Runtime Specialization for Sparse Matrix-Vector Multiplication

Baris Aktemur, Ozyegin University, Istanbul
baris.aktemur@ozyegin.edu.tr

Joint work with Sam Kamin and Maria Garzaran (UIUC), and the group members at UIUC and Ozyegin
Thanks to TUBITAK and NSF for their support

Motivation

- Specialize sparse matrix-vector multiplication for a matrix that is multiplied by many vectors
- Many variants of multiplication function, varying performance on different platforms
- Goal: A library that chooses the best multiplication function for a given matrix

2

Sparse matrix-vector multiplication

$$\begin{bmatrix} 0 & 1 & 2 & 0 & 0 \\ 0 & 0 & 3 & 4 & 0 \\ 5 & 0 & 0 & 6 & 7 \\ 8 & 0 & 9 & 0 & 0 \\ 0 & 10 & 0 & 11 & 0 \end{bmatrix}$$

vals:
{1,2,3,4,5,6,7,8,9,10,11}

rows:
{0,2,4,7,9,11}

cols:
{1,2,2,3,0,3,4,0,2,1,3}

Non-zero elements of the matrix

The index of the first elt. of each row in **vals** array

Column index of each non-zero element

```
void mult(int n, int *rows, int *cols, double *vals,
          double *v, double *w) {
  for(int i = 0; i < n; i++) {
    double y = w[i];
    for(int k = rows[i]; k < rows[i+1]; k++)
      y += vals[k] * v[cols[k]];
    w[i] = y;
  }
}
```

Matrix
Output vector

w = Mv + w

3

Specialization: Unfolding

$$\begin{bmatrix} 0 & 1 & 2 & 0 & 0 \\ 0 & 0 & 3 & 4 & 0 \\ 5 & 0 & 0 & 6 & 7 \\ 8 & 0 & 9 & 0 & 0 \\ 0 & 10 & 0 & 11 & 0 \end{bmatrix}$$

vals:
{1,2,3,4,5,6,7,8,9,10,11}

rows:
{0,2,4,7,9,11}

cols:
{1,2,2,3,0,3,4,0,2,1,3}

```
void mult(double *v, double *w) {
   w[0] += 1 * v[1] + 2 * v[2];
   w[1] += 3 * v[2] + 4 * v[3];
   w[2] += 5 * v[0] + 6 * v[3] + 7 * v[4];
   w[3] += 8 * v[0] + 9 * v[2];
   w[4] += 10 * v[1] + 11 * v[3];
}
```

4

Specialization: Unfolding

$$\begin{bmatrix} 0 & 1 & 2 & 0 & 0 \\ 0 & 0 & 3 & 4 & 0 \\ 5 & 0 & 0 & 6 & 7 \\ 8 & 0 & 9 & 0 & 0 \\ 0 & 10 & 0 & 11 & 0 \end{bmatrix}$$

vals:
{1,2,3,4,5,6,7,8,9,10,11}

rows:
{0,2,4,7,9,11}

cols:
{1,2,2,3,0,3,4,0,2,1,3}

```
void mult(double *v, double *w) {
   w[0] += 1 * v[1] + 2 * v[2];
   w[1] += 3 * v[2] + 4 * v[3];
   w[2] += 5 * v[0] + 6 * v[3] + 7 * v[4];
   w[3] += 8 * v[0] + 9 * v[2];
   w[4] += 10 * v[1] + 11 * v[3];
}
```

5

Specialization: Unfolding

$$\begin{bmatrix} 0 & 1 & 2 & 0 & 0 \\ 0 & 0 & 3 & 4 & 0 \\ 5 & 0 & 0 & 6 & 7 \\ 8 & 0 & 9 & 0 & 0 \\ 0 & 10 & 0 & 11 & 0 \end{bmatrix}$$

vals:
{1,2,3,4,5,6,7,8,9,10,11}

rows:
{0,2,4,7,9,11}

cols:
{1,2,2,3,0,3,4,0,2,1,3}

```
void mult(double *v, double *w) {
   w[0] += 1 * v[1] + 2 * v[2];
   w[1] += 3 * v[2] + 4 * v[3];
   w[2] += 5 * v[0] + 6 * v[3] + 7 * v[4];
   w[3] += 8 * v[0] + 9 * v[2];
   w[4] += 10 * v[1] + 11 * v[3];
}
```

6

Specialization: Unfolding

- Source code vs machine code
- Compiler creates a pool of distinct values
- Assume 3 distinct values in the matrix: 3, 5, 9

```
w[0] += 9*v[2] + 9*v[3] + 5*v[8] + 3*v[9];
w[1] += 5*v[8] + 3*v[9] + 9*v[11];
```

```
double M[3] = {9, 5, 3}; // only 3 distinct values
double m0 = M[0], m1 = M[1], m2 = M[2];
w[0] += m0*v[2] + m0*v[3] + m1*v[8] + m2*v[9];
w[1] += m1*v[8] + m2*v[9] + m0*v[11];
```

7

Specialization: Unfolding

- Source code vs machine code
- CSE and arithmetic optimizations


```
double M[3] = {9, 5, 3}; // only 3 distinct values
double m0 = M[0], m1 = M[1], m2 = M[2];
w[0] += m0*v[2] + m0*v[3] + m1*v[8] + m2*v[9];
w[1] += m1*v[8] + m2*v[9] + m0*v[11];
```

```
double M[3] = {9, 5, 3};
double m0 = M[0], m1 = M[1], m2 = M[2];
double temp = m1*v[8] + m2*v[9];
w[0] += m0*(v[2] + v[3]) + temp;
w[1] += temp + m0*v[11];
```

8

Specialization: Unfolding

- Source code vs machine code
- Compiler creates a pool of distinct values
- Two cases:
  - Very few distinct values: reduced memory loads
  - Not so few distinct values: Bad locality
- Lesson learned: if very few distinct values, unfolding can make a big difference

9

Specialization: Unrolling

```
void mult(double *v, double *w) {
  ...
  for (i = 0; i < n; i++) {
    ww = 0.0;
    for (k = rows[i]; k <= rows[i+1]-3; k += 3) {
      ww += vals[k] * v[cols[k]]
          + vals[k + 1] * v[cols[k + 1]]
          + vals[k + 2] * v[cols[k + 2]];
    }
    for (; k < rows[i+1]; k++)
      ww += vals[k]*v[cols[k]];
    w[i] += ww;
  }
```

Unrolled 3 times

Left-over elements

- No runtime specialization is needed for this method.

10

Specialization: CSRbyNZ

- Group the rows with the same number of NZ elements
- Generate a loop for each group.

```
for (i = 0; i < 4; i++) {
    row = rows[a]; a++;
    w[row] += vals[b]   * v[cols[b]]
            + vals[b+1] * v[cols[b+1]];
    b += 2;
}
for (i = 0; i < 1; i++) {
    row = rows[a]; a++;
    w[row] += vals[b]   * v[cols[b]]
            + vals[b+1] * v[cols[b+1]]
            + vals[b+2] * v[cols[b+2]];
    b += 3;
}
```

4 rows with 2 elements

1 row with 3 elements

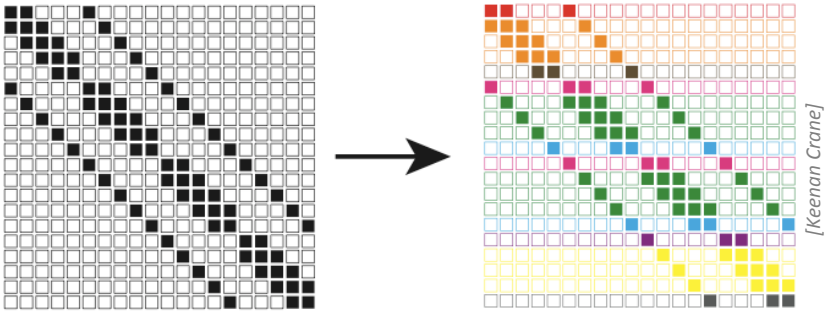
0	**1**	**2**	0	0
0	0	**3**	**4**	0
5	0	0	**6**	**7**
8	0	**9**	0	0
0	**10**	0	**11**	0

- Requires reordering of matrix elements
- Modest code size

11

Specialization: stencil

- Use the same code for rows that have the same shape wrt the diagonal: stencil.



[Keenan Crane]

12

```
for(each row i with stencil s={j0,j1,…,jk}) {
  int nextelt = start of elements from row i
  w[i] = vals[nextelt++]*v[i+j0] + … + vals[nextelt++]*v[i+jk];
}

for(each row i with stencil s'={j'0,j'1,…,j'm}) {
  int nextelt = start of elements from row i
  w[i] = m[nextelt++]*v[i+j'0] + … + m[nextelt++]*v[i+j'm];
}
…
```

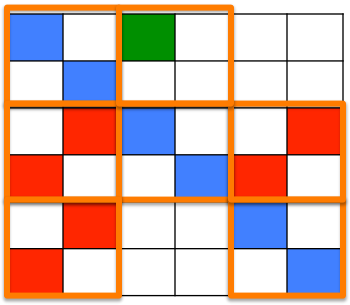
- Essentially, we are unrolling the code within a stencil.
- Requires reordering of matrix elements
- As many for-loops as there are stencils.
- Long code if there are too many stencils.

13

Specialization: genOSKI

- Group the blocks with the same block pattern
- Generate a loop for each group.

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



- In theory, all code generation can take place offline, but too many possibilities

14

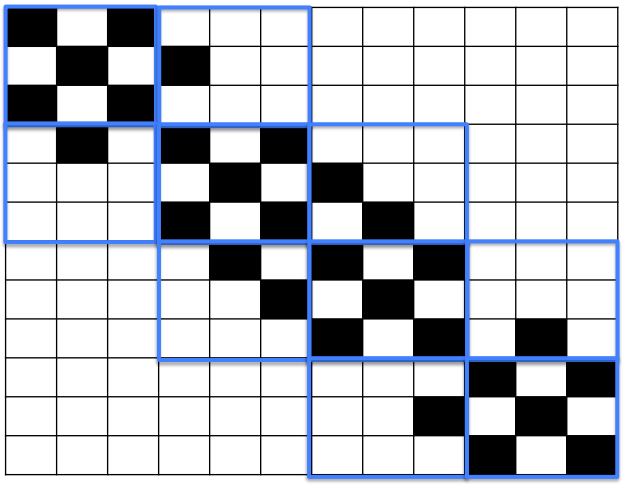
OSKI by Yelick's Group at Berkeley

```
void mult(int bn, int *b_rows, int *b_cols, double *b_vals,
          double *v, double *w) {
    for (int i = 0; i < bn; i++,y += 2) {
        double d0 = y[0];
        double d1 = y[1];
        for (int jj=b_row_start[i]; jj < b_row_start[i+1];
                jj++, b_col_idx++, b_value+=2*3) {
            d0 += b_value[0] * v[b_col_idx[0] + 0];
            d1 += b_value[3] * v[b_col_idx[0] + 0];
            d0 += b_value[1] * v[b_col_idx[0] + 1];
            d1 += b_value[4] * v[b_col_idx[0] + 1];
            d0 += b_value[2] * v[b_col_idx[0] + 2];
            d1 += b_value[5] * v[b_col_idx[0] + 2];
        }
        y[0] = d0;  y[1] = d1;
    }
}
```

2x3 Blocking

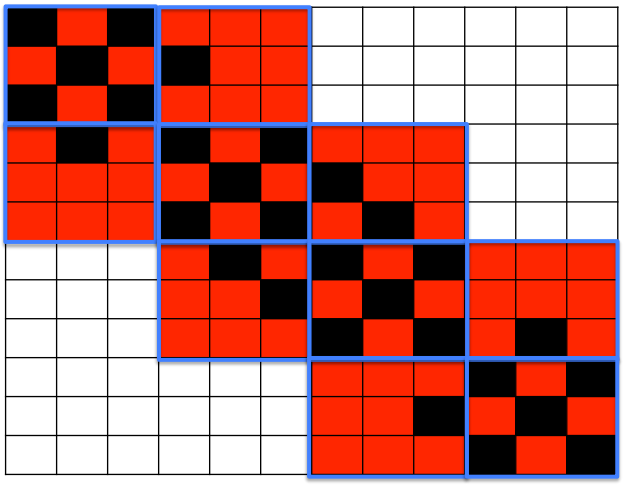
15

OSKI by Yelick's Group at Berkeley



16

OSKI by Yelick's Group at Berkeley



- Artificially fill in the blocks with zero values.

17

More…

- BiCSB [Buluc, Williams, Oliker, Demmel]
  - Sparse matrix data structure for parallel SpMV
  - Uses bitmasked register blocks
- CSX [Kourtis, Karakasis, Goumas, Koziris]
  - Compressed Sparse eXtended format
  - Allows to have a variety of structures in a sparse matrix
  - Does code generation
- Diagonal
  - Allows vectorization for dense bands
  - Requires zero-fills

18

Many choices

- Unrolling
- Unfolding
- CSRbyNZ
- Stencil
- GenOSKI
- OSKI
- BiCSB
- CSX
- Diagonal
- Intel MKL

19

Timings

- Most methods are row-oriented. Easy to parallelize
- 4 threads. OpenMP.
- Compile with `icc -O3`
- Repeat multiplication for 10000 times

20

Machines

Name	Processor	Cores	L1(I/D)	L2	L3	Mem	OS
loome2	Intel i7 880	4	128K	1M	8M	8	CentOS 5.8
loome3	Intel i5 2400	4	32K	256K	6M	8	CentOS 5.8
i2pc3	Intel Xeon E7-4860	40	64K	256K	24M	64	Sci.Linux 6.3
turing	Intel Xeon E5-2620	6	32K	256K	15M	16	Ubuntu 12.04
milner	AMD FX-8320	8	64K/16K	2M	8M	8	ArchLinux 3.13

Used icc 14.0 on all machines except milner, where we used gcc 4.8.2.

21

Matrices

Matrix	rows	nnz(M)	size(MB)	nz/row	group
email-EuAll	265214	0.420	5.81	1.5	SNAP
cit-HepPh	34546	0.421	4.95	12.2	SNAP
soc-Epinions1	75888	0.508	6.11	6.7	SNAP
soc-sign	77357	0.516	6.20	6.6	SNAP
web-NotreDame	325729	1.497	18.37	4.5	SNAP
webbase-1M	1000005	3.105	39.35	3.1	Williams
e40r5000	17281	0.553	6.40	32	SPARSKIT
fidapm11	22294	0.617	7.15	27.7	SPARSKIT
fidapm37	9152	0.766	8.80	83.6	SPARSKIT
m133-b3	200200	0.801	9.92	4	JGD_Homology
torso2	115967	1.033	12.20	8.9	Norris
fidap011	16614	1.091	12.50	65.6	SPARSKIT
cfd2	123440	1.604	18.83	12.9	Rothberg
m14b	214765	1.679	20.03	7.8	Dimacs10
s3dkt3m2	90449	1.888	21.95	20	CYLSHELL
conf6_0-8x8-20	49152	1.917	22.12	39	QCD
ship_003	121728	1.949	22.77	16	DNVS
cage12	130228	2.032	23.75	15.6	vanHeukelum
debr	1048576	2.097	27.99	1.9	AG-Monien
mc2depi	525825	2.100	26.04	3.9	Williams
s3dkq4m2	90449	2.259	26.19	24.9	CYLSHELL
engine	143571	2.424	28.29	16.8	Dimacs10
thermomech_dK	204316	2.846	33.35	13.9	Botonakis

22

Matrix	loome2	Speedup	loome3	speedup
email-EuAll	CSRbyNZ	1.42	BiCSB	1.77
cit-HepPh	CSRbyNZ	1.11	CSRbyNZ	1.16
soc-Epinions1	CSRbyNZ	1.48	BiCSB	1.49
soc-sign	unfolding	2.74	unfolding	2.64
web-NotreDame	genOSKI4	1.13	MKL	1.00
webbase-1M	unfolding	1.27	unfolding	1.30
e40r5000	stencil	1.17	genOSKI4	1.71
fidapm11	CSRbyNZ	1.39	CSX	1.18
fidapm37	genOSKI4	1.33	CSX	1.38
m133-b3	unfolding	1.19	unfolding	1.16
torso2	stencil	1.66	stencil	1.43
fidap011	CSX	1.43	genOSKI4	1.11
cfd2	CSX	1.18	CSX	1.18
m14b	CSRbyNZ	1.38	BiCSB	1.62
s3dkt3m2	stencil	1.62	stencil	1.51
conf6_0-8x8-20	stencil	1.40	genOSKI4	1.40
ship_003	CSRbyNZ	1.03	CSX	1.19
cage12	CSX	1.22	genOSKI4	1.11
debr	CSRbyNZ	1.14	CSRbyNZ	1.09
mc2depi	stencil	1.26	stencil	1.21
s3dkq4m2	stencil	1.51	stencil	1.45
engine	unfolding	3.22	unfolding	2.83
thermomech_dK	genOSKI4	1.09	genOSKI4	1.09
Average		1.45		1.42

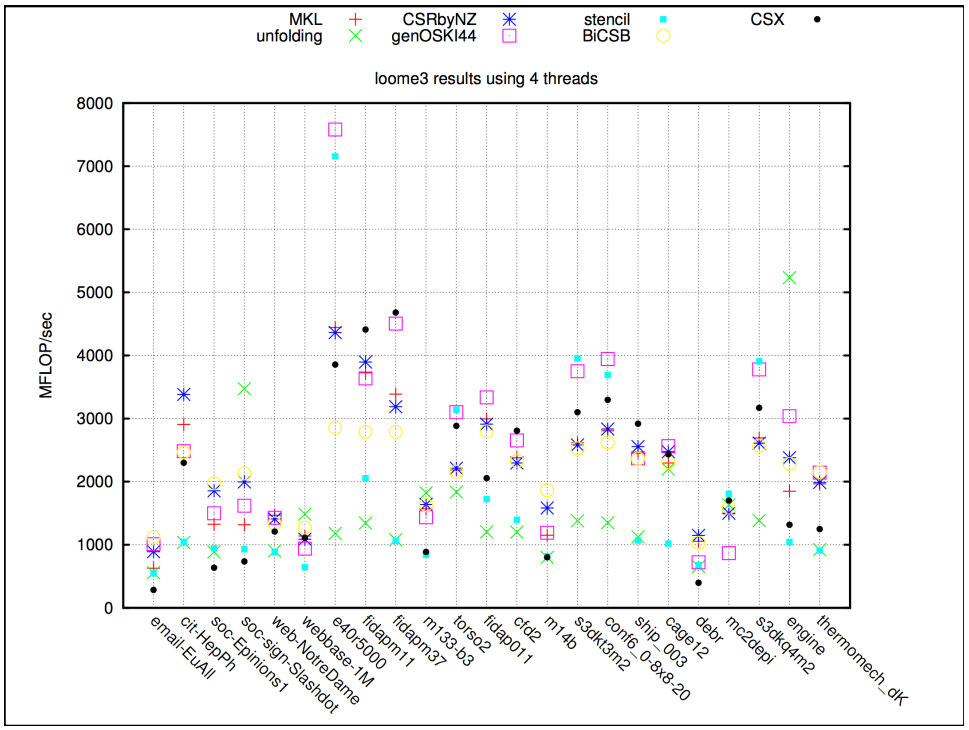
23

Matrix	i2pc3	speedup	turing	speedup	milner	speedup
email-EuAll	unfolding	1.55	CSRbyNZ	2.05	genOSKI4	2.11
cit-HepPh	unfolding	1.07	CSRbyNZ	1.30	CSRbyNZ	1.53
soc-Epinions	unfolding	2.01	CSRbyNZ	2.20	CSRbyNZ	2.17
soc-sign	unfolding	2.82	unfolding	2.77	unfolding	3.52
web-NotreD	genOSKI4	1.16	genOKI4	1.23	genOSKI4	1.57
webbase-1M	unfolding	1.78	unfolding	1.31	genOSKI4	1.38
e40r5000	stencil	1.01	MKL	1.00	genOSKI4	1.73
fidapm11	MKL	1.00	CSRbyNZ	1.96	CSRbyNZ	1.40
fidapm37	MKL	1.00	MKL	1.00	genOSKI4	1.38
m133-b3	unfolding	1.36	CSRbyNZ	1.91	CSRbyNZ	1.50
torso2	unfolding	1.38	unfolding	2.08	genOSKI5	2.03
fidap011	MKL	1.00	genOSKI4	1.49	genOSKI4	1.45
cfd2	MKL	1.00	genOSKI4	1.09	genOSKI4	1.55
m14b	stencil	1.21	BiCSB	1.56	CSRbyNZ	1.43
s3dkt3m2	stencil	1.19	stencil	2.16	genOSKI5	2.04
conf6_0-8x8	MKL	1.00	genOSKI4	1.62	genOSKI4	1.76
ship_003	unfolding	1.04	CSRbyNZ	1.00	genOSKI4	1.31
cage12	unfolding	1.44	genOSKI4	1.10	genOSKI4	1.28
debr	CSRbyNZ	1.45	BiCSB	1.15	CSRbyNZ	1.36
mc2depi	unfolding	1.49	genOSKI5	1.52	genOSKI5	1.76
s3dkq4m2	MKL	1.00	stencil	1.81	genOSKI5	1.73
engine	unfolding	3.82	unfolding	6.20	unfolding	1.82
thermomech	genOSKI4	1.03	genOSKI4	1.41	genOSKI4	1.61
Average		1.42		1.77		1.71

Matrices

Matrix	#stencils	genOSKI4	genOSKI5	distVals	Row_nz
email-EuAll	161683	499	1088	420045	311
cit-HepPh	31814	315	683	421578	162
soc-Epinions1	49442	3281	8439	307854	326
soc-sign	40649	1212	2867	2	279
web-NotreDame	126894	4135	9474	1497134	312
webbase-1M	504865	4394	11141	222	370
e40r5000	601	130	265	368750	25
fidapm11	4682	1197	2576	88275	22
fidapm37	8391	876	2102	350166	70
m133-b3	200200	489	1627	2	1
torso2	3148	81	108	806653	3
fidap011	7432	1684	3315	211502	71
cfd2	46535	3422	7823	1480984	27
m14b	172130	3331	9099	1679018	22
s3dkt3m2	935	97	143	29116	23
conf6_0-8x8-20	648	22	156	84553	1
ship_003	105098	3982	15702	49424	60
cage12	130228	1100	4495	350	28
debr	786432	7	9	2097149	3
mc2depi	2298	50	57	3584	3
s3dkq4m2	1131	380	680	8632	29
engine	84195	108	538	1	147
thermomech_dK	204290	17	329	1967432	9

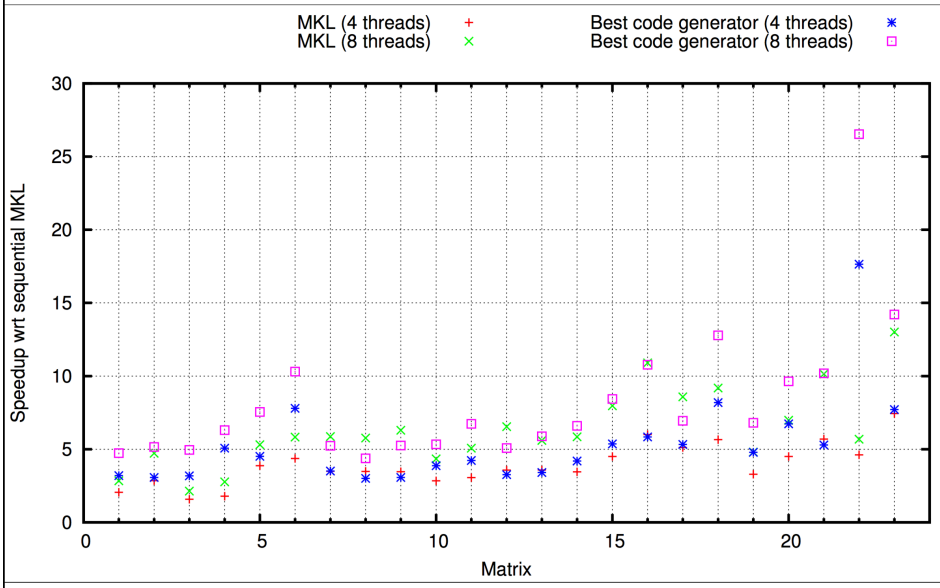
25



		loome2	loome3	i2pc3	turing	milner
CSR	Avg. Speedup	0.96	0.90	0.83	0.97	-
	# matrices is best	0	0	0	0	-
	# matrices is better	7	6	4	6	-
	Avg. Speedup if better	1.08	1.09	1.14	1.15	-
stencil	Avg. Speedup	0.77	0.79	1.00	0.81	0.93
	# matrices is best	6	4	3	2	0
	# matrices is better	6	6	11	6	7
	Avg. Speedup if better	1.44	1.42	1.31	1.68	1.6
genOSKI4	Avg. Speedup	1.06	1.15	1.04	1.28	1.58
	# matrices is best	3	5	2	6	11
	# matrices is better	15	15	12	19	23
	Avg. Speedup if better	1.18	1.31	1.22	1.37	1.58
genOSKI5	Avg. Speedup	0.98	1.05	0.99	1.19	1.50
	# matrices is best	0	0	0	1	4
	# matrices is better	10	15	11	15	23
	Avg. Speedup if better	1.22	1.31	1.23	1.38	1.50
unfolding	Avg. Speedup	0.83	0.82	1.22	1.03	0.84
	# matrices is best	4	4	11	4	2
	# matrices is better	5	5	13	6	4
	Avg. Speedup if better	1.91	1.80	1.67	2.6	2.05
CSRbyNZ	Avg. Speedup	1.13	1.09	1.09	1.29	1.32
	# matrices is best	7	2	1	8	6
	# matrices is better	18	15	12	17	23
	Avg. Speedup if better	1.18	1.16	1.28	1.42	1.32
CSX	Avg. Speedup	0.86	0.89	-	-	-
	# matrices is best	3	4	-	-	-
	# matrices is better	9	10	-	-	-
	Avg. Speedup if better	1.27	1.19	-	-	-
BiCSB	Avg. Speedup	0.66	1.07	0.63	0.99	-
	# matrices is best	0	3	0	0	-
	# matrices is better	2	10	3	11	-
	Avg. Speedup if better	1.04	1.30	1.26	1.22	-
Best Specialization	Avg. speedup	1.45	Xx	XX	XX	1.71
	#matrices is better	23	19	17	21	23
	Avg. Speedup if better	1.45	1.43	1.58	1.84	1.71

27

4 vs. 8 Threads



28

PART I Take Away

- Most of the time, speed up can be obtained by code generation
- Matrix properties partially explain the speedups
- Different machines, different results

29

PART II – Rapid Code Generation

- For torso2 (~1M nz)

```
$ time icc -O3 *.c
real    1m32.878s
user    1m27.501s
sys     0m4.516s
```

- For webbase (~3M nz)

```
$ time icc -O3 *.c
real    19m20.506s
user    19m0.403s
sys     0m15.893s
```

- A compiler that allows generation of C at runtime
  - Usual quote/unquote syntax
  - Produces LLVM IR code, then lets LLVM do the rest
  - Still, code generation times in the orders of 10K x multiplication
- Many optimizations are unnecessary, but a custom list of passes is even too slow

30

A purpose-built compiler

- Directly emit binary code to a memory buffer

```
for (auto &eltIterator : elements) {
  int colIndex = (*eltIterator)->col;
  //  movsd "sizeof(double)*colIndex"(%rdi), %xmm"vRegIndex"
  emitMOVSDrmInst(sizeof(double)*colIndex, X86::RDI, vRegIndex);
  vRegIndex++;
}
```

31

Purpose-built code generator

- A library that directly emits binary code

Matrix	Nz	Stencil			CSRbyNZ			Unfolding		
		Analysis	Total	Factor	Analysis	Total	Factor	Analysis	Total	Factor
s3dkt3m2	1.888M	58	61	16	23	23	6	283	559	145
engine	2.424M	126	393	64	35	36	6	649	1010	166
torso2	1.033M	38	46	16	13	13	5	151	198	71

time in msec

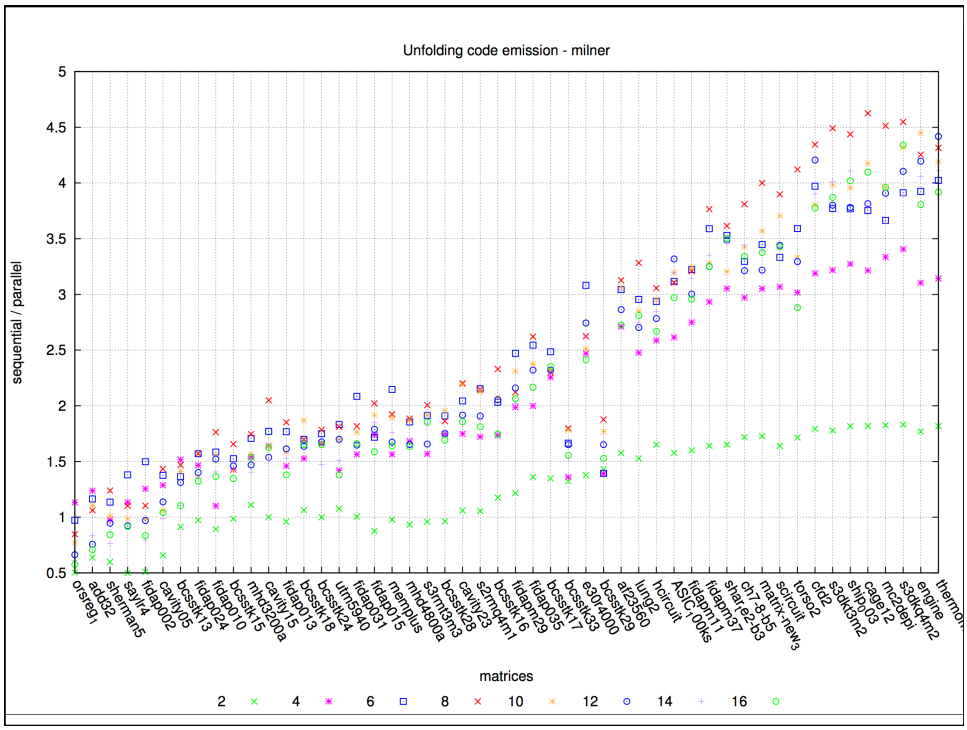
w.r.t. running plain CSR multiplication with this matrix

32

Generating code in parallel

- Emit to binary buffers in parallel
- Unfolding is most suitable for this

33



Generating Assembly Code

- Contrary to the "don't write assembly manually" advice
- Reverse-engineering approach
  - Examine the compiler's output, generate instructions similarly
- Emitting code in a much more direct and simpler way than a compiler does
- How good is the produced code?

35

Code Quality

(CSRbyNZ – sequential)

email-EuAll	0.960	cfd2	0.990
cit-HepPh	0.861	m14b	0.980
soc-Epinions1	0.867	s3dkt3m2	0.965
soc-sign	0.782	conf6_0-8x8-20	0.975
web-NotreDame	0.956	ship_003	0.962
webbase-1M	1.025	cage12	0.964
e40r5000	0.949	debr	1.004
fidapm11	0.991	mc2depi	0.967
fidapm37	0.910	s3dkq4m2	0.956
m133-b3	0.999	engine	0.927
torso2	0.937	thermomech_dK	0.999
fidap011	0.946		

Ratio of our code to icc-compiled code.
Small is better.

36

Code Quality

(GenOSKI 3x3 – sequential)

email-EuAll	1.165	cfd2	0.883
cit-HepPh	1.012	m14b	0.923
soc-Epinions1	0.960	s3dkt3m2	0.943
soc-sign	0.981	conf6_0-8x8-20	0.968
web-NotreDame	0.912	ship_003	0.890
webbase-1M	0.988	cage12	0.954
e40r5000	0.956	debr	1.011
fidapm11	0.915	mc2depi	0.987
fidapm37	0.984	s3dkq4m2	0.949
m133-b3	0.996	engine	0.906
torso2	0.978	thermomech_dK	0.984
fidap011	0.903		

Ratio of our code to icc-compiled code.
Small is better.
email-EuAll: GenOSKI performs well. Should examine.

37

Code Quality

(Stencil – parallel)

email-EuAll	0.996	cfd2	0.765
cit-HepPh	0.861	m14b	1.098
soc-Epinions1	0.862	*s3dkt3m2	0.925
soc-sign	0.872	*conf6_0-8x8-20	1.022
web-NotreDame	0.995	ship_003	0.954
webbase-1M	0.980	cage12	1.084
e40r5000*	0.845	debr	1.308
fidapm11	0.530	*mc2depi	0.877
fidapm37	0.762	*s3dkq4m2	0.941
m133-b3	1.436	engine	0.886
torso2*	0.991	thermomech_dK	1.071
fidap011	0.761		

Ratio of our code to icc-compiled code.
Small is better.
(*) Matrices for which stencil matters more.

38

Code Quality

(Unfolding – sequential, with arith. opt., distinct value pool)

email-EuAll	0.830	cfd2	0.953
cit-HepPh	0.895	m14b	0.950
soc-Epinions1	0.857	s3dkt3m2	0.909
soc-sign	1.211	conf6_0-8x8-20	0.854
web-NotreDame	0.921	ship_003	1.021
webbase-1M	1.388	cage12	0.960
e40r5000	1.105	debr	0.921
fidapm11	0.803	mc2depi	1.444
fidapm37	0.779	s3dkq4m2	0.916
m133-b3	1.205	engine	1.995
torso2	1.340	thermomech_dK	0.987
fidap011	0.872		

The matrices for which we are worse are also those for which unfolding matters.

We need to implement Common Subexpression Elimination.

39

Low-Level Optimizations

- Next: Supporting case for programmer-guided optimizations

40

Low-Level Optimizations

```
d6:  f2 0f 59 6a 58                   mulsd   88(%rdx), %xmm5
db:  f2 0f 59 72 60                   mulsd   96(%rdx), %xmm6
e0:  f2 0f 59 7a 68                   mulsd   104(%rdx), %xmm7
e5:  f2 44 0f 59 42 70                mulsd   112(%rdx), %xmm8
eb:  f2 44 0f 59 4a 78                mulsd   120(%rdx), %xmm9
f1:  f2 44 0f 59 92 80 00 00 00       mulsd   128(%rdx), %xmm10
fa:  f2 44 0f 59 9a 88 00 00 00       mulsd   136(%rdx), %xmm11
```

xmm8-15 require 1 more byte

Mem offset >= 128 requires 3 more bytes

```
fb:  f2 0f 59 42 68                   mulsd   104(%rdx), %xmm0
100: f2 0f 59 4a 70                   mulsd   112(%rdx), %xmm1
105: 48 8d 52 78                      leaq    120(%rdx), %rdx
109: f2 0f 59 12                      mulsd   (%rdx), %xmm2
10d: f2 0f 59 5a 08                   mulsd   8(%rdx), %xmm3
112: f2 0f 59 62 10                   mulsd   16(%rdx), %xmm4
```

41

Low-Level Optimizations

- Restricting the registers to xmm0-7
  - Forces us to accumulate partial results more often
  - 3% improvement on the performance of unfolding
- Restricting the memory offset to [0-128)
  - 14% improvement on the performance of unfolding

42

Part II – Take Away

- Rapid code generation is possible
- But painful
- Low-level optimizations needed

PART III – Which specializer?

- Different machines, different performances
- Goal: an auto-tuned SpMV library
  - Train on a particular machine using many matrices
  - Given a matrix at runtime, quickly predict the best method and generate it

Auto-tuning – Initial Attempts

- Features (all of them normalized)
  - N, NZ,
  - # of stencils with popularity 1 and > 1, sum of stencil lengths
  - # of row nz's, sum of row nz lengths
  - # of distinct vals, average nz per row
  - # of 4x4 patterns with < 4 nz's, # of 4x4 patterns with >= 4 nz's, sum of 4x4 pattern lengths
  - # of 5x5 patterns with < 4 nz's, # of 5x5 patterns with >= 4 nz's, sum of 4x4 pattern lengths
- Target: Name of a specialization method

45

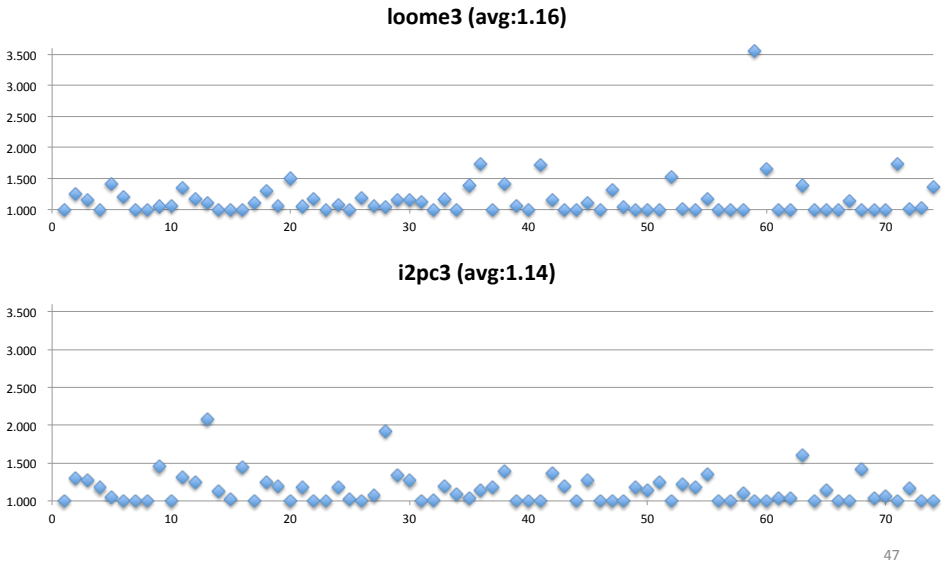
Auto-tuning – Initial Attempts

- Multiclass classification using Support Vector Machine (SVM)
- Matrix set of 74
- Cross-one-out approach
  - Train features and runtimes for 73 matrices
  - Predict the remaining matrix

46

Autotuning's pick vs. actual winner

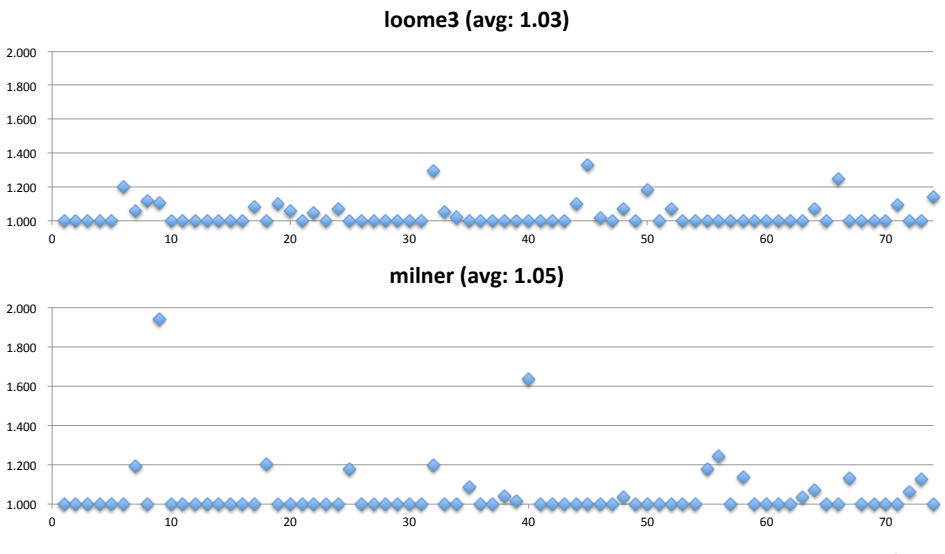
(icc-compiled code)



47

Autotuning's pick vs. actual winner

(our compiler's code)



48

PART III – Take Away

- Promising results for using autotuning to predict the best code generation method for a particular matrix and machine
- Much better results using the purpose-built code generator due to much less complexity in the way code is generated

49

Summary

- Can we speed up SpMV by code generation?
- Quickly generating code
- Code quality, certain low-level optimizations
  - Generated code is not a familiar domain for compilers
- Encouraging results for autotuning

- Shonan Challenge?
  - Loop unfolding + arithmetic optimizations + CSE

50