Combining Staged Programming and Empirical Optimization

Barış Aktemur

Özyeğin University, Istanbul

baris.aktemur@ozyegin.edu.tr

Motivation

- Main motivation behind staging is to speed up programs
- Long running programs are a better target for staging
- Apply staging to HPC (where speed matters a lot) that uses large data
  - Sparse matrix-vector multiplication

- Joint work with Sam Kamin and his research group at UIUC, and Asim Yildiz at OzU
  - Thanks to TUBITAK and NSF for their support

Sparse matrix-vector multiplication

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

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

Non-zero elements of the matrix

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

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

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

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

Specialized Multiplication

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

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}

Condensed Sparse Row (**CSR**)

```
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];
}
```

Specialized 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}

```
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];
}
```

Specialized 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}

```
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];
}
```

Specialized Multiplication

- Banded, sparse, square matrices

Matrix name	Field	Size	No. of NZ elements	Ratio of Duration
utm300	Nuclear physics	300	1344	**0.800**
fidap018	Finite elements	5773	69231	**0.826**
fidap012	Finite elements	3973	79017	**0.769**
memplus	Circuit design	17758	99147	**0.952**
fidap035	Finite elements	19716	217972	**2.631**
af23560	Fluid dynamics	23560	460598	**1.754**
e40r5000	Fluid dynamics	17281	553562	**2.500**

speed up

slow down

Matrices from http://math.nist.gov/MatrixMarket/

$$\frac{\text{Time of specialized code}}{\text{Time of general CSR code}}$$

Problem

- Big matrix => Very long code
  - Compilers omit optimizing long code
  - Poor register allocation
  - Bad instruction cache usage
- Data-cache usage
  - Non-uniform access to vector elements, because the matrix is sparse
  - Cache must be utilized effectively

Solution

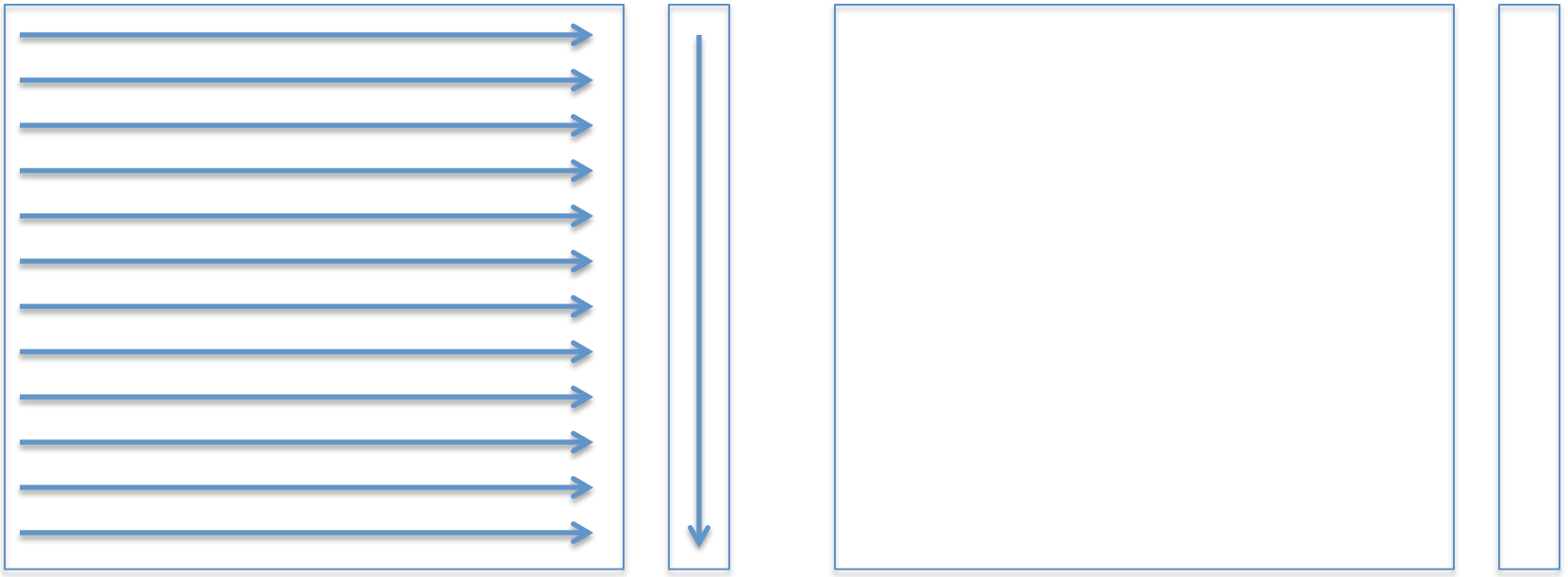
1. Generate code that is short enough to fit the I-cache, and to be optimized by compilers.

2. Access the matrix/vector in an order that uses the D-cache effectively.

- Sure, but how?

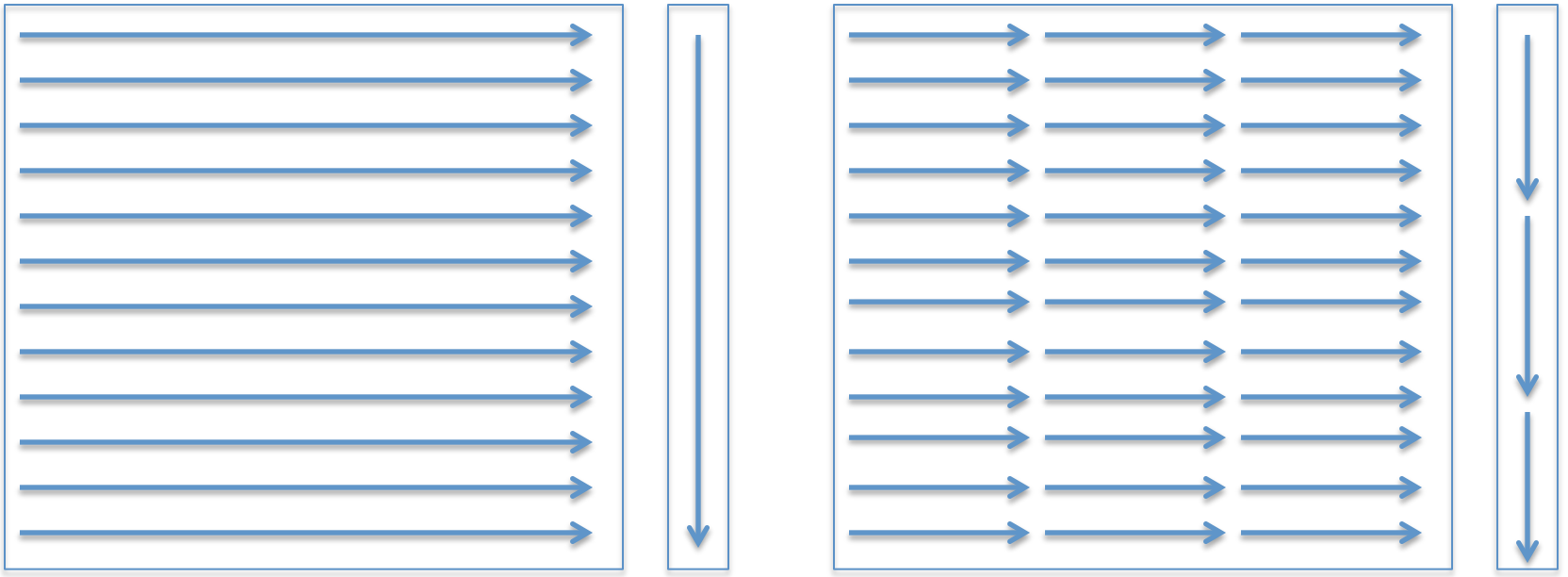
MxN Unrolling [OSKI]

- Instead of row-major order, access the matrix in MxN blocks



MxN Unrolling [OSKI]

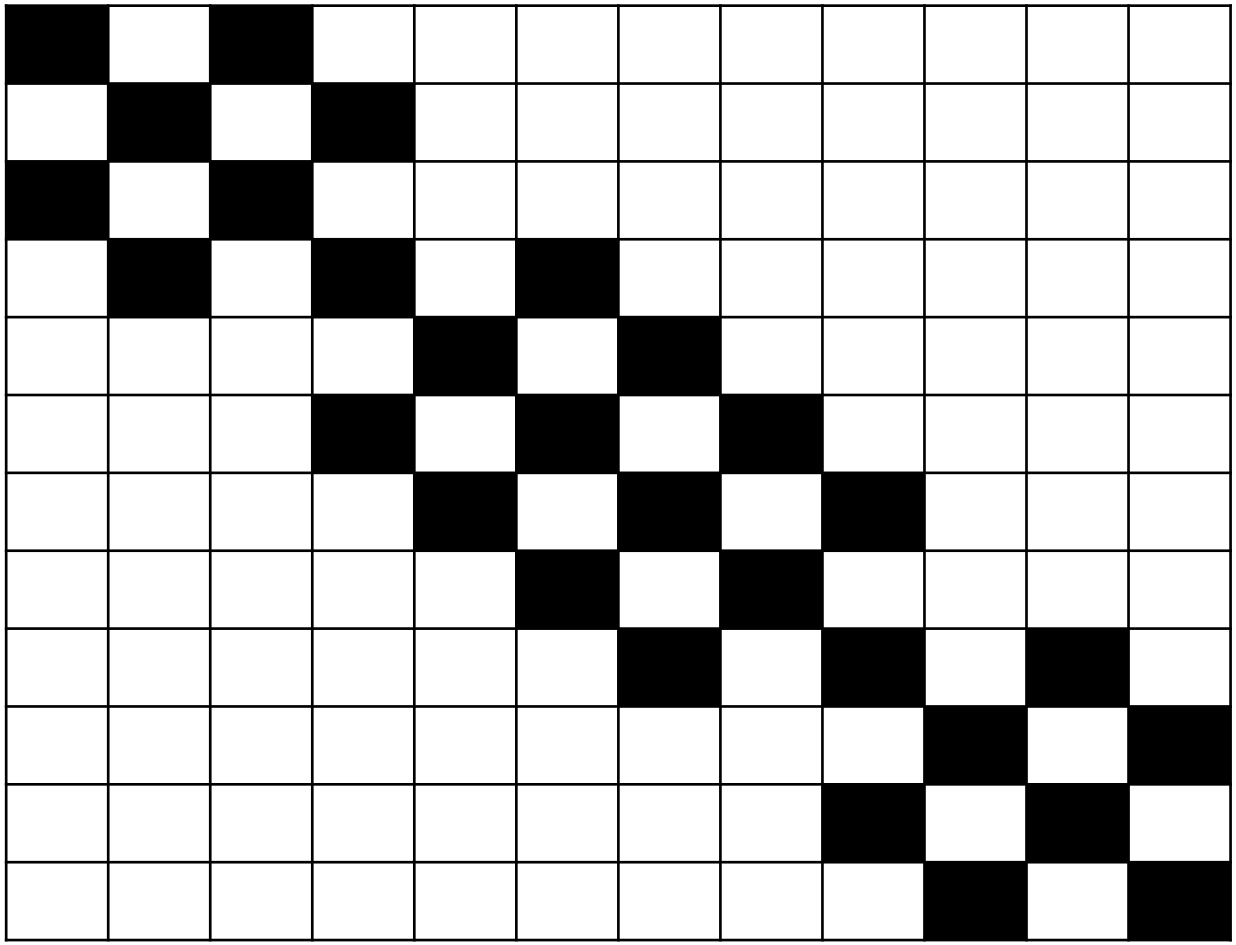
- Instead of row-major order, access the matrix in MxN blocks



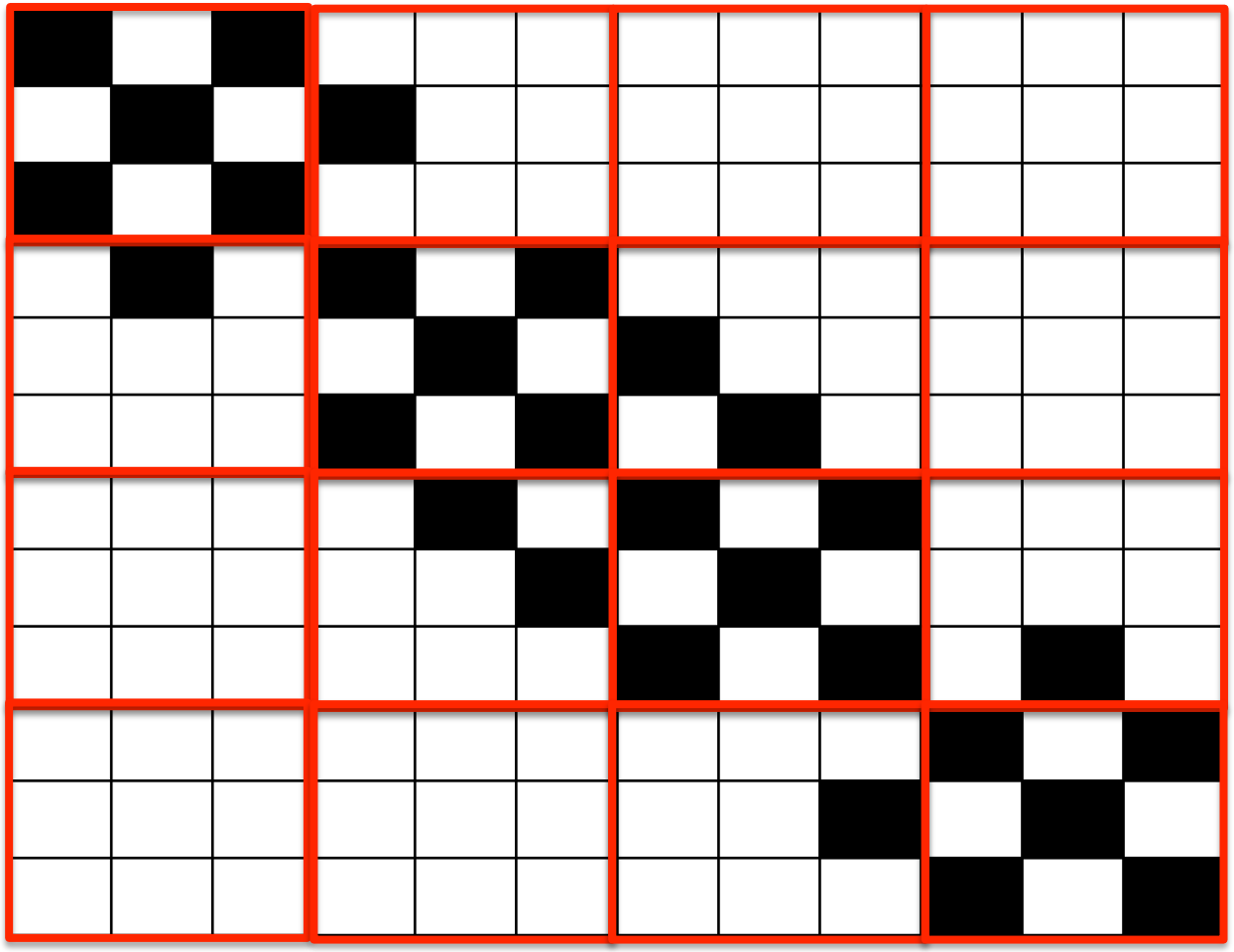
```
void mult(int bn, int *b_rows, int *b_cols, double *b_vals,
          double *v, double *w) {
    double *y = w;
    int *b_row_start = b_rows;
    int *b_col_idx = b_cols;
    double *b_value = b_vals;
    for (int i = 0; i < bn; i++,y += 2) {
        double d0 = y[0];
        double d1 = y[1];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

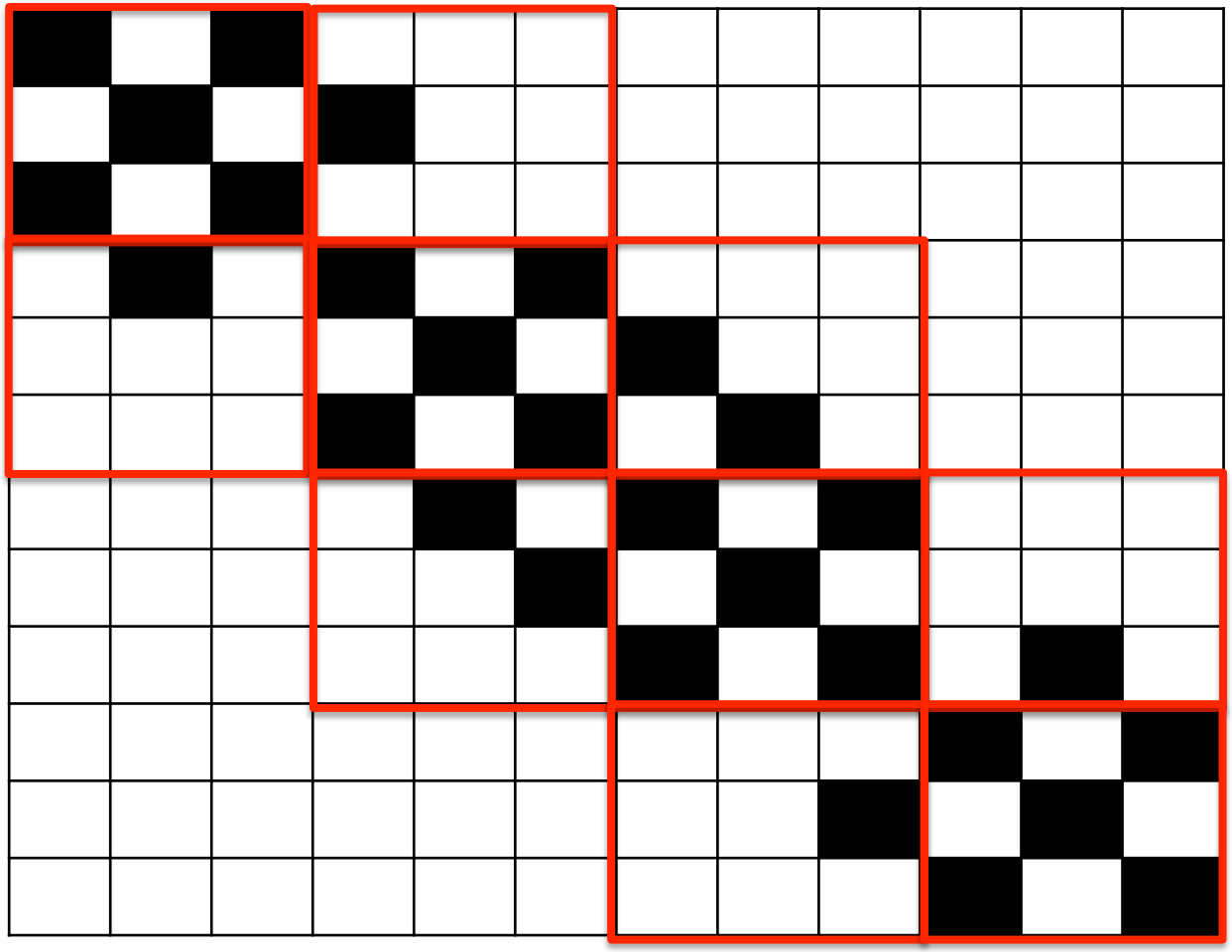
MxN Blocking



MxN Blocking



Generate specialized code for every non-empty block

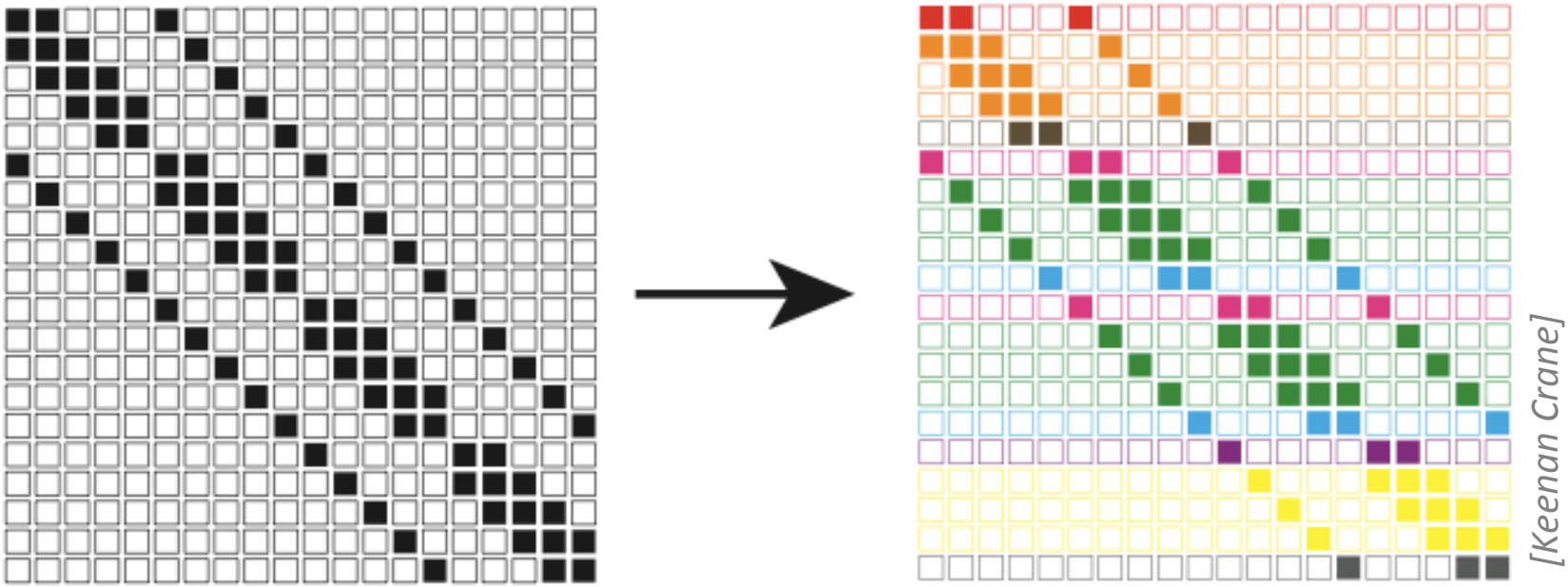


```
void mult(double *v, double *w) {
    block_0_0(v,w);
    block_0_3(v,w);
    block_3_0(v,w);
    block_3_3(v,w);
}
void block_0_0(double *v, double *w) {
    w[0] = w[0] + 1 * v[1] + 2 * v[2];
    w[1] = w[1] + 3 * v[2];
    w[2] = w[2] + 5 * v[0];
}
void block_0_3(double v[], double w[]) {
    w[1] = w[1] + 4 * v[3];
    w[2] = w[2] + 6 * v[3] + 7 * v[4];
}
void block_3_0(double v[], double w[]) {
    w[3] = w[3] + 8 * v[0] + 9 * v[2];
    w[4] = w[4] + 10 * v[1];
}
void block_3_3(double v[], double w[]) {
    w[4] = w[4] + 11 * v[3];
}
```

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

Stencil-based specialization

- Use the same code for rows that have the same stencil.



[Keenan Crane]

- Stencil for row i: $\{j\text{-}i \mid M_{i,j} \neq 0\}$
  - Row 0 = (■,0,■,■,0,0,0,■,0,0,0) → {0,2,3,7}
  - Row 1 = (■,0,■,■,0,0,0,■,0,0,0) → {-1,1,2,6}
  - Row 3 = (0,0,0,■,0,■,■,0,0,0,■) → {0,2,3,7}

```
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.

As many for-loops as there are stencils.

Problems if there are too many stencils.

Banded Stencil

- Poor performance if high number of stencils
- Many matrices have a banded-like structure
  - Elements usually around the diagonal
  - Some scattered to other places
  - Superficially increases the stencil number
- Approach: Do stencil-based staging for ***M*** elements around the diagonal, fully unroll the rest

CSR by NZ

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

```
void mult(int *rows, int *cols, double *vals,
          double *v, double *w) {
    int i, row;
    int a = 0, b = 0;
    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 < 5; 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;
    }
}
```

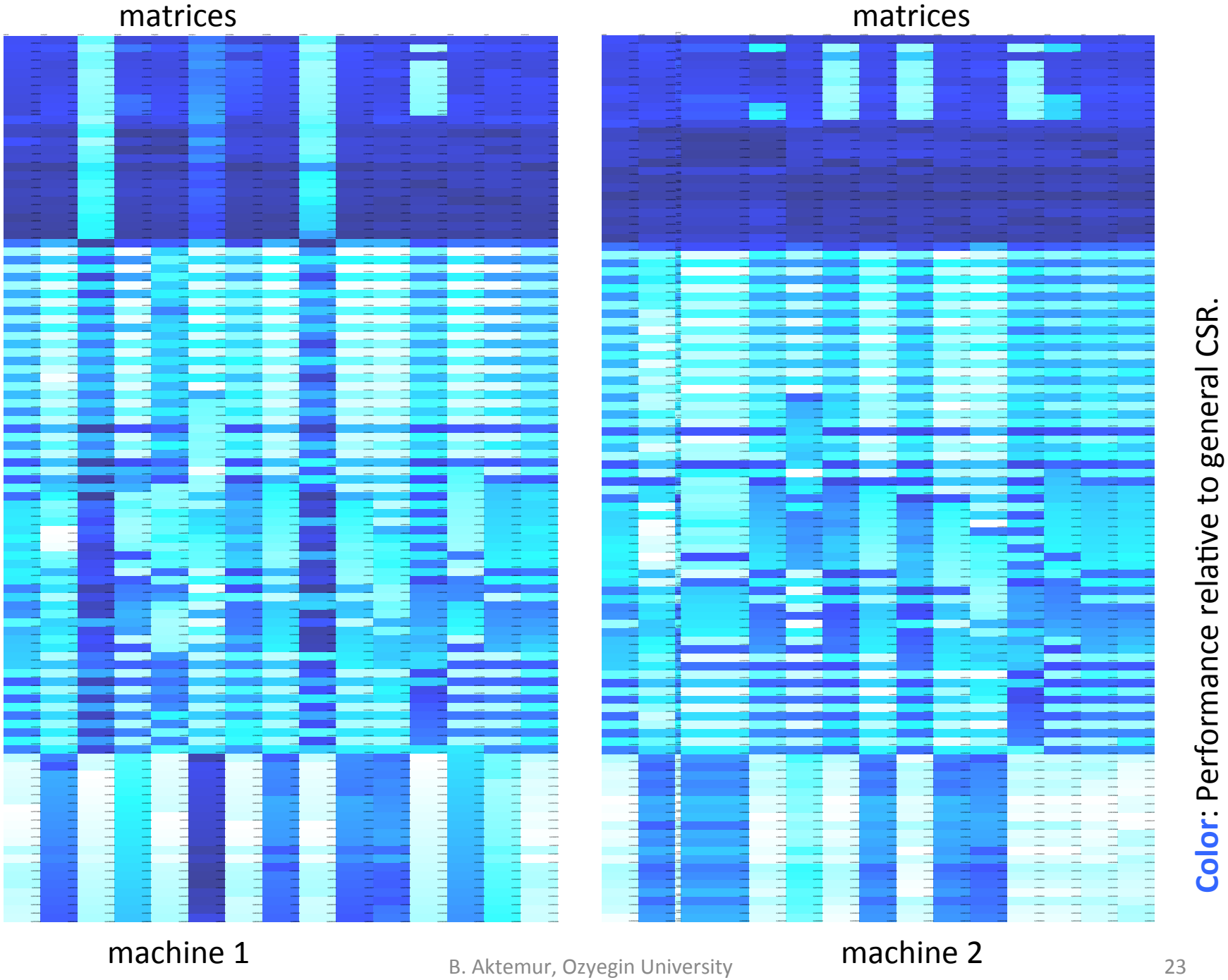
4 rows with 2 elements

5 rows with 3 elements

Many choices

- MxN Unrolling
- MxN Blocking
- M-Banded stencil
- CSR by NZ
- ~~Stencil~~ *(special case of M-banded)*
- ~~General CSR~~ *(special case of MxN unrolling)*
- ~~Full unrolling~~ *(special case of MxN blocking)*

- Several variations by using different parameters
- Maybe hybrid specializers?


matrices
matrices
specializers
machine 1
machine 2
Color: Performance relative to general CSR.

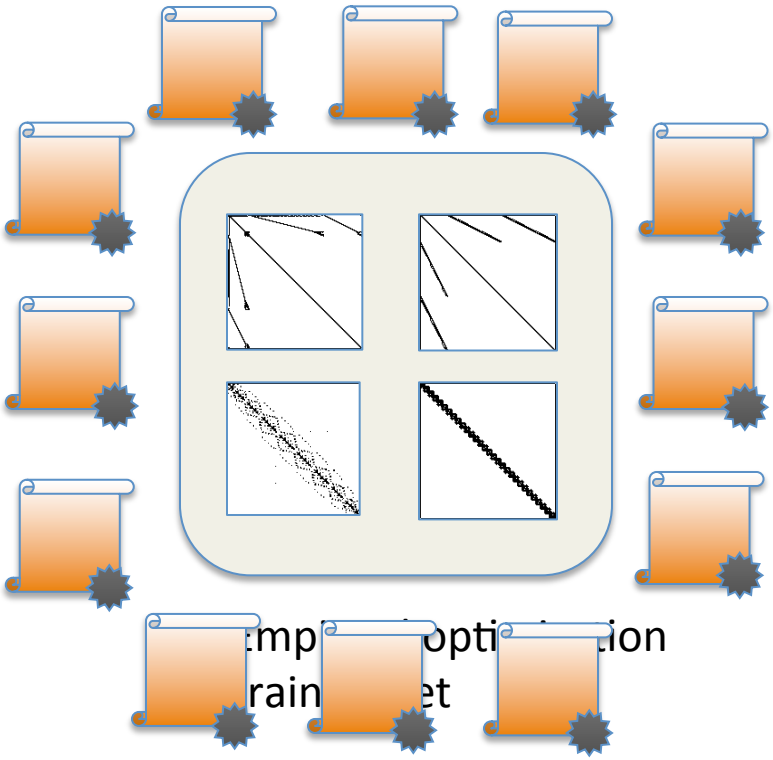
Which specializer should I use?

- It is difficult to know the performance of the output of a certain specializer before-hand.
  - Modern CPUs are complicated
  - Performance portability is not guaranteed
- Varying performance based on matrix contents and computer architecture (in particular, cache size and structure)
- It is unrealistic to generate all the possibilities and benchmark

Empirical optimization / auto-tuning

- Run several versions of an algorithm during install-time
  - Learning phase
  - Which version performs the best for which kind of data on this particular machine?
- Detect the properties of the run-time data, execute the program version known to run fastest for similar data
- FFTW, Spiral, Sparsity (aka OSKI), ATLAS

Design of a sparse matrix-vector multiplication library

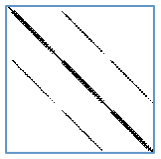


Performance of each generated code for each matrix

What to store here exactly?

install-time

run-time



Predictor

"Voting" to improve the chance of success

What data to collect during experiments?

Matrix	Spec.	Time
add32	Blocking[2,3]	…
add32	Blocking[3,4]	…
add32	Blocking[1,2]	…
add32	Stencil	…
…	…	…
fidap037	Stencil	…
fidap037	Banded stencil 200	…
fidap037	CSRbyNZ	…
…	…	…

Need matrix characteristics to associate to timings

What data to collect during experiments?

Matrix	Spec.	Time	n	nz
add32	Blocking[2,3]	…	…	…
add32	Blocking[3,4]	…	…	…
add32	Blocking[1,2]	…	…	…
add32	Stencil	…	…	…
…	…	…	…	…
fidap037	Stencil	…	…	…
fidap037	Banded stencil 200	…	…	…
fidap037	CSRbyNZ	…	…	…
…	…	…	…	…

Need matrix characteristics to associate to timings

Did not really work. Decided that something else must be tried.

Cache behavior.

What data to collect during experiments?

Matrix	Spec.	Time	Ir	I1mr	I2mr	Dr	D1mr	D2mr	Dw	D1mw	D2mw
add32	Bl										
add32											
add32											
add32											
…											
fidap0											
fidap0											
fidap037	CSRbyNZ	…	…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…	…	…	…

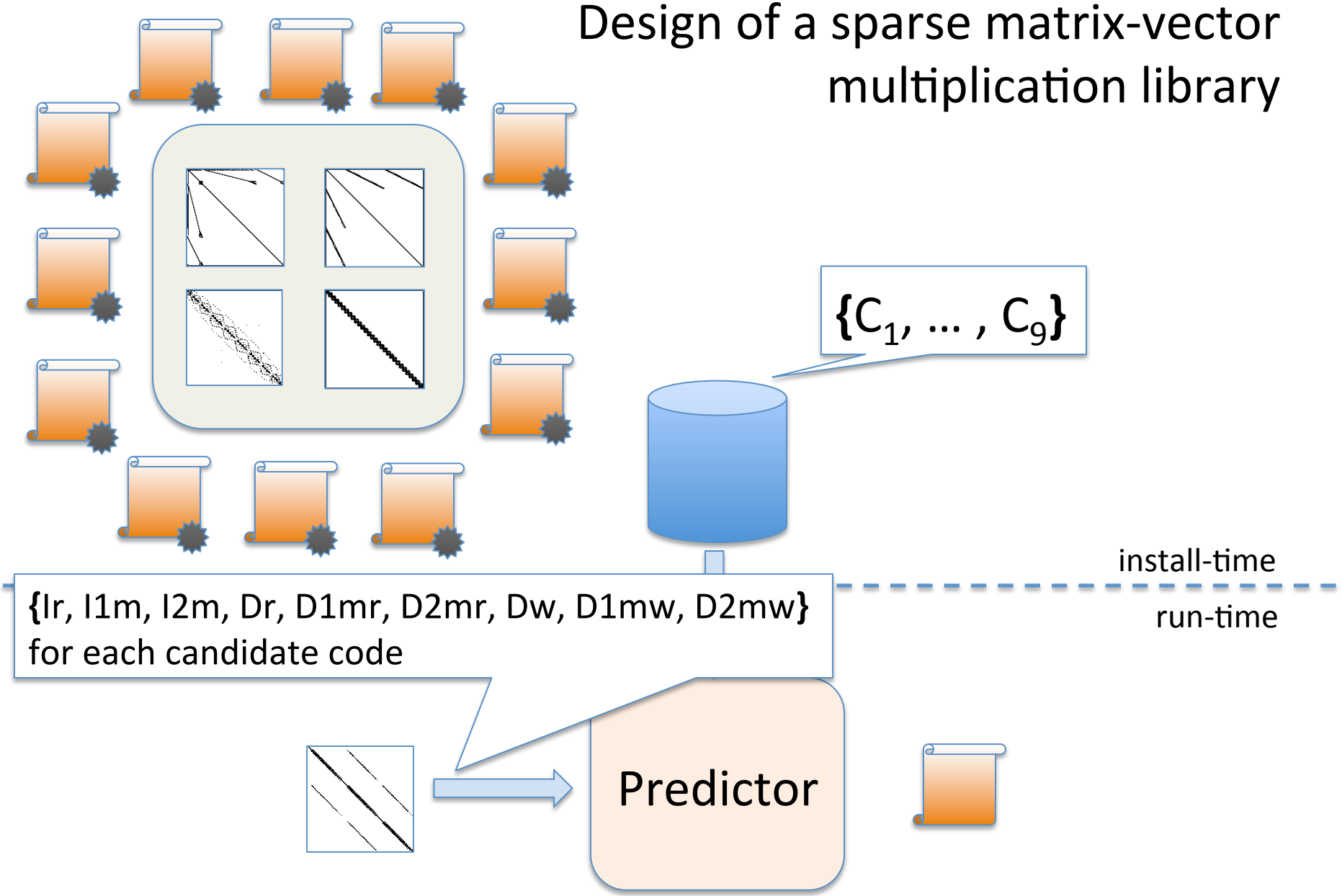
ins | L1 | L2 I | Nu | L1 da | L2 da | Num | L1 da | L2 data cache write misses

Learn
{C_1, … , C_9} such that

$$\text{Time} = C_1*\text{Ir} + C_2*\text{I1mr} + C_3*\text{I2mr} + C_4*\text{Dr} + C_5*\text{D1mr} + C_6*\text{D2mr} + C_7*\text{Dw} + C_8*\text{D1mw} + C_9*\text{D2mw}$$

Used **valgrind** to collect information

Design of a sparse matrix-vector multiplication library

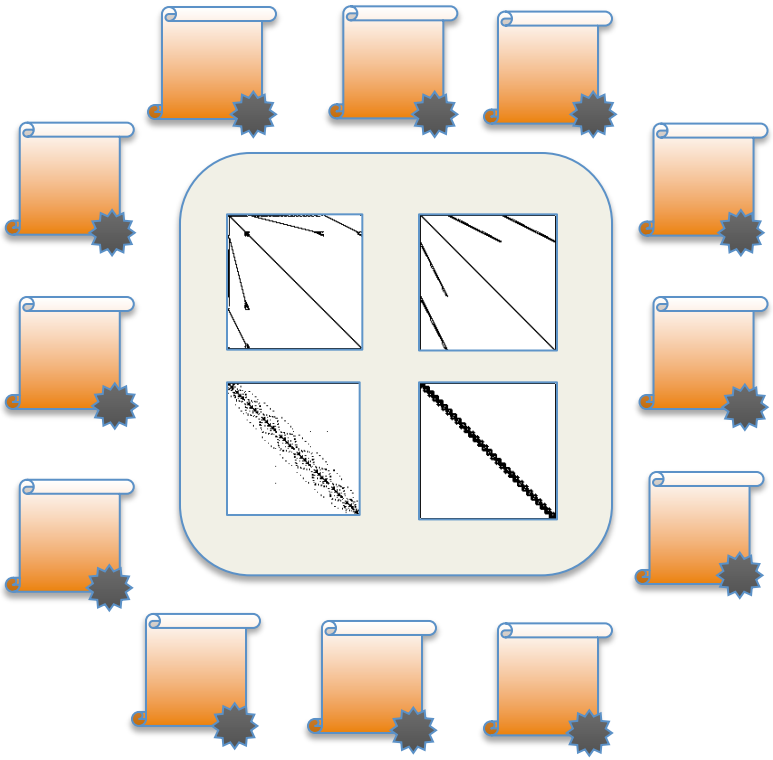


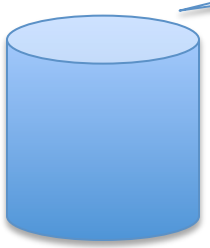
All times are relative to the corresponding general CSR time.

Matrix	Intel Xeon		Intel Core 2 Duo	
	Best	Predicted	Best	Predicted
add32	CSRbyNZ 61.3%	Blocking[∞] 65.9%	Stencil 42.2%	Stencil 42.2%
cavity02	Stencil 55.5%	Banded[20] 84.3%	Unroll[3,1] 78.7%	Unroll[3,1] 78.7%
cavity23	Banded[500] 70.6%	Banded[500] 70.6%	Stencil 75.2%	Stencil 75.2%
fidap005	Blocking[∞] 41.3%	Blocking[∞] 41.3%	Blocking[∞] 21.3%	Blocking[∞] 21.3%
fidap037	CSRbyNZ 80.1%	CSRbyNZ 80.1%	Banded[10] 89.4%	Banded[10] 89.4%
mhd3200a	Stencil 55.9%	Stencil 55.9%	Stencil 62.4%	Stencil 62.4%
memplus	CSRbyNZ 77.8%	Blocking[∞] 80.8%	CSRbyNZ 61.9%	Blocking[∞] 61.9%
nnc666	Blocking[∞] 53.1%	Blocking[∞] 53.1%	Stencil 49.2%	Stencil 49.2%

Matrices from http://math.nist.gov/MatrixMarket/

Design of a sparse matrix-vector multiplication library

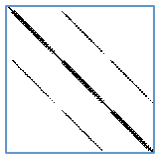




$\{C_1, \ldots, C_9\}$

OK, but how are we going to find these values **without** generating the specialized programs?

{Ir, I1m, I2m, Dr, D1mr, D2mr, Dw, D1mw, D2mw} for each candidate code



Predictor

Custom Cache Simulator for Each Specializer

```
Ir  I1mr  ILmr  Dr  D1mr  DLmr  Dw  D1mw  DLmw
 .     .     .   .     .     .   .     .     .  int vals[] = {1, 2, 3, 4};
 .     .     .   .     .     .   .     .     .  int rows[] = {0, 2, 3, 4};
 .     .     .   .     .     .   .     .     .  int cols[] = {1, 2, 0, 1};
 .     .     .   .     .     .   .     .     .
 .     .     .   .     .     .   .     .     .  double *w, *v;
 .     .     .   .     .     .   .     .     .
 .     .     .   .     .     .   .     .     .
 3     0     0   0     0     0   3     0     0  void mult() {
 .     .     .   .     .     .   .     .     .      int k,j;
 .     .     .   .     .     .   .     .     .      double ww;
 .     .     .   .     .     .   .     .     .
12     0     0   3     0     0   0     0     0      for (j=0; j < 4; j++) {
 4
34                                                 j+1]; k++)
16                                                 vals[k];
 4
 .
 4
 .
 4
 3
 4
 .
 1
 .
 .     .     .   .     .     .   .     .     .      return 0;
 3     0     0   2     0     0   0     0     0  }
```

Sort of reverse engineering under valgrind's guidance

Regular cache simulator:
executable code * matrix -> result

Our cache simulator:
matrix -> result

Hmm…

Custom Cache Simulator for Each Specializer

- We don't know the to-be-generated program, but we know the generator
- Order of data accesses is straightforward to infer
  - Look at only a part of matrix data to estimate misses and hits
- Inferring access to I-cache is challenging
  - Compilers may reorder instructions
  - Optimizations may non-uniformly reduce instructions
  - Highly dependent on the chipset

Summary

- Large-data increases the search space of candidate codes
- Using empirical optimization to select the best staged version
- Challenges
  - What criteria to use to predict the performance of a program?
  - If the answer is cache behavior, how to accurately estimate this for non-existing code?
  - Not mentioned today: cost of generation/data analysis