Productivity Language for Sparse Matrix Formats

Ali Sinan Köksal, Gilad Arnold, Rastislav Bodík, Mooly Sagiv

Motivation

Sparse matrix formats and operations (*e.g.*, SpMV) are hard to implement and verify in low-level languages like C.

RB-CSR

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

$[\,0\;\;2\;\;3\,]$

$[\,0\;\;1\;\;1\,]$

$[\,a\;0\;0\;b\;\;0\;0\;c\;0\;\;0\;0\;d\;e\,]$

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 += 6) {
    int j = Ai[k];
    double x0 = x[j], x1 = x[j + 1],
           x2 = x[j + 2];
    y0 += Av[0] * x0; y1 += Av[3] * x0;
    y0 += Av[1] * x1; y1 += Av[4] * x1;
    y0 += Av[2] * x2; y1 += Av[5] * x2;
  }
  y[0] = y0; y[1] = y1;
}
```

Research question: can sparse formats/operations be...

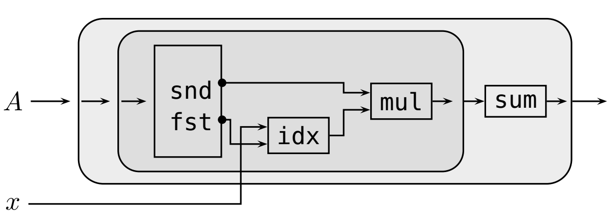
- Formulated as simple dataflow problems? Implemented in a small, high-level language?
- Formally proved correct?
- Compiled into efficient low-level code? Parallelized easily? Scale well?

Implementing sparse codes in LL

We represent sparse data with nested lists and pairs.

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

Example: CSR SpMV using dataflow.



Implemented in LL, concise variant:

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

Python-style comprehension also possible:

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

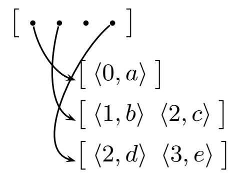
What about register-blocking? Need to (a) break the vector x into blocks, (b) replace scalar by matrix-vector multiplication, and (c) add concatenation of blocked results.

```
xb = block (2, x):
 A; [[(snd, xb[fst]); densemv]; sum]; concat
```

Code generation

A naïve nested layout is undesirable. Instead, we (a) flatten nested lists and (b) layout lists of pairs as a pair of lists.

naïve



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

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

$[\,0\;\;1\;\;2\;\;2\;\;3\,]$

$[\,a\;\;b\;\;c\;\;d\;\;e\,]$

Compilation is syntax-directed: pipelines are implemented using temporaries and maps/reduces using loops. Optimization is applied to fuse operations in adjacent maps/reduces. We use rich type information to specialize loop iteration counts and optimize pointer increments, and deploy data dependency analysis to eliminate redundant use of sublist boundaries indirection (below).

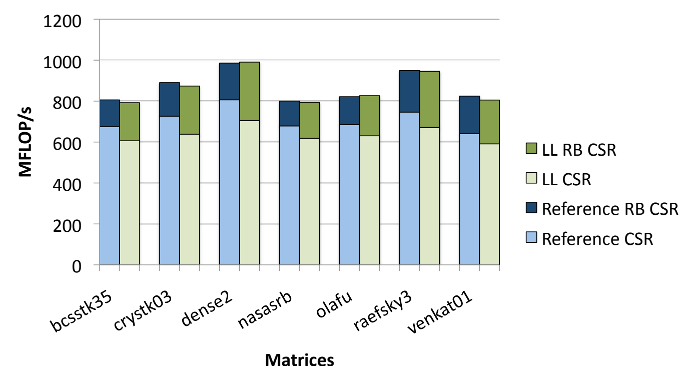
before

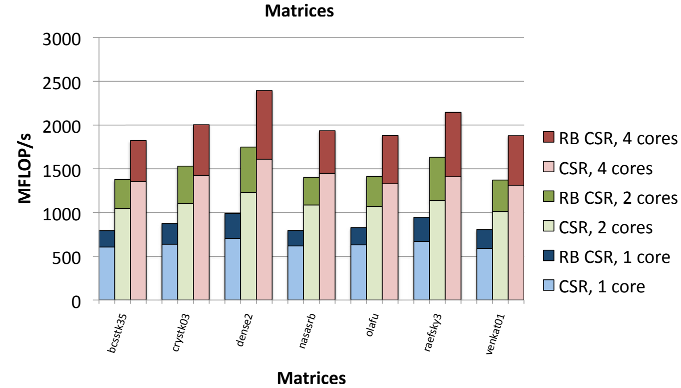
```
for (/* ... */) {
 for (/* ... */) {
  ...
   for (/* 0 <= t38 < # rows */) {
    GET_LLF (t36, t35, t38);
    for (/* 0 <= t45 # values */) {
      ...
    }

} } }
```

after

```
t36.len = 2;
t36.d = (*in).d.d1.d.d;
for (/* ... */) {
 for (/* ... */) {
  ...
   for (/* 0 <= t38 < # rows */) {

    for (/* 0 <= t45 < 2 */ ) {
      ...
    }
    t36.d += 2;
} } }
```

Our compiled CSR/RB-CSR SpMV performs close to hand-written (sequential) C code, and scales well to multiple cores.





Benchmarking environment: 2.3 GHz single socket quad-core AMD Opteron processor with 8GB of memory.