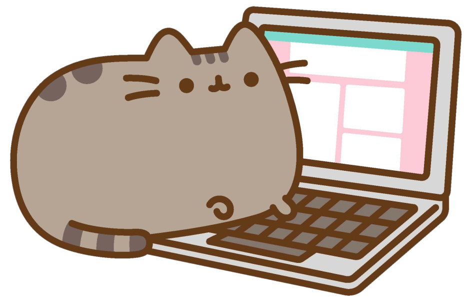
CSE 6230:
HPC Tools and Applications

+



Lecture 15: Sparse Matrix-Vector Multiplication (SpMV)

Helen Xu
hxu615@gatech.edu

Georgia Tech College of Computing
School of Computational Science and Engineering

Announcements

- HW4 deadline postponed to next Tu, March 5 @ 5pm

- Please sign up for project groups by next Monday, March 4, by 5pm

- Project proposal writeup / slides due Tu, March 12 (2 weeks) - we may need to schedule some presentations outside of class time - we will release an initial schedule after groups are formed.

- HW5 out - deadline postponed to Tu, Mar 26 (after the spring break)

- As a reminder, please don’t use AI tools (e.g., ChatGPT, etc.). Also, you are welcome to discuss with other students about the HW, but please don’t look directly at or copy other’s code.

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

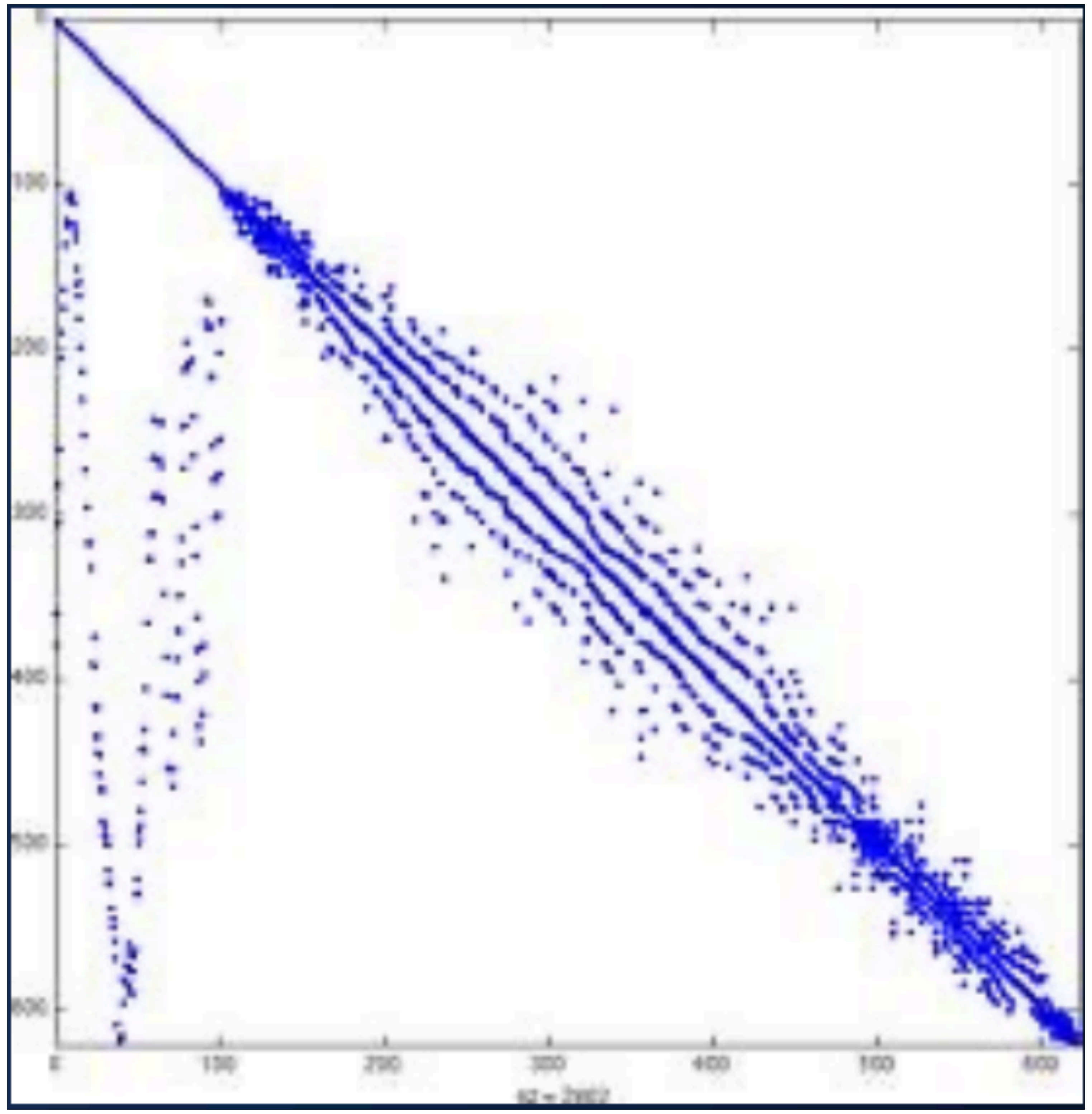
From UC Berkeley CS267

What is a sparse matrix?

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

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

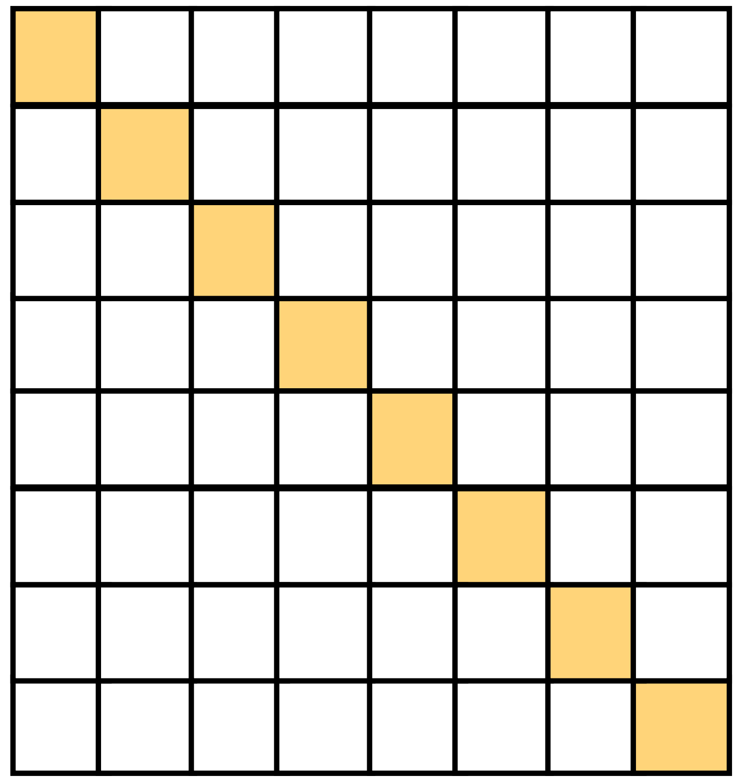
Sparse matrices arise in many applications
- Simulating climate
- Analyzing images (photos, MRIs,…)
- Web page ranking for search
- Graphs, including Graph Neural Nets



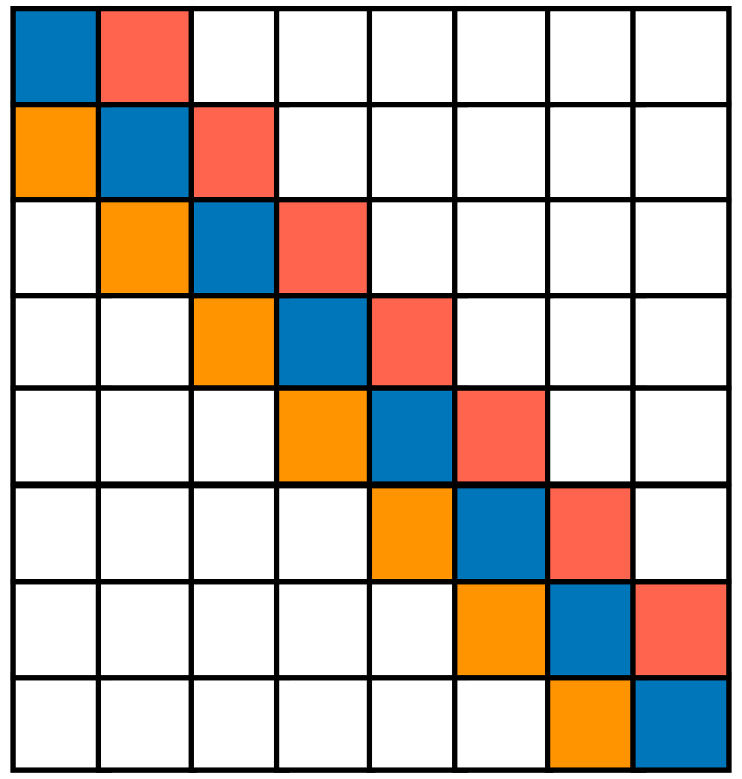
https://sparse.tamu.edu/

From UC Berkeley CS267

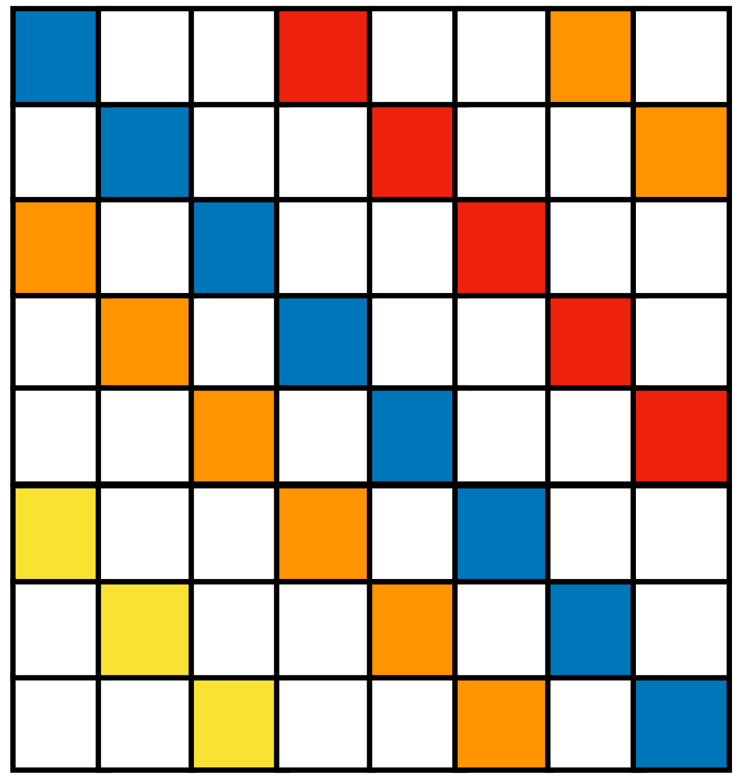
Examples of sparse matrices



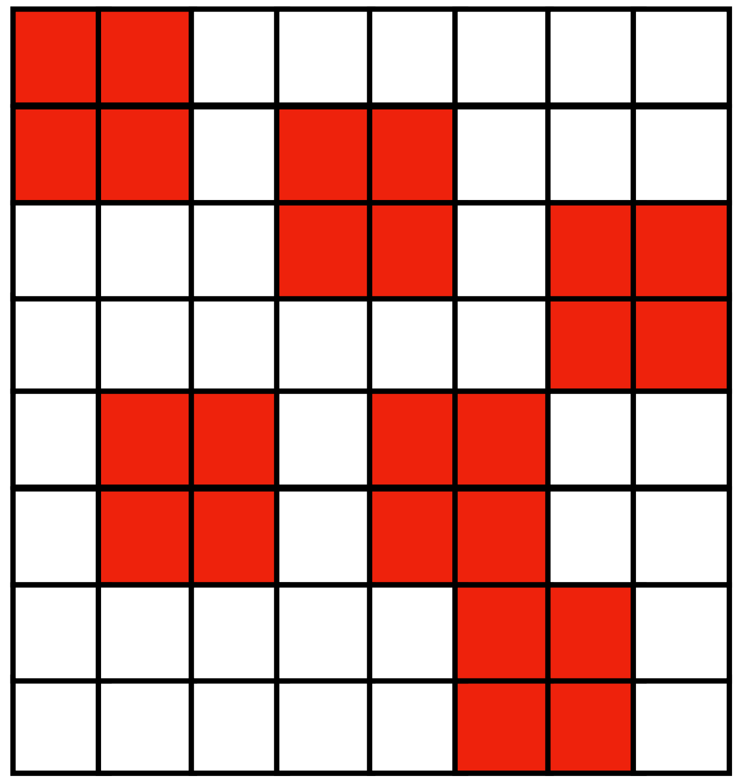
Diagonal



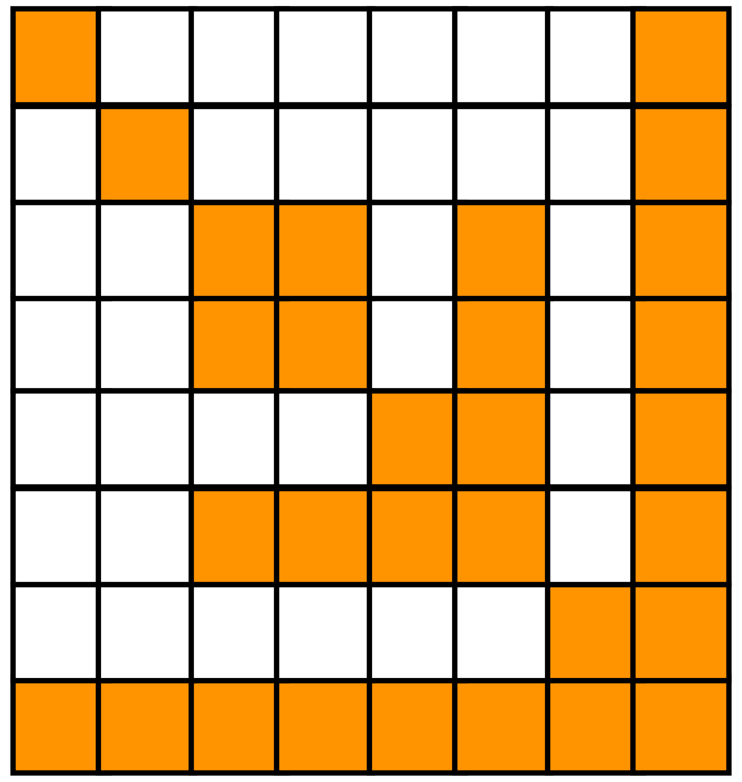
Tridiagonal



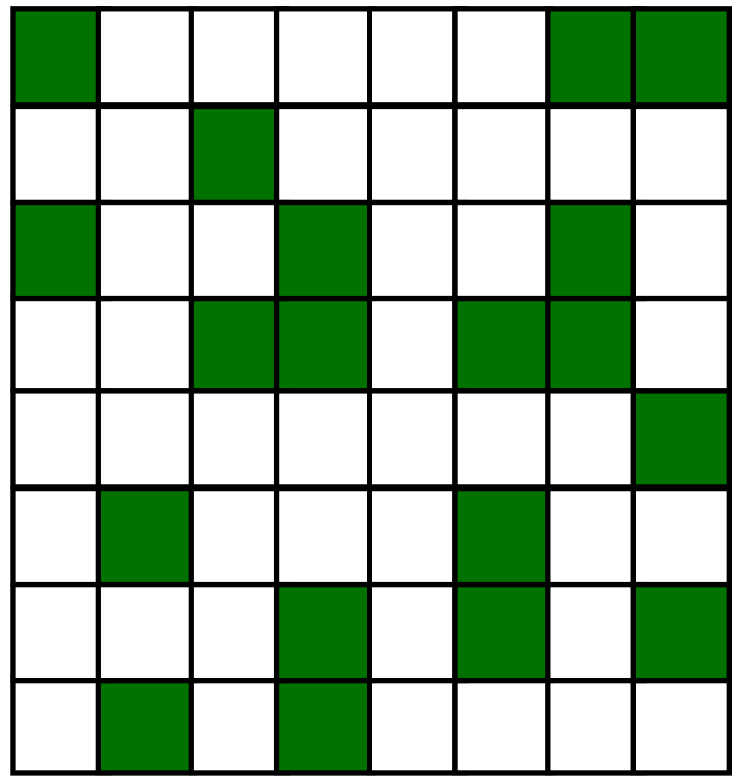
“Generalized diagonal”



Block matrix
(2x2 dense blocks)



Symmetric

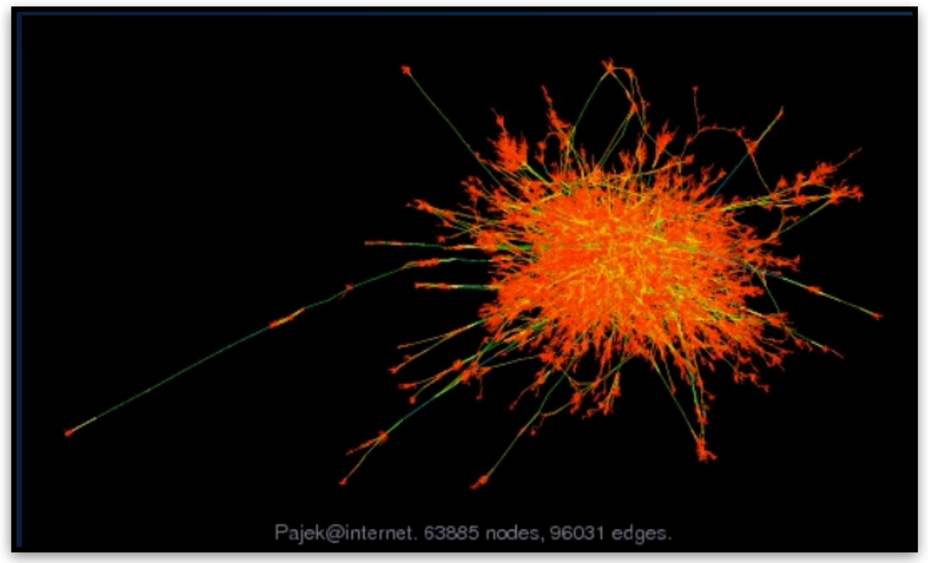


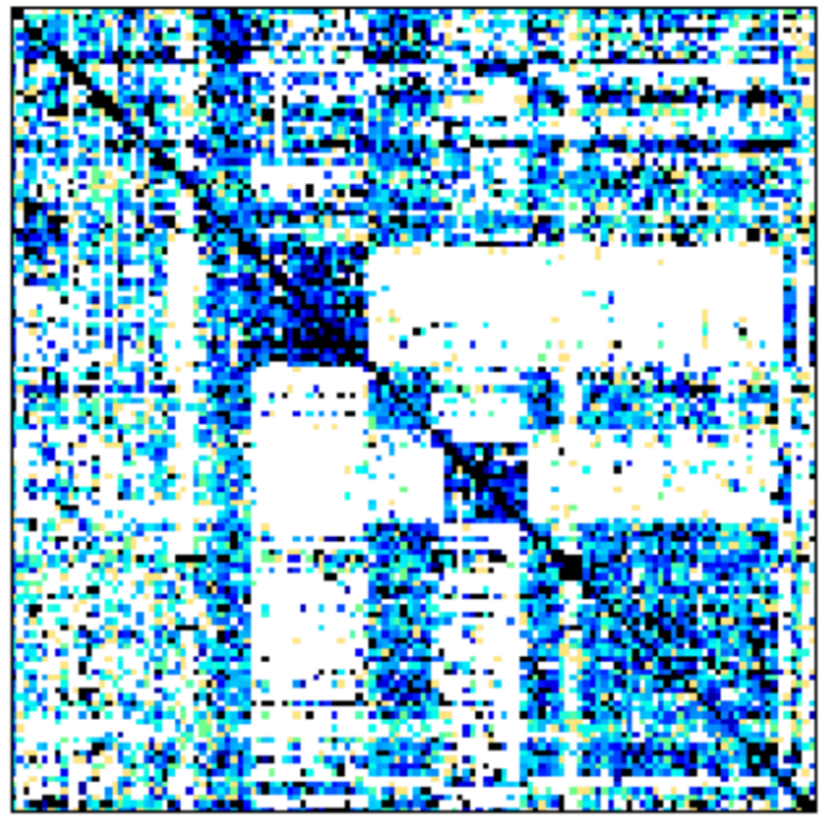
Irregular

From UC Berkeley CS267

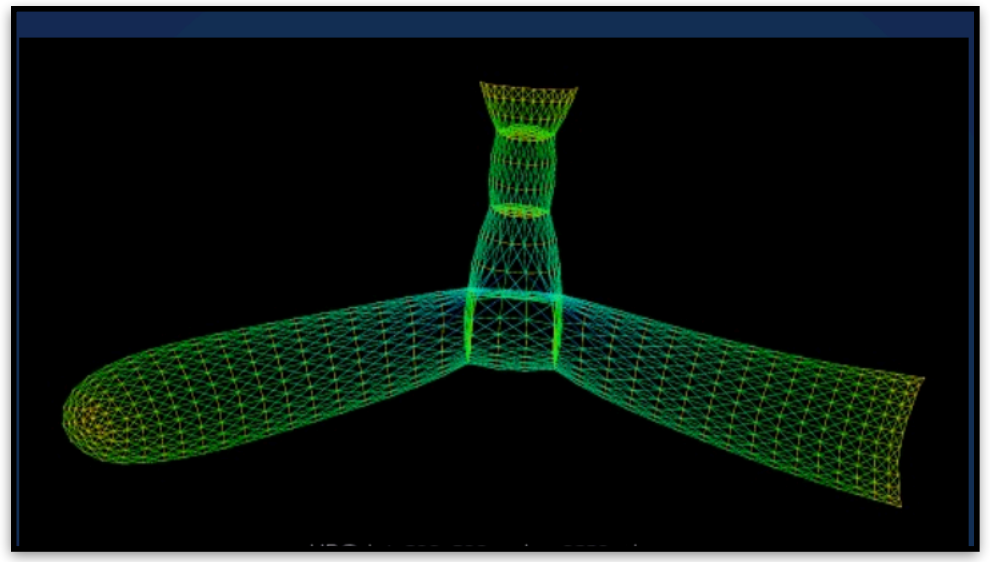
Sparse matrices are everywhere

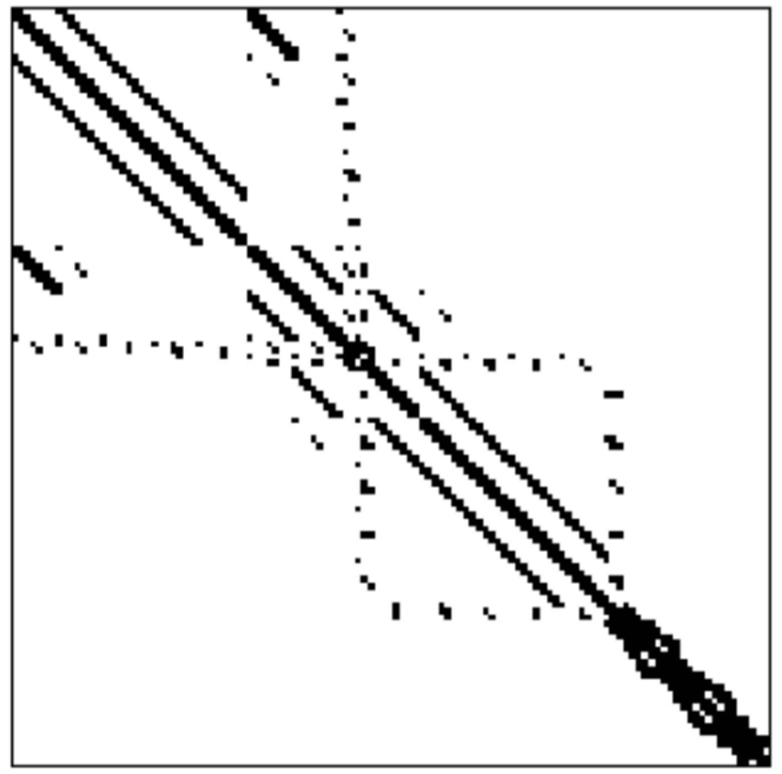
Internet connectivity



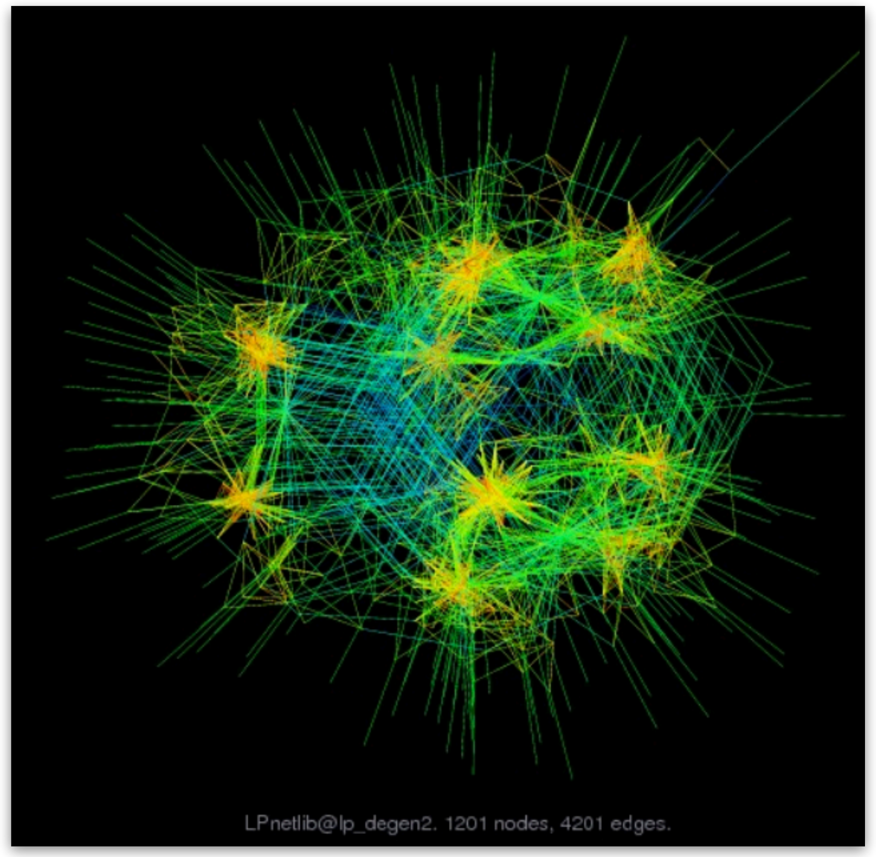


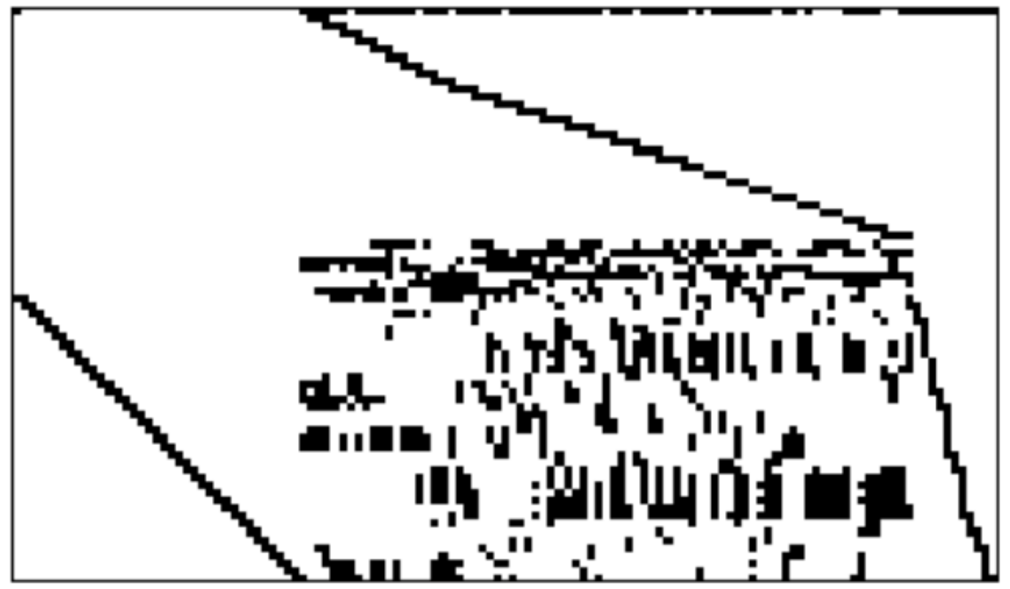
Structural design





Linear programming





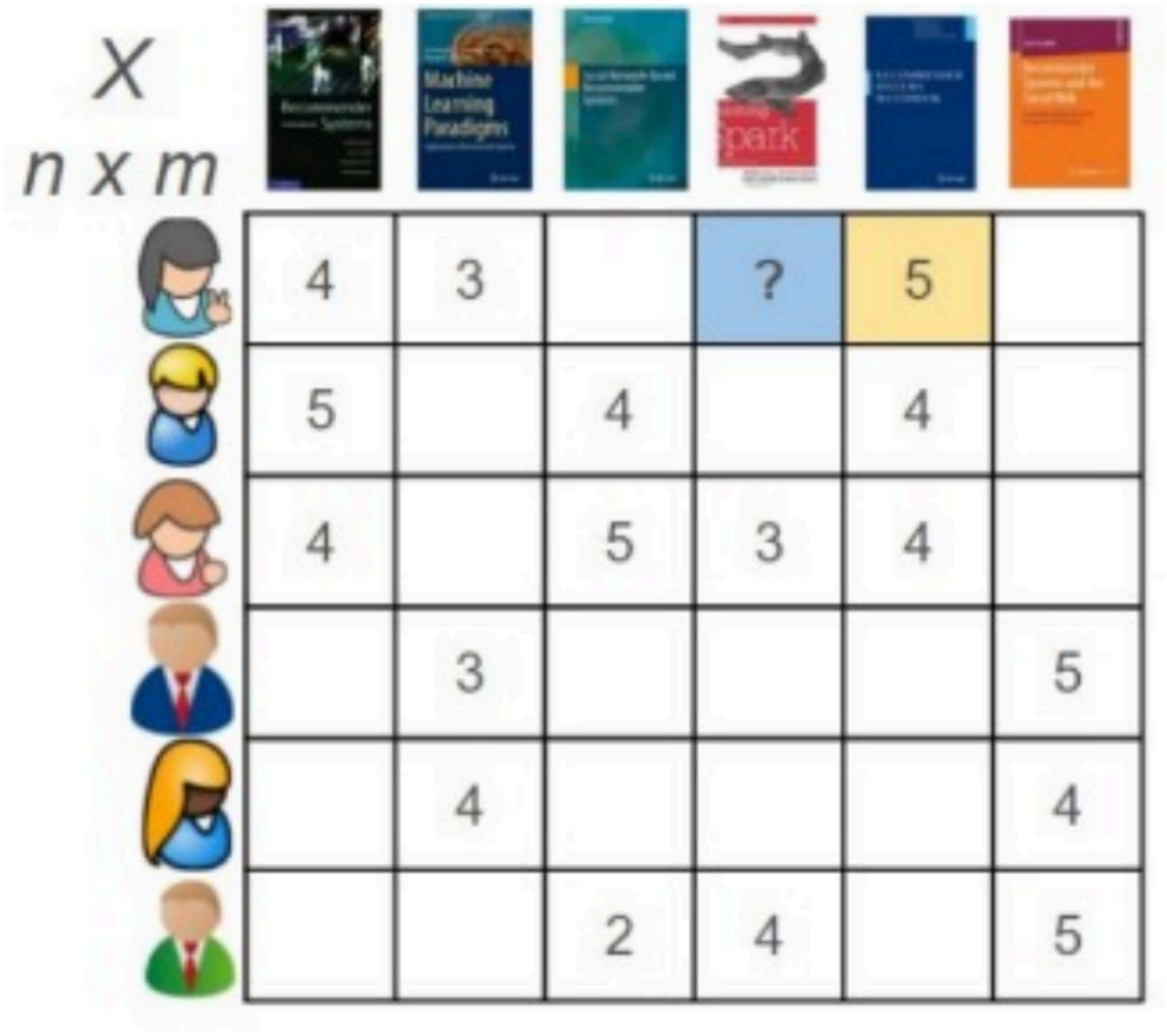
From UC Berkeley CS267

Recommendation matrix

Products

Population

Ratings



4	3		?	5	
5		4		4	
4		5	3	4	
	3				5
	4				4
		2	4		5

From UC Berkeley CS267

Image from Oliver Gindele, Data Scientist at Datatonic

Image Segmentation

Image segmentation – Identify the **object boundaries** in an image

Compute **affinity matrix** – Is it likely that two pixels belong to the same object?

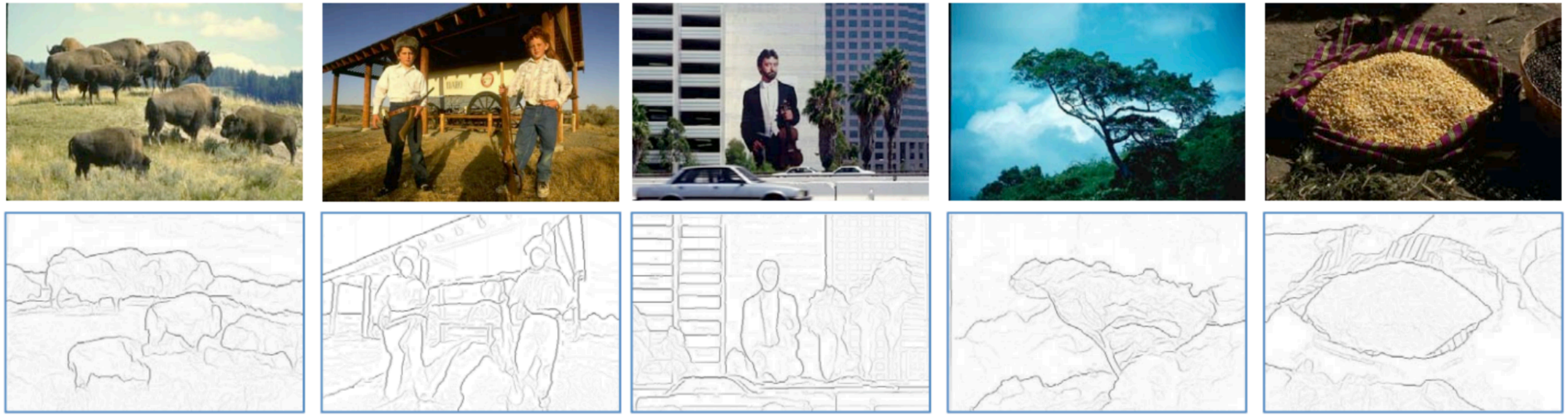


Figure 7. Selected image contours

"Efficient, High-Quality Image Contour Detection" Catanzaro, Su, Sundaram, Lee, Murphy, Keutzer, International Conference on Computer Vision, September 2009

From UC Berkeley CS267

More applications

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

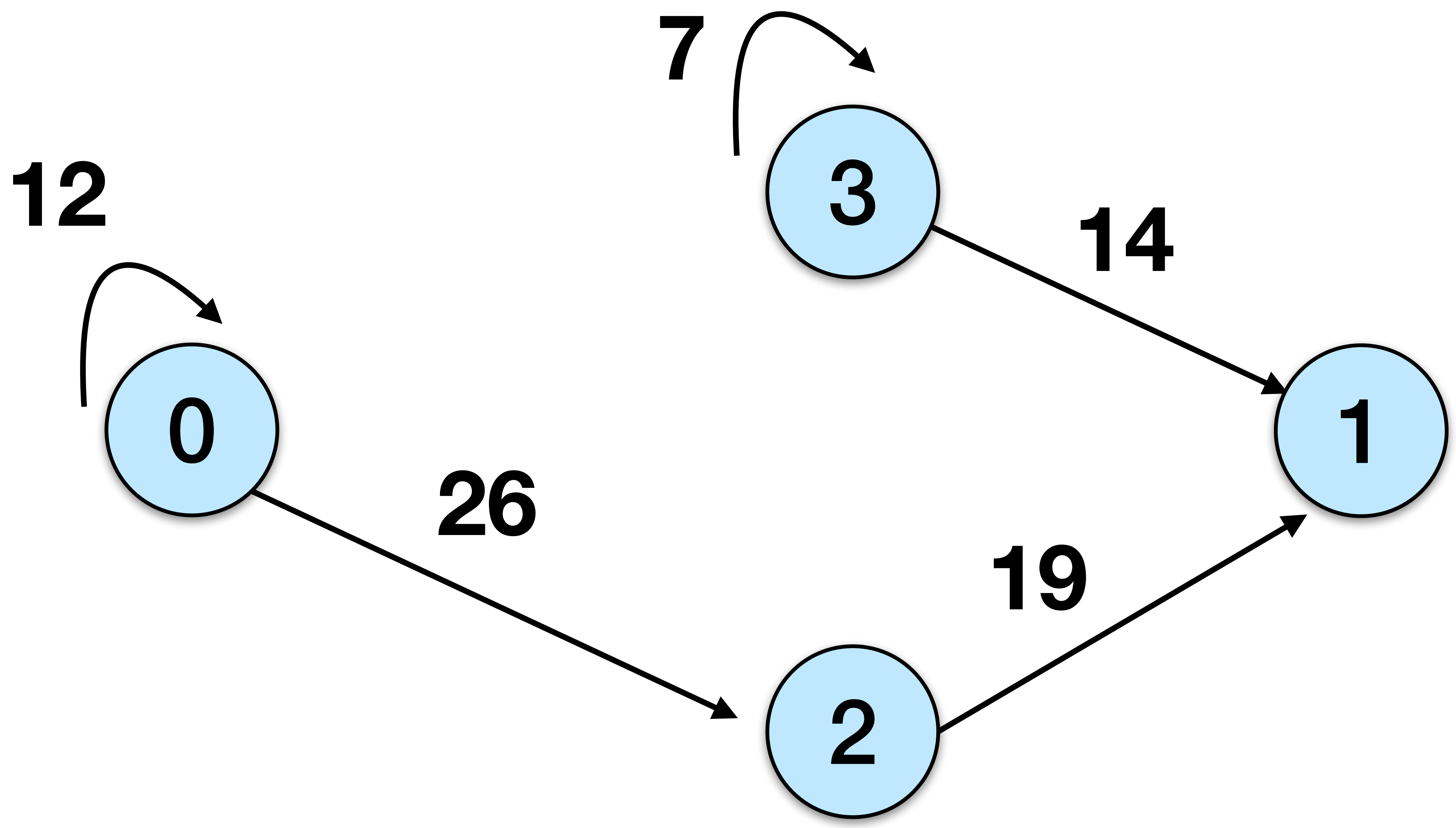
And many more…

From UC Berkeley CS267

Sparse Matrix Formats

Adjacency matrix representation

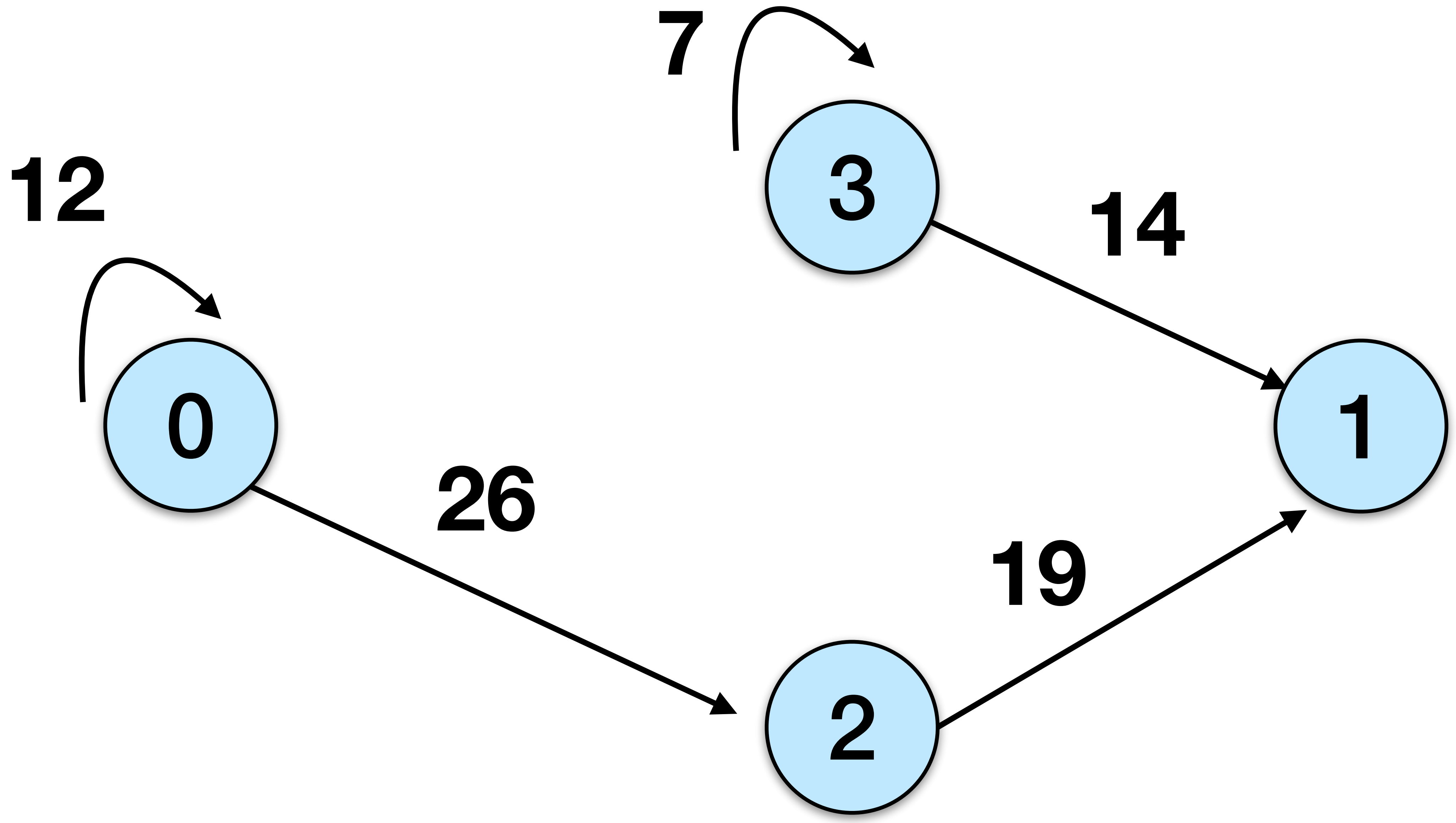
Any sparse matrix representation can be used for sparse graphs, and vice versa.



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

From UC Berkeley CS267

Coordinate representation (COO)



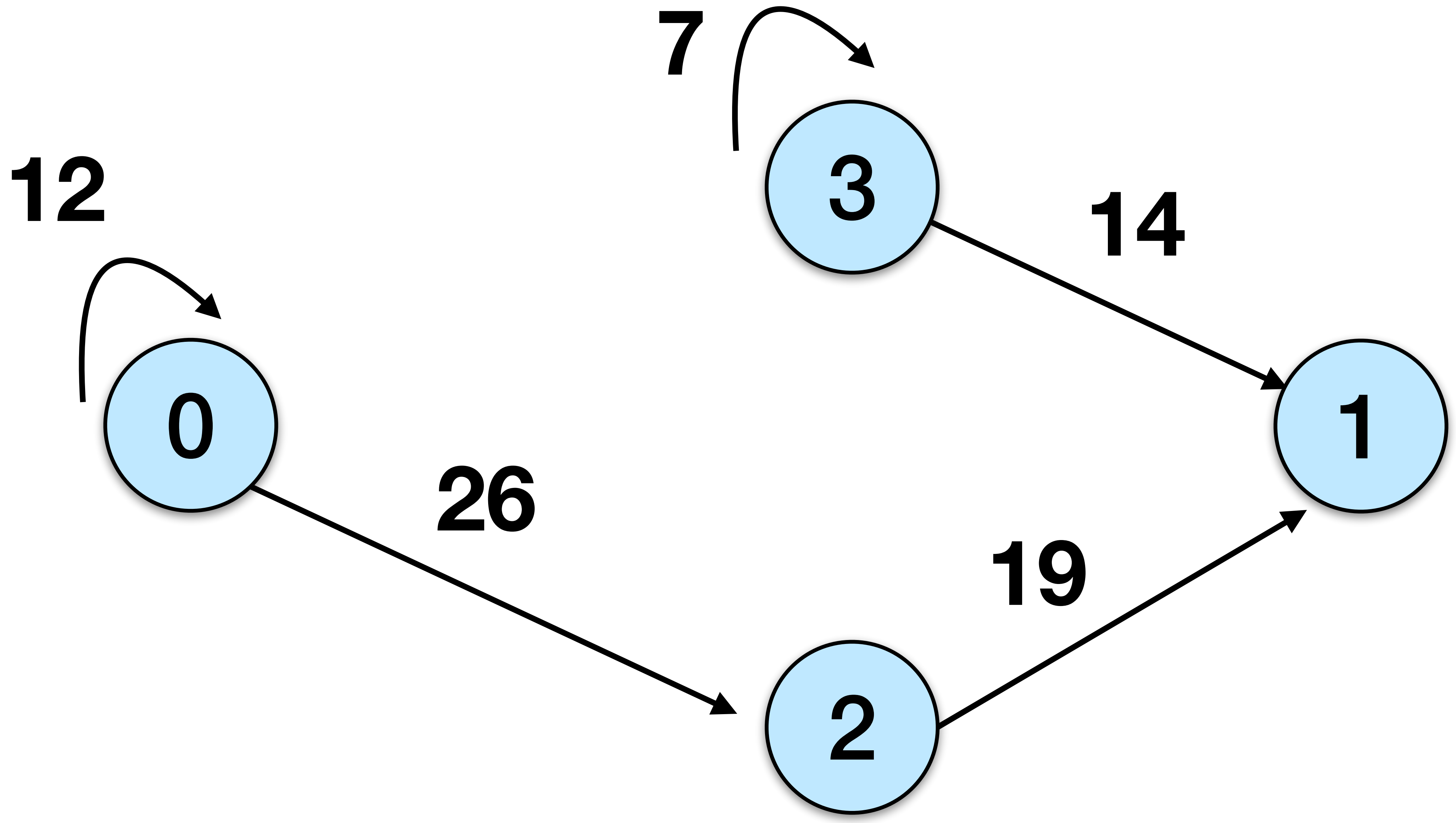
	0	1	2	3
0	12		26	
1				
2		19		
3		14		7

(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)

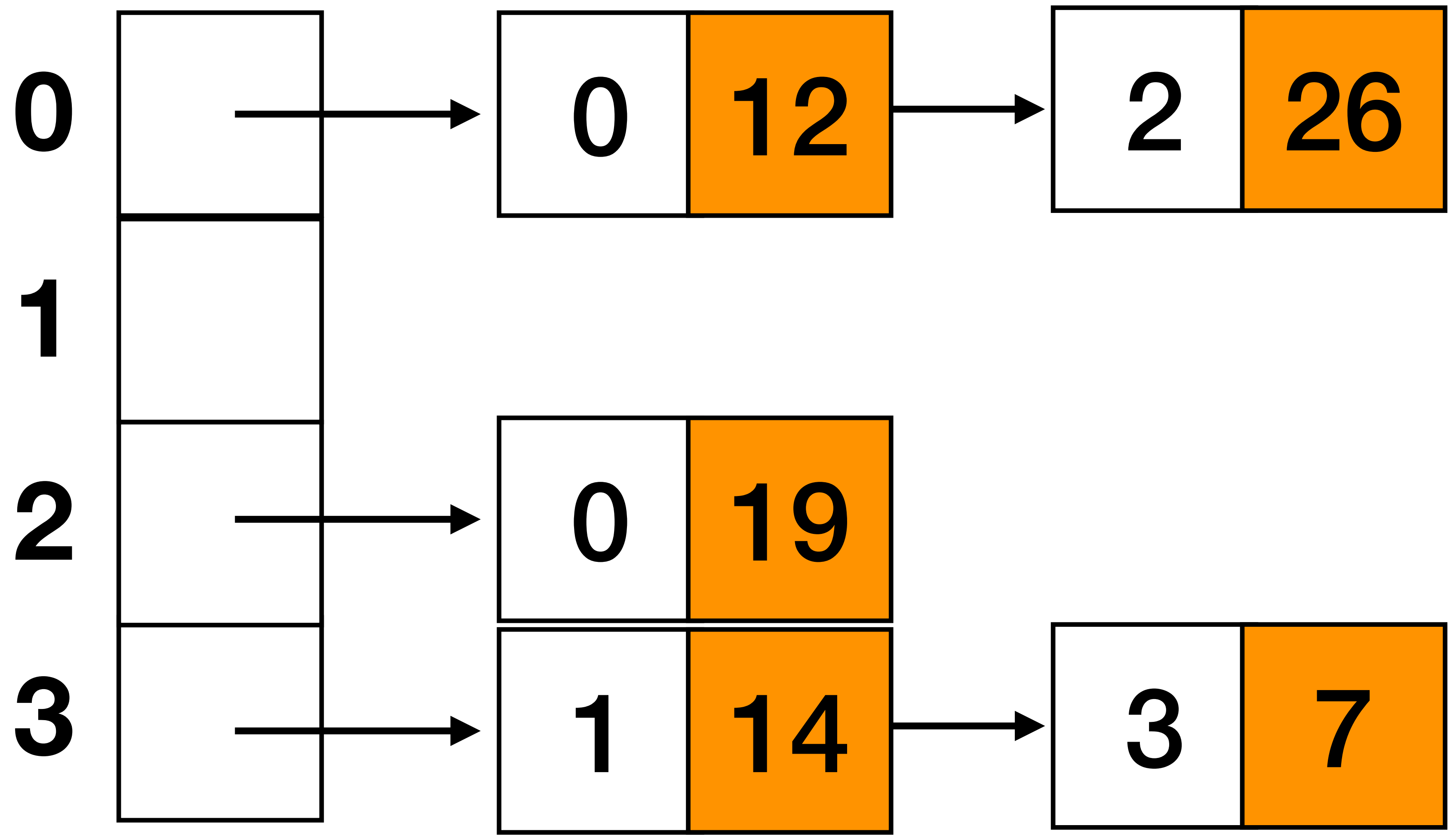
From UC Berkeley CS267

Adjacency lists



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

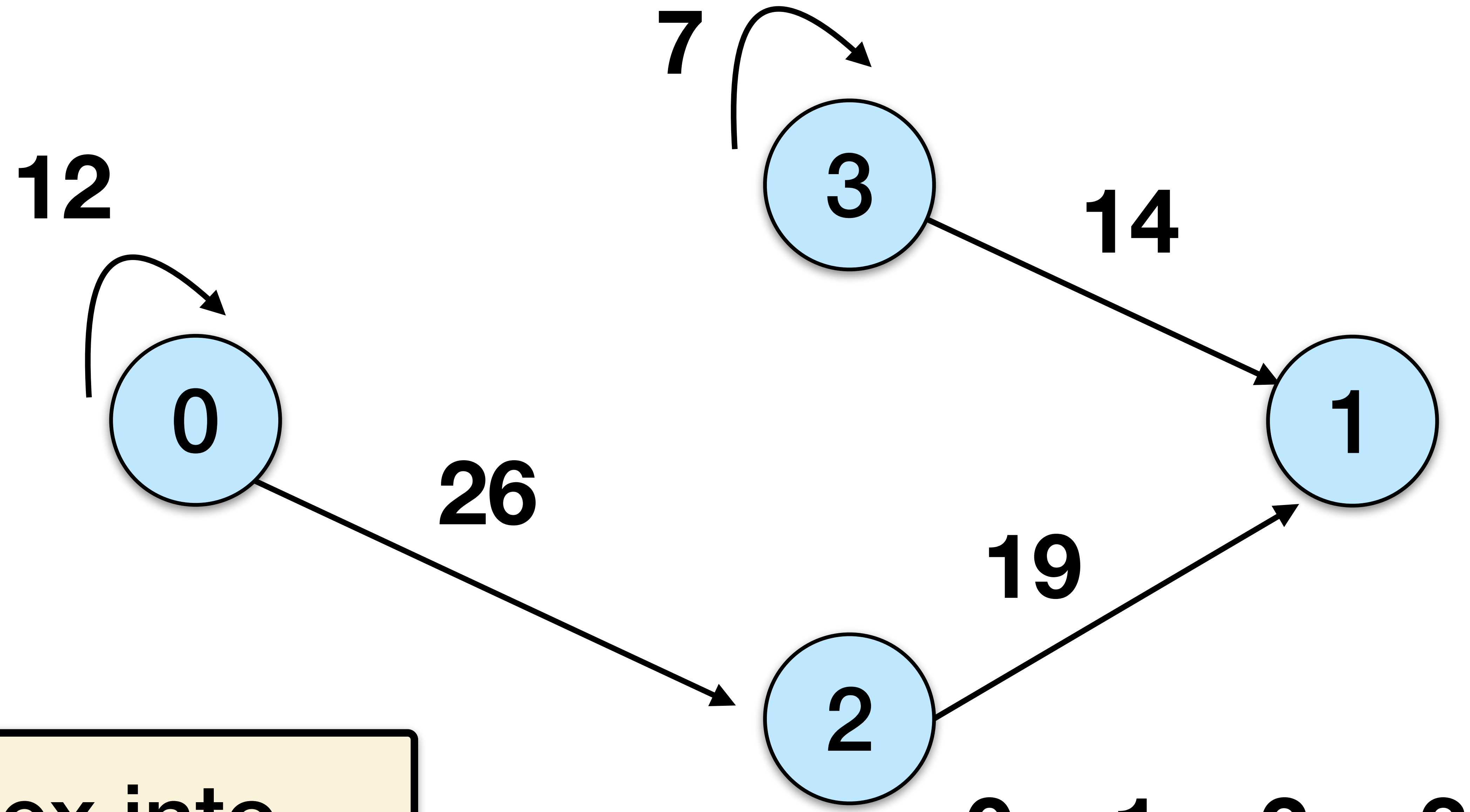
Source stored implicitly

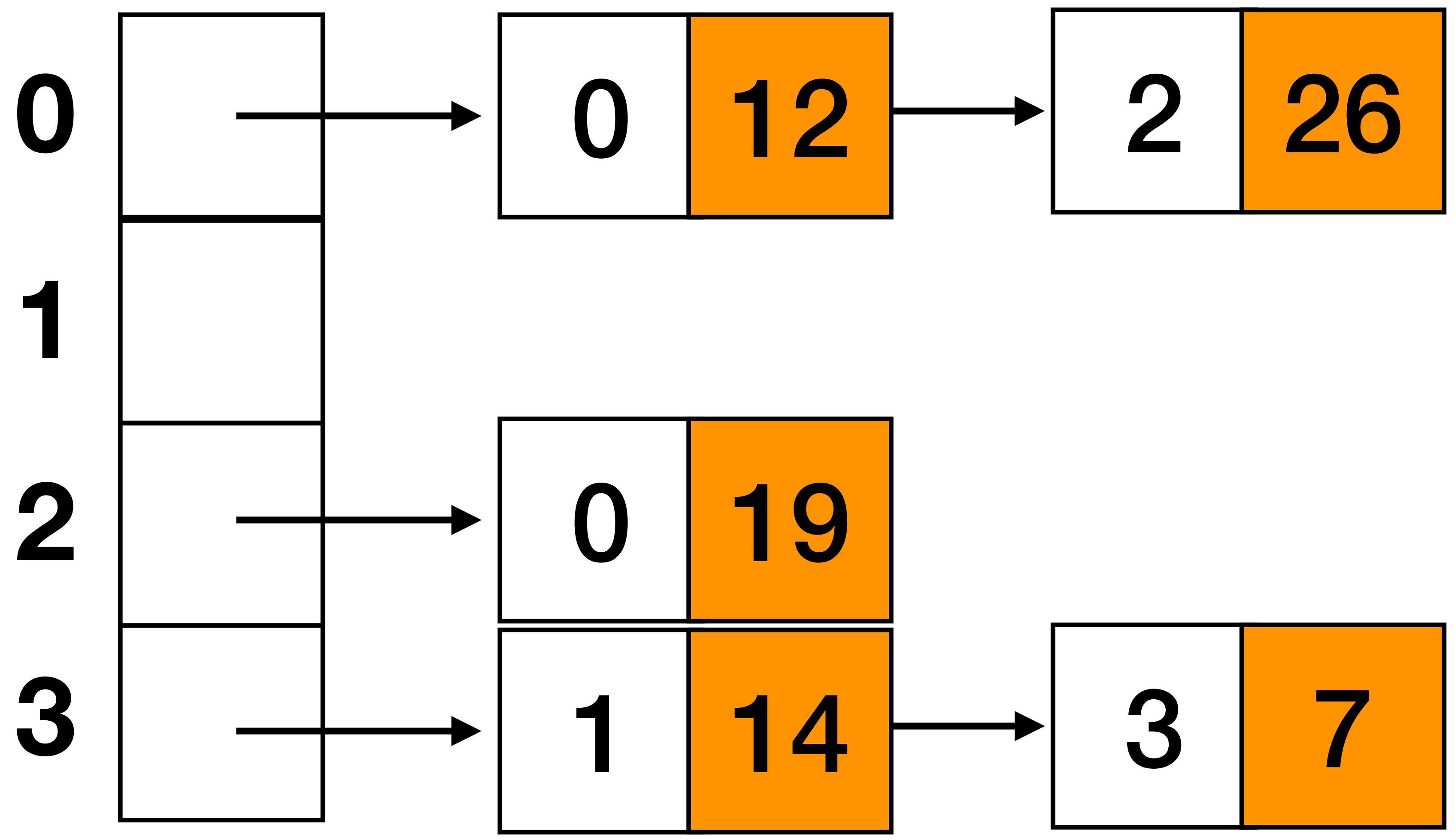


From UC Berkeley CS267

Compressed Sparse Row (CSR)

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





Index into adjacency array

Adjacencies

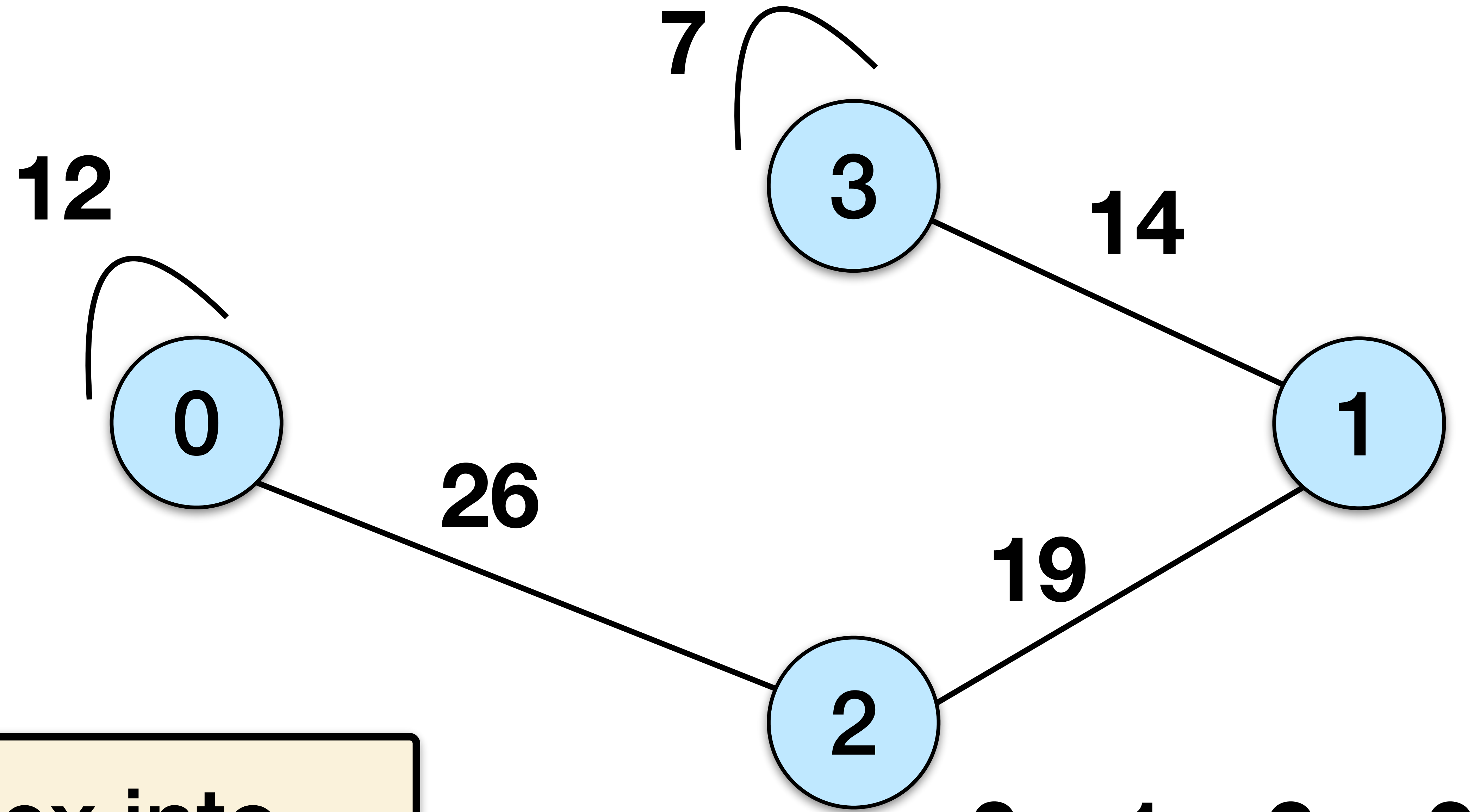
Weights

	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)**

From UC Berkeley CS267

Directed vs Undirected

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

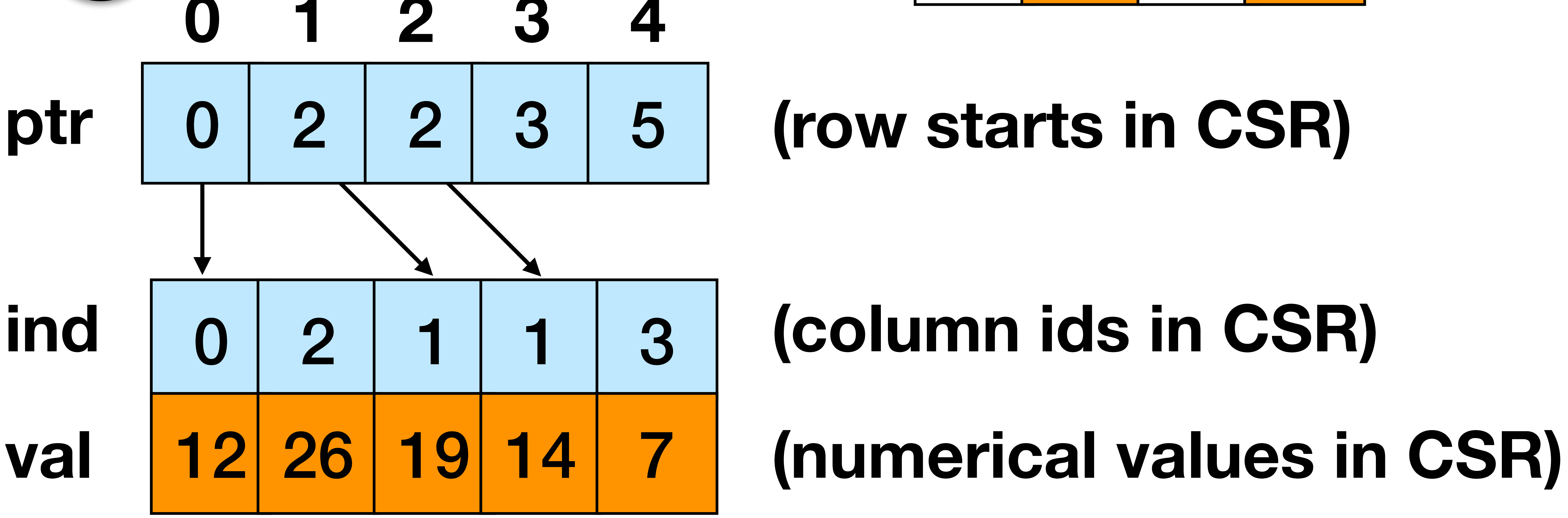


	0	1	2	3
0	12		26	
1			19	14
2	26	19		
3		14		7

Index into adjacency array

Adjacencies

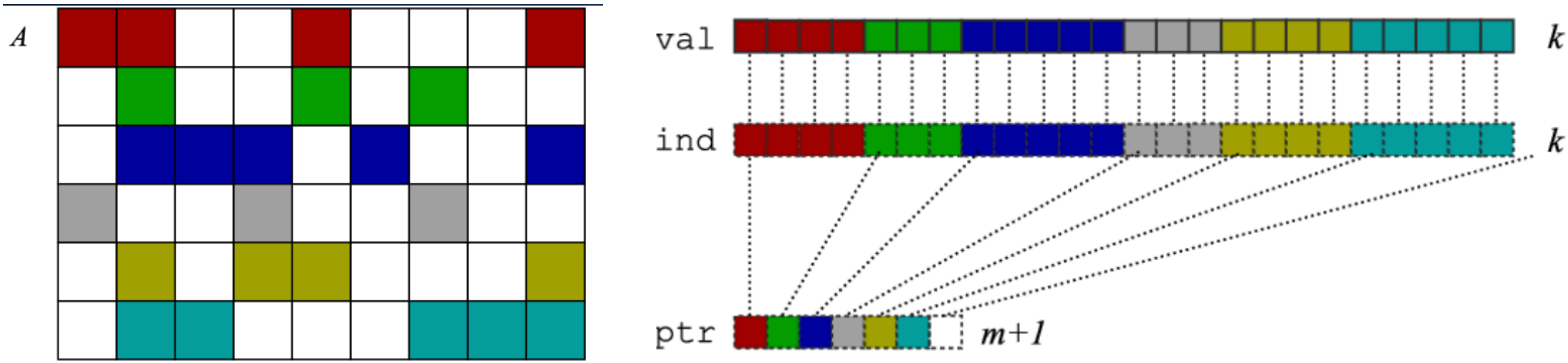
Weights



	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)

From UC Berkeley CS267

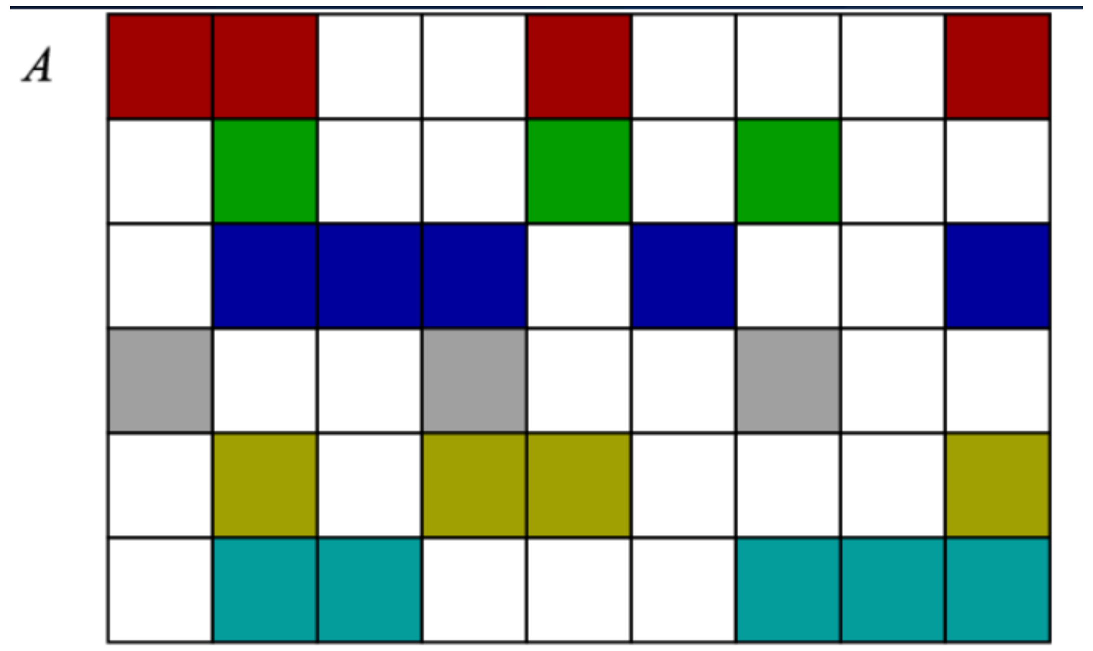
Compressed Sparse Row (CSR) Storage



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

From UC Berkeley CS267



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

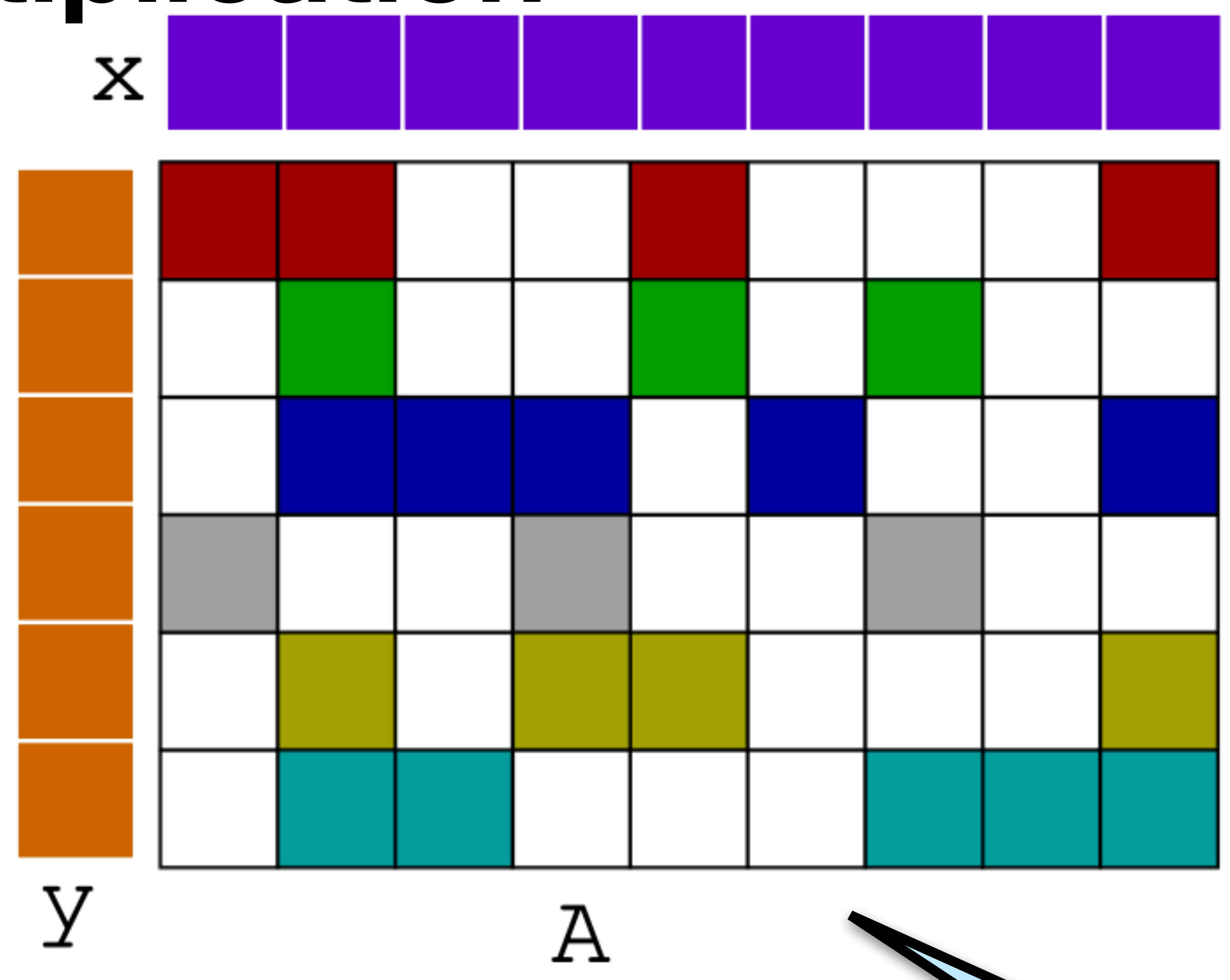
…and many other specialized formats!

From UC Berkeley CS267

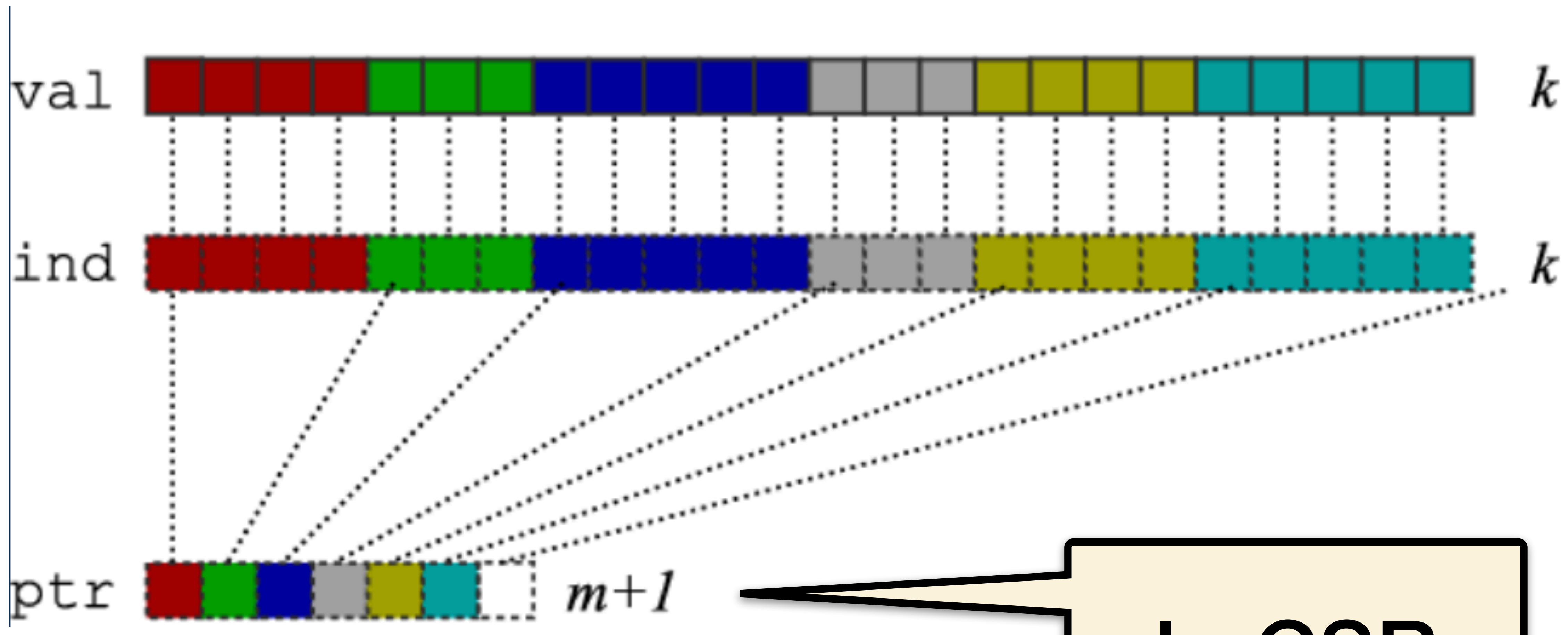
Serial SpMV

Serial SpMV in CSR

SpMV: Sparse Matrix-Vector Multiplication



Representation of A



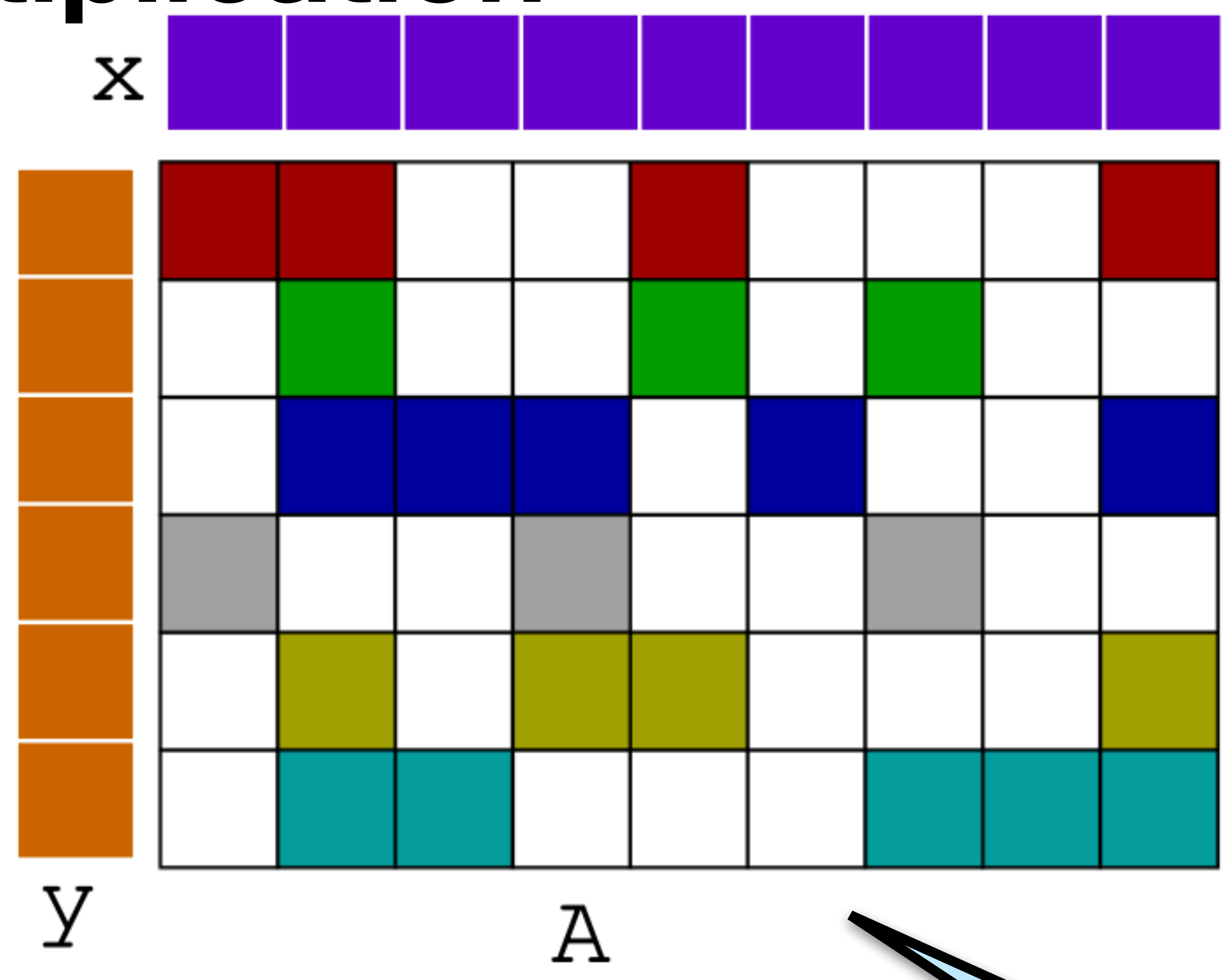
In CSR

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

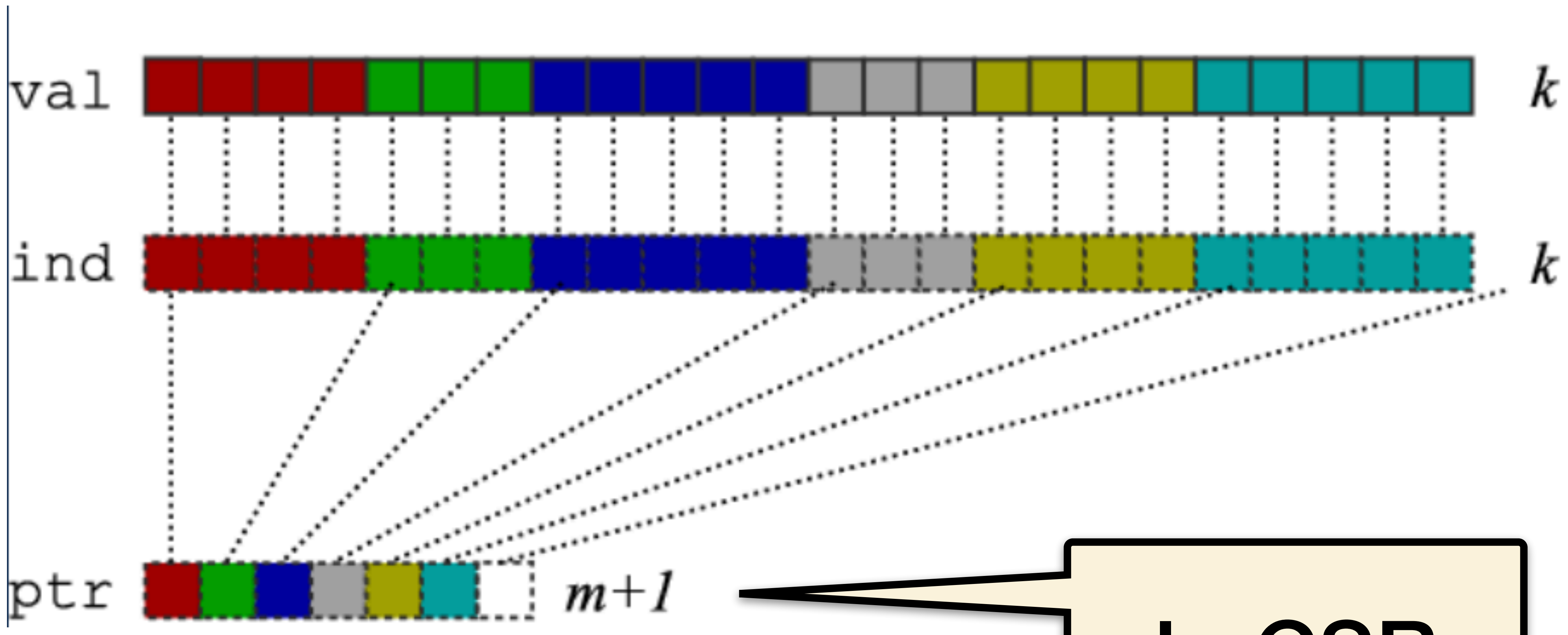
From UC Berkeley CS267

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]]
```

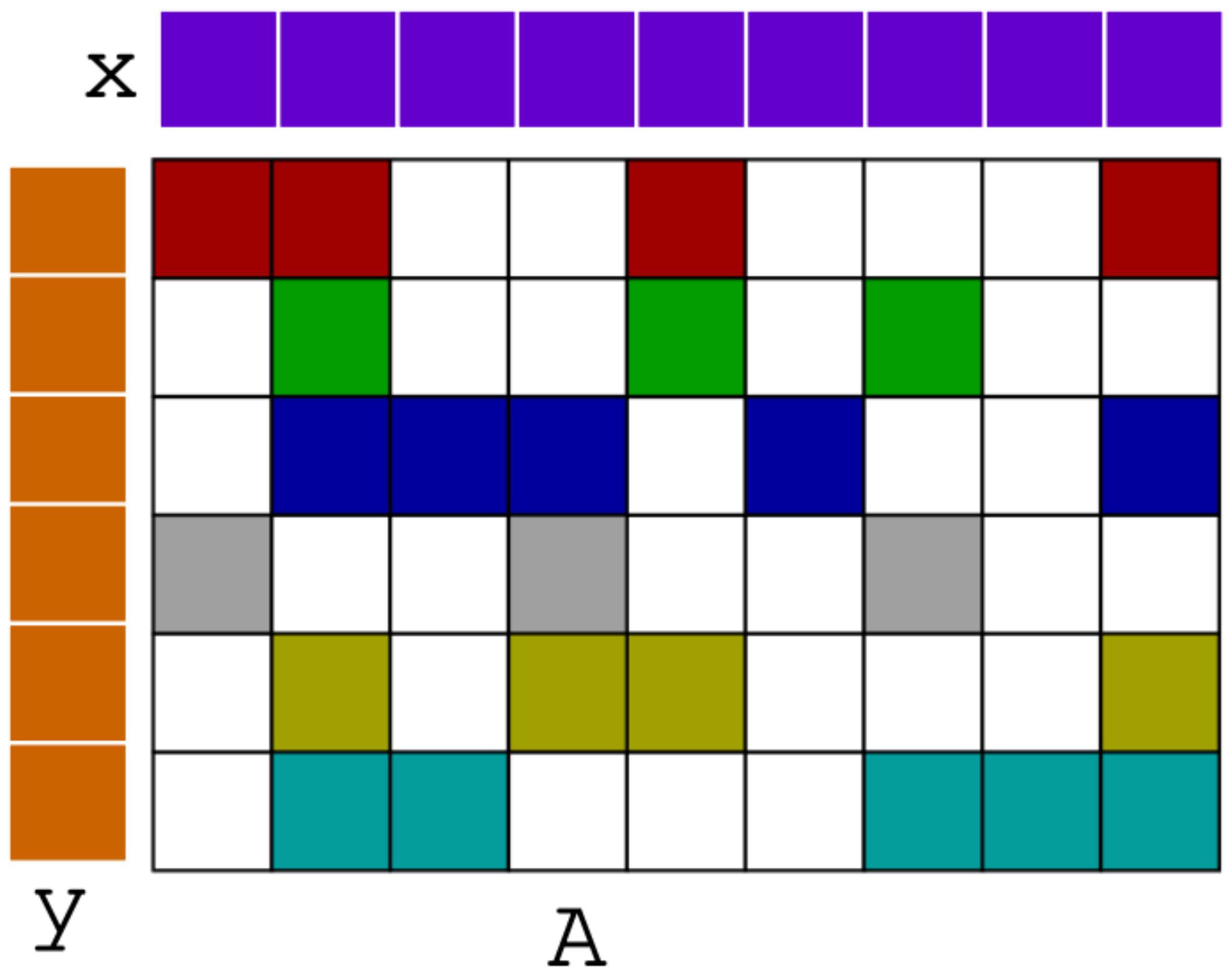
- No reuse in A
- Maximum reuse in y as written
- Reuse in x?

From UC Berkeley CS267

Parallel SpMV

SpMV in CSR: OpenMP Parallel

```
#pragma omp parallel num_threads(thread_num)
{
#pragma omp for private(j, i, tmp) schedule(static)
  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]
    }
  }
}
```

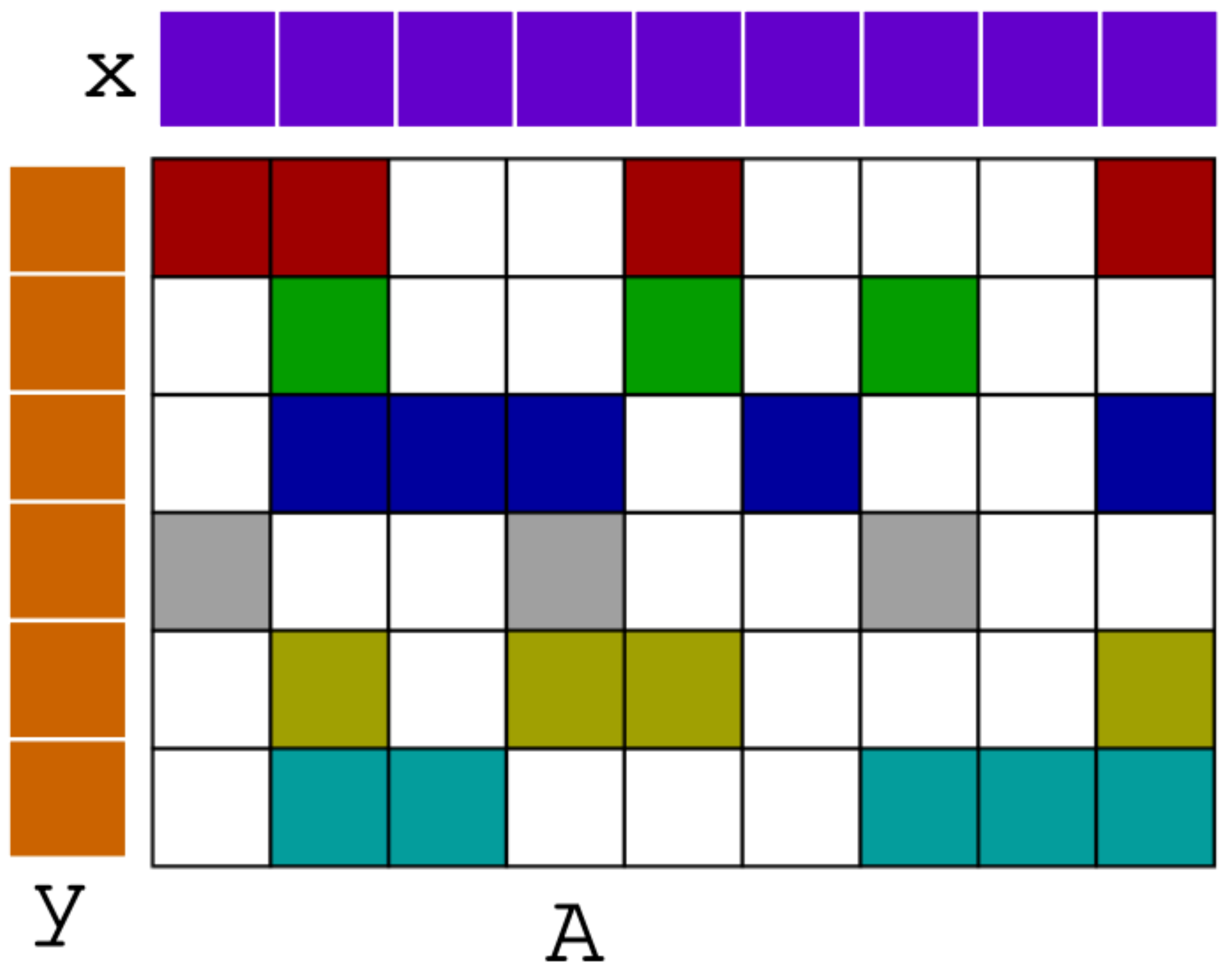


From UC Berkeley CS267

SpMV in CSR: OpenMP Parallel

```
#pragma omp parallel num_threads(thread_num)
{
#pragma omp for private(j, i, tmp) schedule(dynamic)
  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]
    }
  }
}
```

May prefer dynamic because rows are not uniformly sized



From UC Berkeley CS267

SpMV for 8-wide SIMD

```
void avx2_csr_spmv( float *A, int32_t *nIdx, int32_t **indices, float *x, int32_t m, float *y) {
  int32_t A_offset = 0;
  for(int32_t i = 0; i < m; i++) {
    int32_t nElem = nIdx[i]; float t = 0.0f;
    __m256 vT = _mm256_set_ps(0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f);
    int32_t smLen = nElem - (nElem & 7);
    for(int32_t j = 0; j < smLen; j+=8) {
      __m256i vIdx = _mm256_loadu_si256((__m256i*)&(indices[i][j]));
      __m256 vX = _mm256_i32gather_ps((float const*)x,vIdx,4);
      __m256 vA = _mm256_loadu_ps(&A[A_offset + j]);
      vT = _mm256_add_ps(vT, _mm256_mul_ps(vX,vA));
    }
    t += sum8(vT);
    for(int32_t j = smLen; j < nElem; j++) {
      int32_t idx = indices[i][j];
      t += x[idx]*A[A_offset + j];
    }
    y[i] = t;
    A_offset += nElem;
  }
}
```

From UC Berkeley CS267

SpMV for 8-wide SIMD

```
void avx2_csr_spmv( float *A, int32_t *nIdx, int32_t **indices, float *x, int32_t m, float *y) {
  int32_t A_offset = 0;
  for(int32_t i = 0; i < m; i++) {
    int32_t nElem = nIdx[i]; float t = 0.0f;
    __m256 vT = _mm256_set_ps(0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f);
    int32_t smLen = nElem - (nElem & 7);
    for(int32_t j = 0; j < smLen; j+=8) {
      __m256i vIdx = _mm256_loadu_si256((__m256i*)&(indices[i][j]));
      __m256 vX = _mm256_i32gather_ps((float const*)x,vIdx,4);
      __m256 vA = _mm256_loadu_ps(&A[A_offset + j]);
      vT = _mm256_add_ps(vT, _mm256_mul_ps(vX,vA));
    }
    t += sum8(vT);
    for(int32_t j = smLen; j < nElem; j++) {
      int32_t idx = indices[i][j];
      t += x[idx]*A[A_offset + j];
    }
    y[i] = t;
    A_offset += nElem;
  }
}
```

Iterate over rows

From UC Berkeley CS267

SpMV for 8-wide SIMD

```
void avx2_csr_spmv( float *A, int32_t *nIdx, int32_t **indices, float *x, int32_t m, float *y) {
  int32_t A_offset = 0;
  for(int32_t i = 0; i < m; i++) {
    int32_t nElem = nIdx[i]; float t = 0.0f;
    __m256 vT = _mm256_set_ps(0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f);
    int32_t smLen = nElem - (nElem & 7);
    for(int32_t j = 0; j < smLen; j+=8) {
      __m256i vIdx = _mm256_loadu_si256((__m256i*)&(indices[i][j]));
      __m256 vX = _mm256_i32gather_ps((float const*)x,vIdx,4);
      __m256 vA = _mm256_loadu_ps(&A[A_offset + j]);
      vT = _mm256_add_ps(vT, _mm256_mul_ps(vX,vA));
    }
    t += sum8(vT);
    for(int32_t j = smLen; j < nElem; j++) {
      int32_t idx = indices[i][j];
      t += x[idx]*A[A_offset + j];
    }
    y[i] = t;
    A_offset += nElem;
  }
}
```

Iterate over rows

For each 8 nonzeroes

From UC Berkeley CS267

SpMV for 8-wide SIMD

```
void avx2_csr_spmv( float *A, int32_t *nIdx, int32_t **indices, float *x, int32_t m, float *y) {
  int32_t A_offset = 0;
  for(int32_t i = 0; i < m; i++) {
    int32_t nElem = nIdx[i]; float t = 0.0f;
    __m256 vT = _mm256_set_ps(0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f);
    int32_t smLen = nElem - (nElem & 7);
    for(int32_t j = 0; j < smLen; j+=8) {
      __m256i vIdx = _mm256_loadu_si256((__m256i*)&(indices[i][j]));
      __m256 vX = _mm256_i32gather_ps((float const*)x,vIdx,4);
      __m256 vA = _mm256_loadu_ps(&A[A_offset + j]);
      vT = _mm256_add_ps(vT, _mm256_mul_ps(vX,vA));
    }
    t += sum8(vT);
    for(int32_t j = smLen; j < nElem; j++) {
      int32_t idx = indices[i][j];
      t += x[idx]*A[A_offset + j];
    }
    y[i] = t;
    A_offset += nElem;
  }
}
```

Iterate over rows

For each 8 nonzeroes

Gather X, load A, multiply, add to y

From UC Berkeley CS267

SpMV for 8-wide SIMD

```
void avx2_csr_spmv( float *A, int32_t *nIdx, int32_t **indices, float *x, int32_t m, float *y) {
  int32_t A_offset = 0;
  for(int32_t i = 0; i < m; i++) {
    int32_t nElem = nIdx[i]; float t = 0.0f;
    __m256 vT = _mm256_set_ps(0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f);
    int32_t smLen = nElem - (nElem & 7);
    for(int32_t j = 0; j < smLen; j+=8) {
      __m256i vIdx = _mm256_loadu_si256((__m256i*)&(indices[i][j]));
      __m256 vX = _mm256_i32gather_ps((float const*)x,vIdx,4);
      __m256 vA = _mm256_loadu_ps(&A[A_offset + j]);
      vT = _mm256_add_ps(vT, _mm256_mul_ps(vX,vA));
    }
    t += sum8(vT);
    for(int32_t j = smLen; j < nElem; j++) {
      int32_t idx = indices[i][j];
      t += x[idx]*A[A_offset + j];
    }
    y[i] = t;
    A_offset += nElem;
  }
}
```

Iterate over rows

For each 8 nonzeroes

Gather X, load A, multiply, add to y

Sum elements in vector

From UC Berkeley CS267

SpMV for 8-wide SIMD

```
void avx2_csr_spmv( float *A, int32_t *nIdx, int32_t **indices, float *x, int32_t m, float *y) {
  int32_t A_offset = 0;
  for(int32_t i = 0; i < m; i++) {
    int32_t nElem = nIdx[i]; float t = 0.0f;
    __m256 vT = _mm256_set_ps(0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f);
    int32_t smLen = nElem - (nElem & 7);
    for(int32_t j = 0; j < smLen; j+=8) {
      __m256i vIdx = _mm256_loadu_si256((__m256i*)&(indices[i][j]));
      __m256 vX = _mm256_i32gather_ps((float const*)x,vIdx,4);
      __m256 vA = _mm256_loadu_ps(&A[A_offset + j]);
      vT = _mm256_add_ps(vT, _mm256_mul_ps(vX,vA));
    }
    t += sum8(vT);
    for(int32_t j = smLen; j < nElem; j++) {
      int32_t idx = indices[i][j];
      t += x[idx]*A[A_offset + j];
    }
    y[i] = t;
    A_offset += nElem;
  }
}
```

Iterate over rows

For each 8 nonzeroes

Gather X, load A, multiply, add to y

Sum elements in vector

In case #cols does not evenly divide vector width

From UC Berkeley CS267

SpMV with CSR in CUDA

```
// Parallel SpMV using CSR format
__global__ void spmv(int *ptr, int *ind, float *val,
                     int m, float *x, float *y) {
  for (int i = blockIdx.x * blockDim.x + threadIdx.x;
                     i < m; i += blockDim.x * gridDim.x) {
    float yi = 0;
    for (int j = ptr[i]; j < ptr[i+1]; j++) {
      yi += values[j] * x[col_ind[j]];
    }
    y[i] = yi;
  }
}
```

Iterate over rows in A

From UC Berkeley CS267

SpMV with CSR in CUDA

```
// Parallel SpMV using CSR format
__global__ void spmv(int *ptr, int *ind, float *val,
               int m, float *x, float *y) {
  for (int i = blockIdx.x * blockDim.x + threadIdx.x;
               i < m; i += blockDim.x * gridDim.x) {
    float yi = 0;
    for (int j = ptr[i]; j < ptr[i+1]; j++) {
      yi += values[j] * x[col_ind[j]];
    }
    y[i] = yi;
  }
}
```

Iterate over rows in A

Iterate over nonzeroes in row

From UC Berkeley CS267

SpMV with CSR in CUDA

```
// Parallel SpMV using CSR format
__global__ void spmv(int *ptr, int *ind, float *val,
               int m, float *x, float *y) {
  for (int i = blockIdx.x * blockDim.x + threadIdx.x;
               i < m; i += blockDim.x * gridDim.x) {
    float yi = 0;
    for (int j = ptr[i]; j < ptr[i+1]; j++) {
      yi += values[j] * x[col_ind[j]];
    }
    y[i] = yi;
  }
}
```

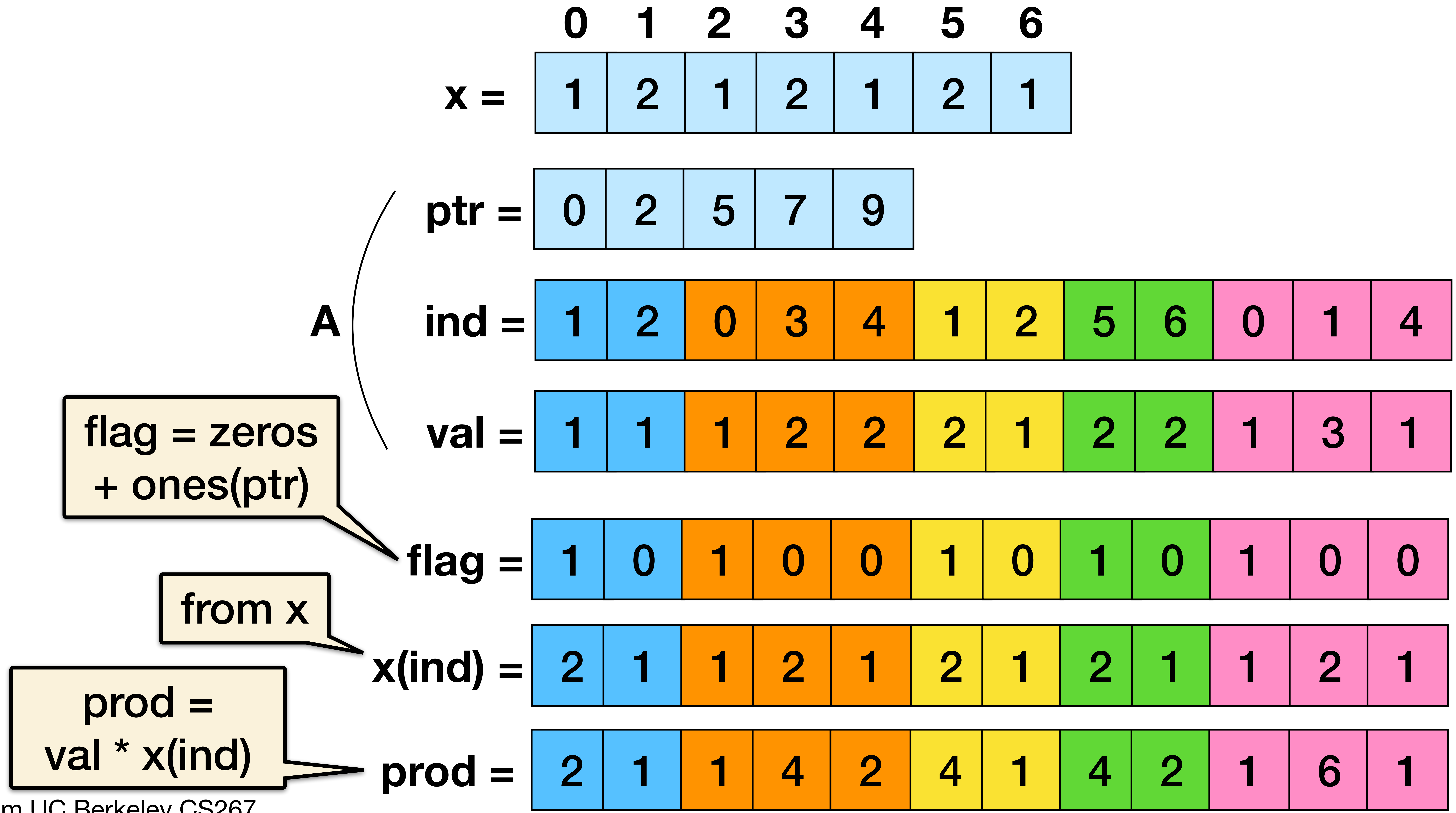
Iterate over rows in A

Iterate over nonzeroes in row

Assign results

From UC Berkeley CS267

Segmented Suffix Scan



	0	1	2	3	4	5	6
x =	1	2	1	2	1	2	1

A

ptr =	0	2	5	7	9

ind =	1	2	0	3	4	1	2	5	6	0	1	4
val =	1	1	1	2	2	2	1	2	2	1	3	1
flag =	1	0	1	0	0	1	0	1	0	1	0	0
x(ind) =	2	1	1	2	1	2	1	2	1	1	2	1
prod =	2	1	1	4	2	4	1	4	2	1	6	1

flag = zeros + ones(ptr)

from x

prod = val * x(ind)

From UC Berkeley CS267

Segmented Suffix Scan

	0	1	2	3	4	5	6					
x =	1	2	1	2	1	2	1					
ptr =	0	2	5	7	9							
ind =	1	2	0	3	4	1	2	5	6	0	1	4
val =	1	1	1	2	2	2	1	2	2	1	3	1
flag =	1	0	1	0	0	1	0	1	0	1	0	0
prod =	2	1	1	4	2	4	1	4	2	1	6	1
sums =	2	3	1	5	7	4	5	4	6	1	7	8

flag = zeros + ones(ptr)

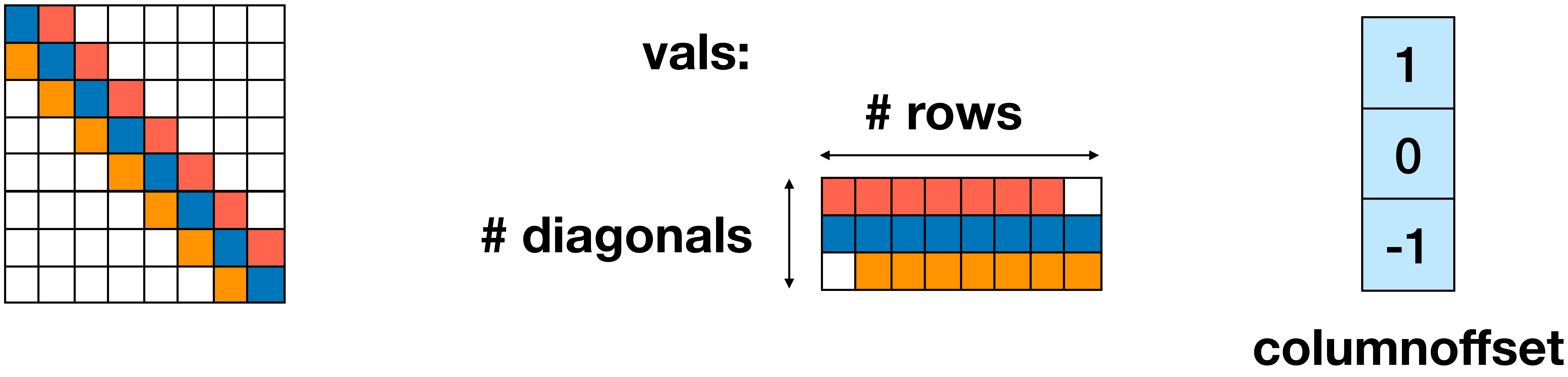
prod = val * x(ind)

segmented scan on flag, prod

y = end of segments

From UC Berkeley CS267

SpMV Diagonal Format in OpenMP



vals:

# rows

# diagonals

1

0

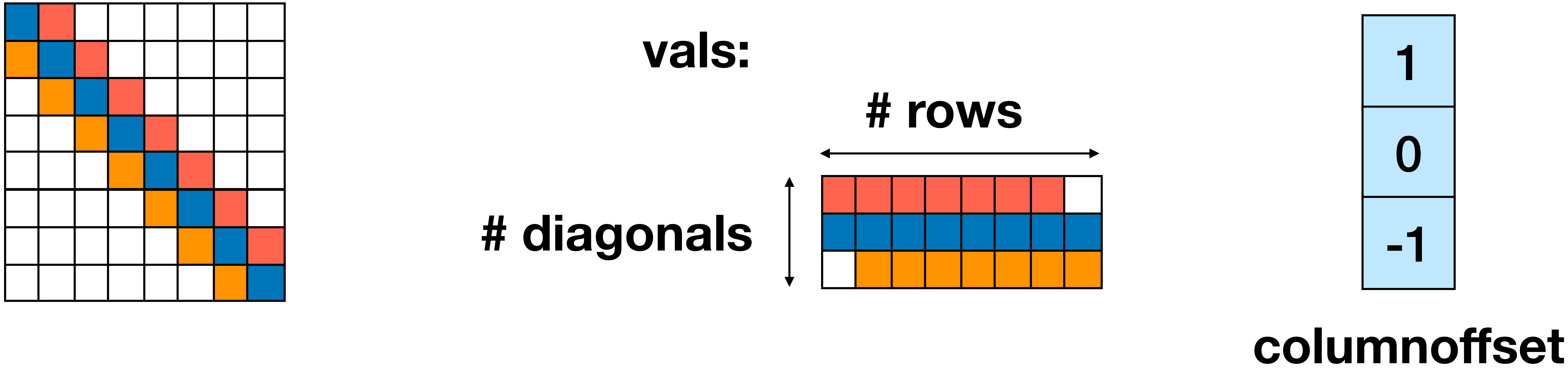
-1

columnoffset

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

From UC Berkeley CS267

SpMV Diagonal Format in OpenMP



```
for each diagonal k do
  #pragma omp parallel for
  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

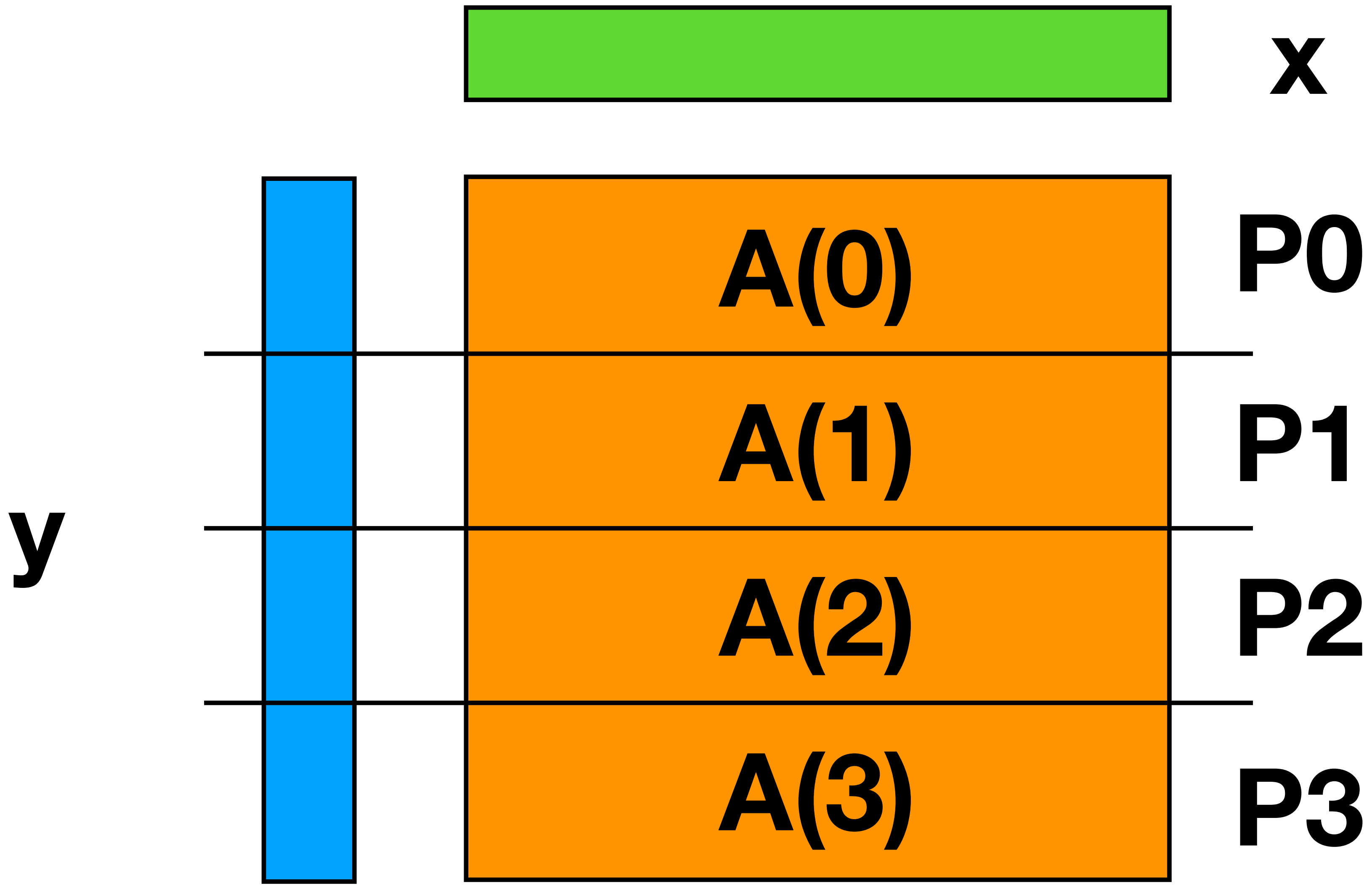
From UC Berkeley CS267

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:



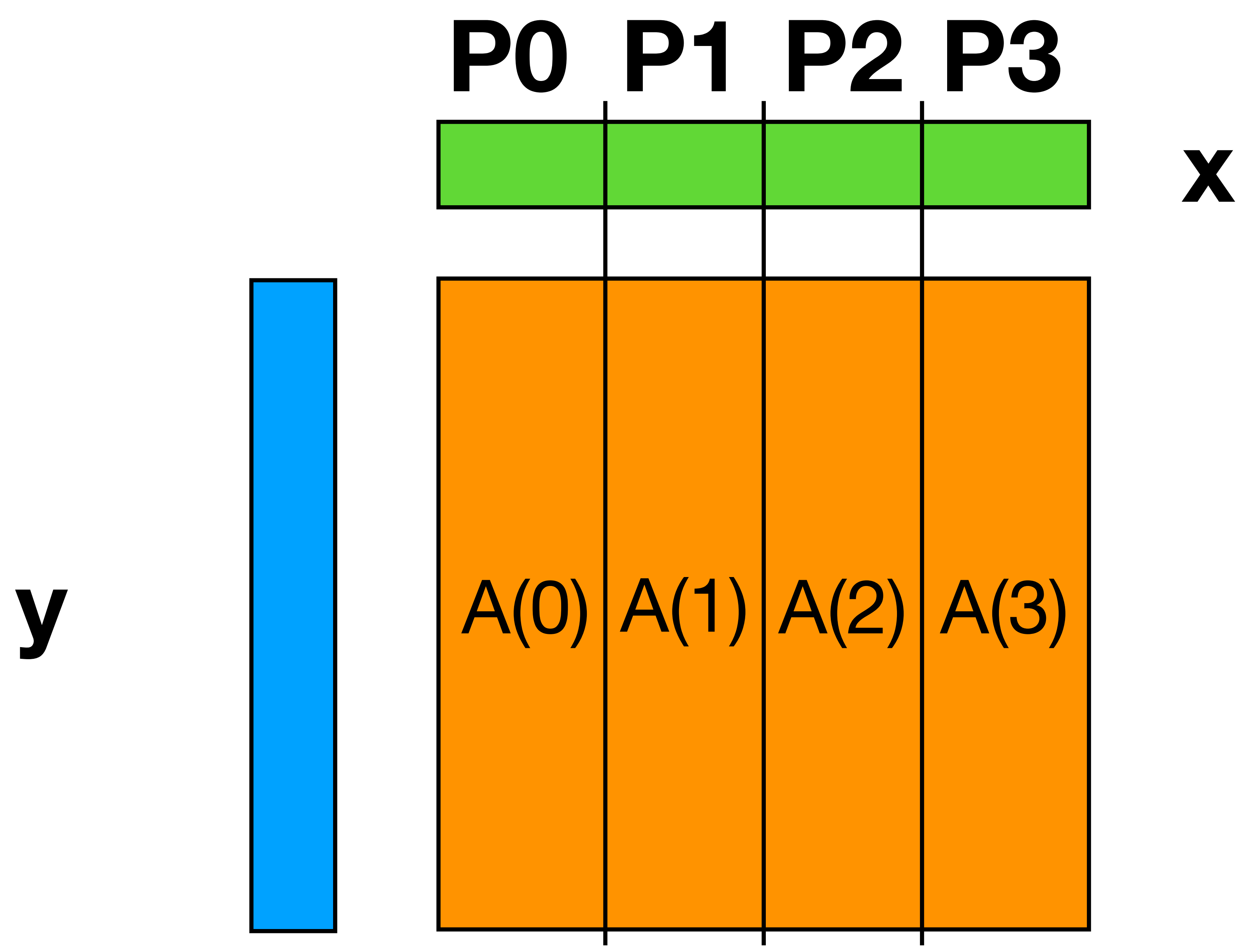
```
foreach processor p
  Broadcast x[p] chunk owned by p

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

Broadcast + local dot product

From UC Berkeley CS267

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

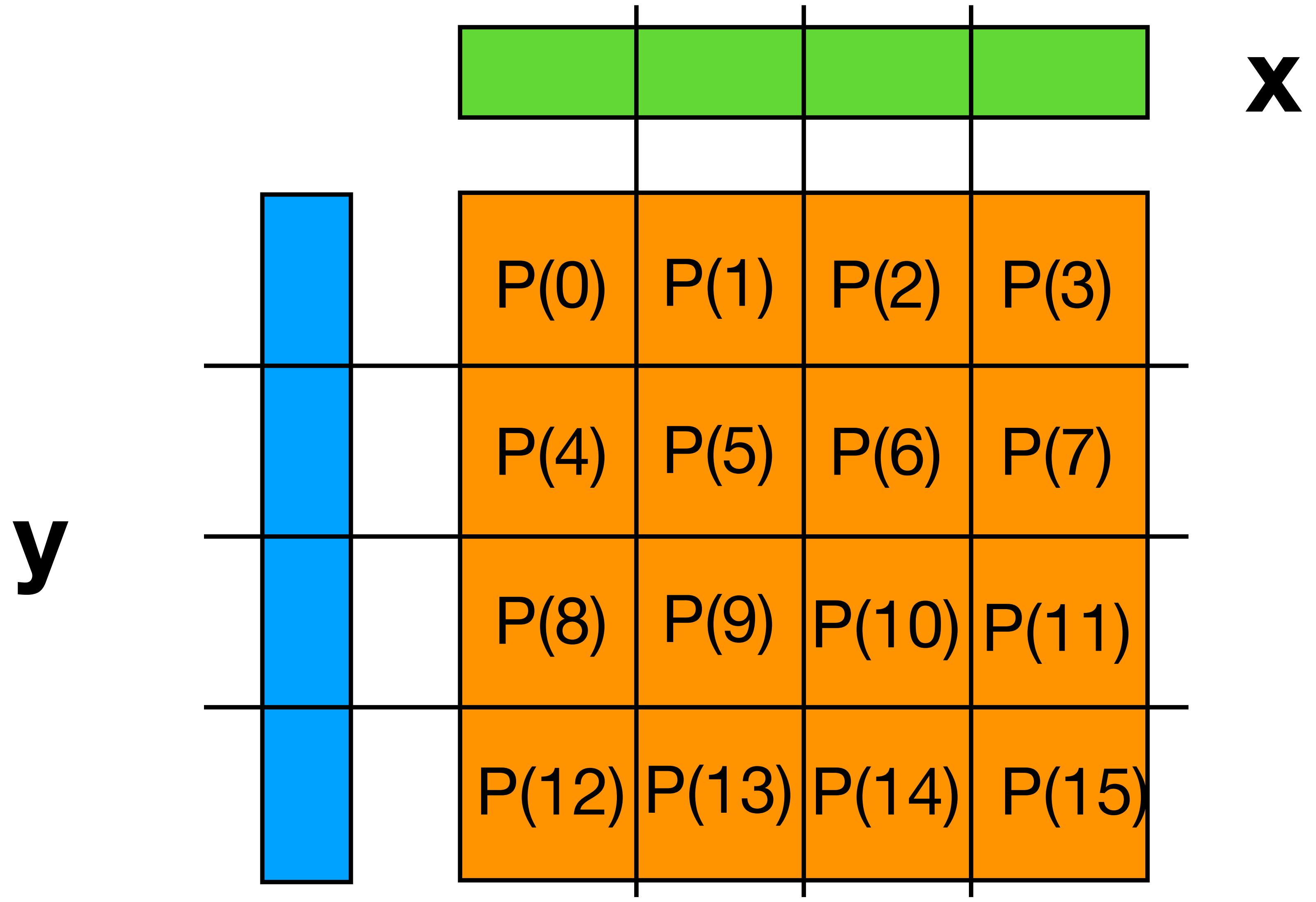
From UC Berkeley CS267

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

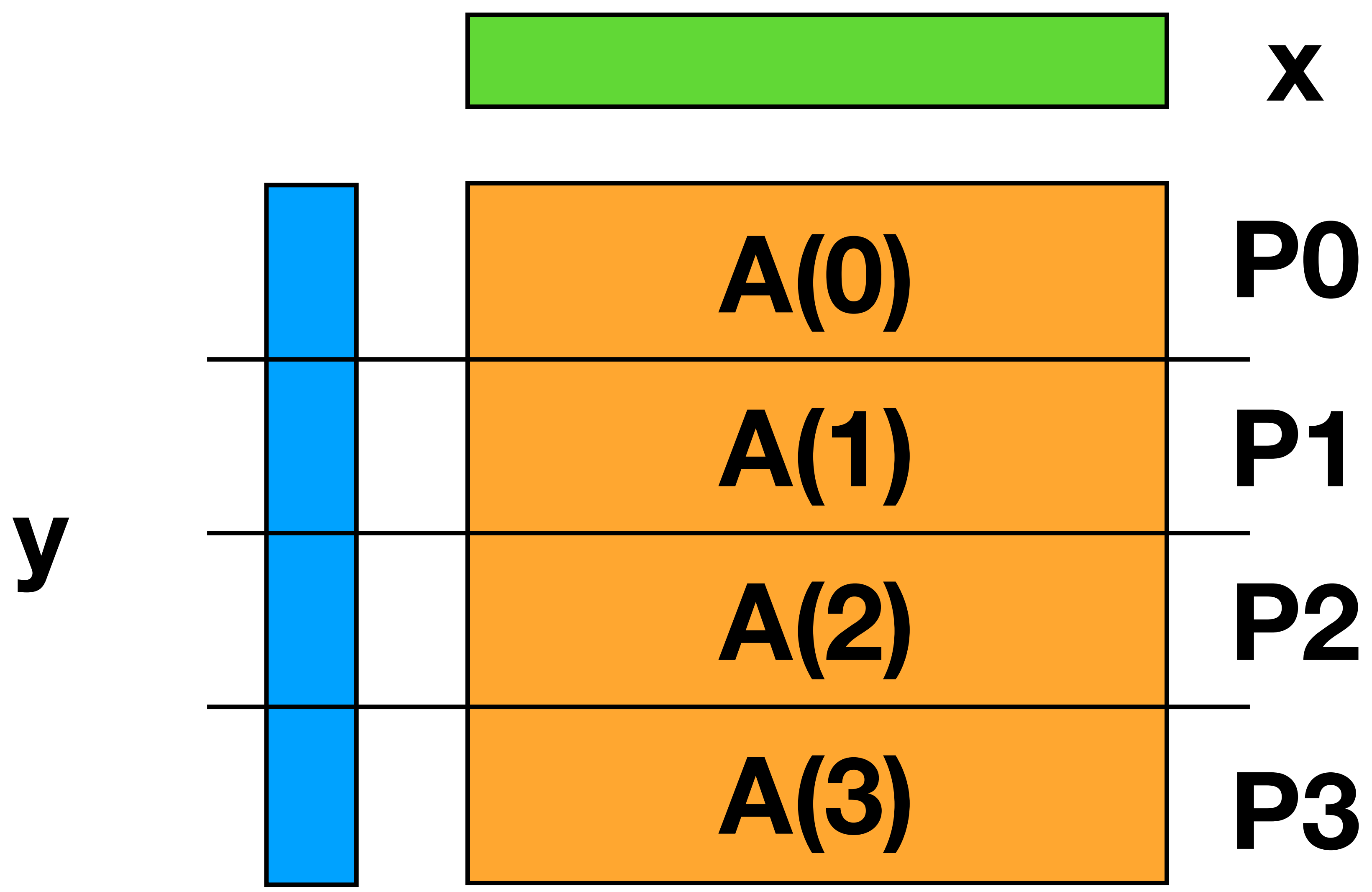


From UC Berkeley CS267

Distributed SpMV

Row parallelism (y & A partitioned)

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

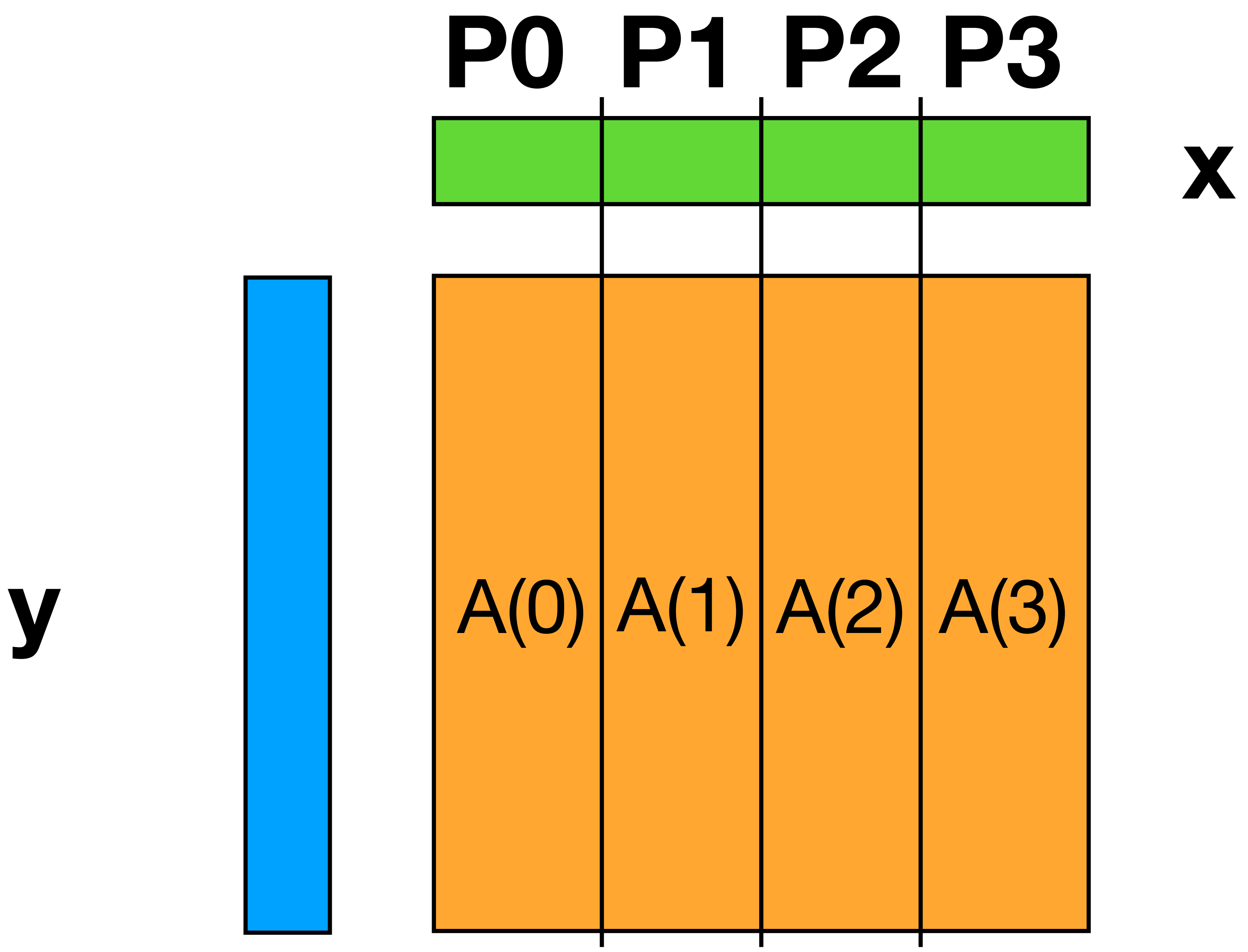


From UC Berkeley CS267

Distributed SpMV

Column parallelism (x & A partitioned)

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

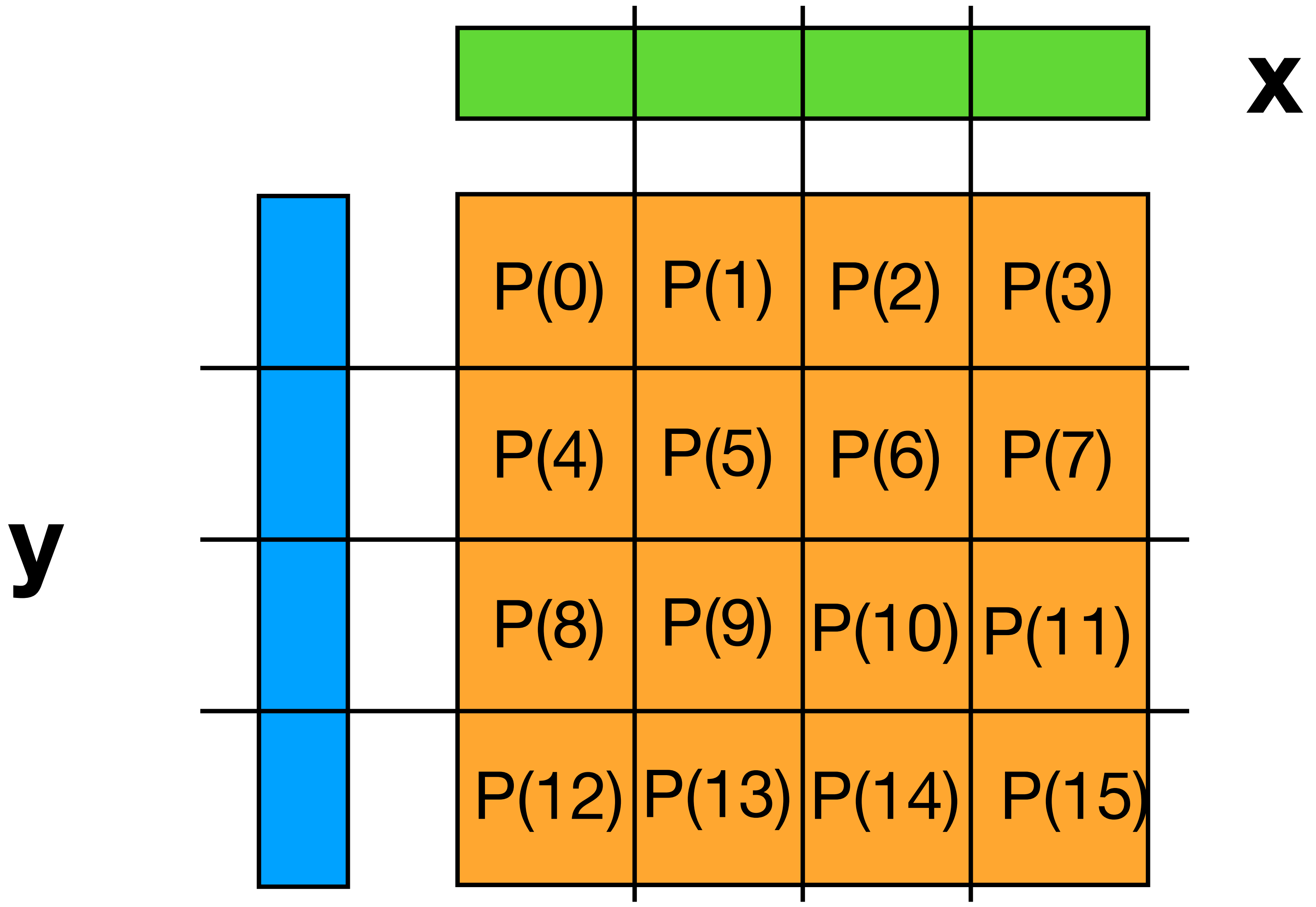


From UC Berkeley CS267

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



P(0)	P(1)	P(2)	P(3)
P(4)	P(5)	P(6)	P(7)
P(8)	P(9)	P(10)	P(11)
P(12)	P(13)	P(14)	P(15)

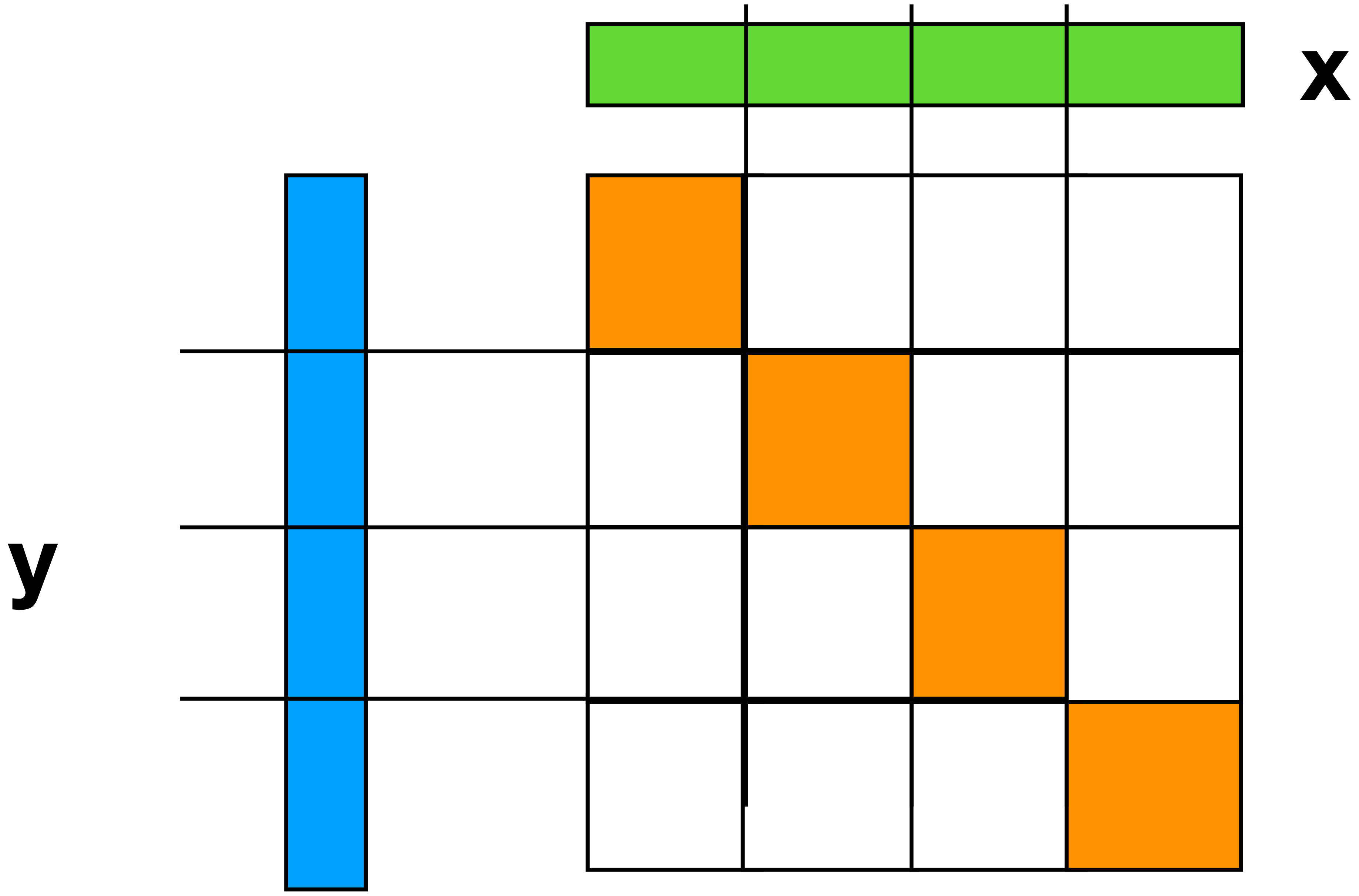
From UC Berkeley CS267

https://www.nas.nasa.gov/software/npb.html

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 non 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.

From UC Berkeley CS267

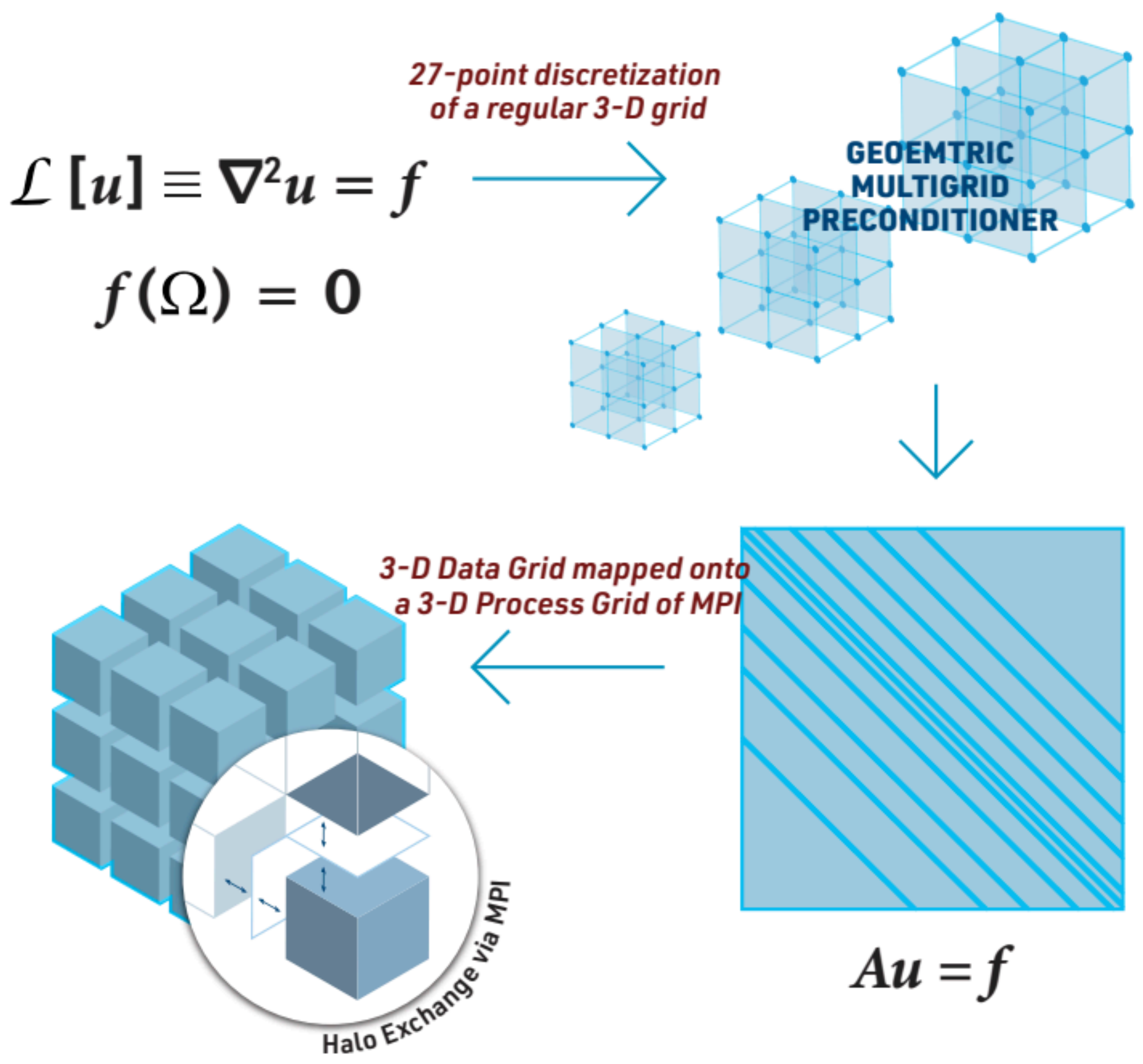
High Performance Conjugate Gradients (HPCG) Benchmark

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

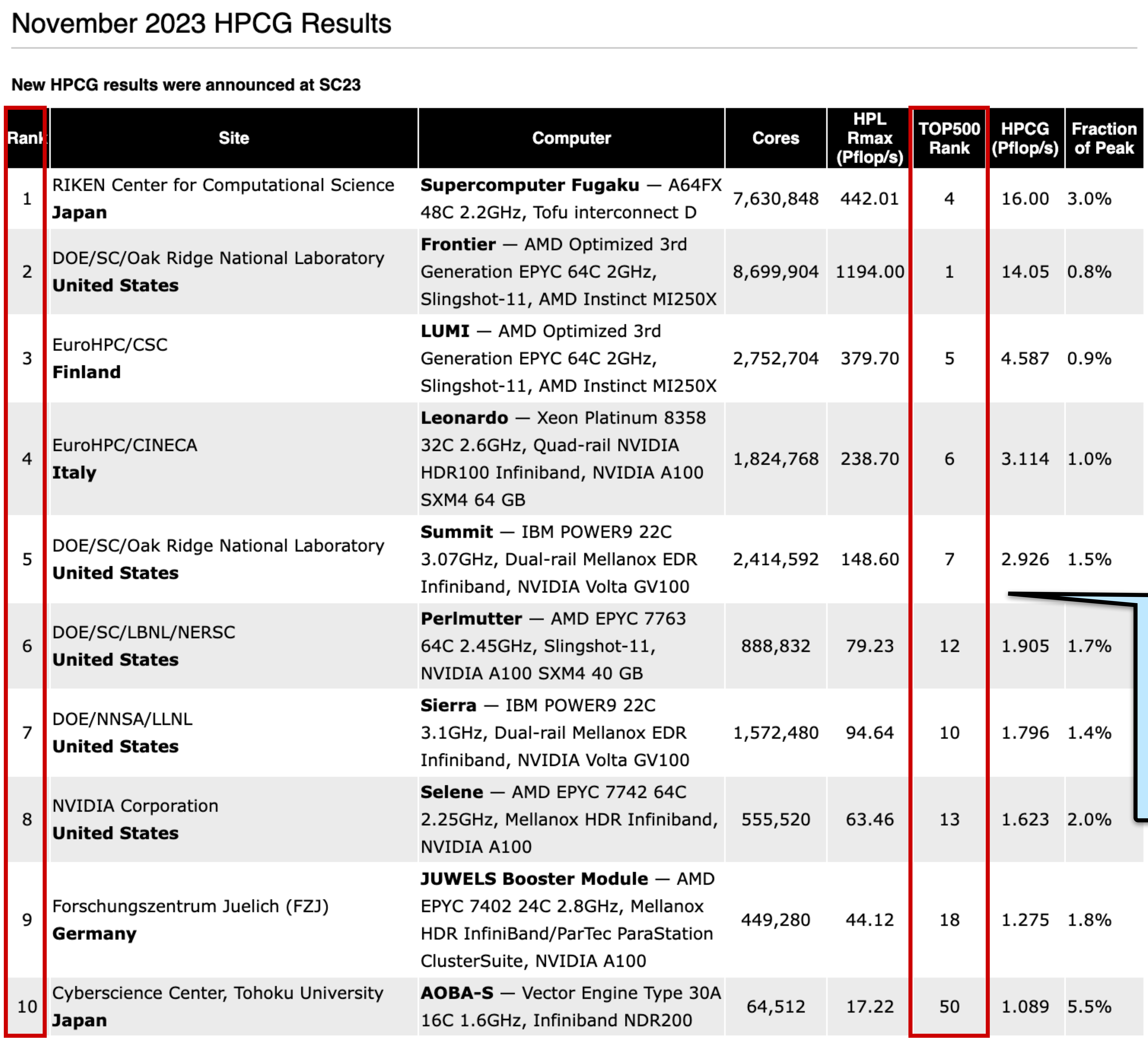
Designed to exercise computational and data access patterns that more closely match a **different and broader set of applications**.

HPCG includes performance of the following basic operations:

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


$\mathcal{L}[u] \equiv \nabla^2 u = f$
$f(\Omega) = 0$
27-point discretization of a regular 3-D grid
GEOEMTRIC MULTIGRID PRECONDITIONER
3-D Data Grid mapped onto a 3-D Process Grid of MPI
Halo Exchange via MPI
$Au = f$

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



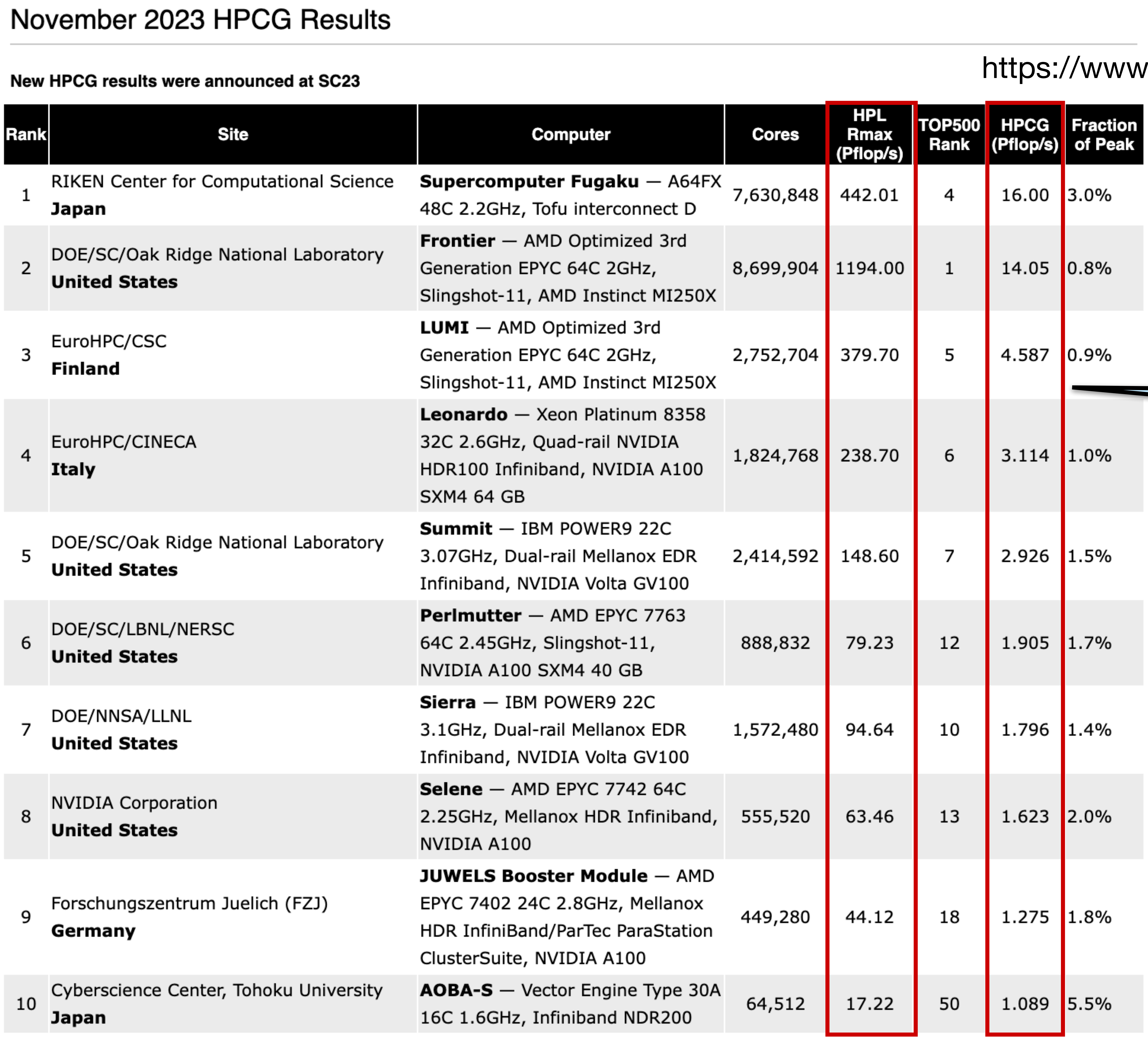
November 2023 HPCG Results

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

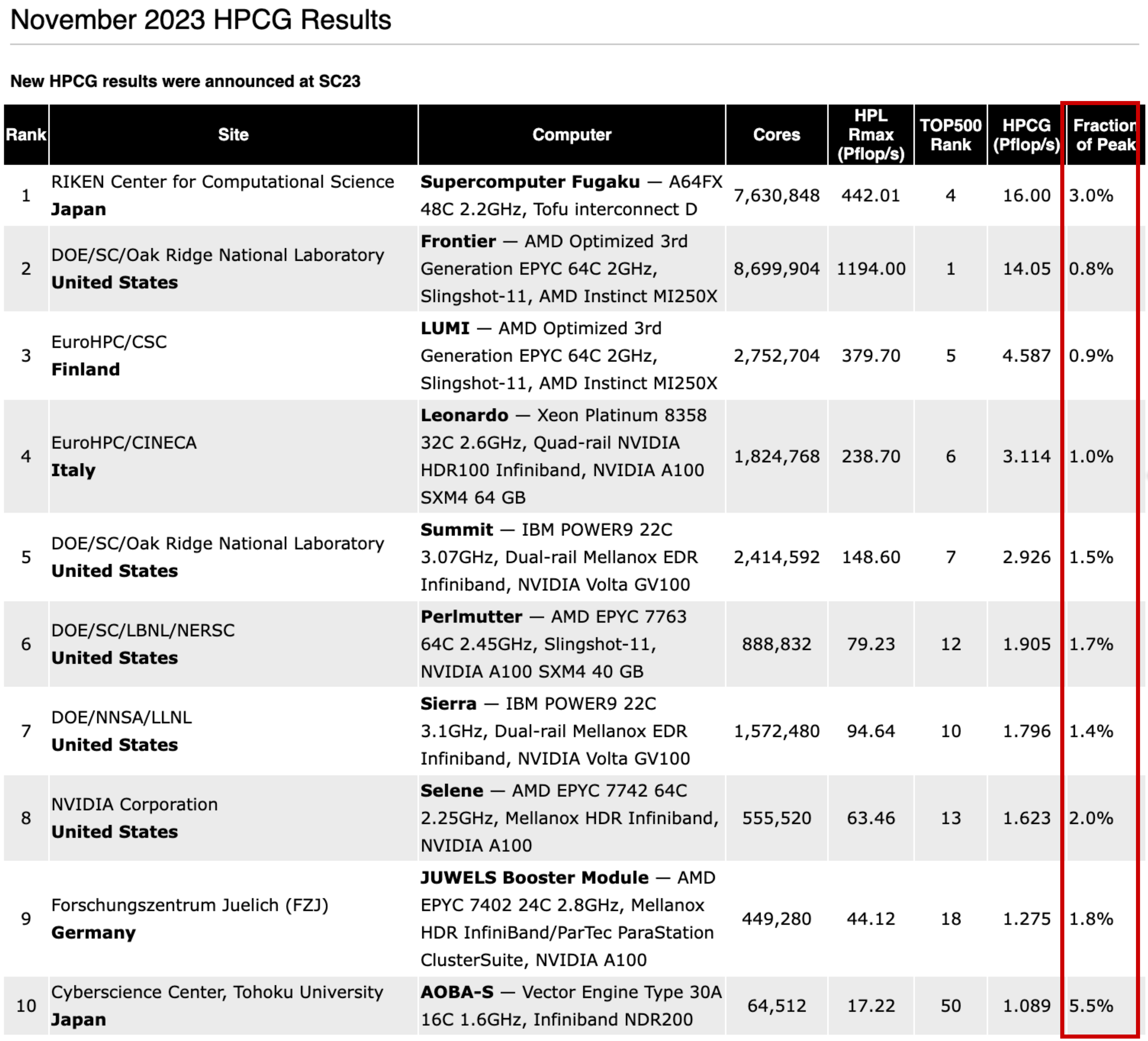
New HPCG results were announced at SC23

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%

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

Much lower Pflop/s compared to Linpack

From UC Berkeley CS267



November 2023 HPCG Results

New HPCG results were announced at SC23

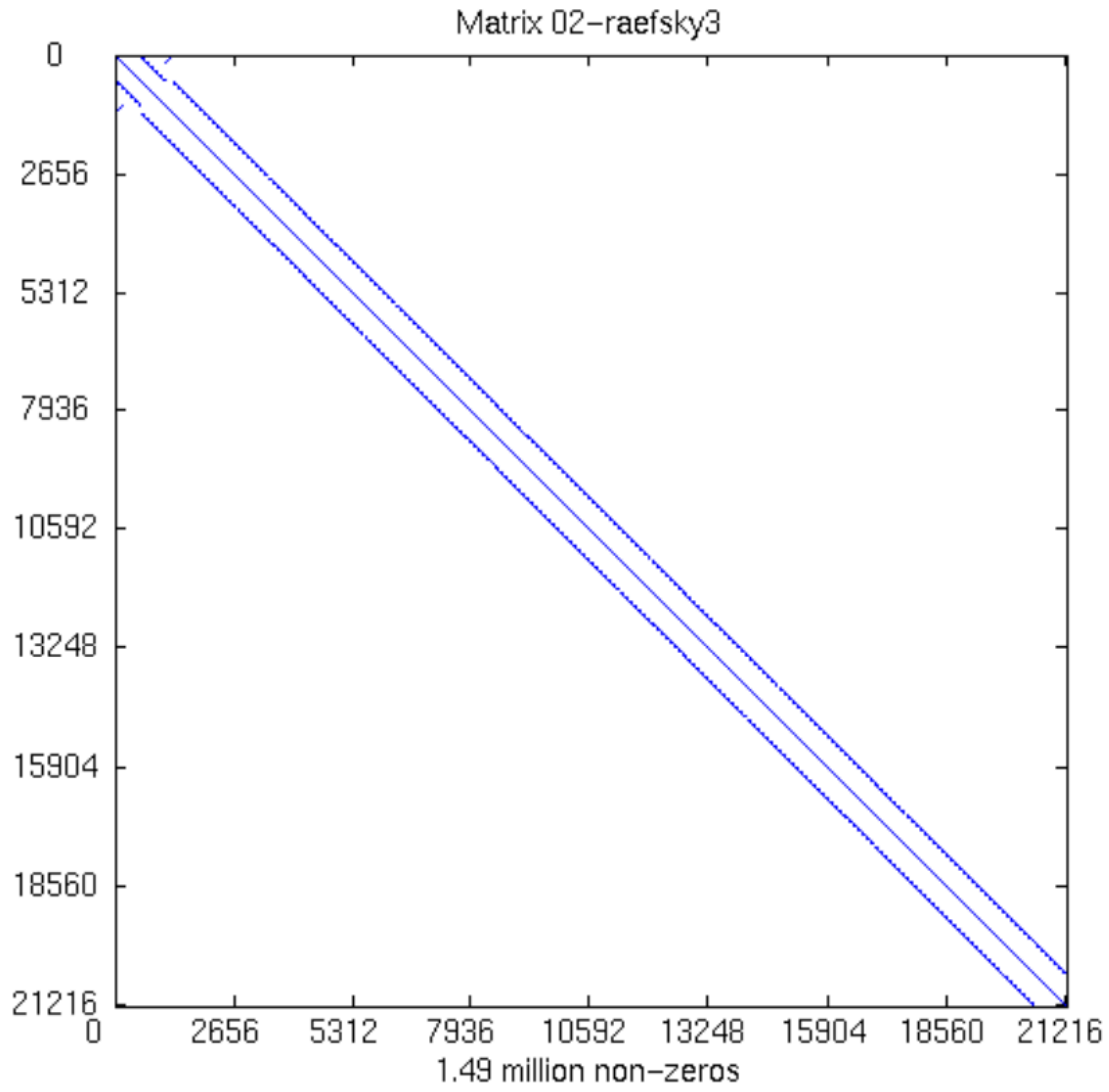
Rank	Site	Computer	Cores	HPL Rmax (Pflop/s)	TOP500 Rank	HPCG (Pflop/s)	Fraction of Peak
1	RIKEN Center for Computational Science **Japan**	**Supercomputer Fugaku** — A64FX 48C 2.2GHz, Tofu interconnect D	7,630,848	442.01	4	16.00	3.0%
2	DOE/SC/Oak Ridge National Laboratory **United States**	**Frontier** — 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

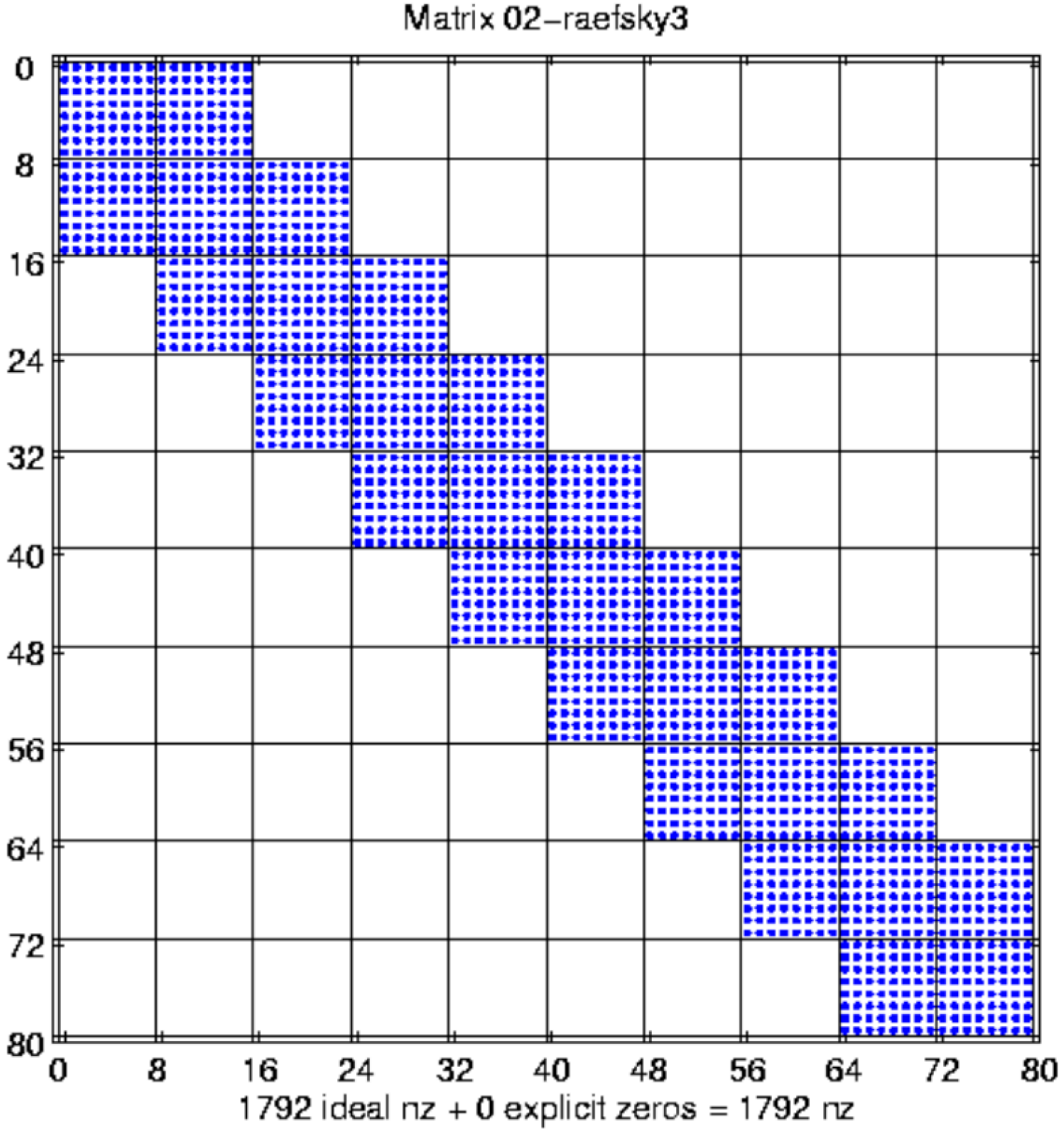
From UC Berkeley CS267

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.

From UC Berkeley CS267

Using block structure in SpMV

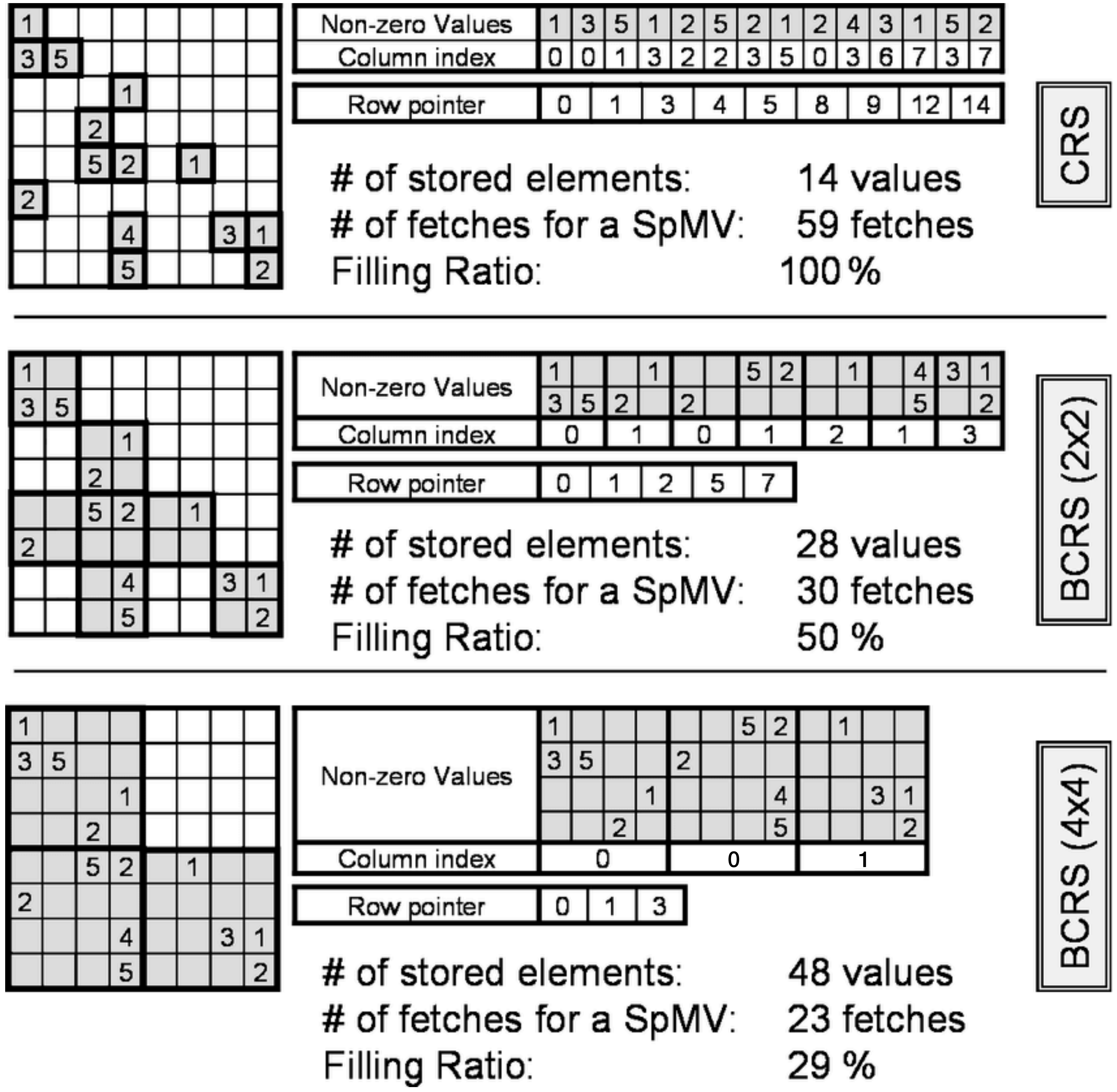
The **bottleneck is the time to fetch** the matrix from memory

- 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



$$\begin{bmatrix} 1 & & & & & & & \\ 3 & 5 & & & & & & \\ & & & 1 & & & & \\ & & 2 & & & & & \\ & & 5 & 2 & & 1 & & \\ 2 & & & & & & & \\ & & & 4 & & & 3 & 1 \\ & & & 5 & & & & 2 \end{bmatrix}$$

Non-zero Values	1	3	5	1	2	5	2	1	2	4	3	1	5	2
Column index	0	0	1	3	2	2	3	5	0	3	6	7	3	7

Row pointer	0	1	3	4	5	8	9	12	14

# of stored elements:	14 values
# of fetches for a SpMV:	59 fetches
Filling Ratio:	100 %

CRS

$$\begin{bmatrix} 1 & & & & & & & \\ 3 & 5 & & & & & & \\ & & & 1 & & & & \\ & & 2 & & & & & \\ & & 5 & 2 & & 1 & & \\ 2 & & & & & & & \\ & & & 4 & & & 3 & 1 \\ & & & 5 & & & & 2 \end{bmatrix}$$

Non-zero Values	1			1			5	2		1		4	3	1
	3	5	2		2							5		2
Column index	0		1		0		1		2		1		3	

Row pointer	0	1	2	5	7

# of stored elements:	28 values
# of fetches for a SpMV:	30 fetches
Filling Ratio:	50 %

BCRS (2x2)

$$\begin{bmatrix} 1 & & & & & & & \\ 3 & 5 & & & & & & \\ & & & 1 & & & & \\ & & 2 & & & & & \\ & & 5 & 2 & & 1 & & \\ 2 & & & & & & & \\ & & & 4 & & & 3 & 1 \\ & & & 5 & & & & 2 \end{bmatrix}$$

Non-zero Values	1						5	2		1		
	3	5			2							
				1				4			3	1
			2					5				2
Column index	0				0				1			

Row pointer	0	1	3

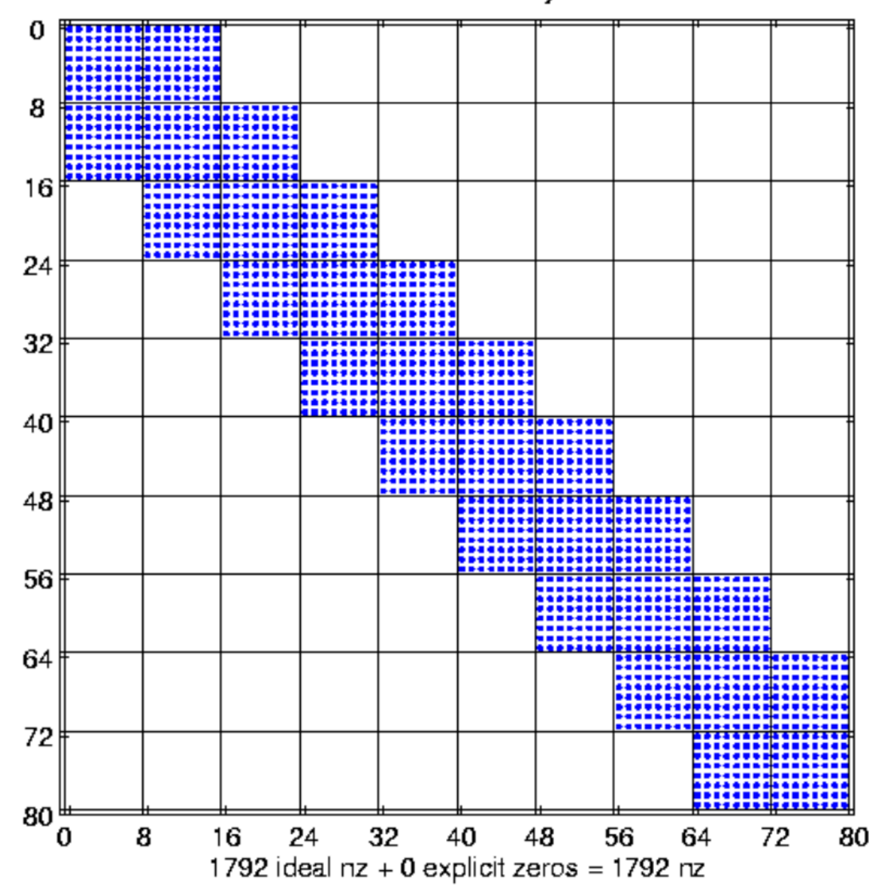
# of stored elements:	48 values
# of fetches for a SpMV:	23 fetches
Filling Ratio:	29 %

BCRS (4x4)

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

From UC Berkeley CS267

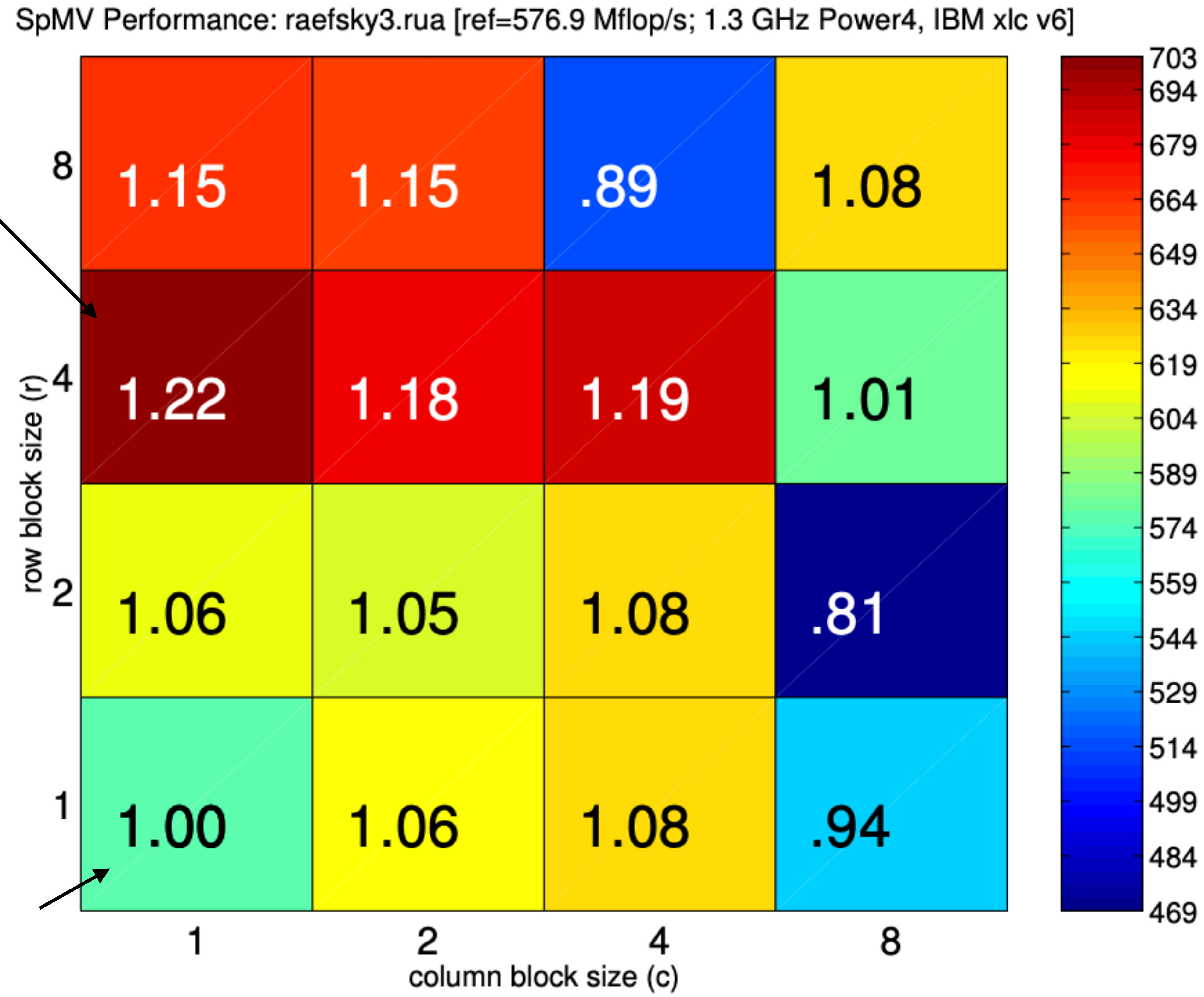
The need for search



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

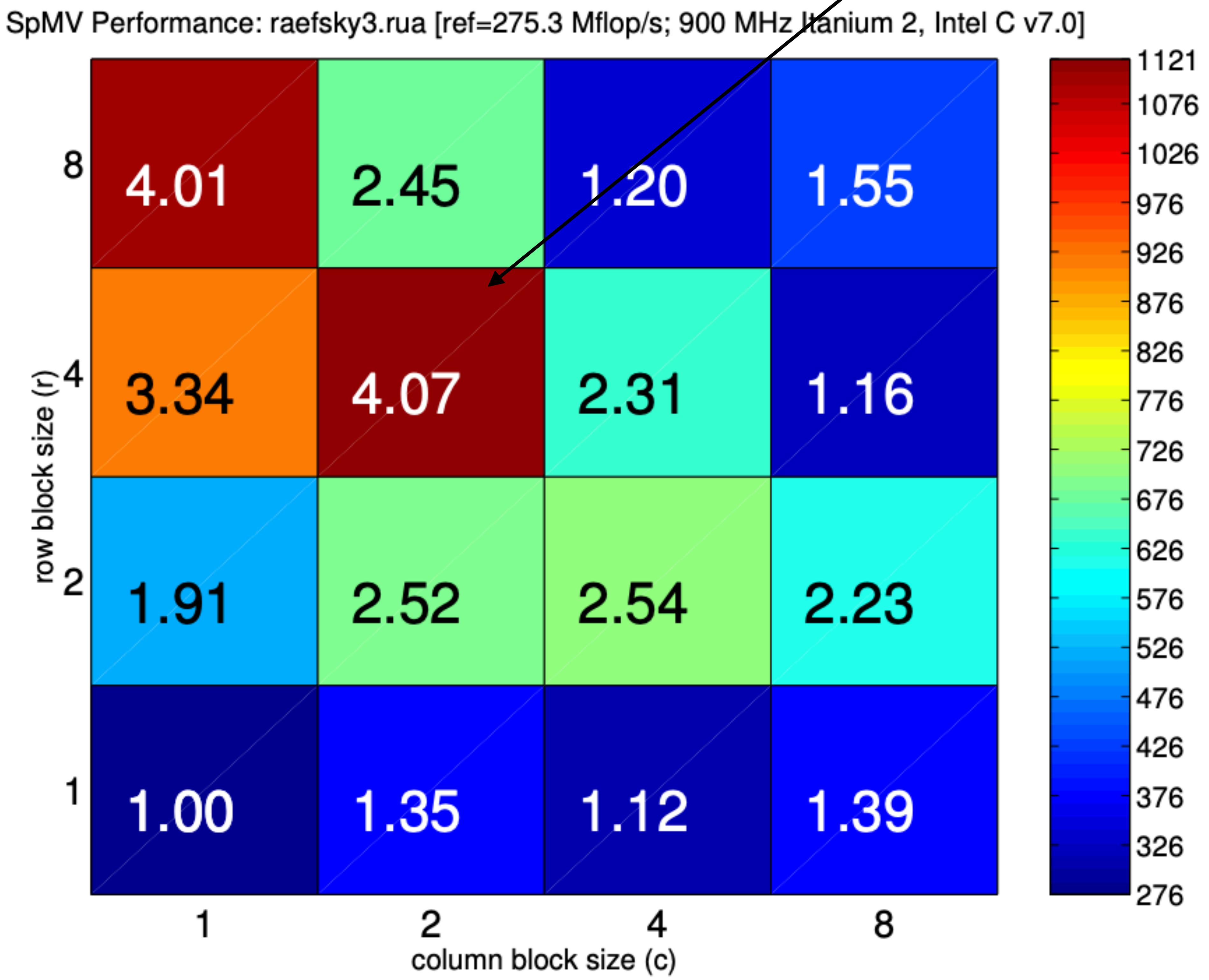
Out of 6 platforms tested, 8x8 is the best on one (coming up)

Best: 4x1



Reference

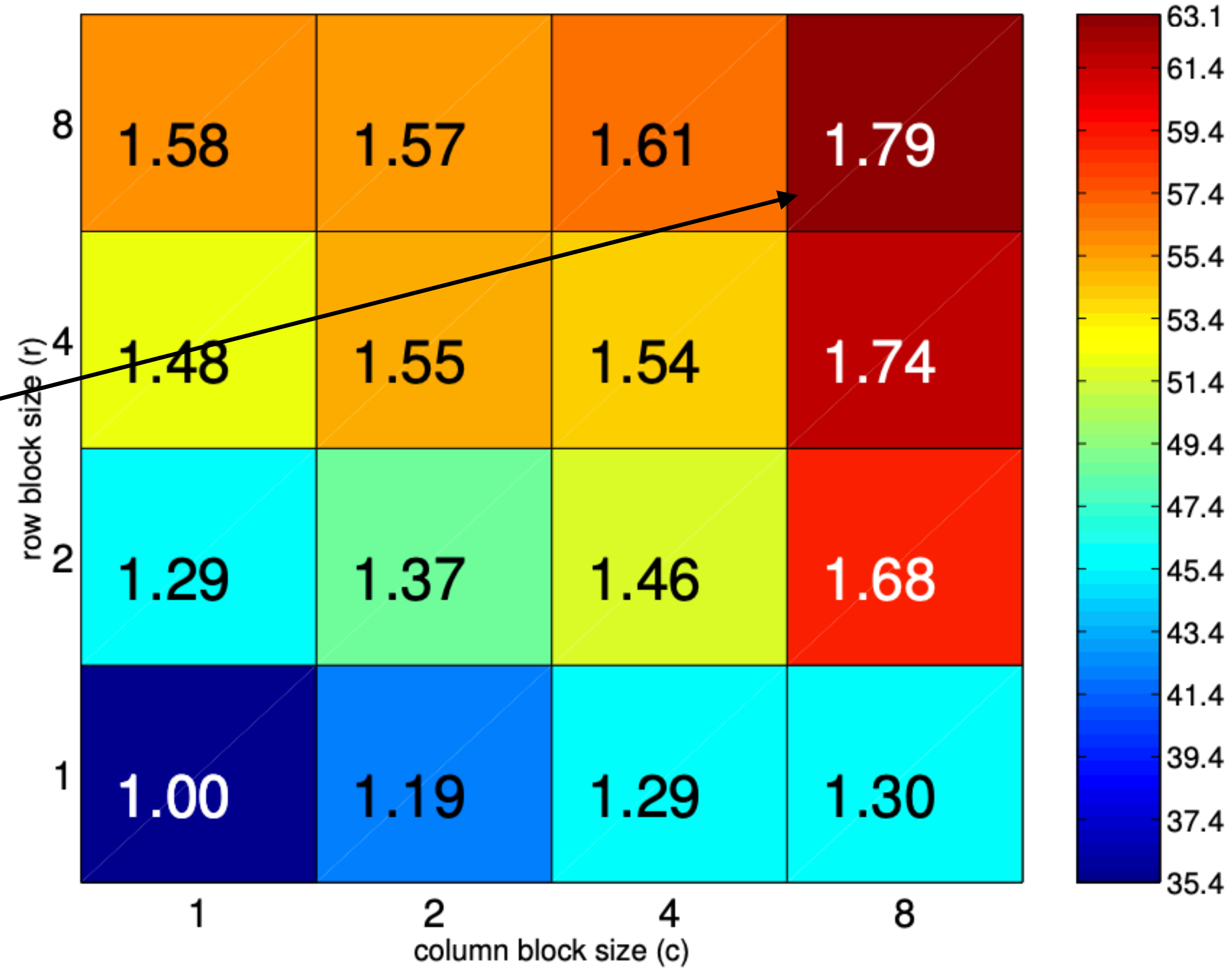
Best: 4x2



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

From UC Berkeley CS267

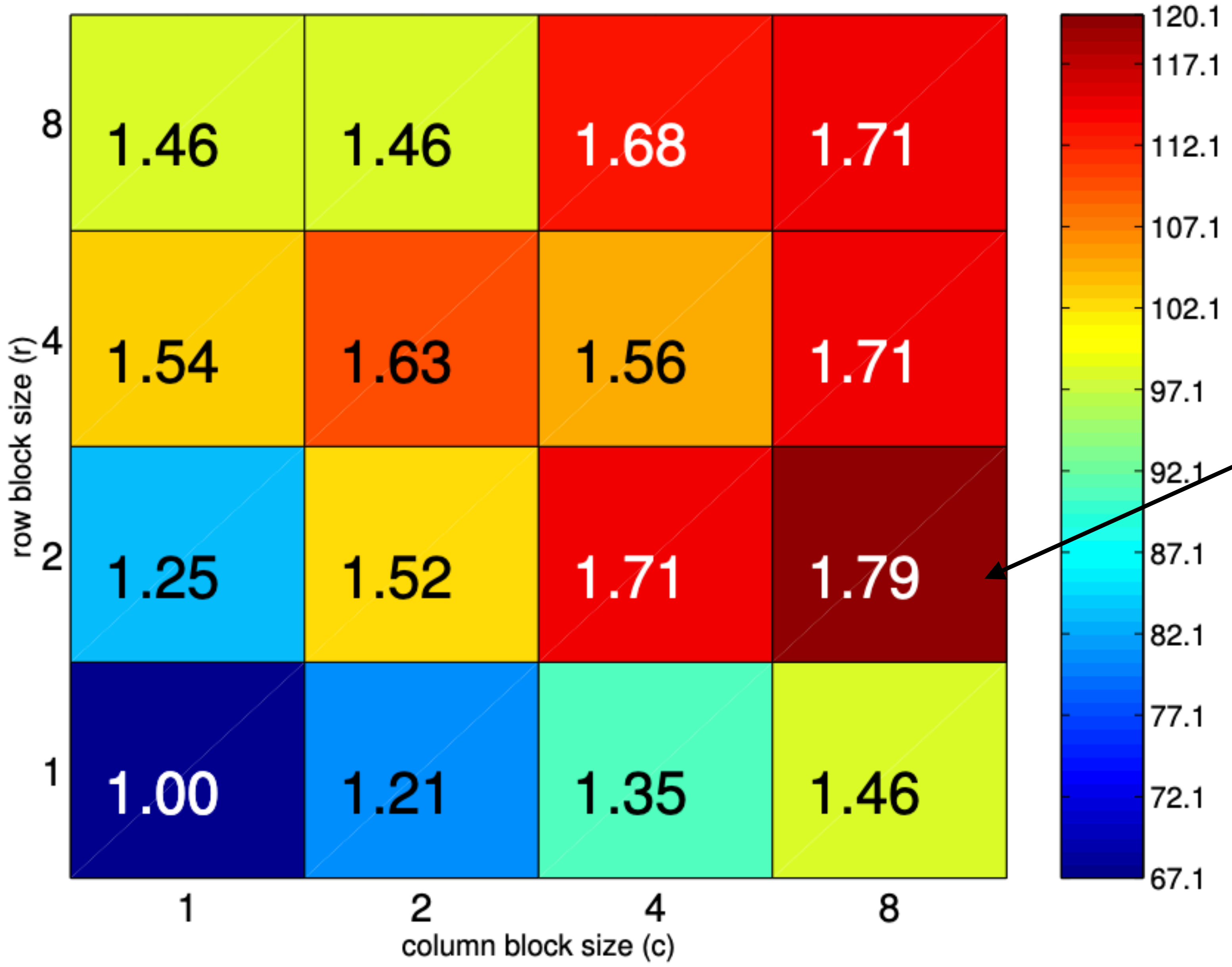
SpMV Performance: raefsky3.rua [ref=35.3 Mflop/s; 333 MHz Sun Ultra 2i, Sun C v6.0]



	1	2	4	8
8	1.58	1.57	1.61	1.79
4	1.48	1.55	1.54	1.74
2	1.29	1.37	1.46	1.68
1	1.00	1.19	1.29	1.30

Best: 8x8

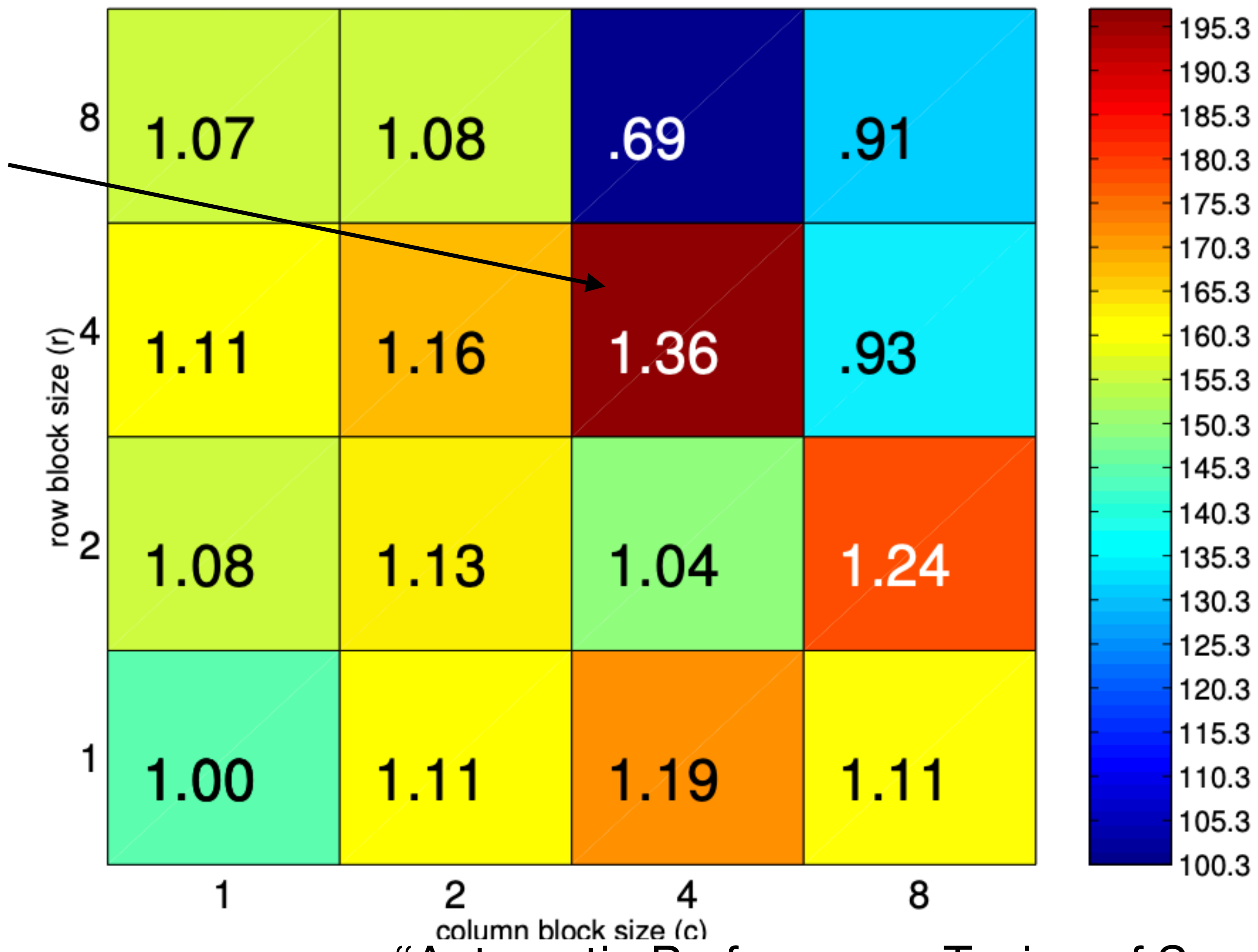
SpMV Performance: raefsky3.rua [ref=67.1 Mflop/s; 800 MHz Pentium III–M, Intel C v7.0]



	1	2	4	8
8	1.46	1.46	1.68	1.71
4	1.54	1.63	1.56	1.71
2	1.25	1.52	1.71	1.79
1	1.00	1.21	1.35	1.46

Best: 2x8

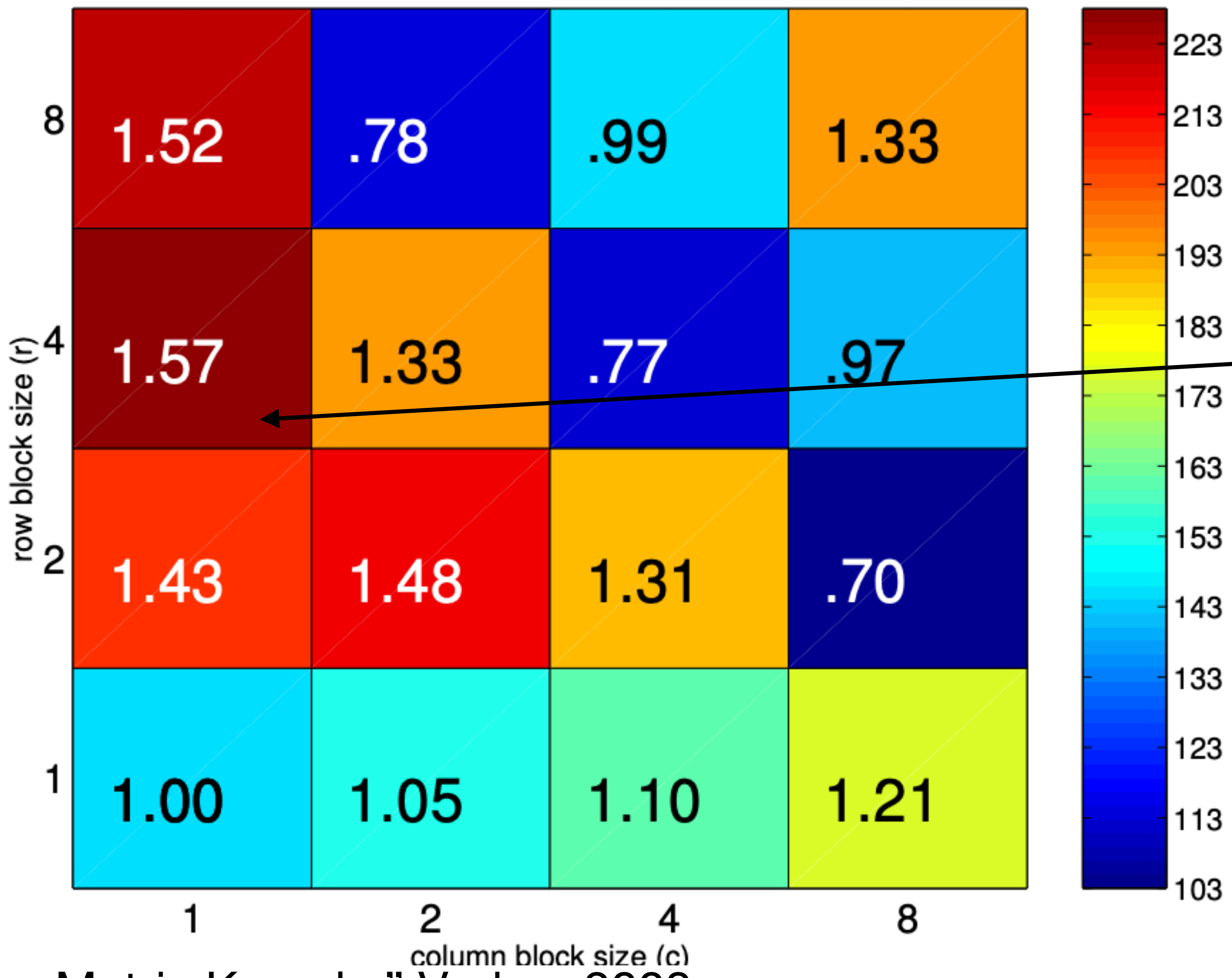
SpMV Performance: raefsky3.rua [ref=144.7 Mflop/s; 375 MHz Power3, IBM xlc v6]



	1	2	4	8
8	1.07	1.08	.69	.91
4	1.11	1.16	1.36	.93
2	1.08	1.13	1.04	1.24
1	1.00	1.11	1.19	1.11

Best: 4x4

SpMV Performance: raefsky3.rua [ref=145.8 Mflop/s; 800 MHz Itanium, Intel C v7]



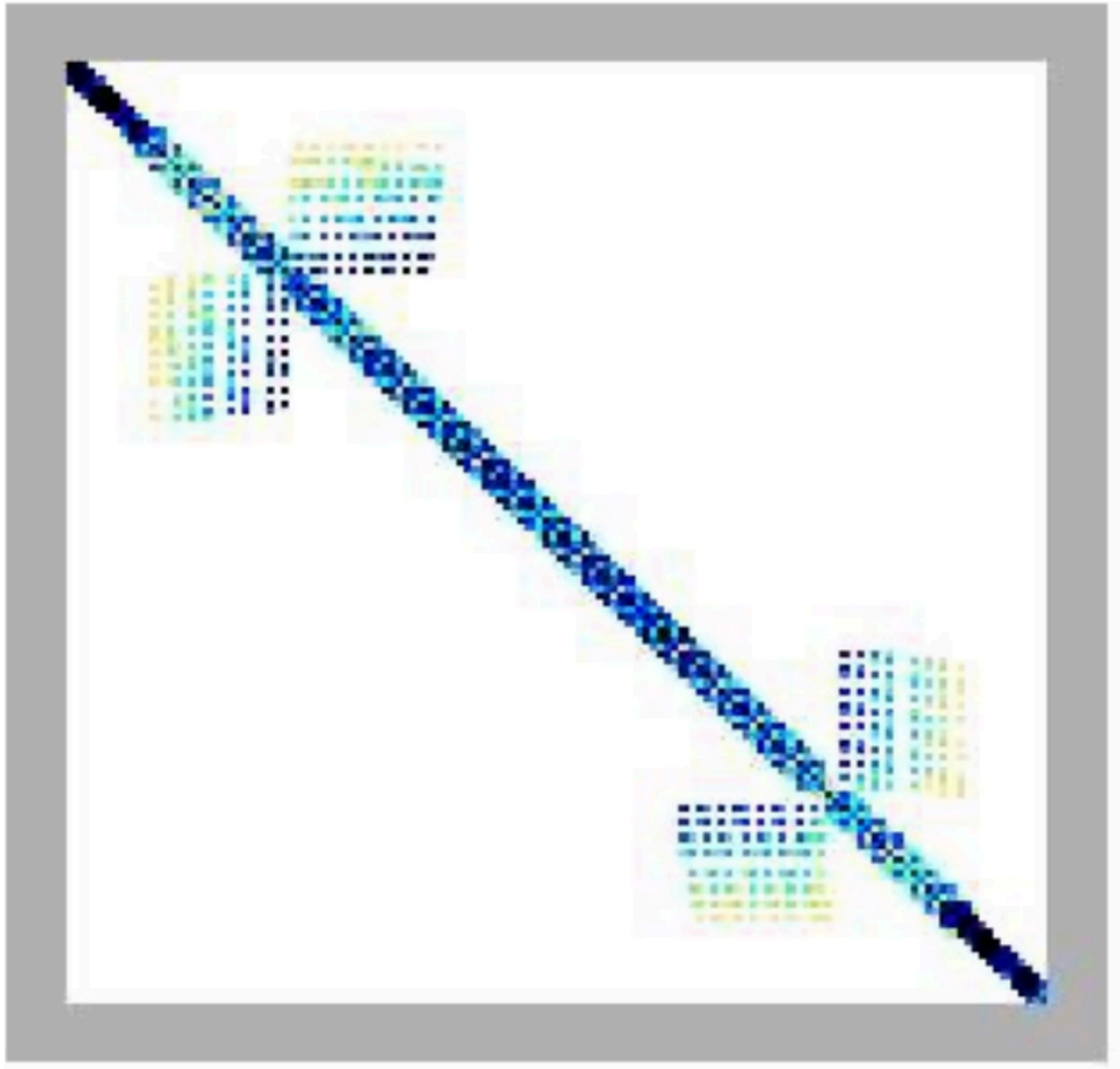
	1	2	4	8
8	1.52	.78	.99	1.33
4	1.57	1.33	.77	.97
2	1.43	1.48	1.31	.70
1	1.00	1.05	1.10	1.21

Best: 4x1

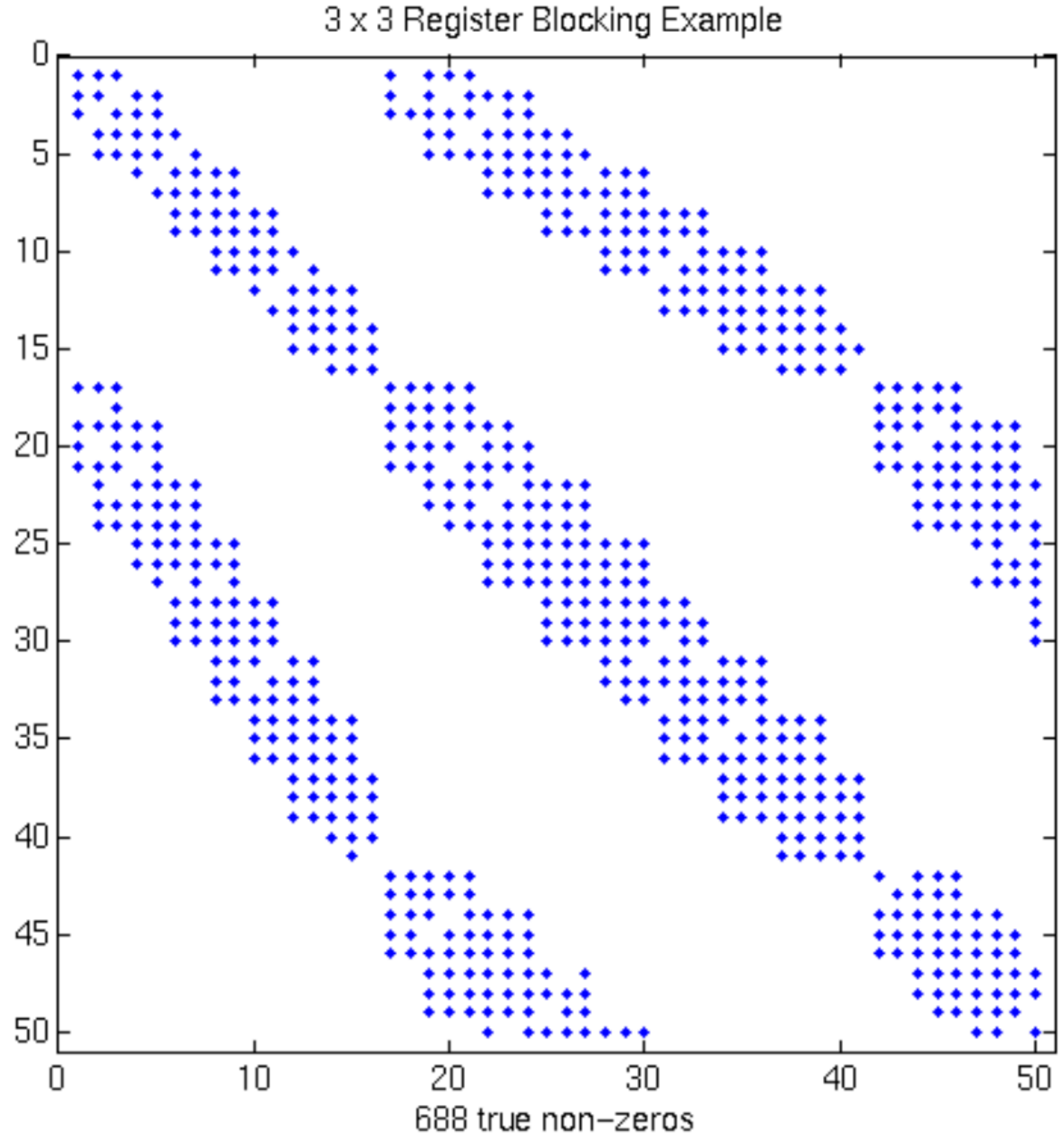
“Automatic Performance Tuning of Sparse Matrix Kernels,” Vuduc, 2003.

From UC Berkeley CS267

But most matrices don’t block so easily



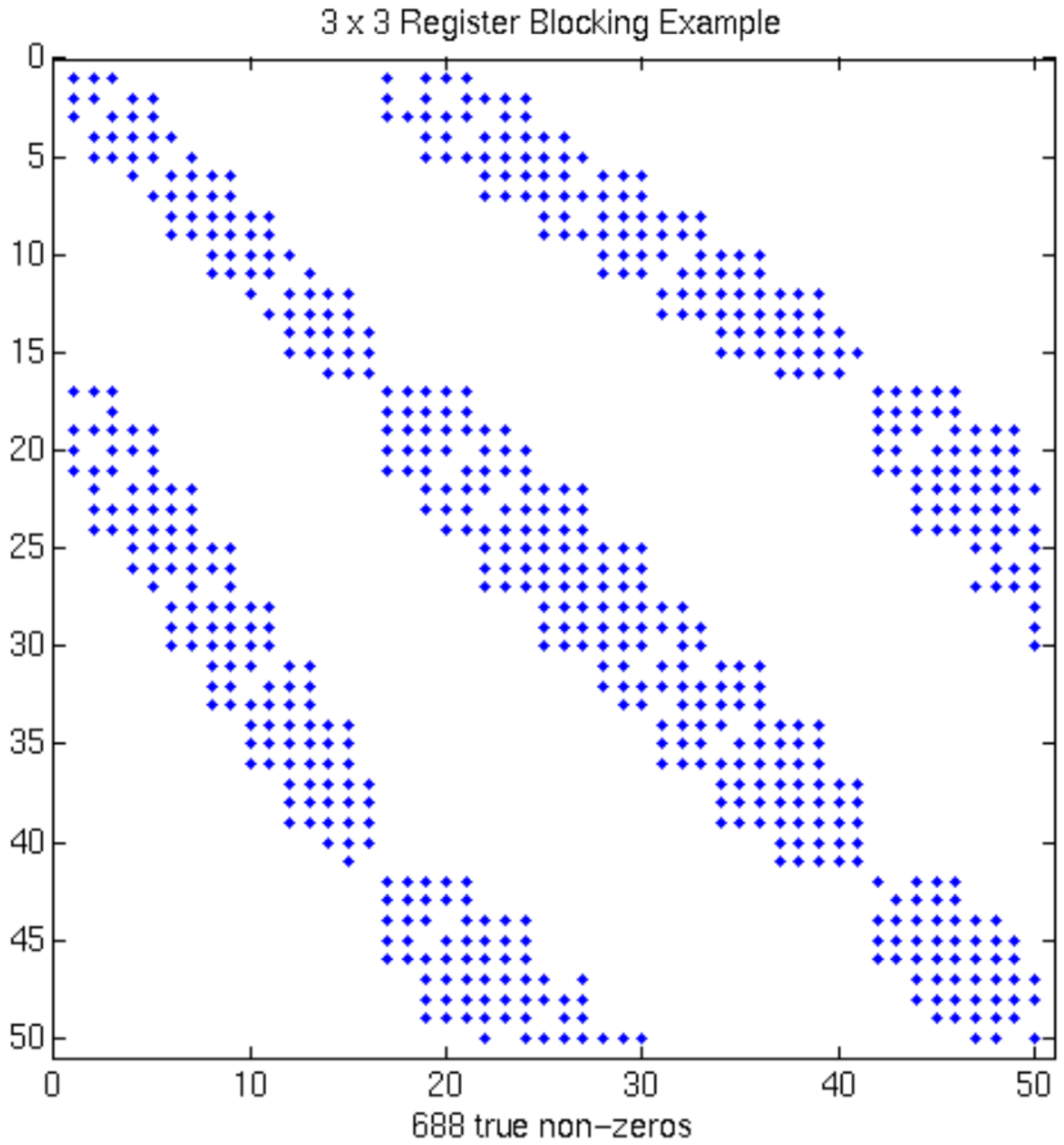
Zoom into top corner



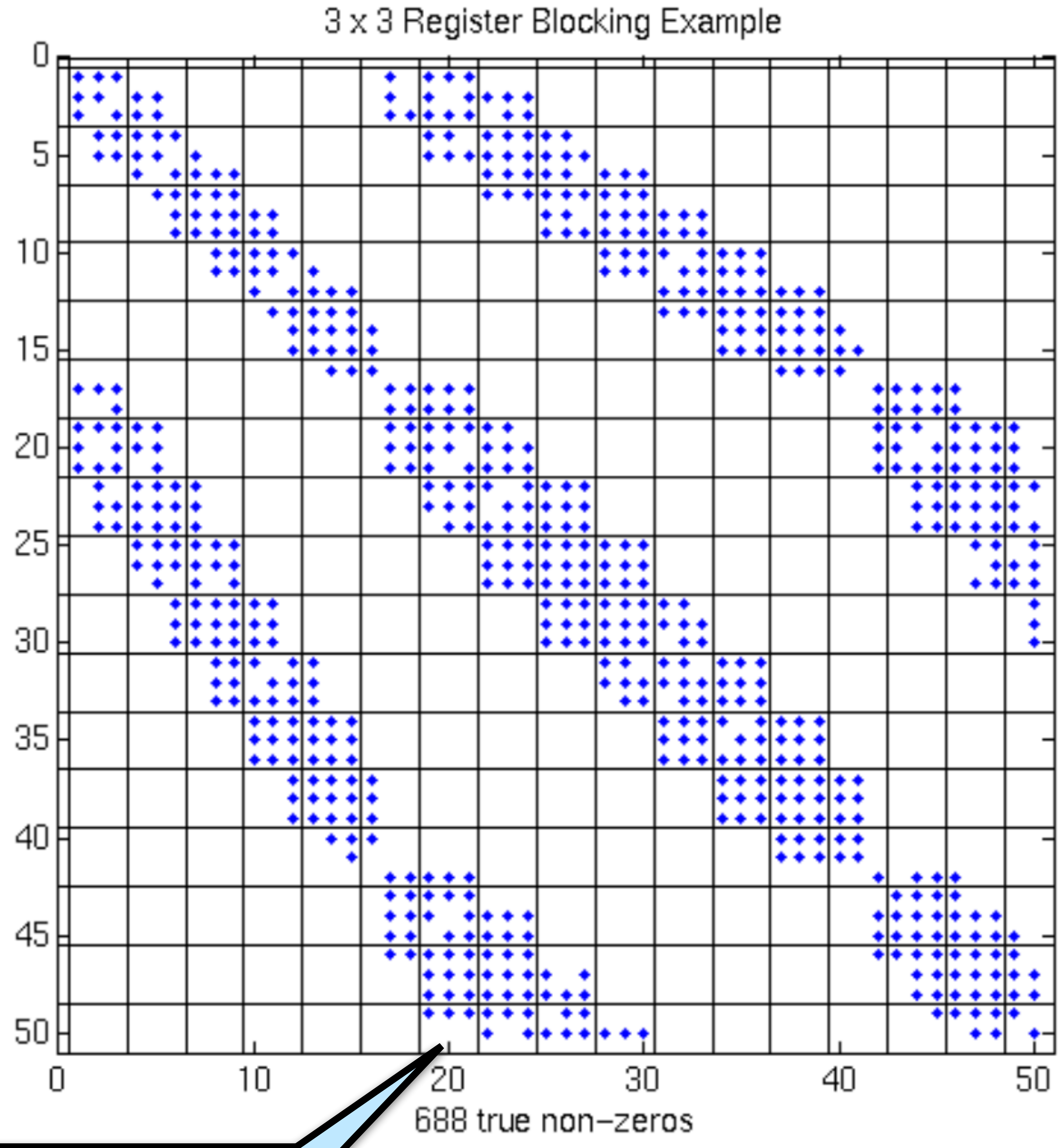
Fluid dynamics problem - more complicated nonzero structure
Total nnz = 1.1M

From UC Berkeley CS267

3x3 blocks look natural, but…



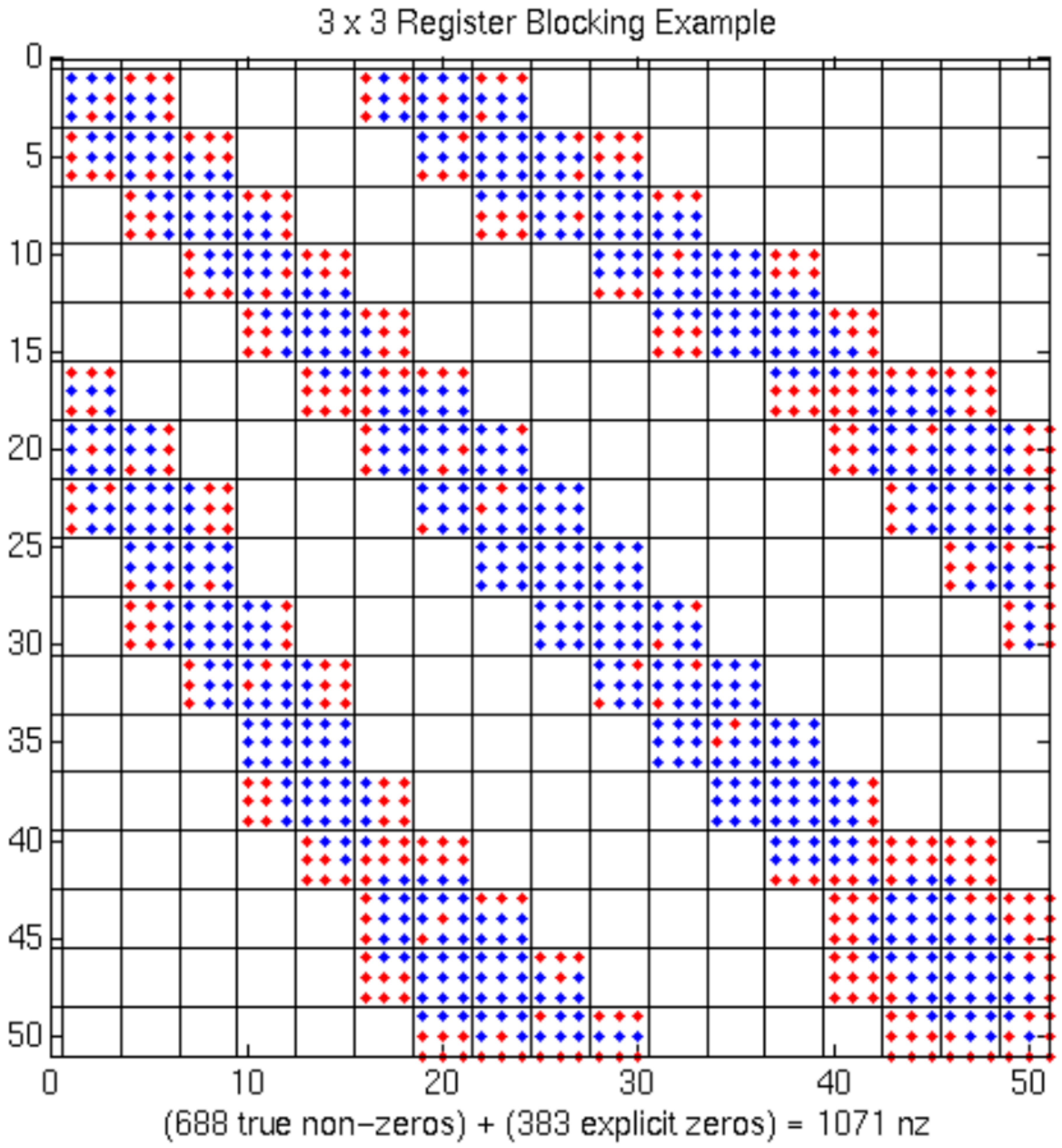
3x3 blocks



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

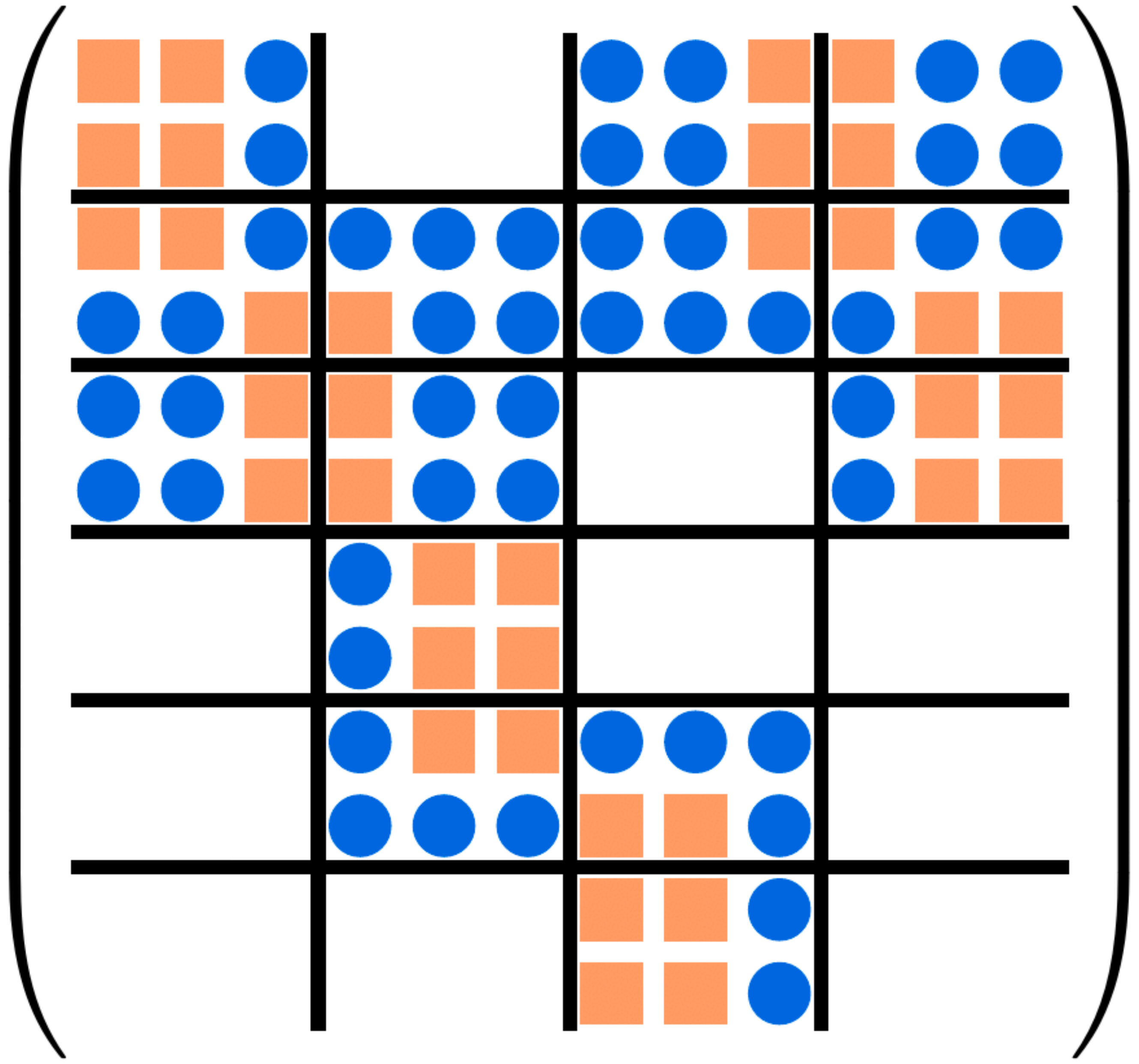
From UC Berkeley CS267

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 $k_{\mathbf{b}}(A)$

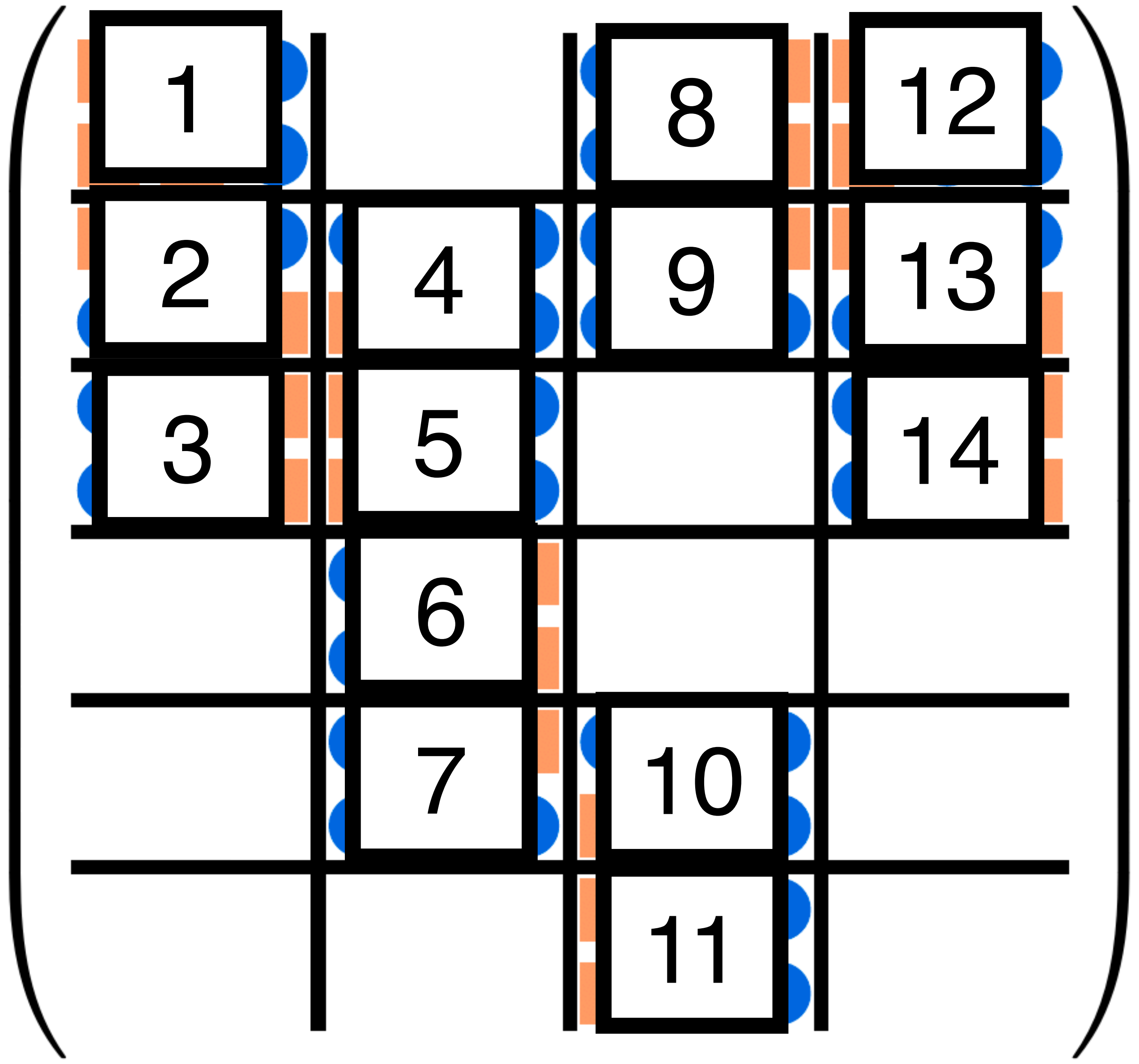
$$f_{\mathbf{b}}(A) = \frac{b_1 b_2 \ldots b_N k_{\mathbf{b}}(A)}{k(A)}$$

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

Nonempty blocks

Nonzeros

Greater ~ more wasted space



Blocking $\mathbf{b}$

$b_1 = 2$

$b_2 = 3$

$k_{\mathbf{b}} = 14$

$k(\mathcal{A}) = 36$

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.

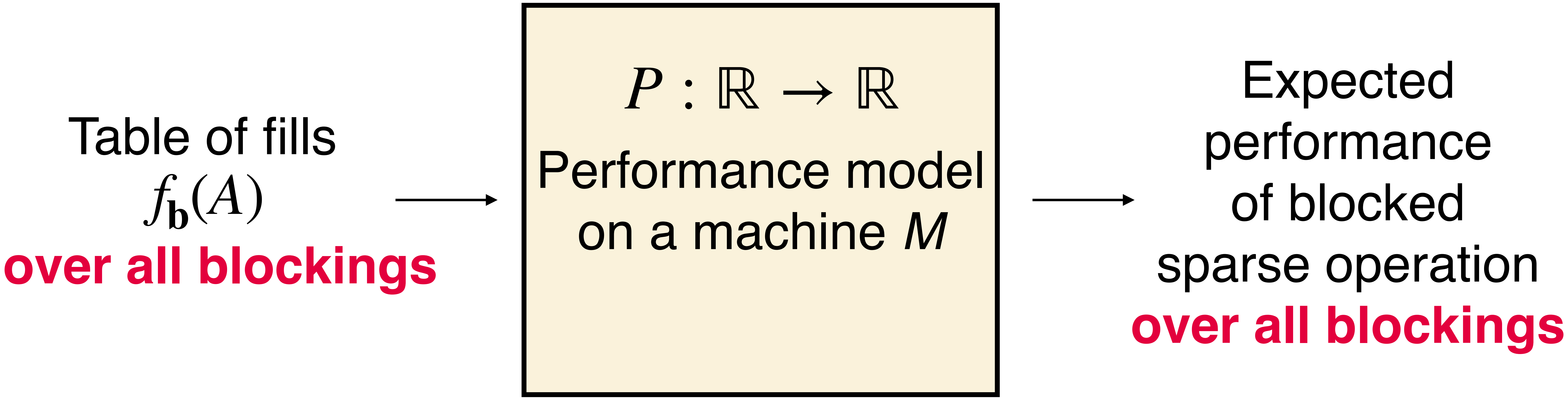


Table of fills
$f_{\mathbf{b}}(A)$
over all blockings

$P : \mathbb{R} \to \mathbb{R}$
Performance model
on a machine *M*

Expected
performance
of blocked
sparse operation
over all blockings

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

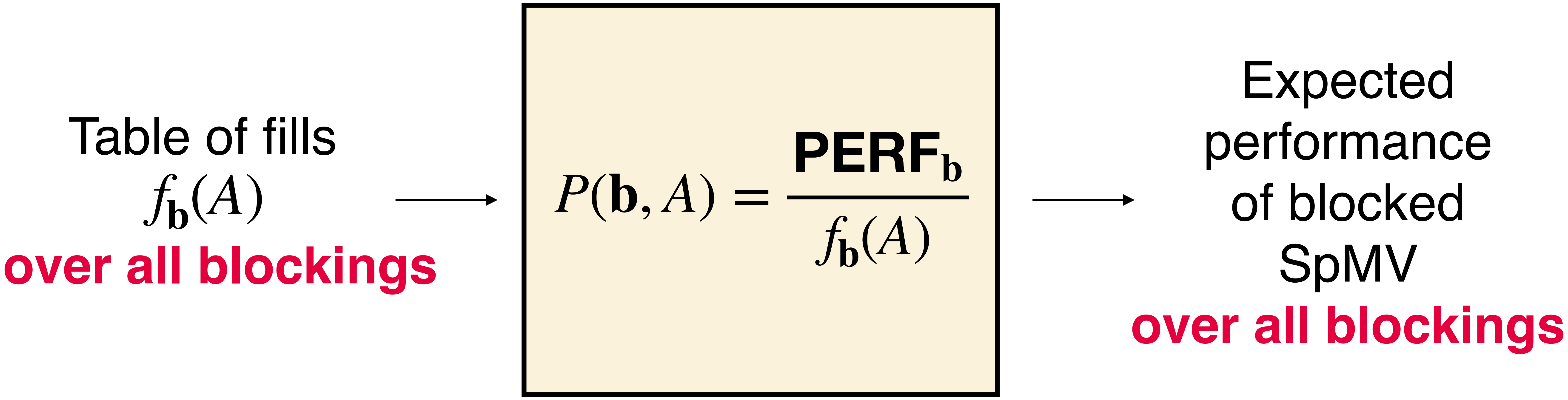


Table of fills

$f_{\mathbf{b}}(A)$

over all blockings

$$P(\mathbf{b}, A) = \frac{\textbf{PERF}_{\mathbf{b}}}{f_{\mathbf{b}}(A)}$$

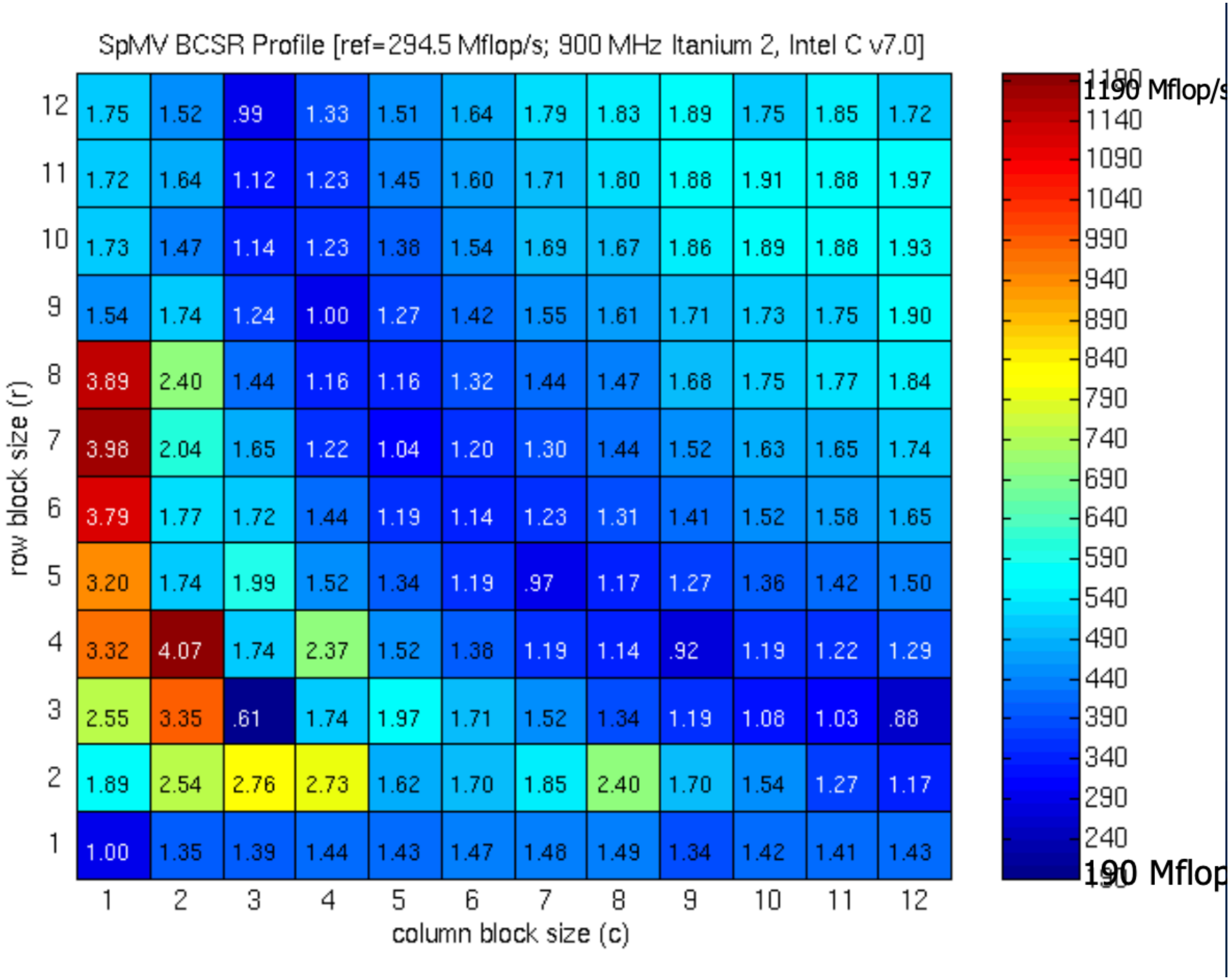
Expected performance of blocked SpMV

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

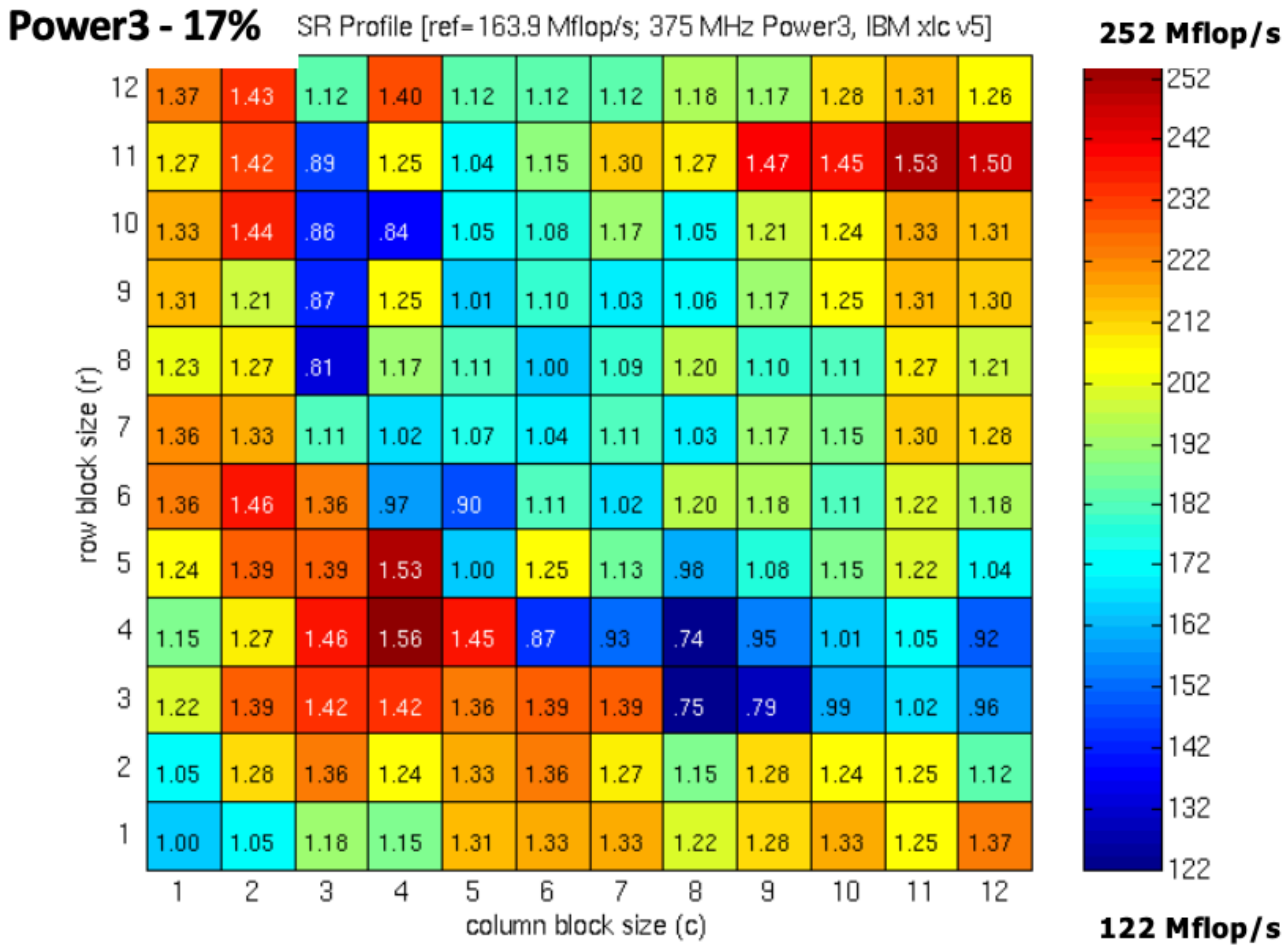


r \ c	1	2	3	4	5	6	7	8	9	10	11	12
12	1.75	1.52	.99	1.33	1.51	1.64	1.79	1.83	1.89	1.75	1.85	1.72
11	1.72	1.64	1.12	1.23	1.45	1.60	1.71	1.80	1.88	1.91	1.88	1.97
10	1.73	1.47	1.14	1.23	1.38	1.54	1.69	1.67	1.86	1.89	1.88	1.93
9	1.54	1.74	1.24	1.00	1.27	1.42	1.55	1.61	1.71	1.73	1.75	1.90
8	3.89	2.40	1.44	1.16	1.16	1.32	1.44	1.47	1.68	1.75	1.77	1.84
7	3.98	2.04	1.65	1.22	1.04	1.20	1.30	1.44	1.52	1.63	1.65	1.74
6	3.79	1.77	1.72	1.44	1.19	1.14	1.23	1.31	1.41	1.52	1.58	1.65
5	3.20	1.74	1.99	1.52	1.34	1.19	.97	1.17	1.27	1.36	1.42	1.50
4	3.32	4.07	1.74	2.37	1.52	1.38	1.19	1.14	.92	1.19	1.22	1.29
3	2.55	3.35	.61	1.74	1.97	1.71	1.52	1.34	1.19	1.08	1.03	.88
2	1.89	2.54	2.76	2.73	1.62	1.70	1.85	2.40	1.70	1.54	1.27	1.17
1	1.00	1.35	1.39	1.44	1.43	1.47	1.48	1.49	1.34	1.42	1.41	1.43

From UC Berkeley CS267

“Automatic Performance Tuning of Sparse Matrix Kernels,” Vuduc, 2003.

Register Profiles

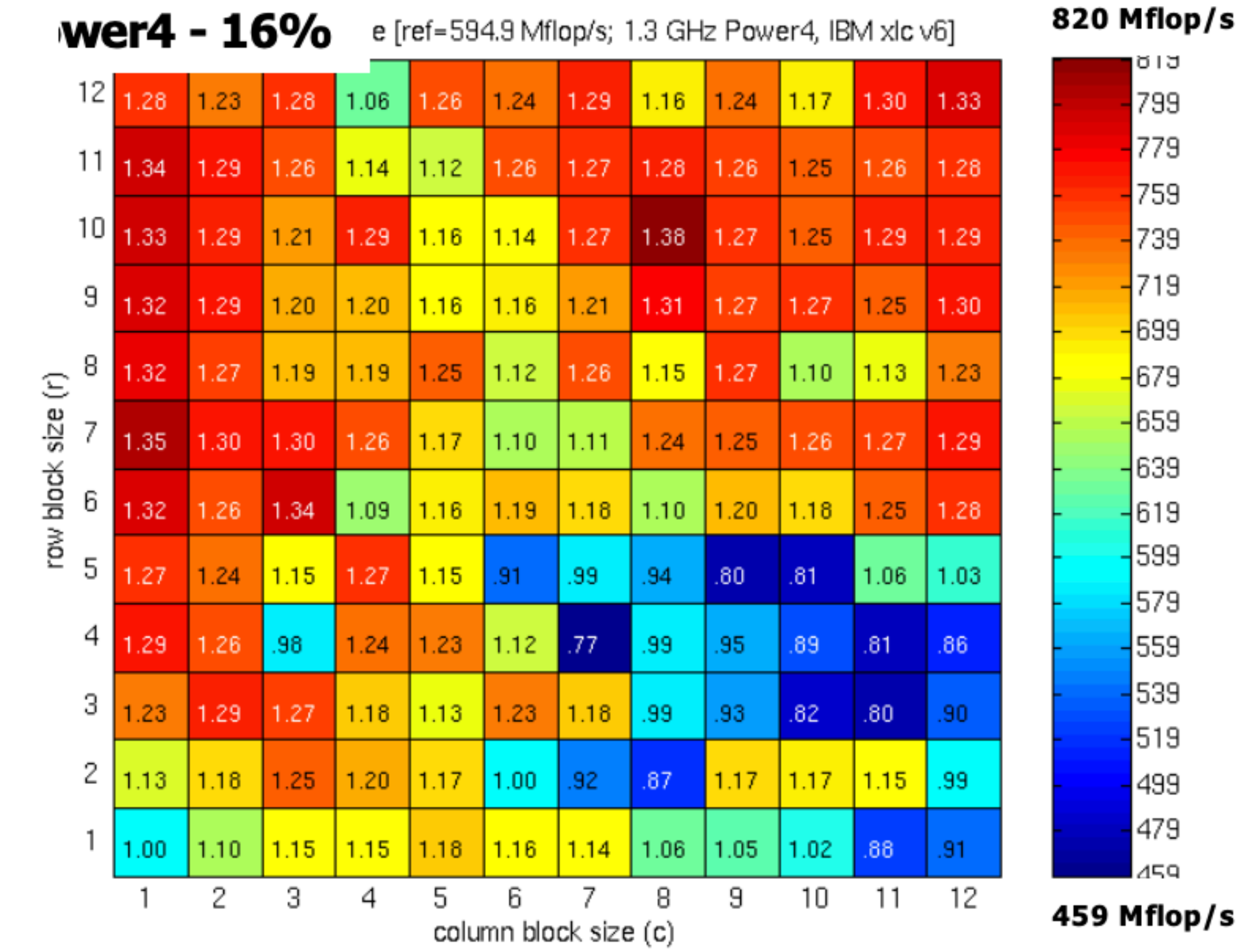


Power3 - 17%

252 Mflop/s

122 Mflop/s

	1	2	3	4	5	6	7	8	9	10	11	12
12	1.37	1.43	1.12	1.40	1.12	1.12	1.12	1.18	1.17	1.28	1.31	1.26
11	1.27	1.42	.89	1.25	1.04	1.15	1.30	1.27	1.47	1.45	1.53	1.50
10	1.33	1.44	.86	.84	1.05	1.08	1.17	1.05	1.21	1.24	1.33	1.31
9	1.31	1.21	.87	1.25	1.01	1.10	1.03	1.06	1.17	1.25	1.31	1.30
8	1.23	1.27	.81	1.17	1.11	1.00	1.09	1.20	1.10	1.11	1.27	1.21
7	1.36	1.33	1.11	1.02	1.07	1.04	1.11	1.03	1.17	1.15	1.30	1.28
6	1.36	1.46	1.36	.97	.90	1.11	1.02	1.20	1.18	1.11	1.22	1.18
5	1.24	1.39	1.39	1.53	1.00	1.25	1.13	.98	1.08	1.15	1.22	1.04
4	1.15	1.27	1.46	1.56	1.45	.87	.93	.74	.95	1.01	1.05	.92
3	1.22	1.39	1.42	1.42	1.36	1.39	1.39	.75	.79	.99	1.02	.96
2	1.05	1.28	1.36	1.24	1.33	1.36	1.27	1.15	1.28	1.24	1.25	1.12
1	1.00	1.05	1.18	1.15	1.31	1.33	1.33	1.22	1.28	1.33	1.25	1.37

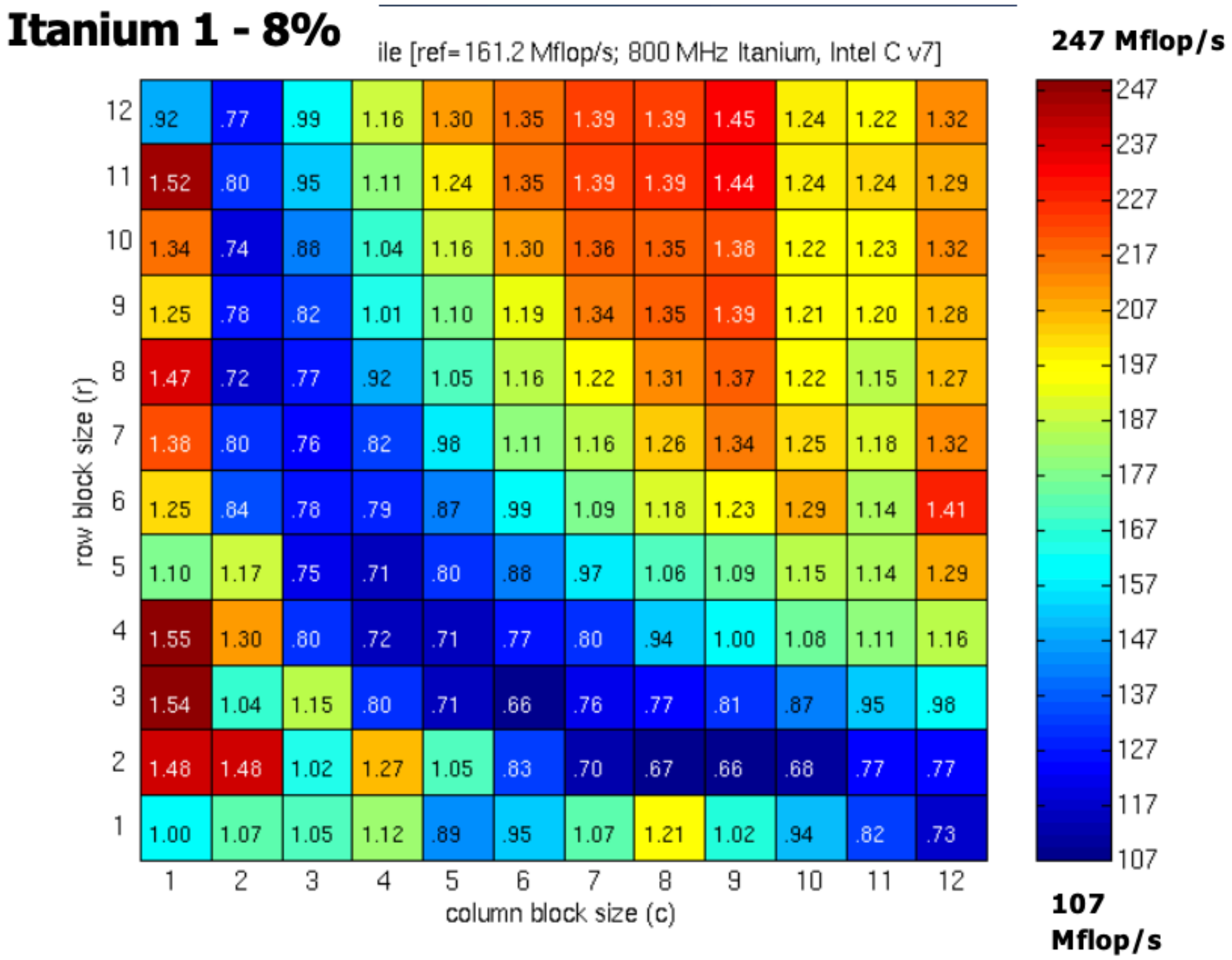


wer4 - 16%

820 Mflop/s

459 Mflop/s

	1	2	3	4	5	6	7	8	9	10	11	12
12	1.28	1.23	1.28	1.06	1.26	1.24	1.29	1.16	1.24	1.17	1.30	1.33
11	1.34	1.29	1.26	1.14	1.12	1.26	1.27	1.28	1.26	1.25	1.26	1.28
10	1.33	1.29	1.21	1.29	1.16	1.14	1.27	1.38	1.27	1.25	1.29	1.29
9	1.32	1.29	1.20	1.20	1.16	1.16	1.21	1.31	1.27	1.27	1.25	1.30
8	1.32	1.27	1.19	1.19	1.25	1.12	1.26	1.15	1.27	1.10	1.13	1.23
7	1.35	1.30	1.30	1.26	1.17	1.10	1.11	1.24	1.25	1.26	1.27	1.29
6	1.32	1.26	1.34	1.09	1.16	1.19	1.18	1.10	1.20	1.18	1.25	1.28
5	1.27	1.24	1.15	1.27	1.15	.91	.99	.94	.80	.81	1.06	1.03
4	1.29	1.26	.98	1.24	1.23	1.12	.77	.99	.95	.89	.81	.86
3	1.23	1.29	1.27	1.18	1.13	1.23	1.18	.99	.93	.82	.80	.90
2	1.13	1.18	1.25	1.20	1.17	1.00	.92	.87	1.17	1.17	1.15	.99
1	1.00	1.10	1.15	1.15	1.18	1.16	1.14	1.06	1.05	1.02	.88	.91

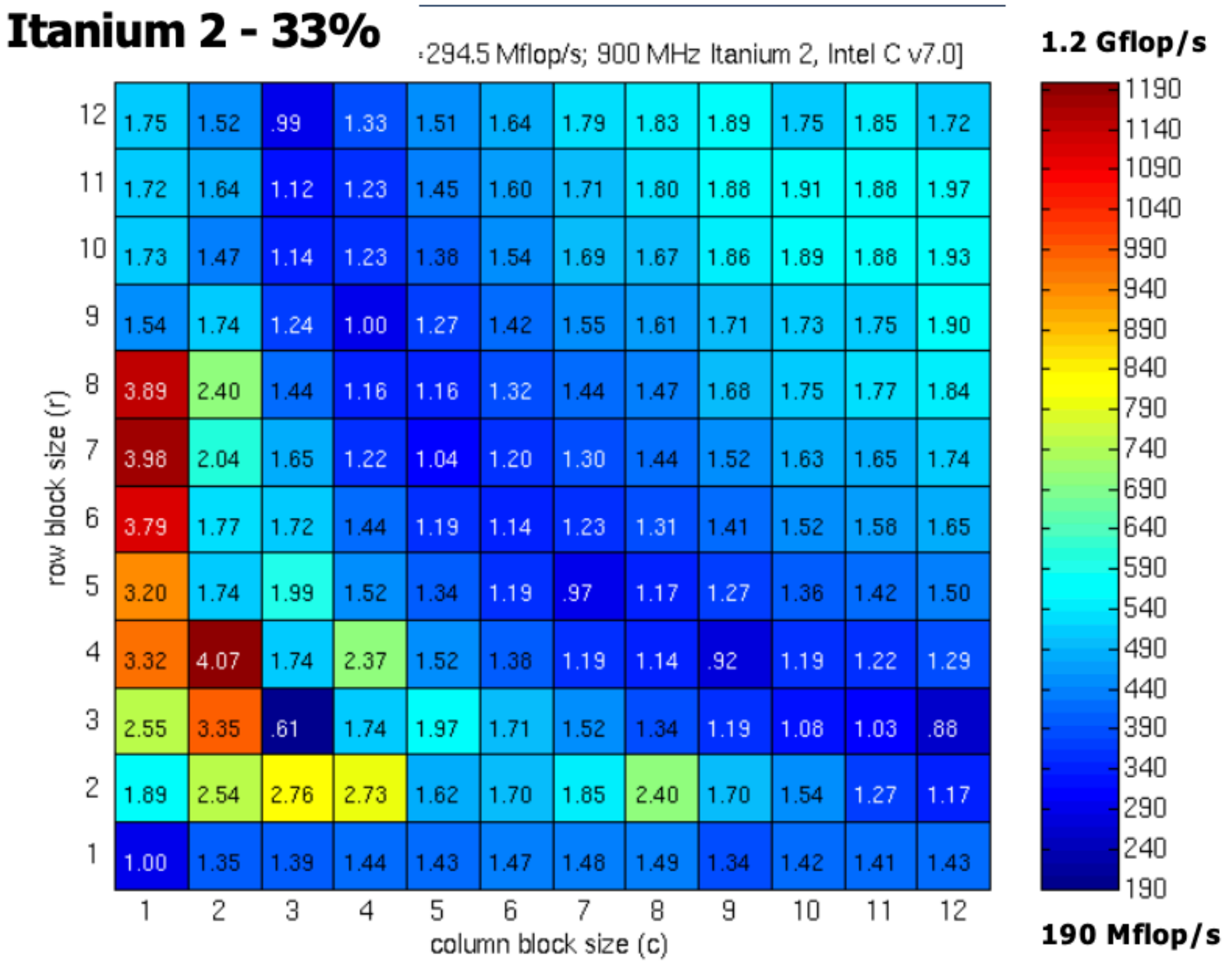


Itanium 1 - 8%

247 Mflop/s

107 Mflop/s

	1	2	3	4	5	6	7	8	9	10	11	12
12	.92	.77	.99	1.16	1.30	1.35	1.39	1.39	1.45	1.24	1.22	1.32
11	1.52	.80	.95	1.11	1.24	1.35	1.39	1.39	1.44	1.24	1.24	1.29
10	1.34	.74	.88	1.04	1.16	1.30	1.36	1.35	1.38	1.22	1.23	1.32
9	1.25	.78	.82	1.01	1.10	1.19	1.34	1.35	1.39	1.21	1.20	1.28
8	1.47	.72	.77	.92	1.05	1.16	1.22	1.31	1.37	1.22	1.15	1.27
7	1.38	.80	.76	.82	.98	1.11	1.16	1.26	1.34	1.25	1.18	1.32
6	1.25	.84	.78	.79	.87	.99	1.09	1.18	1.23	1.29	1.14	1.41
5	1.10	1.17	.75	.71	.80	.88	.97	1.06	1.09	1.15	1.14	1.29
4	1.55	1.30	.80	.72	.71	.77	.80	.94	1.00	1.08	1.11	1.16
3	1.54	1.04	1.15	.80	.71	.66	.76	.77	.81	.87	.95	.98
2	1.48	1.48	1.02	1.27	1.05	.83	.70	.67	.66	.68	.77	.77
1	1.00	1.07	1.05	1.12	.89	.95	1.07	1.21	1.02	.94	.82	.73



Itanium 2 - 33%

1.2 Gflop/s

190 Mflop/s

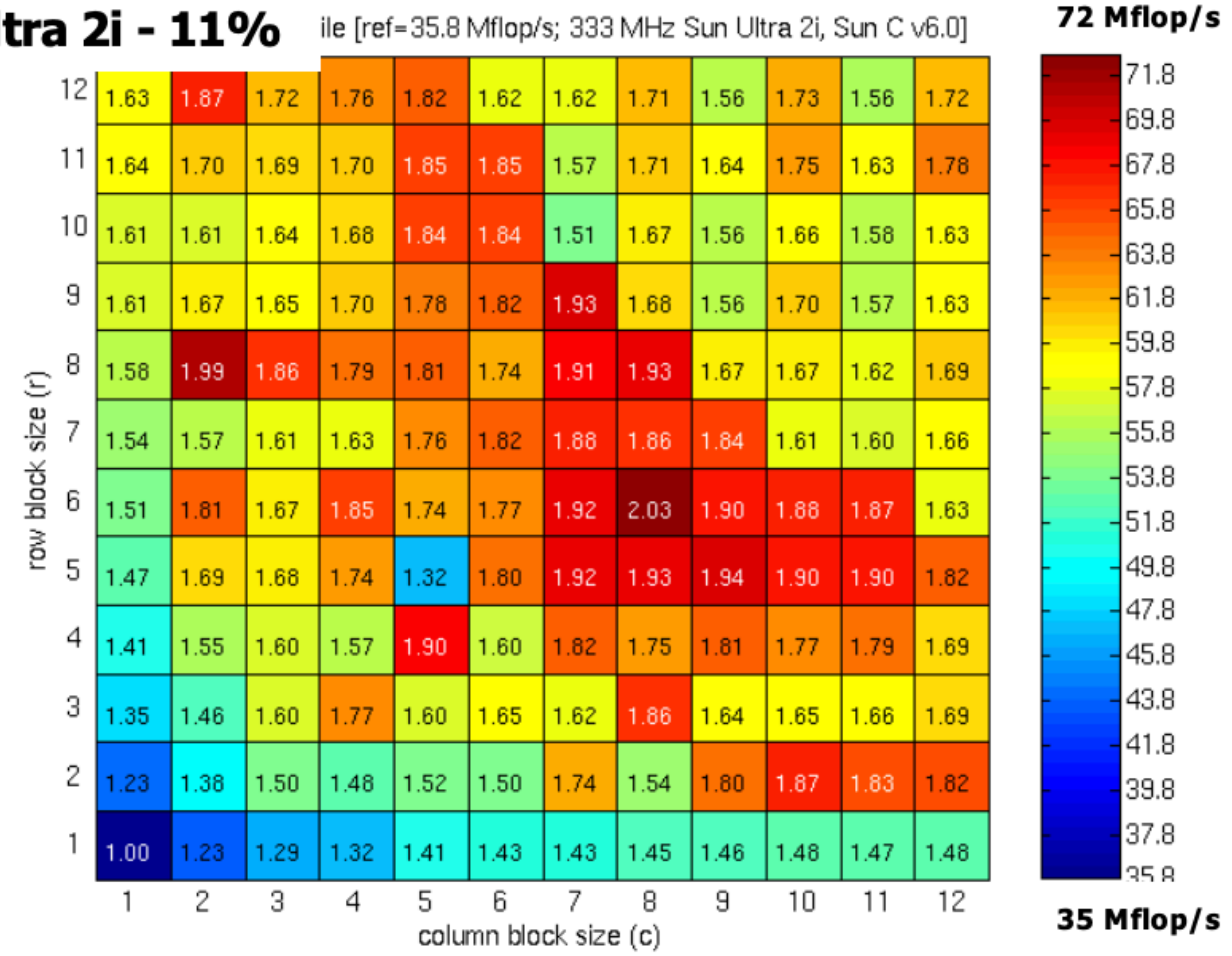
	1	2	3	4	5	6	7	8	9	10	11	12
12	1.75	1.52	.99	1.33	1.51	1.64	1.79	1.83	1.89	1.75	1.85	1.72
11	1.72	1.64	1.12	1.23	1.45	1.60	1.71	1.80	1.88	1.91	1.88	1.97
10	1.73	1.47	1.14	1.23	1.38	1.54	1.69	1.67	1.86	1.89	1.88	1.93
9	1.54	1.74	1.24	1.00	1.27	1.42	1.55	1.61	1.71	1.73	1.75	1.90
8	3.89	2.40	1.44	1.16	1.16	1.32	1.44	1.47	1.68	1.75	1.77	1.84
7	3.98	2.04	1.65	1.22	1.04	1.20	1.30	1.44	1.52	1.63	1.65	1.74
6	3.79	1.77	1.72	1.44	1.19	1.14	1.23	1.31	1.41	1.52	1.58	1.65
5	3.20	1.74	1.99	1.52	1.34	1.19	.97	1.17	1.27	1.36	1.42	1.50
4	3.32	4.07	1.74	2.37	1.52	1.38	1.19	1.14	.92	1.19	1.22	1.29
3	2.55	3.35	.61	1.74	1.97	1.71	1.52	1.34	1.19	1.08	1.03	.88
2	1.89	2.54	2.76	2.73	1.62	1.70	1.85	2.40	1.70	1.54	1.27	1.17
1	1.00	1.35	1.39	1.44	1.43	1.47	1.48	1.49	1.34	1.42	1.41	1.43

From UC Berkeley CS267

“Automatic Performance Tuning of Sparse Matrix Kernels,” Vuduc, 2003.

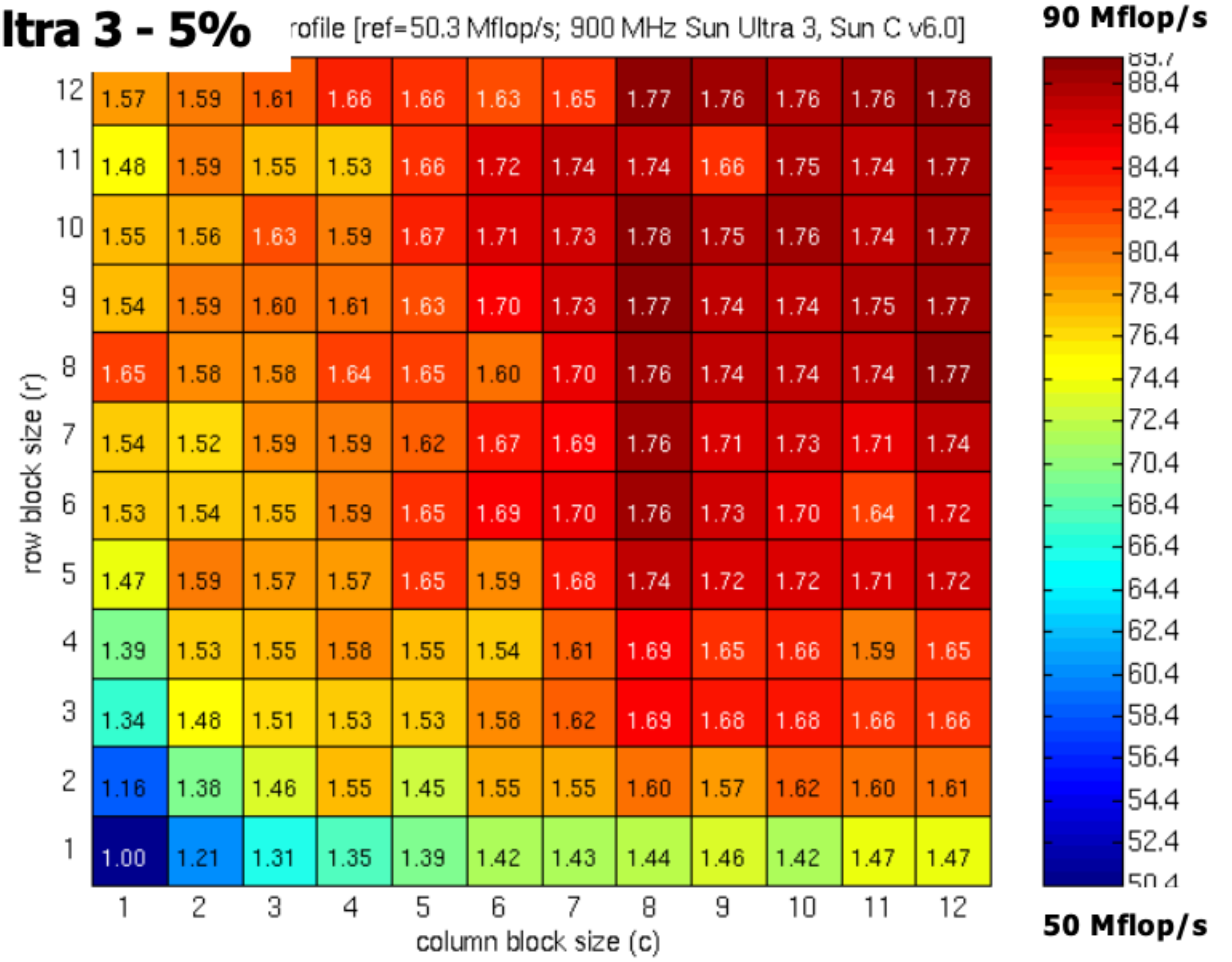
Register Profiles

Ultra 2i - 11%



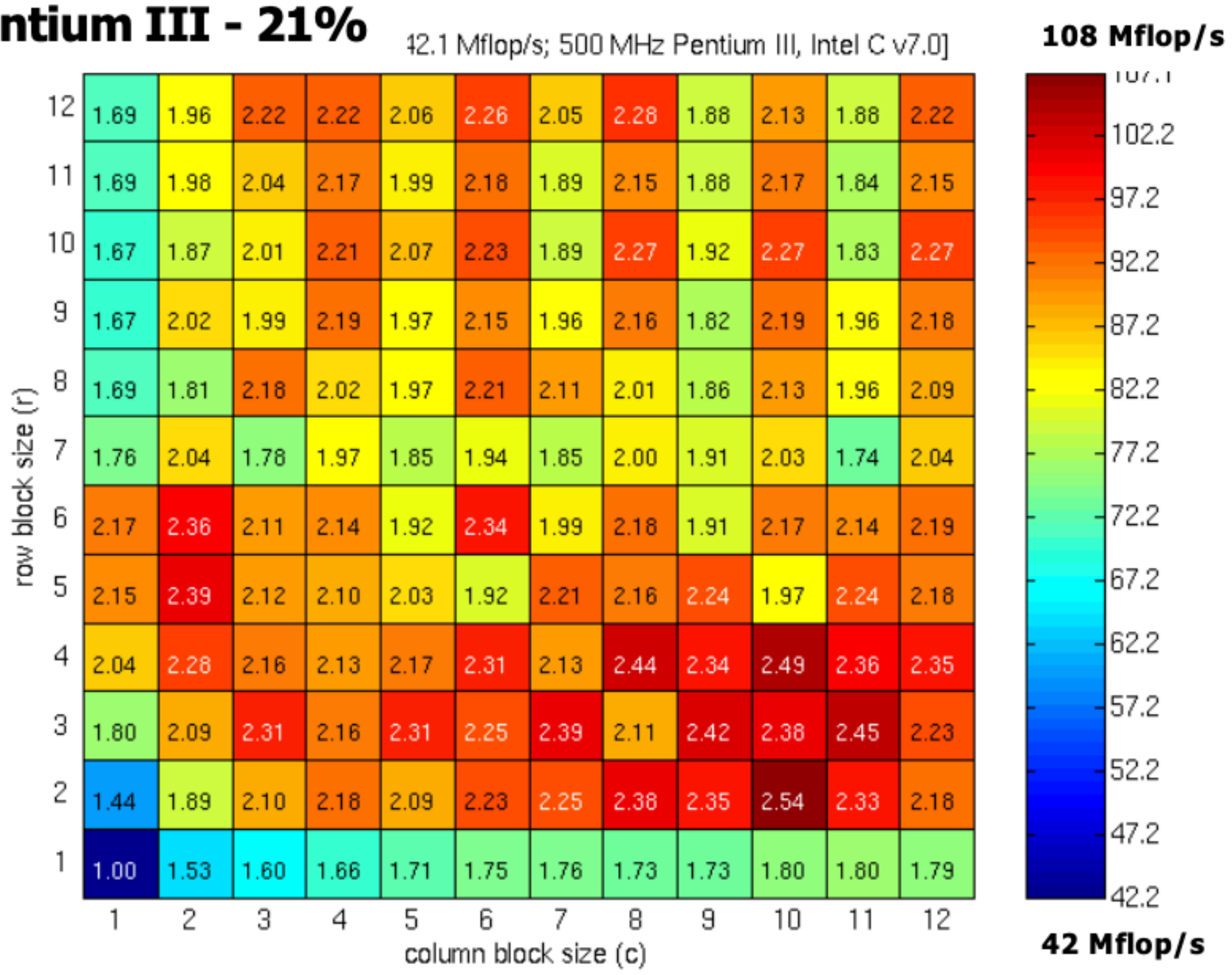
	1	2	3	4	5	6	7	8	9	10	11	12
12	1.63	1.87	1.72	1.76	1.82	1.62	1.62	1.71	1.56	1.73	1.56	1.72
11	1.64	1.70	1.69	1.70	1.85	1.85	1.57	1.71	1.64	1.75	1.63	1.78
10	1.61	1.61	1.64	1.68	1.84	1.84	1.51	1.67	1.56	1.66	1.58	1.63
9	1.61	1.67	1.65	1.70	1.78	1.82	1.93	1.68	1.56	1.70	1.57	1.63
8	1.58	1.99	1.86	1.79	1.81	1.74	1.91	1.93	1.67	1.67	1.62	1.69
7	1.54	1.57	1.61	1.63	1.76	1.82	1.88	1.86	1.84	1.61	1.60	1.66
6	1.51	1.81	1.67	1.85	1.74	1.77	1.92	2.03	1.90	1.88	1.87	1.63
5	1.47	1.69	1.68	1.74	1.32	1.80	1.92	1.93	1.94	1.90	1.90	1.82
4	1.41	1.55	1.60	1.57	1.90	1.60	1.82	1.75	1.81	1.77	1.79	1.69
3	1.35	1.46	1.60	1.77	1.60	1.65	1.62	1.86	1.64	1.65	1.66	1.69
2	1.23	1.38	1.50	1.48	1.52	1.50	1.74	1.54	1.80	1.87	1.83	1.82
1	1.00	1.23	1.29	1.32	1.41	1.43	1.43	1.45	1.46	1.48	1.47	1.48

Ultra 3 - 5%



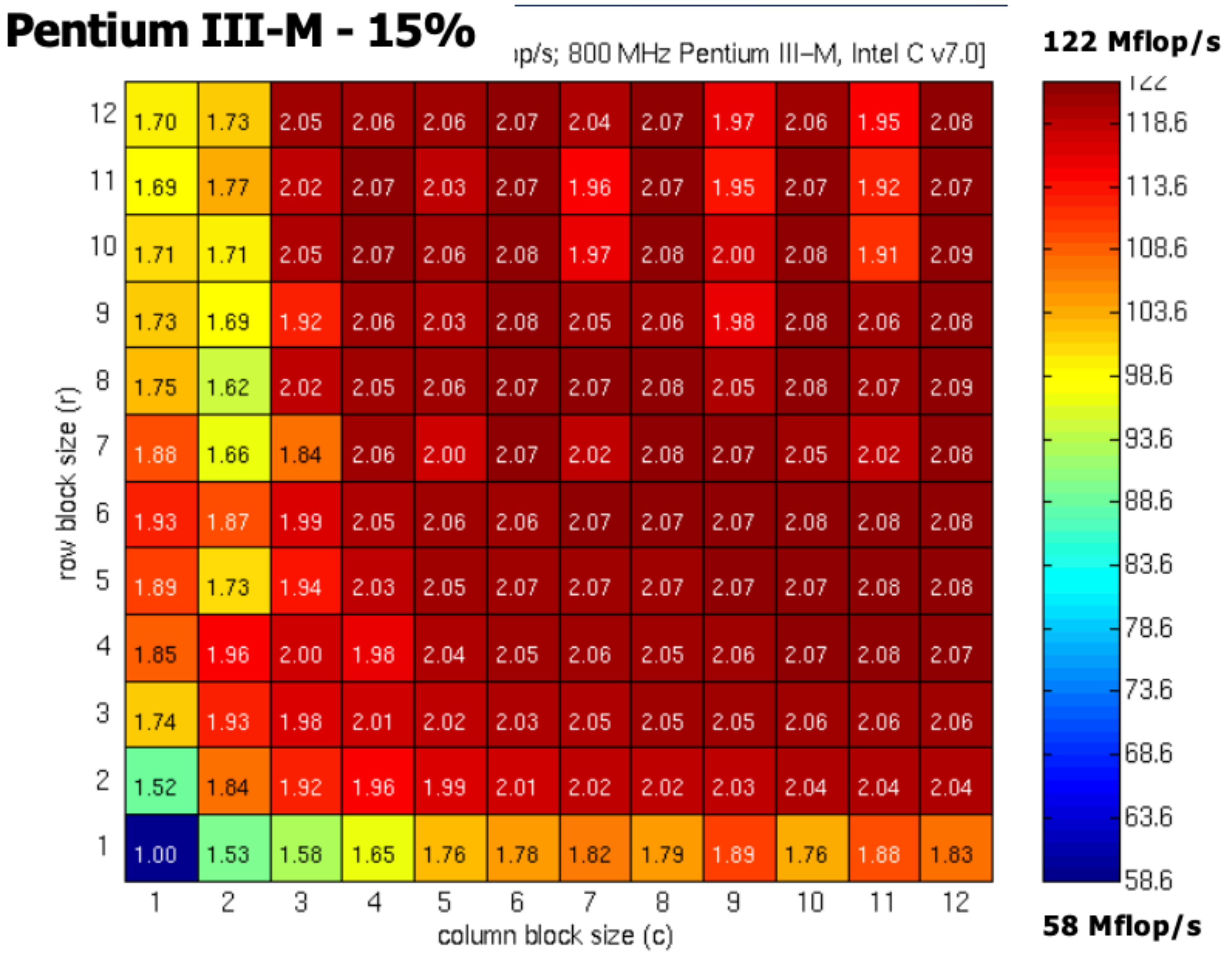
	1	2	3	4	5	6	7	8	9	10	11	12
12	1.57	1.59	1.61	1.66	1.66	1.63	1.65	1.77	1.76	1.76	1.76	1.78
11	1.48	1.59	1.55	1.53	1.66	1.72	1.74	1.74	1.66	1.75	1.74	1.77
10	1.55	1.56	1.63	1.59	1.67	1.71	1.73	1.78	1.75	1.76	1.74	1.77
9	1.54	1.59	1.60	1.61	1.63	1.70	1.73	1.77	1.74	1.74	1.75	1.77
8	1.65	1.58	1.58	1.64	1.65	1.60	1.70	1.76	1.74	1.74	1.74	1.77
7	1.54	1.52	1.59	1.59	1.62	1.67	1.69	1.76	1.71	1.73	1.71	1.74
6	1.53	1.54	1.55	1.59	1.65	1.69	1.70	1.76	1.73	1.70	1.64	1.72
5	1.47	1.59	1.57	1.57	1.65	1.59	1.68	1.74	1.72	1.72	1.71	1.72
4	1.39	1.53	1.55	1.58	1.55	1.54	1.61	1.69	1.65	1.66	1.59	1.65
3	1.34	1.48	1.51	1.53	1.53	1.58	1.62	1.69	1.68	1.68	1.66	1.66
2	1.16	1.38	1.46	1.55	1.45	1.55	1.55	1.60	1.57	1.62	1.60	1.61
1	1.00	1.21	1.31	1.35	1.39	1.42	1.43	1.44	1.46	1.42	1.47	1.47

Pentium III - 21%



	1	2	3	4	5	6	7	8	9	10	11	12
12	1.69	1.96	2.22	2.22	2.06	2.26	2.05	2.28	1.88	2.13	1.88	2.22
11	1.69	1.98	2.04	2.17	1.99	2.18	1.89	2.15	1.88	2.17	1.84	2.15
10	1.67	1.87	2.01	2.21	2.07	2.23	1.89	2.27	1.92	2.27	1.83	2.27
9	1.67	2.02	1.99	2.19	1.97	2.15	1.96	2.16	1.82	2.19	1.96	2.18
8	1.69	1.81	2.18	2.02	1.97	2.21	2.11	2.01	1.86	2.13	1.96	2.09
7	1.76	2.04	1.78	1.97	1.85	1.94	1.85	2.00	1.91	2.03	1.74	2.04
6	2.17	2.36	2.11	2.14	1.92	2.34	1.99	2.18	1.91	2.17	2.14	2.19
5	2.15	2.39	2.12	2.10	2.03	1.92	2.21	2.16	2.24	1.97	2.24	2.18
4	2.04	2.28	2.16	2.13	2.17	2.31	2.13	2.44	2.34	2.49	2.36	2.35
3	1.80	2.09	2.31	2.16	2.31	2.25	2.39	2.11	2.42	2.38	2.45	2.23
2	1.44	1.89	2.10	2.18	2.09	2.23	2.25	2.38	2.35	2.54	2.33	2.18
1	1.00	1.53	1.60	1.66	1.71	1.75	1.76	1.73	1.73	1.80	1.80	1.79

Pentium III-M - 15%

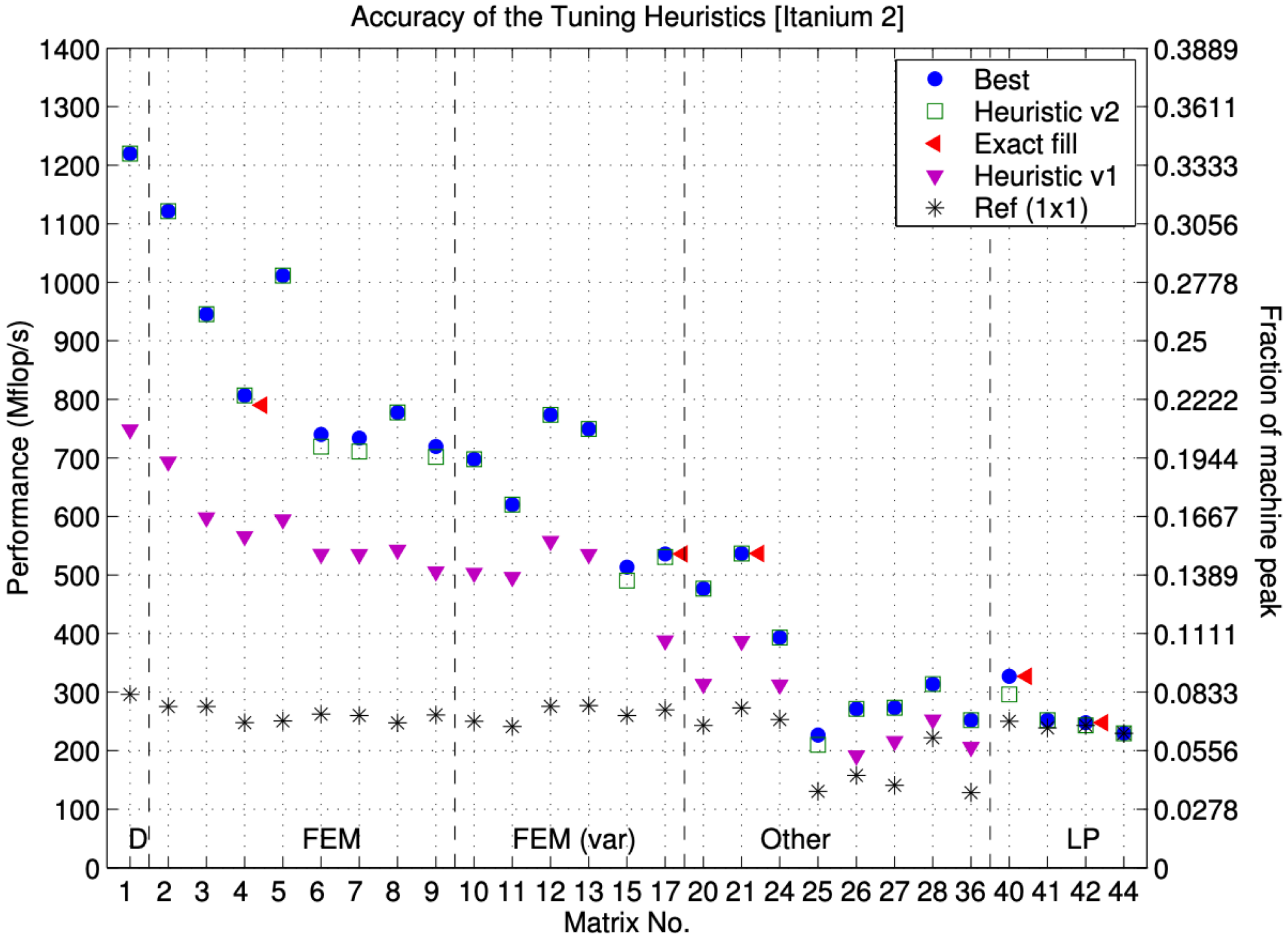


	1	2	3	4	5	6	7	8	9	10	11	12
12	1.70	1.73	2.05	2.06	2.06	2.07	2.04	2.07	1.97	2.06	1.95	2.08
11	1.69	1.77	2.02	2.07	2.03	2.07	1.96	2.07	1.95	2.07	1.92	2.07
10	1.71	1.71	2.05	2.07	2.06	2.08	1.97	2.08	2.00	2.08	1.91	2.09
9	1.73	1.69	1.92	2.06	2.03	2.08	2.05	2.06	1.98	2.08	2.06	2.08
8	1.75	1.62	2.02	2.05	2.06	2.07	2.07	2.08	2.05	2.08	2.07	2.09
7	1.88	1.66	1.84	2.06	2.00	2.07	2.02	2.08	2.07	2.05	2.02	2.08
6	1.93	1.87	1.99	2.05	2.06	2.06	2.07	2.07	2.07	2.08	2.08	2.08
5	1.89	1.73	1.94	2.03	2.05	2.07	2.07	2.07	2.07	2.07	2.08	2.08
4	1.85	1.96	2.00	1.98	2.04	2.05	2.06	2.05	2.06	2.07	2.08	2.07
3	1.74	1.93	1.98	2.01	2.02	2.03	2.05	2.05	2.05	2.06	2.06	2.06
2	1.52	1.84	1.92	1.96	1.99	2.01	2.02	2.02	2.03	2.04	2.04	2.04
1	1.00	1.53	1.58	1.65	1.76	1.78	1.82	1.79	1.89	1.76	1.88	1.83

From UC Berkeley CS267

“Automatic Performance Tuning of Sparse Matrix Kernels,” Vuduc, 2003.

Heuristic Tuning vs Best

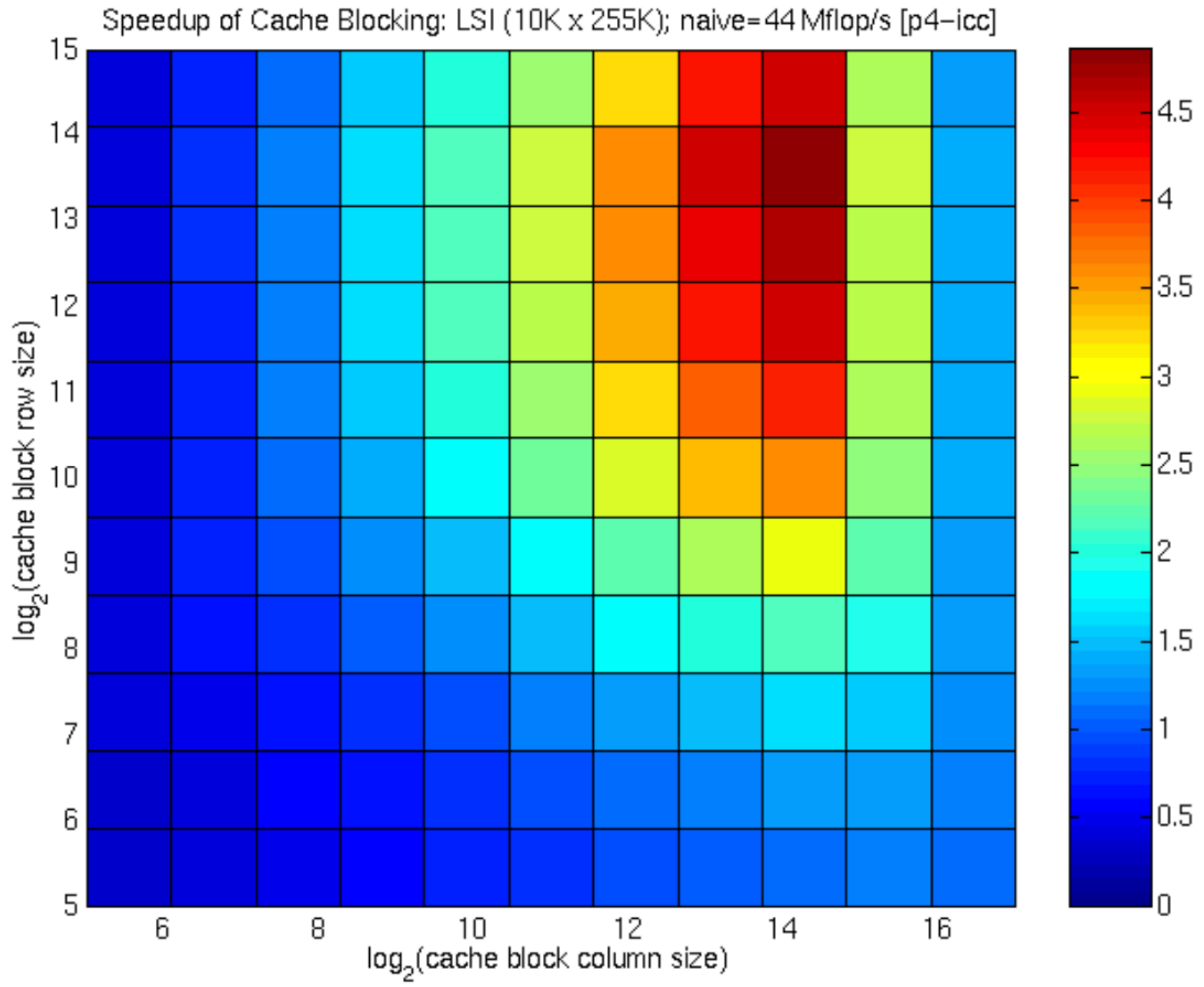


From UC Berkeley CS267

“Automatic Performance Tuning of Sparse Matrix Kernels,” Vuduc, 2003.

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

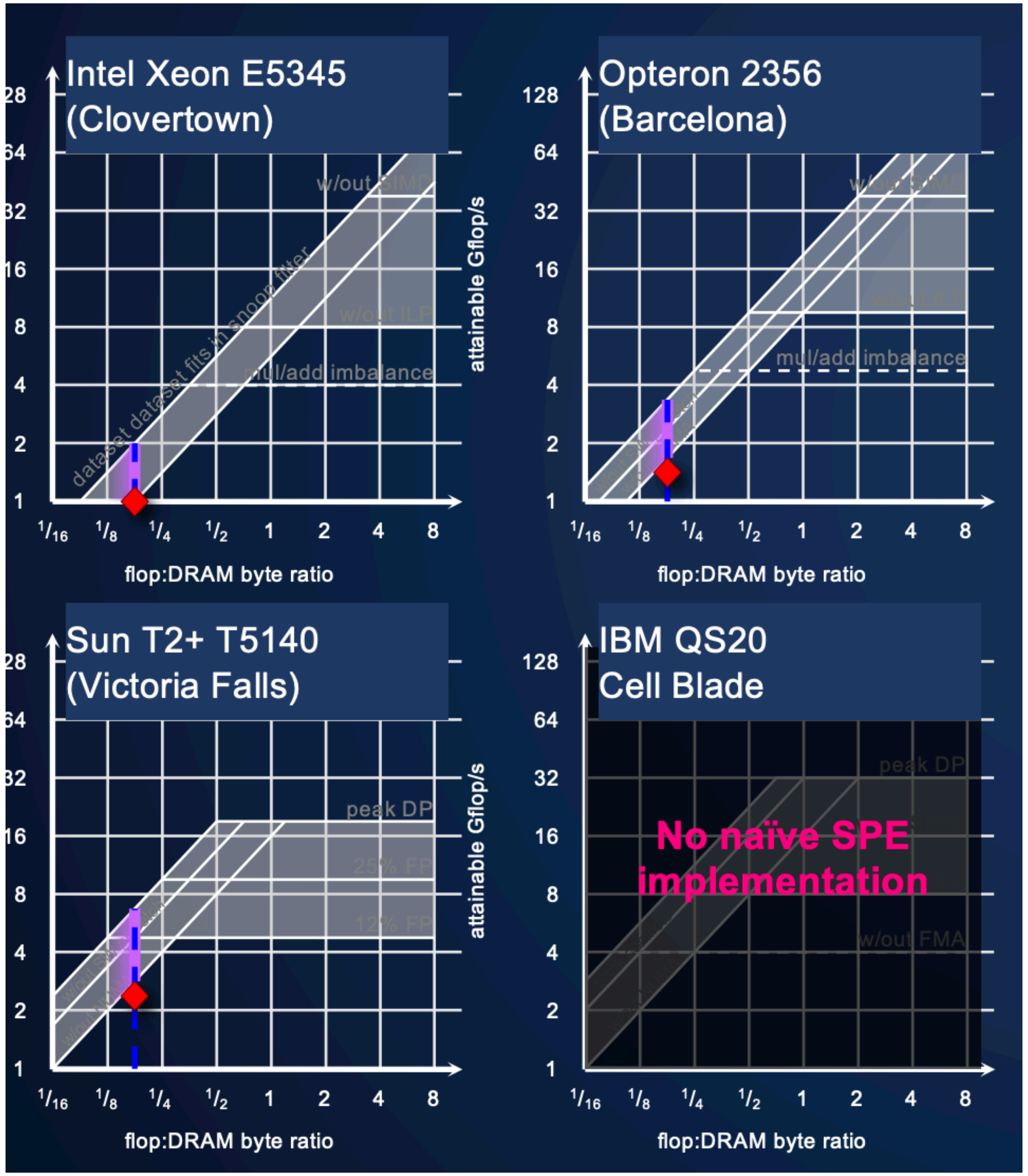
Baseline: 44 Mflop/s

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.

Roofline model for SpMV (out-of-the-box parallel)



Two unit stride streams

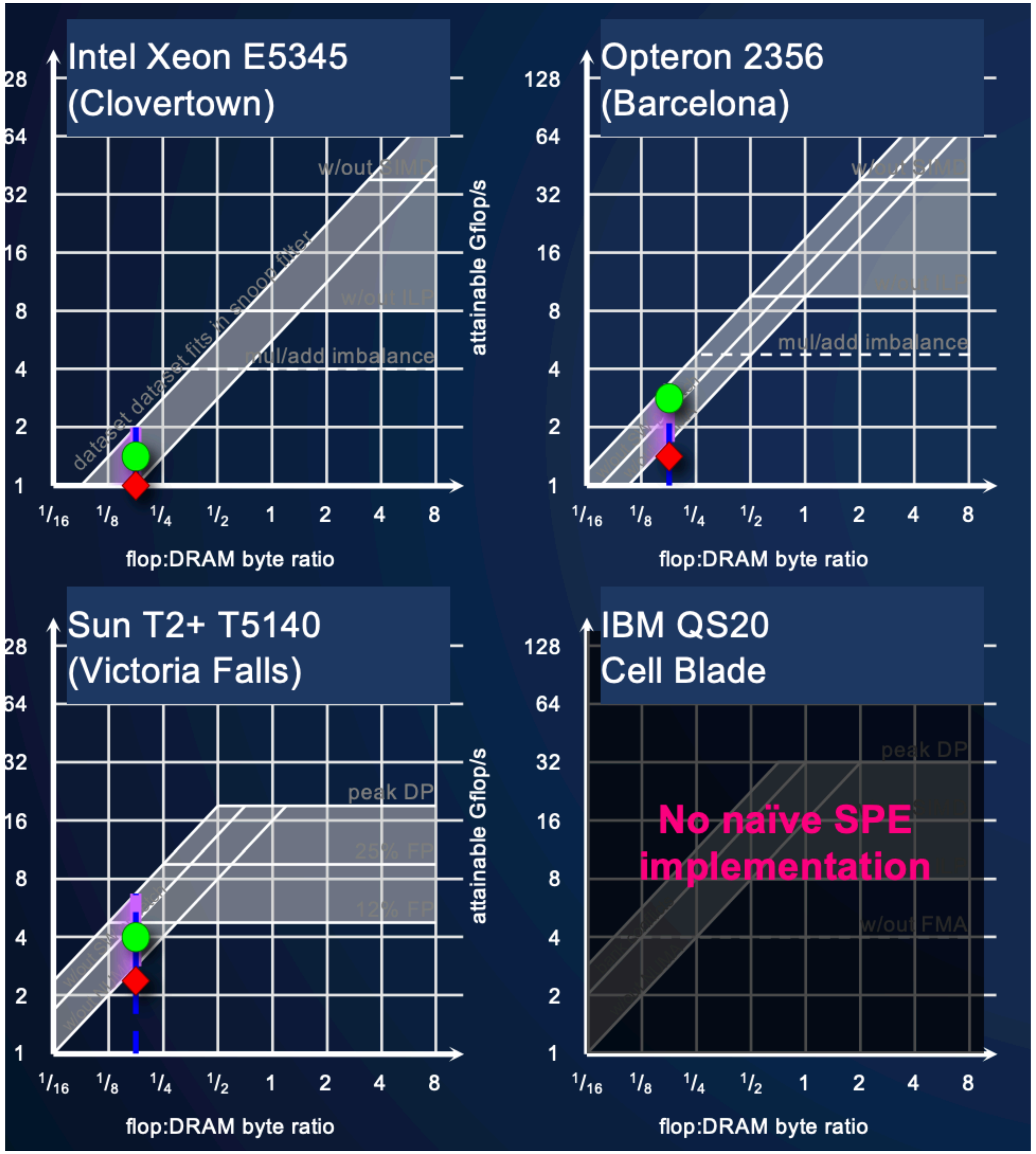
Inherent FMA

No instruction-level or data-level parallelism

Naïve compulsory: flop:byte < 0.166

From UC Berkeley CS267

Roofline model for SpMV
(NUMA & SW prefetch)



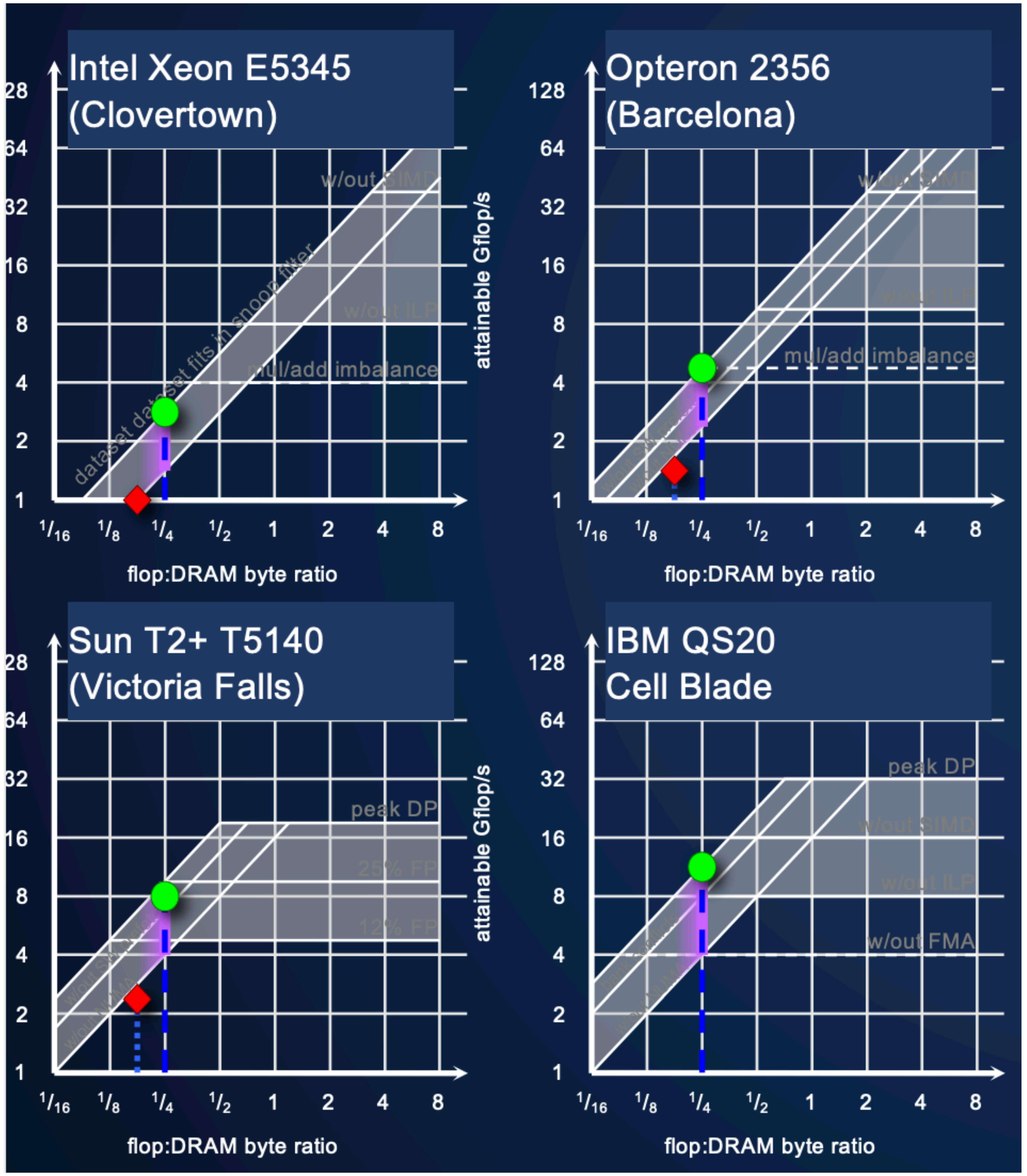
Compulsory flop:byte ~ 0.166

Utilize all memory channels

Add software prefetching

From UC Berkeley CS267

Roofline model for SpMV (matrix compression)

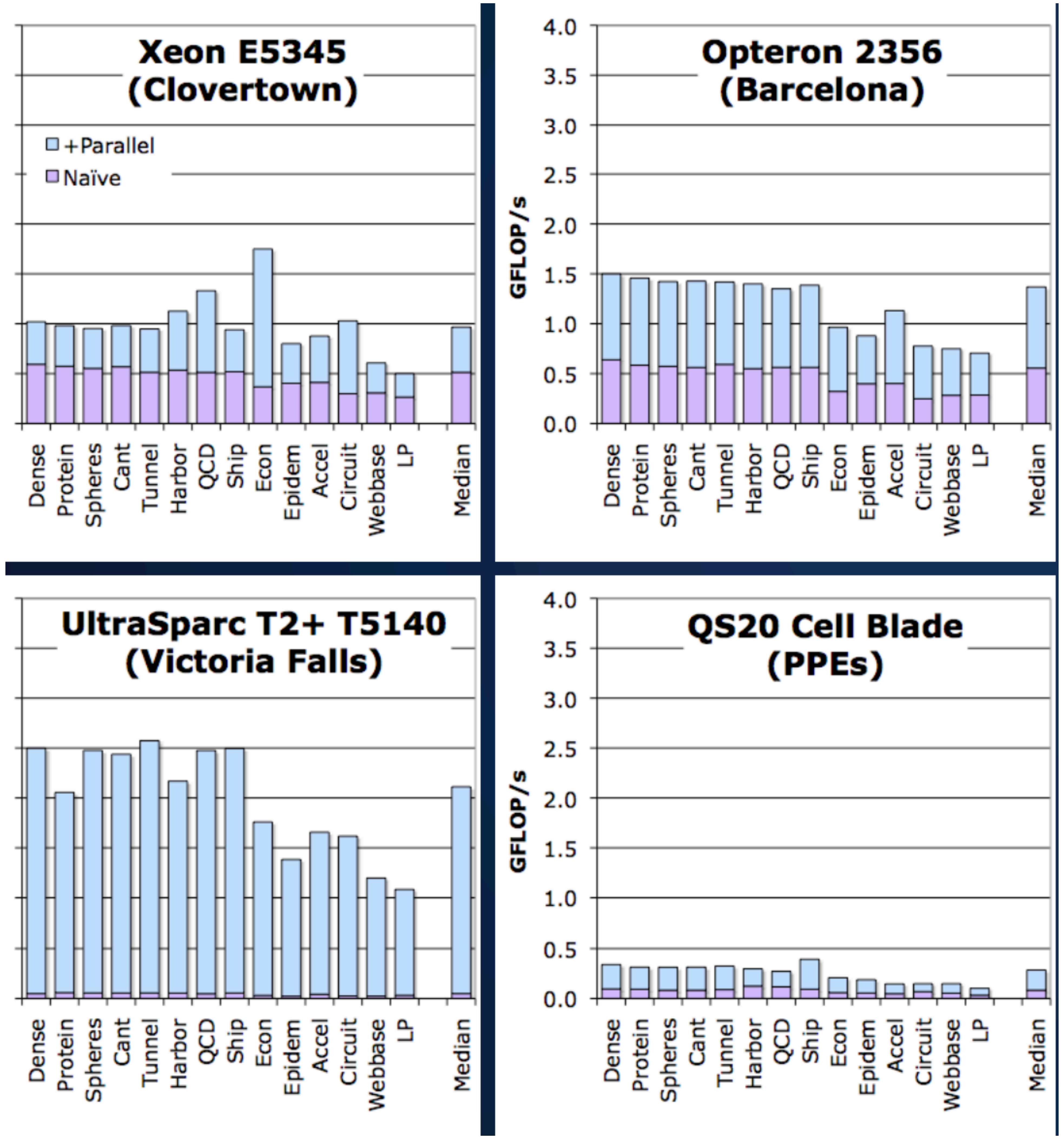


Inherent FMA

Register blocking improves ILP, DLP, flop:byte ratio

From UC Berkeley CS267

SpMV Performance (simple parallelization)



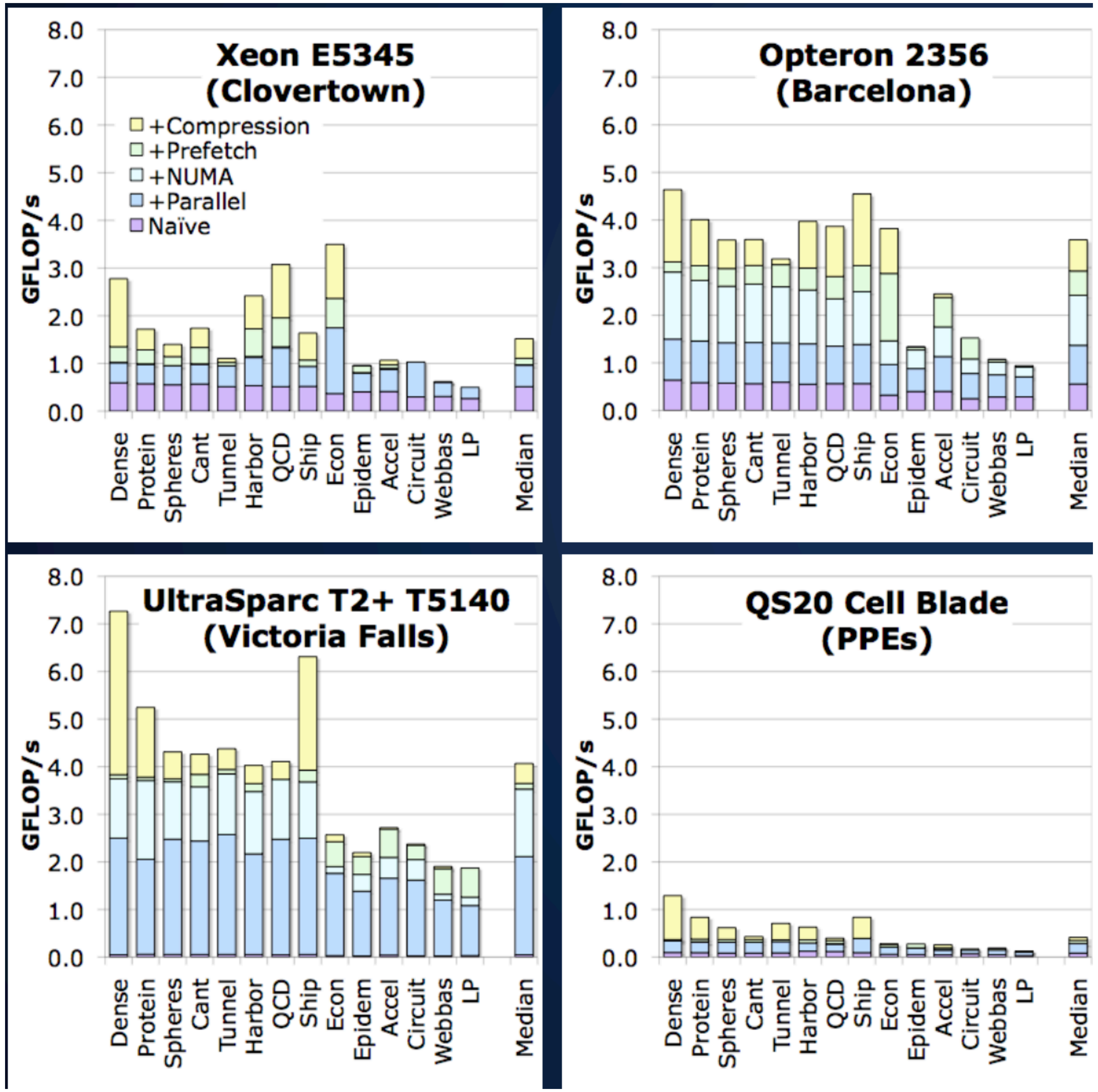
Out-of-the box SpMV performance on a suite of 14 matrices

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

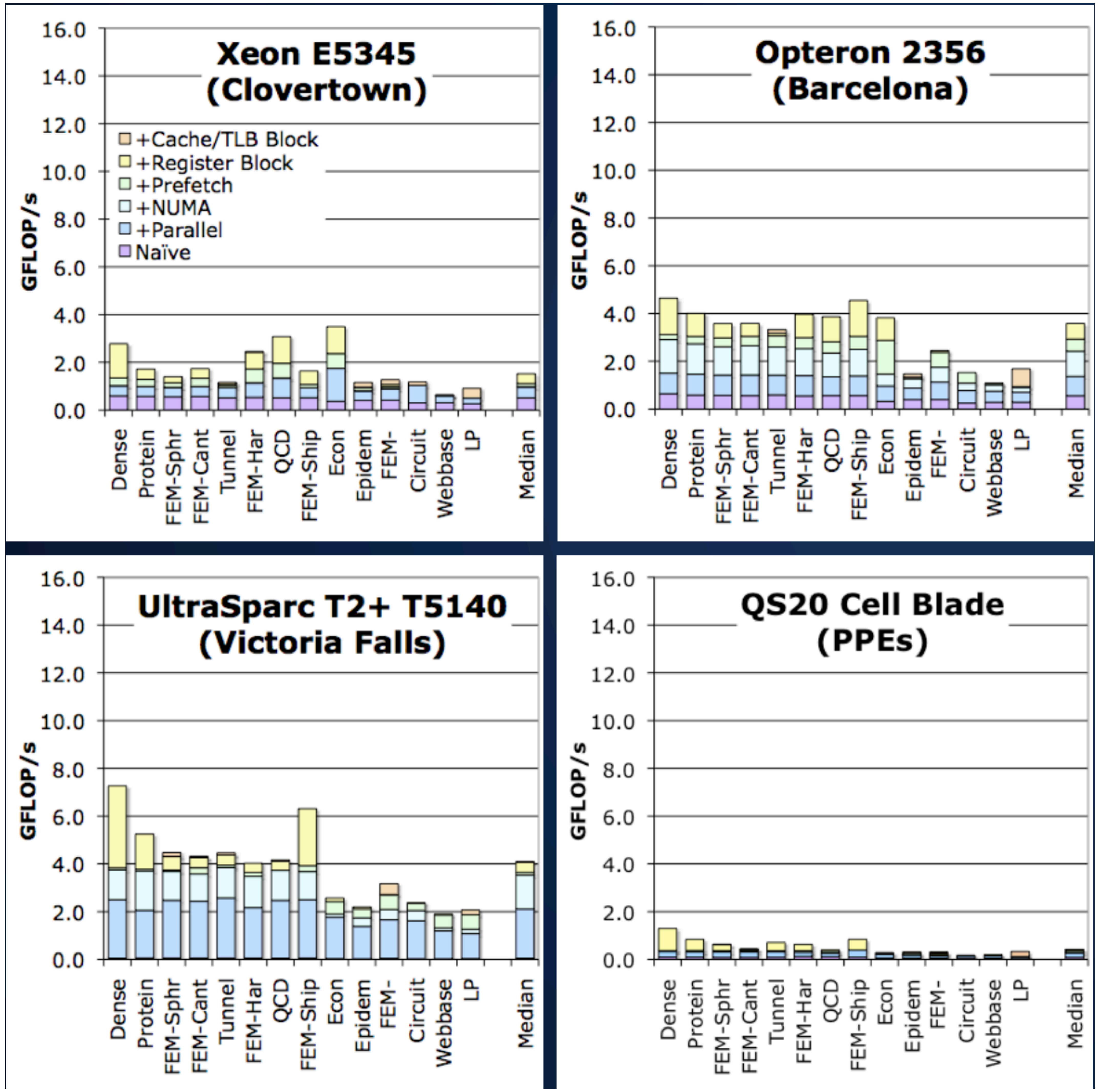
- register blocking
- other formats
- smaller indices

Benefit is matrix-dependent.

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

From UC Berkeley CS267

SpMV Performance (cache and TLB blocking)



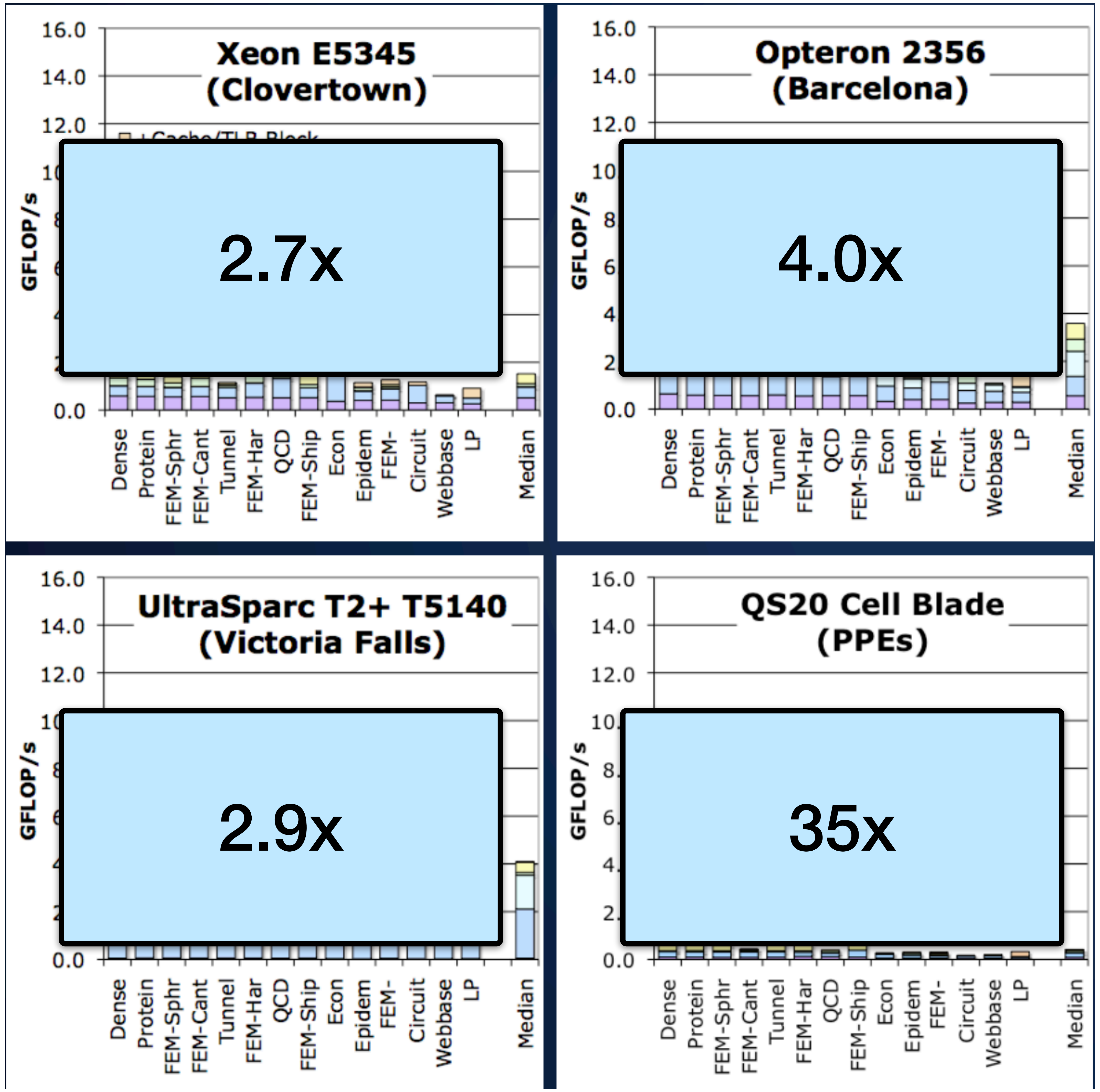
Fully auto-tuned SpMV performance across the suite of matrices

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

From UC Berkeley CS267

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.