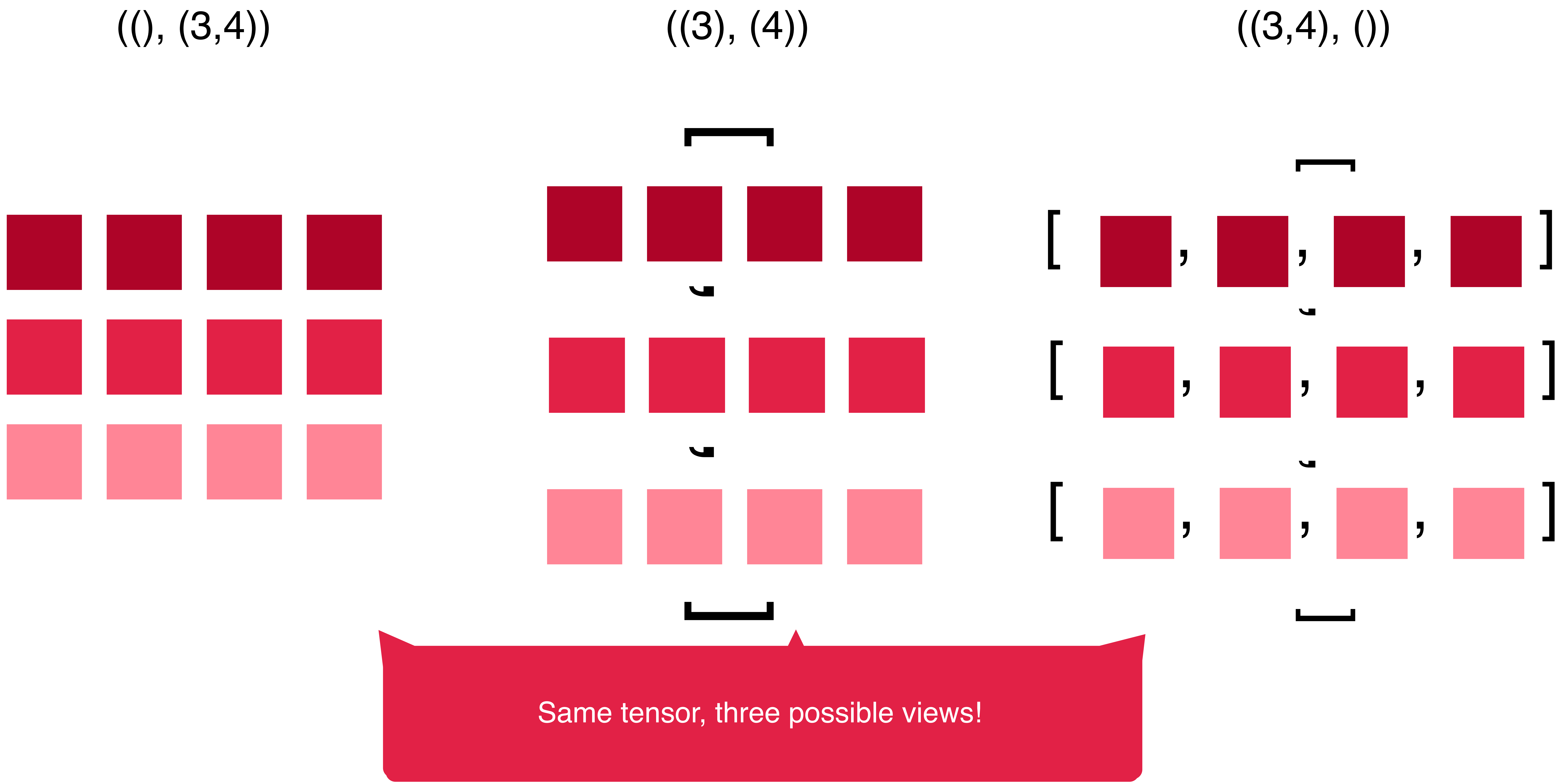
access dimensions
(iterated over)

((), (3,4))

compute dimensions
(computed on)

An **access pattern** looks like…




((), (3,4))
((3), (4))
((3,4), ())
Same tensor, three possible views!

Transformer	Input(s)	Output Shape
`access`	$((a_0, \ldots), (\ldots, a_n))$ and non-negative integer i	$((a_0, \ldots, a_{i-1}), (a_i, \ldots, a_n))$
`cartProd`	$((a_0, \ldots, a_n), (c_0, \ldots, c_p))$ and $((b_0, \ldots, b_m), (c_0, \ldots, c_p))$	$((a_0, \ldots, a_n, b_0, \ldots, b_m), (2, c_0, \ldots, c_p))$
`windows`	$((a_0, \ldots, a_m), (b_0, \ldots, b_n))$, window sh	$((a_0, \ldots, a_m, b'_0, \ldots, b'_n), (w_0, \ldots, w_n))$, $/s_i\rceil$
`slice`	$((a_0, \ldots),$ dimension	$- l$
`squeeze`	$((a_0, \ldots), (\ldots, a_n))$, dimension index d; we assume $a_d = 1$	$((a_0, \ldots), (\ldots, a_n))$ with a_d removed
`flatten`	$((a_0, \ldots, a_m), (b_0, \ldots, b_n))$	$((a_0 \cdots a_m), (b_0 \cdots b_n))$
`reshape`	$((a_0, \ldots, a_m), (b_0, \ldots, b_n))$, access pattern shape literal $((c_0, \ldots, c_p), (d_0, \ldots, d_q))$	$((c_0, \ldots, c_p), (d_0, \ldots, d_q))$, if $a_0 \cdots a_m = c_0 \cdots c_p$ and $b_0 \cdots b_n = d_0 \cdots d_q$

Table 1. Glenside's access pattern transformers.

We can redefine common tensor and list operators with access pattern semantics, which gives us the **Glenside IR**—details in paper!

Outline

- Motivating Example: Matrix Multiplication
- Access Pattern Definition
- Case Studies
  - Reimplementing Matrix Multiplication with Access Patterns
  - Implementing 2D Convolution with Access Patterns
  - Hardware Mapping as Program Rewriting
  - Flexible Hardware Mapping with Equality Saturation

Outline

- Motivating Example: Matrix Multiplication
- Access Pattern Definition
- Case Studies
  - Reimplementing Matrix Multiplication with Access Patterns
  - Implementing 2D Convolution with Access Patterns
  - Hardware Mapping as Program Rewriting
  - Flexible Hardware Mapping with Equality Saturation

Given matrices A and B, pair each row of A with each column of B, compute their dot products, and arrange the results back into a matrix.

```
(access A 1)                 ; ((3), (4))
```

Access A as a list of its rows

```
(access A 1)
```

; ((3), (4))

```
(access A 1)             ;  ((3), (4))

 (access B 1)            ;  ((4), (2))
```

```
(access A 1)               ;  ((3), (4))
(transpose                 ;  ((2), (4))
  (access B 1)             ;  ((4), (2))
  (list 1 0))
```

```
(access A 1)            ; ((3), (4))
(transpose              ; ((2), (4))
  (access B 1)          ; ((4), (2))
  (list 1 0))
```

Access B as a list of its rows, then transpose into a list of its columns

```
(cartProd                    ;  ((3, 2), (2, 4))
 (access A 1)                ;  ((3), (4))
 (transpose                  ;  ((2), (4))
  (access B 1)               ;  ((4), (2))
  (list 1 0)))
```

Create every row–column pair

```
(cartProd                ;  ((3, 2), (2, 4))
  (access A 1)           ;  ((3), (4))
  (transpose             ;  ((2), (4))
    (access B 1)         ;  ((4), (2))
    (list 1 0)))
```

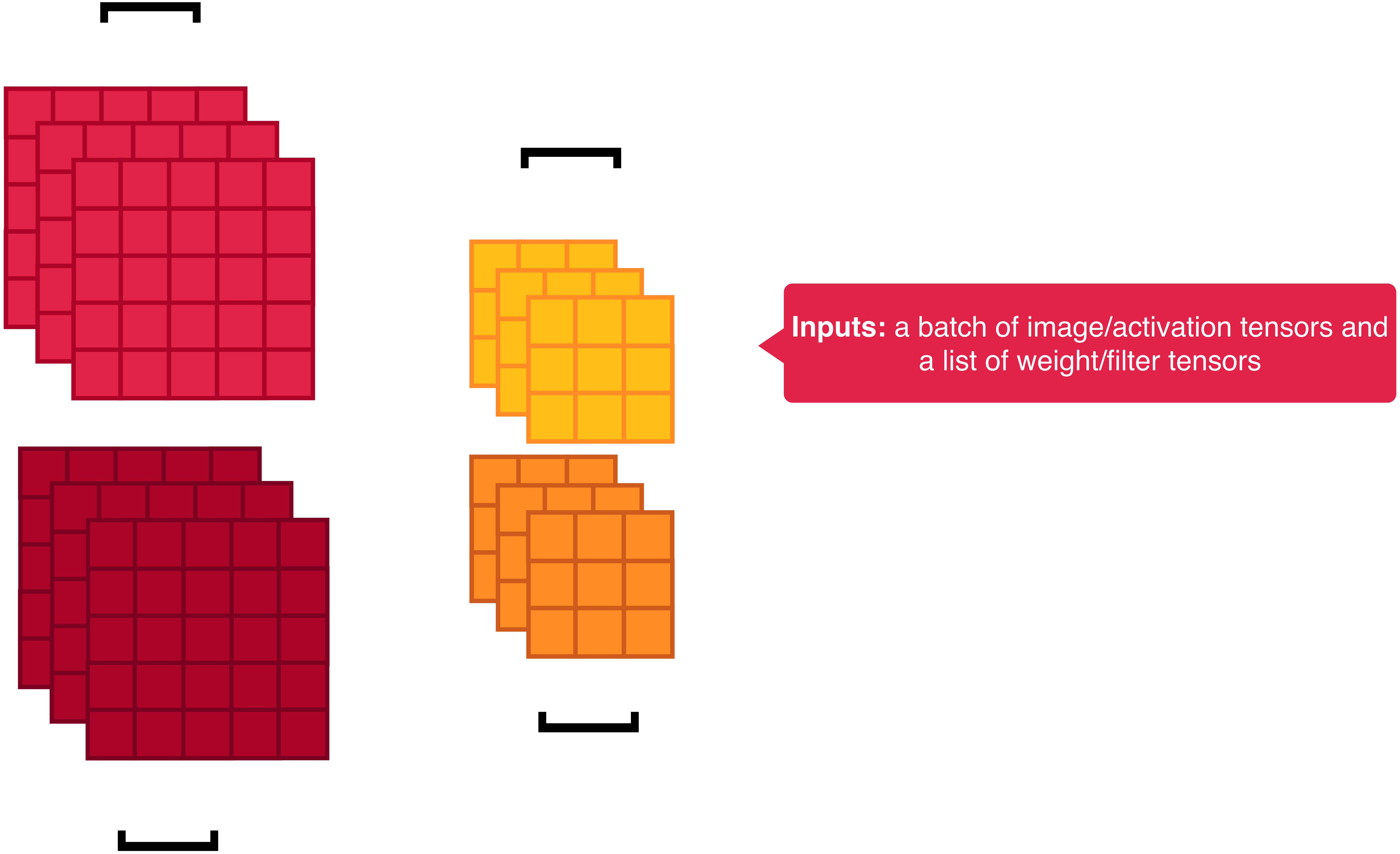
```
(compute dotProd          ;  ((3, 2), ())
 (cartProd                ;  ((3, 2), (2, 4))
  (access A 1)            ;  ((3), (4))
  (transpose              ;  ((2), (4))
   (access B 1)           ;  ((4), (2))
   (list 1 0))))
```

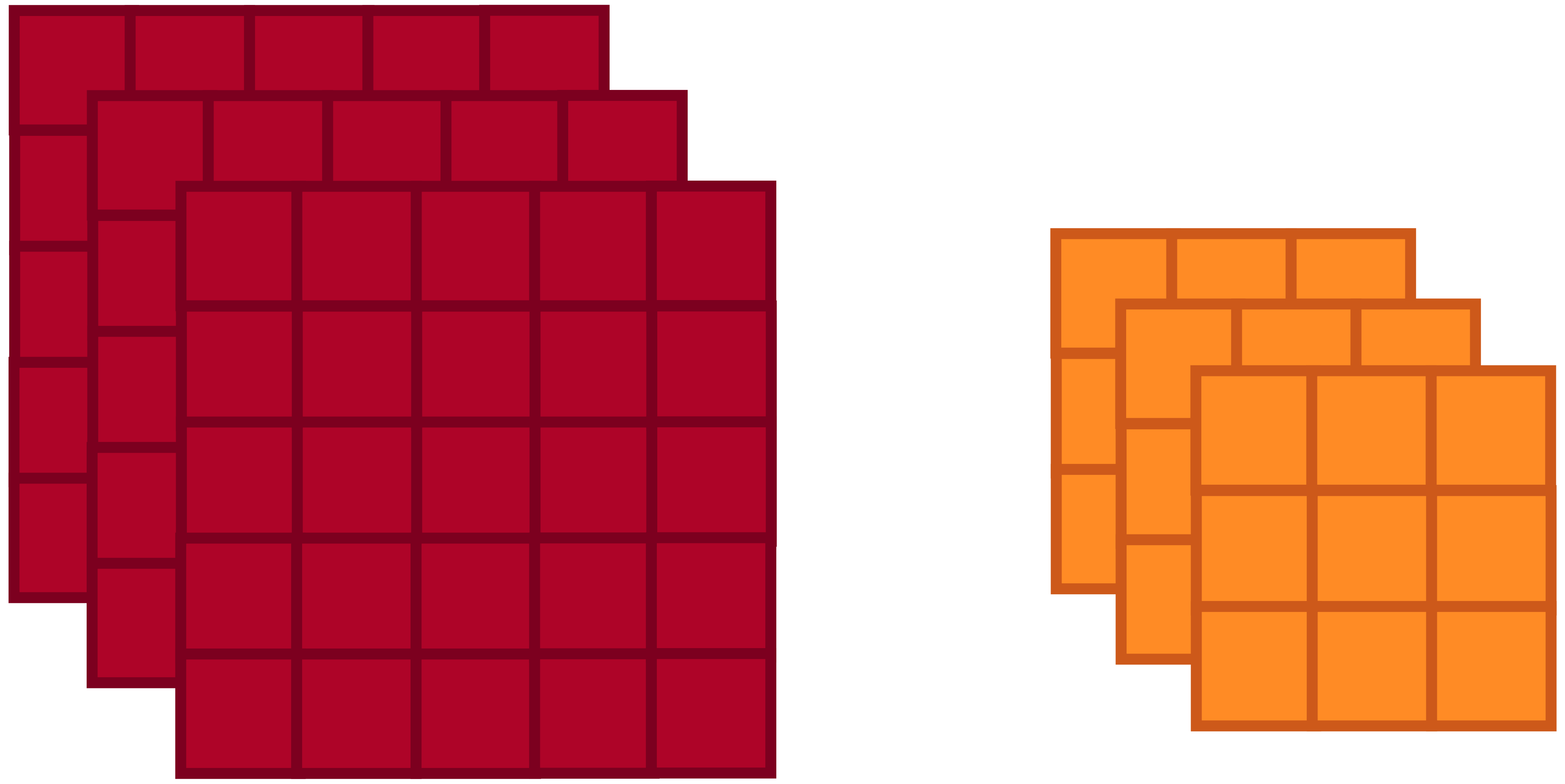
Compute dot product of every row–column pair

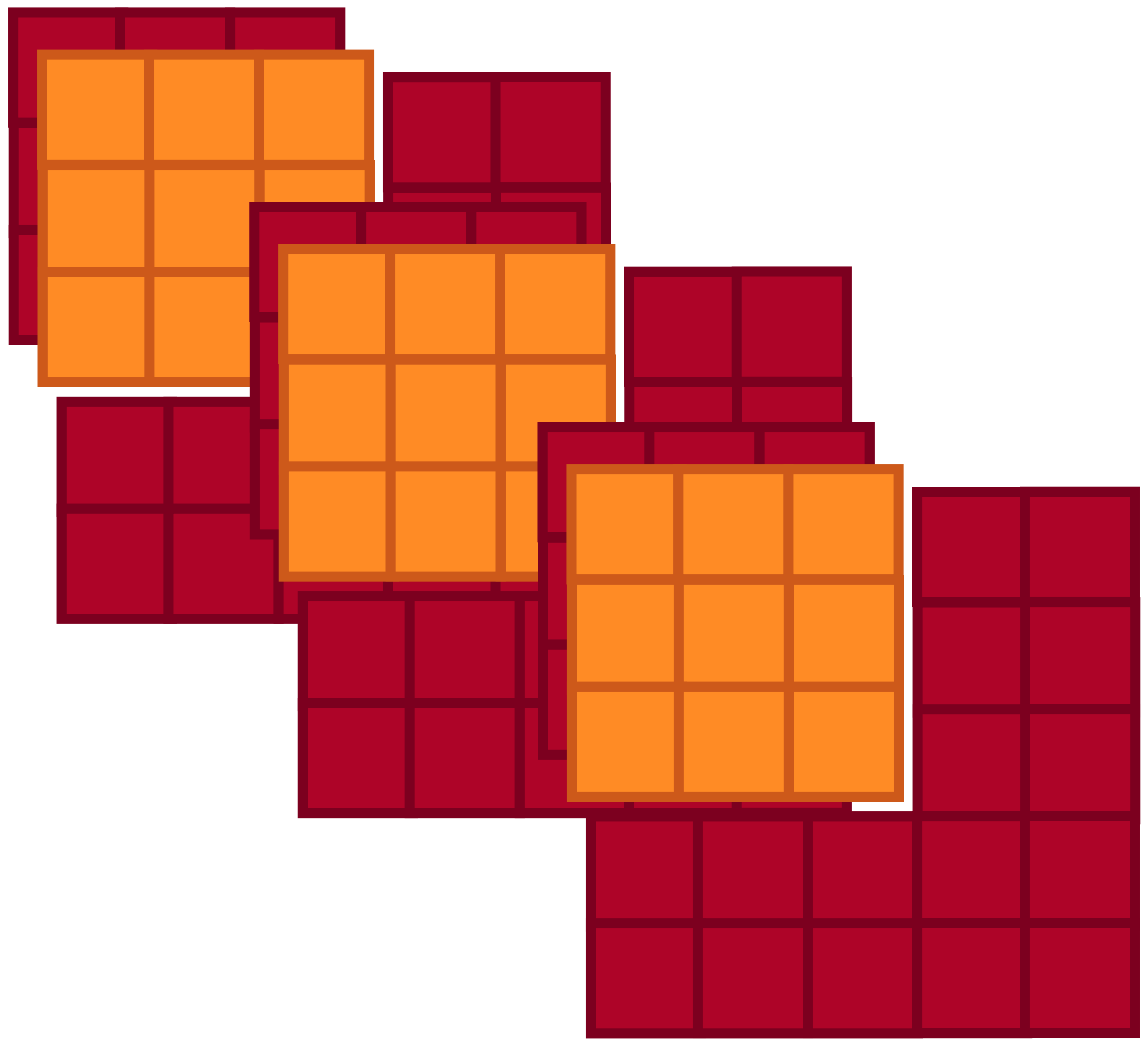
```
(compute dotProd                ;  ((3, 2), ())
 (cartProd                      ;  ((3, 2), (2, 4))
  (access A 1)                  ;  ((3), (4))
  (transpose                    ;  ((2), (4))
   (access B 1)                 ;  ((4), (2))
   (list 1 0))))
```

Outline

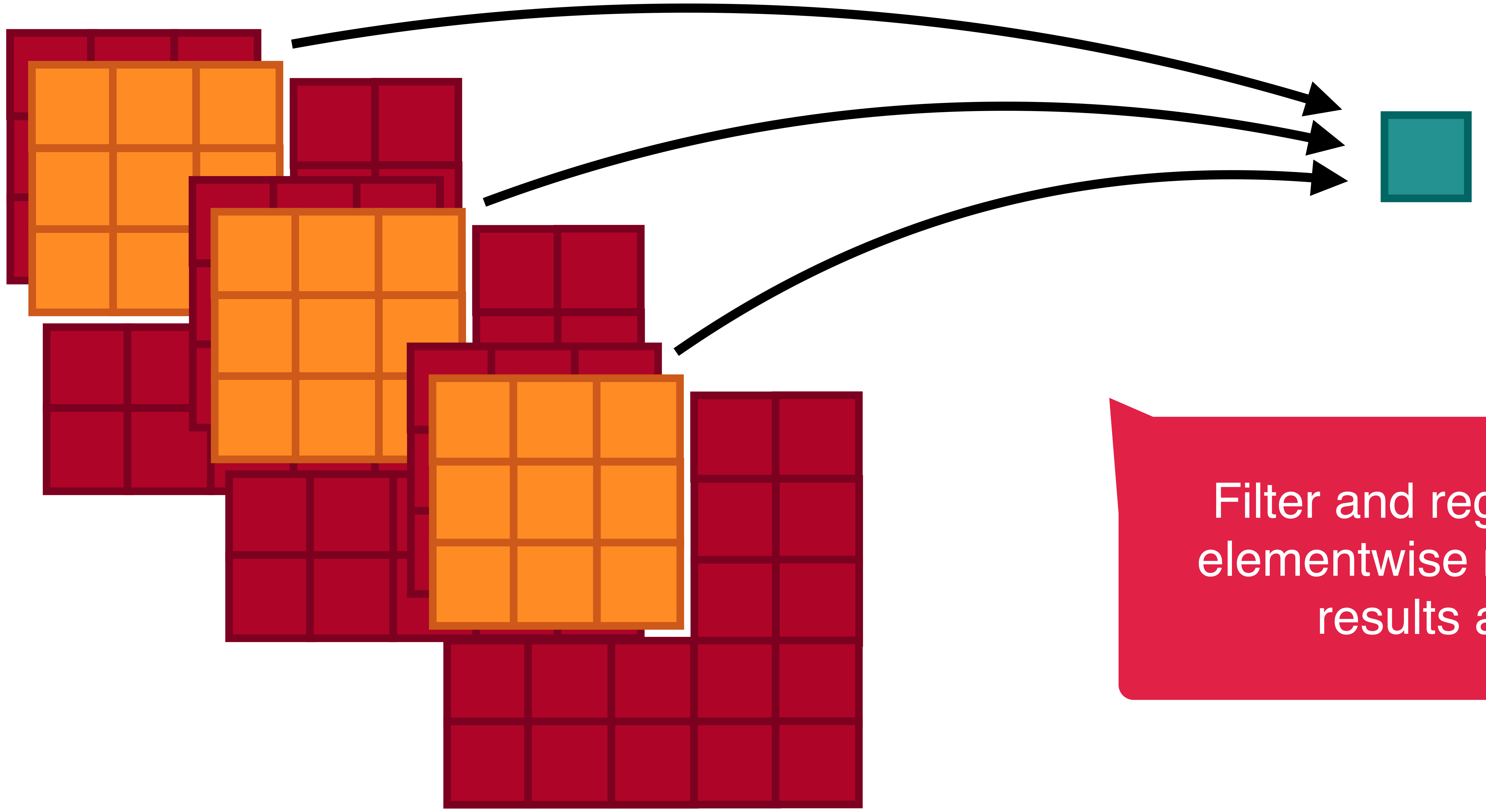
- Motivating Example: Matrix Multiplication
- Access Pattern Definition
- Case Studies
  - Reimplementing Matrix Multiplication with Access Patterns
  - Implementing 2D Convolution with Access Patterns
  - Hardware Mapping as Program Rewriting
  - Flexible Hardware Mapping with Equality Saturation


Inputs: a batch of image/activation tensors and a list of weight/filter tensors

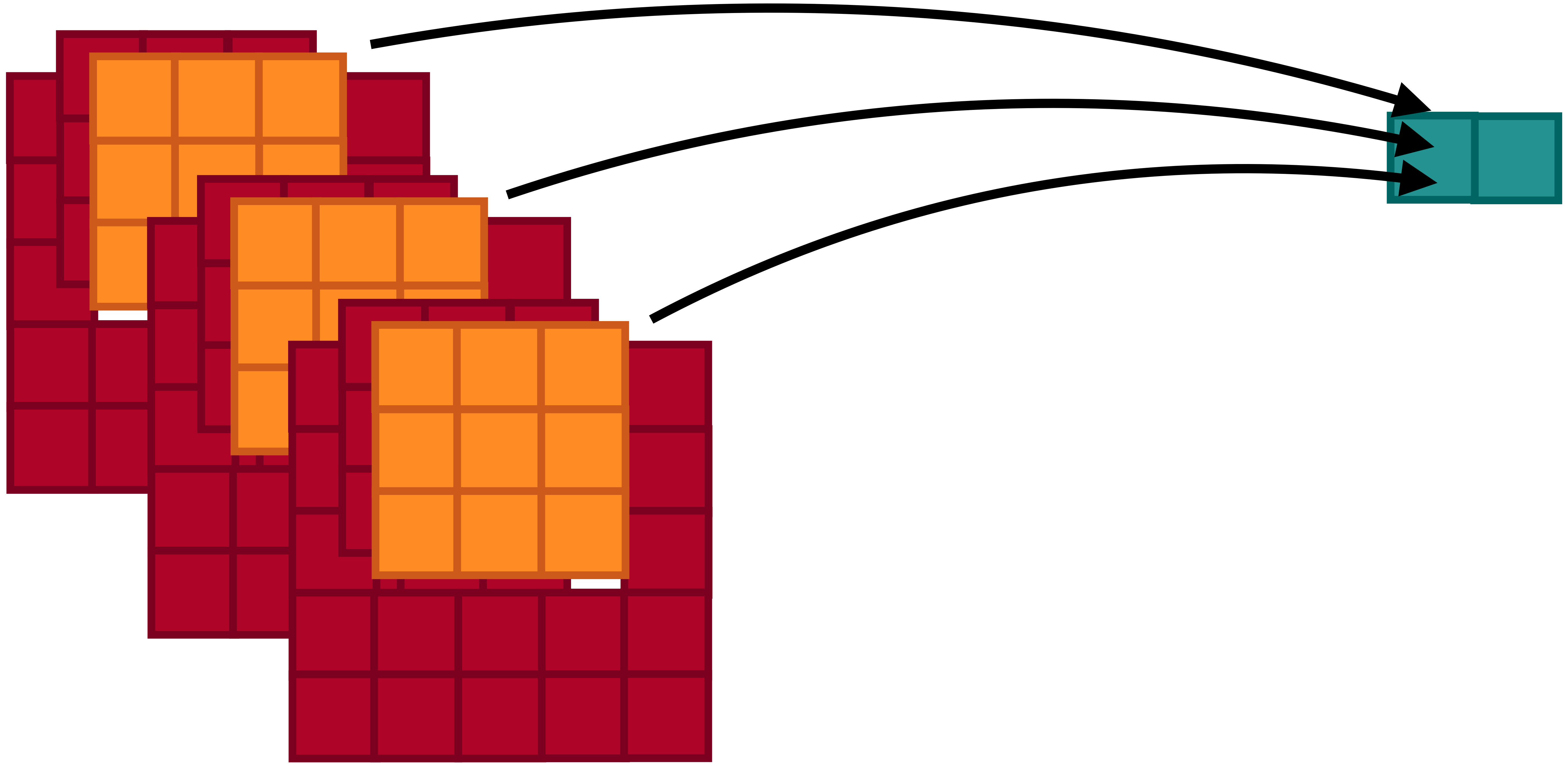


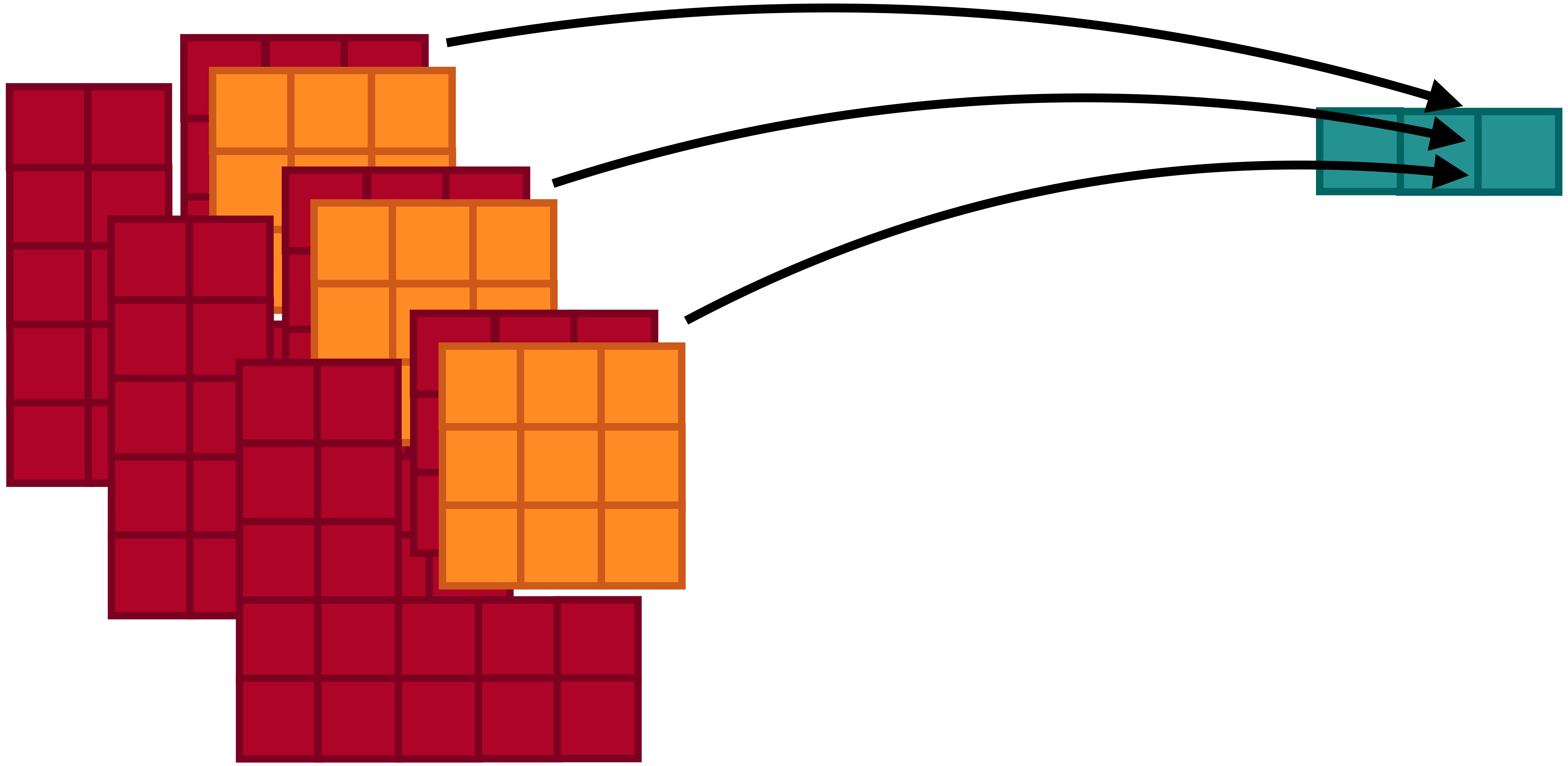


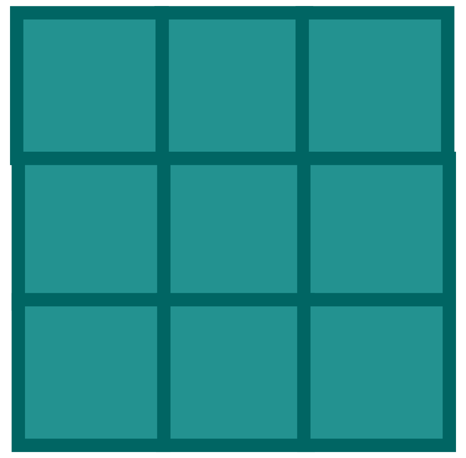
Filter and region of image are elementwise multiplied and the results are summed

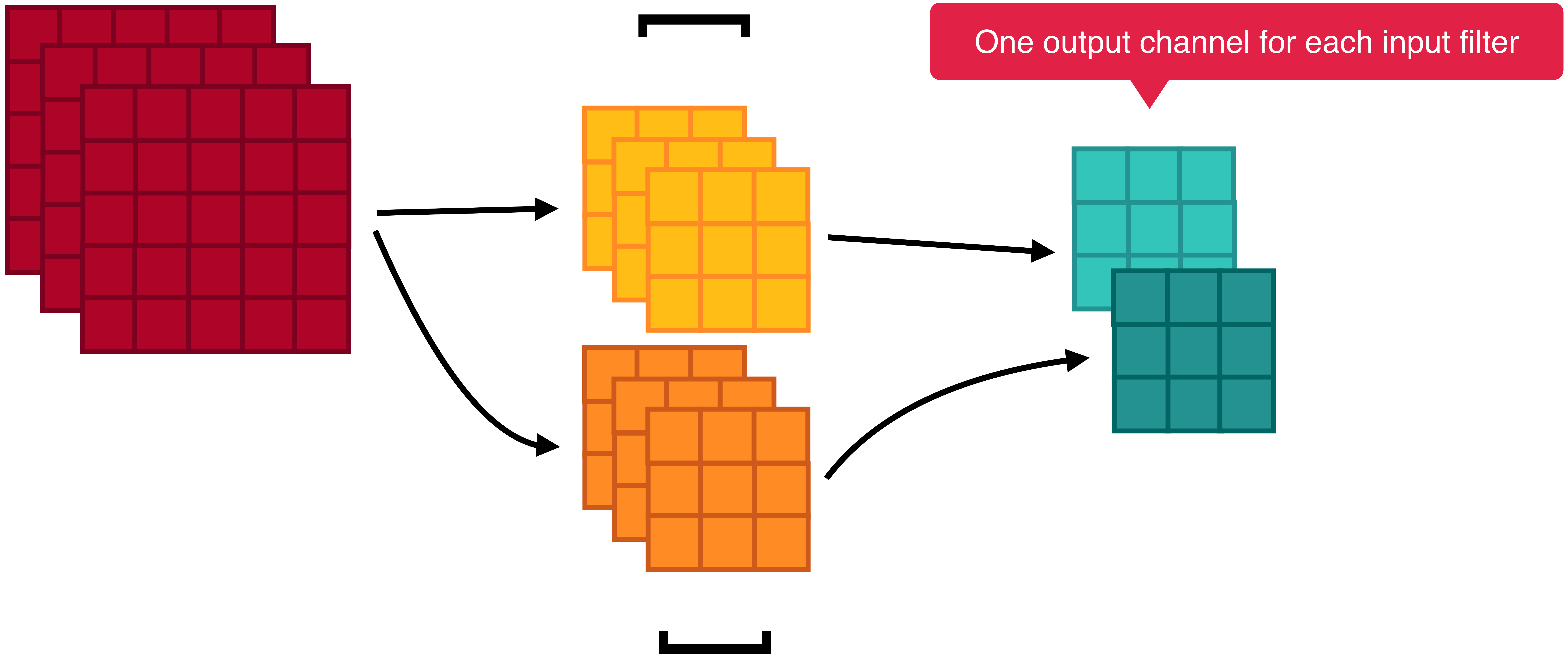


Filter and region of image are elementwise multiplied and the results are summed








One output channel for each input filter

Access weights as a list of 3D filters

```
(access weights 1)                    ; ((O), (C, Kh, Kw))
```

Access activations as a batch of 3D images

```
(access activations 1)        ; ((N), (C, H, W))

(access weights 1)            ; ((O), (C, Kh, Kw))
```

Form windows over input images

```
(windows
 (access activations 1)              ; ((N), (C, H, W))



(access weights 1)                   ; ((O), (C, Kh, Kw))
```

```
(windows
 (access activations 1)          ; ((N), (C, H, W))
 (shape C Kh Kw)
 (shape 1 Sh Sw))
(access weights 1)               ; ((O), (C, Kh, Kw))
```

These parameters control window shape and strides

At each location in each new image, there is a (C, K_h, K_w)-shaped window

```
(windows                                ; ((N, 1, H', W'), (C, Kh, Kw))
 (access activations 1)                 ; ((N), (C, H, W))
 (shape C Kh Kw)
 (shape 1 Sh Sw))
(access weights 1)                      ; ((O), (C, Kh, Kw))
```

Pair windows with filters

```
(cartProd                            ; ((N, 1, H', W', O), (2, C, Kh, Kw))
 (windows                            ; ((N, 1, H', W'), (C, Kh, Kw))
  (access activations 1)             ; ((N), (C, H, W))
  (shape C Kh Kw)
  (shape 1 Sh Sw))
 (access weights 1))                 ; ((O), (C, Kh, Kw))
```

Compute dot product of each window–filter pair

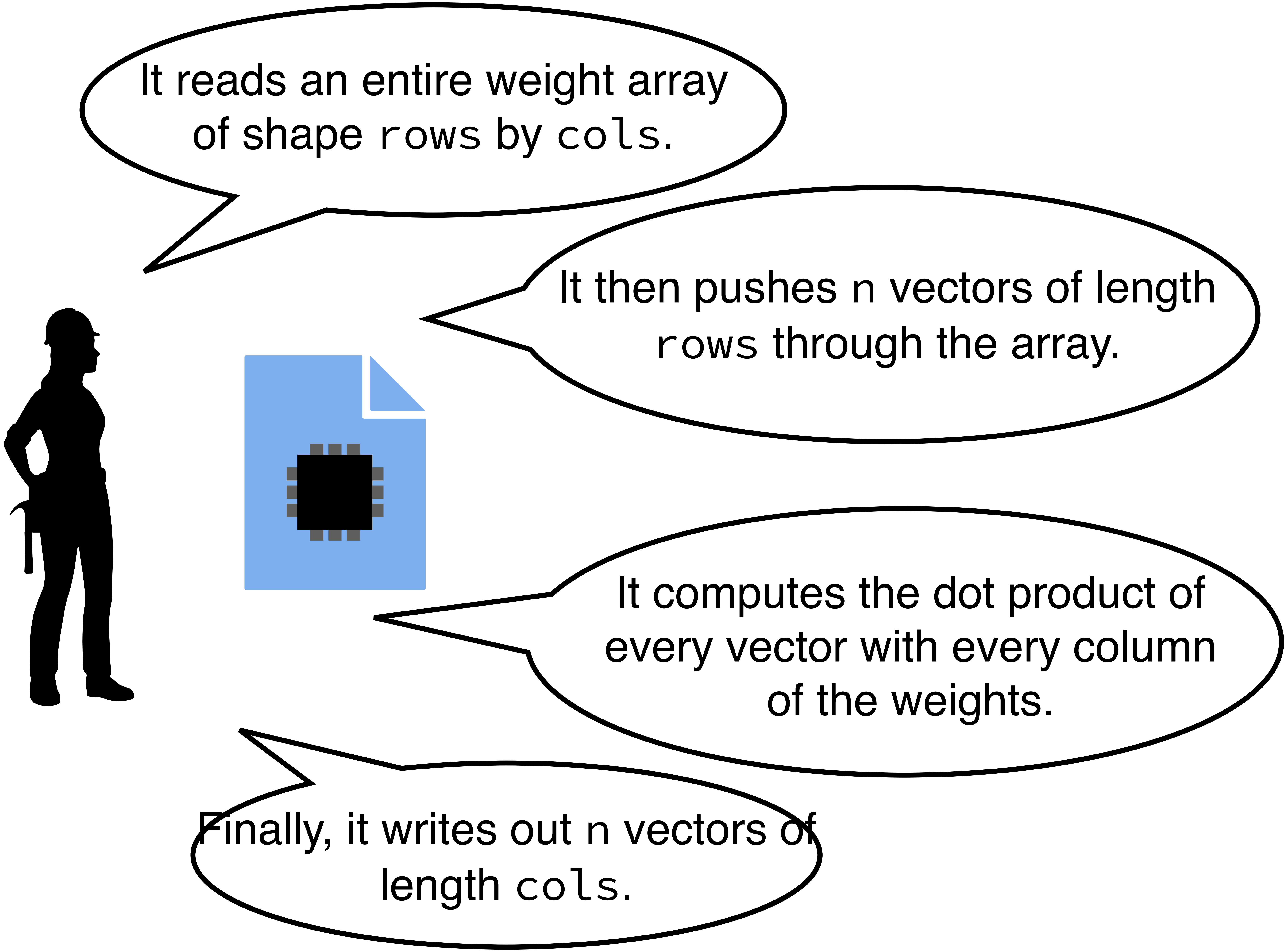
```
(compute dotProd              ; ((N, 1, H', W', O), ())
 (cartProd                    ; ((N, 1, H', W', O), (2, C, Kh, Kw))
  (windows                    ; ((N, 1, H', W'), (C, Kh, Kw))
   (access activations 1)     ; ((N), (C, H, W))
   (shape C Kh Kw)
   (shape 1 Sh Sw))
  (access weights 1)))        ; ((O), (C, Kh, Kw))
```

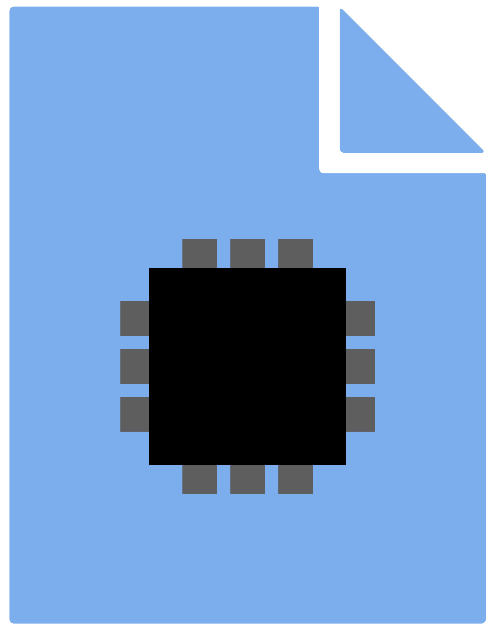
```
(transpose                                   ;  ((N, O, H', W'), ())
 (squeeze
  (compute dotProd                           ;  ((N, 1, H', W', O), ())
   (cartProd                                 ;  ((N, 1, H', W', O), (2, C, Kh, Kw))
    (windows                                 ;  ((N, 1, H', W'), (C, Kh, Kw))
     (access activations 1)                  ;  ((N), (C, H, W))
     (shape C Kh Kw)
     (shape 1 Sh Sw))
    (access weights 1)))                     ;  ((O), (C, Kh, Kw))
  1)
 (list 0 3 1 2))
```

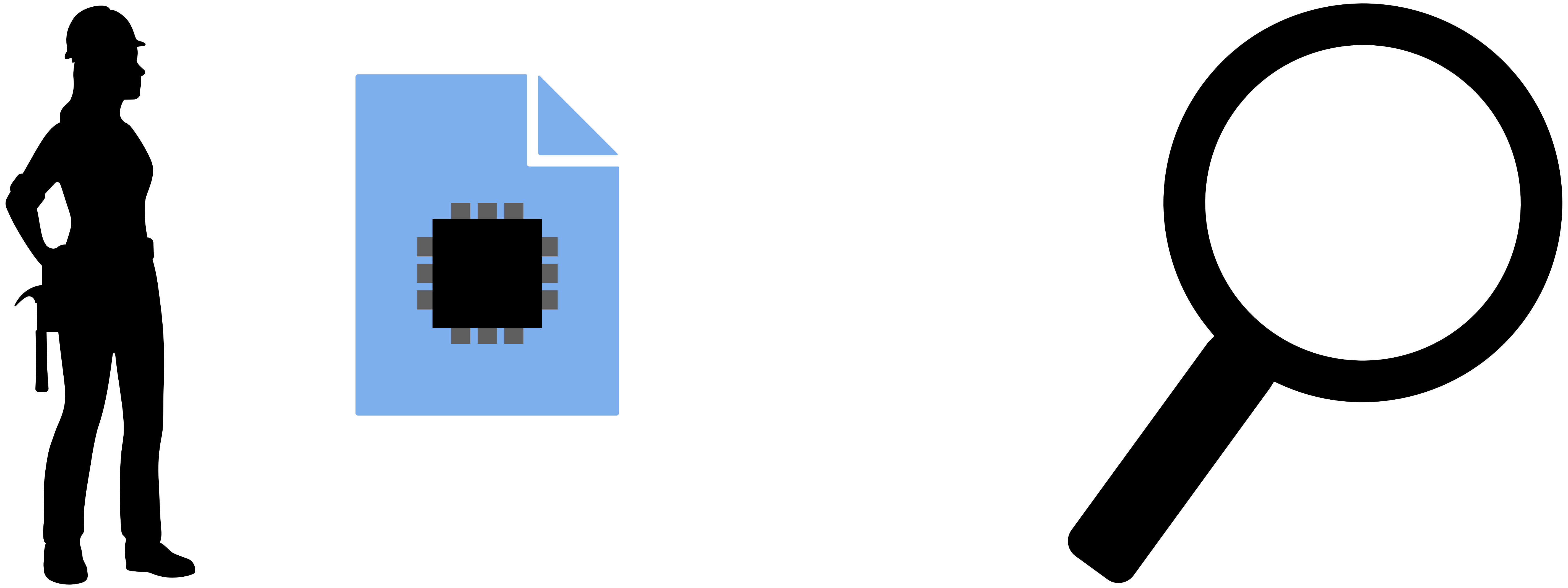
Remove and rearrange dimensions

Outline

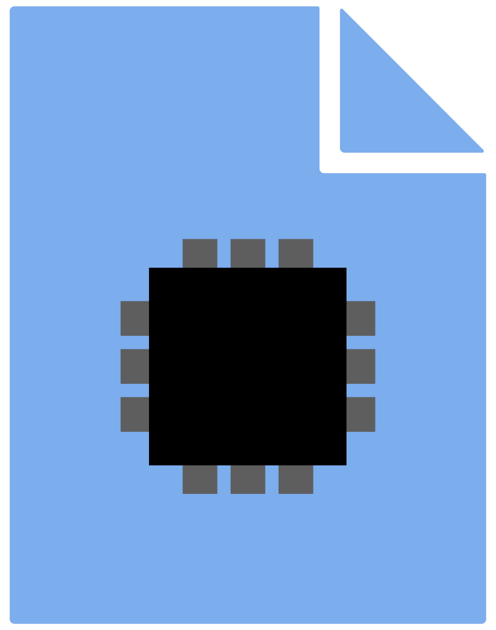
- Motivating Example: Matrix Multiplication
- Access Pattern Definition
- Case Studies
  - Reimplementing Matrix Multiplication with Access Patterns
  - Implementing 2D Convolution with Access Patterns
  - Hardware Mapping as Program Rewriting
  - Flexible Hardware Mapping with Equality Saturation


It reads an entire weight array of shape rows by cols.
It then pushes n vectors of length rows through the array.
It computes the dot product of every vector with every column of the weights.
Finally, it writes out n vectors of length cols.





Can we represent hardware as
a searchable pattern?



```
(compute dotProd
 (cartProd ?a0 ?a1))

  where ?a0 is of shape
   ((?n), (?rows))

  and ?a1 is of shape
   ((?cols), (?rows))
```

With Glenside, we can!

```
(compute dotProd
 (cartProd ?a0 ?a1))

  where ?a0 is of shape
   ((?n), (?rows))

  and ?a1 is of shape
   ((?cols), (?rows))
```

→

We can directly rewrite to hardware invocations!

```
(systolicArray ?rows ?cols ?a0 ?a1)
```

```
(compute dotProd
 (cartProd ?a0 ?a1))

  where ?a0 is of shape
   ((?n), (?rows))

  and ?a1 is of shape
   ((?cols), (?rows))
```

→

```
(systolicArray ?rows ?cols ?a0 ?a1)
```

```
(compute dotProd
 (cartProd
  (access A 1)
  (transpose
   (access B 1)
   (list 1 0))))
```

```
(compute dotProd
 (cartProd ?a0 ?a1))

  where ?a0 is of shape
   ((?n), (?rows))          ⟶   (systolicArray ?rows ?cols ?a0 ?a1)

  and ?a1 is of shape
   ((?cols), (?rows))
```

```
(compute dotProd
 (cartProd
  (access A 1)
  (transpose
   (access B 1)
   (list 1 0))))
```

```
(compute dotProd
 (cartProd ?a0 ?a1))

  where ?a0 is of shape
   ((?n), (?rows))

  and ?a1 is of shape
   ((?cols), (?rows))
```

→ **(systolicArray ?rows ?cols** ?a0 ?a1)

(systolicArray

4 2

(access A 1)

(transpose

(access B 1)

(list 1 0))))

Outline

- Motivating Example: Matrix Multiplication
- Access Pattern Definition
- Case Studies
  - Reimplementing Matrix Multiplication with Access Patterns
  - Implementing 2D Convolution with Access Patterns
  - Hardware Mapping as Program Rewriting
  - Flexible Hardware Mapping with Equality Saturation

```
(transpose
 (squeeze
  (compute dotProd
   (cartProd
    (windows
     (access activations 1)
     (shape C Kh Kw)
     (shape 1 Sh Sw))
    (access weights 1)))
  1)
 (list 0 3 1 2))
```

```
(compute dotProd
 (cartProd
  (access A 1)
  (transpose
   (access B 1)
   (list 1 0))))
```

```
(transpose
 (squeeze
  (compute dotProd
   (cartProd
    (windows
     (access activations 1)
     (shape C Kh Kw)
     (shape 1 Sh Sw))
    (access weights 1)))
  1)
 (list 0 3 1 2))
```

Convolution and matrix multiplication have similar structure!

```
(compute dotProd
 (cartProd
  (access A 1)
  (transpose
   (access B 1)
   (list 1 0))))
```

```
(transpose
 (squeeze
  (compute dotProd
   (cartProd
    (windows
     (access activations 1)
     (shape C Kh Kw)
     (shape 1 Sh Sw))
    (access weights 1)))
  1)
 (list 0 3 1 2))
```

Can we apply our hardware rewrite?

```
(compute dotProd
 (cartProd ?a0 ?a1))

  where ?a0 is of shape
   ((?n), (?rows))

  and ?a1 is of shape
   ((?cols), (?rows))
```

```
(transpose
 (squeeze
  (compute dotProd
   (cartProd
    (windows     ; ((N, 1, H', W'), (C, Kh, Kw))
     (access activations 1)
     (shape C Kh Kw)
     (shape 1 Sh Sw))
    (access weights 1)))  ; ((O), (C, Kh, Kw))
  1)
 (list 0 3 1 2))
```

```
(compute dotProd
 (cartProd ?a0 ?a1))
```

where ?a0 is of shape ((?n), (?rows))

and ?a1 is of shape ((?cols), (?rows))

Our access pattern shapes do not pass the rewrite's conditions

```
(transpose
 (squeeze
  (compute dotProd
   (cartProd
    (windows     ; ((?n), (?rows))
     (access activations 1)
     (shape C Kh Kw)
     (shape 1 Sh Sw))
    (access weights 1)))  ; ((?cols), (?rows))
  1)
 (list 0 3 1 2))
```

Can we flatten our access patterns?

```
(compute dotProd
 (cartProd ?a0 ?a1))

  where ?a0 is of shape
   ((?n), (?rows))

  and ?a1 is of shape
   ((?cols), (?rows))
```

```
?a → (reshape (flatten ?a) ?shape)
```

Flattens and immediately reshapes an access pattern

```
?a → (reshape (flatten ?a) ?shape)
```

Flattens and immediately reshapes an access pattern

```
(transpose
 (squeeze
  (compute dotProd
   (cartProd
    (windows      ; ((N, 1, H', W'), (C, Kh, Kw))
     (access activations 1)
     (shape C Kh Kw)
     (shape 1 Sh Sw))
    (access weights 1)))  ; ((O), (C, Kh, Kw))
  1)
 (list 0 3 1 2))
```

```
(transpose
 (squeeze
  (compute dotProd
   (cartProd
    (reshape (flatten (windows  ; ((N, 1, H', W'), (C, Kh, Kw))
     (access activations 1)
     (shape C Kh Kw)
     (shape 1 Sh Sw))) ?shape0)
    (reshape (flatten (access weights 1)) ?shape1))) ; ((O), (C, Kh, Kw))
  1)
 (list 0 3 1 2))
```

```
(transpose
 (squeeze
  (compute dotProd
   (cartProd
    (reshape (flatten (windows  ; ((N, 1, H', W'), (C, Kh, Kw))
     (access activations 1)
     (shape C Kh Kw)
     (shape 1 Sh Sw))) ?shape0)
    (reshape (flatten (access weights 1)) ?shape1))) ; ((O), (C, Kh, Kw))
  1)
 (list 0 3 1 2))
```

But our access pattern shapes haven't changed!

```
(transpose
 (squeeze
  (compute dotProd
   (cartProd
    (reshape (flatten (windows  ; ((N, 1, H’, W’), (C, Kh, Kw))
      (access activations 1)
      (shape C Kh Kw)
      (shape 1 Sh Sw))) ?shape0)
    (reshape (flatten (access weights 1)) ?shape1))) ; ((O), (C, Kh, Kw))
  1)
 (list 0 3 1 2))
```

We need to “bubble” the reshapes to the top

These rewrites “bubble” `reshape` through `cartProd` and `compute dotProd`

```
(cartProd
 (reshape ?a0 ?shape0)
 (reshape ?a1 ?shape1)) → (reshape (cartProd ?a0 ?a1) ?newShape)


(compute dotProd
 (reshape ?a ?shape))   → (reshape (compute dotProd ?a) ?newShape)
```

```
(transpose
 (squeeze
  (compute dotProd
   (cartProd
    (reshape (flatten (windows  ; ((N, 1, H’, W’), (C, Kh, Kw))
     (access activations 1)
     (shape C Kh Kw)
     (shape 1 Sh Sw))) ?shape0)
    (reshape (flatten (access weights 1)) ?shape1))) ; ((O), (C, Kh, Kw))
  1)
 (list 0 3 1 2))
```

```
(transpose
 (squeeze
  (reshape (compute dotProd
   (cartProd
    (flatten (windows ; ((N · 1 · H’ · W’), (C · Kh · Kw))
     (access activations 1)
     (shape C Kh Kw)
     (shape 1 Sh Sw)))
    (flatten (access weights 1)))) ?shape) ; ((O), (C · Kh · Kw))
  1)
 (list 0 3 1 2))
```

reshapes have been moved out, and the access patterns are flattened!

```
(transpose
 (squeeze
  (reshape (compute dotProd
   (cartProd
    (flatten (windows ; ((N · 1 · H' · W'), (C · Kh · Kw))
     (access activations 1)
     (shape C Kh Kw)
     (shape 1 Sh Sw)))
    (flatten (access weights 1)))) ?shape) ; ((O), (C · Kh · Kw))
  1)
 (list 0 3 1 2))
```

```
(compute dotProd
 (cartProd ?a0 ?a1))

where ?a0 is of shape
 ((?n), (?rows))

and ?a1 is of shape
 ((?cols), (?rows))
```

Our rewrite can now map convolution to matrix multiplication hardware!

```
?a → (reshape (flatten ?a) ?shape)

(cartProd
 (reshape ?a0 ?shape0)
 (reshape ?a1 ?shape1)) → (reshape (cartProd ?a0 ?a1) ?newShape)

(compute dotProd
 (reshape ?a ?shape))    → (reshape (compute dotProd ?a) ?newShape)
```

These rewrites *rediscover* the im2col transformation!

In conclusion,

In conclusion,

we have presented **access patterns** as a new tensor representation,

In conclusion,

we have presented **access patterns** as a new tensor representation,

we have used them to build the **pure, low-level, binder free IR Glenside,**

In conclusion,

we have presented **access patterns** as a new tensor representation,

we have used them to build the **pure, low-level, binder free IR Glenside,**

and have shown how they **enable hardware-level tensor program rewriting**.

https://github.com/gussmith23/glenside

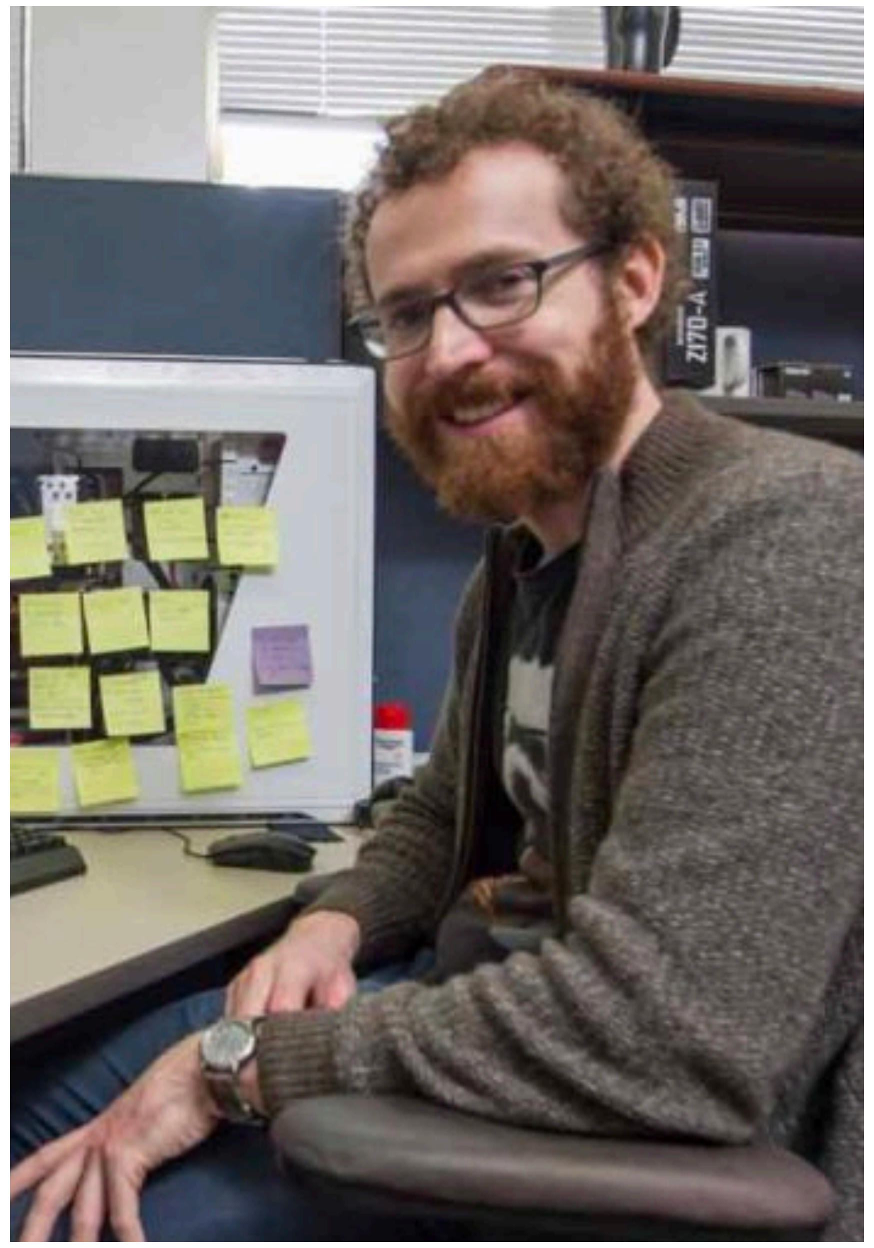
Glenside is an actively maintained Rust library! Try it out and open issues if you have questions!



sampl

PLSE

DARPA




CRISP
Center for Research in Intelligent
Storage and Processing in Memory

ada
Applications Driving Architectures

Thank you!