Progress on run-time specialization for matrix-vector multiplication

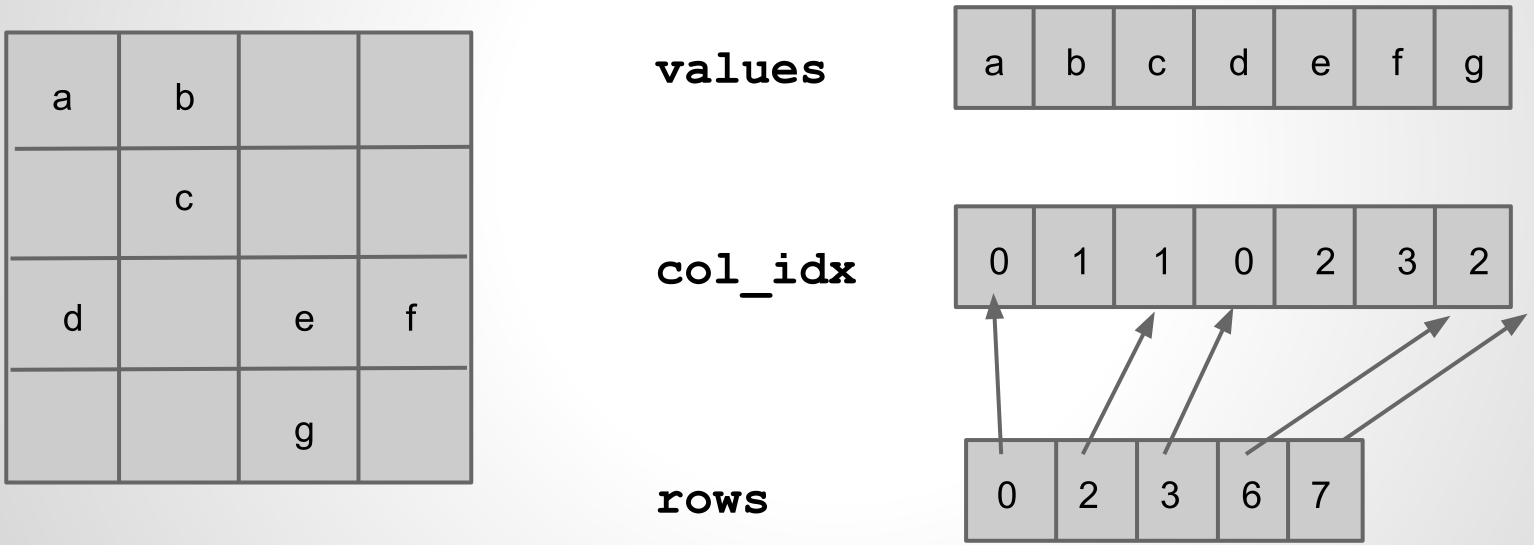
Sam Kamin, Univ. of Illinois Urbana-Champaign (now Google)
joint work with Maria Garzaran (UIUC) and Baris Aktemur (Ozyegin U, Istanbul)

Problem overview

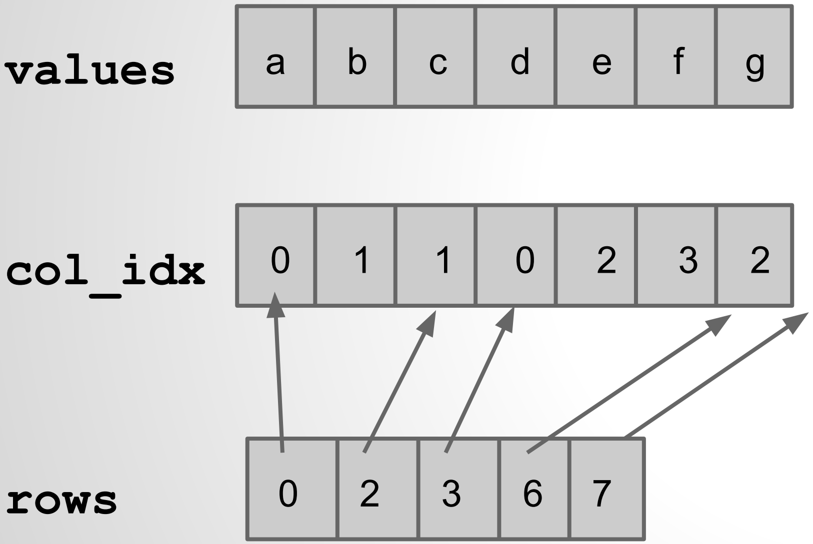
- Optimize sparse matrix/vector multiplication, using specialization on matrix (i.e. matrix is static over many multiplications)
- Issues
  - Many methods (generative and non-generative)
  - Performance varies by machine & matrix
- Goal: Library that *tunes* itself to machine, *chooses* best method for any matrix.

Sparse matrix-vector multiplication

Standard representation: compressed sparse rows (CSR):



Sparse matrix-vector multiplication




```
// M is mxm; w is output vector
for (i=0; i<m; i++) { // row i
  double w_i = 0;
  for (j=row[i]; j<row[i+1]; j++){
    w_i += *values * v[*col_idx];
    values++; col_idx++;
  }
  w[i] = w_i;
}
```

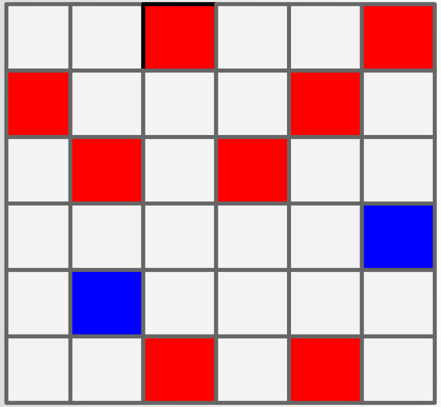
Unroll inner loop k times to get CSR_k. In our experiments, CSR_2 is often much faster than CSR (up to 25%); relatively small gains after that.

Specialization: Unfolding

a	b		
	c		
d		e	f
		g	

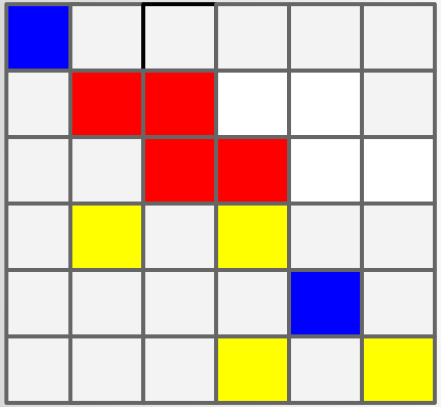
```
w[0] = a*v[0] + b*v[1];
w[1] = c*v[1];
w[2] = d*v[0] + e*v[2] + f*v[3];
w[3] = g*v[2];
```

Specialization: Arrange rows by non-zero count



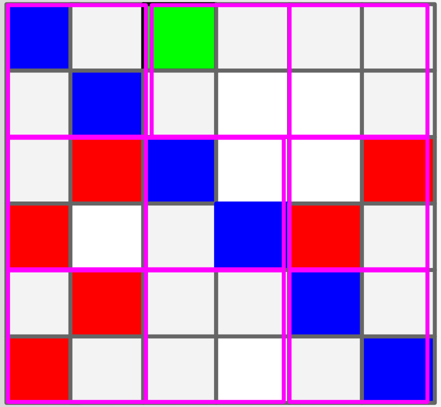
```
// Four rows have two elements
for (i in {0, 1, 2, 5}) {
  double w_i = 0;
  w_i = (*values++ * v[*col_idx++]);
  w_i += (*values++ * v
[*col_idx++]);
}
w[i] = w_i;
// Two rows have one element
for (i in {3, 4}) {
  double w_i = 0;
  w_i = (*values++ * v[*col_idx++]);
}
w[i] = w_i;
```

Specialization: Arrange rows by "stencil"



```
// Two rows rows have only diagonal elements
for (i in {0, 4})
  w[i] = (*values++ * v[i]);
// Two rows have elements at i, i+1
for (i in {1, 2})
  w[i] = *values++ * v[i]
         + *values++ * v[i+1];
// Two rows have elements at i-2, i
for (i in {3, 5})
  w[i] = *values++ * v[i-2]
         + *values++ * v[i];
```

Specialization: Arrange blocks by pattern



```
// Blocks with blue pattern
for (i,j in {(0,0),(2,2),(4,4)}) {
  w[i] += *values++ * v[j];
  w[i+1] += *values++ * v[j+1];
}
// Blocks with green pattern
for (i,j in {(2,0)})
  w[i] += *values++ * v[i];
// Blocks with red pattern
for (i,j in {(0,2), (4,2), (0,4)}) {
  w[i] += *values++ * v[i+1];
  w[i+1] += *values++ * v[i];
}
```

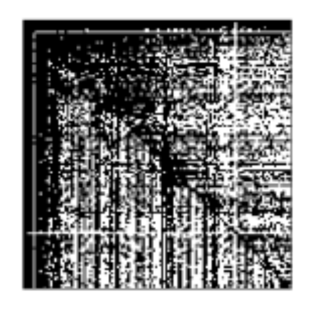
SpMV multiplication methods

- CSR, CSR_2, CSR_3, …
- Loop per row count (“CSRbyNZ”)
- Loop per stencil
- Unfolding
- OSKI (autotune to choose block size; pre-generated code for blocks)
- genOSKI (generate code for patterns of non-zeros in blocks)
- Diagonal; CSR with compression

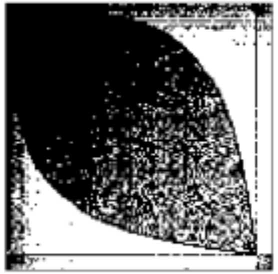
Timing results

- Most methods are row-oriented, so pretty easily parallelized
- OpenMP, 4 threads; clang -O3
- Variety of Intel machines: Core i5, Core i7, Xeon E7 & L7555
- Speed-ups relative to MKL

Selected matrices

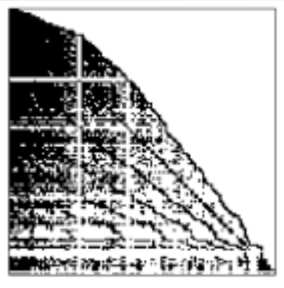


email-EuAll
n = 265,214
nz = 420,000

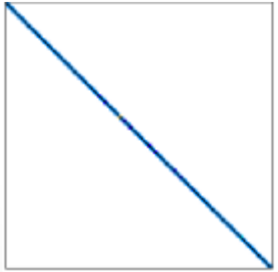


soc-epinions
n = 75,888
nz = 508,837

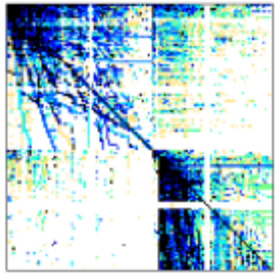
m133-b3
n = 200,200
nz = 800,800



web-NotreDame
n = 325,729
nz = 1,493,000



s3dkt3m2
n = 90,449
nz = 1,888,336



web-base1m
n = 1,000,005
nz = 3,105,000

Timing results for selected matrices

Speed-up relative to MKL, four threads				
	Core i7	Core i5	Xeon E7	Xeon L7555
email-EuAll	CSRbyNz (1.92)	genOSKI4 (1.88)	CSRbyNz (3.02)	Unfolding (4.17)
soc-epinions	CSRbyNz (1.98)	CSRbyNz (1.65)	Unfolding (2.07)	Stencil (2.04)
m133-b3	Unfolding (1.39)	Unfolding (1.31)	Unfolding (2.53)	Unfolding (3.84)
web-NotreDame	genOSKI4 (1.32)	genOSKI4 (1.17)	Unfolding (2.09)	Unfolding (3.05)
s3dkt3m2	Stencil (4.17)	genOSKI5 (1.55)	Stencil (1.43)	Stencil (1.37)
web-base1m	Unfolding (3.05)	Unfolding (1.5)	Unfolding (3.03)	Unfolding (4.96)

Why? Partial answer is matrix properties

Speed-up relative to MKL, four threads						
	nz	**row cnts**	**values**	**stencils**	**4patterns**	**5patterns**
email-EuAll	420,000	311	420,045	161,683	499	1088
soc-epinions	508,837	326	307854	49,442	3281	8439
m133-b3	800,800	1	2	200,200	489	1627
web-NotreDame	1,493,000	312	126,894	126,894	4135	9474
s3dkt3m2	1,888,336	23	29,116	935	97	143
web-base1m	3,105,000	370	222	504,865	4394	11,141

What to do next...

Goal: an auto-tuning library

- Gather machine info at install time
- At run time:
    - Use info about machine and matrix to choose best method
    - *Quickly* generate code
        - Only CSR_i has no latency; other methods require code gen or at least rearrangement of data

What to do next...

How quickly?

- In about half the cases, positions of non-zeros is known ahead of time. Almost all methods depend only on this information.
- On parallel machines, can hedge our bets by running no-latency code on some processors while generating code on others.

Rapid code generation

- Cannot use actual run-time compilation
  - We tried to use LLVM optimization passes, but even carefully chosen and tuned, this was much too expensive.
- Following times are for purpose-built, machine-level code generators

Timing results - sequential cross-over wrt CSR_2

	CSR_2	Stencil			CSRbyNZ			Unfolding		
		analysis	total	factor	analysis	total	factor	analysis	total	factor
s3dkt3m2	3840	58K	61K	16	23K	23K	6	283K	559K	145
engine	6077	126K	393K	64	35K	36K	6	649K	1.01M	166
torso2	27.9M	38K	46K	16	13K	13K	5	151K	198K	71

Summary

- The central question for us was whether we could speed up SpMV multiplication by code generation. In most cases, we can.
- Although most obvious method (unfolding) works pretty often, the problem is in general a lot more difficult than it looks.
- Biggest open question is how to determine what method works best in each circumstance.
- Beyond that, we have the pieces to put together the library we originally envisioned.