How to Write Fast Numerical Code

Spring 2011

Lecture 15

Instructor: Markus Püschel

TA: Georg Ofenbeck

ETH

Eidgenössische Technische Hochschule Zürich

Swiss Federal Institute of Technology Zurich

Reuse Again

- ***Reuse of an algorithm:***

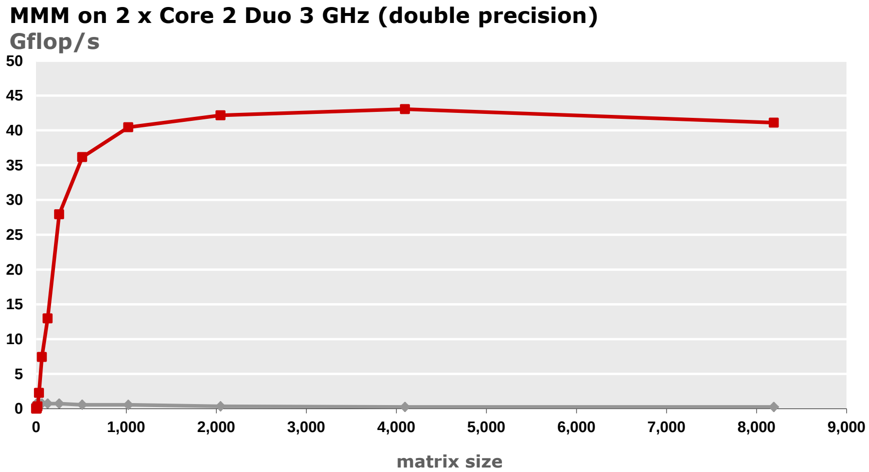
$$\frac{\textit{Number of operations}}{\textit{Size of input + size of output data}}$$

Minimal number of Memory accesses

- **Examples:**
  - Matrix multiplication C = AB + C $\quad \frac{2n^3}{3n^2} = \frac{2}{3}n = O(n)$
  - Discrete Fourier transform $\quad \approx \frac{5n\log_2(n)}{2n} = \frac{5}{2}\log_2(n) = O(\log(n))$
  - Adding two vectors x = x+y $\quad \frac{n}{2n} = \frac{1}{2} = O(1)$

Effects

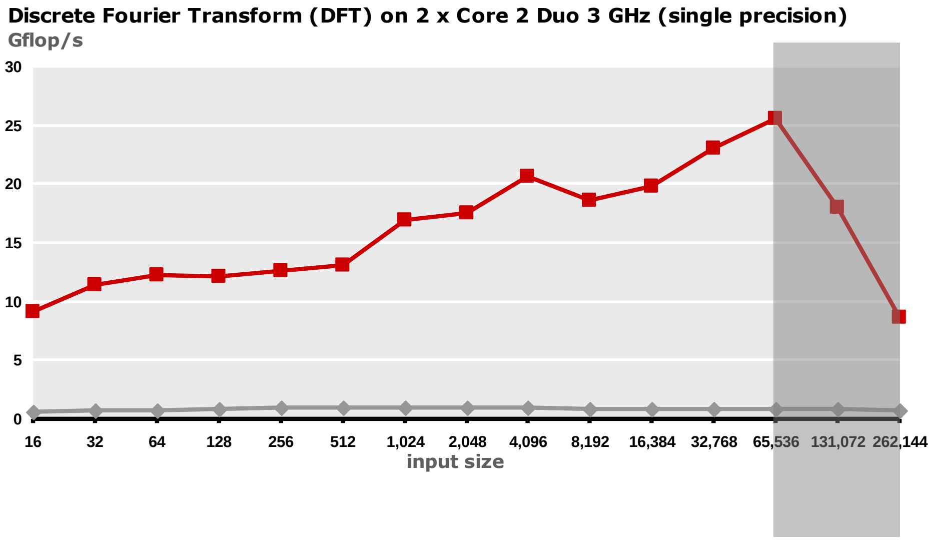
MMM: O(n) reuse



Performance maintained even when data does not fit into caches

Drop will happen once data does not fit into main memory

FFT: O(log(n)) reuse



Performance drop when data does not fit into largest cache

Outside cache: Runtime only determined by memory accesses (memory bound)

Memory Bound Computation

- Typically: Computations with O(1) reuse
- Performance bound based on data traffic may be tighter than performance bound obtained by op count

Example

- **Vector addition: z = x + y on Core 2**

```
void vectorsum(double *x, double *y, double *z, int n)
{
  int i;

  for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];
}
```

Reuse: 1/3

- **Core 2:**
  - Peak performance (no SSE):
  - Throughput L1 cache:
  - Throughput L2 cache:
  - Throughput Main memory:

Resulting bounds

Example

- **Vector addition: z = x + y on Core 2**

```
void vectorsum(double *x, double *y, double *z, int n)
{
  int i;

  for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];
}
```

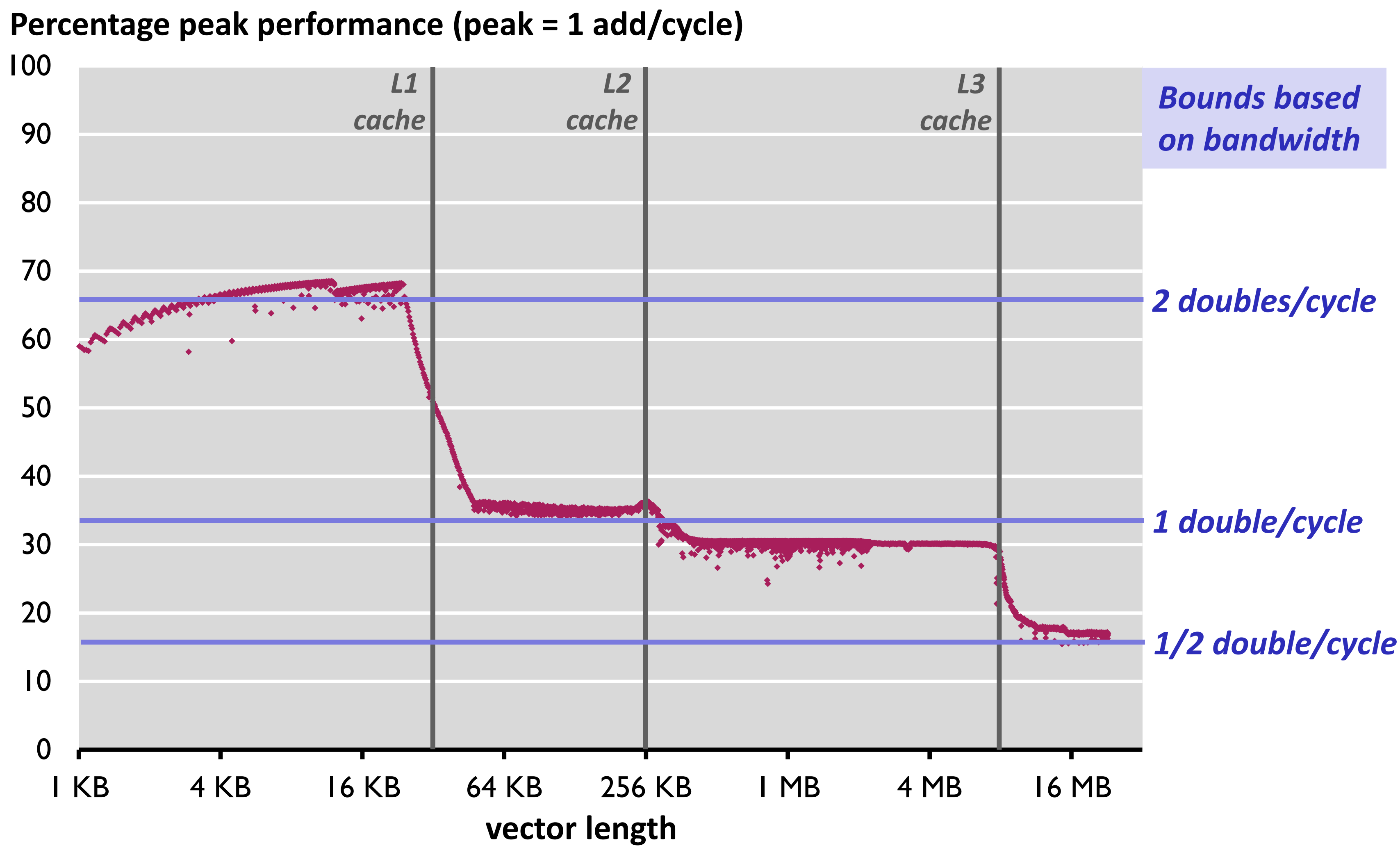
Reuse: 1/3

- **Core 2:**

		Resulting bounds
Peak performance (no SSE):	1 add/cycle	n cycles
Throughput L1 cache:	2 doubles/cycle	3/2 n cycles
Throughput L2 cache:	1 doubles/cycle	3n cycles
Throughput Main memory:	¼ doubles/cycle	12 n cycles

Memory-Bound Computation

z = x + y on Core i7 (one core, no SSE), icc 12.0 /O2 /fp:fast /Qipo



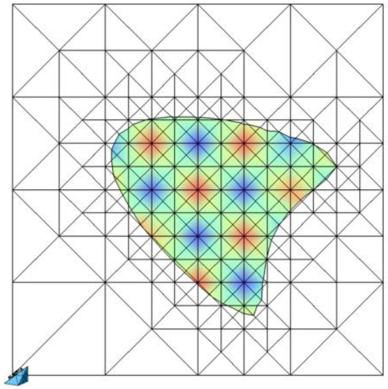
Today

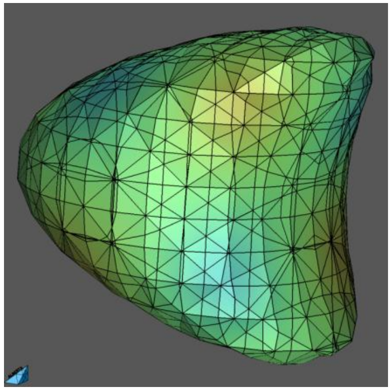
- **Sparse matrix-vector multiplication (MVM)**
- **Sparsity/Bebop**

- **References:**
  - Eun-Jin Im, Katherine A. Yelick, Richard Vuduc. *SPARSITY: An Optimization Framework for Sparse Matrix Kernels, Int'l Journal of High Performance Comp. App.*, 18(1), pp. 135-158, 2004
  - Vuduc, R.; Demmel, J.W.; Yelick, K.A.; Kamil, S.; Nishtala, R.; Lee, B.; *Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply,* pp. 26, Supercomputing, 2002
  - Sparsity/Bebop website

Sparse Linear Algebra

- **Very different characteristics from dense linear algebra (LAPACK etc.)**
- **Applications:**
  - finite element methods
  - PDE solving
  - physical/chemical simulation
    (e.g., fluid dynamics)
  - linear programming
  - scheduling
  - signal processing (e.g., filters)
  - …
- **Core building block: Sparse MVM**

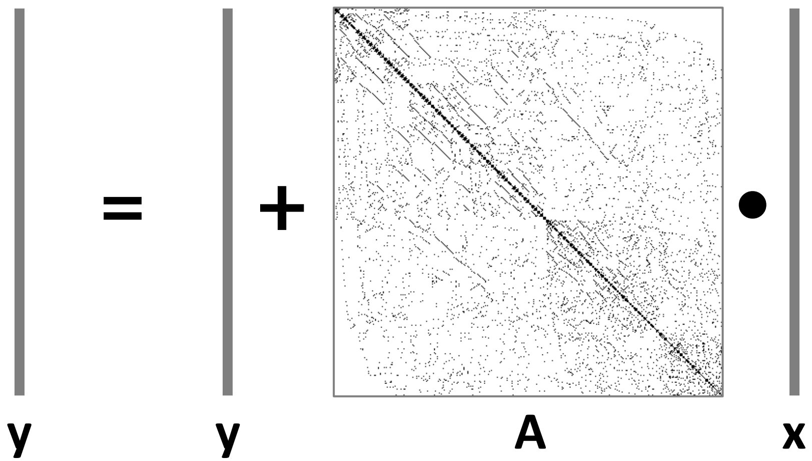




Graphics: http://aam.mathematik.uni-freiburg.de/IAM/homepages/claus/projects/unfitted-meshes_en.html

Sparse MVM (SMVM)

- y = y + Ax, A sparse but known



- Typically executed many times for fixed A
- What is reused?
- Reuse dense versus sparse MVM?

Storage of Sparse Matrices

- **Standard storage is obviously inefficient**
  - Many zeros are stored
  - As a consequence, reuse is decreased
- **Several sparse storage formats are available**
- **Most popular: Compressed sparse row (CSR) format**
  - blackboard

CSR

- **Assumptions:**
  - A is m x n
  - K nonzero entries

A as matrix

b	c		c
	a		
		b	b
		c	

A in CSR:

`values`	b	c	c	a	b	b	c	*length K*
`col_idx`	0	1	3	1	2	3	2	*length K*
`row_start`	0	3	4	6	7			*length m+1*

- **Storage: Θ(max(K, m)), typically Θ(K)**

Sparse MVM Using CSR

y = y + Ax

```
void smvm(int m, const double* value, const int* col_idx,
          const int* row_start, const double* x, double* y)
{
  int i, j;
  double d;

  /* loop over rows */
  for (i = 0; i < m; i++) {
    d = y[i]; /* scalar replacement since reused */

    /* loop over non-zero elements in row i */
    for (j = row_start[i]; j < row_start[i+1]; j++, col_idx++, value++) {
      d += value[j] * x[col_idx[j]];
    }
    y[i] = d;
  }
}
```

CSR + sparse MVM: Advantages?

CSR

- **Advantages:**
  - Only nonzero values are stored
  - All arrays are accessed consecutively in MVM (spatial locality)
- **Disadvantages:**
  - x is not reused
  - Insertion costly

Impact of Matrix Sparsity on Performance

- **Adressing overhead (dense MVM vs. dense MVM in CSR):**
  - ~ 2x slower (Mflop/s, example only)
- **Irregular structure**
  - ~ 5x slower (Mflop/s, example only) for “random” sparse matrices
- **Fundamental difference between MVM and sparse MVM (SMVM):**
  - Sparse MVM is input ***dependent*** (sparsity pattern of A)
  - Changing the order of computation (blocking) requires changing the data structure (CSR)

Bebop/Sparsity: SMVM Optimizations

- *Idea:* **Register blocking**
- *Reason:* **Reuse x to reduce memory traffic**
- *Execution:* **Block SMVM y = y + Ax into micro MVMs**
  - Block size r x c becomes a parameter
  - Consequence: Change A from CSR to r x c block-CSR (BCSR)
- **BCSR: Blackboard**

BCSR (Blocks of Size r x c)

- **Assumptions:**
  - A is m x n
  - Block size r x c
  - $K_{r,c}$ nonzero blocks

A as matrix (r = c = 2)

b	c		c
	a		
		b	b
		c	

A in BCSR (r = c = 2):

`b_values`	b	c	0	c	0	0	c	0	b	b	c	0	*length $rcK_{r,c}$*

`b_col_idx`	0	2	2	*length $K_{r,c}$*
`b_row_start`	0	2	4	*length m/r+1*

- **Storage: $\Theta(rcK_{r,c})$, $rcK_{r,c} \geq K$**

Sparse MVM Using 2 x 2 BCSR

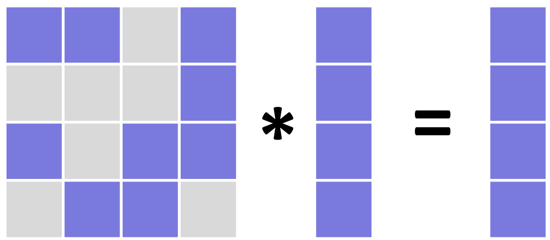
```
void smvm_2x2(int bm, const int *b_row_start, const int *b_col_idx,
              const double *b_value, const double *x, double *y)
{
  int i, j;
  double d0, d1, c0, c1;

  /* loop over block rows */
  for (i = 0; i < bm; i++, y += 2) {
    d0 = y[i];   /* scalar replacement */
    d1 = y[i+1];

    /* dense micro MVM */
    for (j = b_row_start[i]; j < b_row_start[i+1]; j++, b_col_idx++, b_value += 2*2) {
      c0 = x[b_col_idx[j]+0]; /* scalar replacement */
      c1 = x[b_col_idx[j]+1];
      d0 += b_value[0] * c0;
      d1 += b_value[2] * c0;
      d0 += b_value[1] * c1;
      d1 += b_value[3] * c1;
    }
    y[i]   = d0;
    y[i+1] = d1;
  }
}
```

BCSR

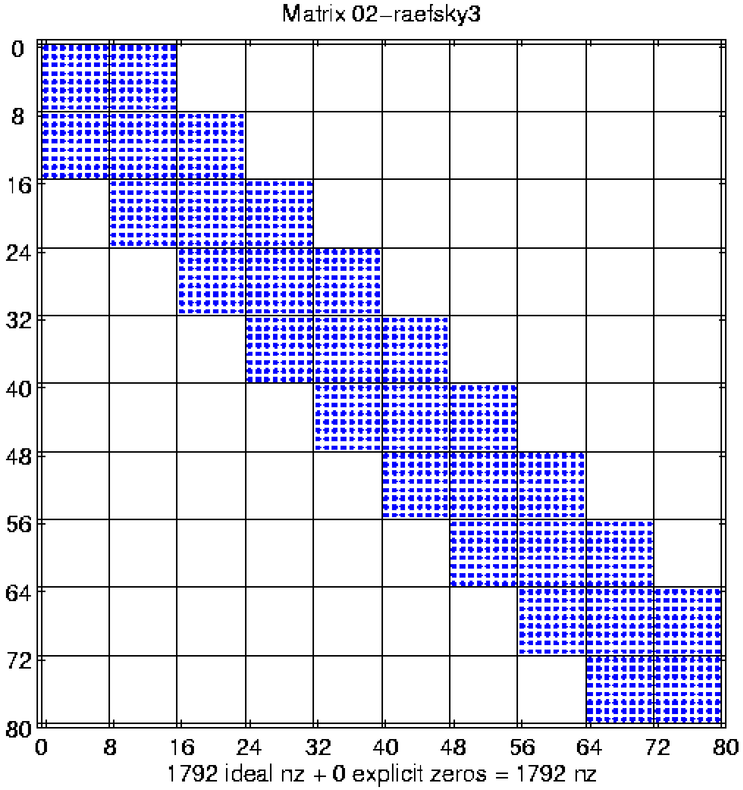
- **Advantages:**
  - Reuse of x and y (same as for dense MVM)
  - Reduces storage for indexes
- **Disadvantages:**
  - Storage for values of A increased (zeros added)
  - Computational overhead (also due to zeros)



- **Main factors (since memory bound):**
  - ***Plus:*** increased reuse on x + reduced index storage = reduced memory traffic
  - ***Minus:*** more zeros = increased memory traffic

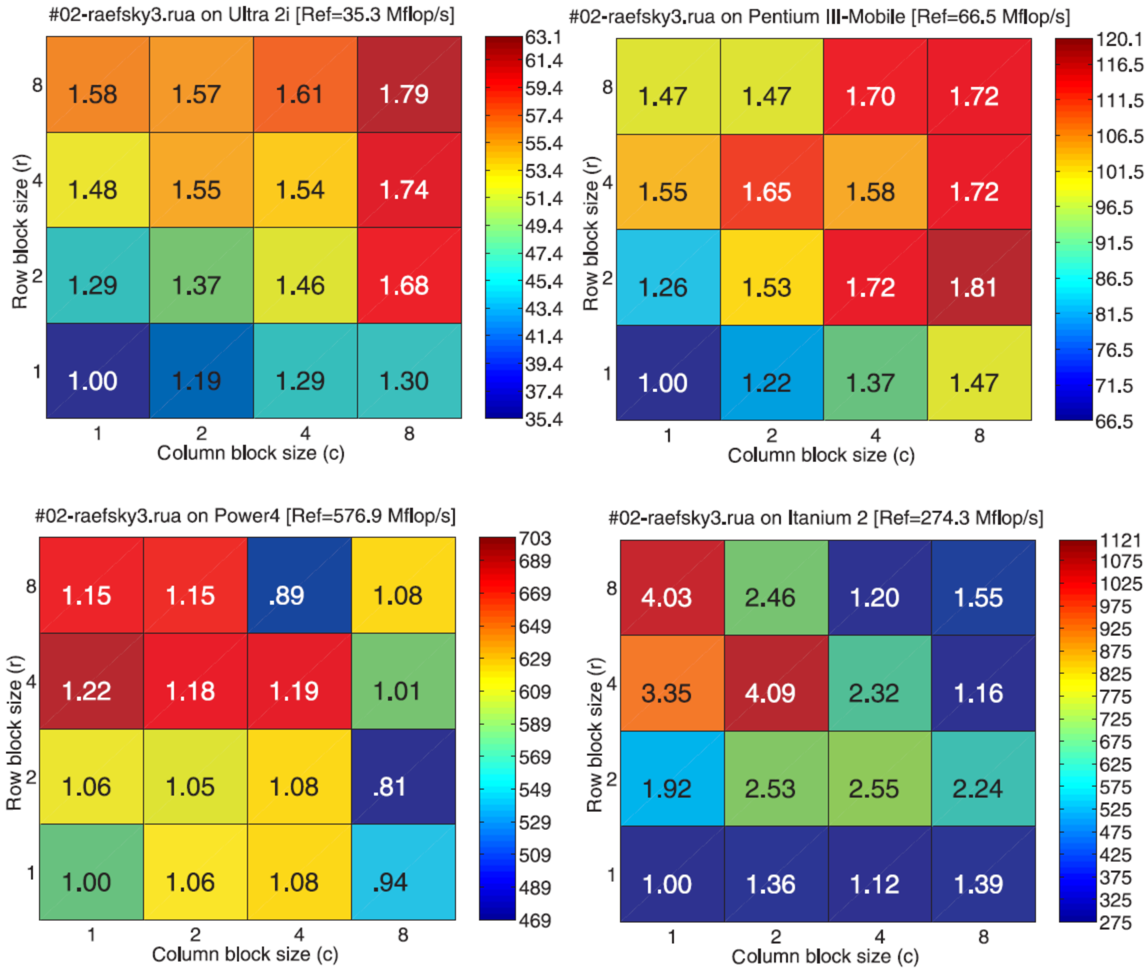
Which Block Size (r x c) is Optimal?

- *Example:* **about 20,000 x 20,000 matrix with perfect 8 x 8 block structure, 0.33% non-zero entries**
- *In this case:* **No overhead when blocked r x c, with r,c divides 8**



source: R. Vuduc, LLNL

Speed-up through r x c Blocking



- machine dependent
- hard to predict

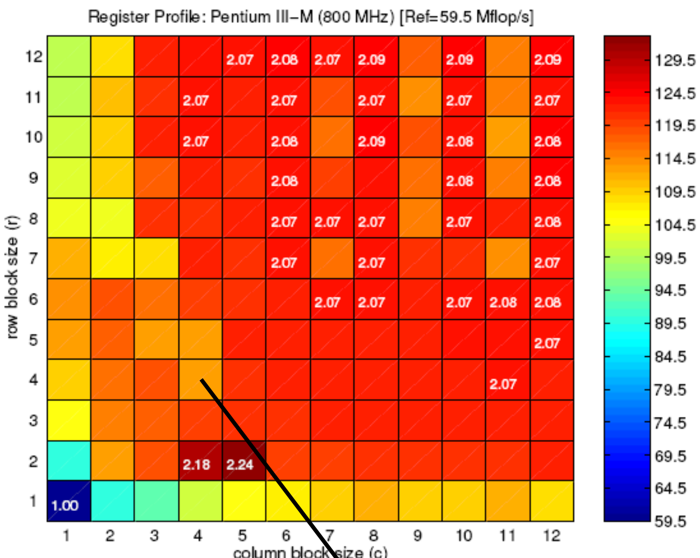
Source: *Eun-Jin Im, Katherine A. Yelick, Richard Vuduc. SPARSITY: An Optimization Framework for Sparse Matrix Kernels, Int'l Journal of High Performance Comp. App., 18(1), pp. 135-158, 2004*

How to Find the Best Blocking for given A?

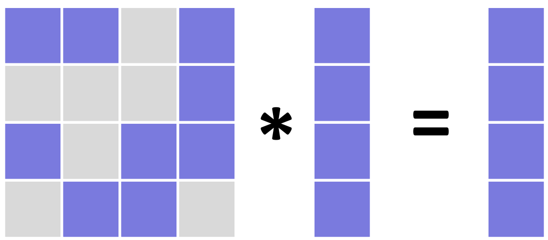
- **Best block size is hard to predict (see previous slide)**
- ***Solution 1:*** **Searching over all r x c within a range, e.g., 1 ≤ r,c ≤ 12**
  - Conversion of A in CSR to BCSR roughly as expensive as 10 SMVMs
  - Total cost: 1440 SMVMs
  - Too expensive
- ***Solution 2:*** **Model**
  - Estimate the gain through blocking
  - Estimate the loss through blocking
  - Pick best ratio

Model: Example

Gain by blocking (dense MVM)



Overhead (average) by blocking



16/9 = 1.77

1.4

1.4/1.77 = 0.79 (no gain)

Model: **Doing that for all r and c and picking best**

Model

- *Goal:* **find best r x c for y = y + Ax**
- *Gain* **through r x c blocking (estimation):**

$$G_{r,c} = \frac{\textit{dense MVM performance in r x c BCSR}}{\textit{dense MVM performance in CSR}}$$

  - dependent on machine, independent of sparse matrix
- *Overhead* **through r x c blocking (estimation)**
  - scan part of matrix A

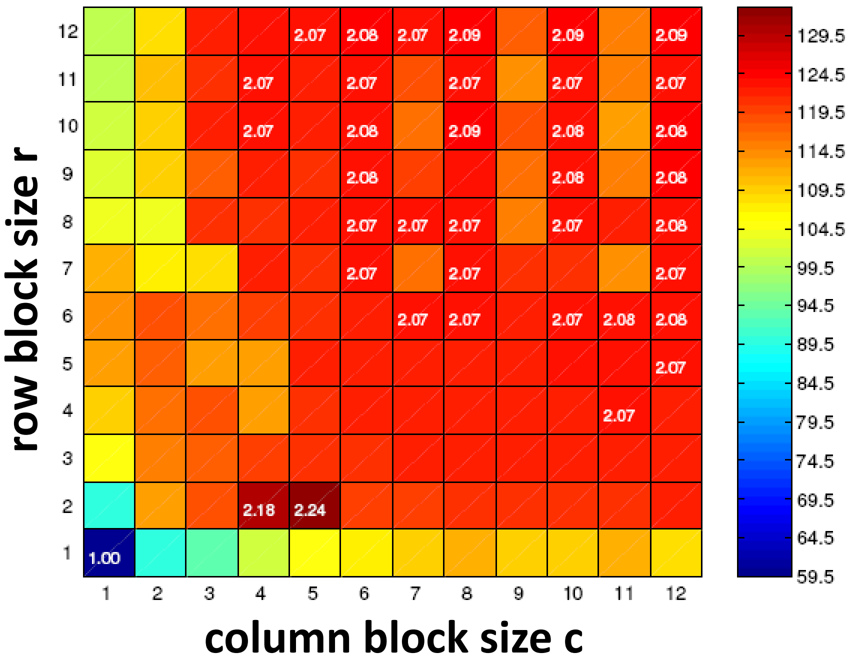
$$O_{r,c} = \frac{\textit{number of matrix values in r x c BCSR}}{\textit{number of matrix values in CSR}}$$

  - independent of machine, dependent on sparse matrix

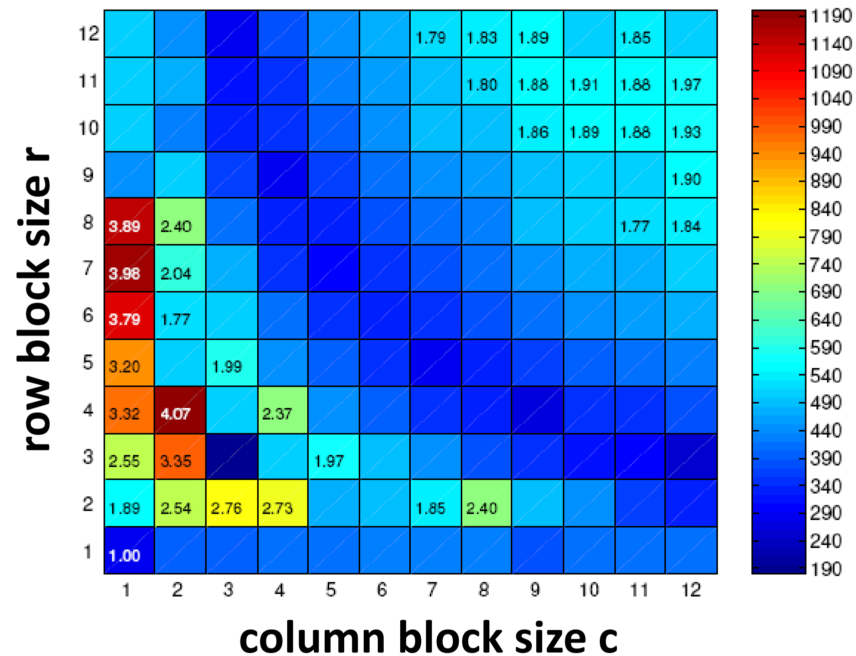
- *Expected gain:* $G_{r,c}/O_{r,c}$

Gain from Blocking (Dense Matrix in BCSR)

Pentium III



Itanium 2



- **machine dependent**
- **hard to predict**

Source: Eun-Jin Im, Katherine A. Yelick, Richard Vuduc. *SPARSITY: An Optimization Framework for Sparse Matrix Kernels*, *Int'l Journal of High Performance Comp. App.*, 18(1), pp. 135-158, 2004

Typical Result

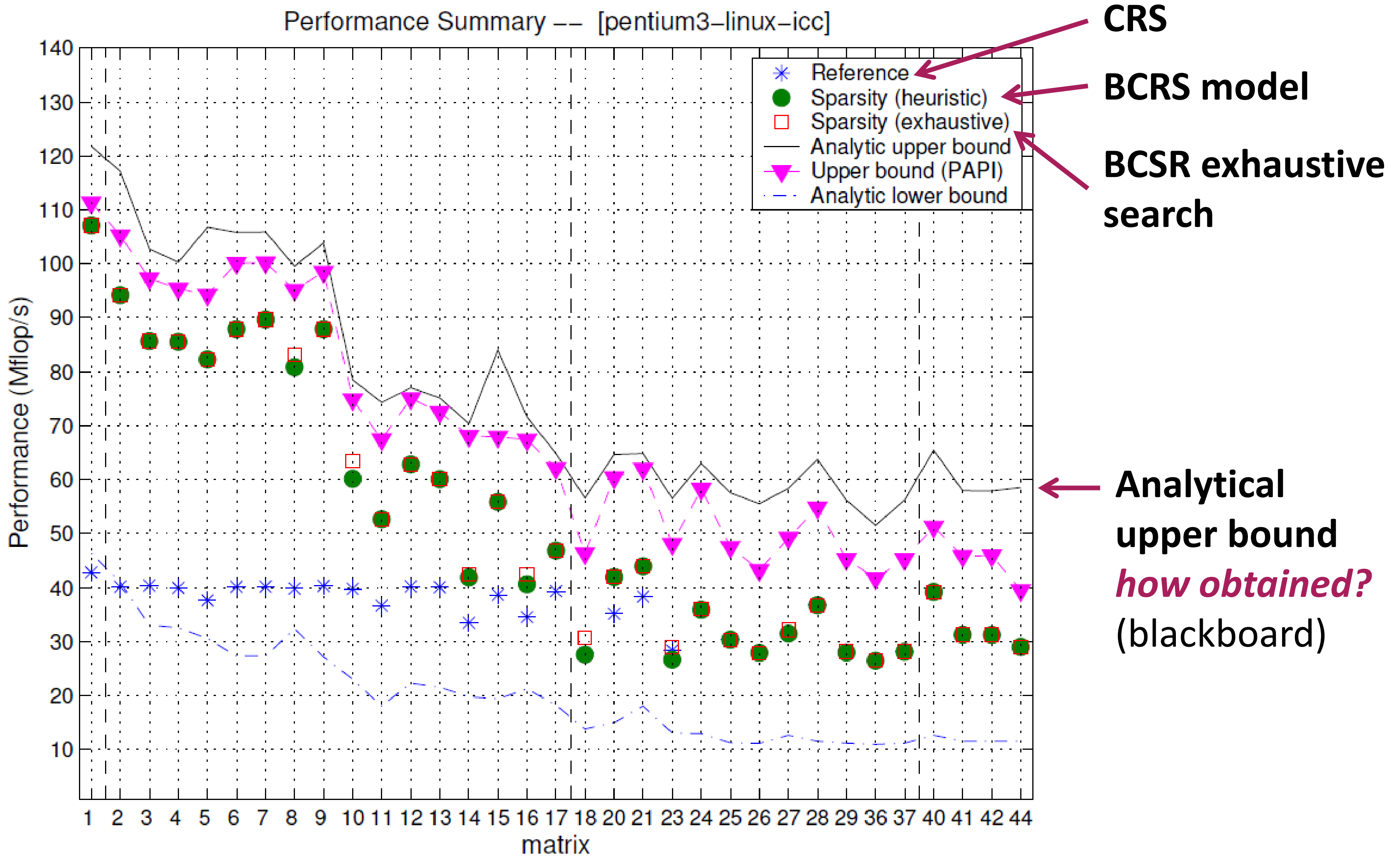


Figure: Eun-Jin Im, Katherine A. Yelick, Richard Vuduc. *SPARSITY: An Optimization Framework for Sparse Matrix Kernels*, *Int'l Journal of High Performance Comp. App.*, 18(1), pp. 135-158, 2004