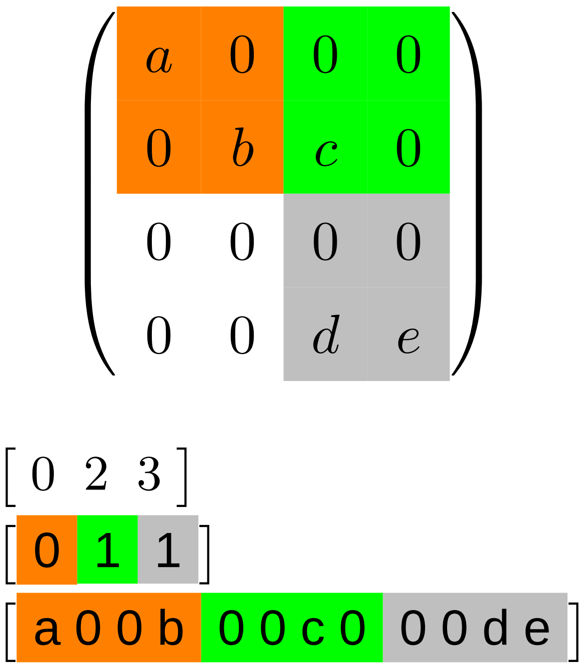
LL: a Little Language for Sparse Matrix Formats

Implementing A and $\cdot$ in $A \cdot x$

Gilad Arnold and Ras Bodík, UC Berkeley
Ali Sinan Köksal, EPFL
Johannes Hölzl, TU München
Mooly Sagiv, Tel-Aviv University

Problem: programming sparse matrix formats

BCSR



$$\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & c & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & d & e \end{pmatrix}$$

$$\begin{bmatrix} 0 & 2 & 3 \end{bmatrix}$$

[0 1 1]

[a 0 0 b 0 0 c 0 0 0 d e]

SpMV

```
for (i = 0; i < M; i++, y += 2) {
  double y0 = y[0], y1 = y[1];
  for (k = Ap[i]; k < Ap[i + 1];
       k++, Av += 4) {
    int j = 2 * k;
    double x0 = x[j], x1 = x[j + 1];
    y0 += Av[0] * x0; y1 += Av[2] * x0;
    y0 += Av[1] * x1; y1 += Av[3] * x1;
  }
  y[0] = y0; y[1] = y1;
}
```

➜ in LL: 1 LOC, same sequential performance + auto parallelism

➜ gets way more complicated: hundreds of LOC

LL: a small data-parallel language

Language for implementing sparse formats

- ➜ audience: efficiency programmers, maybe also experts
- ➜ uses: creation of format and operations on them

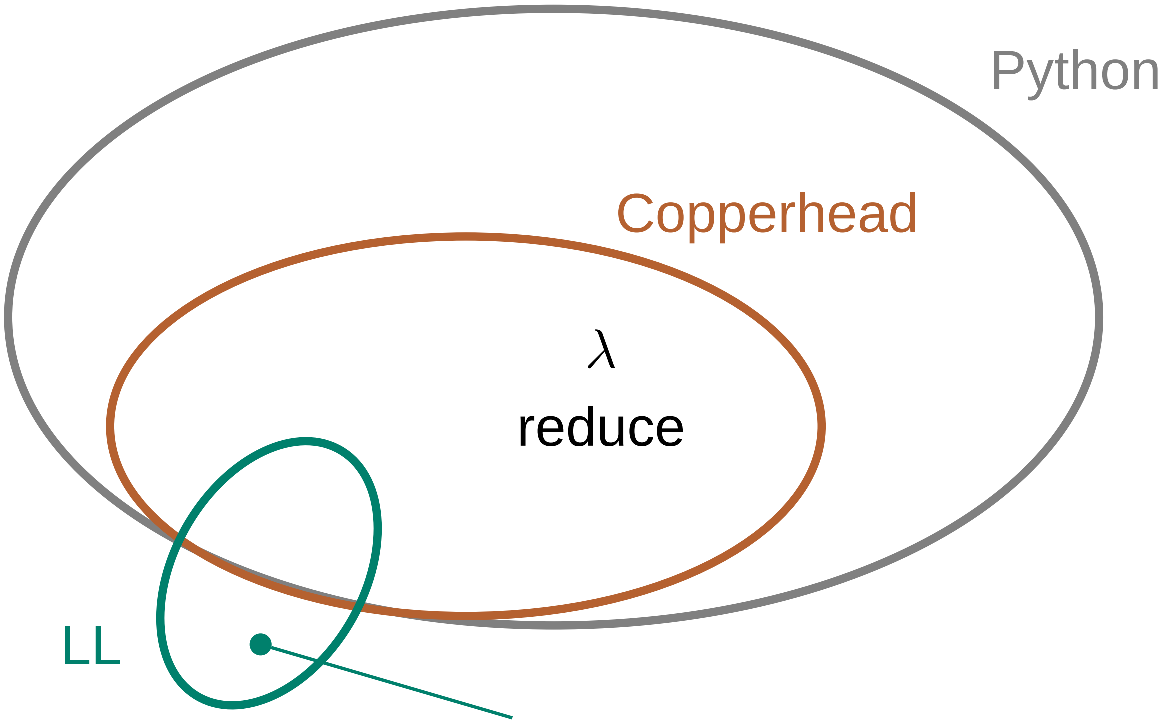
Our approach

- ➜ express sparse code using high-level functional idioms
- ➜ not a new DSL, just a useful subset

Research question: how small can a language be so that. . .

- ➜ we can (still) express sparse codes, naturally?
- ➜ we can compile with a simple compiler?
- ➜ we can verify correctness of programs?

LL, Copperhead, Python



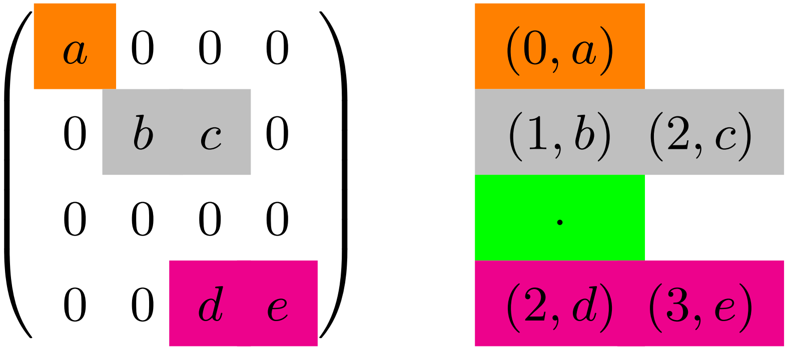
point-free programming

concise + good for compilation

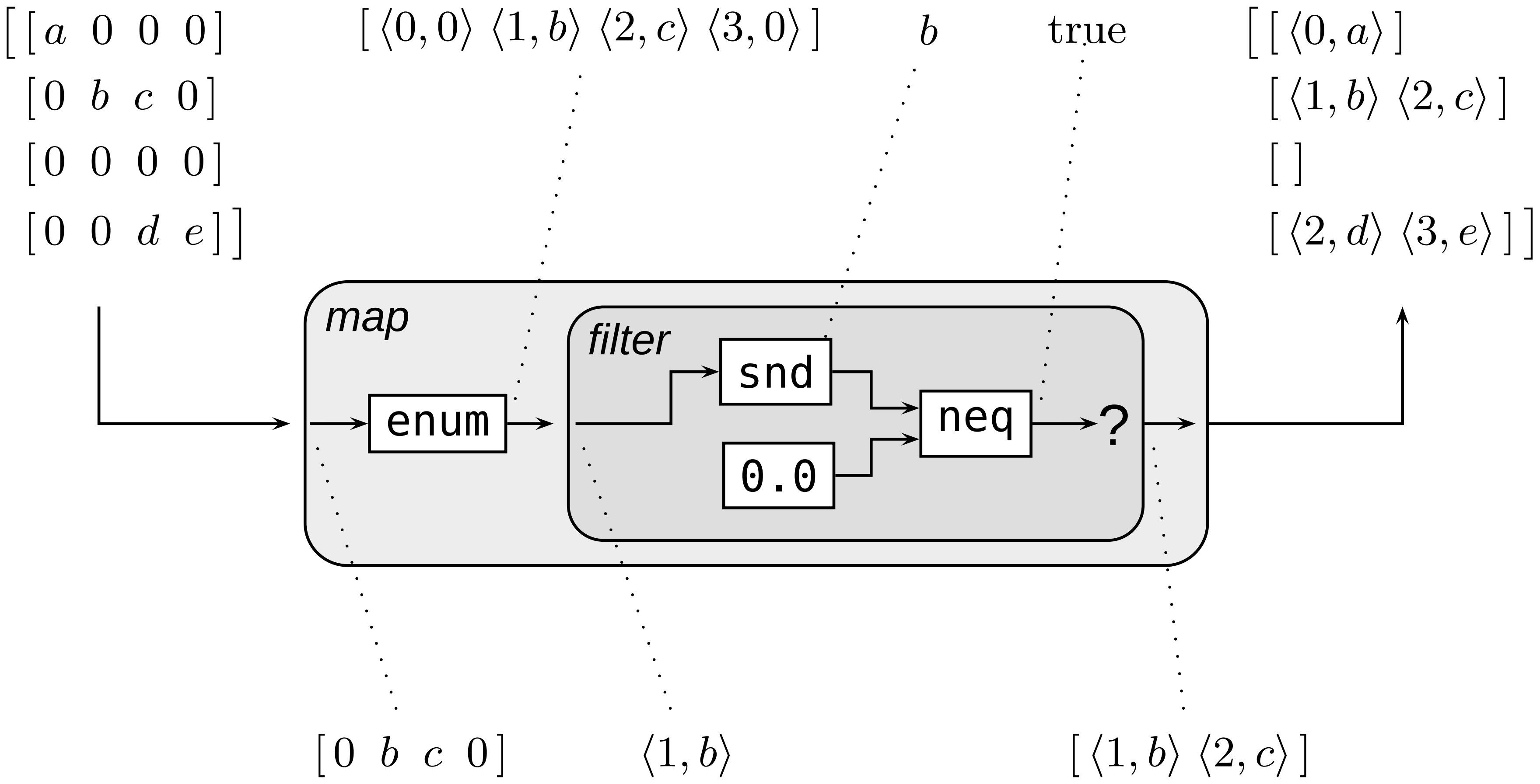
Benefits: data/code optimization, easy to implement, verification*

* not in this talk

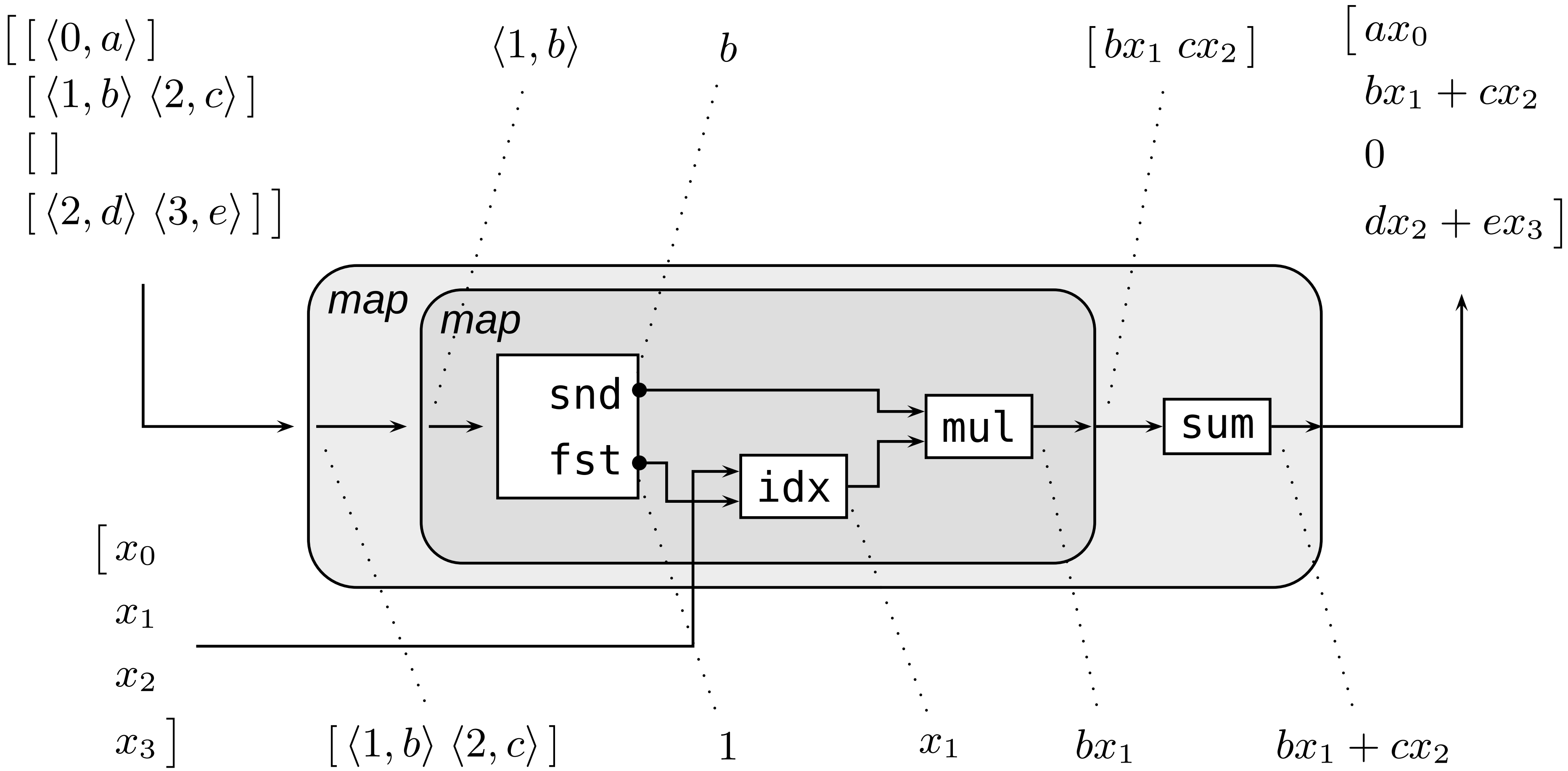
Implementing sparse codes in LL

dense	CSR
 $$\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & c & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & d & e \end{pmatrix}$$	$(0, a)$ $(1, b)\ \ (2, c)$ $\cdot$ $(2, d)\ \ (3, e)$
$$\begin{array}{l} [\,[a\ \ 0\ \ 0\ \ 0\,] \\ \ \ [0\ \ b\ \ c\ \ 0\,] \\ \ \ [0\ \ 0\ \ 0\ \ 0\,] \\ \ \ [0\ \ 0\ \ d\ \ e\,]\,] \end{array}$$	$$\begin{array}{l} [\,[\,\langle 0, a\rangle\,] \\ \ \ [\,\langle 1, b\rangle\ \langle 2, c\rangle\,] \\ \ \ [\,] \\ \ \ [\,\langle 2, d\rangle\ \langle 3, e\rangle\,]\,] \end{array}$$

CSR construction



CSR SpMV



$[\,[\,\langle 0, a\rangle\,]$
$[\,\langle 1, b\rangle\ \langle 2, c\rangle\,]$
$[\,]$
$[\,\langle 2, d\rangle\ \langle 3, e\rangle\,]\,]$

$\langle 1, b\rangle$

b

$[\,bx_1\ cx_2\,]$

$[\,ax_0$
$bx_1 + cx_2$
0
$dx_2 + ex_3\,]$

map

map

`snd`

`fst`

`idx`

`mul`

`sum`

$[\,x_0$
x_1
x_2
$x_3\,]$

$[\,\langle 1, b\rangle\ \langle 2, c\rangle\,]$

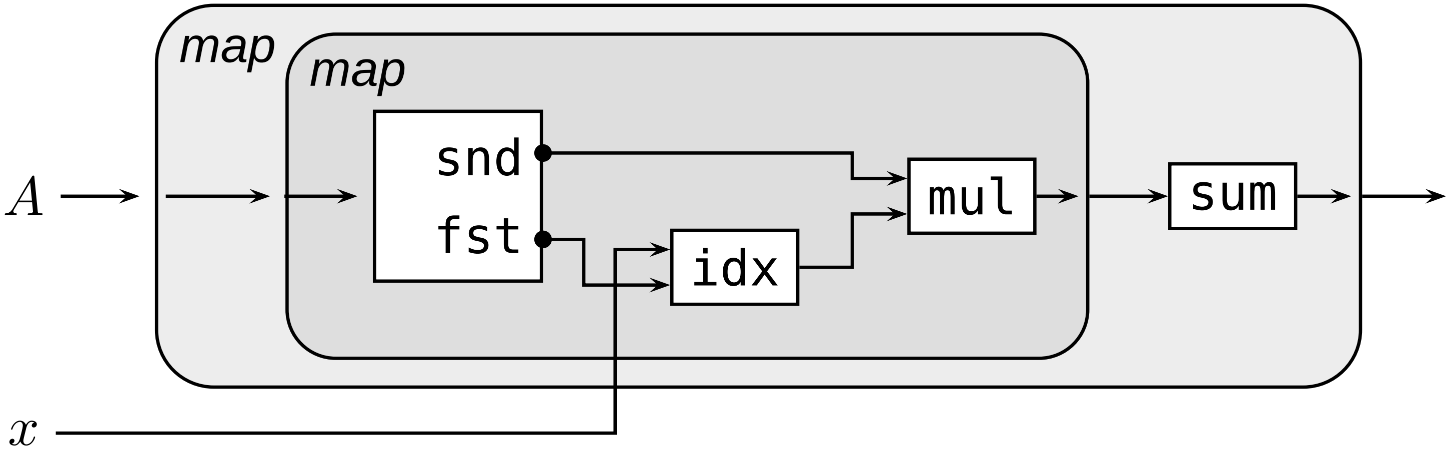
1

x_1

bx_1

$bx_1 + cx_2$

From dataflow to code



Strict specification:

```
A | [[(snd, (x, fst) | idx) | mul] | sum]
```

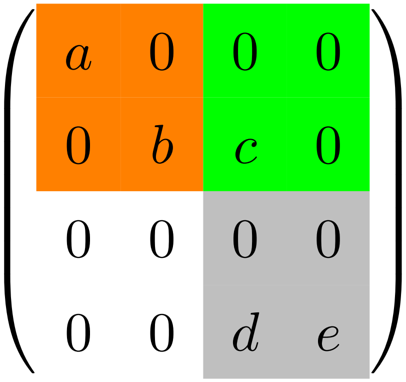
More convenient syntax:

```
A | [[snd * x[fst]] | sum]
```

Python-style:

```
[sum ([v * x[j] for j,v in Ai]) for Ai in A]
```

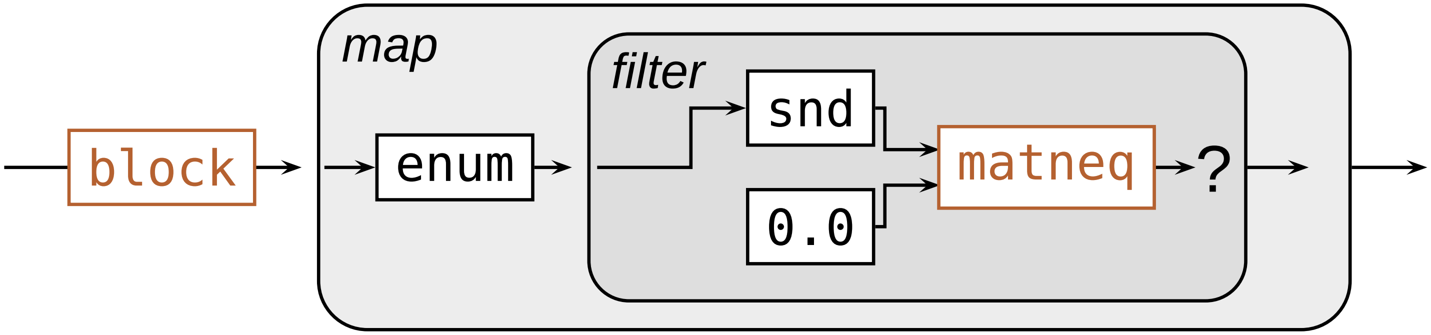
Case study: register-blocked CSR



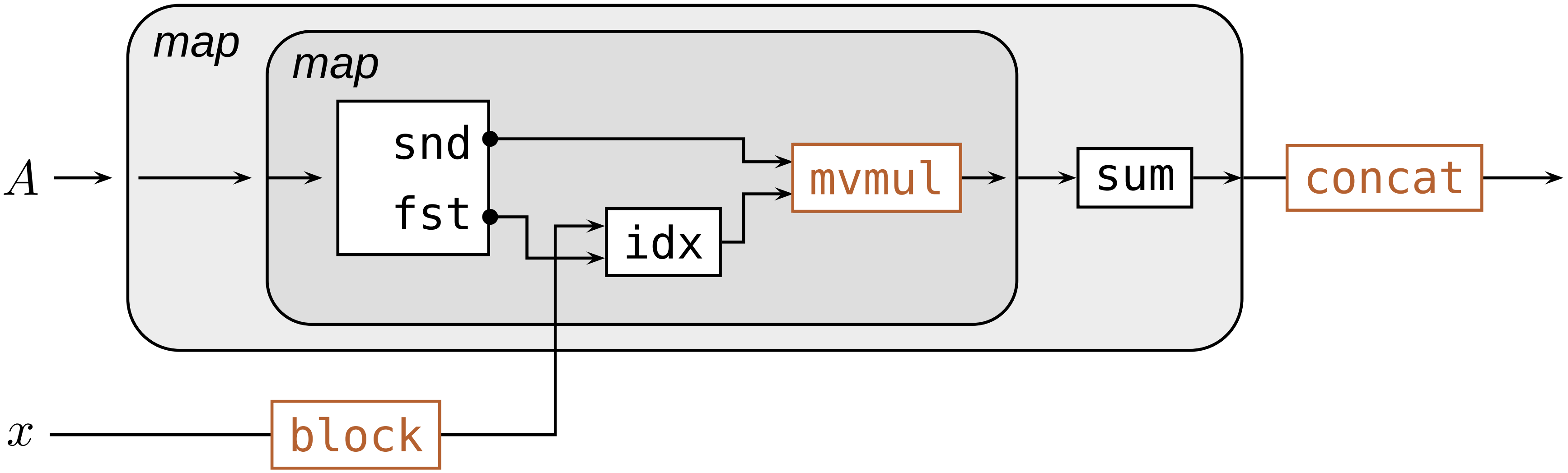
$$\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & c & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & d & e \end{pmatrix}$$

- how to express it in LL?
- how to compile it? (data, control)
- how is performance compared to handwritten code?

BCSR: construction



BCSR: SpMV

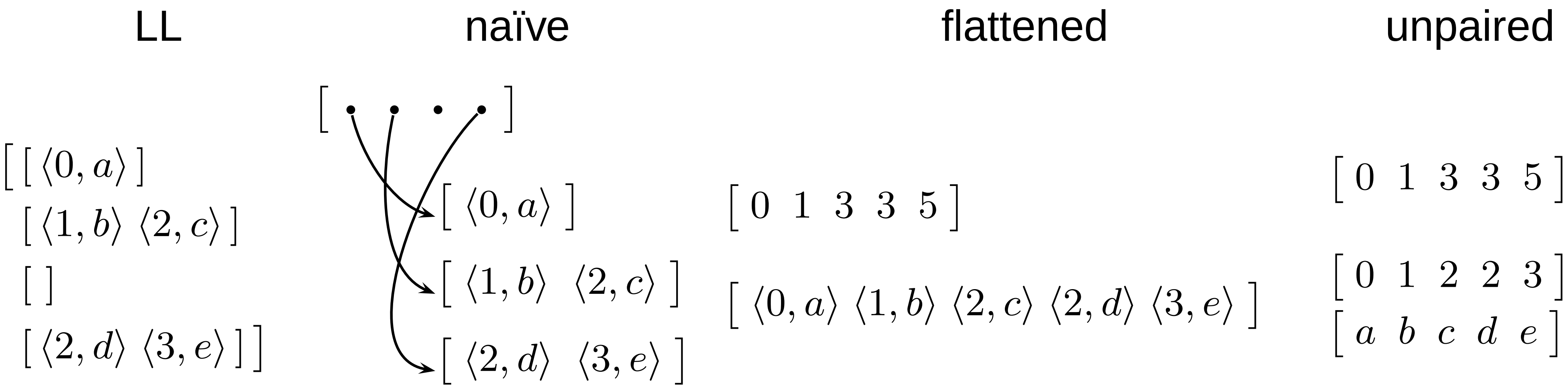


Compiling to efficient low-level code

- ➜ generating low-level data layout
- ➜ translating code

Compilation: data

- ➜ naïve translation is inappropriate
- ➜ algorithmic translation from LL types to C types
  - ① list of lists → linearized list + indirection
  - ② list of pairs → pair of lists (of same length)
- ➜ works for arbitrary (nested) types



LL

$$\begin{array}{l} [\,[\,\langle 0,a\rangle\,] \\ \;\;[\,\langle 1,b\rangle\ \langle 2,c\rangle\,] \\ \;\;[\,] \\ \;\;[\,\langle 2,d\rangle\ \langle 3,e\rangle\,]\,] \end{array}$$

naïve

$$[\ \bullet\ \ \bullet\ \ \bullet\ \ \bullet\ ]$$

$$\begin{array}{l} [\ \langle 0,a\rangle\ ] \\ [\ \langle 1,b\rangle\ \ \langle 2,c\rangle\ ] \\ [\ \langle 2,d\rangle\ \ \langle 3,e\rangle\ ] \end{array}$$

flattened

$$[\ 0\ \ 1\ \ 3\ \ 3\ \ 5\ ]$$

$$[\ \langle 0,a\rangle\ \langle 1,b\rangle\ \langle 2,c\rangle\ \langle 2,d\rangle\ \langle 3,e\rangle\ ]$$

unpaired

$$[\ 0\ \ 1\ \ 3\ \ 3\ \ 5\ ]$$

$$\begin{array}{l} [\ 0\ \ 1\ \ 2\ \ 2\ \ 3\ ] \\ [\ a\ \ b\ \ c\ \ d\ \ e\ ] \end{array}$$

Compilation: data (cont.)

- BCSR: demonstrating nested lists/pairs
  - but sublist indirections of constant size are redundant

LL

$$\left[\left[\langle 0, \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} \rangle \;\; \langle 1, \begin{pmatrix} 0 & 0 \\ c & 0 \end{pmatrix} \rangle \right] \;\; \left[\langle 1, \begin{pmatrix} 0 & 0 \\ d & e \end{pmatrix} \rangle \right] \right]$$

C

$$\begin{bmatrix} 0 & 2 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 2 & 4 & 6 \end{bmatrix}$$
$$\begin{bmatrix} 0 & 2 & 4 & 6 & 8 & 10 & 12 \end{bmatrix}$$
$$\begin{bmatrix} a\,0\,0\,b\,0\,0\,c\,0\,0\,0\,d\,e \end{bmatrix}$$

Compilation: data (cont.)

- ➜ BCSR: demonstrating nested lists/pairs
  - ➜ but sublist indirections of constant size are redundant
  - ➜ they can be inferred and made implicit
- ➜ results in the same layout used by hand-crafted code

LL

$$\left[\left[\langle 0, \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}\rangle \ \langle 1, \begin{pmatrix} 0 & 0 \\ c & 0 \end{pmatrix}\rangle\right] \left[\langle 1, \begin{pmatrix} 0 & 0 \\ d & e \end{pmatrix}\rangle\right]\right]$$

C

$$\begin{bmatrix} 0 & 2 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 2 & 4 & 6 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 2 & 4 & 6 & 8 & 10 & 12 \end{bmatrix}$$

$$\begin{bmatrix} a\,0\,0\,b\,0\,0\,c\,0\,0\,0\,d\,e \end{bmatrix}$$

Compilation: code

Syntax-directed translation: maps/filters to loops, pipelines to temporaries

CSR SpMV in LL:

```
[[snd * x[fst]] | sum]
```

In AST form:

```
map (
 map (
  (snd,
   (x, fst) | idx)
  | mul)
 | sum)
```

Translated C code:

```
for (/* i iterates on rows */) {
  GET_LPIF (t4, in, i);
  data_lf t2 = /* initialize */;

  for (/* j iterates on pairs */) {
    GET_PIF (f9, t4, j);
    data_f    t13 = lib_snd (t9);
    data_i    t17 = lib_fst (t9);
    data_plfi t15 = pair (x, t17);
    data_f    t14 = lib_idx (t15);
    data_pff  t12 = pair (t13, t14);
    data_f    t10 = lib_mul (t12);
    APPEND_PIF (t2, t10);
  }

  t5 = lib_sum (t2);
  APPEND_LPIF (out, t5);
}
```

Compilation: code (cont.)

① bottleneck: repeated allocation of temporary buffers

➜ move buffer allocation outside loops

② bottleneck: consecutive maps/reduces through buffers

➜ fusion: local, syntax-guided

③ bottleneck: fixed-sized sublists treated naïvely

➜ fix loop boundaries $\Longrightarrow$ enables unrolling
➜ constant pointer increments $\Longrightarrow$ ignore indirection buffer
➜ use length-enriched list types

④ bottleneck: repeatedly initializing pointer at inner loops

➜ move initialization through nested loops
➜ requires limited data dependency analysis

Compilation: data parallelism for free

Maps are data-parallel

```
for (/* i iterates on rows */) {

  ...

  APPEND (out, t5);
}
```

Compilation: data parallelism for free

Maps are data-parallel

- ➜ we parallelize them with OpenMP
- ➜ incorporate load balancing

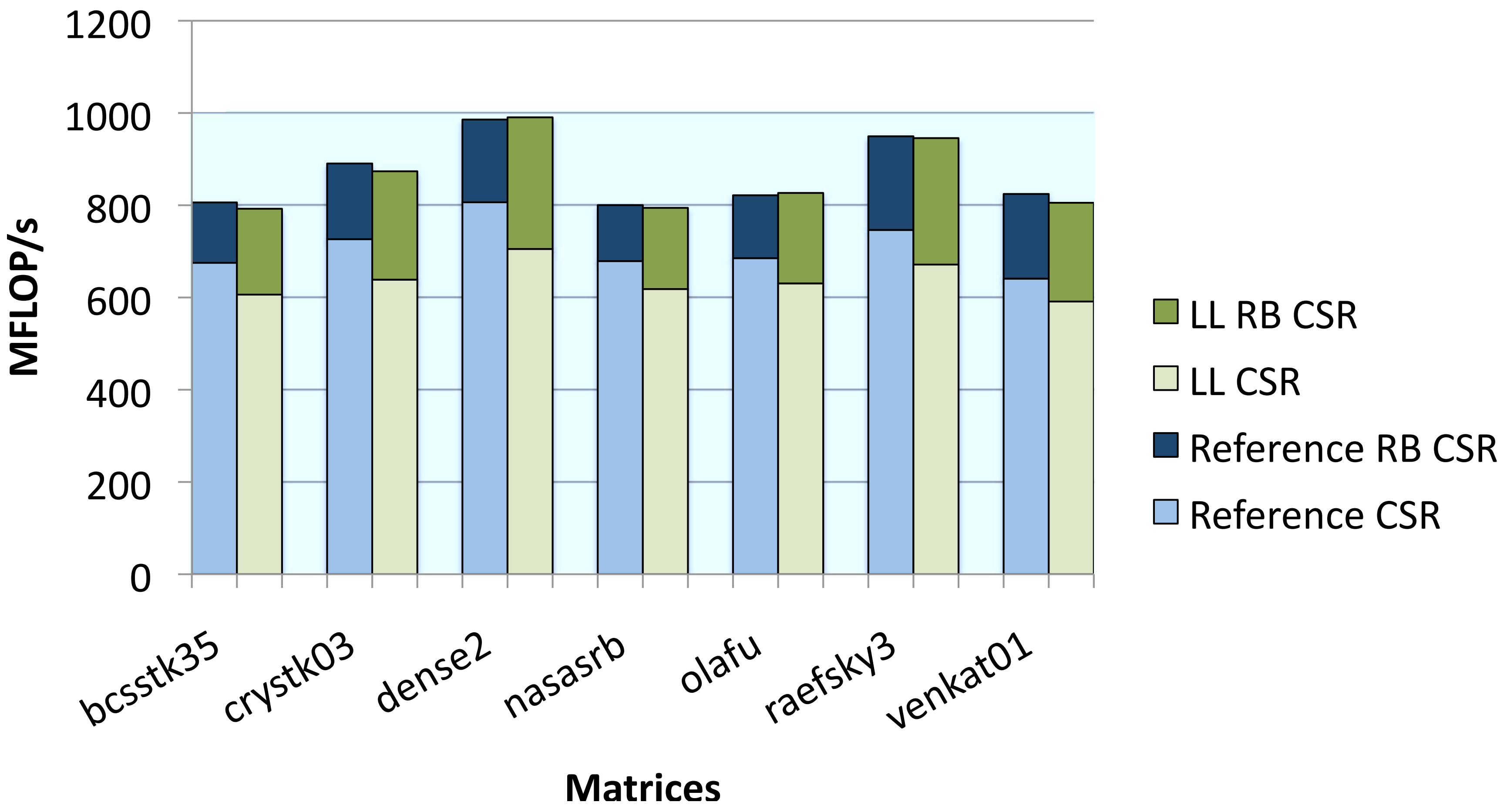
```
/∗ partition rows by workload ∗/
#pragma omp parallel
{
  /∗ t18, t19 get partition boundaries ∗/
  for (/∗ t18 <= i < t19 ∗/) {

    ...

    SET (out, i, t5);
  }
}
```

Empirical evaluation: sequential

Hypothesis: compiled LL performs comparably to handwritten C

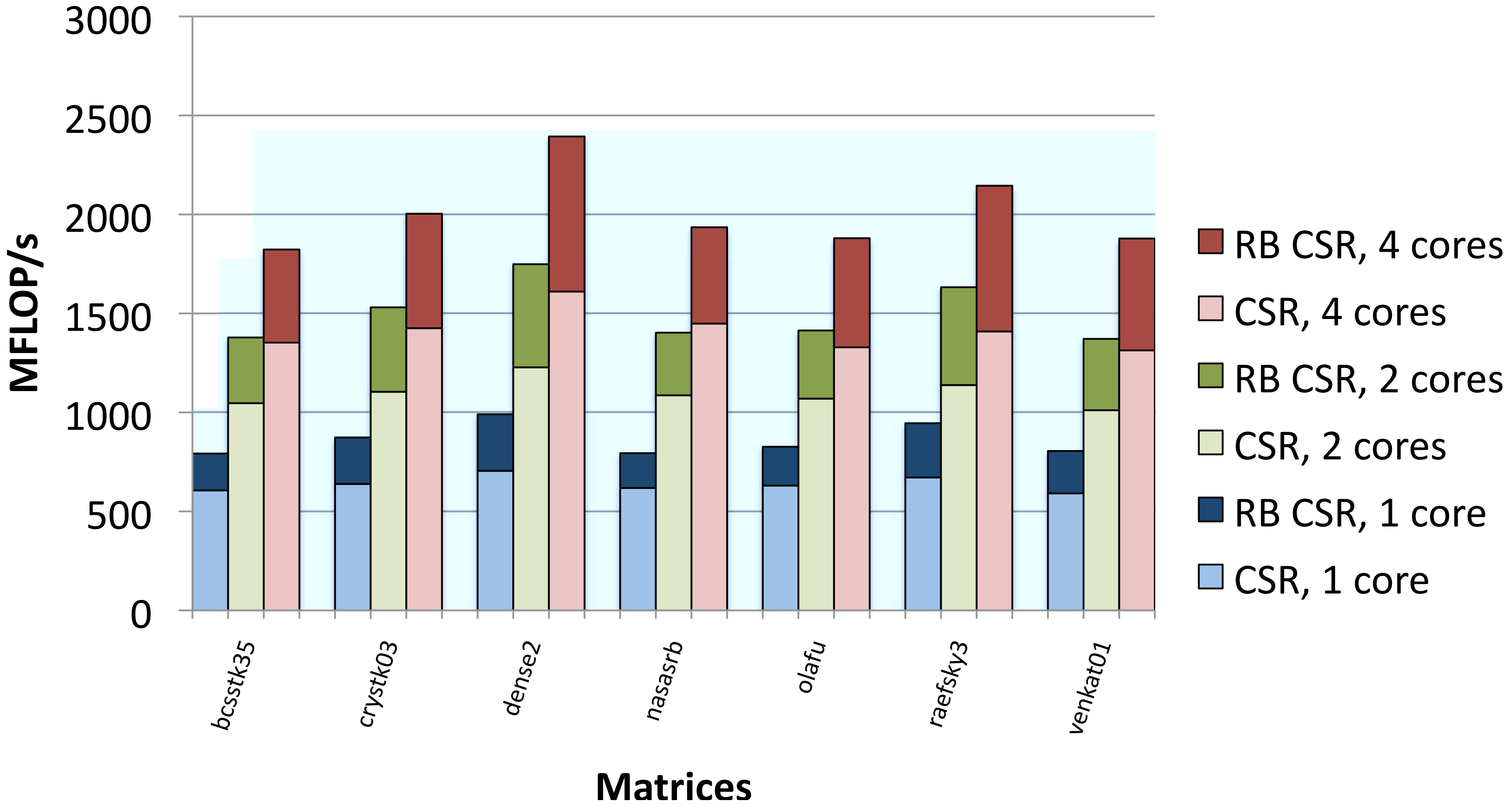


Median speedup: CSR 0.9, BCSR 0.99

Platform: 2.3 GHz single socket quad-core Opteron w/ 8GB RAM

Empirical evaluation: parallel

Hypothesis: compiled LL scales well, also w/ RB



Median speedup CSR: LL-2/LL-1 1.73, LL-4/LL-1 2.23

Median speedup BCSR: LL-2/LL-1 1.74, LL-4/LL-1 2.3

Conclusion: simple is good

- lightweight compiler implemented in 3 months
  - NESL, Data-Parallel Haskell: over a decade
  - Copperhead: 2 years
- success attributed to
  - limited expressivity
  - simple dataflow, functional semantics
  - strong typing
- formal verification of SpMV w/ multiple formats
  - first known result

Roadmap

- ➜ some translation steps still done manually
- ➜ implement more formats, support more operations
- ➜ ParLab tangents: synthesis, autotuning

Questions?