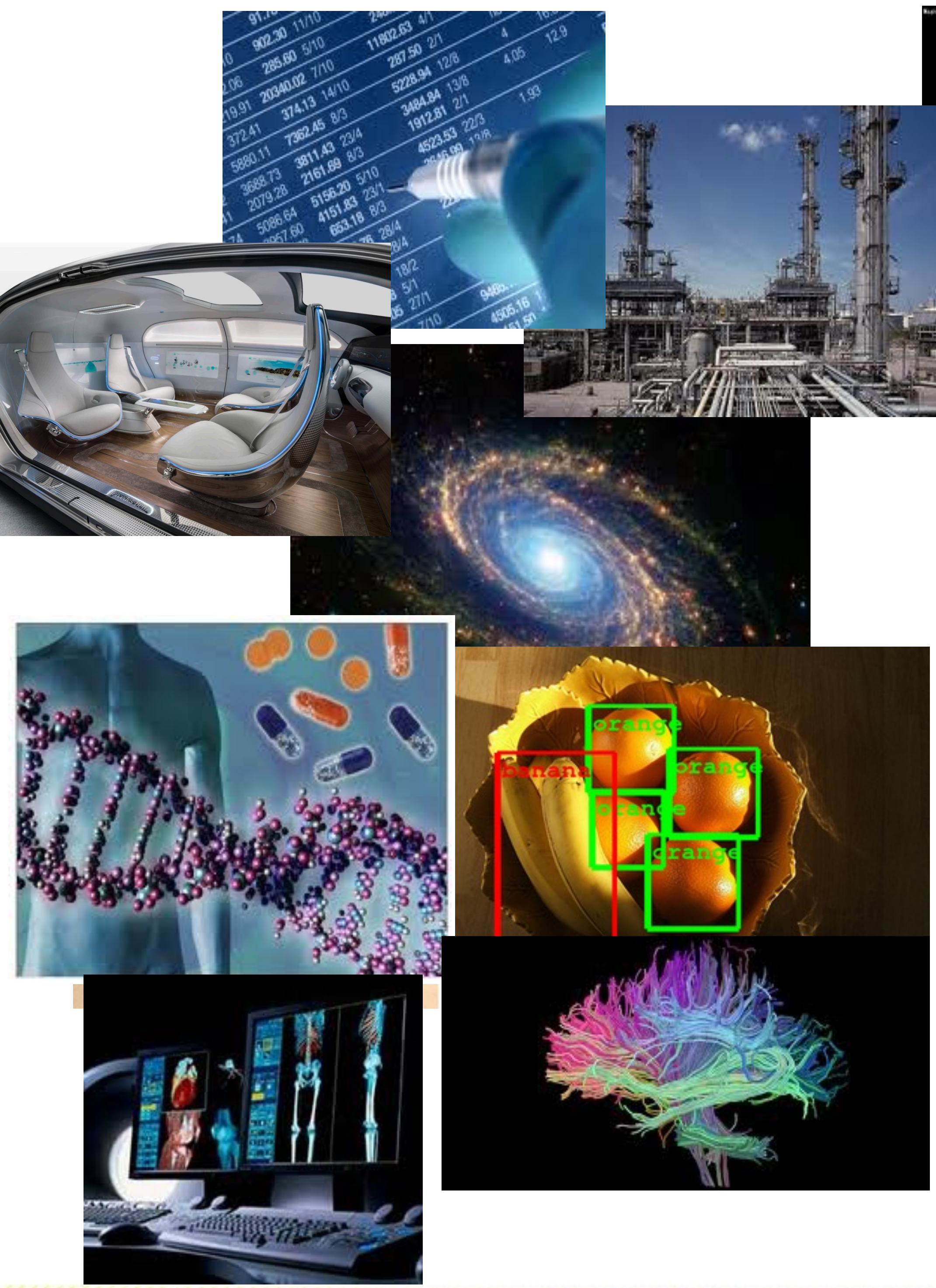
Tensor Comprehensions in SaC

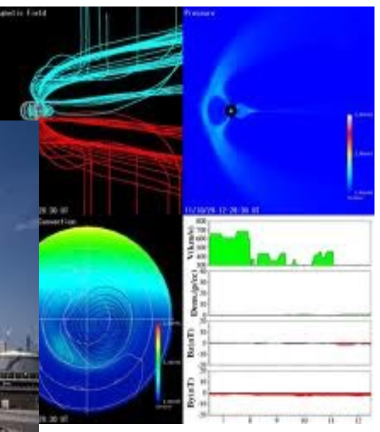
Seminar on Tensor Computation
4.Feb.2022, Oxford

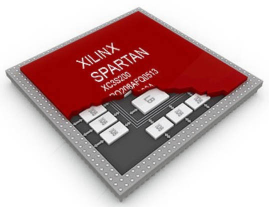
Sven-Bodo Scholz

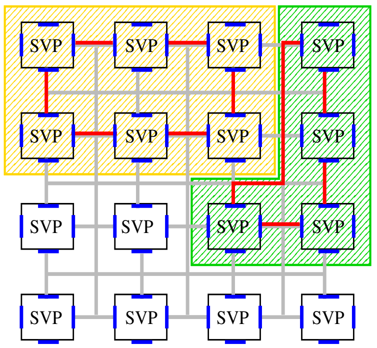
Heriot-Watt University, Edinburgh
Radboud University Nijmegen

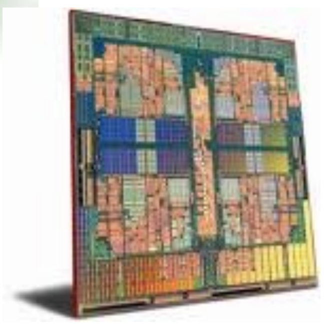
Array Programming as a tool for enabling
HPC^3 for everyone











Tensors, Tensor Notation, Einstein Notation, Ricci Calculus,… and their applications

- N-dimensional index spaces for data and operations on them
- Notation omits ranges and boundaries whenever "obvious"
- Typically nested!

- General Activation Formula: $a_j^{[l]} = g^{[l]}(\sum_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}) = g^{[l]}(z_j^{[l]})$
- $J(x, W, b, y)$ or $J(\hat{y}, y)$ denote the cost function.

Examples of cost function:

- $J_{CE}(\hat{y}, y) = -\sum_{i=0}^{m} y^{(i)} \log \hat{y}^{(i)}$
- $J_1(\hat{y}, y) = \sum_{i=0}^{m} \mid y^{(i)} - \hat{y}^{(i)} \mid$

Isn’t that just array comprehensions?
--- or …. getting to the point?

$$[\ \text{expr} \mid \text{id}_1 \leftarrow \text{generator}_1, \dots, \text{id}_n \leftarrow \text{generator}_n]$$

Here we need lower and upper bounds

We want non-inherently-sequential generators!

We need to “compose” the generators!

Transposition

Mathematics:

$a^T_{i,j} = a_{j,i}$

"typical" array compre[illegible]ion:

```
aT = [ a[j,i] |i i[illegible] .. shape(a)[1], j in 0 .. shape(a)[0] ]
```

SaC 1.0:

```
aT = { [i,j] ->  a[j,i] };
```

Composition is always orthogonal

Set-notation is mapped into data-parallelism

Lower and upper bounds are inferred if possible

Element-wise addition

scalar addition!

Mathematics:

$c_{\vec{\imath}} = a_{\vec{\imath}} + b_{\vec{\imath}}$

recursive call !

"typical" array comprehension:

```
c = [ a[i] + b[i] |i in 0 .. shape(a)[0] ]
```

scalar addition!

SaC 1.0:

```
c = { iv ->  a[iv] + b[iv] };
```

vector of indices!

```
c = { [i] -> a[i] + b[i]};
```

recursive call !

Our physics example

$$a_j^{[l]} = g^{[l]}(\sum_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]})$$

in SaC 1.0:

```
a = { [j] ->  g * (sum({[k] -> w[j,k]*a[k]}) + b[j]) };
```

Concatenation

Bound inference fails !

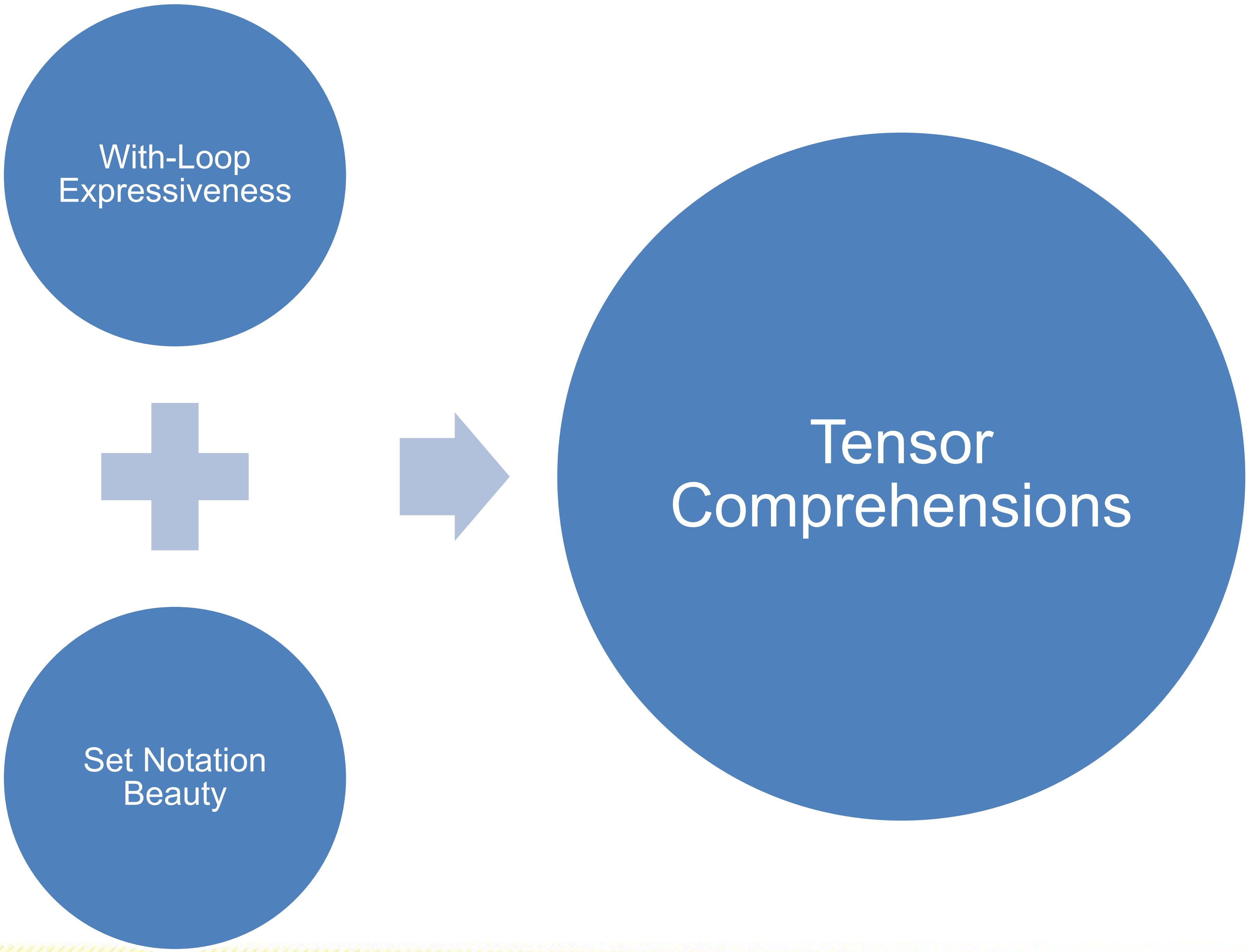
```
c = { [i] -> ( i < len(a) ? a[i] : b[i-len(a)]) };
```

Conditional is ugly!

SaC 1.4, Tensor Comprehensions!

```
c = { [i] -> a[i]                   ;
      [i] -> b[i-len(a)] | [i] < shape(a) + shape(b) };
```

Aim


With-Loop Expressiveness
Set Notation Beauty
Tensor Comprehensions

Tensor Comprehensions:
Full With-Loop expressiveness in Set Notation!

with {
  (lb <= $[i_1,\ldots,i_n]$ < ub) : expr;
  …
  (lb <= $[i_1,\ldots,i_n]$ < ub) : expr;
} : genarray(*shape*, default-expr)

{ $[i_1,\ldots,i_n]$ -> expr | lb <= $[i_1,\ldots,i_n]$ < ub;
  …
  $[i_1,\ldots,i_n]$ -> expr | lb <= $[i_1,\ldots,i_n]$ < ub;
  $[i_1,\ldots,i_n]$ -> default-expr | $[i_1,\ldots,i_n]$ < *shape* }

Beauty Measure #1: Make some parts optional & use the Set Notation inference

$\{ [i_1,\ldots,i_n] \;\texttt{->}\; expr \;|\; lb \texttt{<=} [i_1,\ldots,i_n] < ub;$

…

$[i_1,\ldots,i_n] \;\texttt{->}\; expr \;|\; lb \texttt{<=} [i_1,\ldots,i_n] < ub;$

$[i_1,\ldots,i_n] \;\texttt{->}\; default\text{-}expr \;|\; [i_1,\ldots,i_n] < \mathit{shape} \}$

Optional parts!

Beauty Measure #2: Extend the Inference

Example: take (int[.] s, int[*] a):

default-expr is missing!

{ iv -> a[iv] | iv < s }

Type-inference cannot help!

Consider take ([0], a) where a::int[0,7] !

New Inference:

{ iv -> a[iv] | iv < s;
iv -> genarray (drop (shape (s), shape (a)), 0) }

Key Idea of the Inference

```
{ iv  -> a[iv] | iv < s }
```

Generate default from one expression

```
{ iv  -> a[iv] | iv < s;
  iv -> genarray (shape (a[0*s]), zero (a[0*s]) }
```

Rewrite to manifest some laziness

```
{ iv  -> a[iv] | iv < s;
  iv  -> genarray (drop (shape (s), shape (a)), 0) }
```

Leveraging Demand Analysis

genarray (shape (a[0*s]), zero (a[0*s])

=> Analysis of selection yields: in order to compute the shape of a[0*s], we only need to know the shape of a and the shape of s! (for details see “A Binding Scope Analysis for Generic Programs on Arrays”, IFL’05)

=> A systematic rewrite of the definition of selection yields that

shape (a[0*s]) = sel_s(shape(s), shape(a)) = drop(shape(s), shape(a))

(for details see “Tensor Comprehensions in SaC”, IFL’19)

Hence, we get overall:

genarray (drop (shape (s), shape (a)), 0)

Conclusions

- Array Comprehensions in the context of n-dimensional arrays / arbitrary tensors with homogeneous nesting is surprisingly challenging!
  - Full Expressiveness leads to very extensive specifications.
  - Range inference is non-trivial.
  - Default element inference is even harder.
- The Tensor Comprehensions presented here offer:
  - Full expressiveness
  - Flexibility in the degree of specificational demand
  - Novel default element inference that is independent of the type system
- Leads to a mechanism for manifesting laziness with an eager execution mechanism