Efficient sparse matrix-vector products on Fermi GPUs

István Reguly
Mike Giles

Oxford e-Research Centre
Pázmány Péter Catholic University, Hungary

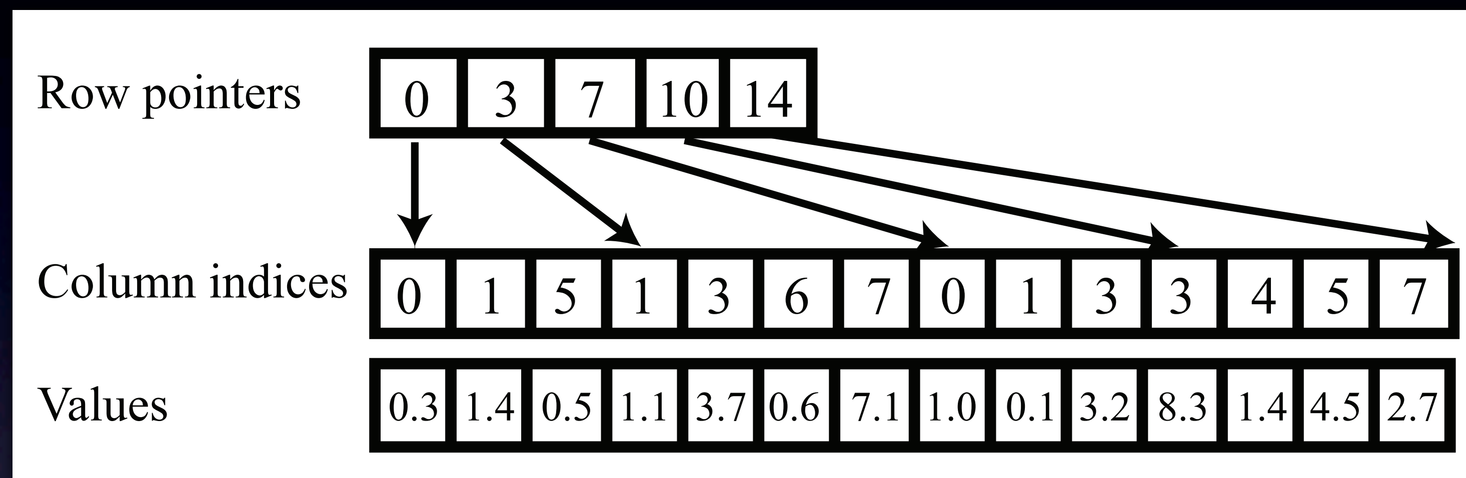
<u>sp</u>arse <u>M</u>atrix-<u>V</u>ector multiplication

- Bandwidth-limited operation
- Plethora of storage formats to improve GPU performance
  - ELLPACK, Hybrid (Bell & Garland)
  - Blocking approaches
- Most widely used format is CSR
- Conversion or preprocessing is expensive

NVIDIA Fermi

- Introduction of 16k/48k L1 and 768k L2 cache
- Global fetch operations load 128 byte cache lines
- 384 lines in L1 cache
- Cache reuse
- Cache trashing is expensive

Compressed Sparse Row

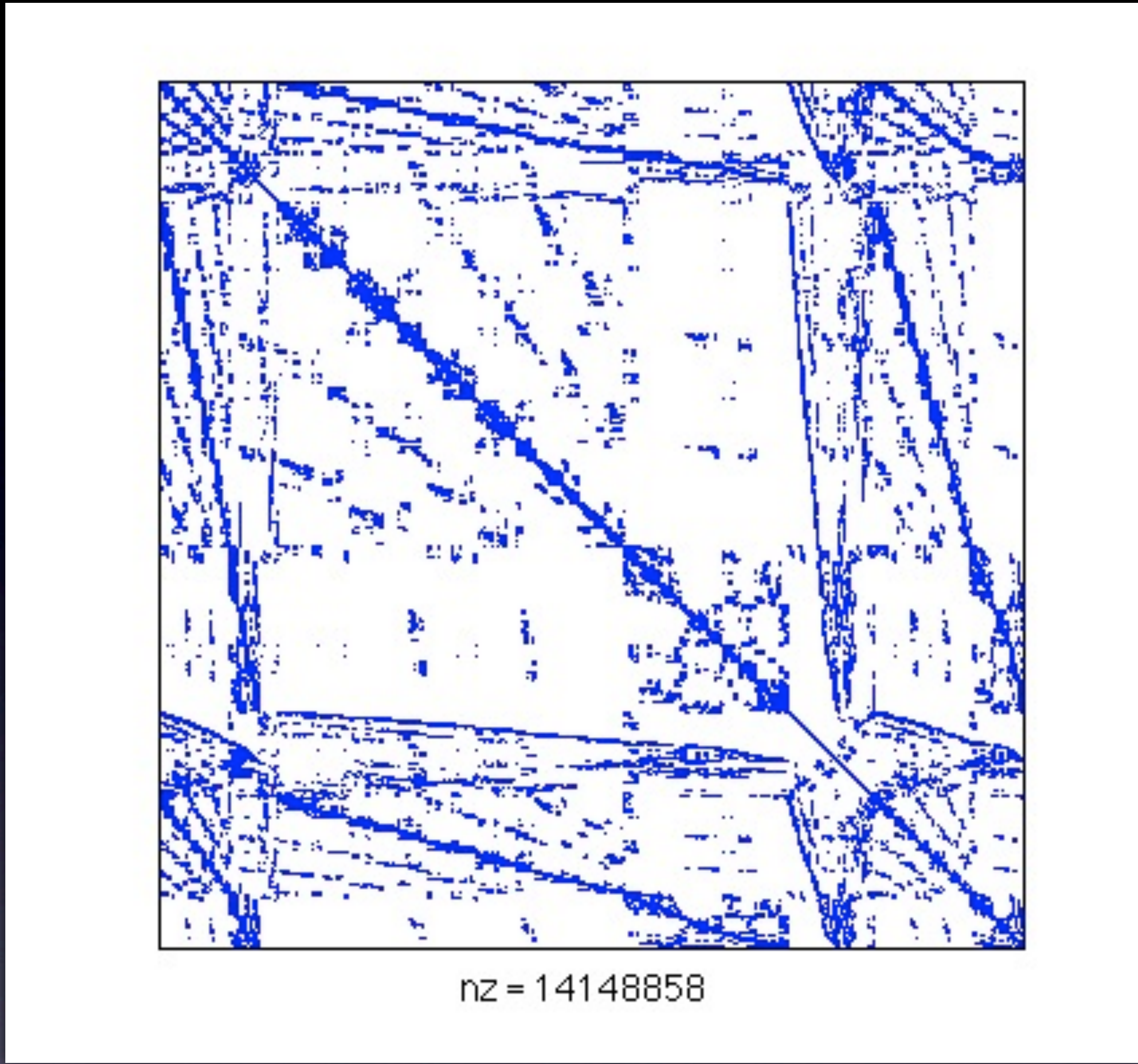



```
int i = blockIdx.x*blockSize + threadIdx.x;
float rowSum = 0;
int rowPtr = rowPtrs[i];
for (int j = 0; j<rowPtrs[i+1]-rowPtr; j+=1) {
        rowSum += values[rowPtr+j] * x[colIdxs[rowPtr+j]];
}
y[i] = rowSum;
```

- No coalescing
- Bad caching

Matrix structure matters

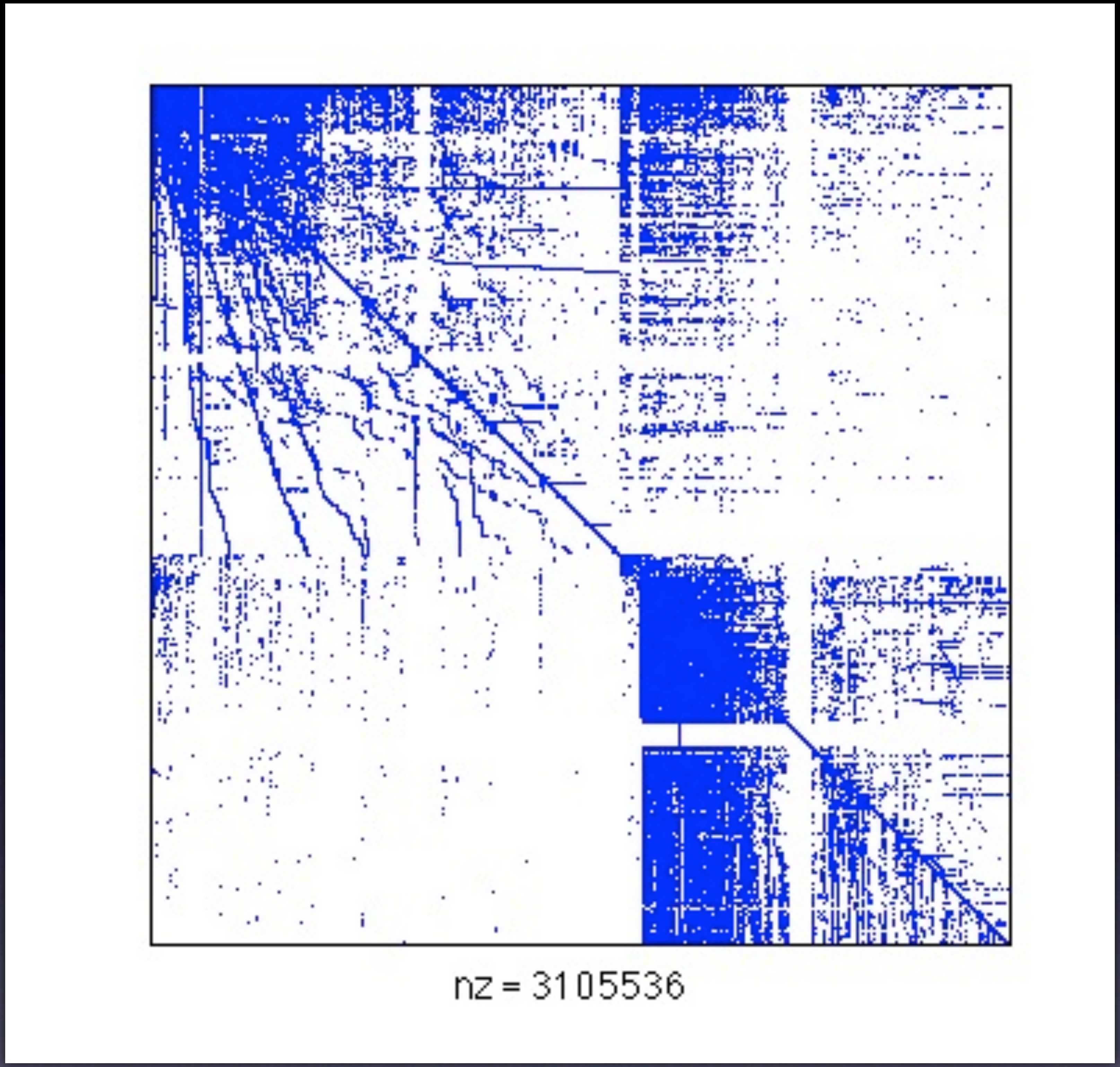
- Average number of non-zeros per row
- Std. deviation of the number of non-zeros per row
- Similar column indices in consecutive rows
- 15 matrices from the University of Florida Sparse Matrix Collection
- Against cuSPARSE 4.0
- NVIDA Tesla C2070



Avg
221

Std. Dev.
3

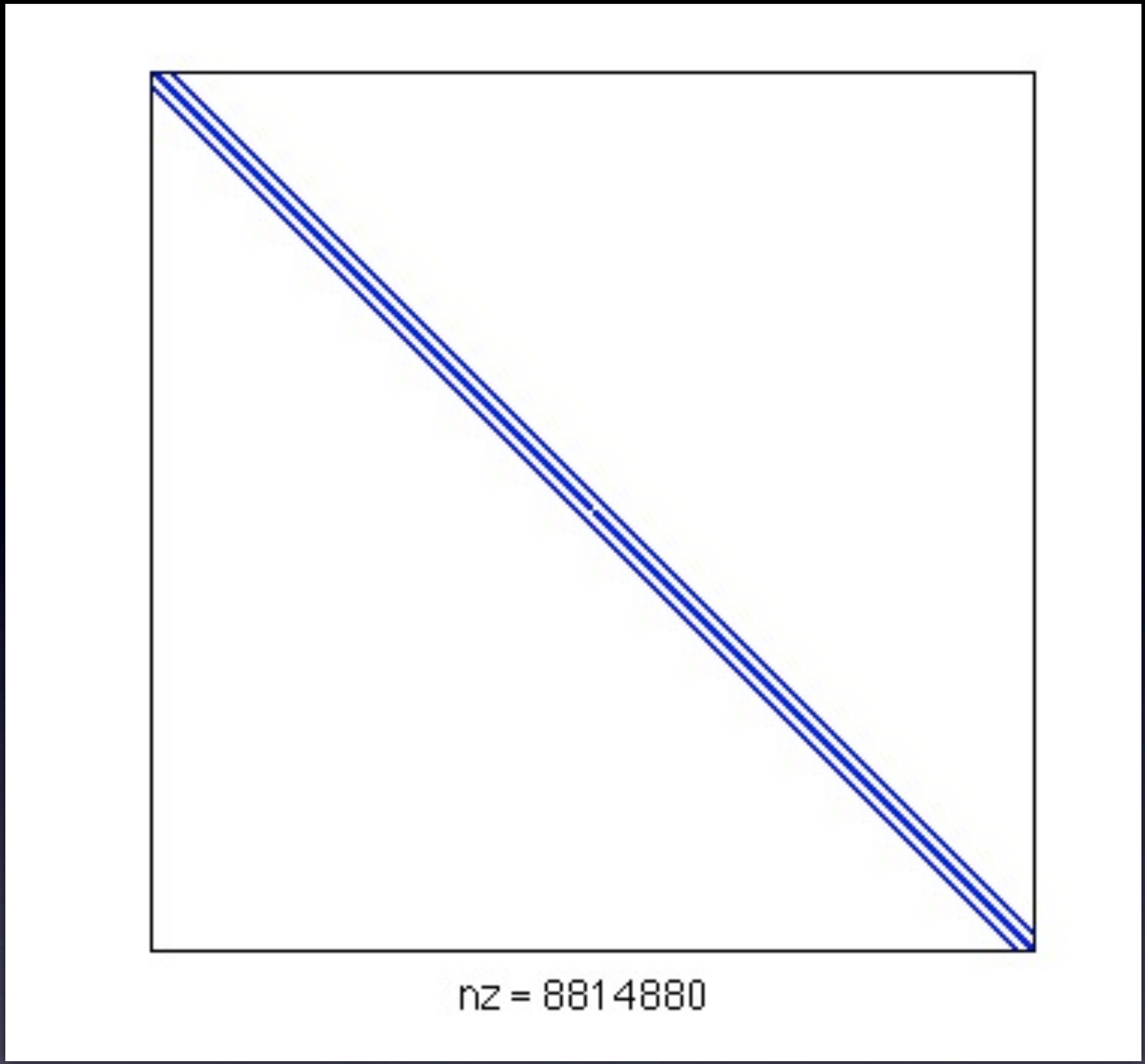
Structural problem



Avg
3.1

Std. Dev.
25.3

Web connectivity



Avg
6.9

Std. Dev.
0.25

Atmospheric modeling

```
__global__ void csrmv(float *values, int *rowPtrs,
                      int *colIdxs, float *x, float *y,
                      int dimRow, int repeat, int coop) {
    int i = (repeat*blockIdx.x*blockDim.x + threadIdx.x)/coop;
    int coopIdx = threadIdx.x%coop;
    int tid = threadIdx.x;
    extern __shared__ volatile float sdata[];
    for (int r = 0; r<repeat; r++) {
        float localSum = 0;
        if (i<dimRow) {
            // do multiplication
            int rowPtr = rowPtrs[i];
            for (int j = coopIdx; j<rowPtrs[i+1]-rowPtr; j+=coop) {
                localSum += values[rowPtr+j] * x[colIdxs[rowPtr+j]];
            }
            // do reduction in shared mem
            sdata[tid] = localSum;
            for(unsigned int s=coop/2; s>0; s>>=1) {
                if (coopIdx < s) sdata[tid] += sdata[tid + s];
            }
            if (coopIdx == 0) y[i] = sdata[tid];

            i += blockDim.x/coop;
        }
    }
}
```

Partial sum

Reduction

Parameters:

- coop
- repeat
- blockSize

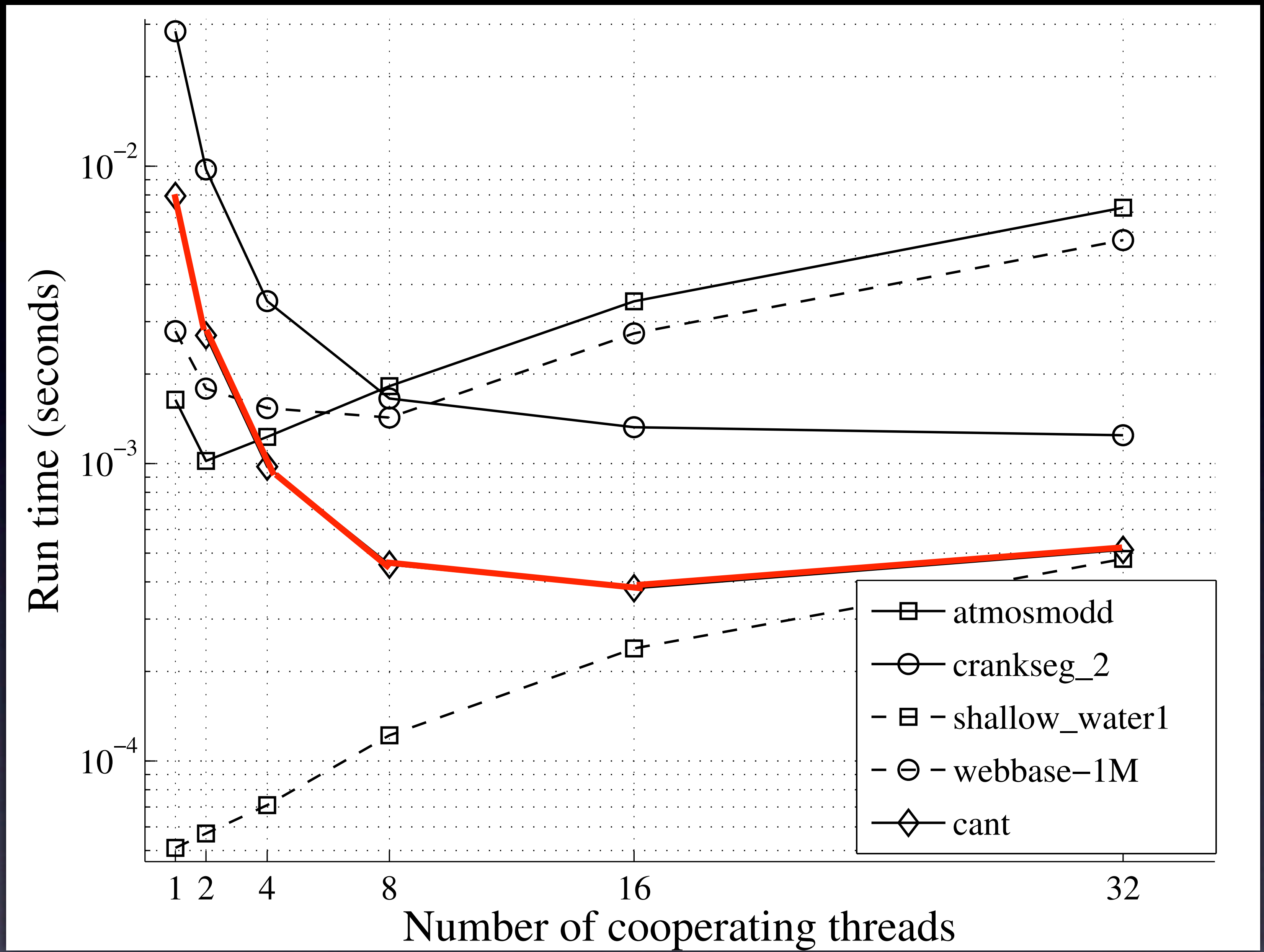
Thread cooperation

coop=4

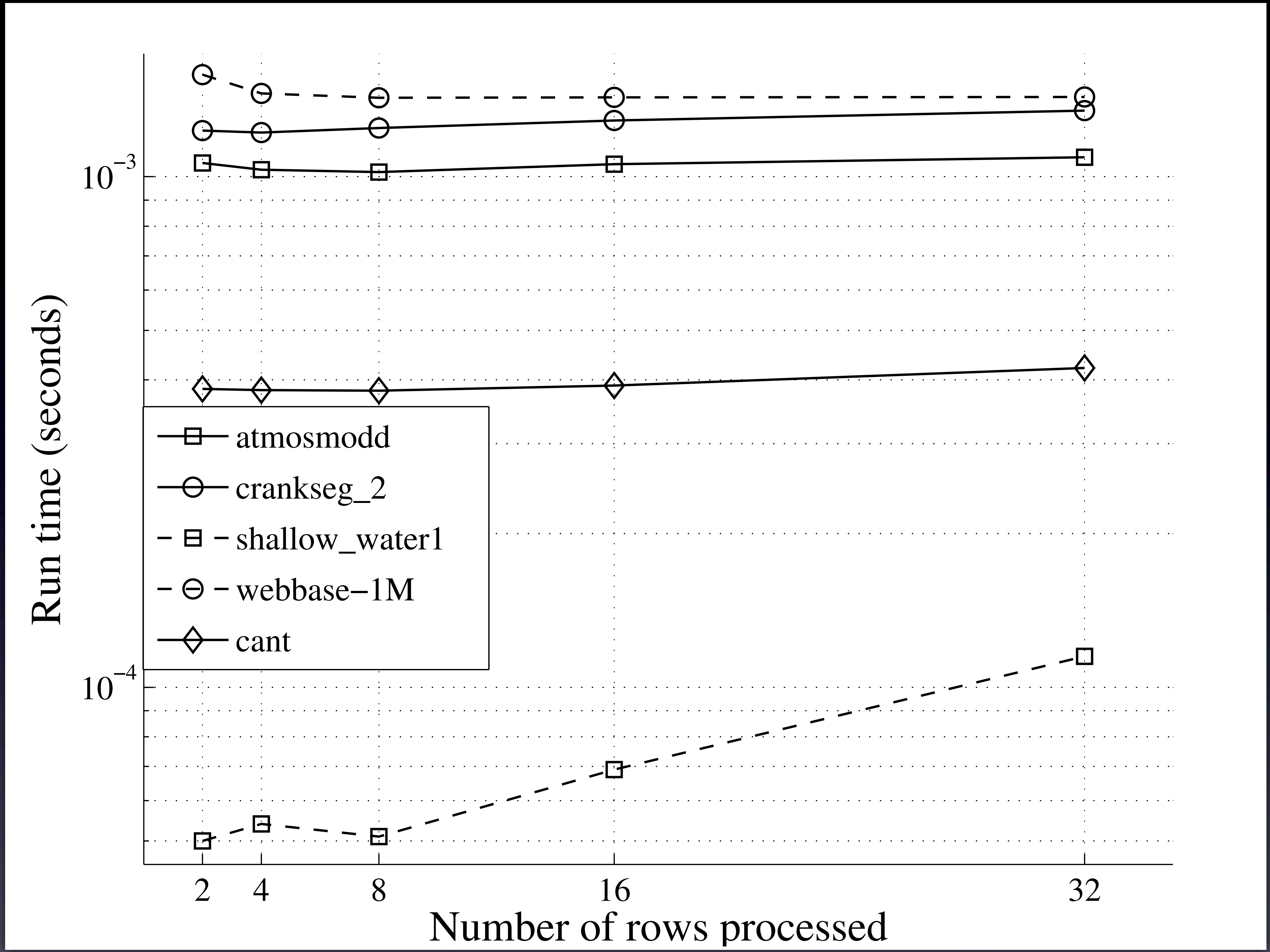
- Coalesced access
- Good caching

- Warp divergence

Number of cooperating threads

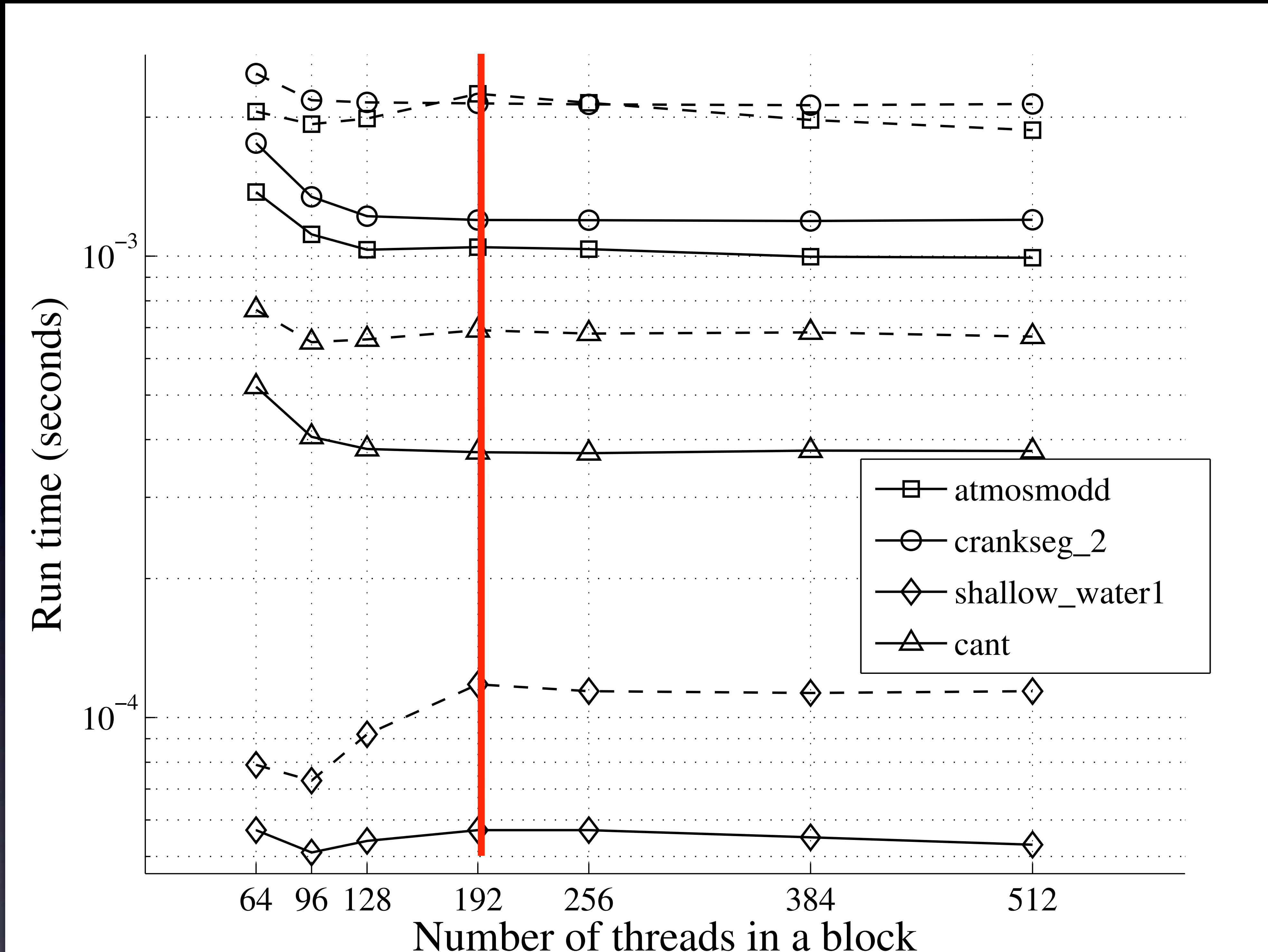


coop	1	2	4	8	16	32
L1 hit rate (%)	19.56	27.3	30.12	69.5	74.3	78.57
Divergence (%)	8	12.8	7.2	9.9	5.3	6.7



Number of rows processed

After fixing coop



Number of threads in a block

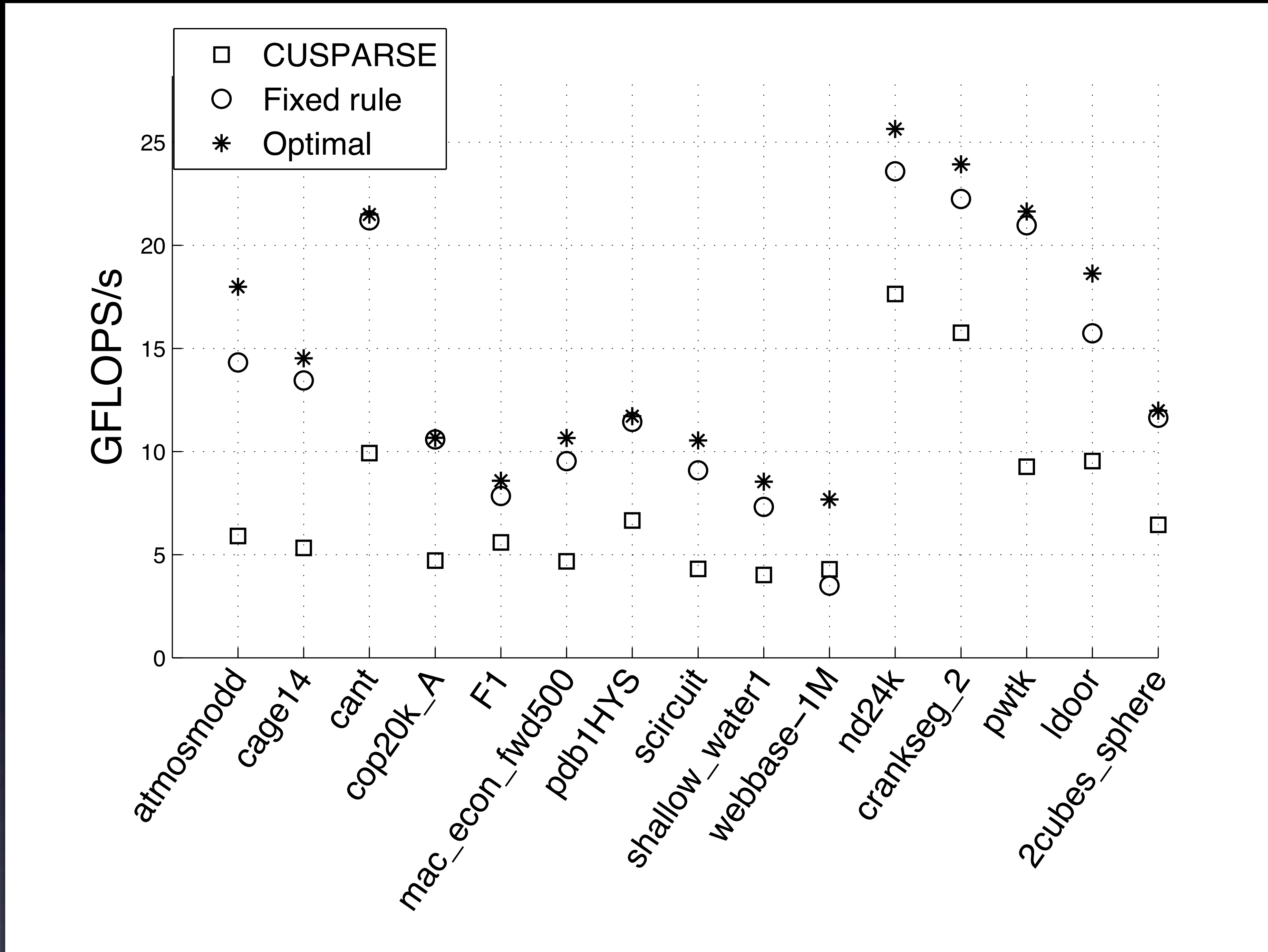
After fixing coop

The fixed rule

- Optimal parameters depend on matrix
- Auto-tuning, exhaustive search...
- General-purpose library
  - “No” overhead is tolerated
  - Constant time method

The fixed rule

- `blockSize` = 128
- `coop` next bigger 2^n to the square root of the average row length
- `gridSize` and `repeat`: at least 1500 blocks

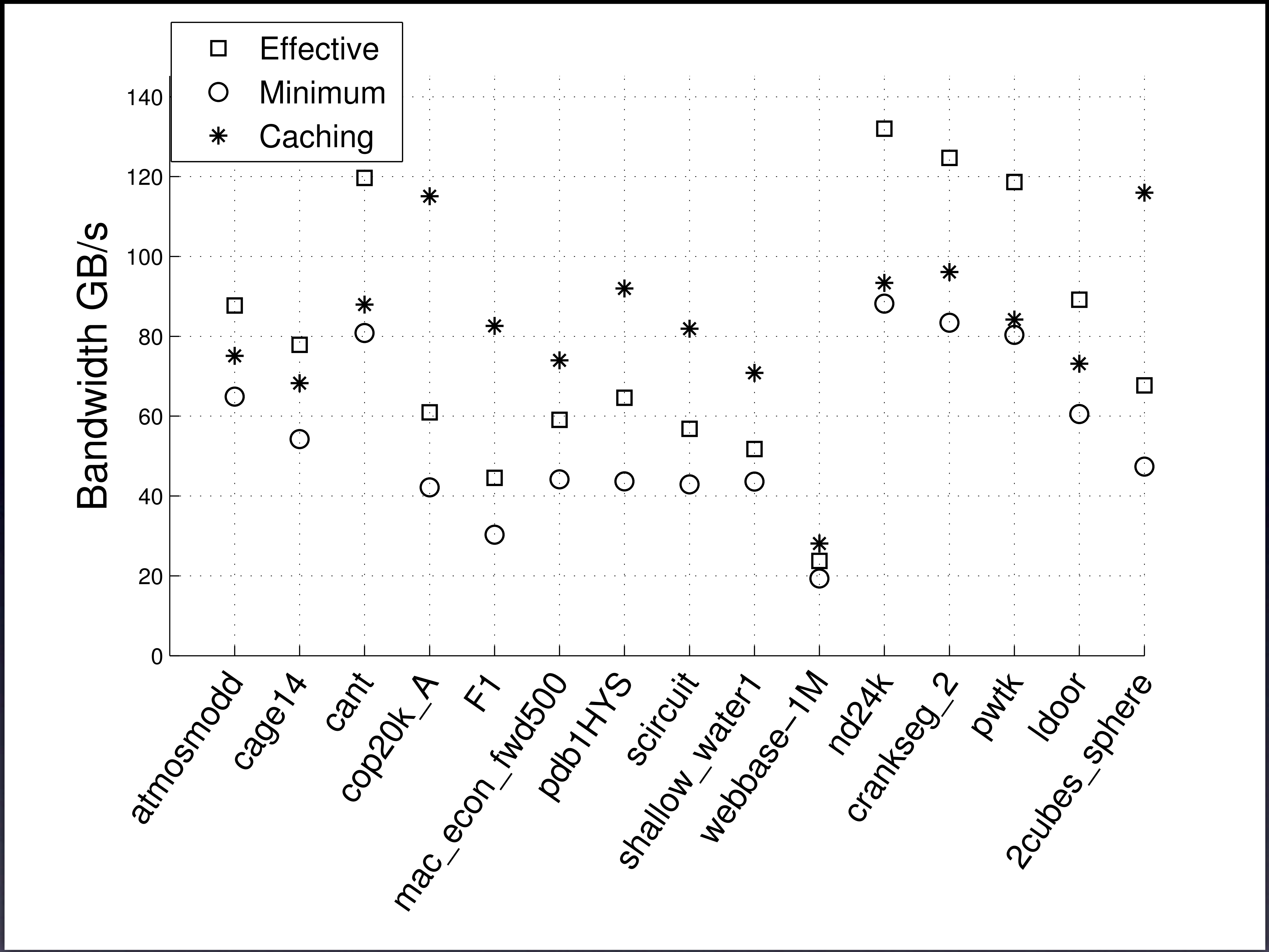


Instruction throughput

~ non-zeros processed per second

Bandwidth estimation

- Minimal, effective bandwidth
- On caching architectures “effective” bandwidth is not the worst case
  - 128 byte cache lines
- Number of cache lines touched by blocks
  - No sharing between blocks
  - Infinite cache size



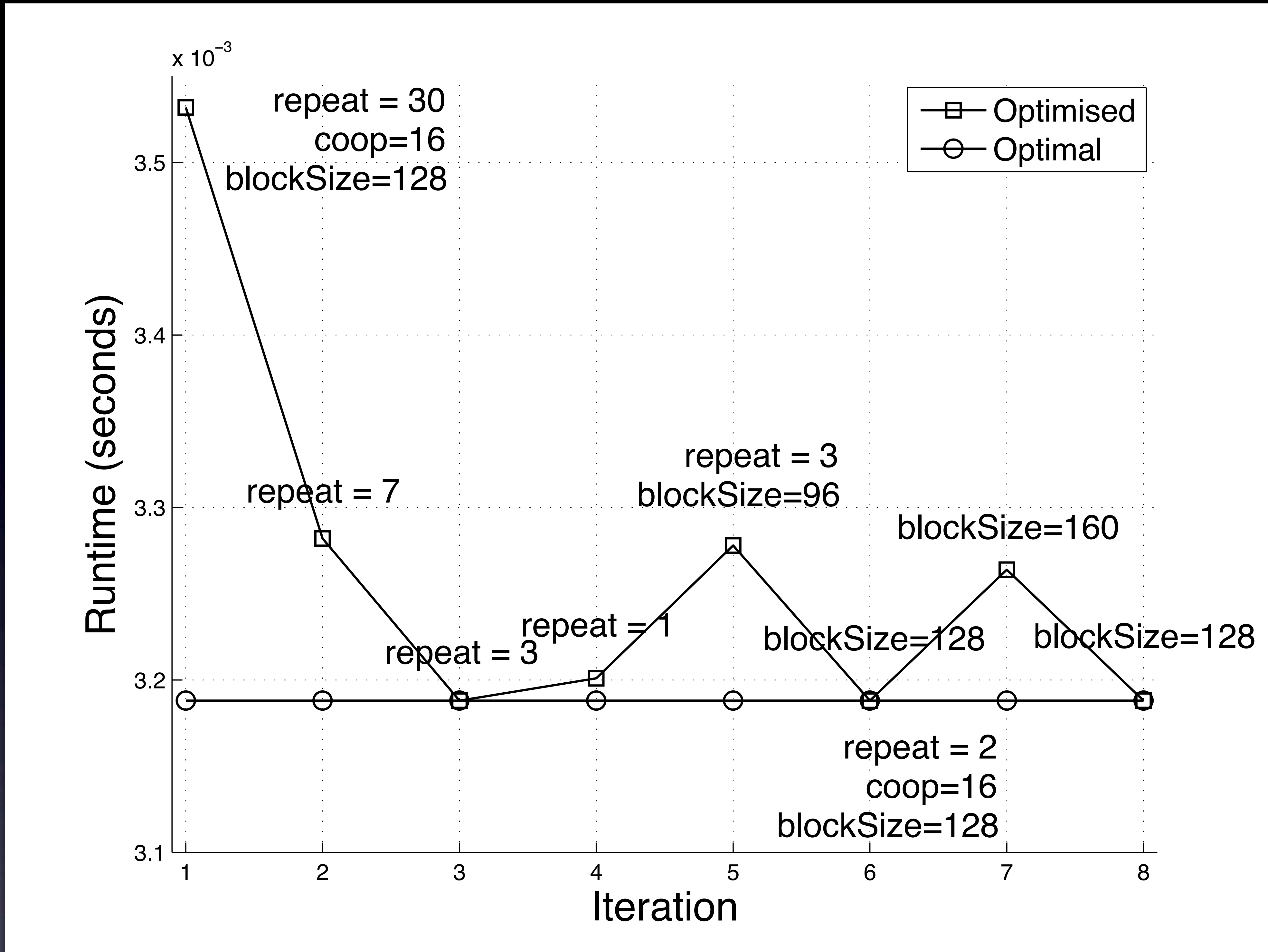
Bandwidth of the fixed rule

Dynamic run-time tuning

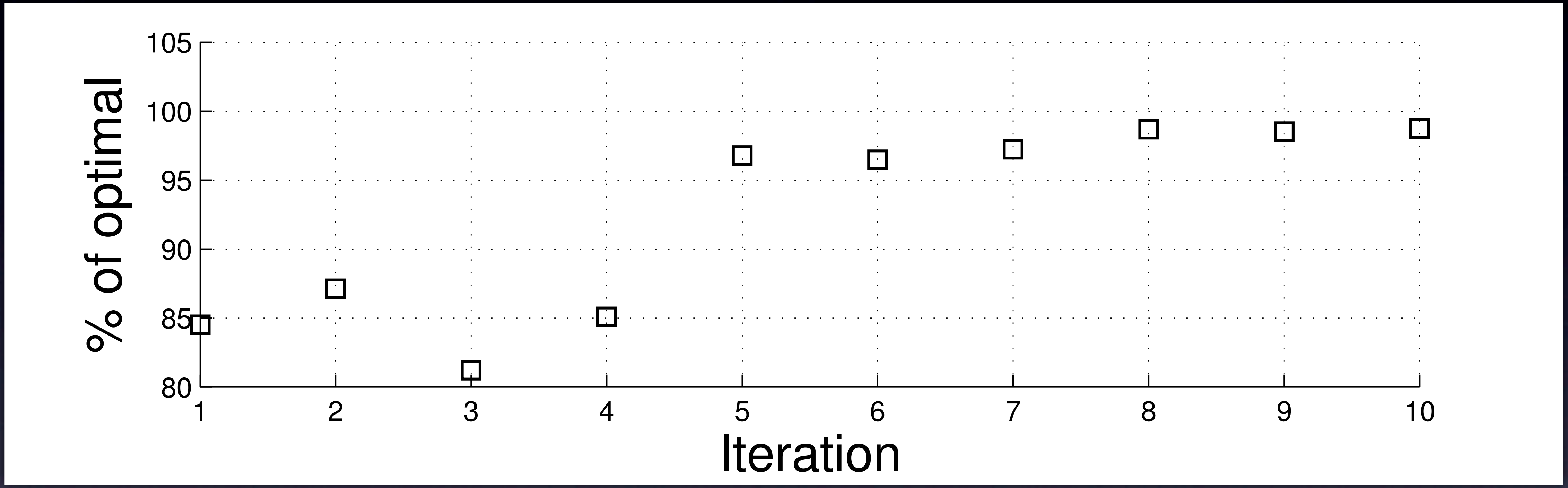
- Iterative algorithms
  - Conjugate Gradient Method
  - Newton-Raphson Method
- Same pointers, same size
  - Assume it has the same structure
  - Tune parameters

Dynamic run-time tuning

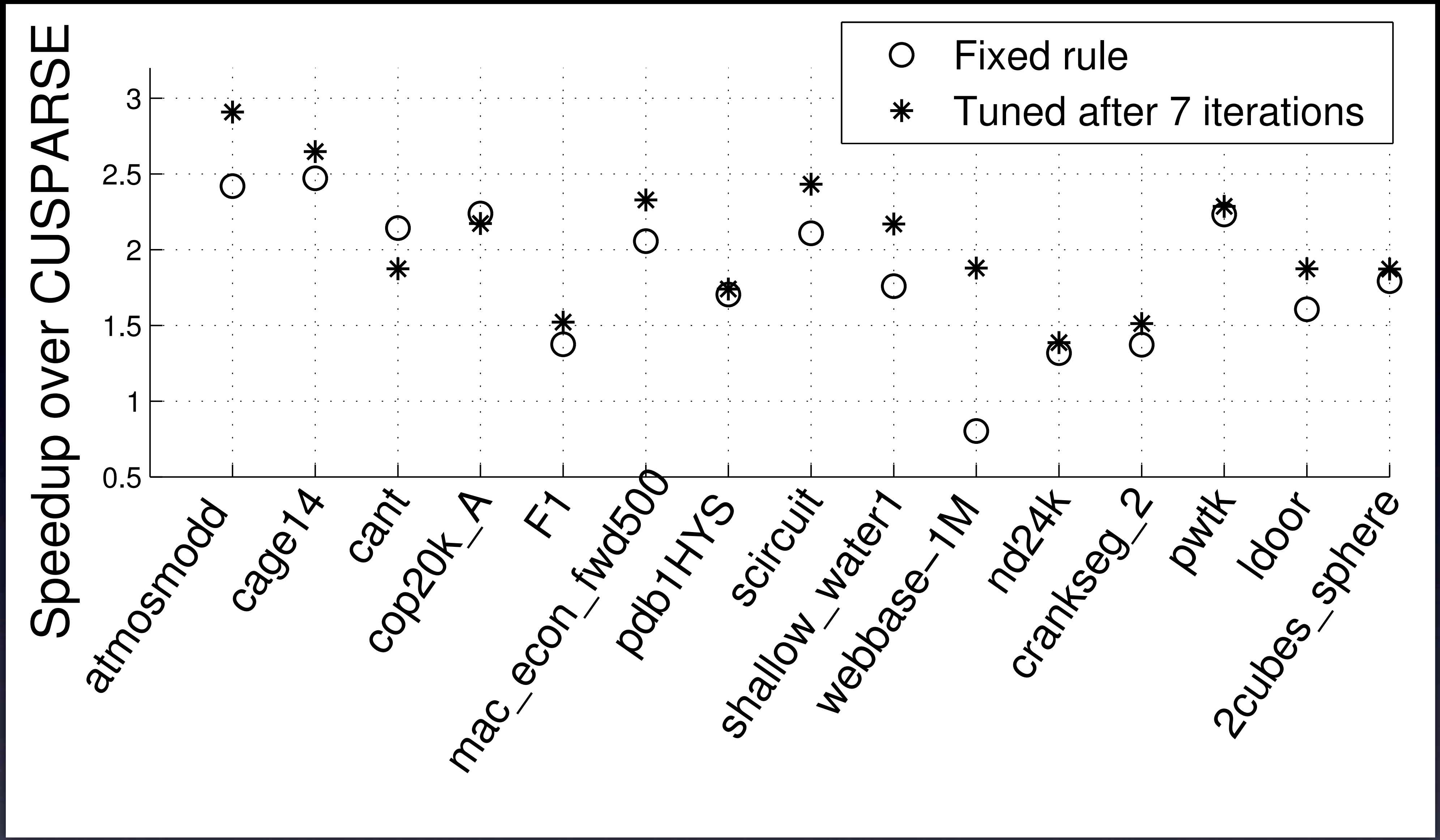
- Blind search on a non-convex domain
  - Bad choices adversely impact performance
- No statistical data
- Empirical algorithm
  - From 84% to 98% in 7 iterations



Tuning on matrix F1

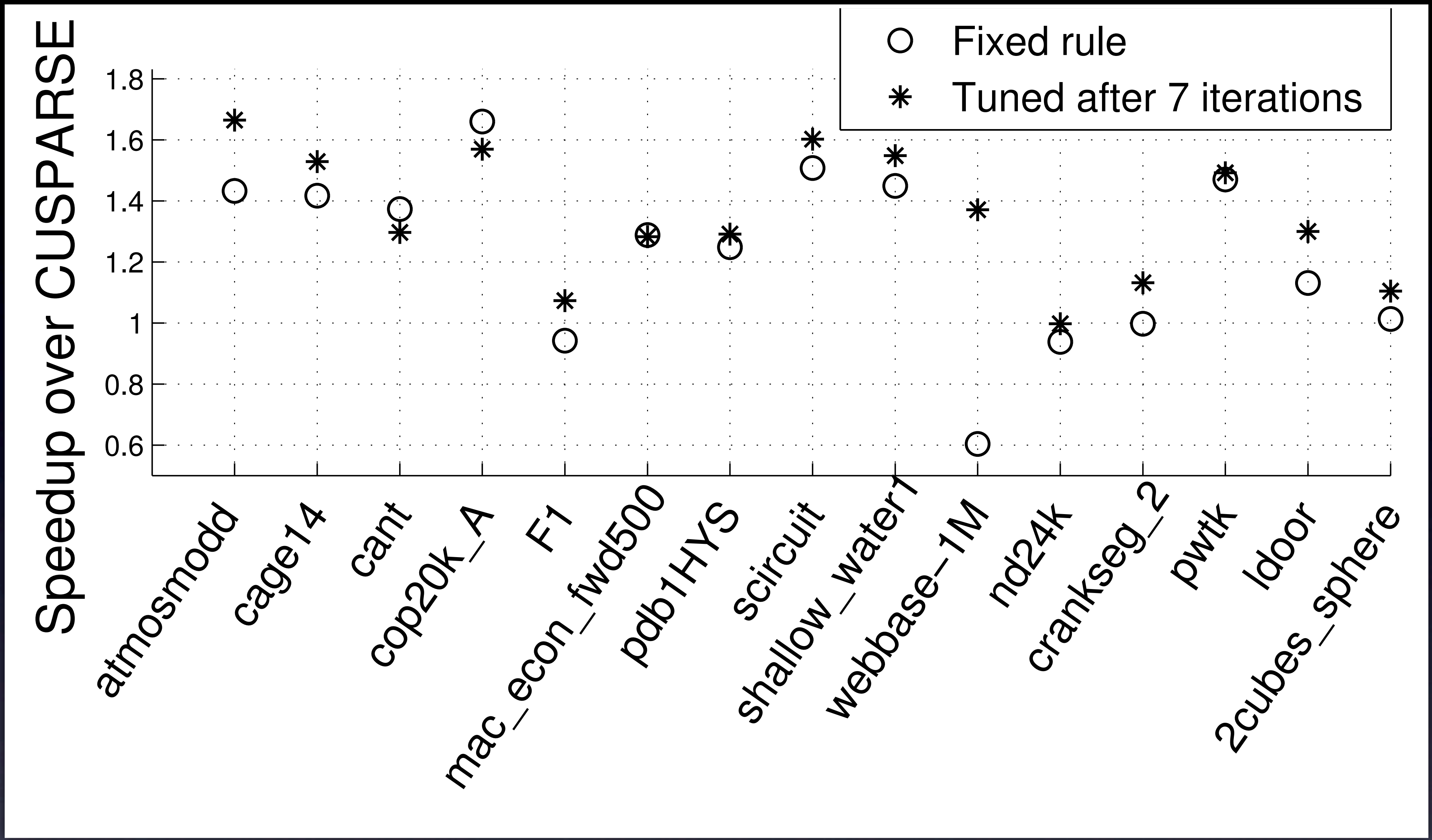


Average performance of 15 matrices



Improvement over cuSPARSE

Single precision: 1.9 times and 2.1 times



Improvement over cuSPARSE

Double precision: 1.25 and 1.35 times

Finite Element Method

- Numerical method for solving PDEs over complicated domains

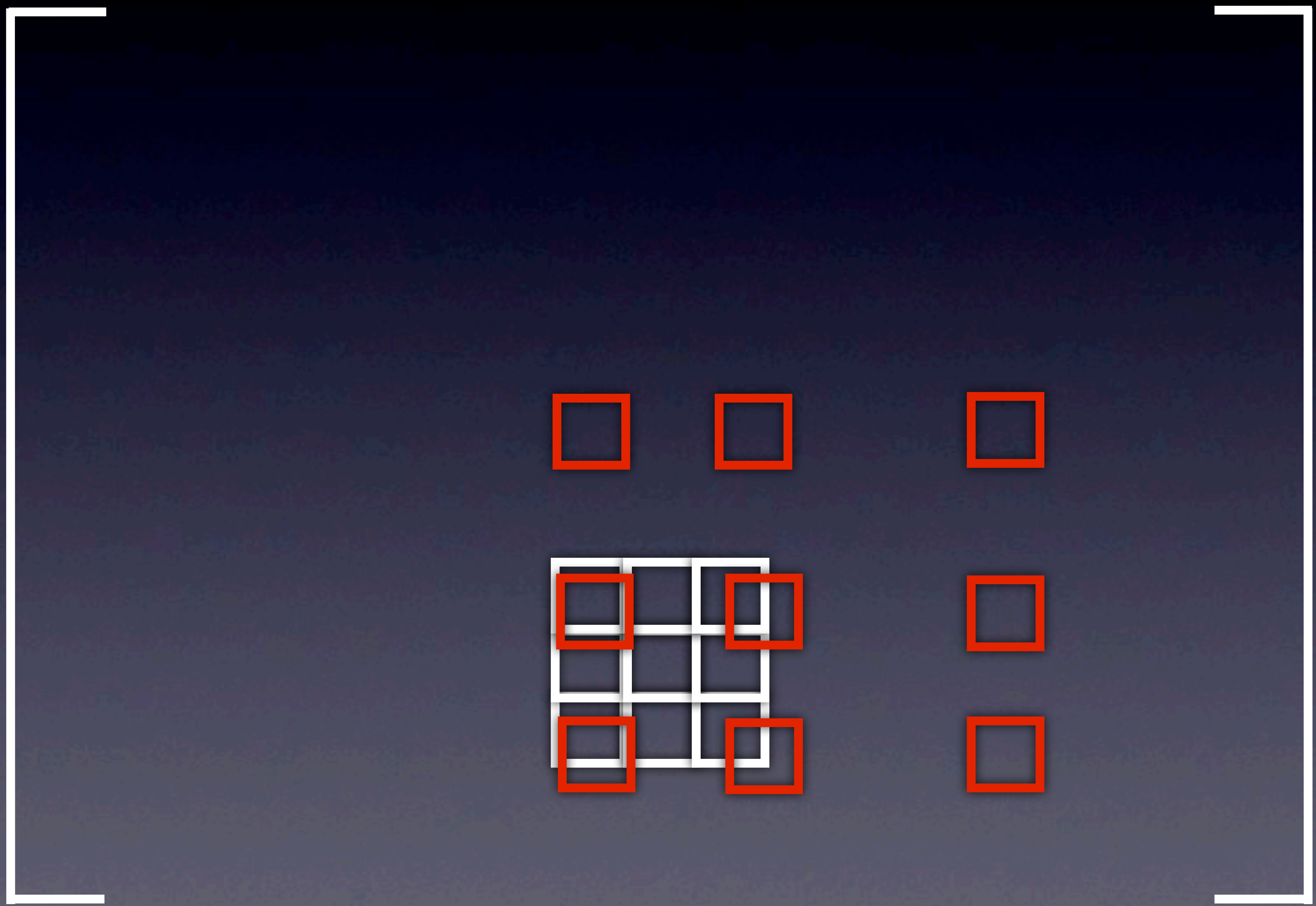
$$\begin{aligned} -\nabla \cdot (\kappa \nabla u) = f \text{ in } \Omega, \\ u = 0 \text{ on } \partial\Omega. \end{aligned}$$

$$K\bar{u} = \bar{l},$$

$$K_{ij} = \int_{\Omega} \kappa \nabla \phi_i \cdot \nabla \phi_j \, dV, \; \forall i, j = 1, 2, \ldots, N_v,$$

$$\bar{l}_i = \int_{\Omega} f \cdot \phi_i, \; \forall i, j = 1, 2, \ldots, N_v.$$

The FEM matrix



Global Matrix Assembly

Compressed Sparse Row (CSR)

Row pointers	0	3	7	10	14									
Column indices	0	1	5	1	3	6	7	0	1	3	3	4	5	7
Values	0.3	1.4	0.5	1.1	3.7	0.6	7.1	1.0	0.1	3.2	8.3	1.4	4.5	2.7

ELLPACK

Column indices

1st row	0	1	5	0
2nd row	1	3	6	7
3rd row	0	1	3	0
4th row	3	4	5	7

Values

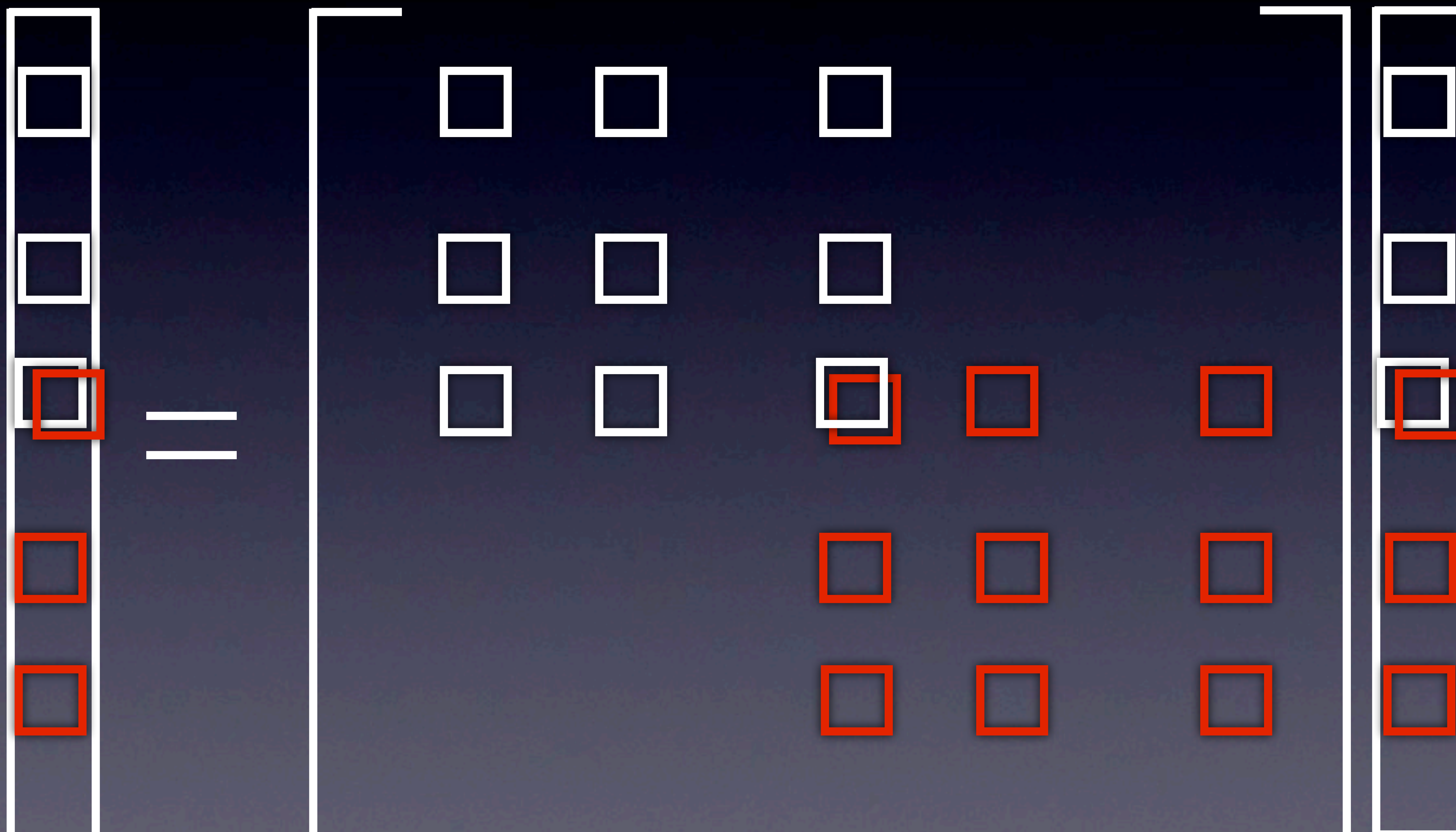
0.3	1.4	0.5	0
1.1	3.7	0.6	7.1
1.0	0.1	3.2	0
8.3	1.4	4.5	2.7

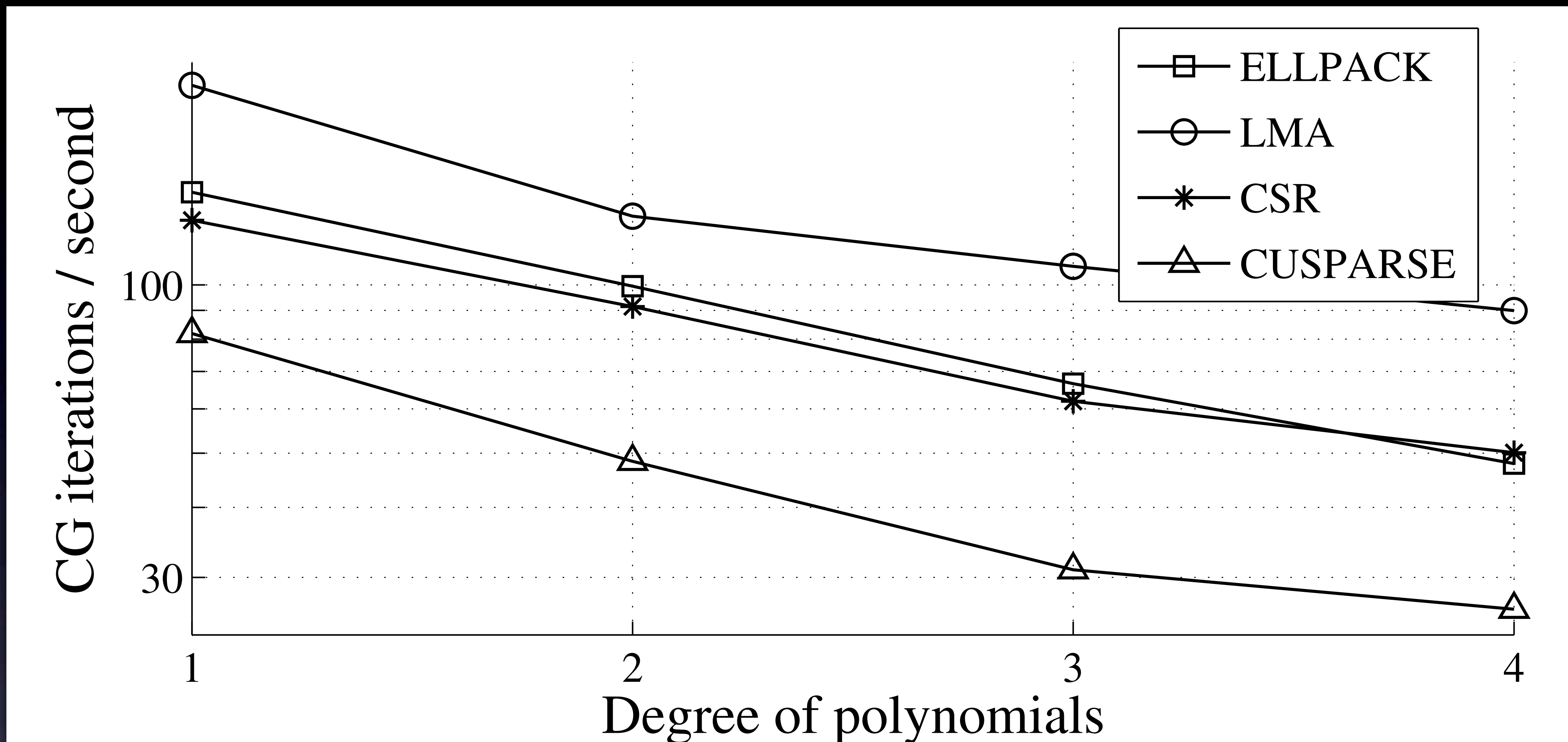
Local Matrix Assembly (LMA)

$$\bar{y} = \mathcal{A}^T(K_e(\mathcal{A}\bar{x})),$$

LM_1	(1,1)	(1,2)	(2,1)	(2,2)
LM_2	(1,1)	(1,2)	(2,1)	(2,2)
LM_3	(1,1)	(1,2)	(2,1)	(2,2)
LM_4	(1,1)	(1,2)	(2,1)	(2,2)
LM_5	(1,1)	(1,2)	(2,1)	(2,2)

Local Matrix Approach





Iterative solution performance

4 million rows

Conclusions

- An understanding of caching on Fermi
- Fixed rule, run-time tuning
- 1.3 - 2.1 times speedup over cuSPARSE
- Help of problems-specific knowledge in FEM, improving performance 2 times
- Closed the performance gap between CSR and ELLPACK/Hybrid formats

Thank you for your attention!

Questions?