CSE 6220/CX 4220
Introduction to HPC

Lecture 20: Sparse Matrix-Vector Multiplication (SpMV)

Helen Xu
hxu615@gatech.edu

Georgia Tech College of Computing
School of Computational Science and Engineering

1990 Nobel Prize in Economics

Our models strained the computer capabilities of the day [1950s]. I observed that **most of the coefficients in our matrices were zero**; i.e. , the nonzeros were “sparse” in the matrix

- Harry Markowitz

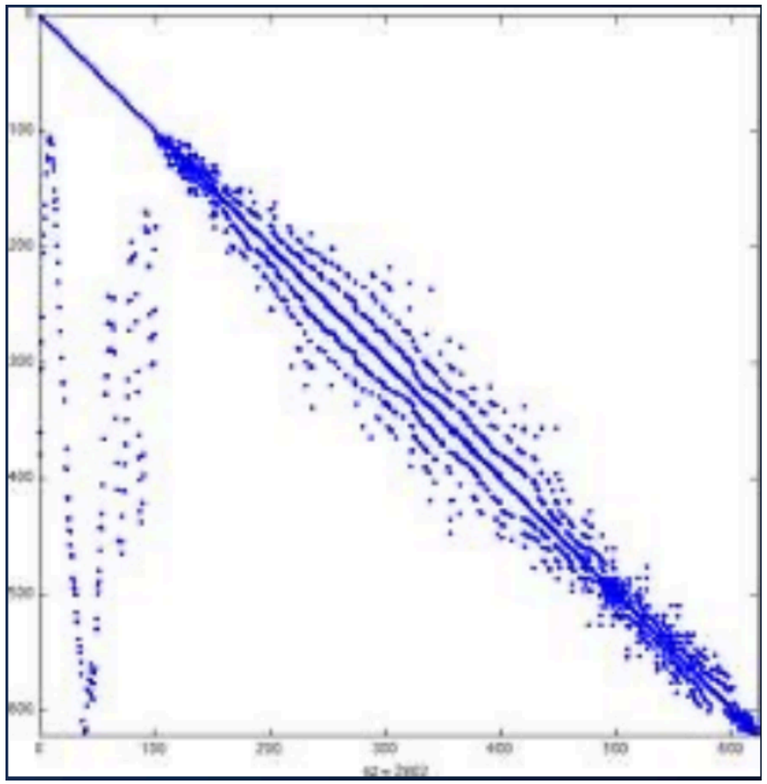
What is a sparse matrix?

A matrix with **primarily zeros** (>> 90%).

Representing them using dense data structures **wastes memory and computation**.

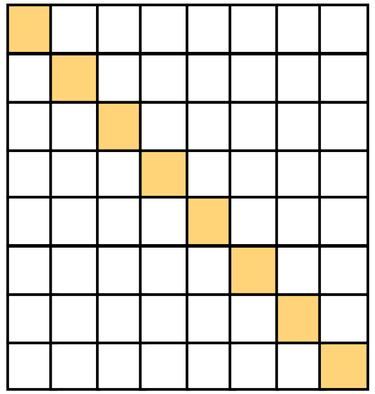
Sparse matrices arise in many applications

- Simulating climate
- Analyzing images (photos, MRIs,…)
- Web page ranking for search
- Graphs, including Graph Neural Nets

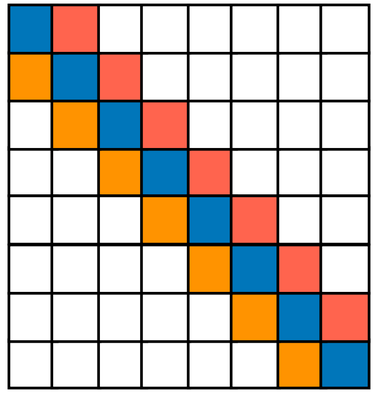


https://sparse.tamu.edu/

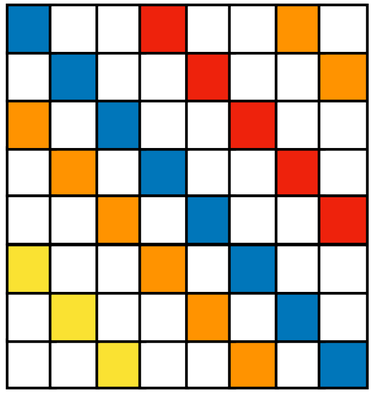
Examples of sparse matrices



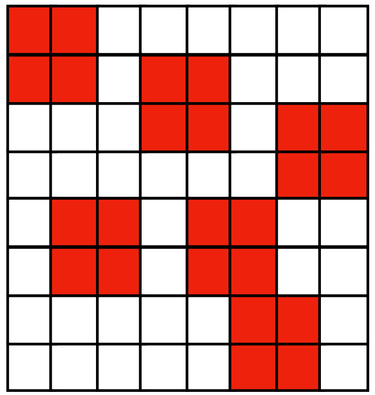
Diagonal



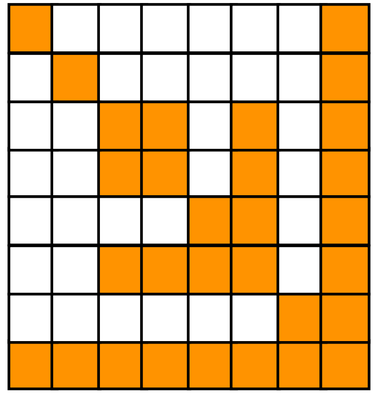
Tridiagonal



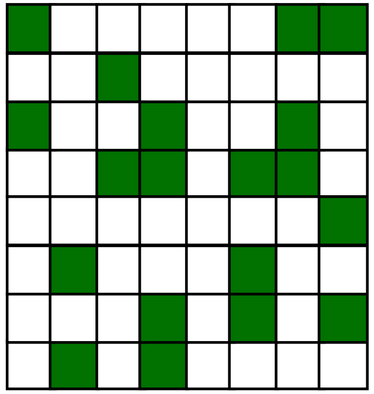
“Generalized diagonal”



Block matrix (2x2 dense blocks)



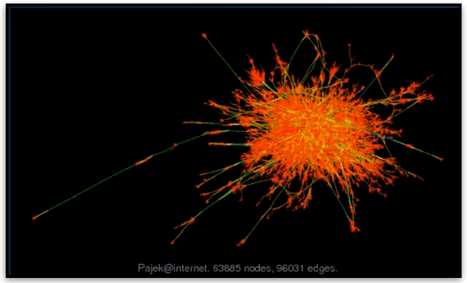
Symmetric

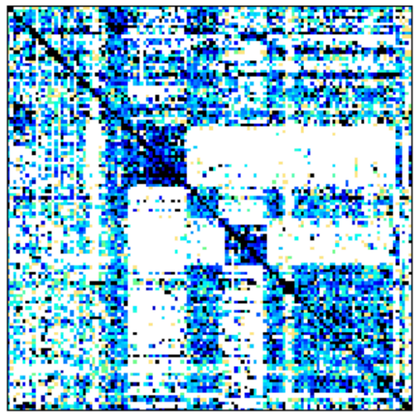


Irregular

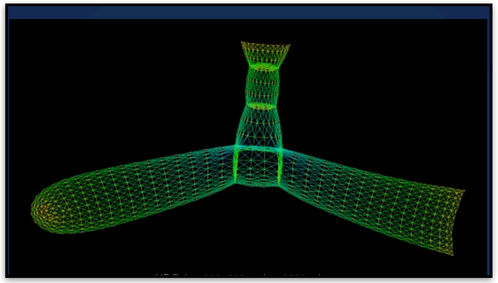
Sparse matrices are everywhere

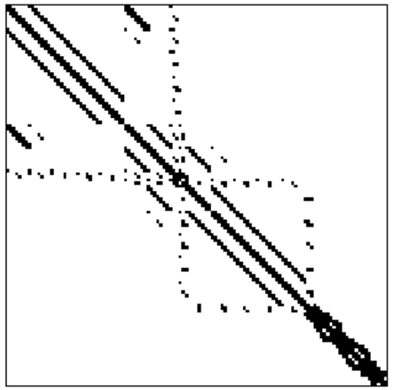
Internet connectivity



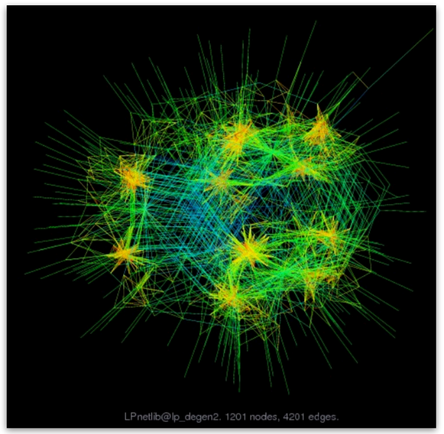


Structural design





Linear programming



Recommendation matrix

Products

Population

Ratings

Applications involving sparse matrices

Graphs
- Google's PageRank (originally an eigen-problem on the web adjacency matrix)
- Transportation network analysis

Text analysis
- Latent Semantic Indexing finds topics in a document corpus by doing a singular value decomposition of a bag-of-words matrix

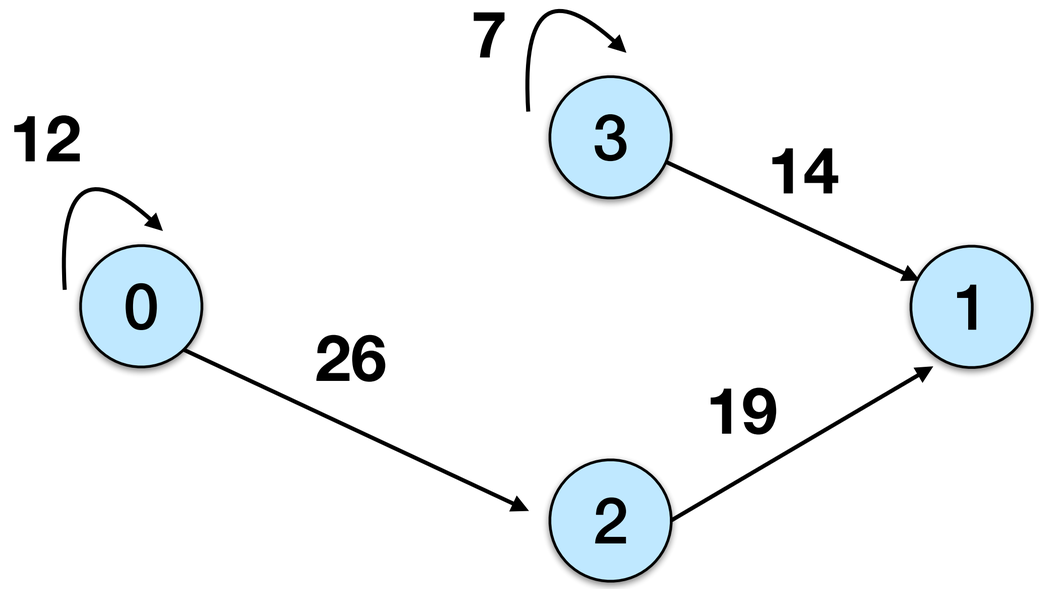
Scientific and engineering
- Solving differential equations (climate modeling, etc.)
- Optimization problems

And many more…

Sparse Matrix Formats

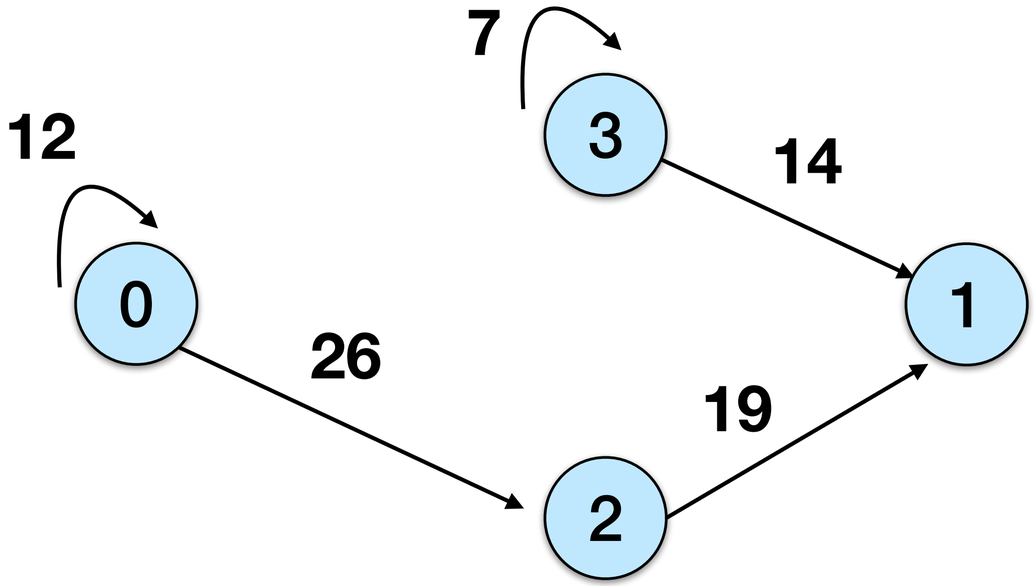
Adjacency matrix representation

Any sparse matrix representation can be used for sparse graphs, and vice versa.



	0	1	2	3
0	12		26	
1				
2		19		
3		14		7

Coordinate representation (COO)

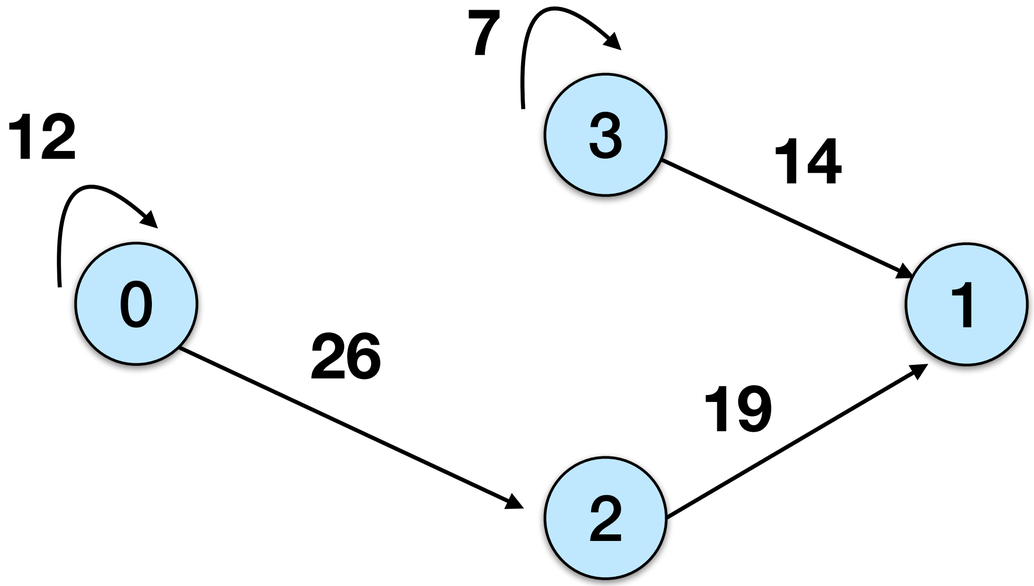


	0	1	2	3
0	12		26	
1				
2		19		
3		14		7

(0, 0, 12)
(0, 2, 26)
(2, 1, 19)
(3, 1, 14)
(3, 3, 7)

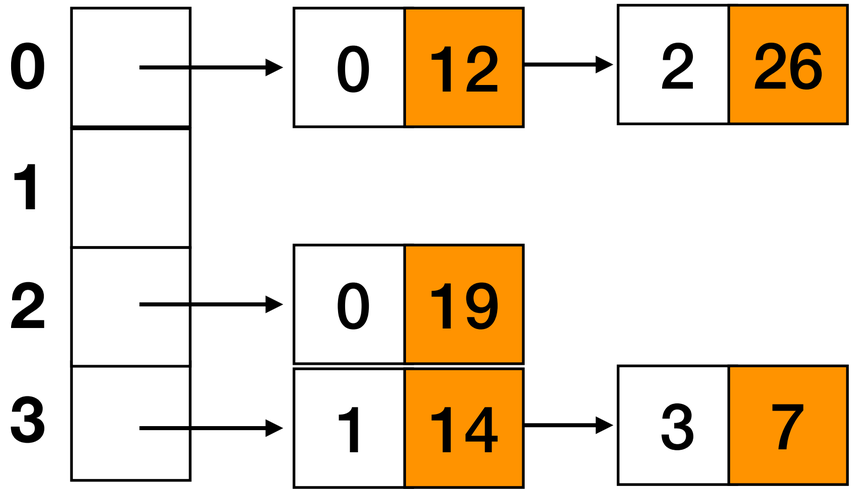
row + column index + weight per nonzero (easy to build / modify)

Adjacency lists



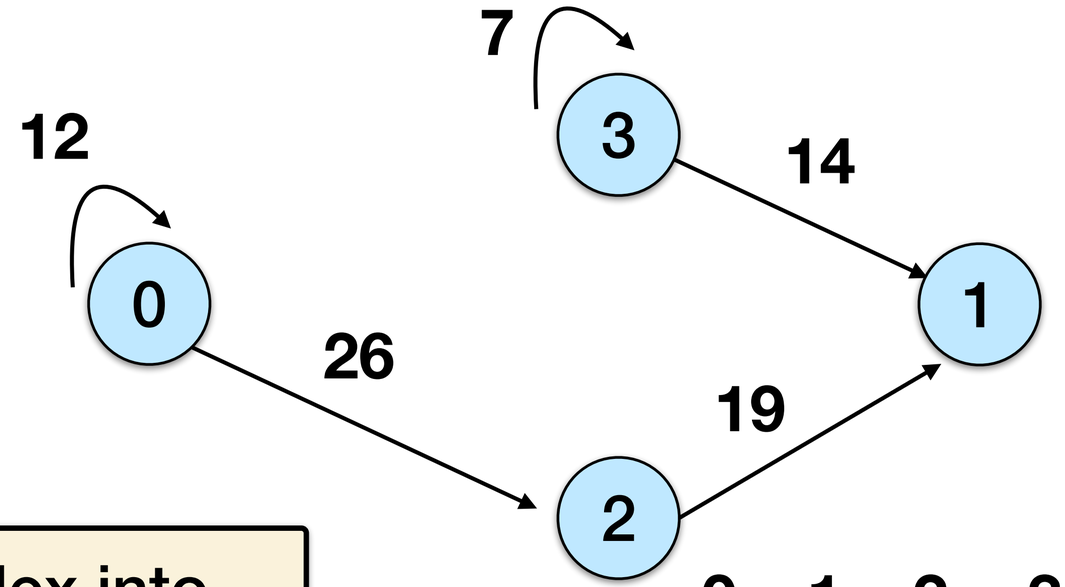
	0	1	2	3
0	12		26	
1				
2		19		
3		14		7

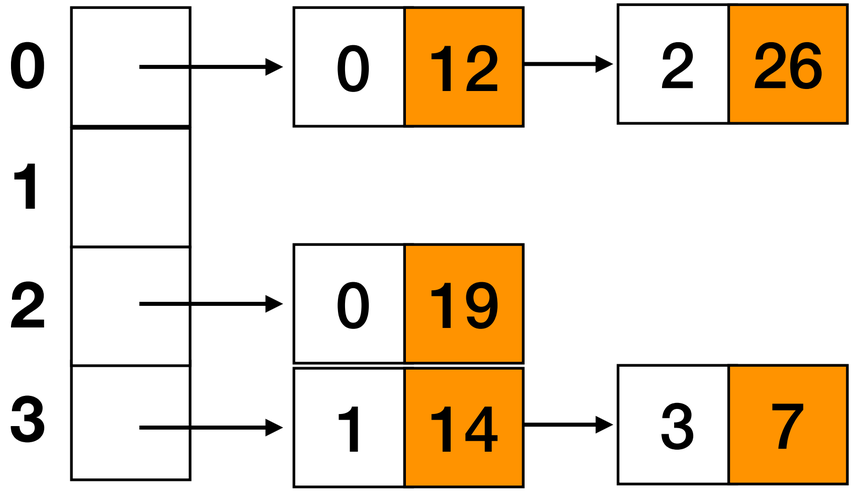
Source stored implicitly



Compressed Sparse Row (CSR)

Compressed sparse row (CSR) = **cache-efficient adjacency lists**





Index into adjacency array

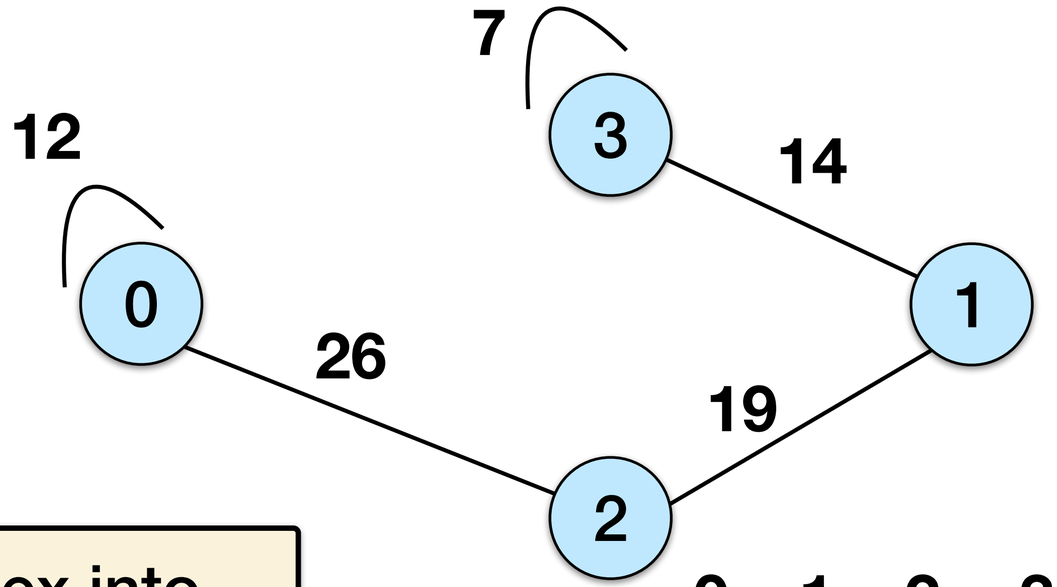
Adjacencies

Weights

	0	1	2	3	4	
ptr	0	2	2	3	5	(row starts in CSR)
ind	0	2	1	1	3	(column ids in CSR)
val	12	26	19	14	7	(numerical values in CSR)

Directed vs Undirected

An undirected graph corresponds to a symmetric matrix - only need to store **half**



	0	1	2	3
0	12		26	
1			19	14
2	26	19		
3		14		7

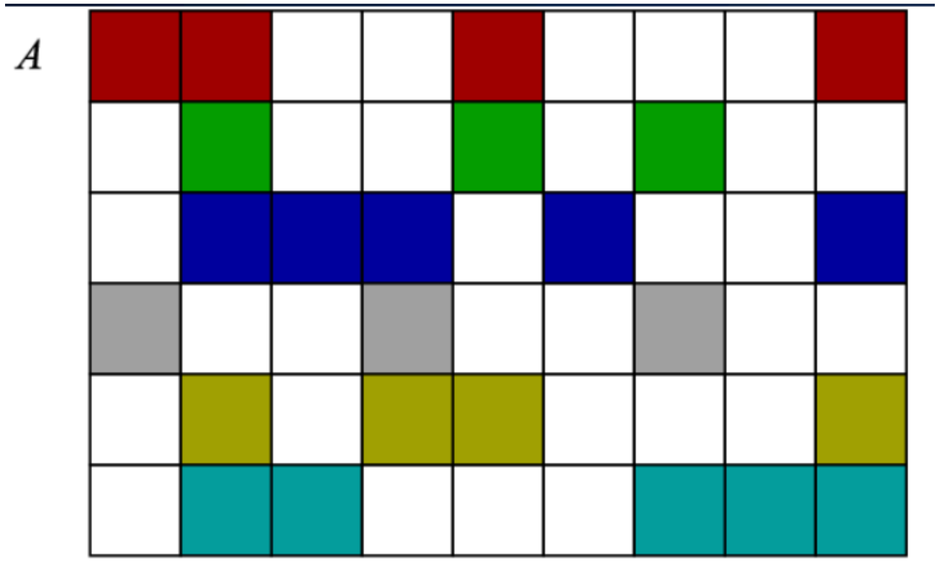
Index into adjacency array

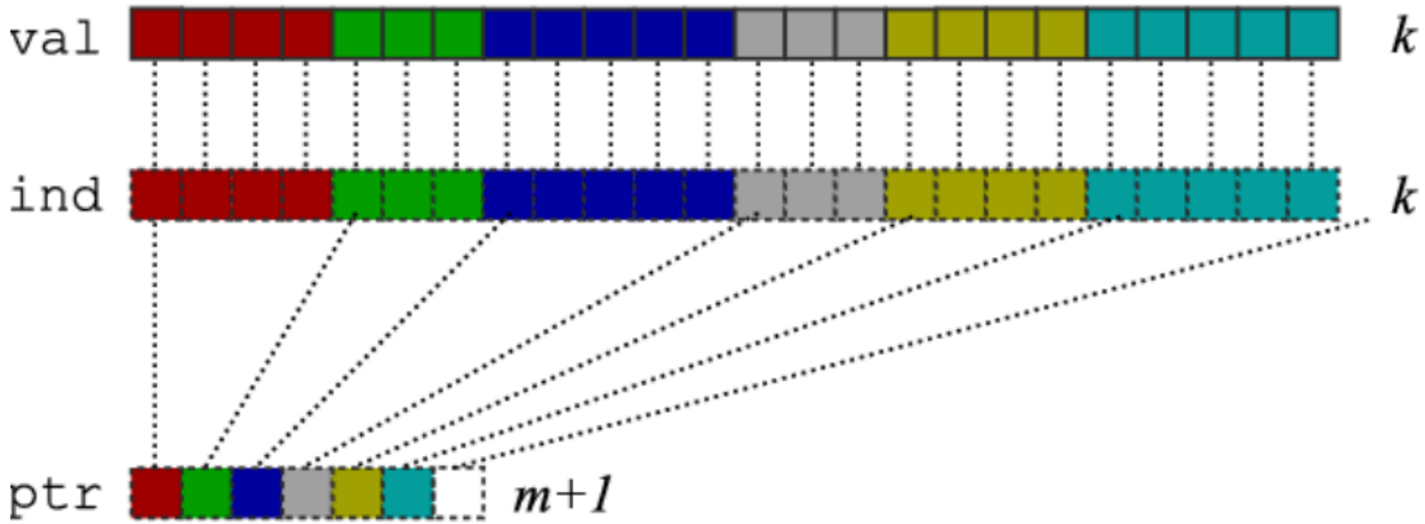
Adjacencies

Weights

	0	1	2	3	4	
ptr	0	2	2	3	5	(row starts in CSR)
ind	0	2	1	1	3	(column ids in CSR)
val	12	26	19	14	7	(numerical values in CSR)

Compressed Sparse Row (CSR) Storage

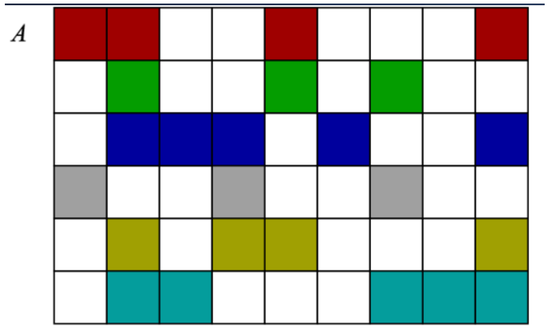




CSR has:

- Size **nnz** = number of nonzeros
- Array of the nonzero **values** (val) of size nnz
- Array of the column **indices** (ind) for each value of size nnz
- Array of row start **pointers** (ptr) of size n = number of rows

Other Storage Formats



Compressed Sparse Row (CSR) is the most common and our focus today

Others include

- Compressed Sparse Column (CSC)
- Diagonal (DIAG): store main diagonal as 1D array; or diagonal bands as 2D (padded)
- Symmetric: store only ½ the array (indexing more complicated)
- **Blocked**: store each block contiguously
  - **Register blocked**: blocks are small and dense, avoid indexes within blocks
  - **Cache blocked**: blocks are large and themselves sparse

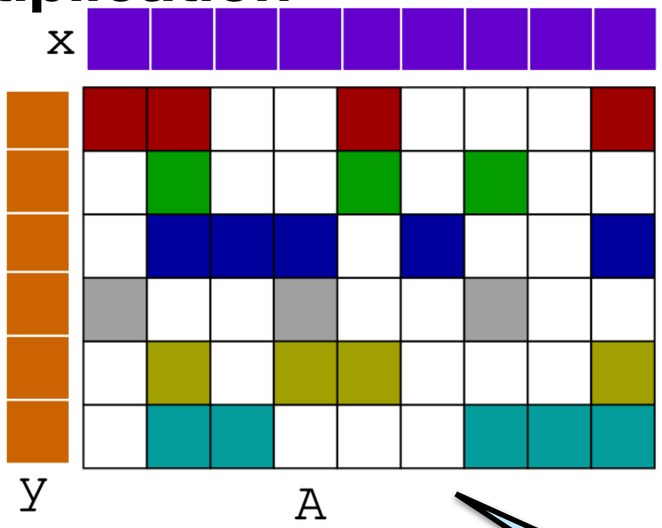
…and many other specialized formats!

From UC Berkeley CS267

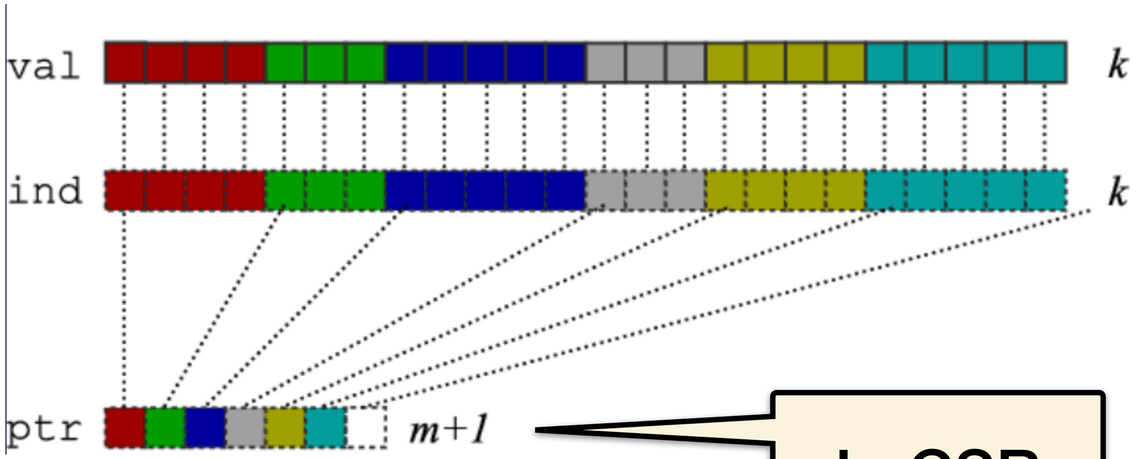
Serial SpMV

Serial SpMV in CSR

SpMV: Sparse Matrix-Vector Multiplication



Representation of A

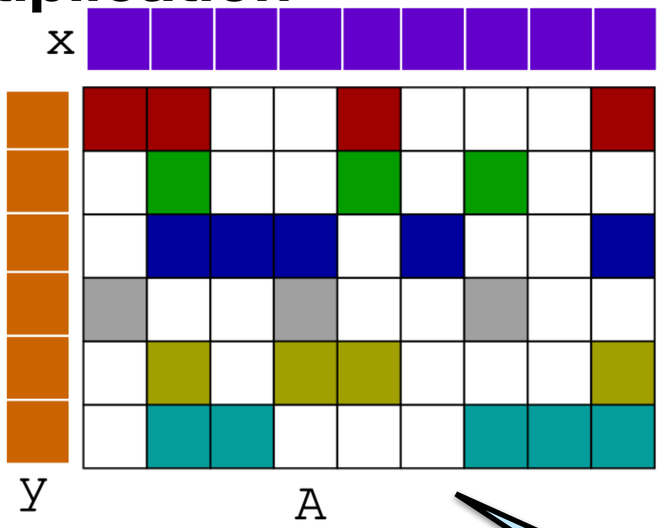


In CSR

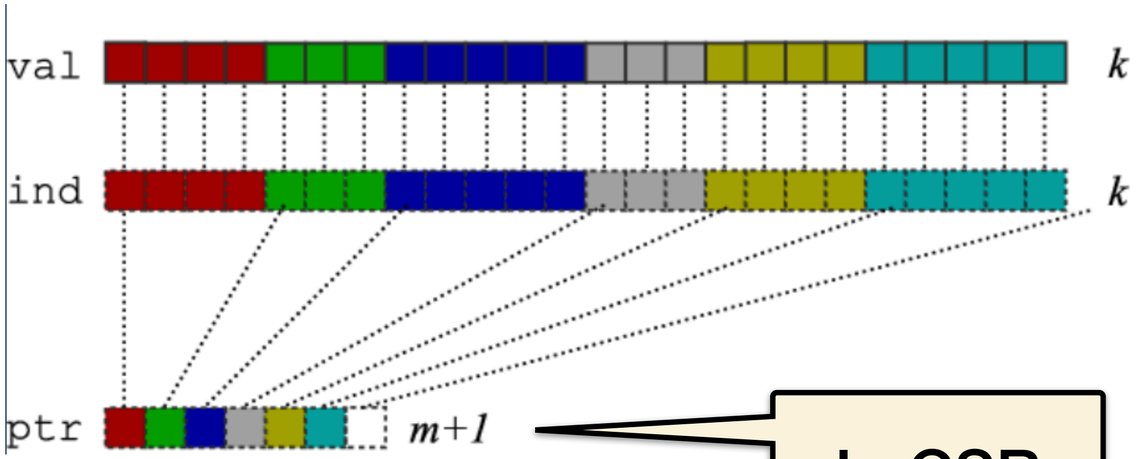
Compute **y(i) = y(i) + A(i, j) * x(j)**

Serial SpMV in CSR

SpMV: Sparse Matrix-Vector Multiplication



Representation of A



In CSR

Compute **y(i) = y(i) + A(i, j) * x(j)**

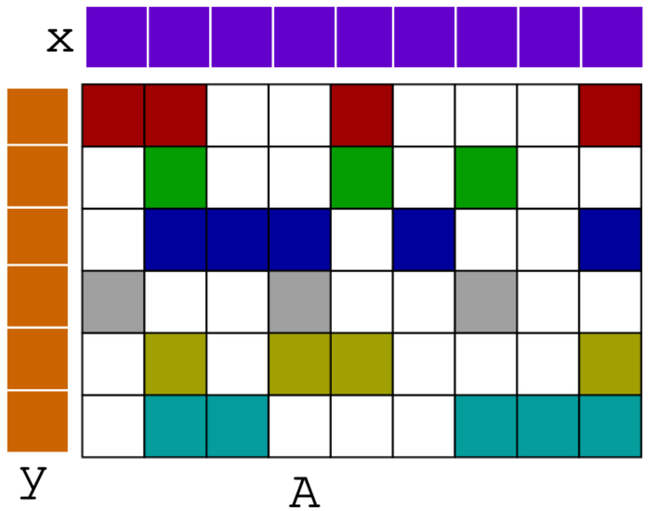
```
for each row i:
  for k = ptr[i] to ptr[i+1] do
    y[i] += val[k] * x[ind[k]]
```

- No reuse in A
- Maximum reuse in y as written
- Reuse in x?

Parallel SpMV

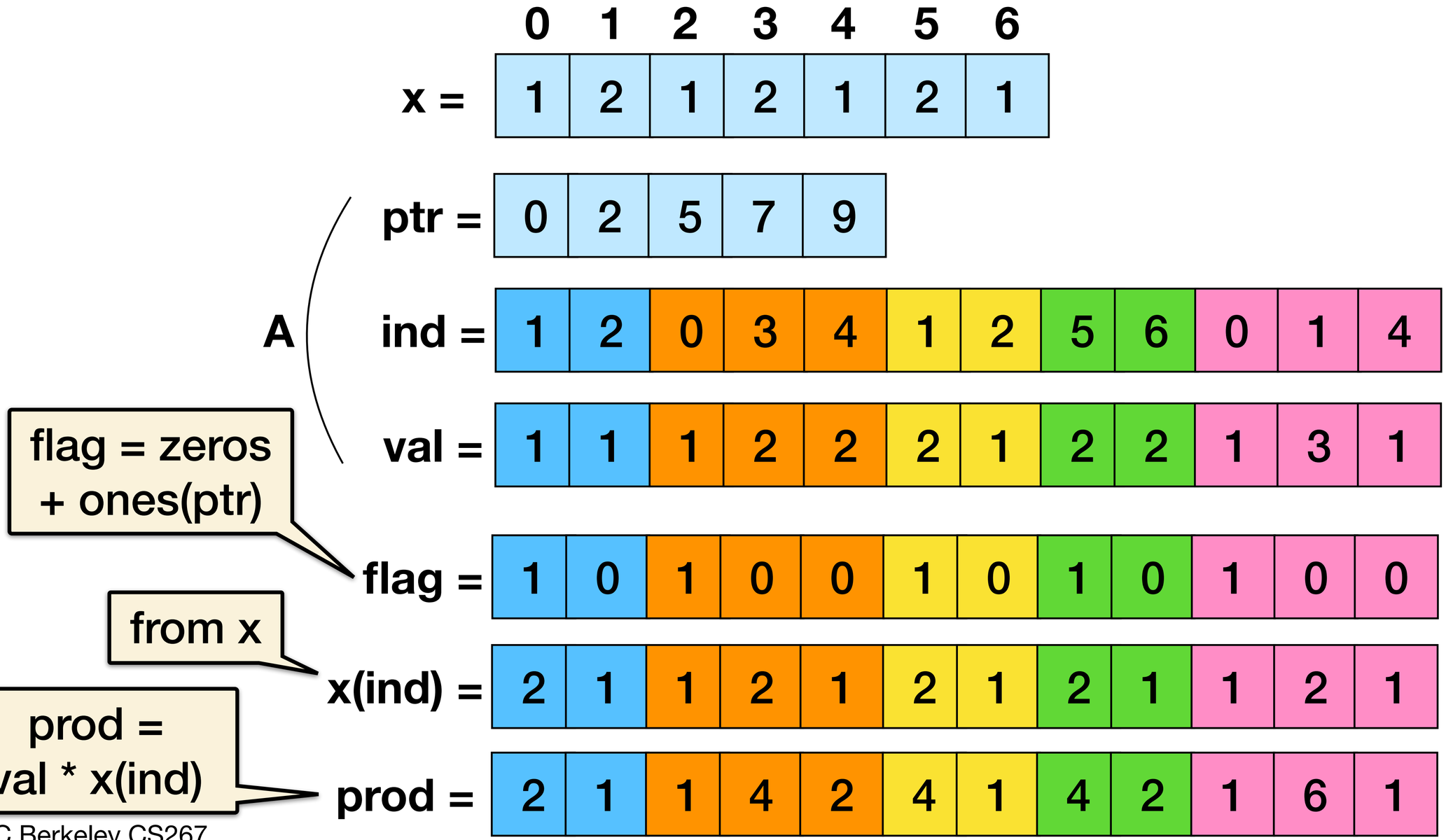
Parallel SpMV in CSR

```
parallel_for (int i = 0; i < m; i++) {
  for (int j = ptr[i]; j < ptr[i+1]; j++) {
    tmp = ind[j];
    y[i] += val[j] * x[tmp]
  }
}
```

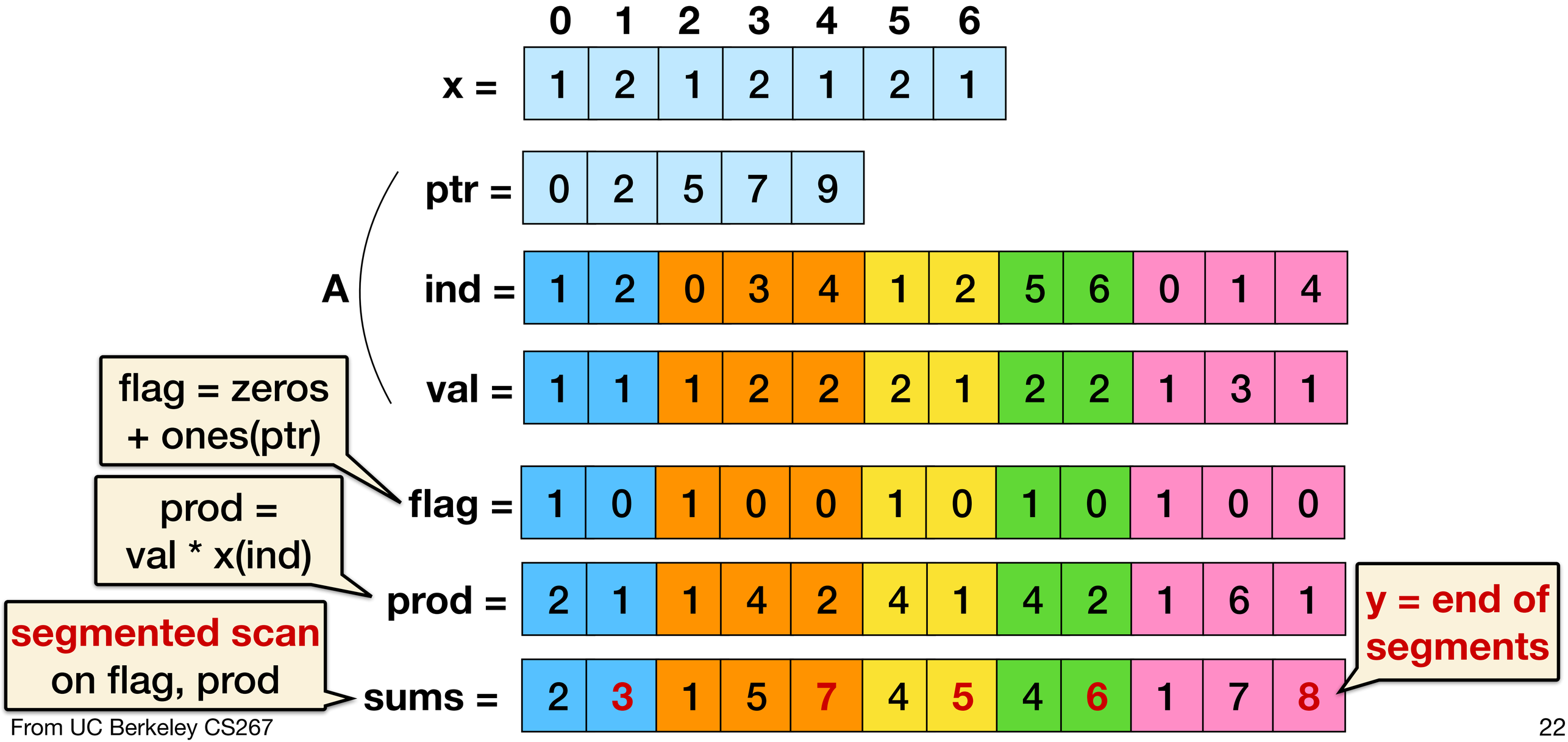


From UC Berkeley CS267

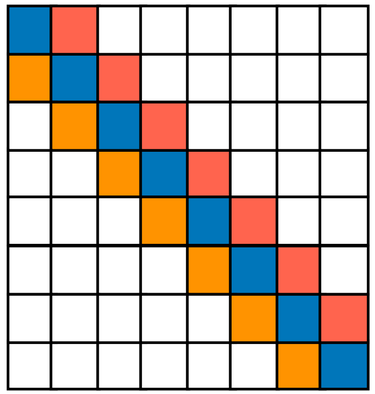
Segmented Suffix Scan

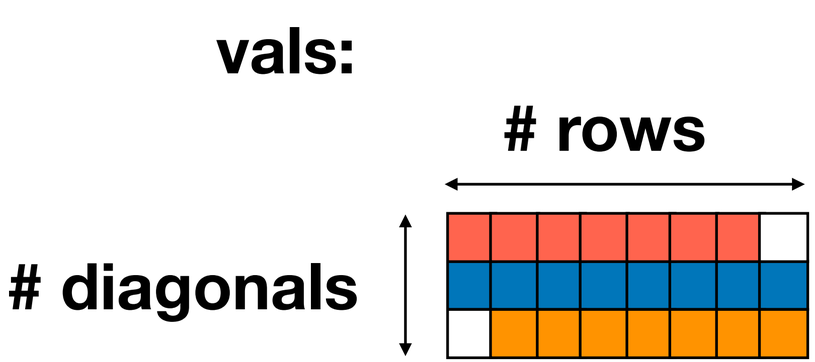


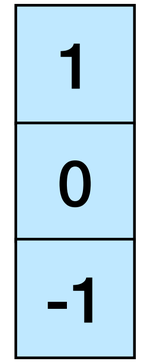
Segmented Suffix Scan



Parallel SpMV in Diagonal Format



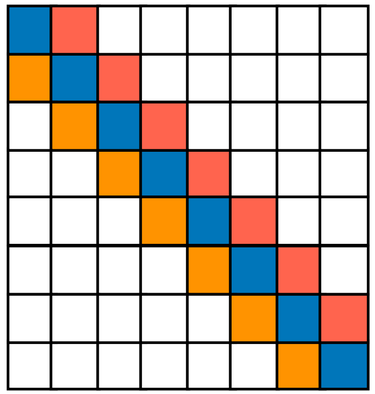




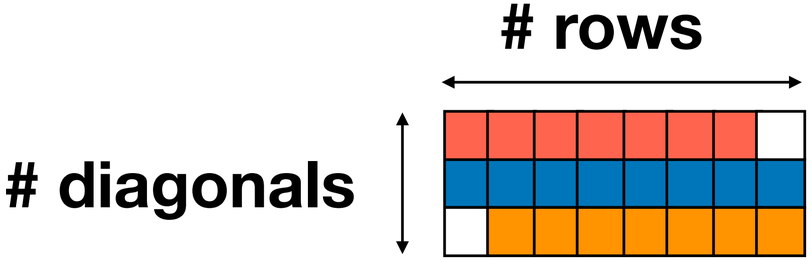
columnoffset

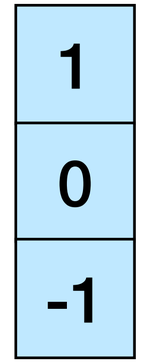
```
for each diagonal k do
  parallel_for each row i do
    column = i + columnoffset[k]
    if (column >= 0 && column < n)
      y[i] = y[i] + val[k][column] * x[column]
```

Parallel SpMV in Diagonal Format



vals:





columnoffset

```
for each diagonal k do
  parallel_for each row i do
    column = i + columnoffset[k]
    if (column >= 0 && column < n)
      y[i] = y[i] + val[k][column] * x[column]
```

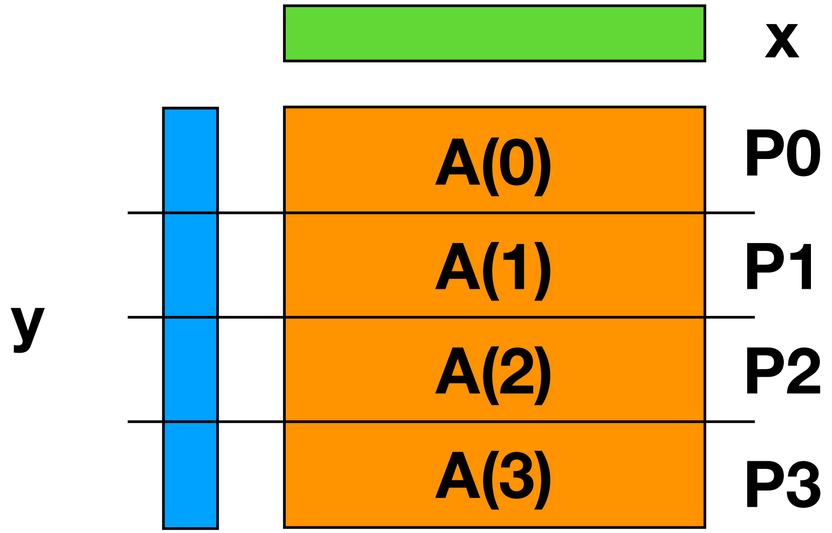
Diagonal is also popular for GPUs/vectors

Distributed SpMV

Distributed Dense Matrix-Vector Product (Row Major)

warmup on dense case

- Compute y = y + A*x, where A is a dense matrix
- Layout: **1D row blocked**
- Algorithm:



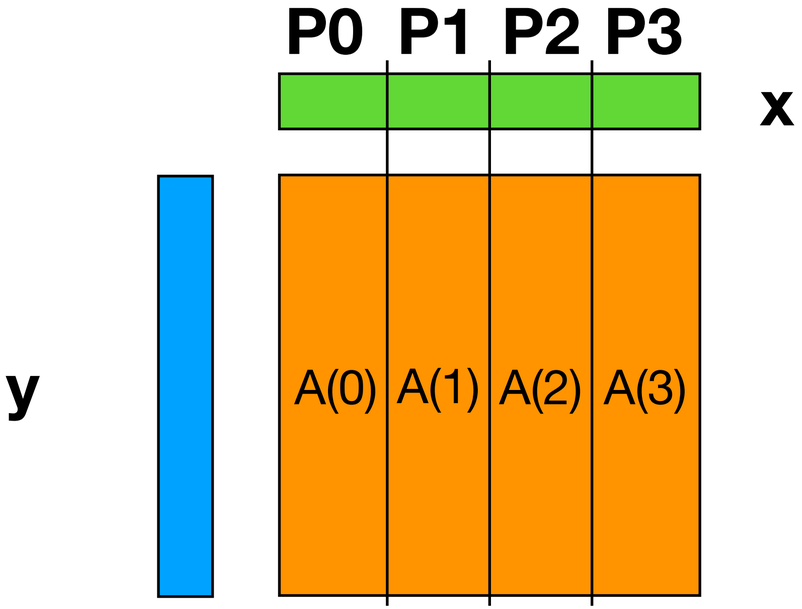

```
Allgather x

for all local i
  for all j
    compute y[i] += A[i, j]*x[j] locally
```

Broadcast + local dot product

Distributed Dense Matrix-Vector Product (Column Major)

- Compute y = y + A*x, where A is a dense matrix
- Layout: **1D column blocked**
- Algorithm:




```
foreach processor p
  make a temp vector of size n
  for all local j
    for all i
      compute temp[i] += A[i, j]*x[j] locally

  y[i] += SumReduce(temp[i])
```

Local daxpy and parallel reduction

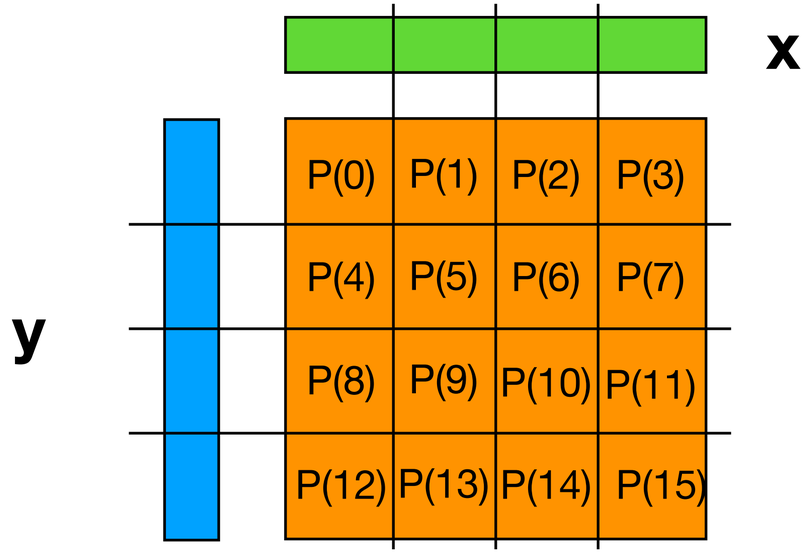
Reduce across all rows

Distributed Dense Matrix-Vector Product (2D Blocked)

A 2D blocked layout uses a broadcast of x and reduction into y.

Both on a subset of processors

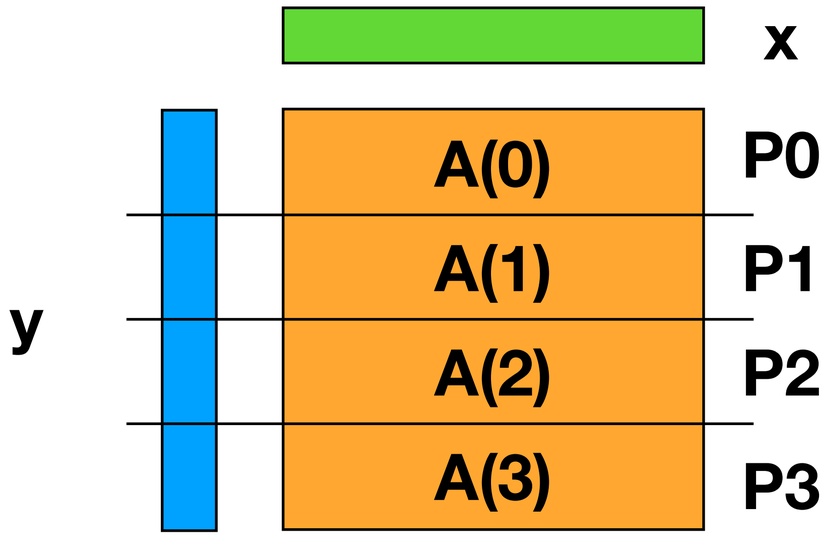
- sqrt(p) for square processor grid
- Can use other rectangular shapes p_row * p_col = p



Distributed SpMV

Row parallelism (y & A partitioned)

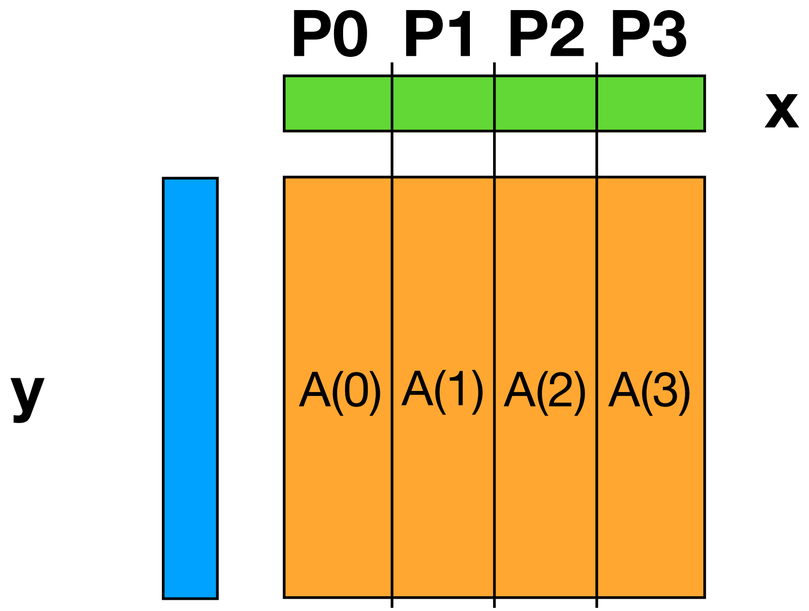
- Replicate x across processors
- Or exchange only necessary elements
- Are nonzeroes clustered, e.g., near the diagonal?



Distributed SpMV

Column parallelism (x & A partitioned)

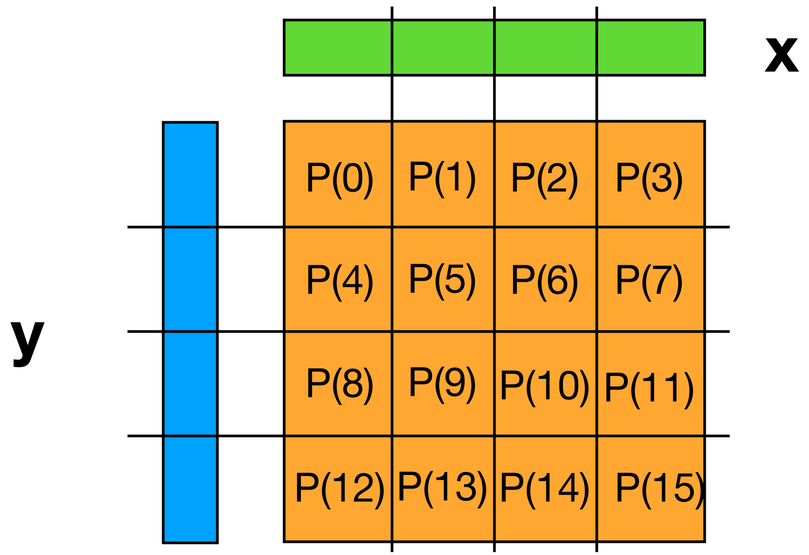
- Make temporary `temp_y = [0,…]` on all processors;
- Update that; and (sparse?) sum-reduce over processors



Distributed SpMV

2D parallelism for large p and **when nonzeros are uniform**

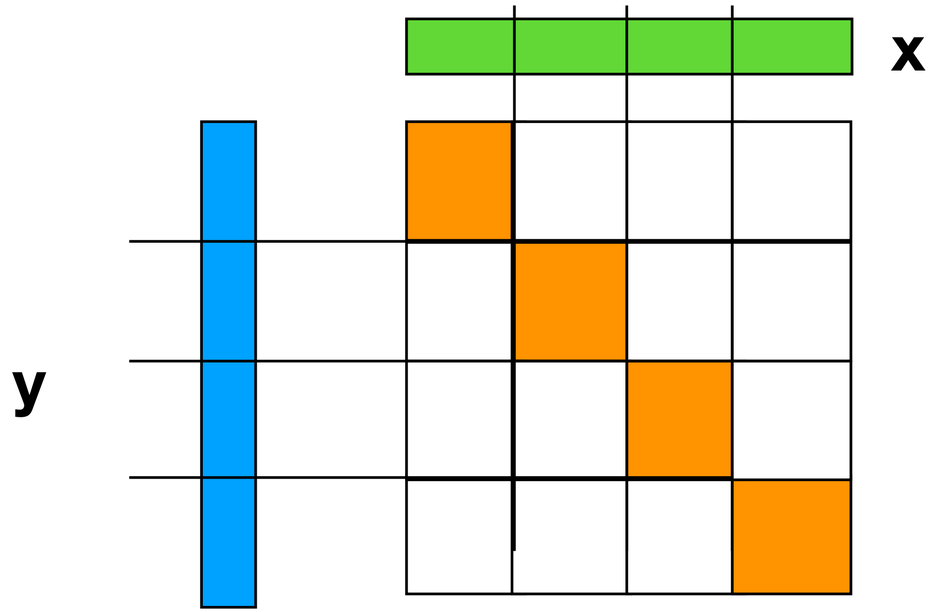
- Divide processors into p1 x p2 (e.g., square grid)
- Hybrid of Row and Column parallelism using teams
- NAS CG benchmark does this (random nonzero pattern)
- Bad load balance for clustered nonzeros



Ideal Sparse Structure: P Diagonal Blocks

"Ideal" matrix structure for parallelism: block diagonal

- If p_i holds x_i and y_i blocks, no vectors to communicate
- If no nonzeroes outside these blocks, no communication is needed!



Dream scenario: reorder rows/columns to get close to this - most nonzeroes in diagonal blocks, few outside.

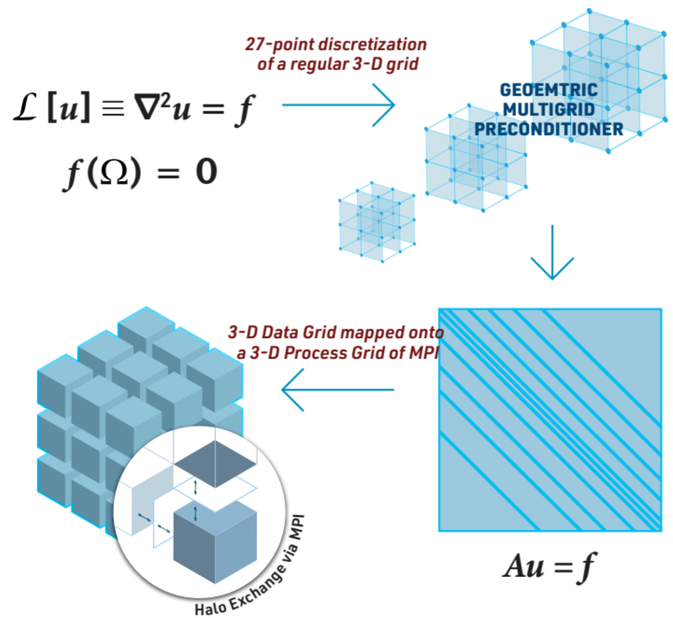
High Performance Conjugate Gradients (HPCG) Benchmark

Complement to LINPACK (dense linear algebra), which is used for the TOP500.

Designed to exercise computational and data access patterns that more closely match a **different and broader set of applications**.

HPCG includes performance of the following basic operations:

- Conjugate gradient (27-point stencil)
- Sparse matrix-vector multiplication
- Global dot product
- Vector update
- and others



https://www.hpcg-benchmark.org/

November 2023 HPCG Results

New HPCG results were announced at SC23

Rank	Site	Computer	Cores	HPL Rmax (Pflop/s)	TOP500 Rank	HPCG (Pflop/s)	Fraction of Peak
1	RIKEN Center for Computational Science **Japan**	**Supercomputer Fugaku** — A64FX 48C 2.2GHz, Tofu interconnect D	7,630,848	442.01	4	16.00	3.0%
2	DOE/SC/Oak Ridge National Laboratory **United States**	**Frontier** — AMD Optimized 3rd Generation EPYC 64C 2GHz, Slingshot-11, AMD Instinct MI250X	8,699,904	1194.00	1	14.05	0.8%
3	EuroHPC/CSC **Finland**	**LUMI** — AMD Optimized 3rd Generation EPYC 64C 2GHz, Slingshot-11, AMD Instinct MI250X	2,752,704	379.70	5	4.587	0.9%
4	EuroHPC/CINECA **Italy**	**Leonardo** — Xeon Platinum 8358 32C 2.6GHz, Quad-rail NVIDIA HDR100 Infiniband, NVIDIA A100 SXM4 64 GB	1,824,768	238.70	6	3.114	1.0%
5	DOE/SC/Oak Ridge National Laboratory **United States**	**Summit** — IBM POWER9 22C 3.07GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Volta GV100	2,414,592	148.60	7	2.926	1.5%
6	DOE/SC/LBNL/NERSC **United States**	**Perlmutter** — AMD EPYC 7763 64C 2.45GHz, Slingshot-11, NVIDIA A100 SXM4 40 GB	888,832	79.23	12	1.905	1.7%
7	DOE/NNSA/LLNL **United States**	**Sierra** — IBM POWER9 22C 3.1GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Volta GV100	1,572,480	94.64	10	1.796	1.4%
8	NVIDIA Corporation **United States**	**Selene** — AMD EPYC 7742 64C 2.25GHz, Mellanox HDR Infiniband, NVIDIA A100	555,520	63.46	13	1.623	2.0%
9	Forschungszentrum Juelich (FZJ) **Germany**	**JUWELS Booster Module** — AMD EPYC 7402 24C 2.8GHz, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, NVIDIA A100	449,280	44.12	18	1.275	1.8%
10	Cyberscience Center, Tohoku University **Japan**	**AOBA-S** — Vector Engine Type 30A 16C 1.6GHz, Infiniband NDR200	64,512	17.22	50	1.089	5.5%

Not the same order as TOP500

From UC Berkeley CS267

November 2023 HPCG Results

https://www.netlib.org/benchmark/hpl/

New HPCG results were announced at SC23

Rank	Site	Computer	Cores	HPL Rmax (Pflop/s)	TOP500 Rank	HPCG (Pflop/s)	Fraction of Peak
1	RIKEN Center for Computational Science **Japan**	**Supercomputer Fugaku** — A64FX 48C 2.2GHz, Tofu interconnect D	7,630,848	442.01	4	16.00	3.0%
2	DOE/SC/Oak Ridge National Laboratory **United States**	**Frontier** — AMD Optimized 3rd Generation EPYC 64C 2GHz, Slingshot-11, AMD Instinct MI250X	8,699,904	1194.00	1	14.05	0.8%
3	EuroHPC/CSC **Finland**	**LUMI** — AMD Optimized 3rd Generation EPYC 64C 2GHz, Slingshot-11, AMD Instinct MI250X	2,752,704	379.70	5	4.587	0.9%
4	EuroHPC/CINECA **Italy**	**Leonardo** — Xeon Platinum 8358 32C 2.6GHz, Quad-rail NVIDIA HDR100 Infiniband, NVIDIA A100 SXM4 64 GB	1,824,768	238.70	6	3.114	1.0%
5	DOE/SC/Oak Ridge National Laboratory **United States**	**Summit** — IBM POWER9 22C 3.07GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Volta GV100	2,414,592	148.60	7	2.926	1.5%
6	DOE/SC/LBNL/NERSC **United States**	**Perlmutter** — AMD EPYC 7763 64C 2.45GHz, Slingshot-11, NVIDIA A100 SXM4 40 GB	888,832	79.23	12	1.905	1.7%
7	DOE/NNSA/LLNL **United States**	**Sierra** — IBM POWER9 22C 3.1GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Volta GV100	1,572,480	94.64	10	1.796	1.4%
8	NVIDIA Corporation **United States**	**Selene** — AMD EPYC 7742 64C 2.25GHz, Mellanox HDR Infiniband, NVIDIA A100	555,520	63.46	13	1.623	2.0%
9	Forschungszentrum Juelich (FZJ) **Germany**	**JUWELS Booster Module** — AMD EPYC 7402 24C 2.8GHz, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, NVIDIA A100	449,280	44.12	18	1.275	1.8%
10	Cyberscience Center, Tohoku University **Japan**	**AOBA-S** — Vector Engine Type 30A 16C 1.6GHz, Infiniband NDR200	64,512	17.22	50	1.089	5.5%

Much lower Pflop/s compared to Linpack

From UC Berkeley CS267

November 2023 HPCG Results

New HPCG results were announced at SC23

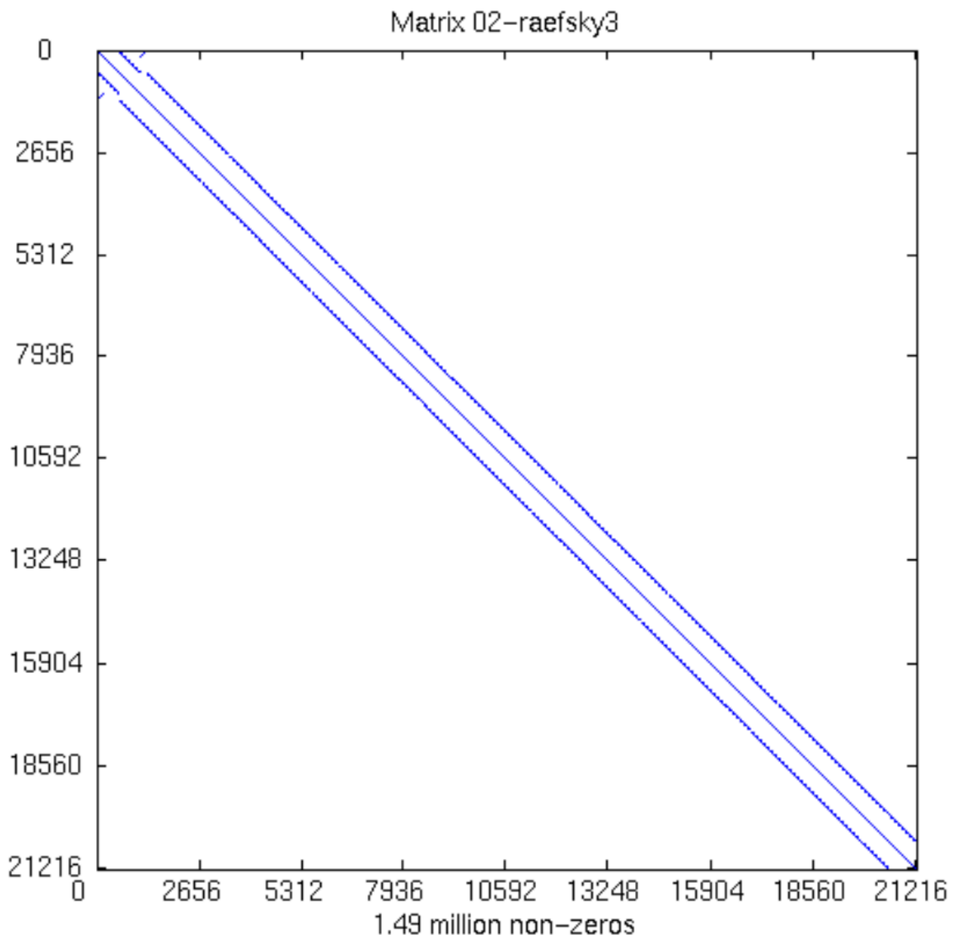
Rank	Site	Computer	Cores	HPL Rmax (Pflop/s)	TOP500 Rank	HPCG (Pflop/s)	Fraction of Peak
1	RIKEN Center for Computational Science **Japan**	**Supercomputer Fugaku** — A64FX 48C 2.2GHz, Tofu interconnect D	7,630,848	442.01	4	16.00	3.0%
2	DOE/SC/Oak Ridge National Laboratory **United States**	**Frontier** — AMD Optimized 3rd Generation EPYC 64C 2GHz, Slingshot-11, AMD Instinct MI250X	8,699,904	1194.00	1	14.05	0.8%
3	EuroHPC/CSC **Finland**	**LUMI** — AMD Optimized 3rd Generation EPYC 64C 2GHz, Slingshot-11, AMD Instinct MI250X	2,752,704	379.70	5	4.587	0.9%
4	EuroHPC/CINECA **Italy**	**Leonardo** — Xeon Platinum 8358 32C 2.6GHz, Quad-rail NVIDIA HDR100 Infiniband, NVIDIA A100 SXM4 64 GB	1,824,768	238.70	6	3.114	1.0%
5	DOE/SC/Oak Ridge National Laboratory **United States**	**Summit** — IBM POWER9 22C 3.07GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Volta GV100	2,414,592	148.60	7	2.926	1.5%
6	DOE/SC/LBNL/NERSC **United States**	**Perlmutter** — AMD EPYC 7763 64C 2.45GHz, Slingshot-11, NVIDIA A100 SXM4 40 GB	888,832	79.23	12	1.905	1.7%
7	DOE/NNSA/LLNL **United States**	**Sierra** — IBM POWER9 22C 3.1GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Volta GV100	1,572,480	94.64	10	1.796	1.4%
8	NVIDIA Corporation **United States**	**Selene** — AMD EPYC 7742 64C 2.25GHz, Mellanox HDR Infiniband, NVIDIA A100	555,520	63.46	13	1.623	2.0%
9	Forschungszentrum Juelich (FZJ) **Germany**	**JUWELS Booster Module** — AMD EPYC 7402 24C 2.8GHz, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, NVIDIA A100	449,280	44.12	18	1.275	1.8%
10	Cyberscience Center, Tohoku University **Japan**	**AOBA-S** — Vector Engine Type 30A 16C 1.6GHz, Infiniband NDR200	64,512	17.22	50	1.089	5.5%

Small fraction of peak

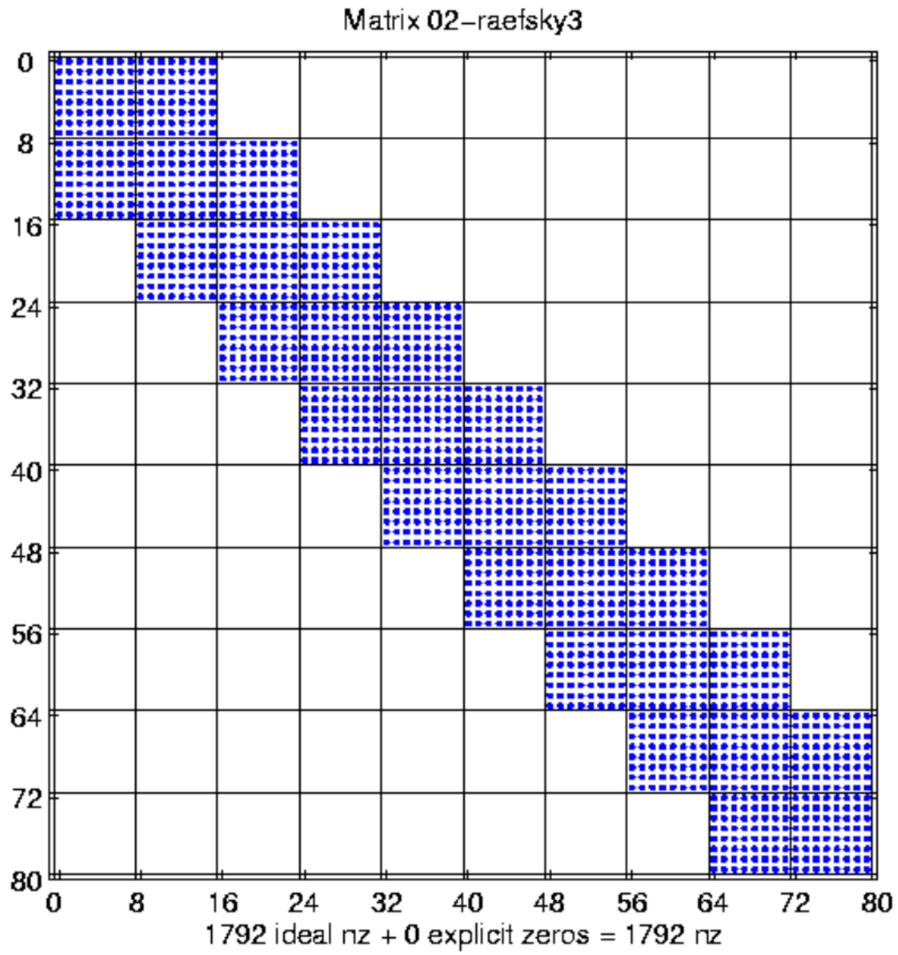
From UC Berkeley CS267

Register/cache blocking and autotuning SpMV

Changing Matrix Format: Register Blocking



Submatrix



"Fast sparse matrix-vector multiplication by exploiting variable block structure," Vuduc and Moon 2005.

Using block structure in SpMV

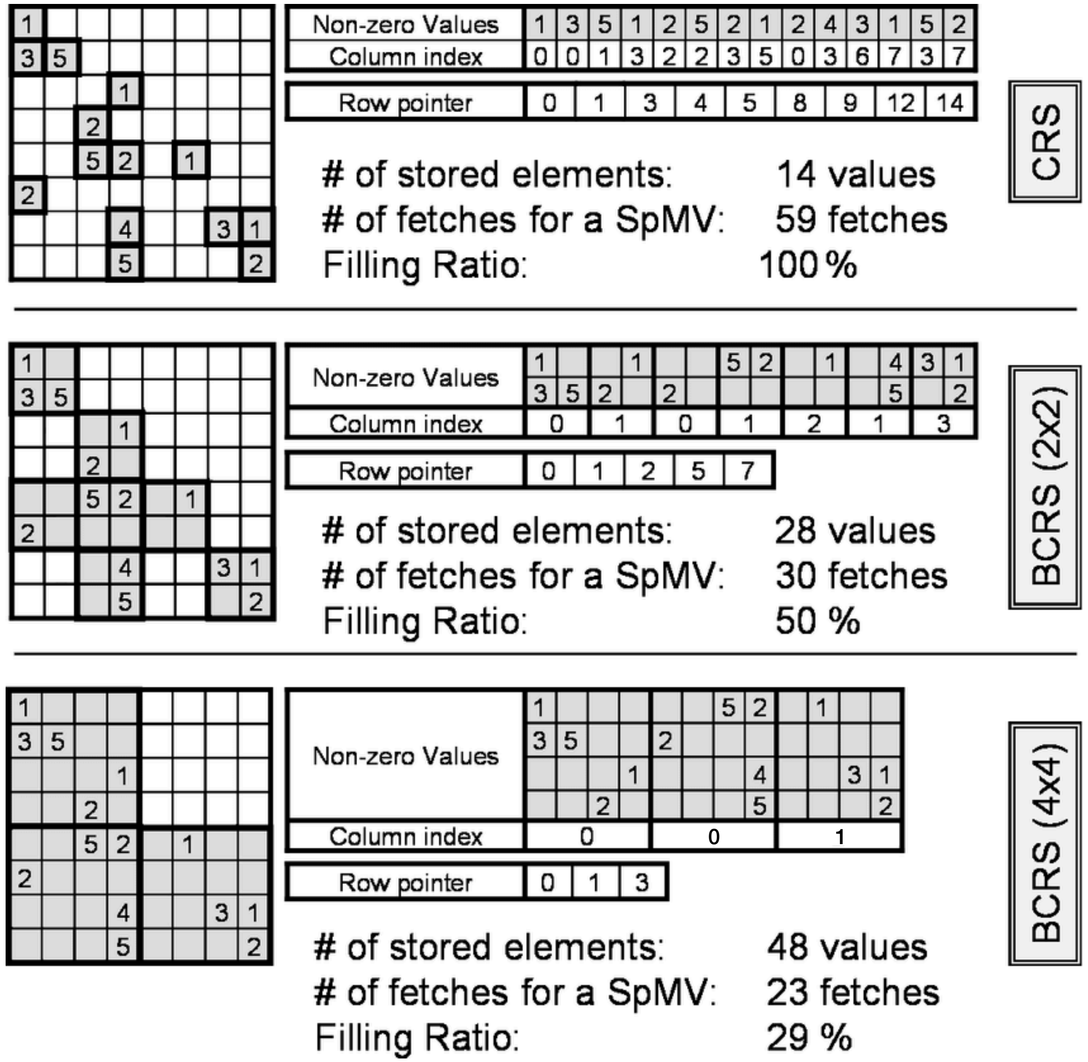
The **bottleneck is the time to fetch** the matrix from memory

- Only 2 flops for each nz in matrix
- Fetching at ~1 int (col idx) + 1 float (value) for 2 flops

Blocked Compressed Sparse Row (BCSR)

- Don't store each nonzero - instead, **store each nonzero r x c block** with 1 column index
- Time to fetch matrix from memory decreases

Change both data structure and algorithm - need to pick r and c and change algorithm accordingly



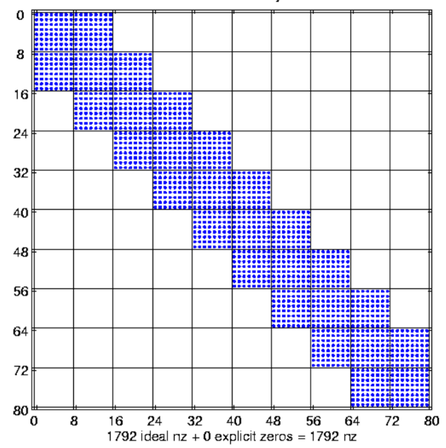
https://www.researchgate.net/figure/Compressed-Row-Storage-CRS-and-Block-Compressed-Row-Storage-BCRS-examples-for-the_fig1_29617855

From UC Berkeley CS267

The need for search

In the previous example, it seems like 8x8 is a natural choice.

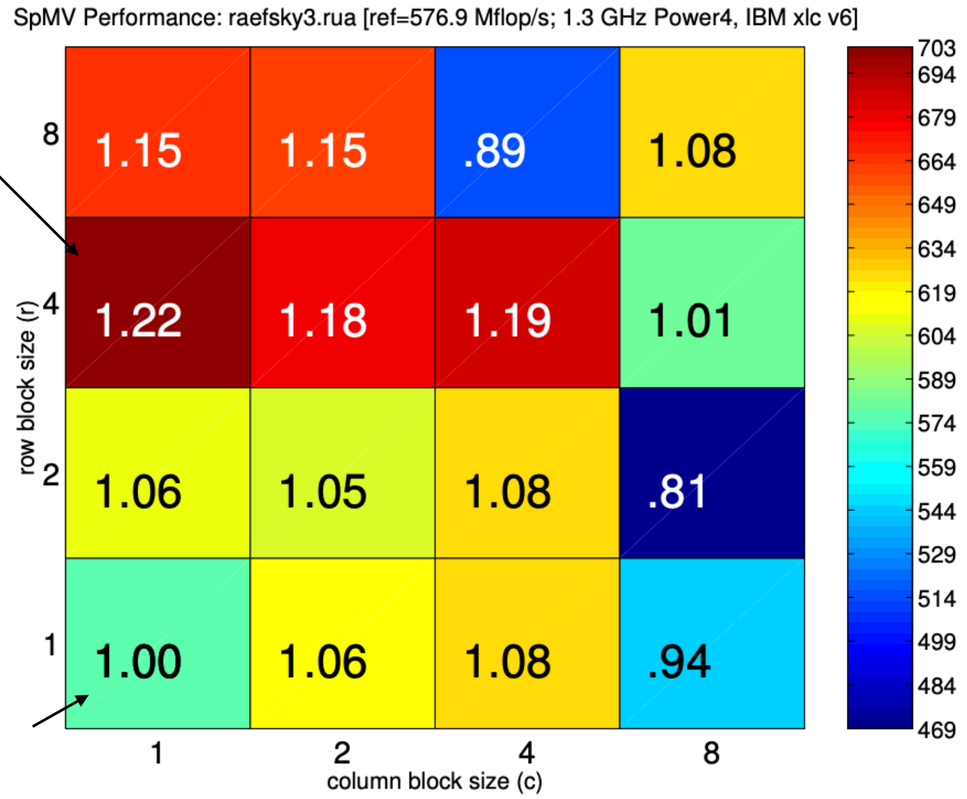
Out of 6 platforms tested, 8x8 is the best on one (coming up)

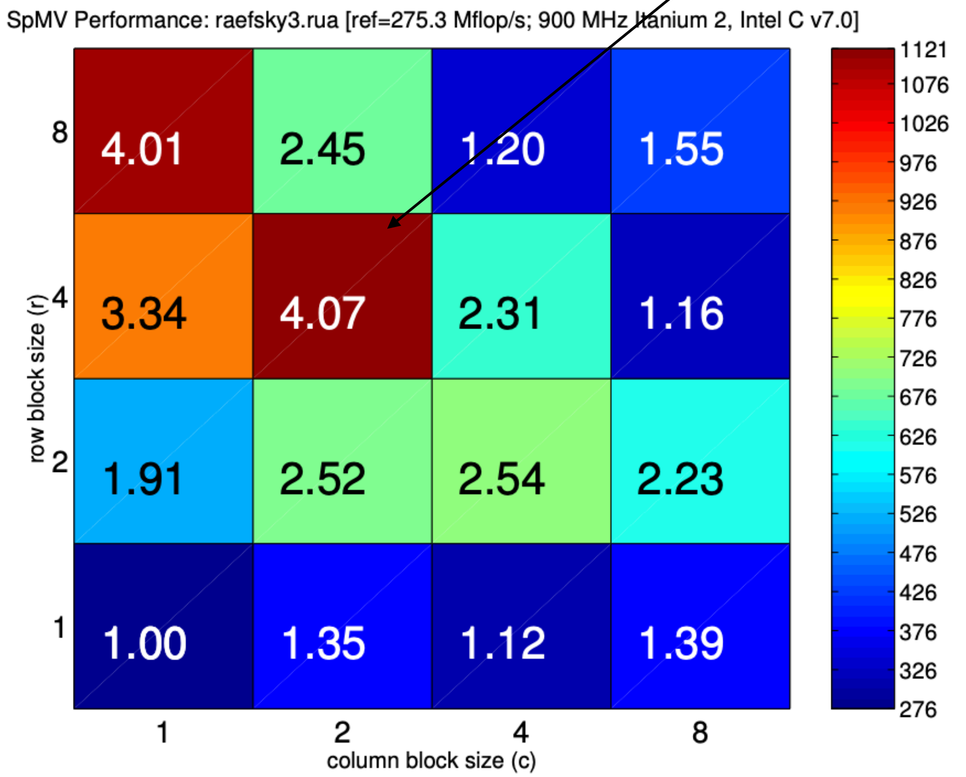


Best: 4x2

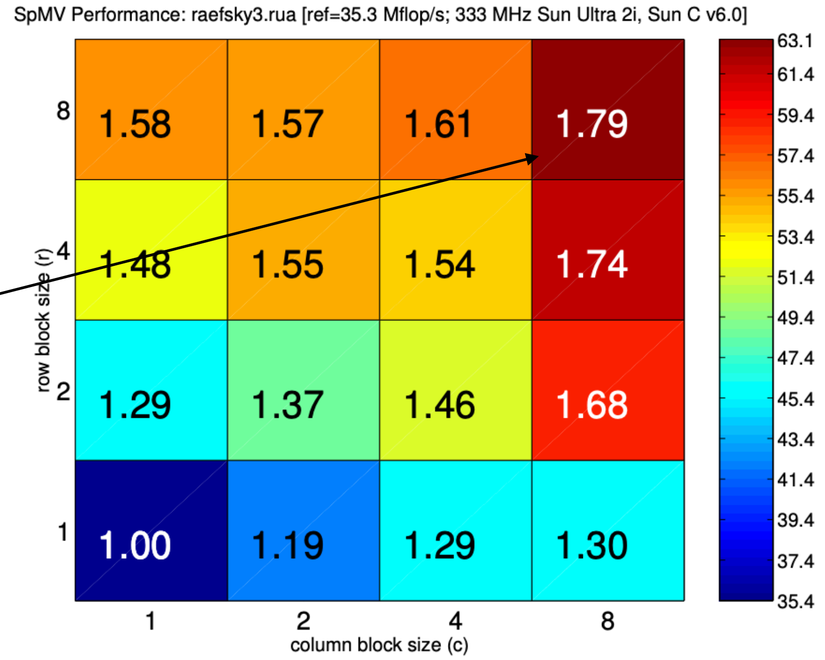
Best: 4x1

Reference

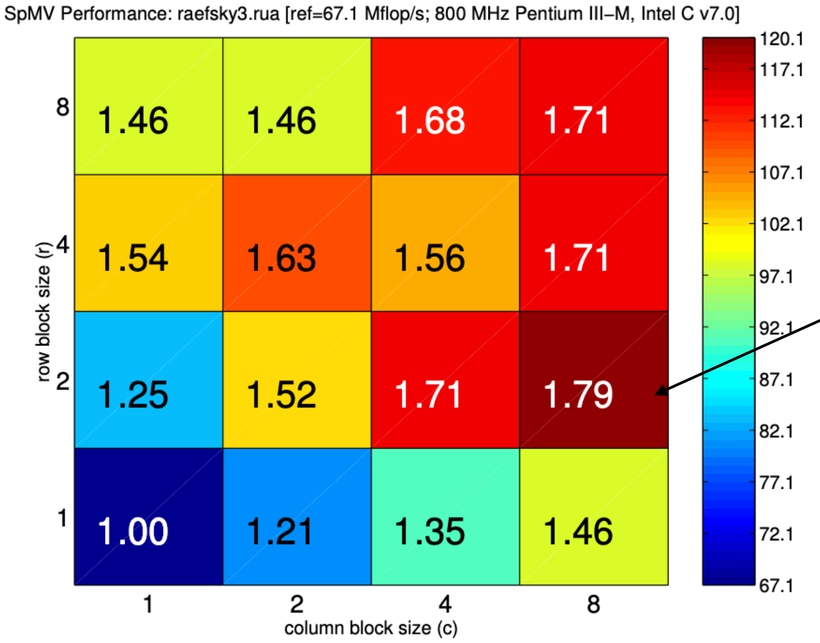




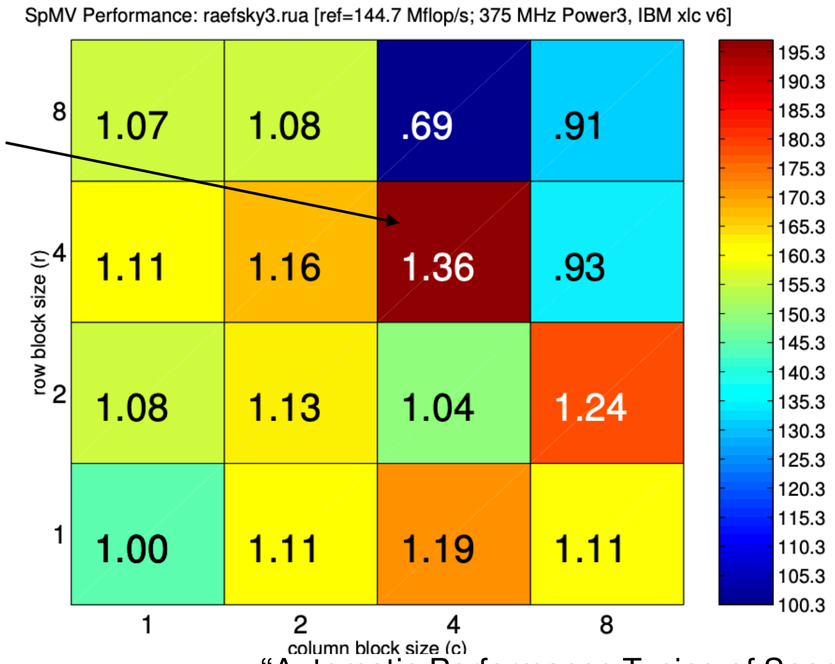
"Automatic Performance Tuning of Sparse Matrix Kernels," Vuduc, 2003.



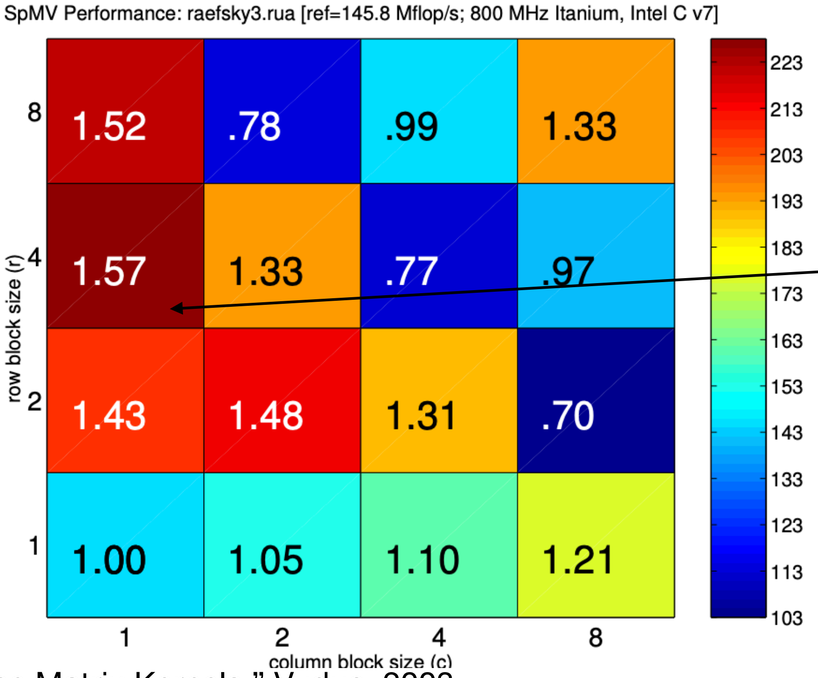
Best: 8x8



Best: 2x8



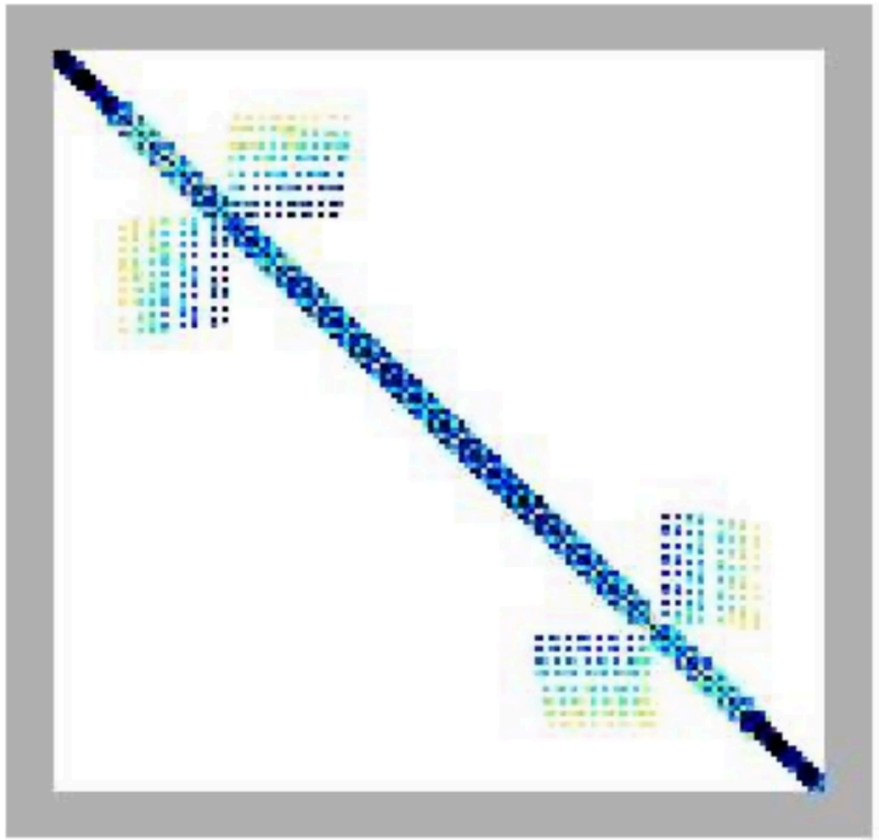
Best: 4x4



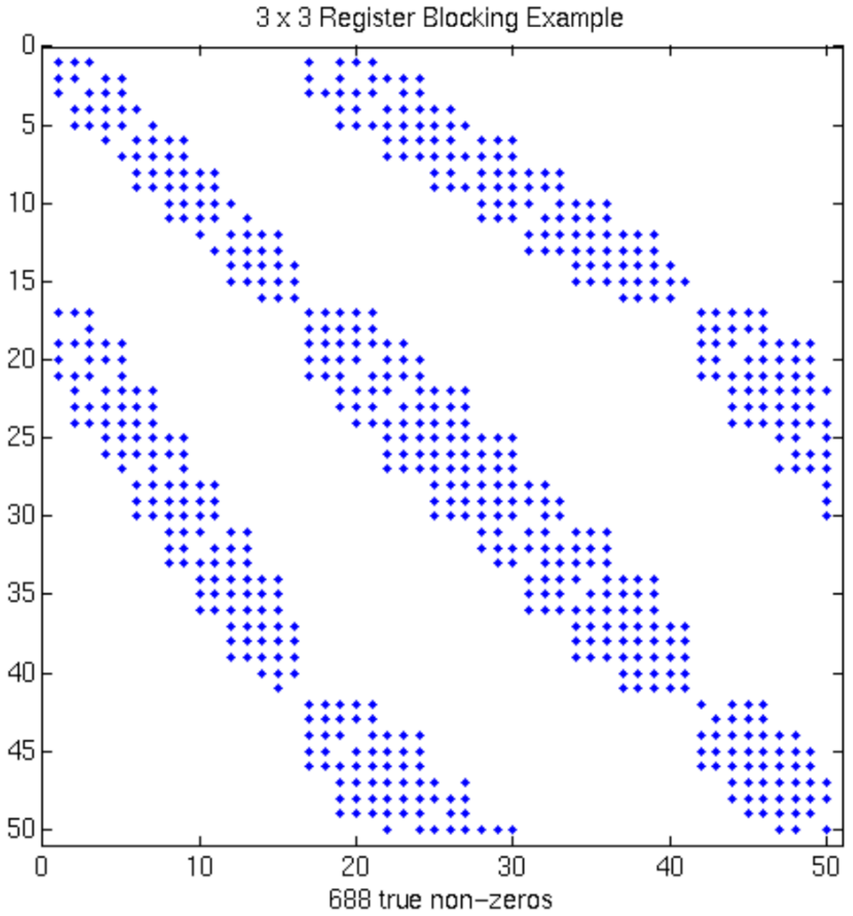
Best: 4x1

"Automatic Performance Tuning of Sparse Matrix Kernels," Vuduc, 2003.

But most matrices don’t block so easily



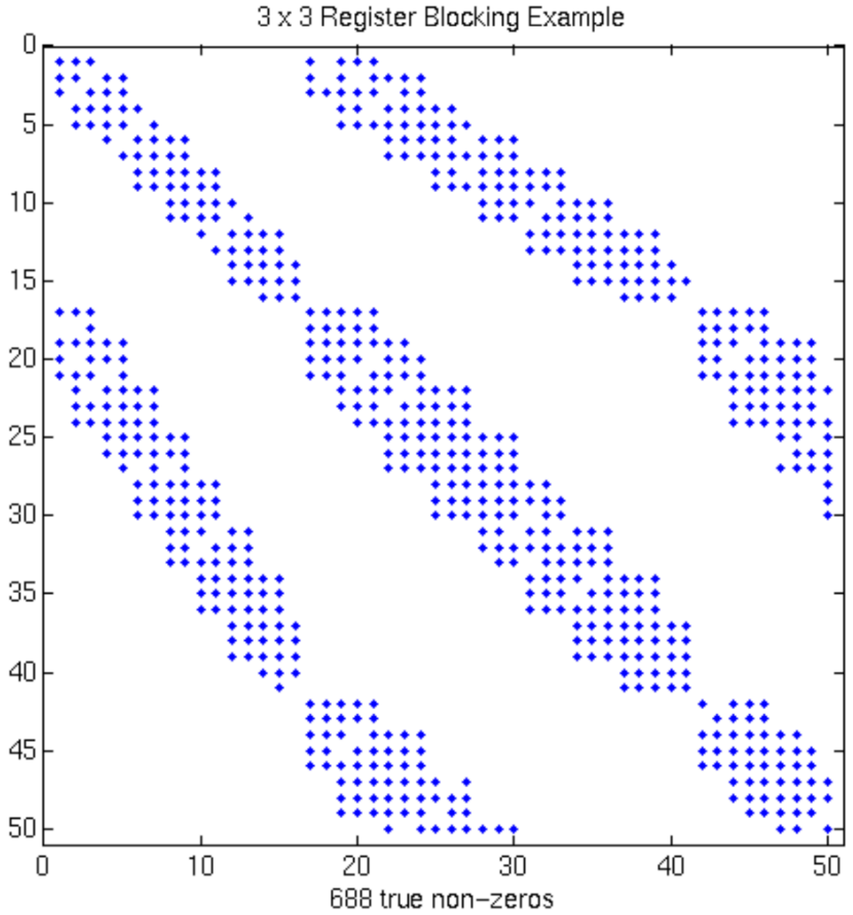
Zoom into top corner



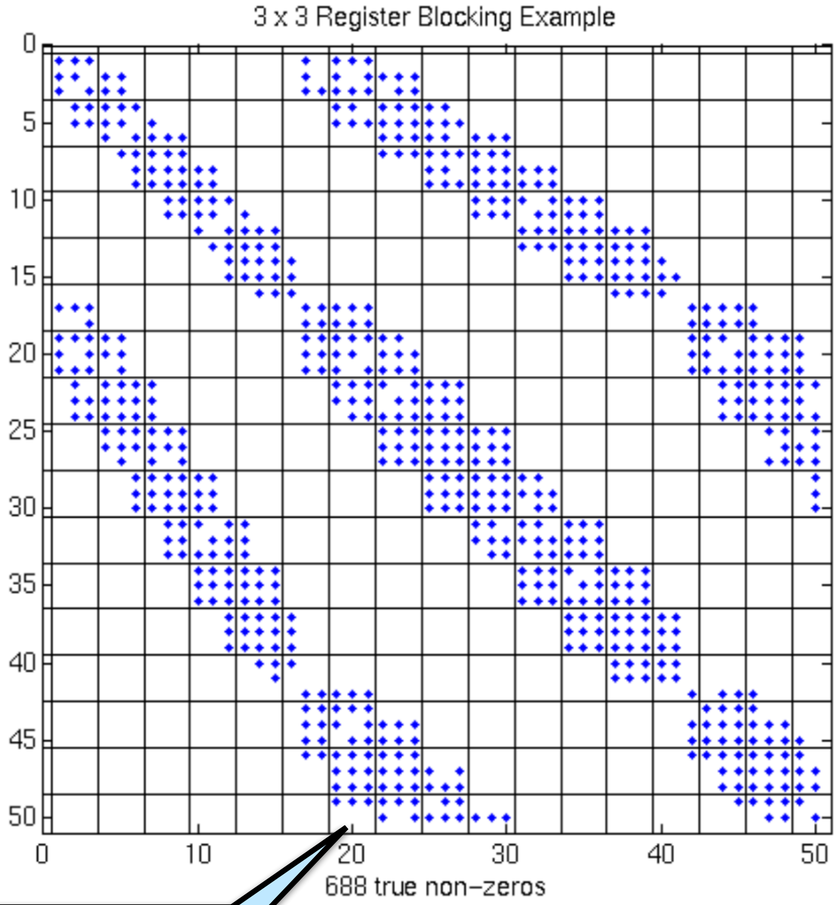
Fluid dynamics problem - more complicated nonzero structure
Total nnz = 1.1M

From UC Berkeley CS267

3x3 blocks look natural, but…



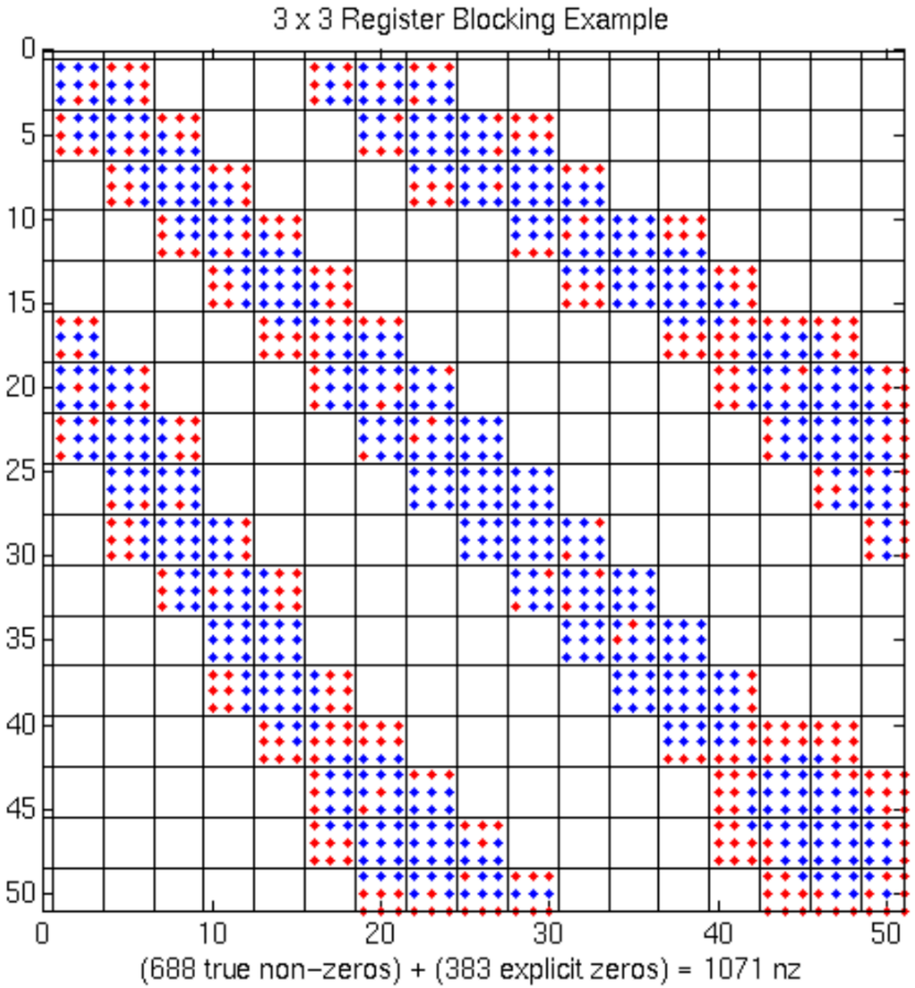
3x3 blocks



Many blocks are not full

From UC Berkeley CS267

Extra work can improve efficiency



- Add explicit zeroes: 1.5x “fill overhead”
- Unroll loops
- More work but faster - 1.5x faster on PIII

From UC Berkeley CS267

Libraries for sparse matrices

How to build optimized library when:

- Formats are not known? Libraries like PETSc and Trilinos will let the user provide format and SpMV

How to build optimized matrix kernel library (BLAS)?

- Nonzero structure is key to optimization

OSKI = Optimized Sparse Kernel Interface

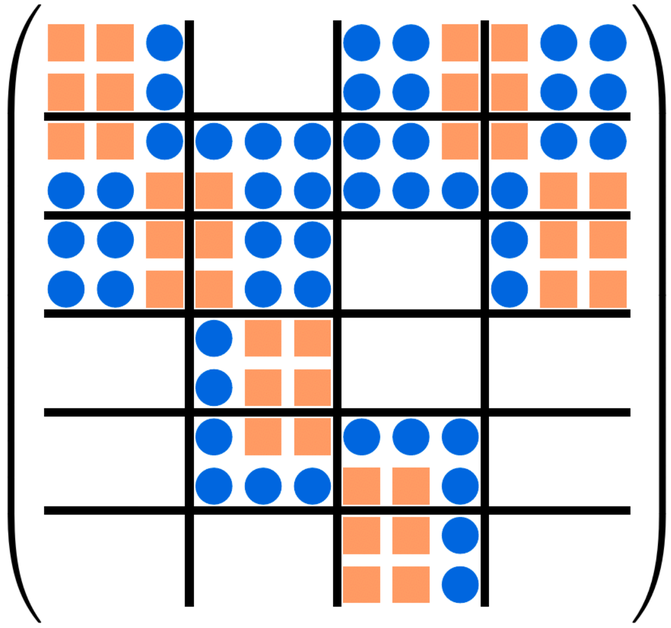
- pOSKI for multicore

Using the Fill to Formalize Blocking Scheme Quality [Im & Yelick, 01]

The fill of a *N*-dimensional tensor *A* is defined with respect to

- the number of nonzeros $k(A)$
- a blocking $\mathbf{b} = (b_1, \ldots, b_N)$
- the number of nonempty blocks $k_{\mathbf{b}}(A)$

$$f_{\mathbf{b}}(A) = \frac{b_1 b_2 \ldots b_N k_{\mathbf{b}}(A)}{k(A)}$$



Using the Fill to Formalize Blocking Scheme Quality [Im & Yelick, 01]

The fill of a *N*-dimensional tensor *A* is defined with respect to
- the number of nonzeros $k(A)$
- a blocking $\mathbf{b} = (b_1, \ldots, b_N)$
- the number of nonempty blocks $k_{\mathbf{b}}(A)$

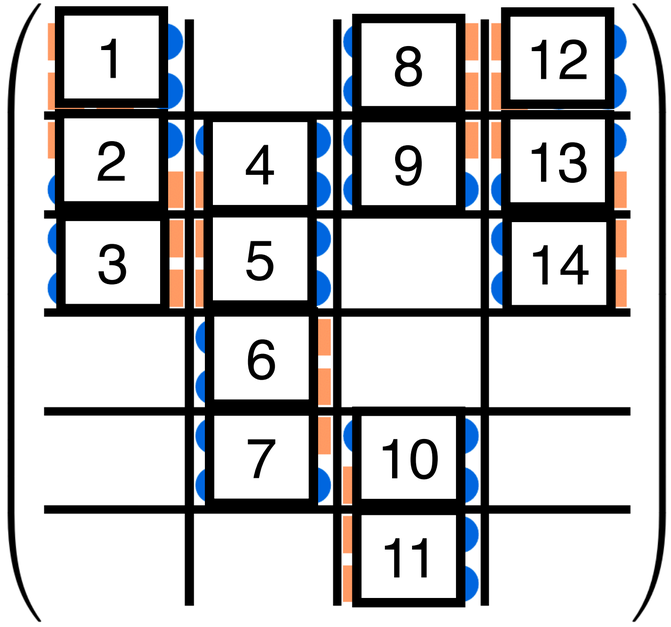
$$f_{\mathbf{b}}(A) = \frac{b_1 b_2 \ldots b_N k_{\mathbf{b}}(A)}{k(A)}$$

$$= \frac{2 \times 3 \times 14}{36} \approx 2.33$$

Nonempty blocks

Nonzeros

Greater ~ more wasted space

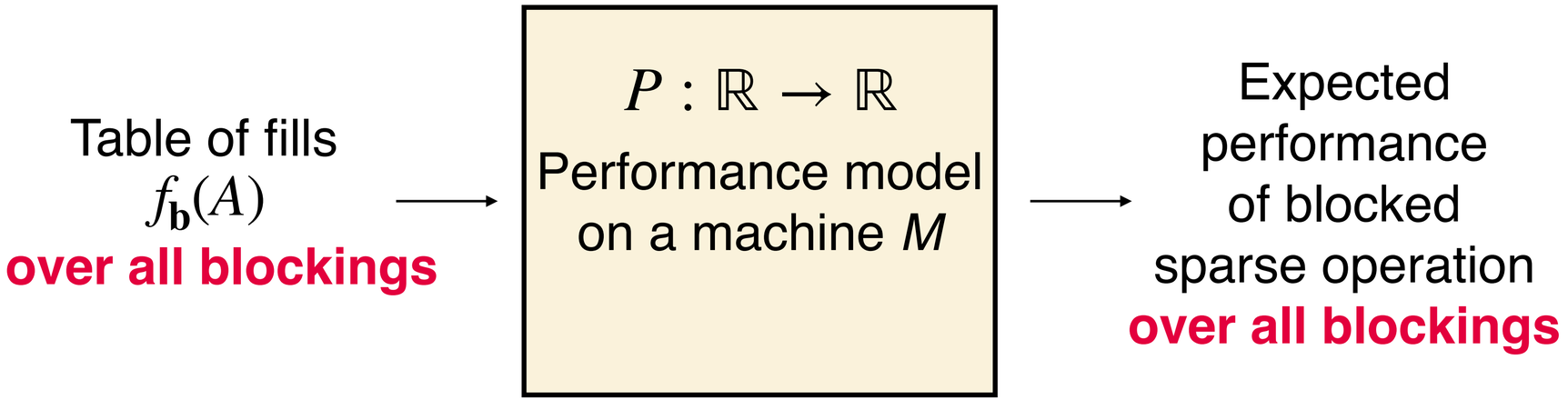


Blocking $\mathbf{b}$

$b_1 = 2$
$b_2 = 3$
$k_{\mathbf{b}} = 14$
$k(A) = 36$

Performance Modeling Using the Fill

A **performance model** is a function that maps the fill under a blocking $\mathbf{b}$ to expected performance in FLOP/s.



Block size selection chooses the block size that maximizes performance based on a performance model.

"Optimizing the Performance of Sparse Matrix-Vector Multiplication," Im, 2000.
"Automatic Performance Tuning of Sparse Matrix Kernels," Vuduc, 2003.

Example: SPARSITY Performance Model for Blocked SpMV [Vuduc et al., 02]

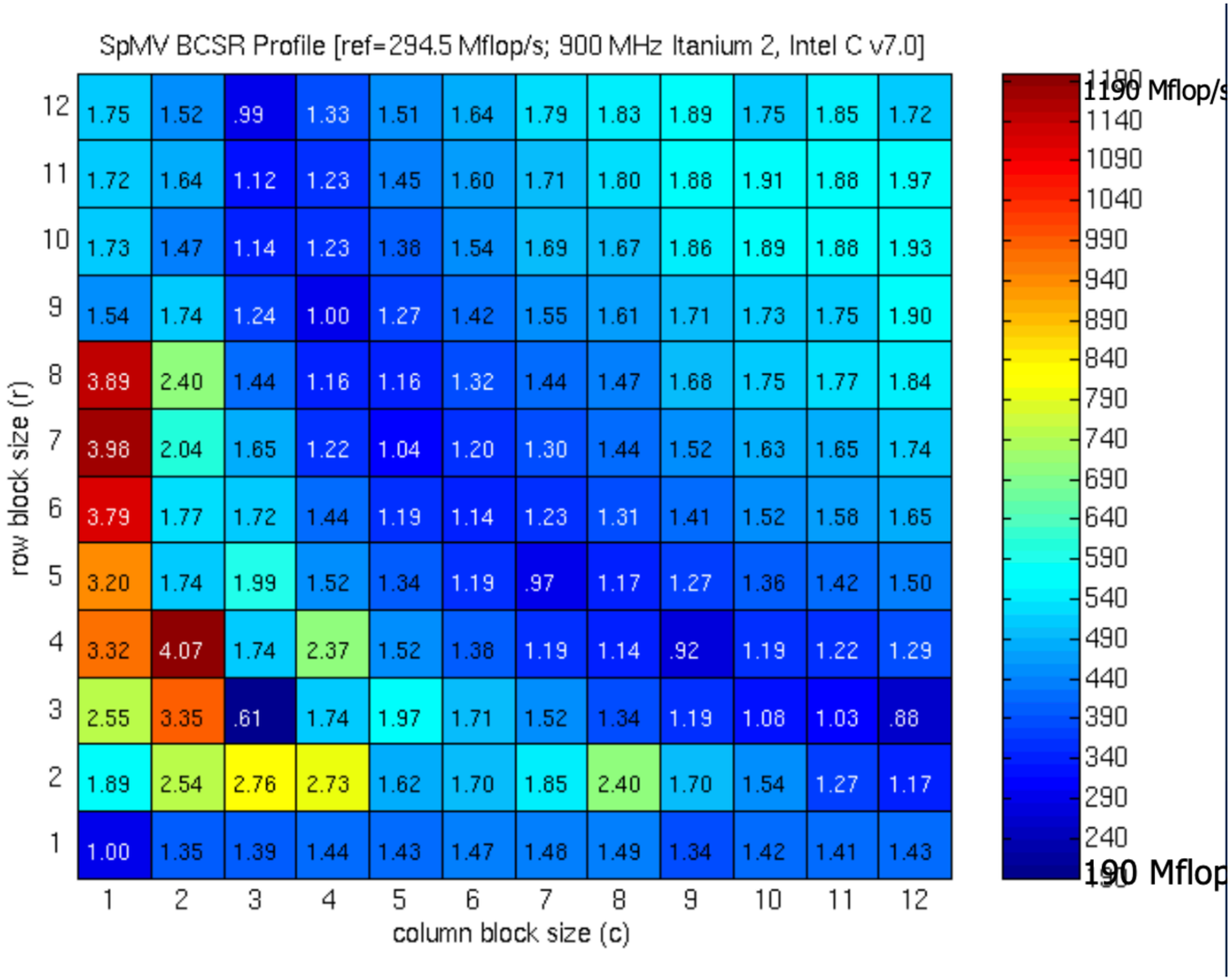
The SPARSITY performance model uses a matrix $\mathbf{PERF}(\mathbf{b})$ of the performance of a machine *M* (in FLOP/s) on a dense matrix stored with blocking scheme $\mathbf{b}$.

Table of fills $f_{\mathbf{b}}(A)$ **over all blockings** → $P(\mathbf{b}, A) = \dfrac{\mathbf{PERF_b}}{f_{\mathbf{b}}(A)}$ → Expected performance of blocked SpMV **over all blockings**

Vuduc et al. show that when the fill was known exactly, performance of the resulting blocking scheme was **optimal or near optimal** (within 5%) [V04]

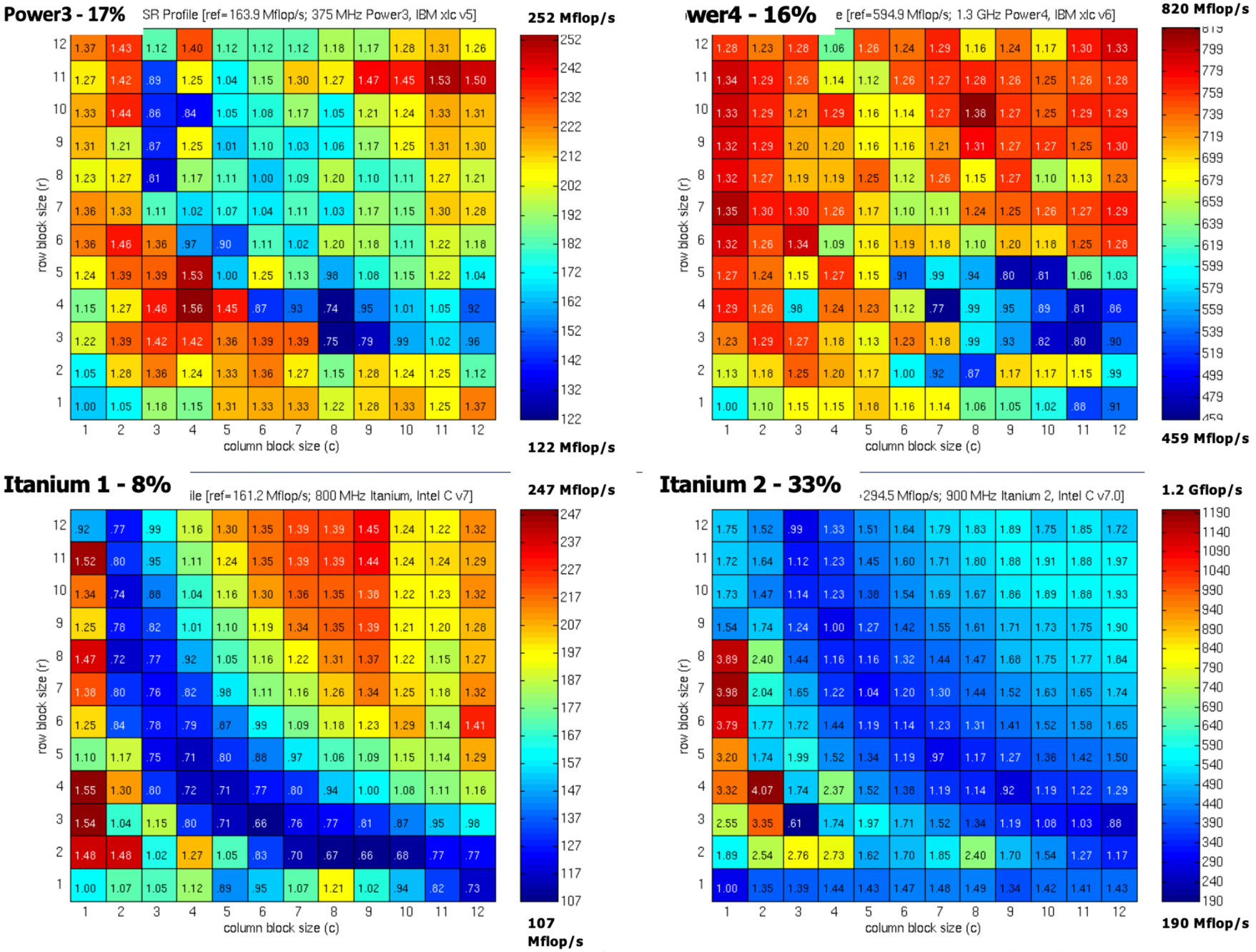
"Optimizing the Performance of Sparse Matrix-Vector Multiplication," Im, 2000.
"Automatic Performance Tuning of Sparse Matrix Kernels," Vuduc, 2003.

Register Profile: dense matrix in sparse format



From UC Berkeley CS267 "Automatic Performance Tuning of Sparse Matrix Kernels," Vuduc, 2003.

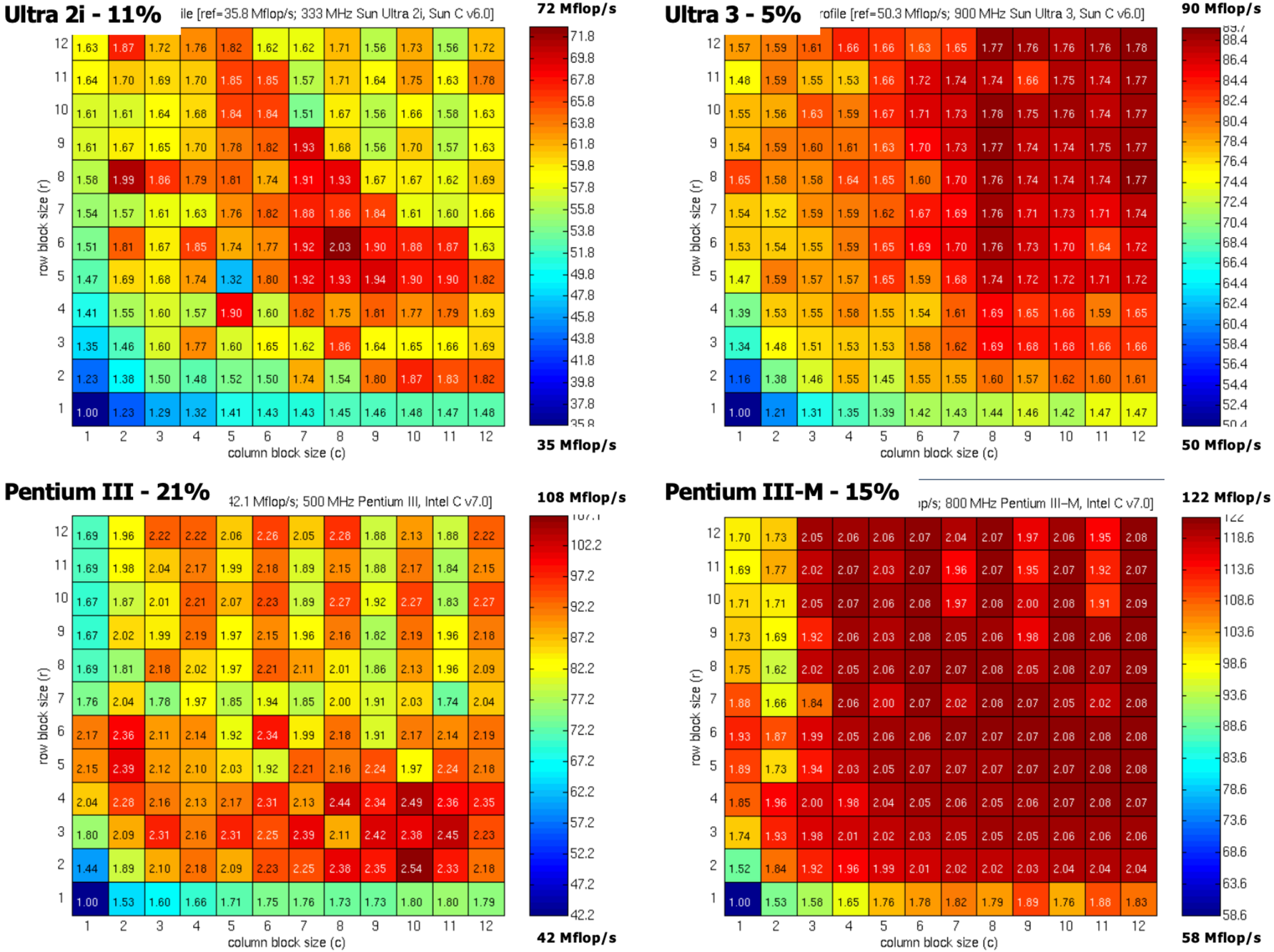
Register Profiles



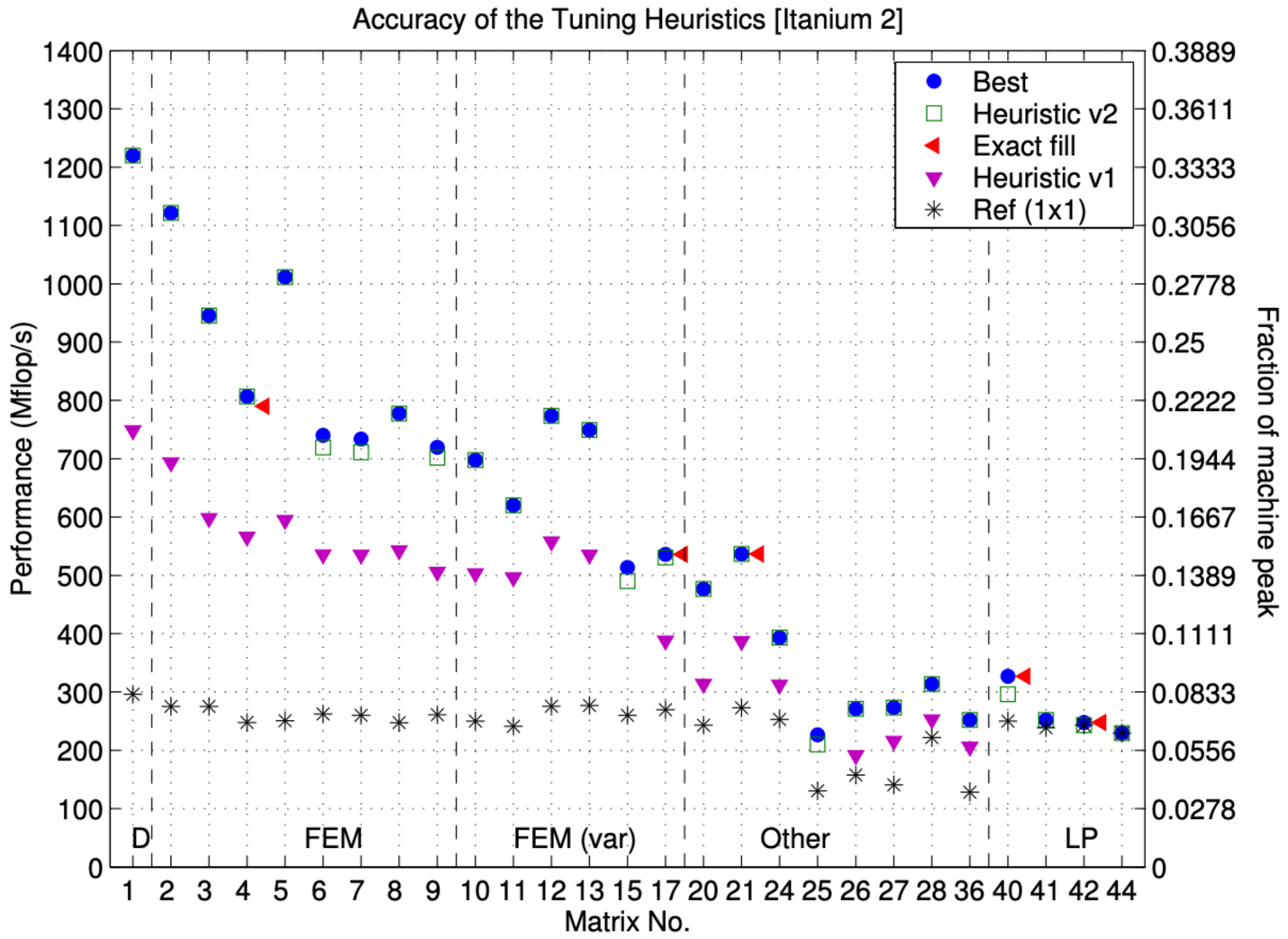
From UC Berkeley CS267

"Automatic Performance Tuning of Sparse Matrix Kernels," Vuduc, 2003.

Register Profiles

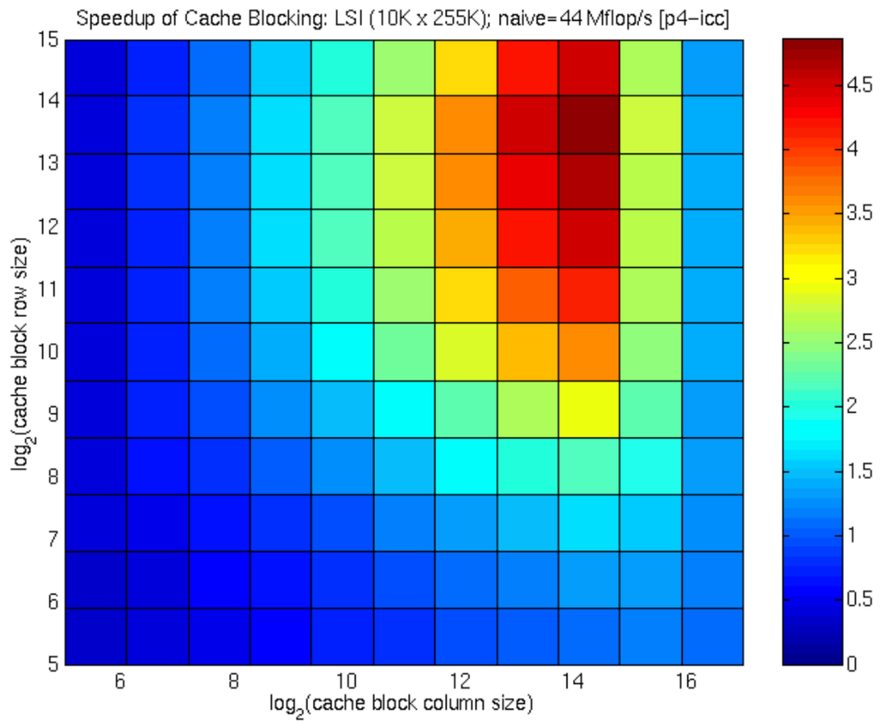


Heuristic Tuning vs Best



What about cache blocking?

- For CSR, dot-product, re-use opportunity is only in the x vector
- Matrices that are have some locality and “well ordered” e.g., near diagonal have good re-use
- Cache blocking can help for “short wide” matrices both on serial and SMPs



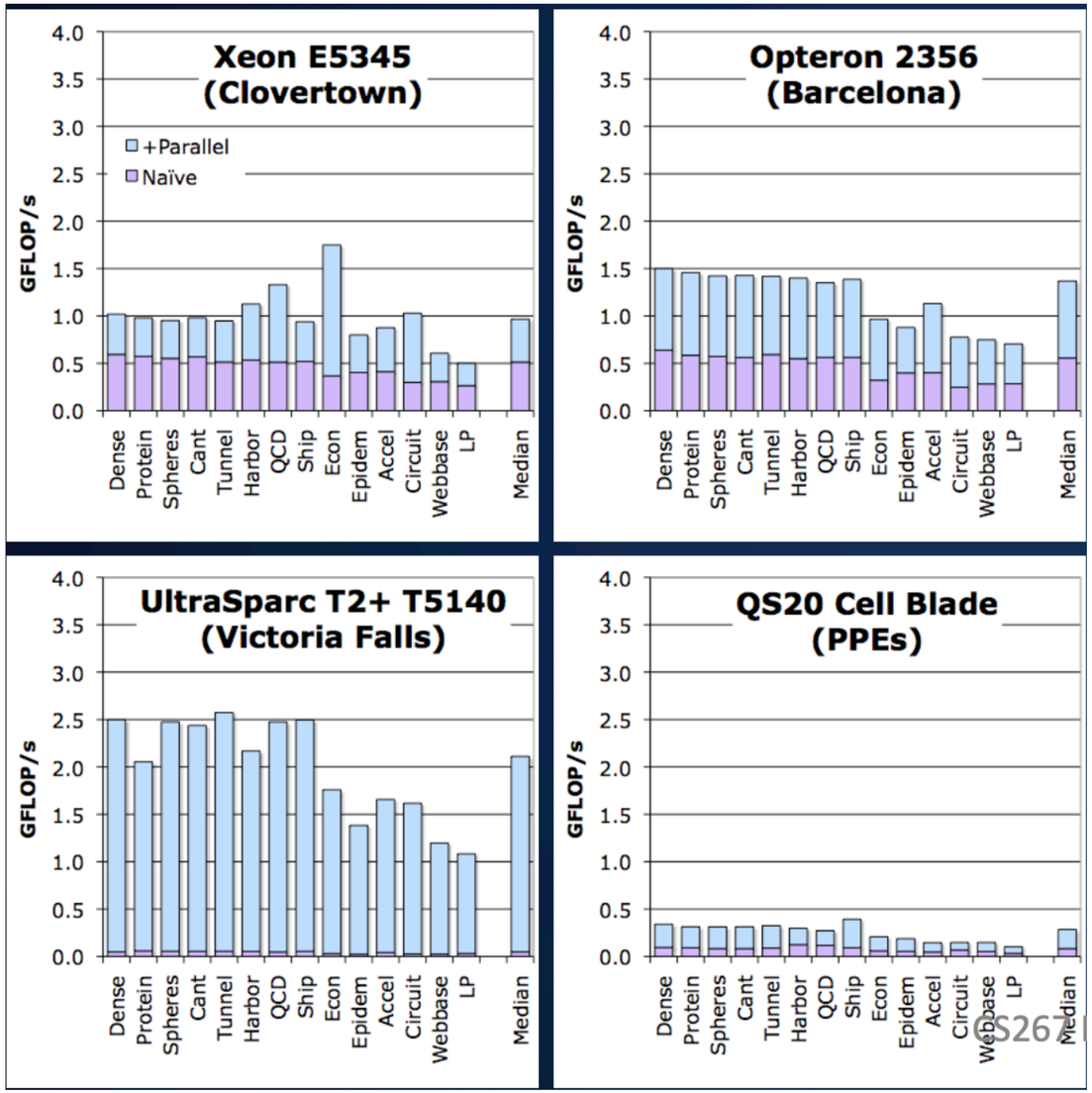
Matrix A
- 100k x 255k
- 3.7M non-zeros

Baseline: 44 Mflop/s

Best block size & performance:
- 16k x 16k
- 210 Mflop/s

Nishtala, et al (2007). When cache blocking of sparse matrix vector multiply works and why.

SpMV Performance (simple parallelization)



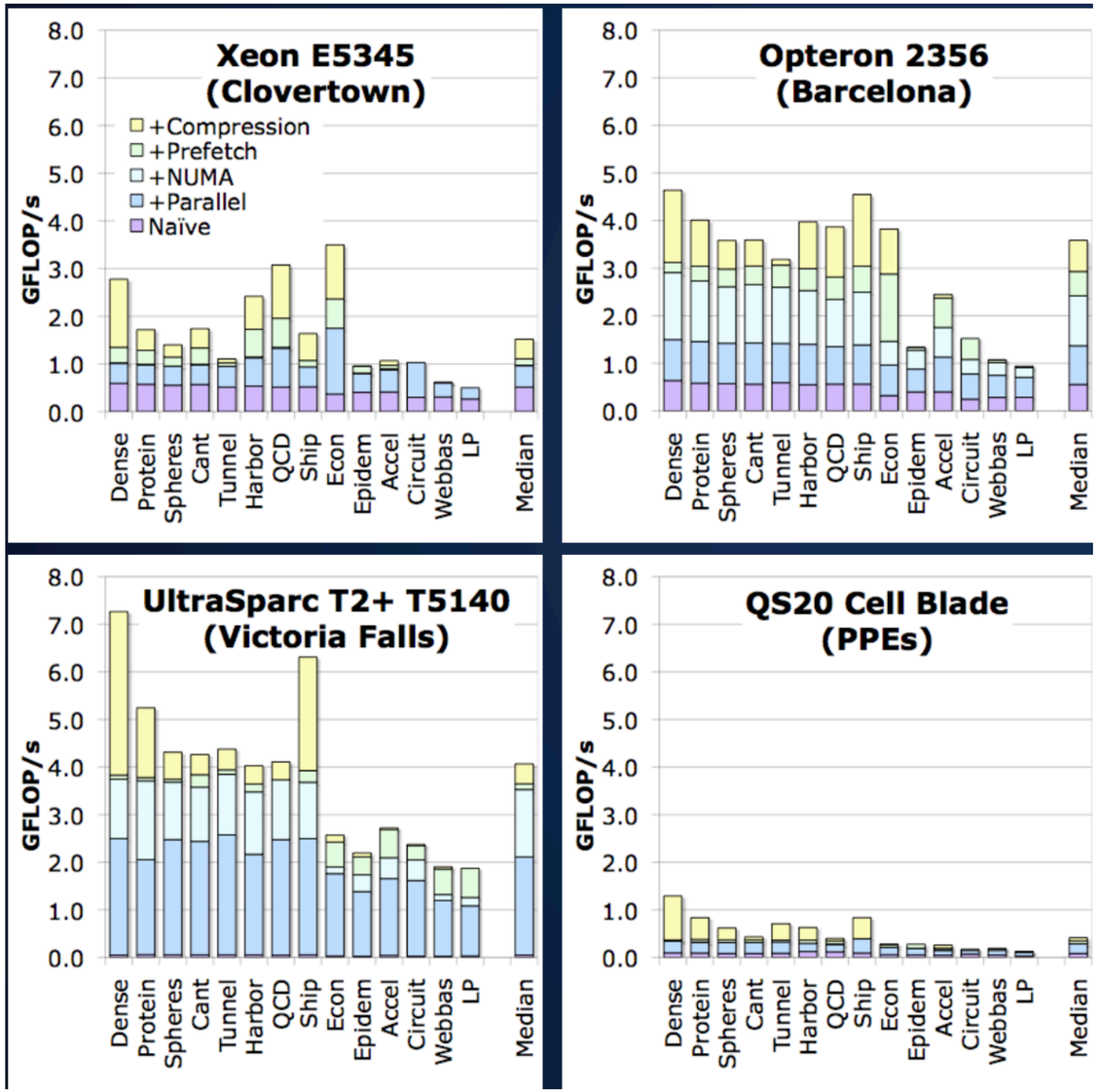
Out-of-the box SpMV performance on a suite of 14 matrices

Simplest solution = parallelization by rows

Scalability isn't great - Can we do better?

From UC Berkeley CS267

SpMV Performance (matrix compression)



After maximizing memory bandwidth, the only hope is to minimize memory traffic.

Compression: exploit

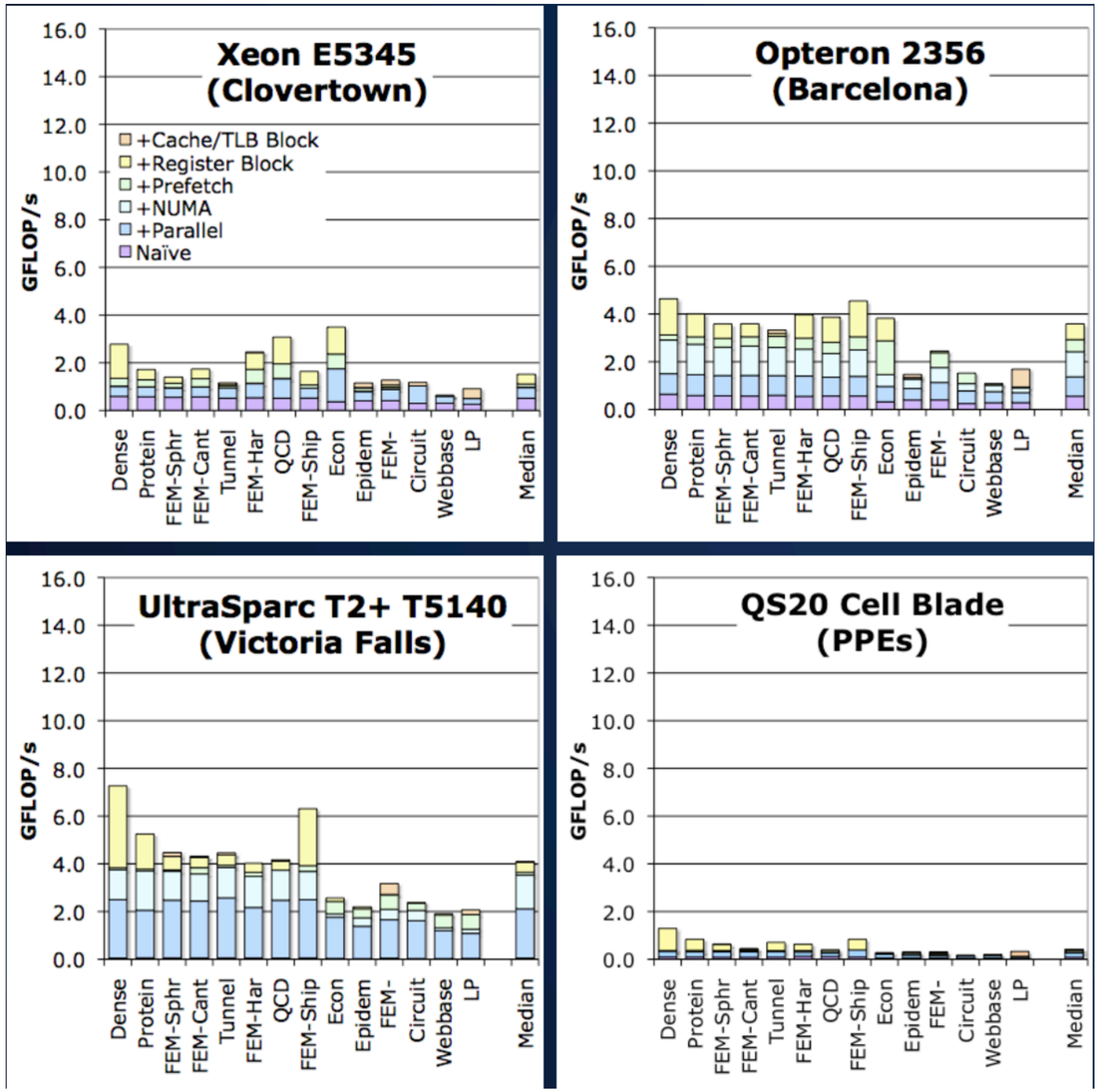
- register blocking
- other formats
- smaller indices

Benefit is matrix-dependent.

Register blocking enables efficient software prefetching (one per cache line)

From UC Berkeley CS267

SpMV Performance (cache and TLB blocking)



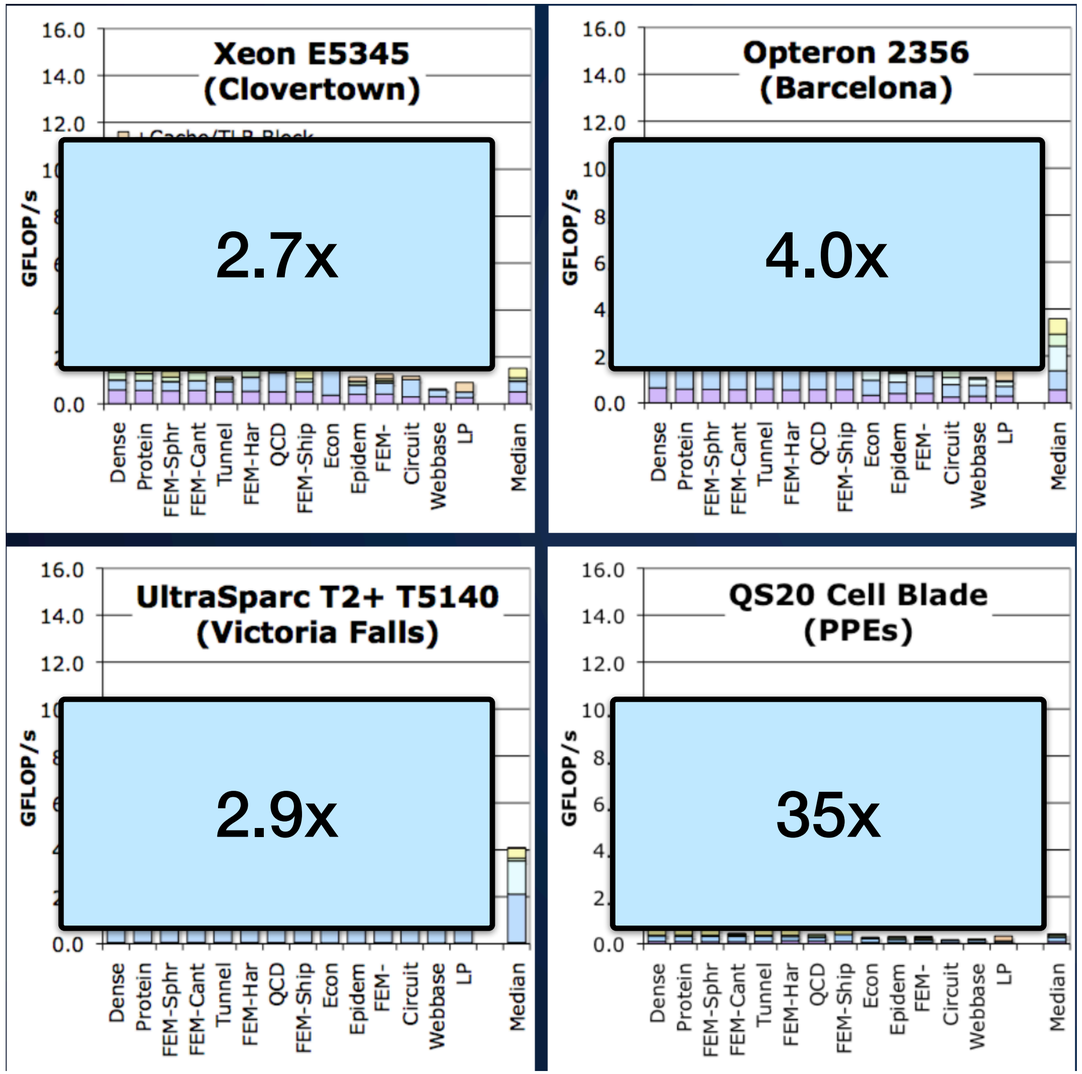
Fully auto-tuned SpMV performance across the suite of matrices

Why do some optimizations work better on some architectures?

- Matrices with naturally small working sets
- Architectures with giant caches

From UC Berkeley CS267

SpMV Performance (cache and TLB blocking)



Fully auto-tuned SpMV performance across the suite of matrices

Why do some optimizations work better on some architectures?

- Matrices with naturally small working sets
- Architectures with giant caches

Summary

- Tuning for sparse matrices: harder than dense ones
- SpMV: benefits lower due to **low Computational Intensity** (read the matrix)
- **Register blocking** and other “compression” can be a big win
- Cache blocking less so; other low level tuning (e.g., prefetch) some
- For distributed memory, **reordering** (e.g., graph partitioning) important
- **Autotuning** possible, but depends on sparsity structure; hybrid offline / online tuning
- After tuning SpMV should be **memory bandwidth limited**
- Optimizing at **higher-level algorithms** (across iterations) can improve reuse.