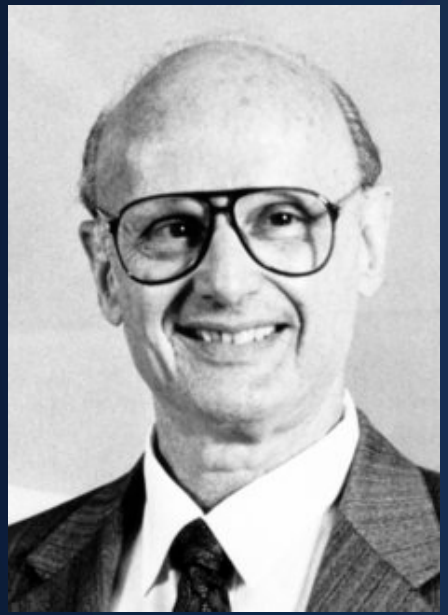
Applications of Parallel Computers

Sparse Matrix-Vector Multiply (SpMV) and Iterative Solvers

https://sites.google.com/lbl.gov/cs267-spr2021

1990 Nobel Prize in Economics

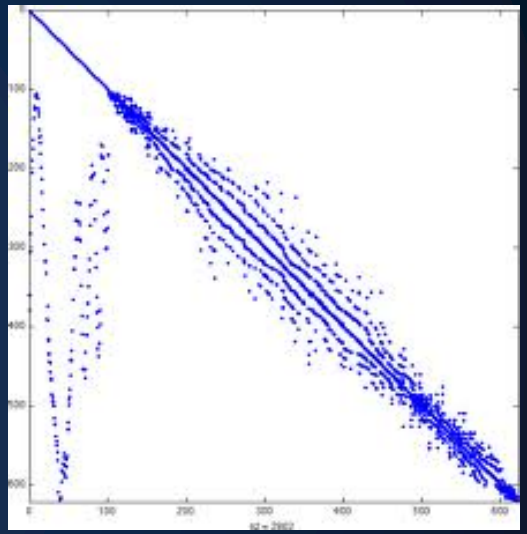


Our models strained the computer capabilities of the day [1950s]. I observed that most of the coefficients in our matrices were zero; *i.e.* , the nonzeros were “sparse” in the matrix

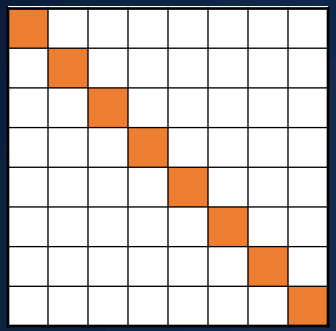
- Harry Markowitz

What is a sparse matrix?

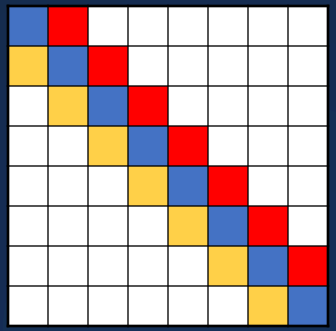
- A matrix with primarily zeros (>> 90%)
- Representing them using dense data structures wastes
  - Memory
  - Computation
- Sparse matrices arise in many applications
  - Simulating climate
  - Analyzing images (photos, MRIs,…)
  - Web page ranking for search
  - Graphs, including Graph Neural Nets



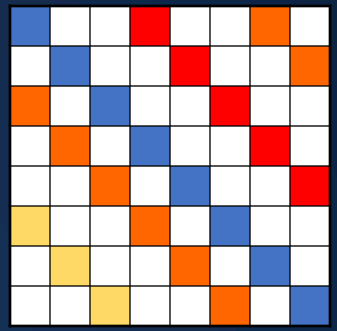
Examples of sparse matrices



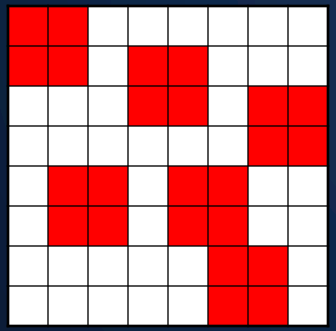
Diagonal



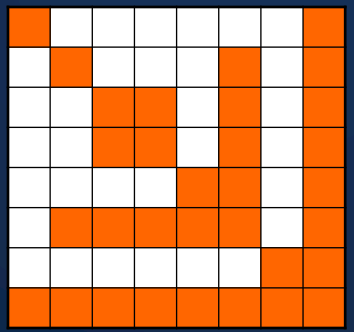
Tridiagonal



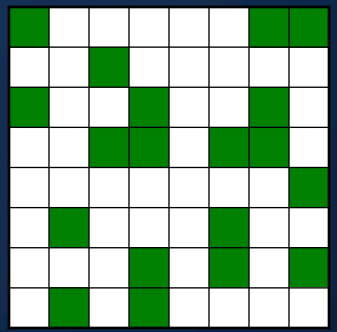
“Generalized diagonal”



Block matrix
(2x2 dense blocks)



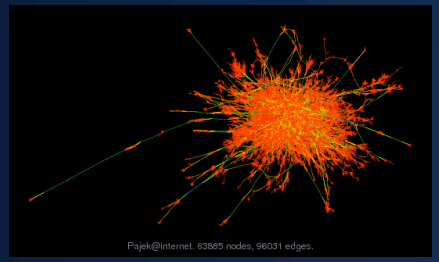
Symmetric

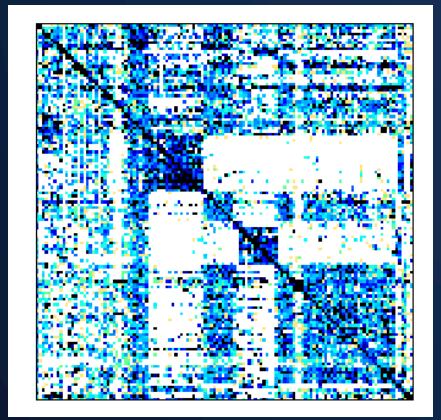


Irregular

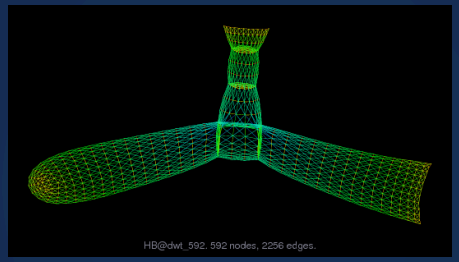
Sparse matrices are everywhere

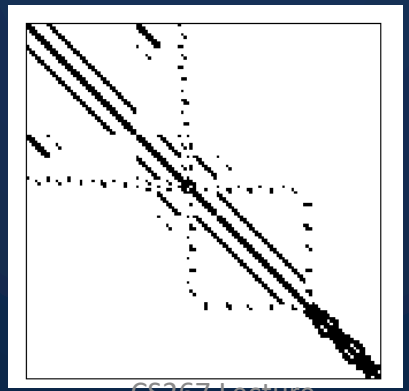
Internet connectivity



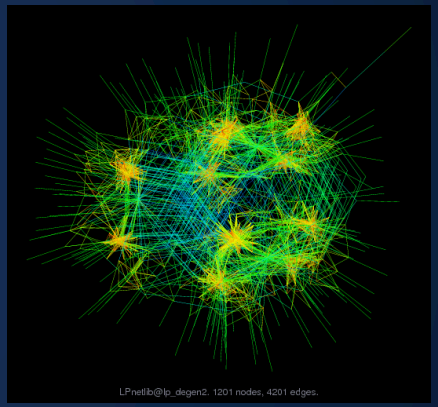


Structural design





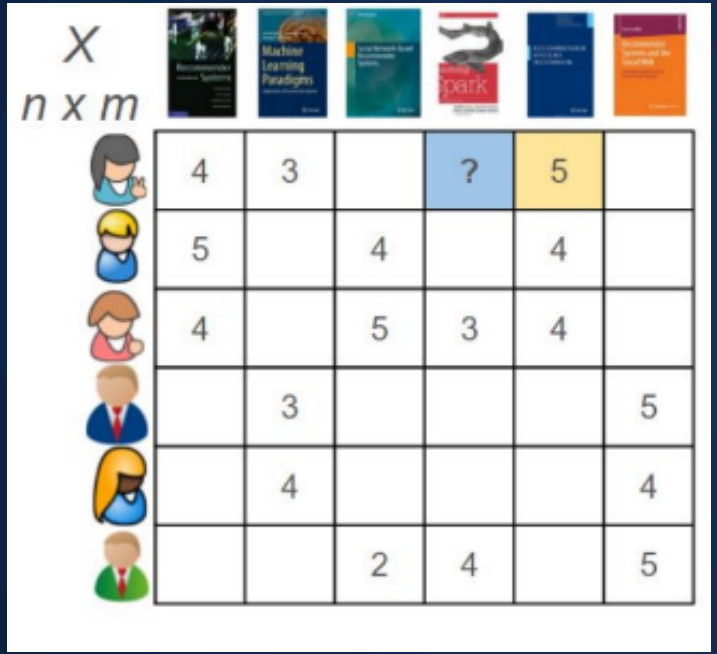
Linear Programming



Recommendation Matrix

Products

Population

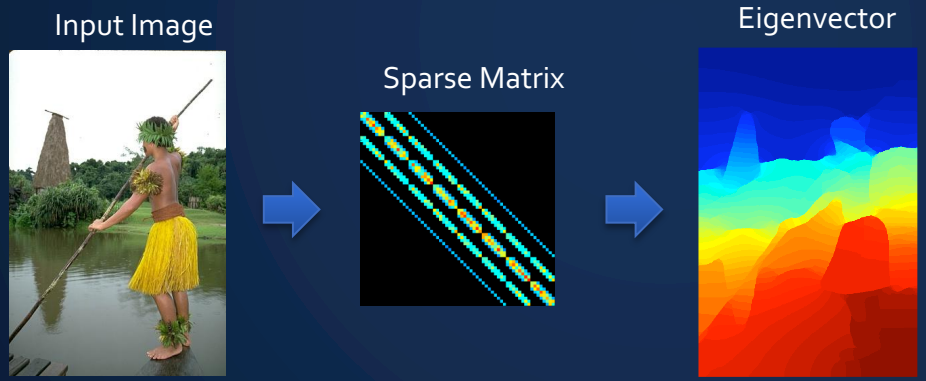


Ratings

Image: 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?
- Eigen problem - Find non-zero x such that $Ax = \lambda x$



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

More applications

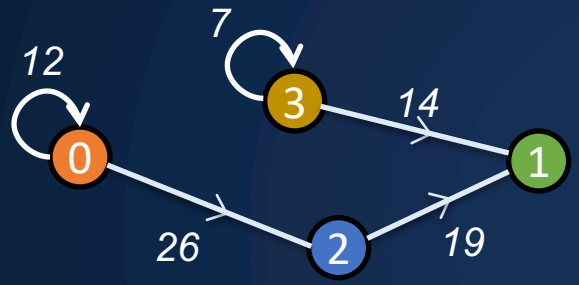
- Graphs
  - Google's PageRank (originally an eigen-problem on the web adjacency matrix)
  - Transportation network analysis
- Text analysis
  - Latent Semantic Indexing finds topics in a document corpus by doing a singular value decomposition of a bag-of-words matrix
- Scientific and engineering
  - Solving differential equations (climate modeling, etc.)
  - Optimization problems
- And many more…

Outline for today

- Sparse matrices in the world
- Sparse matrix formats and serial SpMV
- Parallel and distributed SpMV
- Register / cache blocking and autotuning SpMV
- CA iterative solvers
- Sparse matmul (SpGEMM, SPMM,...)

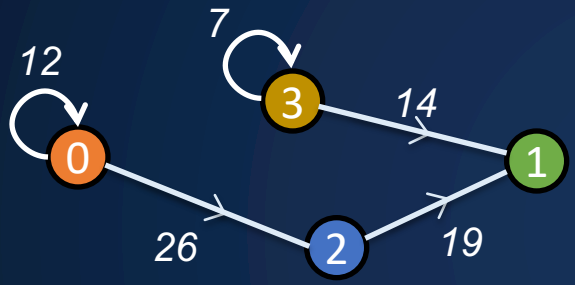
Graph representations

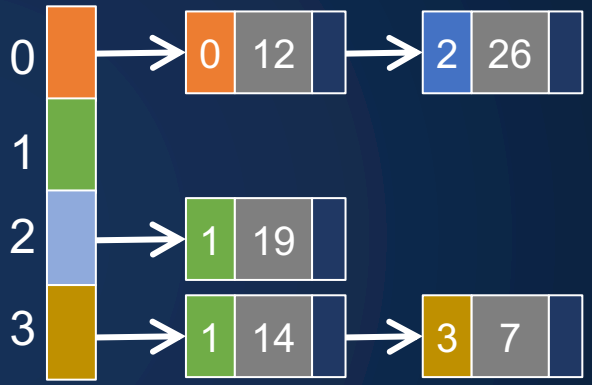
Compressed sparse row (CSR) = cache-efficient adjacency lists



Graph representations

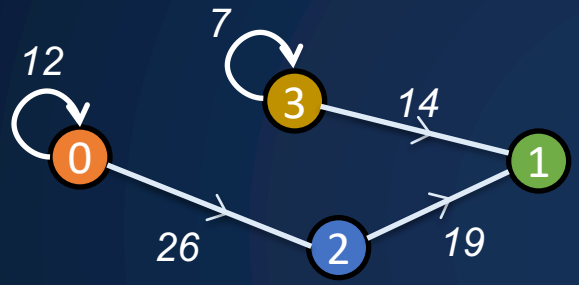
Compressed sparse row (CSR) = cache-efficient adjacency lists

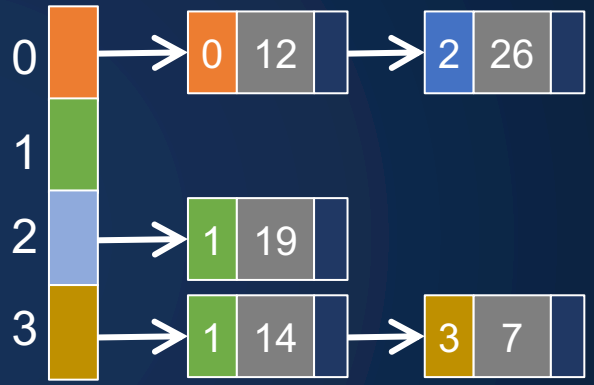




Graph representations

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

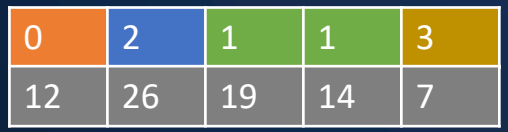




Index into adjacency array

(row starts in CSR)

Adjacencies



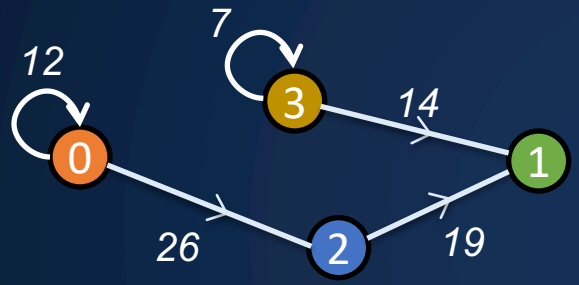
Weights

(column ids in CSR)

(numerical values in CSR)

Graph representations

Compressed sparse row (CSR) = compressed version of dense adjacency matrix



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

Index into adjacency array

0	2	2	3	5

(row starts in CSR)

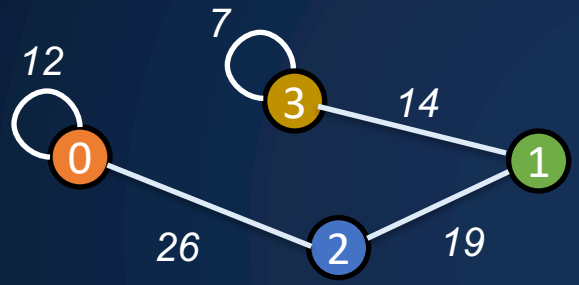
Adjacencies	0	2	1	1	3
Weights	12	26	19	14	7

(column ids in CSR)

(numerical values in CSR)

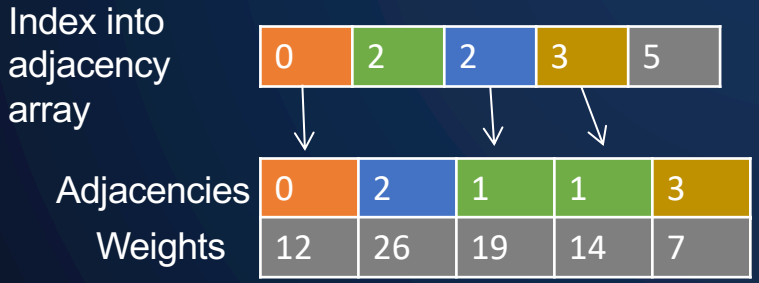
Directed vs Undirected

An undirected graph corresponds to a symmetric matrix – need only store ~half (triangle)



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

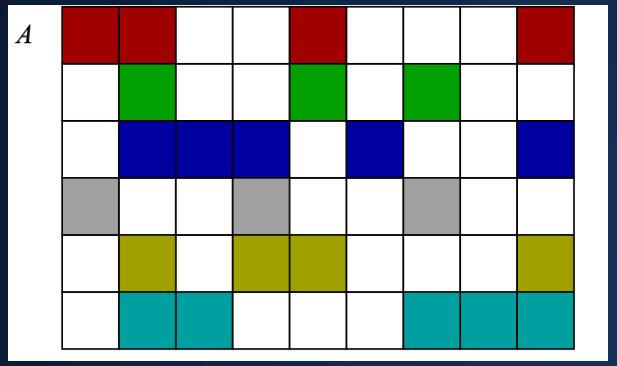
(row starts in CSR)

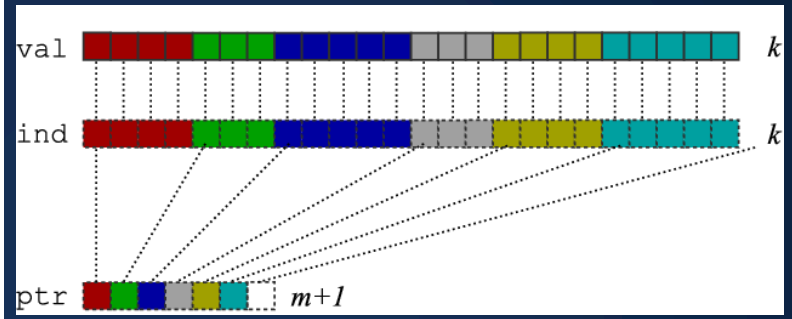


(column ids in CSR)

(numerical values in CSR)

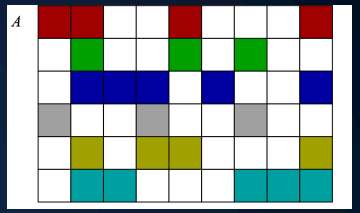
Compressed Sparse Row (CSR) Storage





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

Other Storage Formats



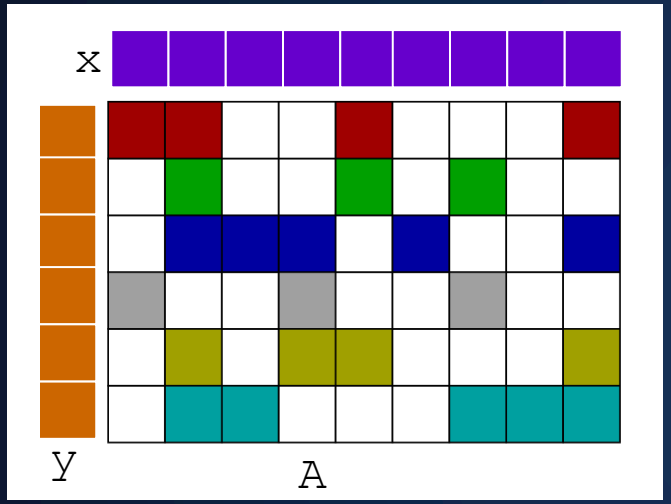
- Compressed Sparse Row (CSR) is most common and our focus today

Others include

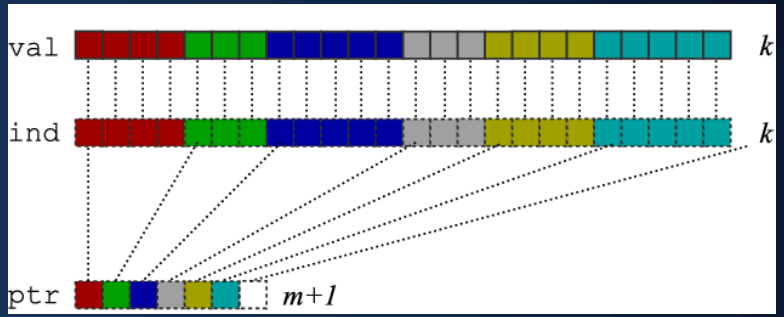
- Compressed Sparse Column (CSC)
- Coordinate (COO): row + column index per nonzero (easy to build / modify)
- Diagonal (DIAG): store main diagonal as 1D array; or diagonal bands as 2D (padded)
- Symmetric: store only ½ the array (indexing more complicated)
- Blocked: store each block contiguously
  - Register blocked: blocks a small and dense, avoid indexes within blocks
  - Cache blocked: blocks are large and themselves sparse

And many more specialized ones!

SpMV in CSR: Serial



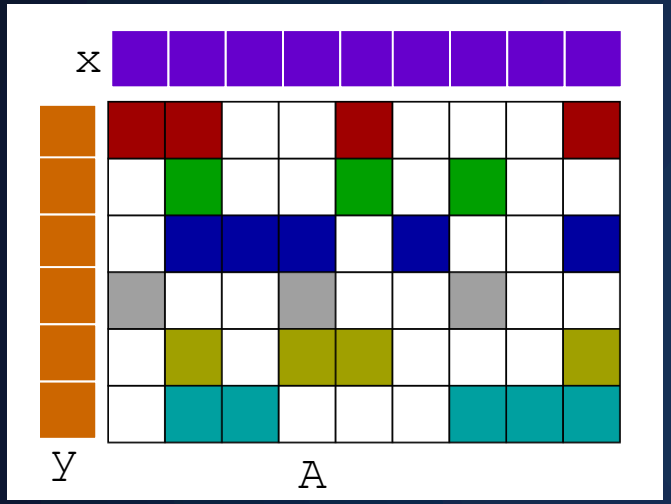
Representation of A



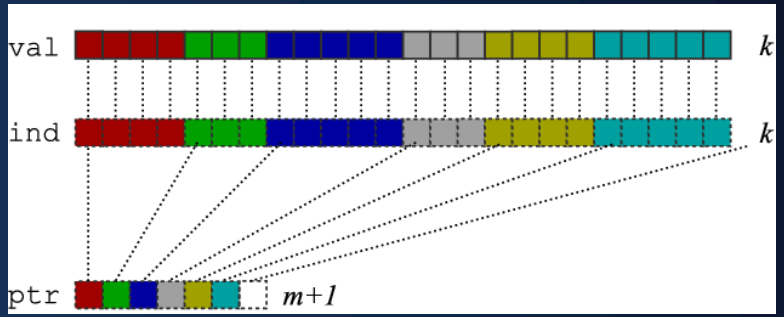
SpMV: Sparse Matrix-Vector Multiplication

CSR: Compressed Sparse Row

SpMV in CSR: Serial

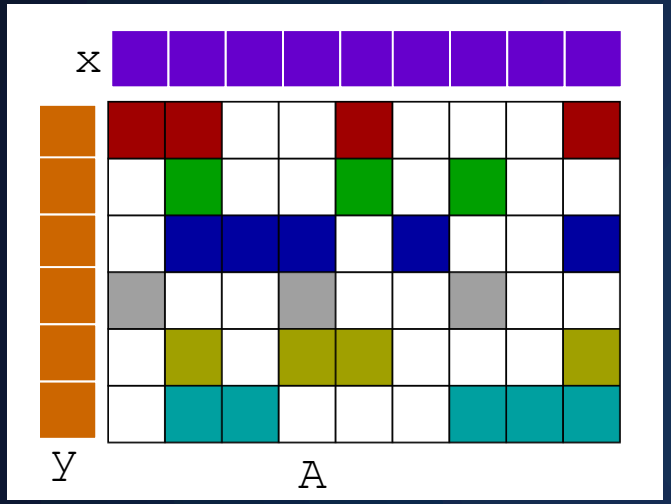


Representation of A

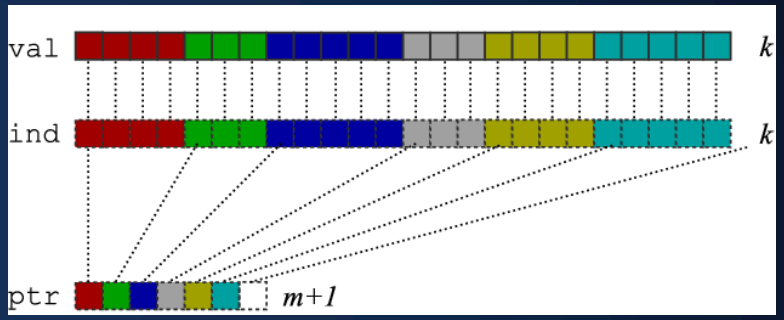


Matrix-vector multiply kernel: y(i) ← y(i) + A(i,j)*x(j)

SpMV in CSR: Serial



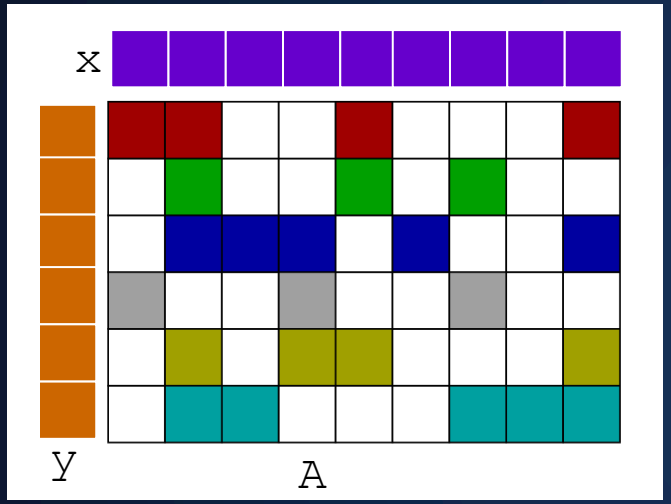
Representation of A



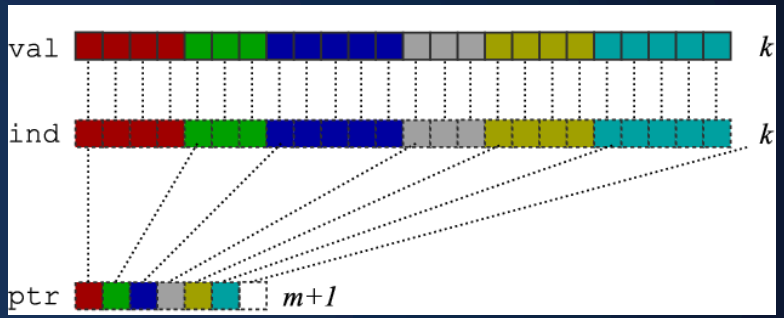
Matrix-vector multiply kernel: y(i) ← y(i) + A(i,j)*x(j)

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

SpMV in CSR: Serial



Representation of A



Matrix-vector multiply kernel: y(i) ← y(i) + A(i,j)*x(j)

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

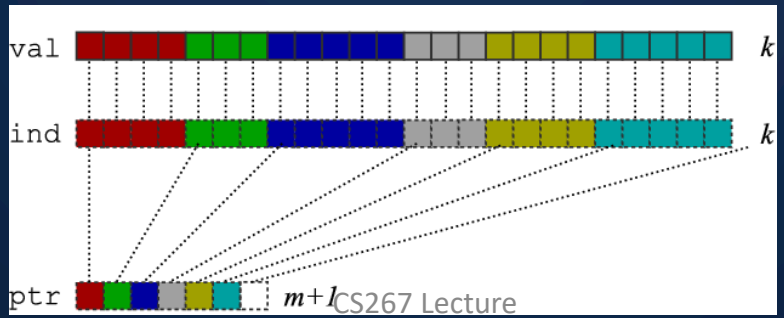
- BLAS2 not BLAS3, so no reuse in A
- Maximum reuse of y (full row) as written
- Re-use of x?

Outline for today

- Sparse matrices in the world
- Sparse matrix formats and serial SpMV
- Parallel and distributed SpMV
- Register / cache blocking and autotuning SpMV
- CA iterative solvers
- Sparse matmul (SpGEMM, SPMM,...)

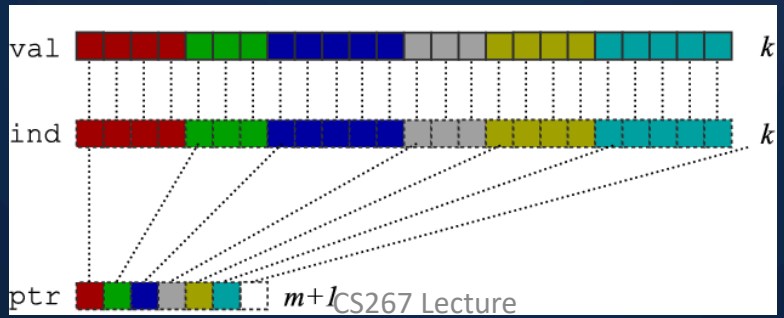
SpMV in CSR: OpenMP Parallel

```
#pragma omp parallel num_threads(thread_num)
{
#pragma omp for private(j, i, tmp) schedule(static)
   for (int i=0; i<m; i++)  {
        for (j = ptr[i]; j < ptr[i+1]; j++)  {
            tmp = ind[j];
            y[i] += val[j] * x[tmp];
        }
    }
}
```



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



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;
  }
}
```

For each row

For each 8 nonzeros

Gather X, load A
multiply, add to Y

In case # cols
does not divide
the vector width

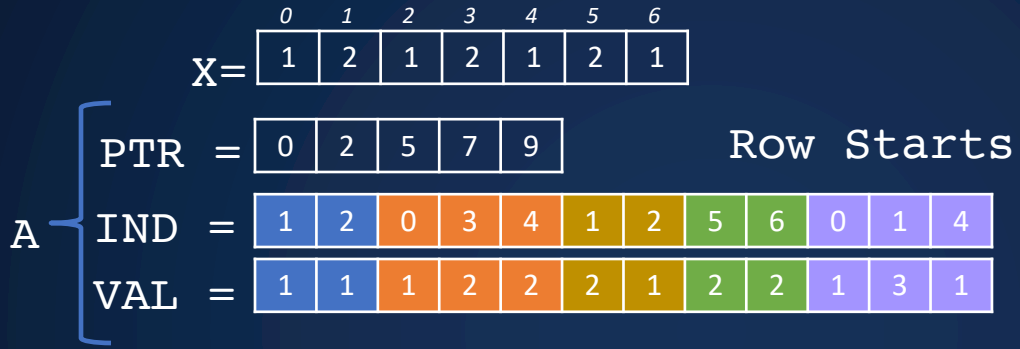
Code courtesy of David Sheffield

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;
    }
 }
```

SpMV (Segmented Suffix Scan)



Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors . Guy E. Blelloch, Michael A. Heroux, and Marco Zagha. CMU-CS-93-173

SpMV (Segmented Suffix Scan)

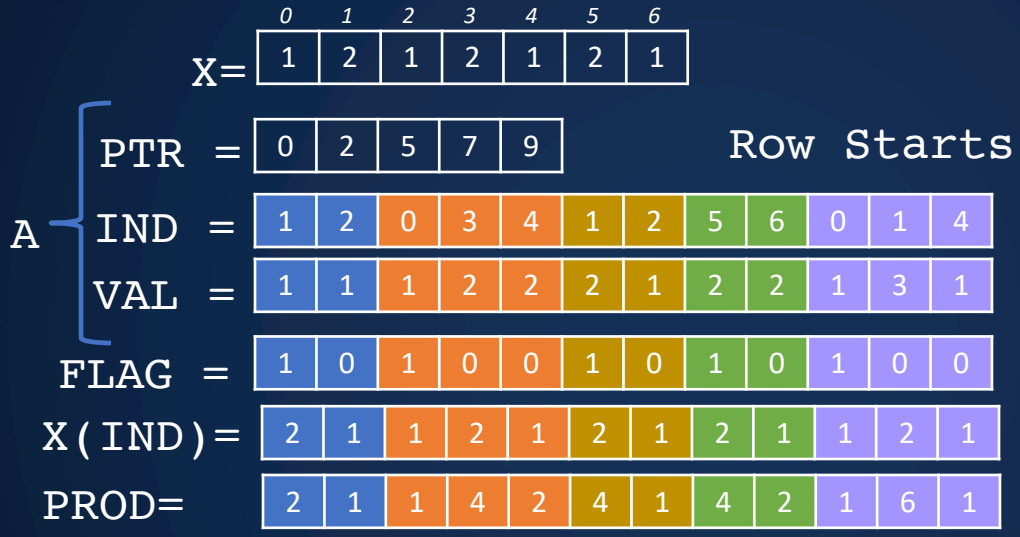

```
FLAG = ZEROS + ONES(PTR)
```

SpMV (Segmented Suffix Scan)


```
FLAG = ZEROS + ONES(PTR)
PROD = VAL * X(IND)
```


Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors . Guy E. Blelloch, Michael A. Heroux, and Marco Zagha. CMU-CS-93-173

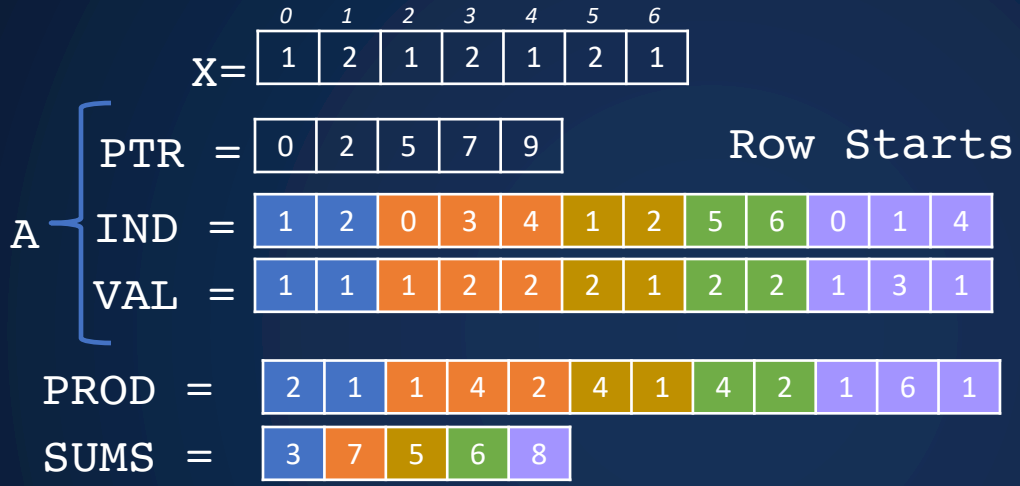
SpMV (Segmented Suffix Scan)




```
FLAG = ZEROS + ONES(PTR)
PROD = VAL * X(IND)
```


Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors . Guy E. Blelloch, Michael A. Heroux, and Marco Zagha. CMU-CS-93-173

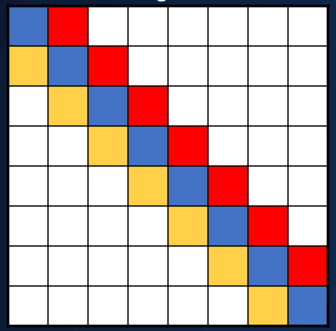
SpMV (Segmented Suffix Scan)

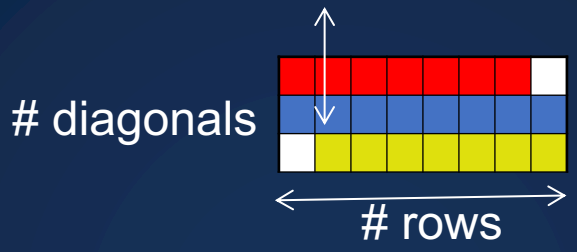



```
FLAG = ZEROS + ONES(PTR)
PROD = VAL * X(IND)
SUMS = SUM_SCAN(PROD,SEGMENT=SEGS
                        PREFIX(FLAG))
Y = SUMS(PTR)
```


Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors . Guy E. Blelloch, Michael A. Heroux, and Marco Zagha. CMU-CS-93-173

SpMV Diagonal format in OpenMP





val

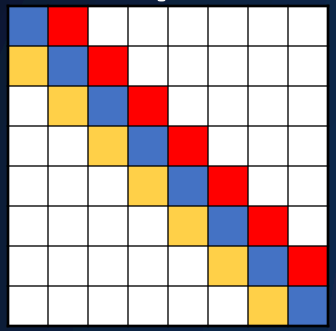
+1
0
-1

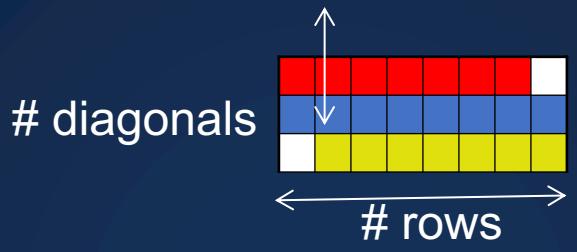
columnoffset

Matrix-vector multiply kernel: $y(i) \leftarrow y(i) + A(i,j)*x(j)$

```
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]
```

SpMV Diagonal format in OpenMP





+1
0
-1

columnoffset

val

Matrix-vector multiply kernel: $y(i) \leftarrow y(i) + A(i,j)*x(j)$

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

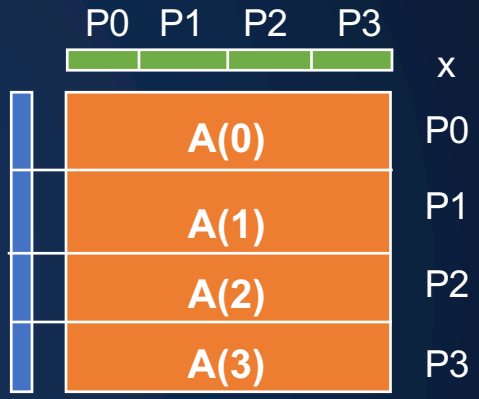
Diagonal also popular for GPUs / Vectors, including “jagged diagonal” to avoid over-padding, and sorting by lengths

Distribute 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
```

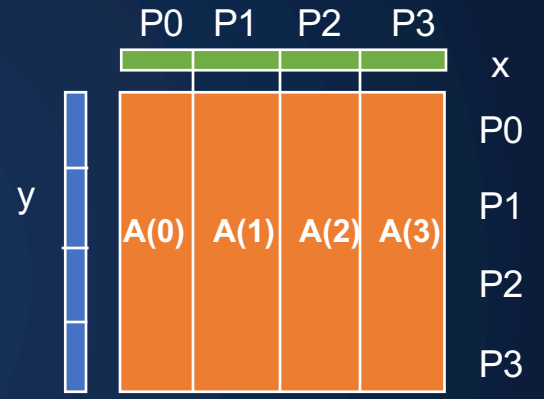


Broadcast + local dot product

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
  Reduce across all rows
   y(i) += SumReduce (temp(i))
```

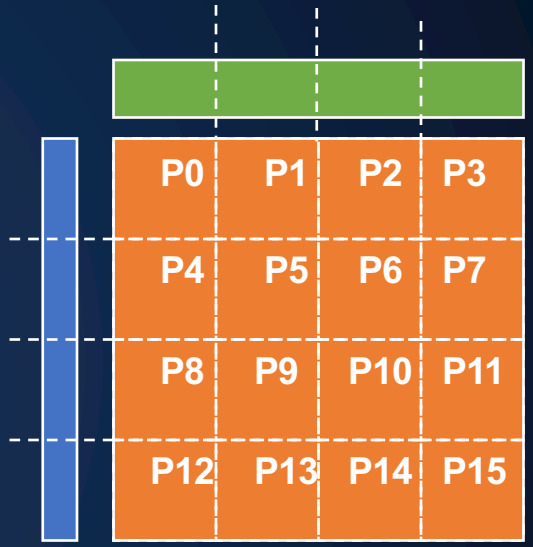


Local daxpy and parallel reduction

Distributed Dense Matrix-Vector Product (2D Blocked)

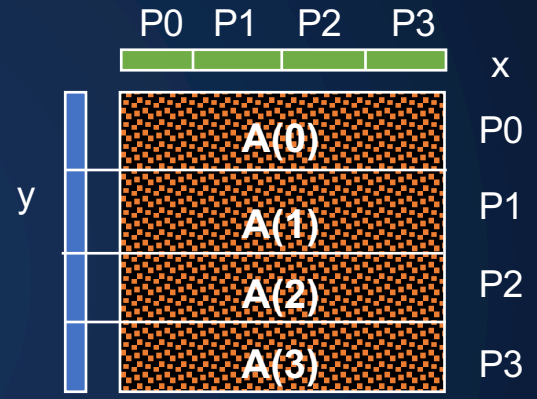
- A 2D blocked layout uses a broadcast and reduction
- Both on a subset of processors
  - sqrt(p) for square processor grid
  - Can use other rectangular shapes

  p_row * p_col = p



Distributed SpMV

1. Row parallelism (y & A partitioned)
   - Replicate x across processors
   - Or exchange only necessary elements
   - Are nonzeros clustered, e.g., near diagonal?

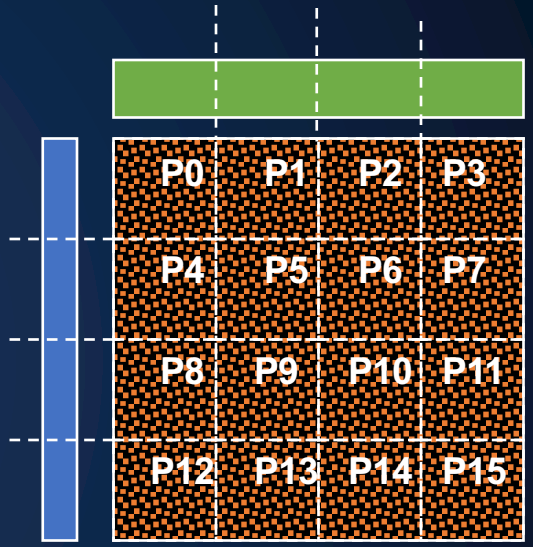


Distributed SpMV

1. Row parallelism (y & A partitioned)
   - Replicate x across processors
   - Or exchange only necessary elements
   - Are nonzeros clustered, e.g., near diagonal?
2. Column parallelism (x & A partitioned)
   - Make temporary temp_y = [0,…] on all processors;
   - Update that; and (sparse?) sum-reduce over processors

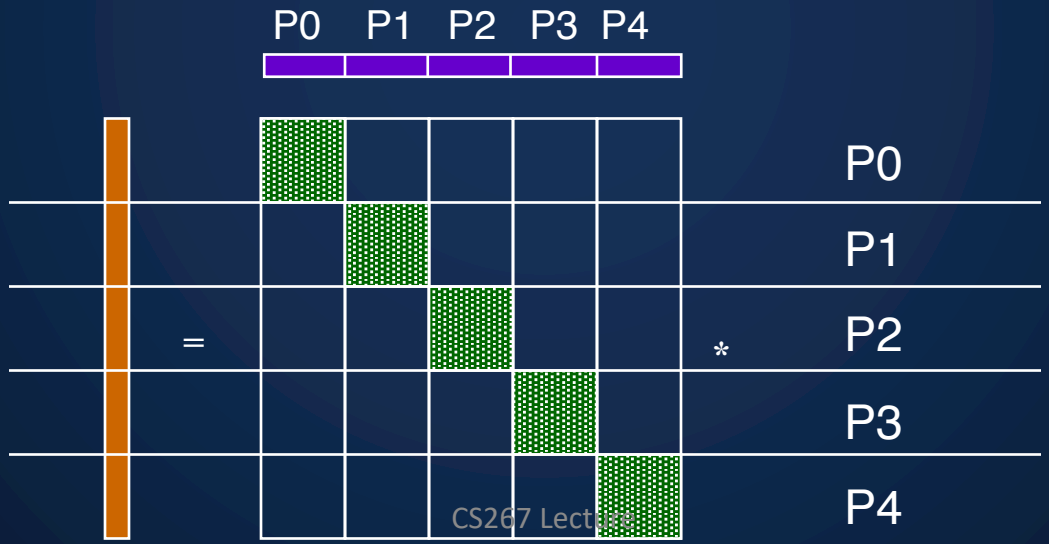
Distributed SpMV

1. Row parallelism (y & A partitioned)
   - Replicate x across processors
   - Or exchange only necessary elements
   - Are nonzeros clustered, e.g., near diagonal?
2. Column parallelism (x & A partitioned)
   - Make temporary temp_y = [0,…] on all processors;
   - Update that; and (sparse?) sum-reduce over processors
3. 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



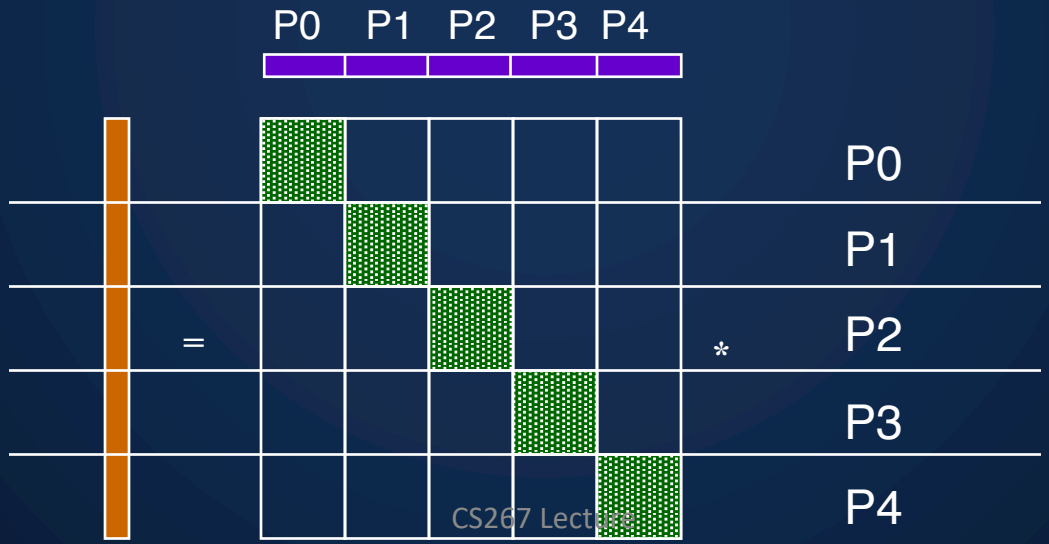
Ideal Sparse Structure: P diagonal Blocks

- "Ideal" matrix structure for parallelism: block diagonal
  - If p_i holds x_i and y_i, blocks, no vectors to communicate
  - If no non-zeros outside these blocks, no communication needed

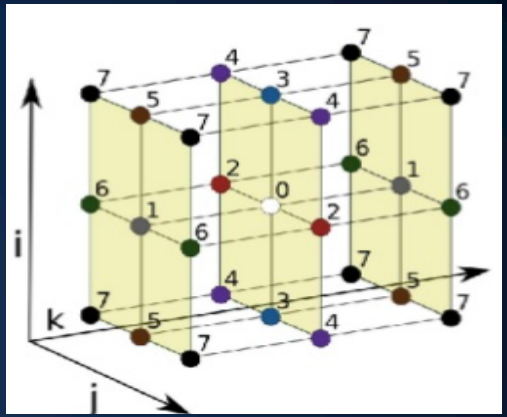


Matrix Reordering via Graph Partitioning

- "Ideal" matrix structure for parallelism: block diagonal
- Can we reorder the rows/columns to get close to this?
  - Most nonzeros in diagonal blocks, few outside



HPCG Benchmark



- Synthetic 3D PDE (FEM, FVM, FDM)
  - Single DOF heat diffusion model
  - Zero Dirichlet BCs, Synthetic RHS s.t. solution = 1
- Spatial domain:
  - Global domain: (n_x np_x x n_y np_y x n_z np_z)
  - Local domain: (n_x x n_y x n_z)
  - Process layout: (np_x x np_y x np_z)
- Sparse matrix:
  - 27 nonzeros per row interior
  - 7-18 on boundary
  - Symmetric positive definite

#	T	Site	Vendor	Computer	Country	HPCG [Pflop/s]	Rmax [Pflop/s]	HPCG/ Peak	HPCG/ HPL
1	1	RIKEN-CCS	Fujitsu	Fugaku Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D	Japan	16.005	442.0	3.0%	3.6%
2	2	Oak Ridge National Laboratory	IBM	Summit IBM Power System, P9 22C 3.07 GHz, Volta GV100, EDR	USA	2.926	148.6	1.5%	2.0%
3	3	Lawrence Livermore National Laboratory	IBM	Sierra IBM Power System, P9 22C 3.1 GHz, Volta GV100, EDR	USA	1.796	94.6	1.4%	1.9%
4	5	NVIDIA Corporation	NVIDIA	Selene DGX A100 SuperPOD, AMD 64C 2.25GHz, NVIDIA A100, Mellanox HDR	USA	1.623	63.5	2.1%	2.6%
5	7	Forschungszentrum Jülich (FZJ)	Atos	JUWELS Booster Module BullSequana XH2000, AMD EPYC 24C 2.8GHz, NVIDIA A100, Mell. HDR	Germany	1.275	44.1	1.8%	2.9%
6	10	Saudi Aramco	HPE	Dammam-7 Cray CS-Storm, Xeon 20C 2.5GHz, NVIDIA T. V100, IB HDR 100	Saudi Arabia	0.881	22.4	1.6%	3.9%
7	8	Eni S.p.A	Dell EMC	HPC5 PowerEdge C4140, Xeon 24C 2.1GHz, NVIDIA V100, Mellanox HDR	Italy	0.860	35.5	1.7%	2.4%
8	19	Japan Aerospace eXploration Agency	Fujitsu	TOKI-SORA PRIMEHPC FX1000, A64FX 48C 2.2GHz, Tofu interconnect D	Japan	0.614	16.6	3.2%	3.7%
9	13	Los Alamos NL / Sandia NL	Cray	Trinity Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries	USA	0.546	20.2	1.3%	2.7%
10	33	National Institute for Fusion Science (NIFS)	NEC	Plasma Simulator SX-Aurora TSUBASA A412-8, Vector Eng. Type10AE 8C 1.58GHz, IB HDR 200	Japan	0.529	7.9	5.0%	6.7%

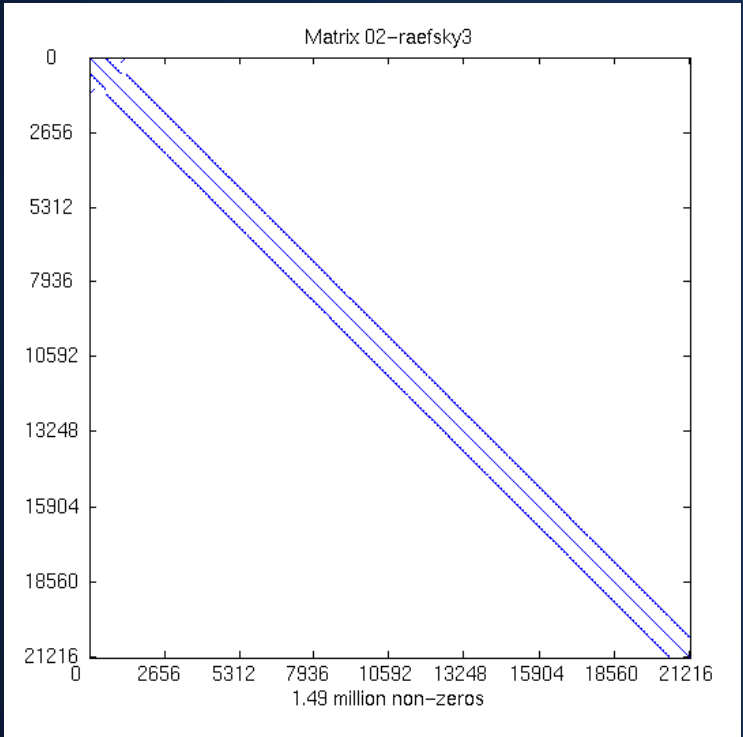
#	T	Site	Vendor	Computer	Country	HPCG [Pflop/s]	Rmax [Pflop/s]	HPCG/ Peak	HPCG/ HPL
1	1	RIKEN-CCS	Fujitsu	Fugaku Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D	Japan	16.005	442.0	3.0%	3.6%
2	2	Oak Ridge National Laboratory	IBM	Summit IBM Power System, P9 22C 3.07 GHz, Volta GV100, EDR	USA	2.926	148.6	1.5%	2.0%
3	3	Lawrence Livermore National Laboratory	IBM	Sierra IBM Power System, P9 22C 3.1 GHz, Volta GV100, EDR	USA	1.796	94.6	1.4%	1.9%
4	5	NVIDIA Corporation	NVIDIA	Selene DGX A100 SuperPOD, AMD 64C 2.25GHz, NVIDIA A100, Mellanox HDR	USA	1.623	63.5	2.1%	2.6%
5	7	Forschungszentrum Jülich (FZJ)	Atos	JUWELS Booster Module BullSequana XH2000, AMD EPYC 24C 2.8GHz, NVIDIA A100, Mell. HDR	Germany	1.275	44.1	1.8%	2.9%
6	10	Saudi Aramco	HPE	Dammam-7 Cray CS-Storm, Xeon 20C 2.5GHz, NVIDIA T. V100, IB HDR 100	Saudi Arabia	0.881	22.4	1.6%	3.9%
7	8	Eni S.p.A	Dell EMC	HPC5 PowerEdge C4140, Xeon 24C 2.1GHz, NVIDIA V100, Mellanox HDR	Italy	0.860	35.5	1.7%	2.4%
8	19	Japan Aerospace eXploration Agency	Fujitsu	TOKI-SORA PRIMEHPC FX1000, A64FX 48C 2.2GHz, Tofu interconnect D	Japan	0.614	16.6	3.2%	3.7%
9	13	Los Alamos NL / Sandia NL	Cray	Trinity Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries	USA	0.546	20.2	1.3%	2.7%
10	33	National Institute for Fusion Science (NIFS)	NEC	Plasma Simulator SX-Aurora TSUBASA A412-8, Vector Eng. Type10AE 8C 1.58GHz, IB HDR 200	Japan	0.529	7.9	5.0%	6.7%

#	T	Site	Vendor	Computer	Country	HPCG [Pflop/s]	Rmax [Pflop/s]	HPCG/ Peak	HPCG/ HPL
1	1	RIKEN-CCS	Fujitsu	Fugaku Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D	Japan	16.005	442.0	3.0%	3.6%
2	2	Oak Ridge National Laboratory	IBM	Summit IBM Power System, P9 22C 3.07 GHz, Volta GV100, EDR	USA	2.926	148.6	1.5%	2.0%
3	3	Lawrence Livermore National Laboratory	IBM	Sierra IBM Power System, P9 22C 3.1 GHz, Volta GV100, EDR	USA	1.796	94.6	1.4%	1.9%
4	5	NVIDIA Corporation	NVIDIA	Selene DGX A100 SuperPOD, AMD 64C 2.25GHz, NVIDIA A100, Mellanox HDR	USA	1.623	63.5	2.1%	2.6%
5	7	Forschungszentrum Jülich (FZJ)	Atos	JUWELS Booster Module BullSequana XH2000, AMD EPYC 24C 2.8GHz, NVIDIA A100, Mell. HDR	Germany	1.275	44.1	1.8%	2.9%
6	10	Saudi Aramco	HPE	Dammam-7 Cray CS-Storm, Xeon 20C 2.5GHz, NVIDIA T. V100, IB HDR 100	Saudi Arabia	0.881	22.4	1.6%	3.9%
7	8	Eni S.p.A	Dell EMC	HPC5 PowerEdge C4140, Xeon 24C 2.1GHz, NVIDIA V100, Mellanox HDR	Italy	0.860	35.5	1.7%	2.4%
8	19	Japan Aerospace eXploration Agency	Fujitsu	TOKI-SORA PRIMEHPC FX1000, A64FX 48C 2.2GHz, Tofu interconnect D	Japan	0.614	16.6	3.2%	3.7%
9	13	Los Alamos NL / Sandia NL	Cray	Trinity Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries	USA	0.546	20.2	1.3%	2.7%
10	33	National Institute for Fusion Science (NIFS)	NEC	Plasma Simulator SX-Aurora TSUBASA A412-8, Vector Eng. Type10AE 8C 1.58GHz, IB HDR 200	Japan	0.529	7.9	5.0%	6.7%

Outline for today

- Sparse matrices in the world
- Sparse matrix formats and serial SpMV
- Parallel and distributed SpMV
- Register / cache blocking and autotuning SpMV
- CA iterative solvers
- Sparse matmul (SpGEMM, SPMM,...)

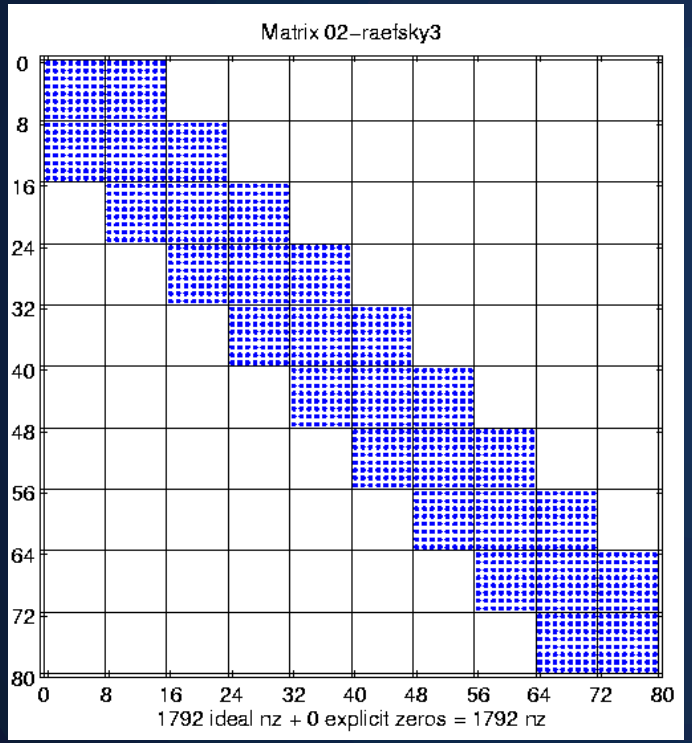
Changing Matrix Format: Register Blocking



- n = 21200
- nnz = 1.5 M
- kernel: SpMV

- Source: NASA structural analysis problem

Changing Matrix Format: Register Blocking



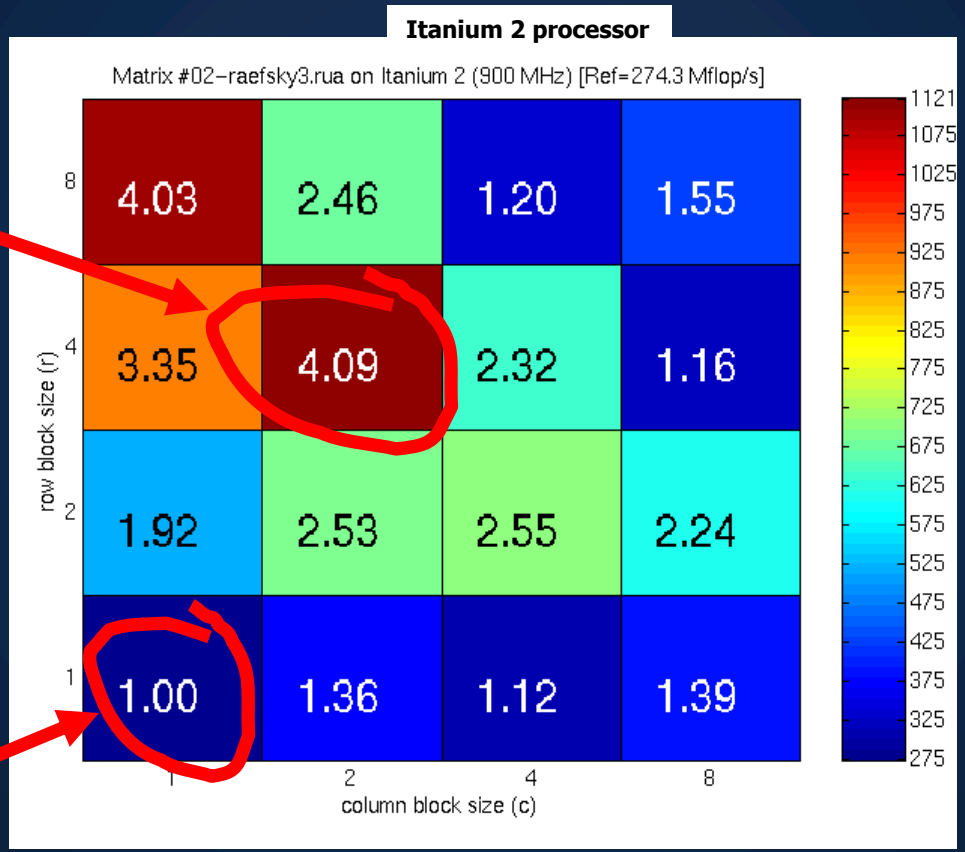
- n = 21200
- nnz = 1.5 M
- kernel: SpMV

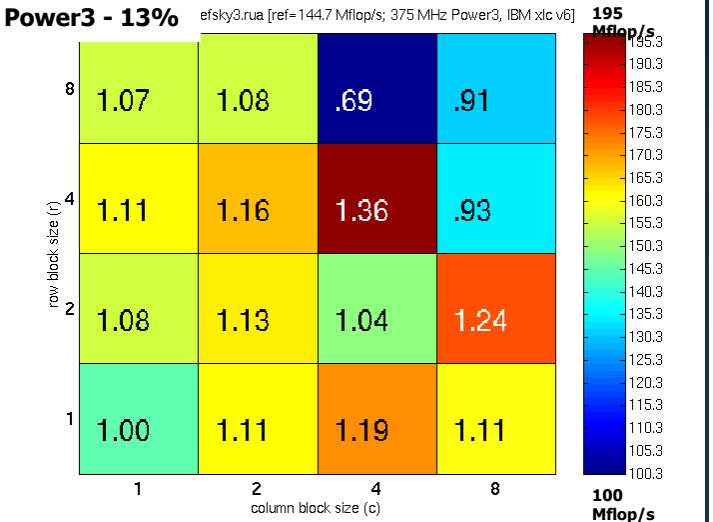
- Source: NASA structural analysis problem

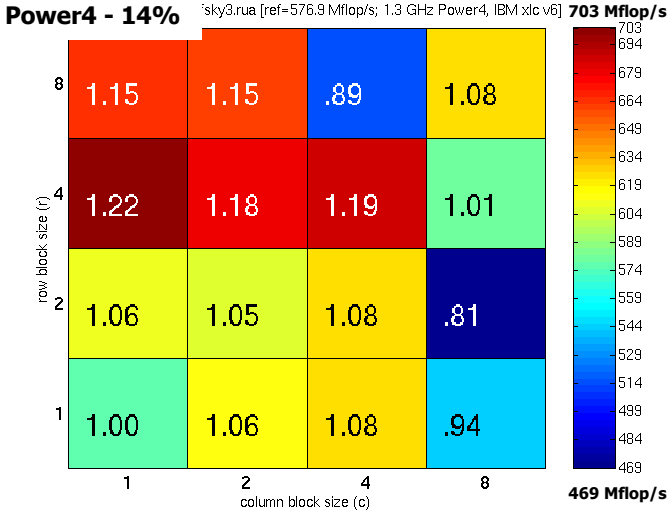
Using block structure in SpMV

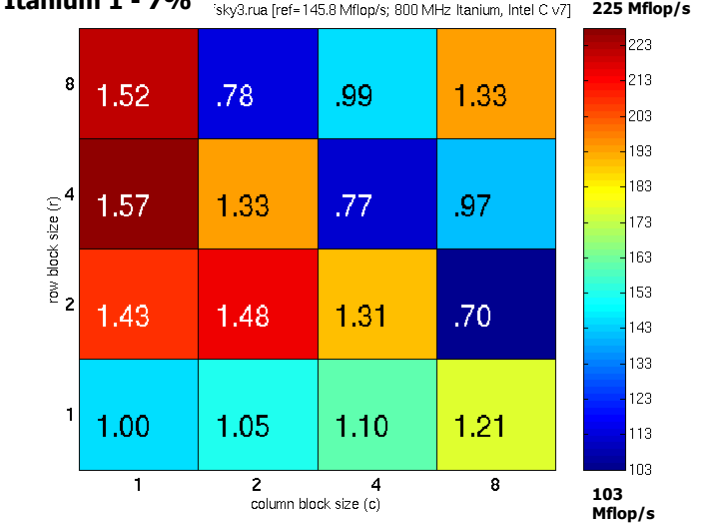
- Bottleneck is time to get matrix from memory
  - Only 2 flops for each nonzero in matrix
  - Fetching at ~1 int (column index) + 1 float (value) for 2 flops
- Don't store each nonzero with index, instead store each nonzero r-by-c block with 1 column index
  - As r*c grows, storage drops by up to 2x, for all 32-bit quantities
  - Time to fetch matrix from memory decreases
- Change both data structure and algorithm
  - Need to pick r and c
  - Need to change algorithm accordingly
- In example, is r=c=8 best choice?
  - Minimizes storage, so looks like a good idea…
- Consider best case: dense matrix in sparse format

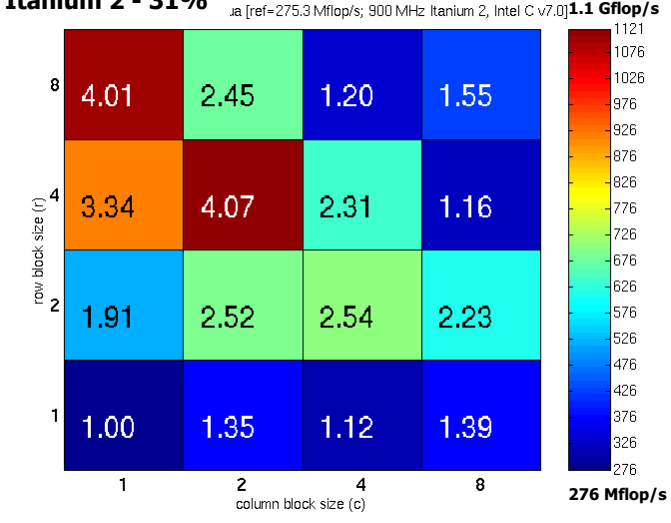
The Need for Search

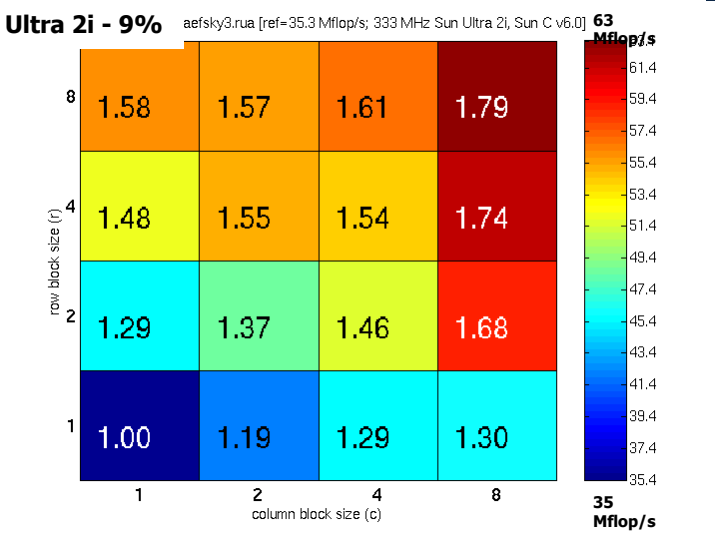


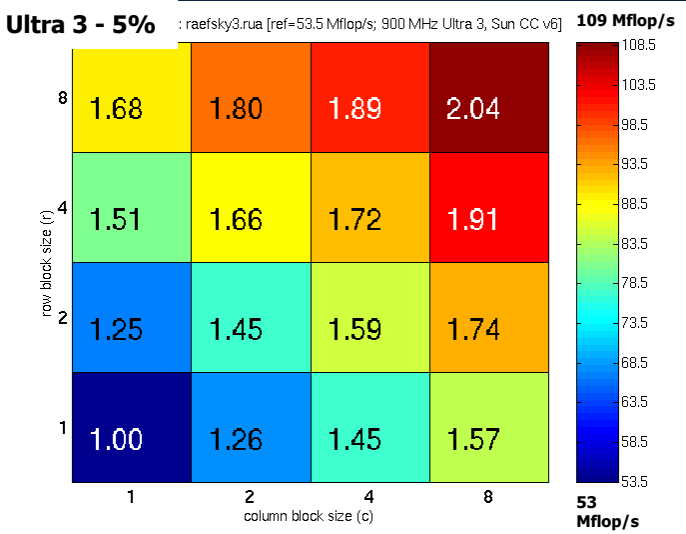

Power3 - 13%
efsky3.rua [ref=144.7 Mflop/s; 375 MHz Power3, IBM xlc v6]
195 Mflop/s
1.07 1.08 .69 .91
1.11 1.16 1.36 .93
1.08 1.13 1.04 1.24
1.00 1.11 1.19 1.11
row block size (r)
column block size (c)
100 Mflop/s

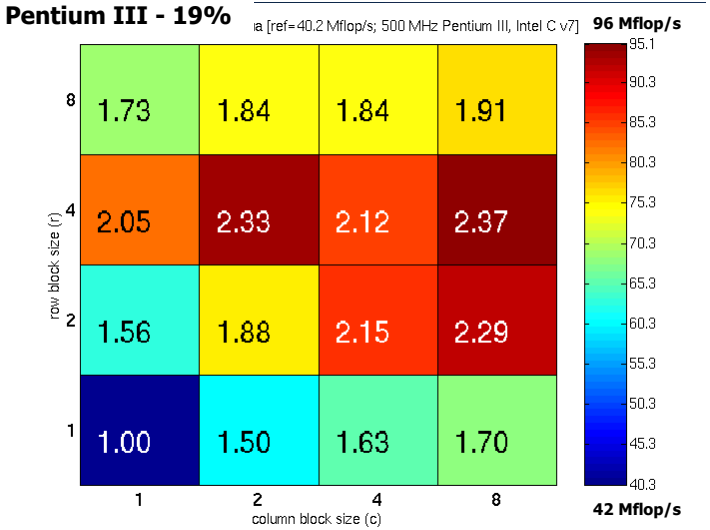

Power4 - 14%
sky3.rua [ref=576.9 Mflop/s; 1.3 GHz Power4, IBM xlc v6]
703 Mflop/s
1.15 1.15 .89 1.08
1.22 1.18 1.19 1.01
1.06 1.05 1.08 .81
1.00 1.06 1.08 .94
row block size (r)
column block size (c)
469 Mflop/s

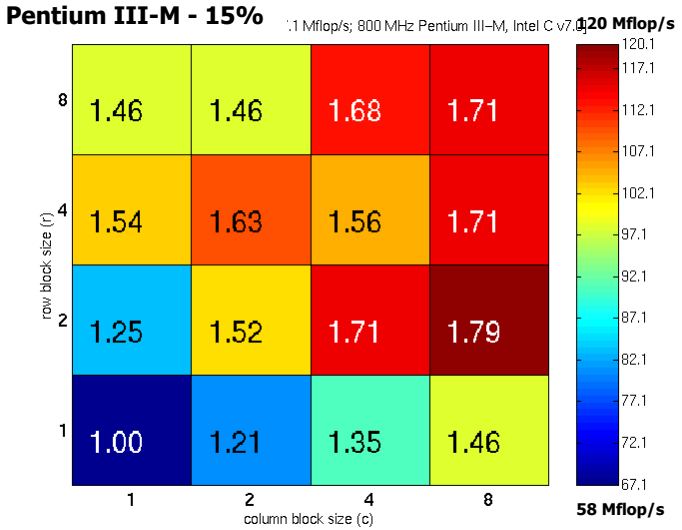

Itanium 1 - 7%
sky3.rua [ref=145.8 Mflop/s; 800 MHz Itanium, Intel C v7]
225 Mflop/s
1.52 .78 .99 1.33
1.57 1.33 .77 .97
1.43 1.48 1.31 .70
1.00 1.05 1.10 1.21
row block size (r)
column block size (c)
103 Mflop/s


Itanium 2 - 31%
ua [ref=275.3 Mflop/s; 900 MHz Itanium 2, Intel C v7.0]
1.1 Gflop/s
4.01 2.45 1.20 1.55
3.34 4.07 2.31 1.16
1.91 2.52 2.54 2.23
1.00 1.35 1.12 1.39
row block size (r)
column block size (c)
276 Mflop/s

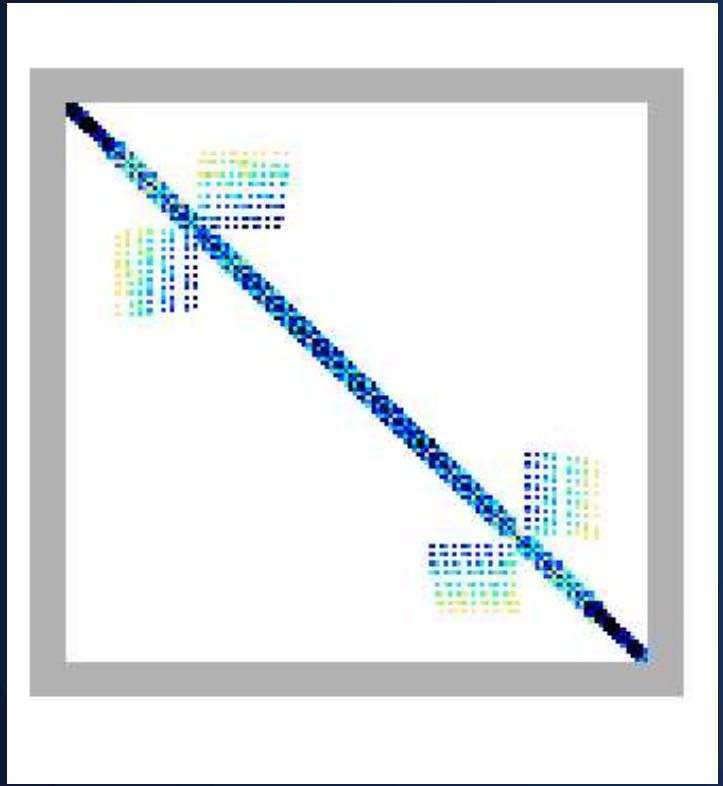

Ultra 2i - 9%
aefsky3.rua [ref=35.3 Mflop/s; 333 MHz Sun Ultra 2i, Sun C v6.0]
63 Mflop/s
35 Mflop/s
row block size (r)
column block size (c)
1.58 1.57 1.61 1.79
1.48 1.55 1.54 1.74
1.29 1.37 1.46 1.68
1.00 1.19 1.29 1.30


Ultra 3 - 5%
: raefsky3.rua [ref=53.5 Mflop/s; 900 MHz Ultra 3, Sun CC v6]
109 Mflop/s
53 Mflop/s
row block size (r)
column block size (c)
1.68 1.80 1.89 2.04
1.51 1.66 1.72 1.91
1.25 1.45 1.59 1.74
1.00 1.26 1.45 1.57


Pentium III - 19%
ia [ref=40.2 Mflop/s; 500 MHz Pentium III, Intel C v7]
96 Mflop/s
42 Mflop/s
row block size (r)
column block size (c)
1.73 1.84 1.84 1.91
2.05 2.33 2.12 2.37
1.56 1.88 2.15 2.29
1.00 1.50 1.63 1.70


Pentium III-M - 15%
:1 Mflop/s; 800 MHz Pentium III-M, Intel C v7.1]
120 Mflop/s
58 Mflop/s
row block size (r)
column block size (c)
1.46 1.46 1.68 1.71
1.54 1.63 1.56 1.71
1.25 1.52 1.71 1.79
1.00 1.21 1.35 1.46

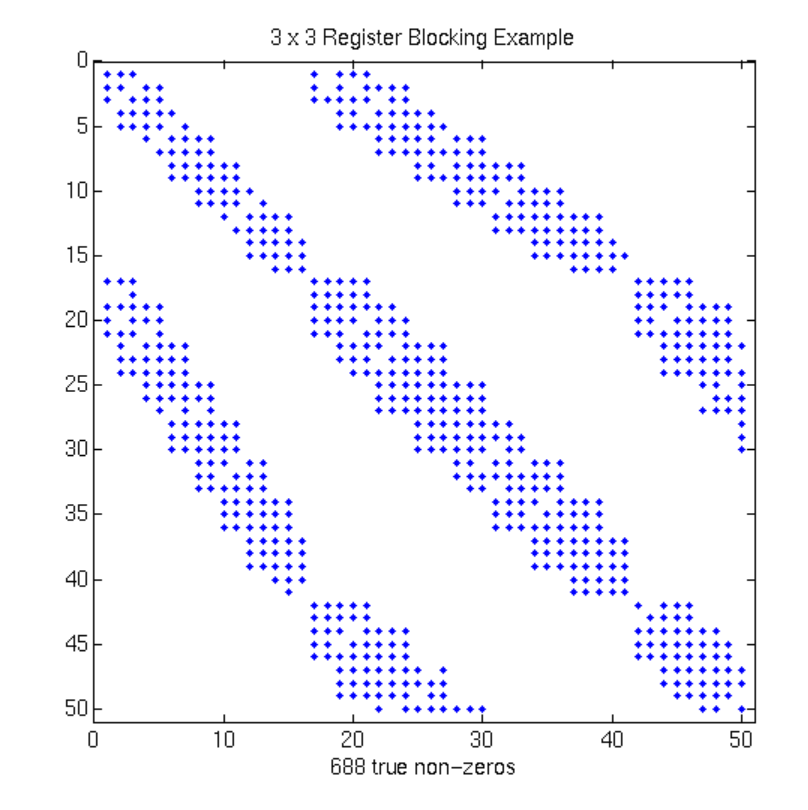
But most matrices don't block so easily



- FEM Fluid dynamics problems
- More complicated non-zero structure in general

- N = 16614
- NNZ = 1.1M

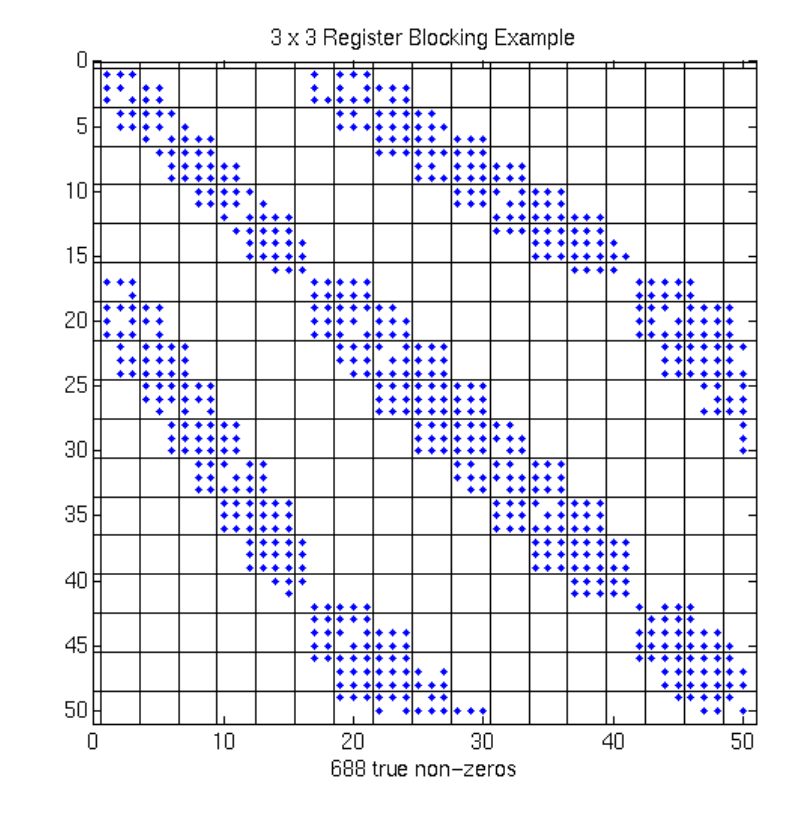
Zoom in to top corner



- More complicated non-zero structure

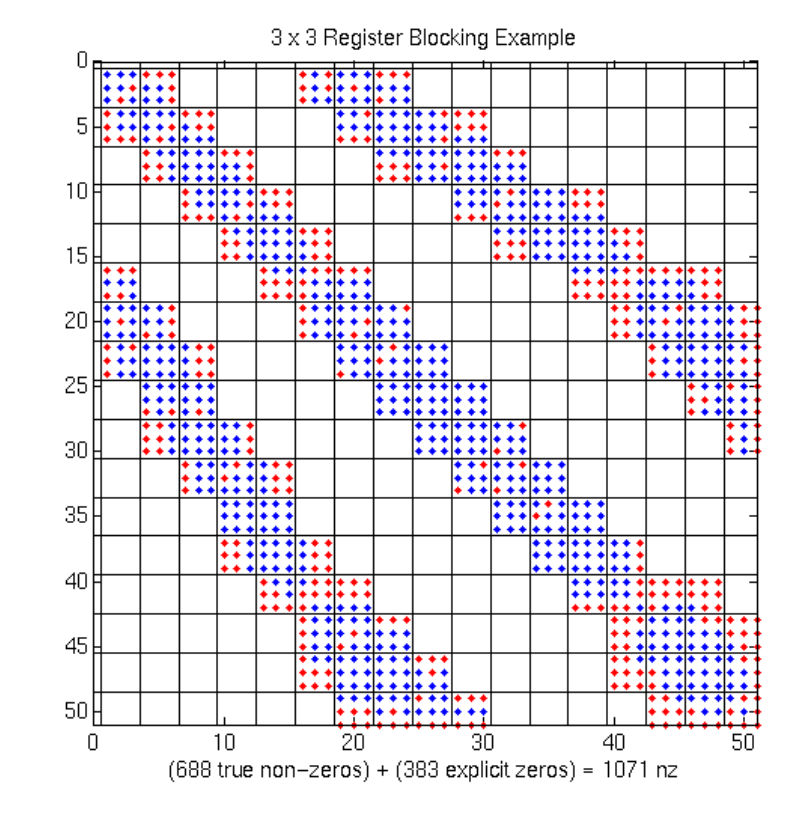
- N = 16614
- NNZ = 1.1M

3x3 blocks look natural, but…



- More complicated non-zero structure
- Example: 3x3 blocks
  - Grid of 3x3 cells
  - Many cell are not full
- N = 16614
- NNZ = 1.1M

Extra work can improve efficiency



- More complicated non-zero structure
- Example: 3x3 blocks
  - Grid of 3x3 cells
  - Add explicit zeros: 1.5x "fill overhead"
  - Unroll loops
- More work but faster
  - 1.5x faster on PIII

Libraries for sparse matrices?

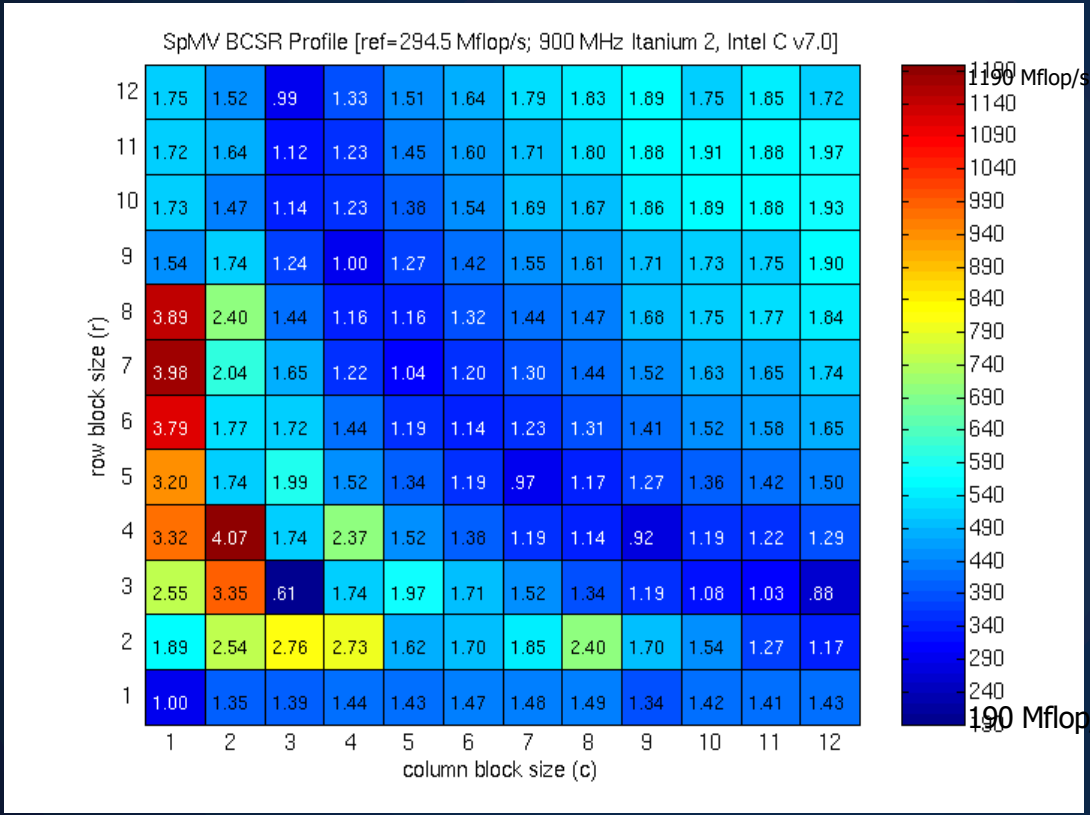
- How to build optimized library when:
  - Formats are not known? Libraries like PETSc and Trilinos will let the user provide format and SpMV
- How to build optimized matrix kernel library (BLAS)?
  - Nonzero structure is key to optimization
- OSKI = Optimized Sparse Kernel Interface
  - pOSKI for multicore
- BeBOP: Berkeley Benchmarking and Optimization Group
  - Many results shown from current and former members
  - Meet weekly (contact Jim or Kathy if interested)

Automatic Register Block Size Selection

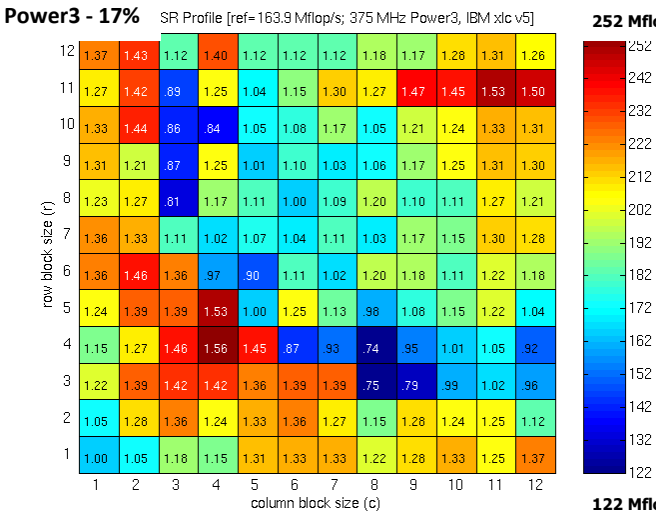
- Selecting the r x c block size
  - **Off-line benchmark of "register profile"**
    - Precompute **Mflops(r,c)** using *dense A in sparse format (blocked sparse row)* for each r x c
    - Once per machine/architecture
  - **Run-time "search"**
    - Sample *A* to estimate **Fill(r,c)** for each r x c
  - **Run-time heuristic model**
    - Choose r, c to minimize **time ~ Fill(r,c) / Mflops(r,c)**

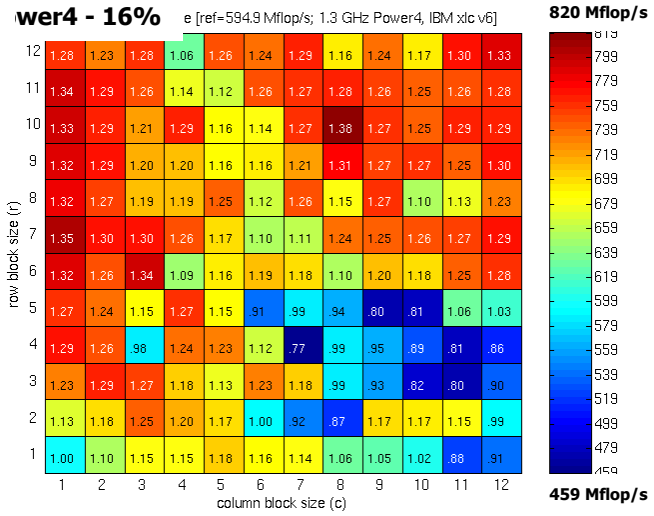
Eun-Jin Im PhD thesis: http://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-00-1104.pdf
Rich Vuduc PhD thesis: http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf

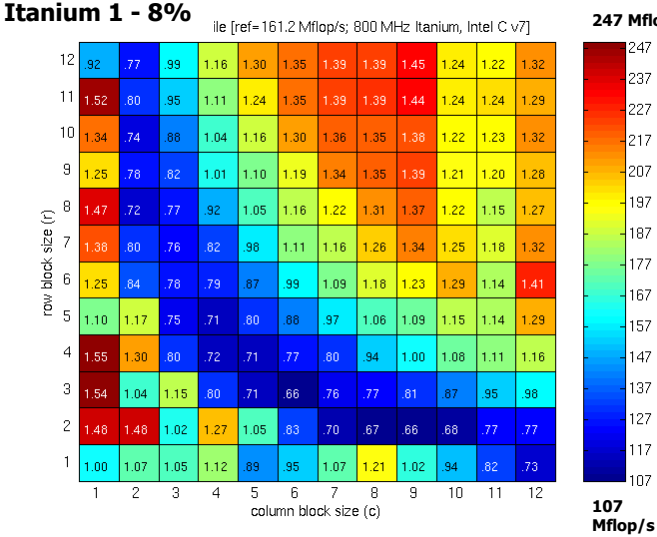
Register Profile: dense matrix in sparse format

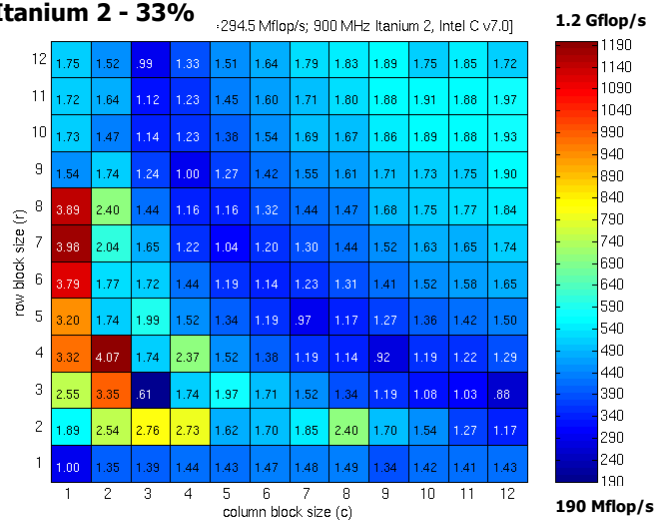


Register Profiles

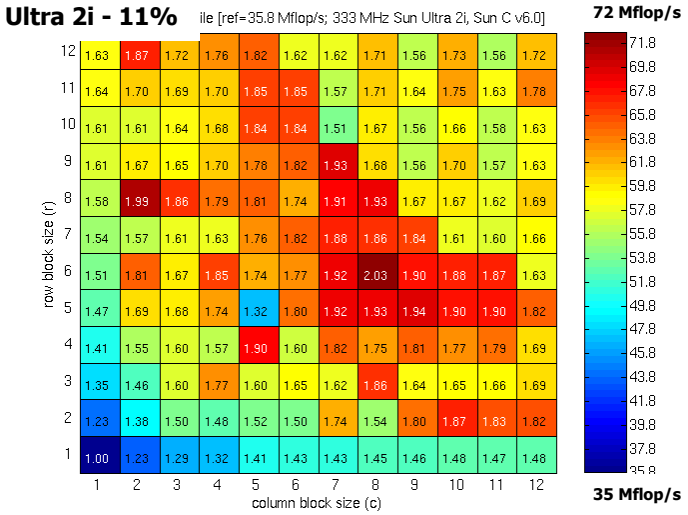


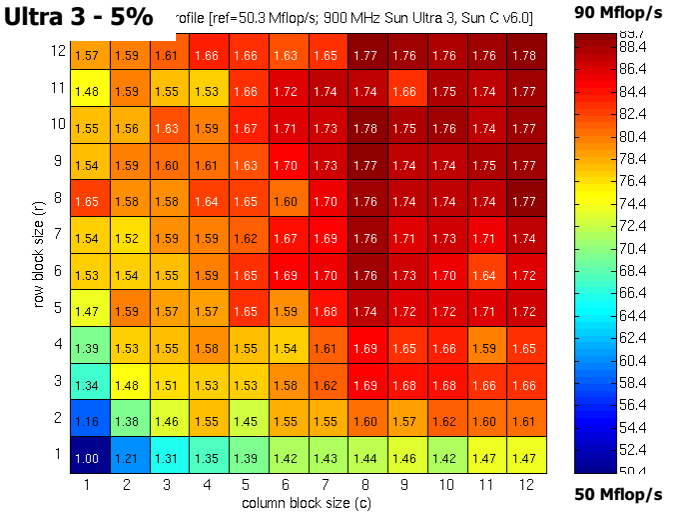


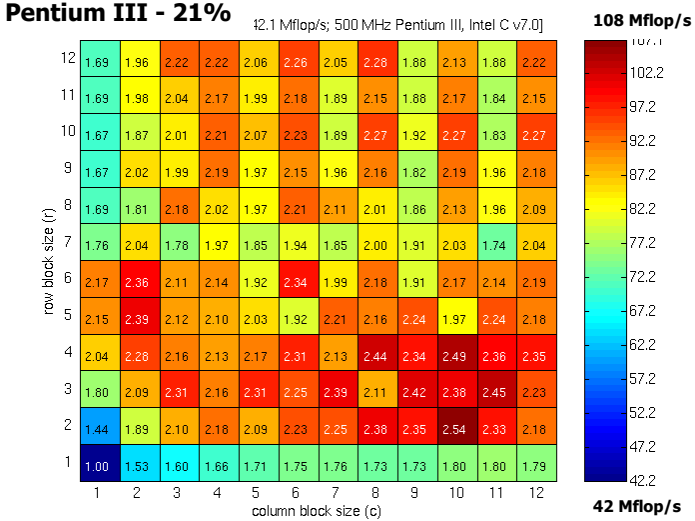


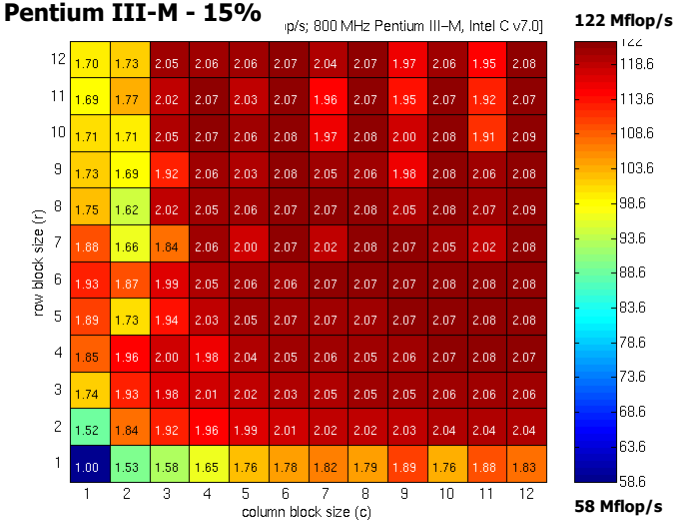


Register Profiles









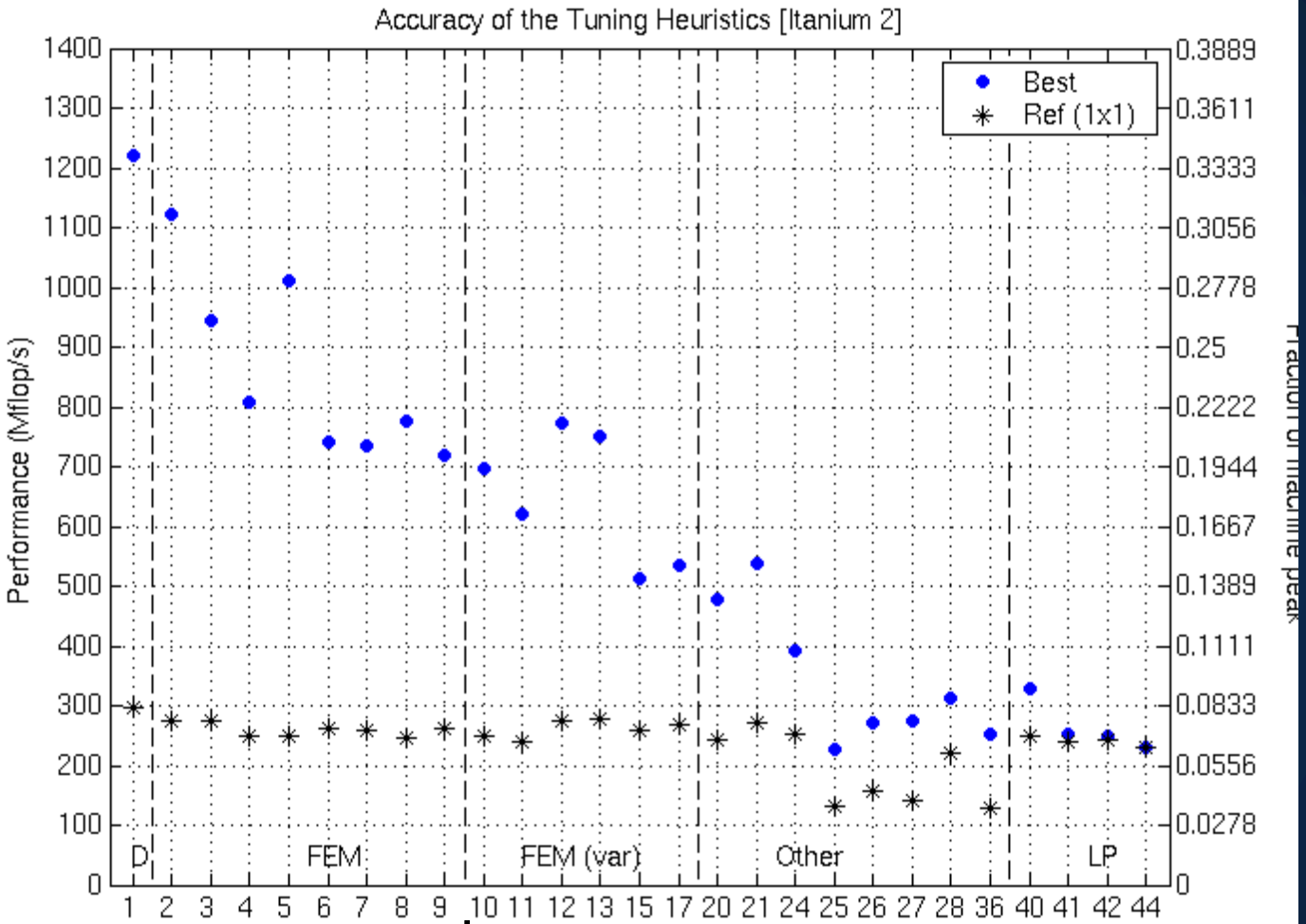
Adaptive Fill Estimation

- Idea: Sample matrix
  - Fraction of matrix to sample: $s \in [0,1]$
  - Cost ~ O(s * nnz)
  - Control cost by controlling s
    - Search at run-time: the constant matters!
- Control s automatically by computing statistical confidence intervals
  - Idea: Monitor variance
- Cost of tuning
  - Lower bound: convert matrix in 5 to 40 unblocked SpMVs
  - Heuristic: 1 to 11 SpMVs

Machine Learning in Automatic Performance Tuning

- **Statistical Models for Empirical Search-Based Performance Tuning** (*International Journal of High Performance Computing Applications*, 18 (1), pp. 65-94, February 2004) Richard Vuduc, J. Demmel, and Jeff A. Bilmes.
- **Predicting and Optimizing System Utilization and Performance via Statistical Machine Learning** (Computer Science PhD Thesis, University of California, Berkeley. UCB//EECS-2009-181) Archana Ganapathi
- **Machine Learning for Predictive Autotuning with Boosted Regression Trees,** (*Innovative Parallel Computing, 2012*) J. Bergstra et al.
- **Practical Bayesian Optimization of Machine Learning Algorithms,** (*NIPS 2012*) J. Snoek et al
- **OpenTuner: An Extensible Framework for Program Autotuning**, (*dspace.mit.edu/handle/1721.1/81958*) S. Amarasinghe et al

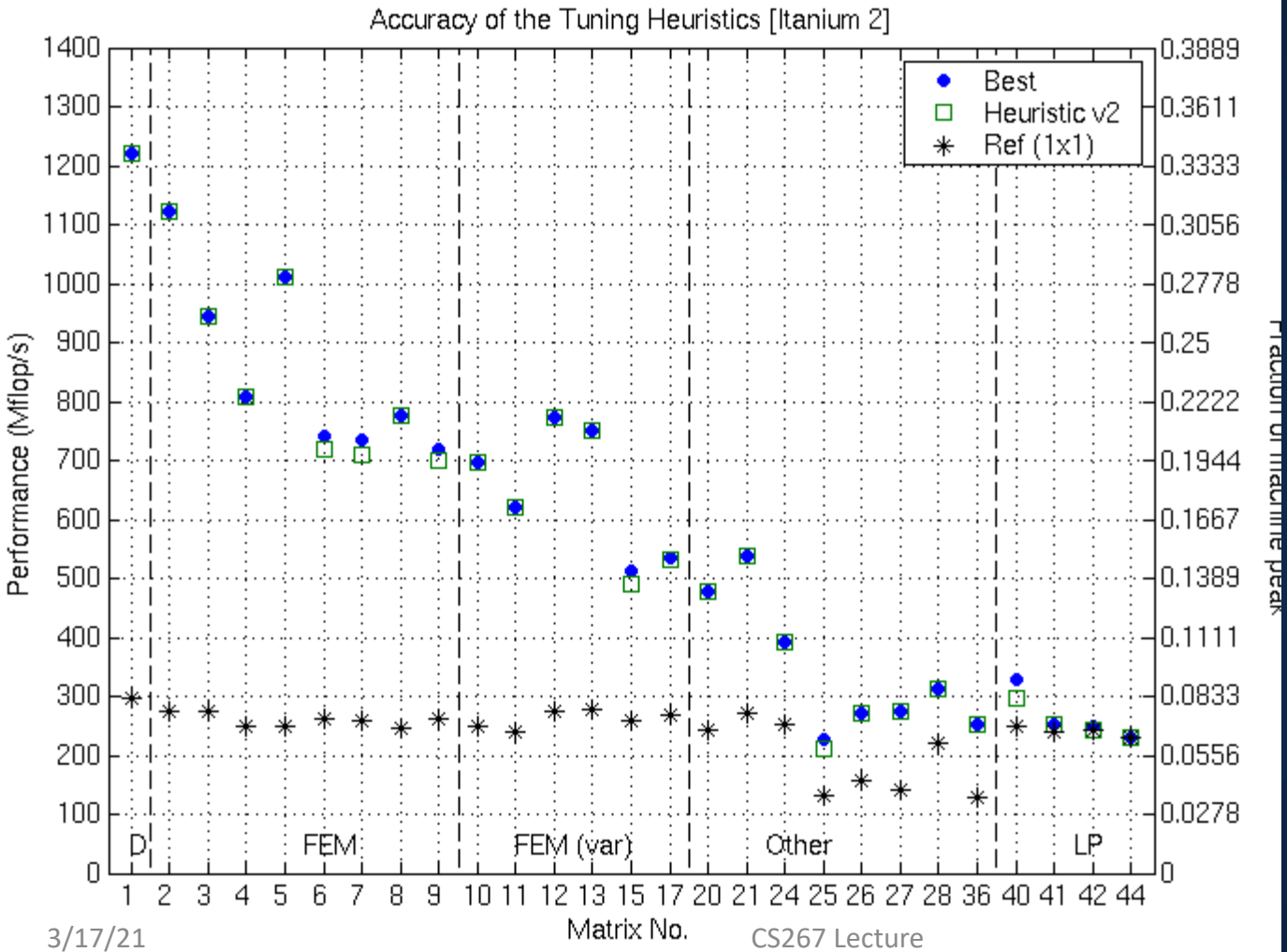
Tuning: Exhaustive



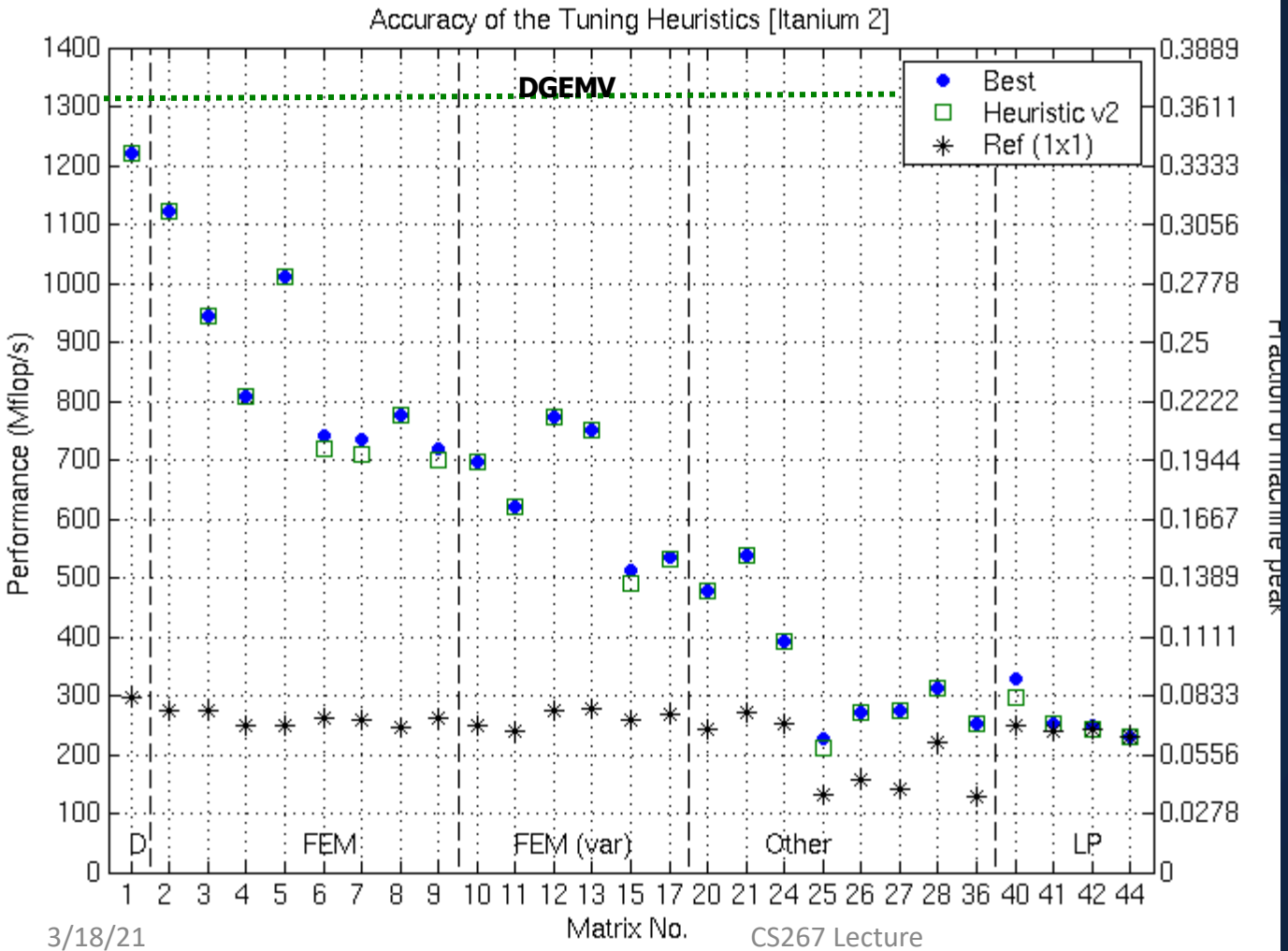
See p. 375 of Vuduc's thesis for matrices

NOTE: "Fair" flops used (ops on explicit zeros not counted as "work")

Tuning: Heuristic



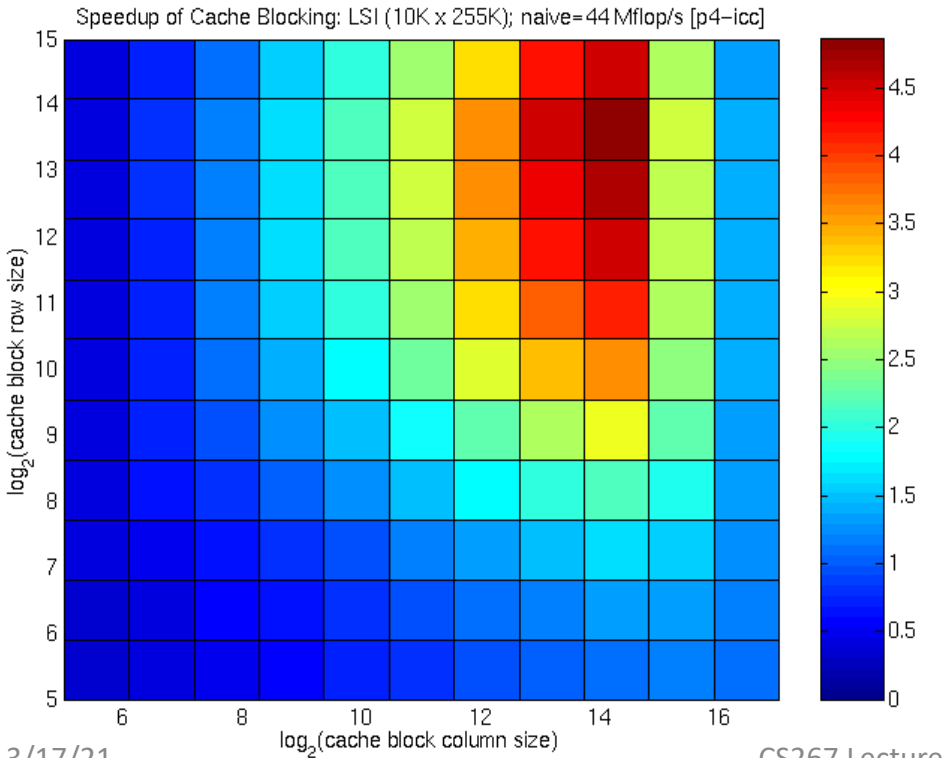
Tuning: Upper Bound



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

Cache Blocking on LSI Matrix



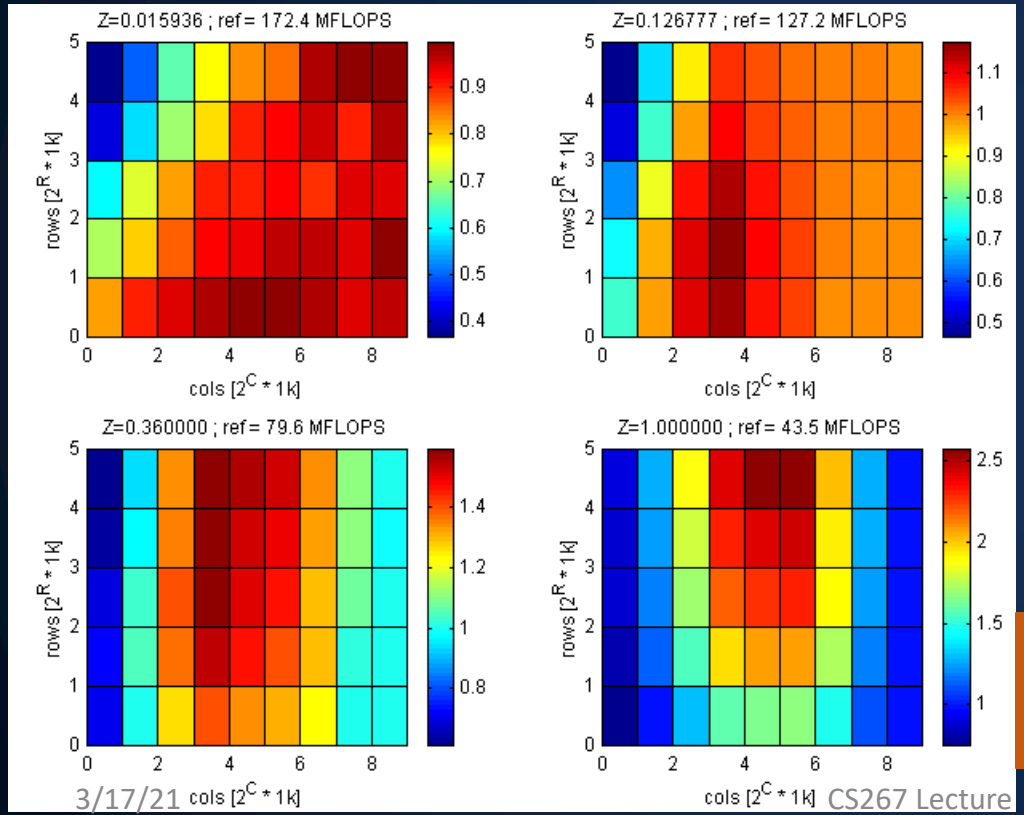
A
10k x 255k
3.7M non-zeros

Baseline:
44 Mflop/s

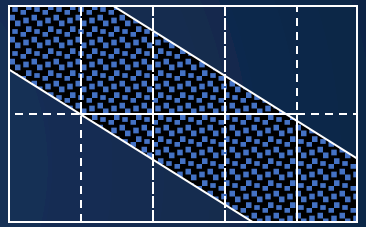
Best block size & performance:
16k x 16k
210 Mflop/s

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

Cache Blocking on Random Matrices



Speedup on four banded random matrices.



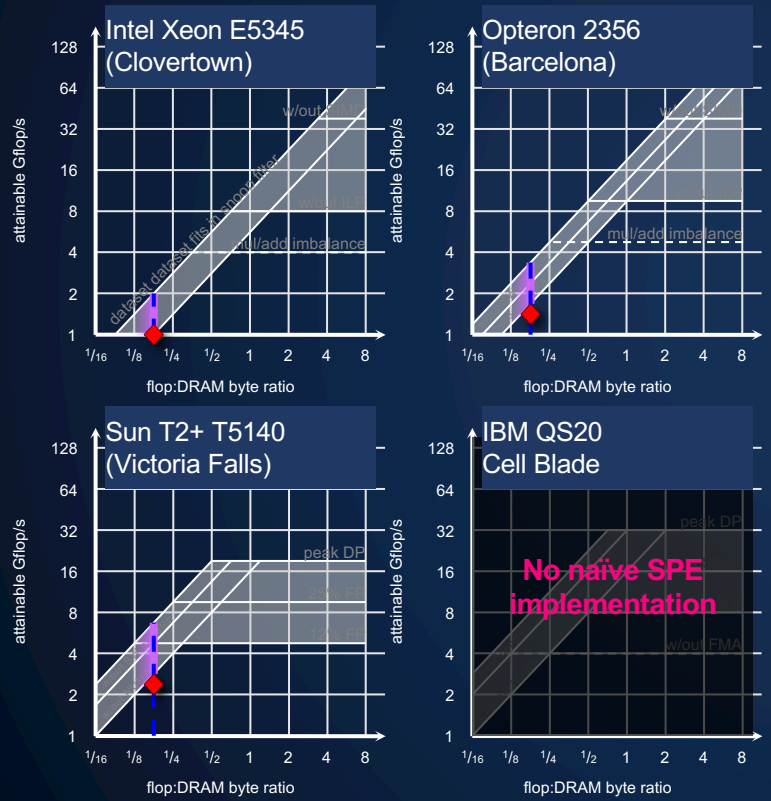
Nishtala, et al (2007). When cache blocking of sparse matrix vector multiply works and why.

Summary of Sequential Performance Optimizations

- Optimizations for SpMV
  - **Register blocking (RB)**: up to **4x** over CSR
  - **Variable block splitting**: **2.1x** over CSR, 1.8x over RB
  - **Diagonals**: **2x** over CSR
  - **Reordering** to create dense structure + **splitting**: **2x** over CSR
  - **Symmetry**: **2.8x** over CSR, 2.6x over RB
  - **Cache blocking**: **2.8x** over CSR
  - **Multiple vectors (SpMM)**: **7x** over CSR
  - And combinations…
- Sparse triangular solve
  - Hybrid sparse/dense data structure: **1.8x** over CSR
- Higher-level kernels
  - $\mathbf{A \cdot A^T \cdot x}$, $\mathbf{A^T \cdot A \cdot x}$: **4x** over CSR, 1.8x over RB
  - $\mathbf{A^2 \cdot x}$: **2x** over CSR, 1.5x over RB
  - $\mathbf{[A \cdot x, A^2 \cdot x, A^3 \cdot x, .. , A^k \cdot x]}$

Roofline model for SpMV

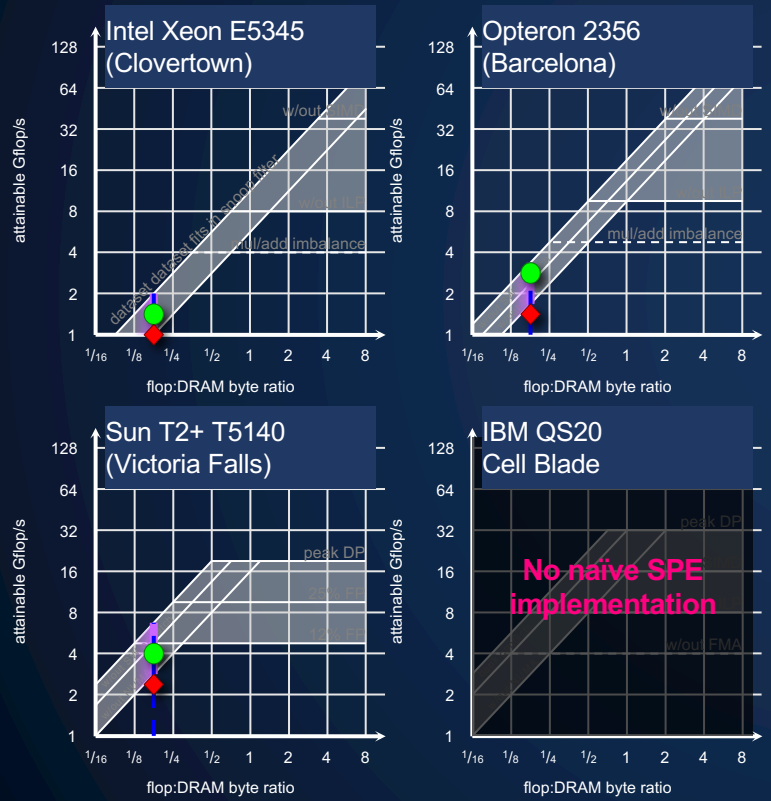
(out-of-the-box parallel)



- Two unit stride streams
- Inherent FMA
- No ILP
- No DLP
- FP is 12-25%
- Naïve compulsory: flop:byte < 0.166
- For simplicity: dense matrix in sparse format

Roofline model for SpMV

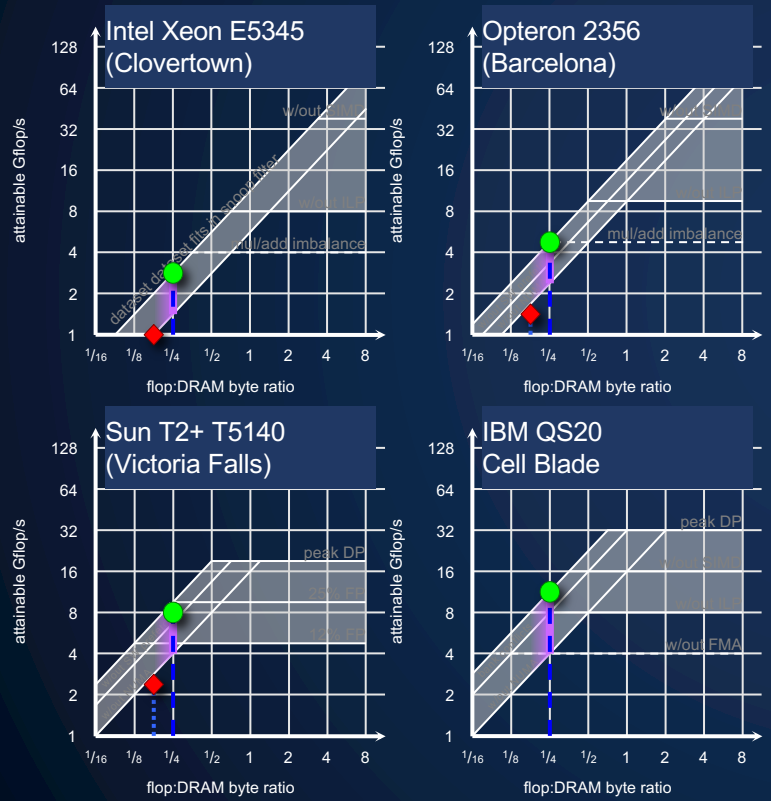
(NUMA & SW prefetch)



- compulsory flop:byte ~ 0.166
- utilize all memory channels
- Add software prefetching

Roofline model for SpMV

(matrix compression)

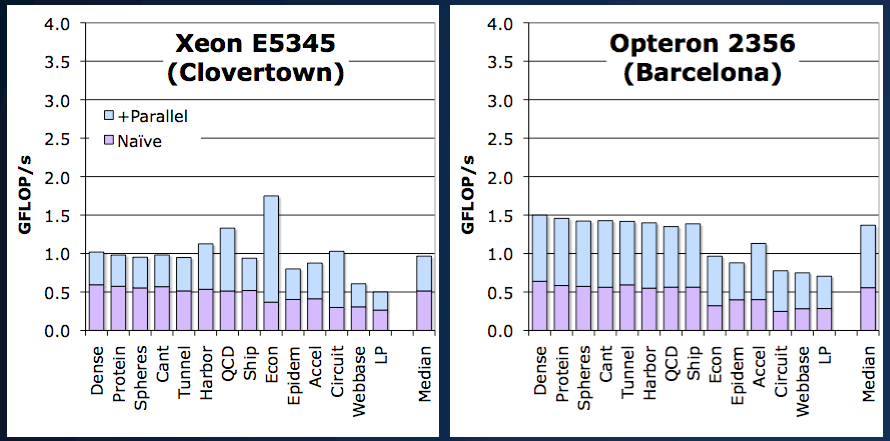


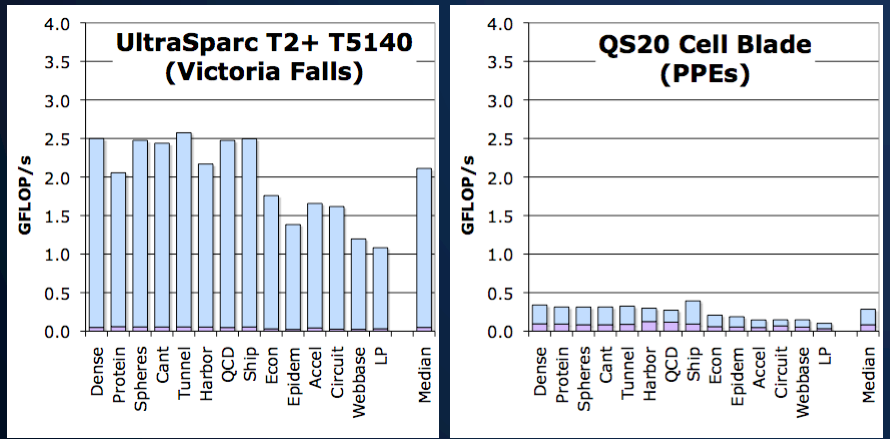
- Inherent FMA
- Register blocking improves ILP, DLP, flop:byte ratio, and FP% of instructions

- Other forms of matrix compression may also help
  - 16-bit indices within blocks
  - Patterns of repeated nonzeros

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?**

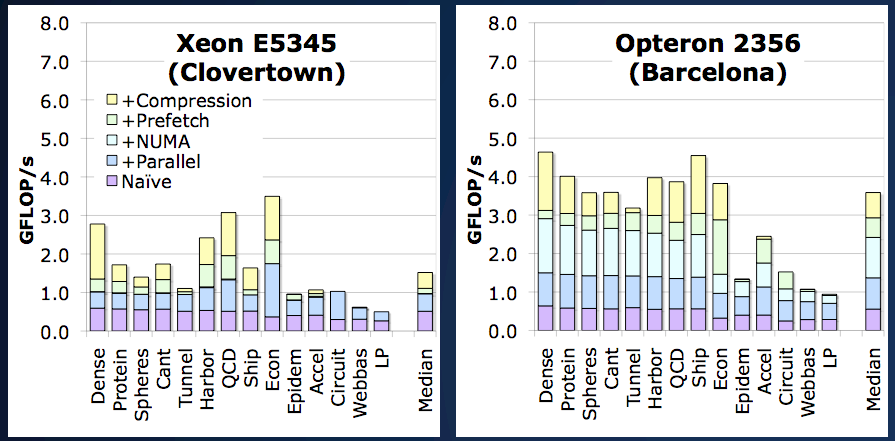
Naïve Pthreads

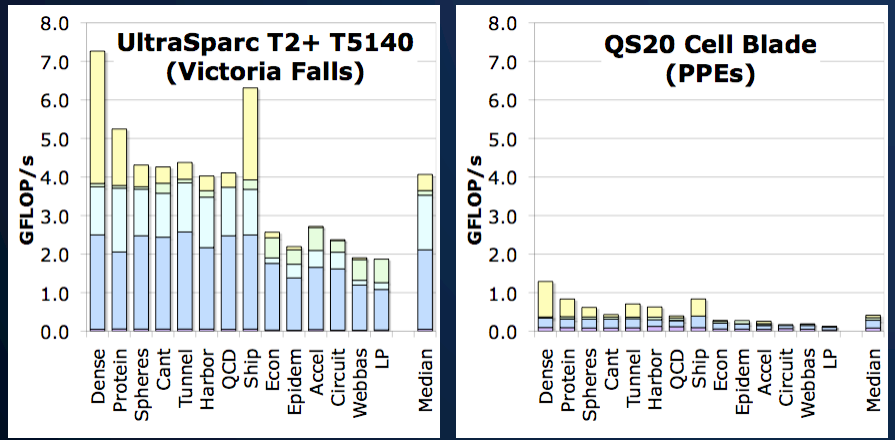
Naïve

Source: Sam Williams

SpMV Performance

(Matrix Compression)



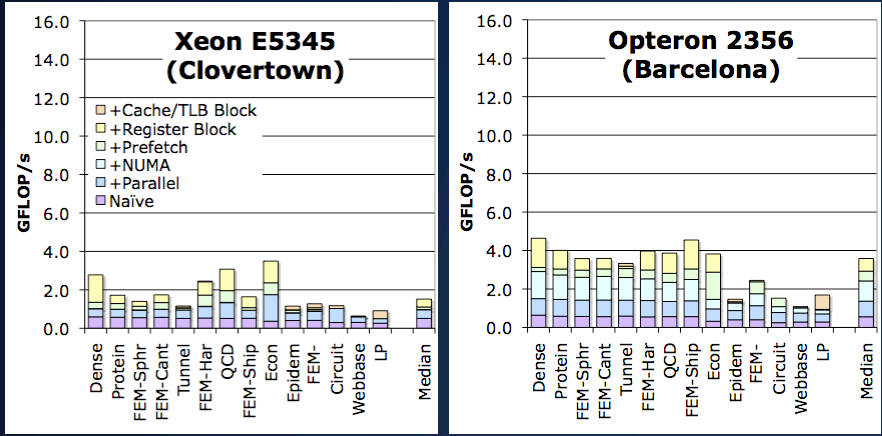


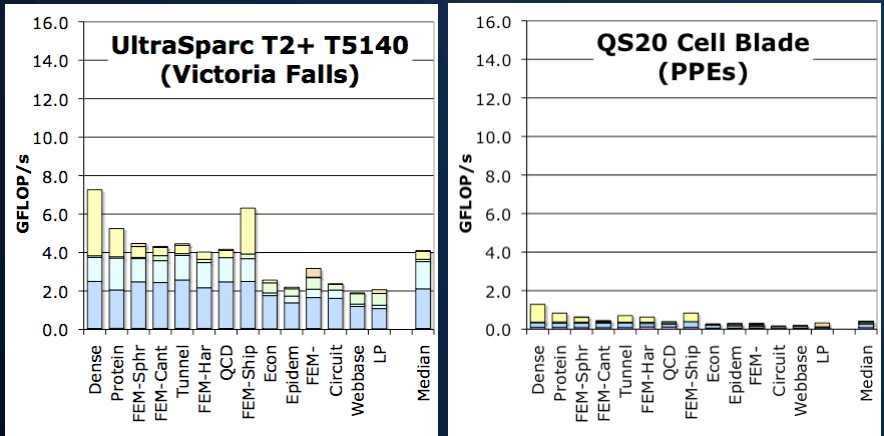
- After maximizing memory bandwidth, the only hope is to minimize memory traffic.
- Compression: exploit
  - register blocking
  - other formats
  - smaller indices
- Use a traffic minimization heuristic rather than search
- Benefit is matrix-dependent.
- Register blocking enables efficient software prefetching (one per cache line)

Source: Sam Williams

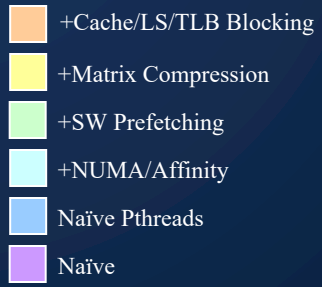
Auto-tuned SpMV Performance

(cache and TLB blocking)





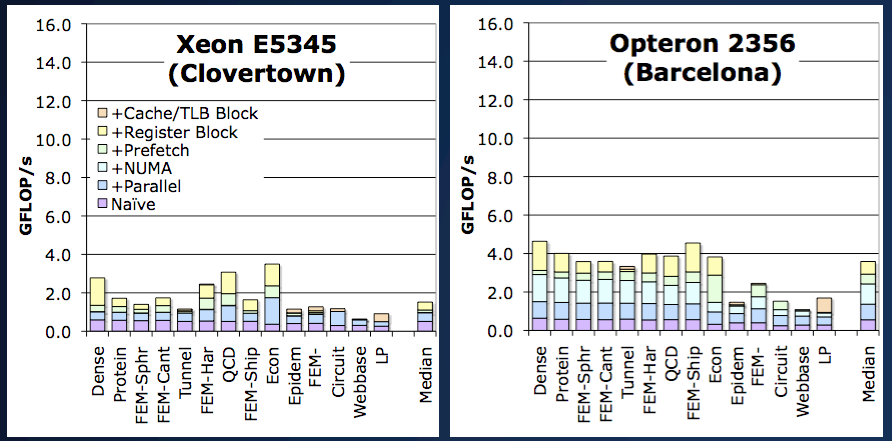
- Fully auto-tuned SpMV performance across the suite of matrices
- Why do some optimizations work better on some architectures?
- **matrices with naturally small working sets**
- **architectures with giant caches**

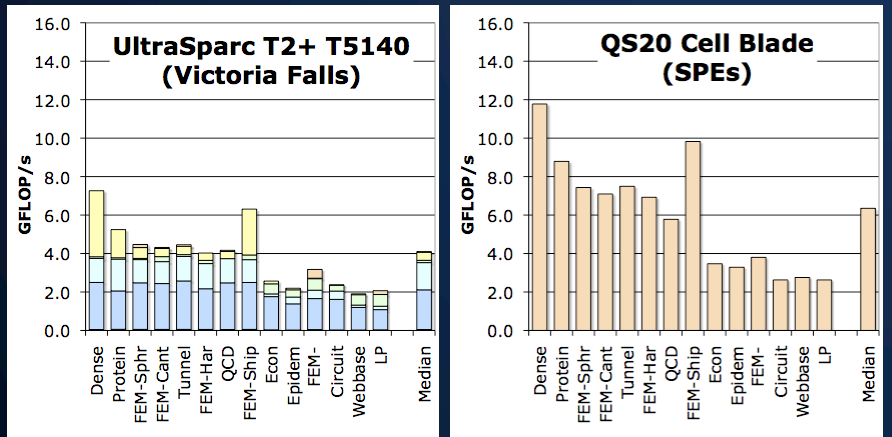


Source: Sam Williams

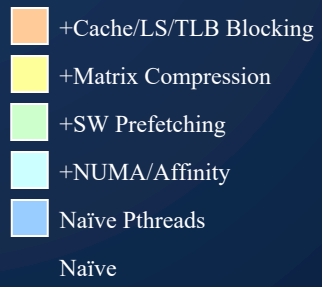
Auto-tuned SpMV Performance

(architecture specific optimizations)





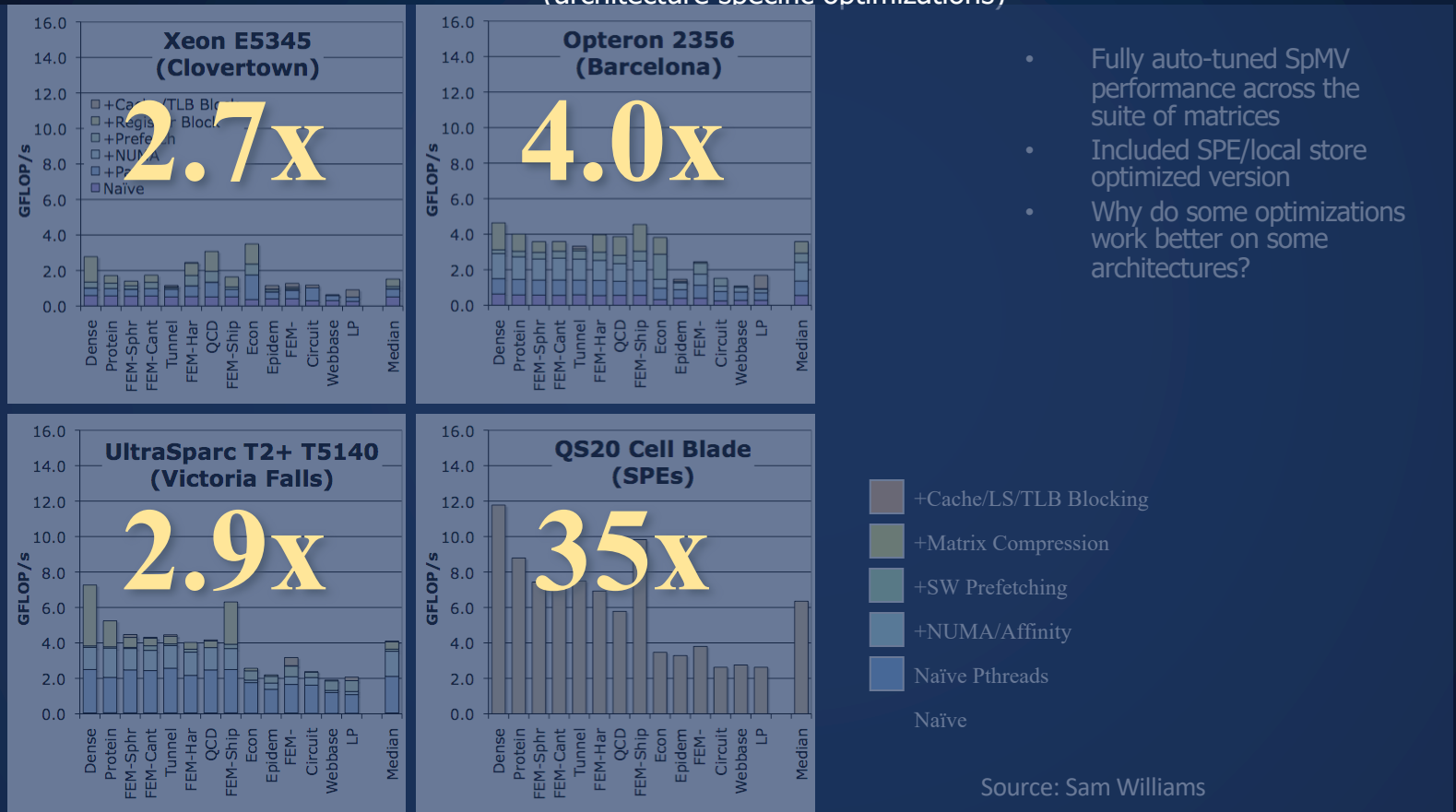
- Fully auto-tuned SpMV performance across the suite of matrices
- Included SPE/local store optimized version
- Why do some optimizations work better on some architectures?



Source: Sam Williams

Auto-tuned SpMV Performance

(architecture specific optimizations)



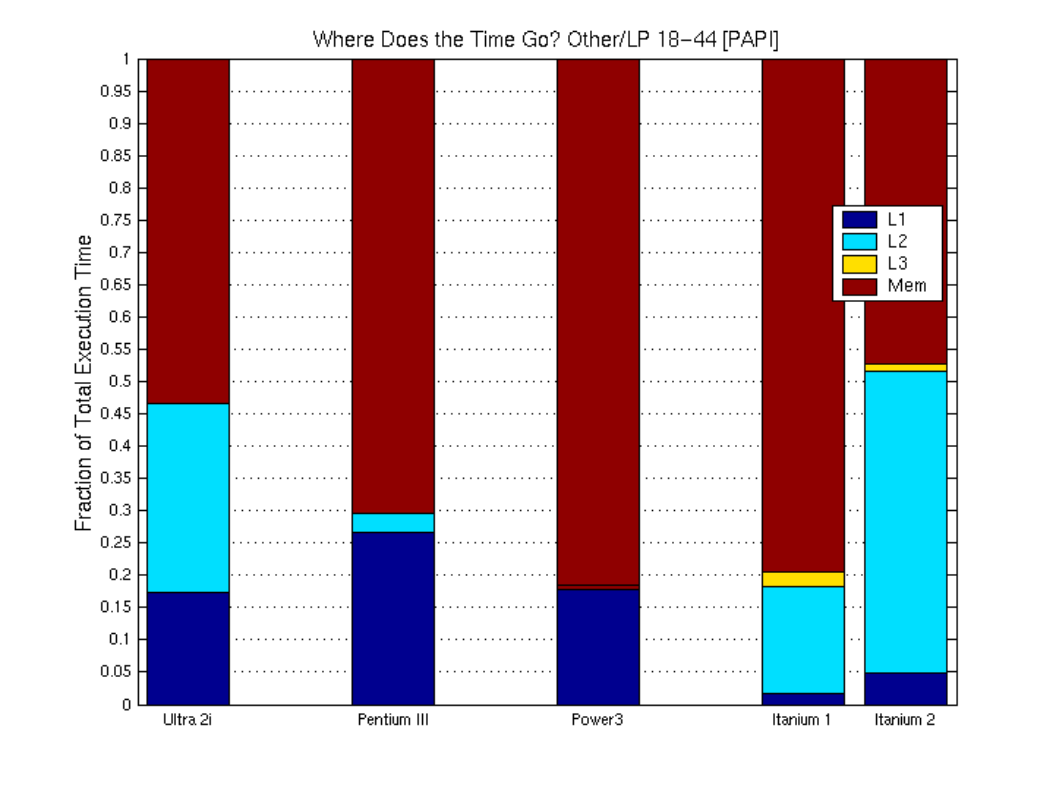
- Fully auto-tuned SpMV performance across the suite of matrices
- Included SPE/local store optimized version
- Why do some optimizations work better on some architectures?

Source: Sam Williams

Outline for today

- Sparse matrices in the world
- Sparse matrix formats and serial SpMV
- Parallel and distributed SpMV
- Register / cache blocking and autotuning SpMV
- CA iterative solvers
- Sparse matmul (SpGEMM, SPMM,...)

Execution Time Breakdown in SpMV



Matrix 40
(PAPI counters)

Is tuning SpMV all we can do?

- Iterative methods all depend on it
- But speedups are limited
  - Just 2 flops per nonzero
  - Communication costs dominate
- Can we beat this bottleneck?
- Need to look at next level in stack:
  - What do algorithms that use SpMV do?
  - Can we reorganize them to avoid communication?
- Only way significant speedups will be possible

Tuning Higher Level Algorithms

- We almost always do many SpMVs, not just one
  - "Krylov Subspace Methods" (KSMs) for Ax=b, Ax = λx
    - Conjugate Gradients, GMRES, Lanczos, …
  - Do a sequence of k SpMVs to get vectors $[x_1, \ldots, x_k]$
  - Find best solution x as linear combination of $[x_1, \ldots, x_k]$

Main cost is k SpMVs, dominated by data movement

Tuning Higher Level Algorithms

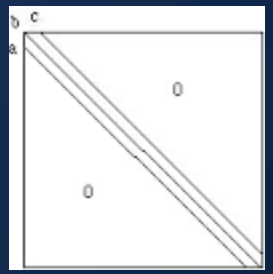
- We almost always do many SpMVs, not just one
  - "Krylov Subspace Methods" (KSMs) for Ax=b, Ax = λx
    - Conjugate Gradients, GMRES, Lanczos, …
  - Do a sequence of k SpMVs to get vectors $[x_1, \ldots, x_k]$
  - Find best solution x as linear combination of $[x_1, \ldots, x_k]$
- Goal: make communication cost independent of k
  - Parallel case: O(log P) messages, not O(k log P) - optimal
    - same bandwidth as before
  - Sequential case: O(1) messages and bandwidth, not O(k) - optimal
- Achievable when matrix partitionable with low surface-to-volume ratio

Main cost is k SpMVs, dominated by data movement

Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]

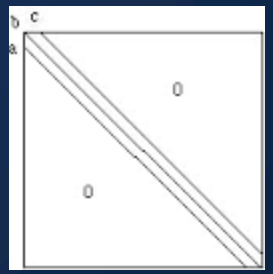
- Example: A tridiagonal, n=32, k=3



Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]

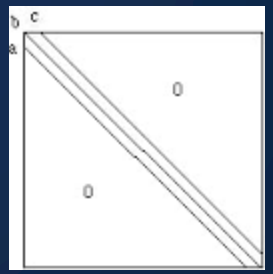
- Example: A tridiagonal, n=32, k=3



Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]

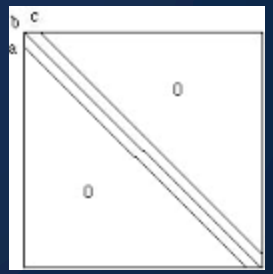
- Example: A tridiagonal, n=32, k=3



Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

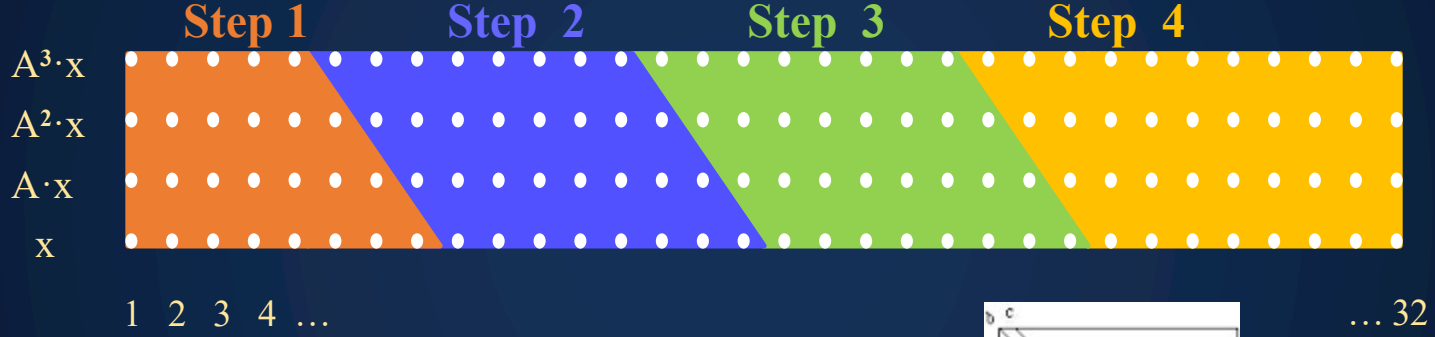
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]

- Example: A tridiagonal, n=32, k=3

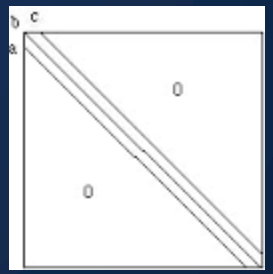


Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]

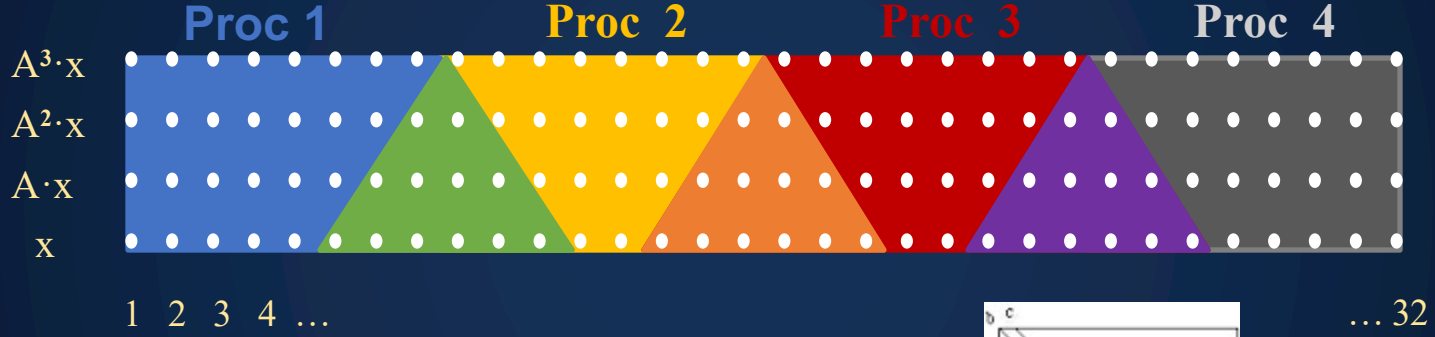


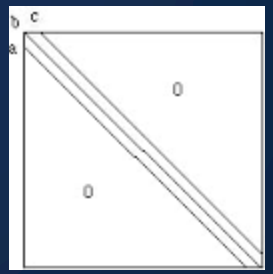
- Example: A tridiagonal, n=32, k=3



Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



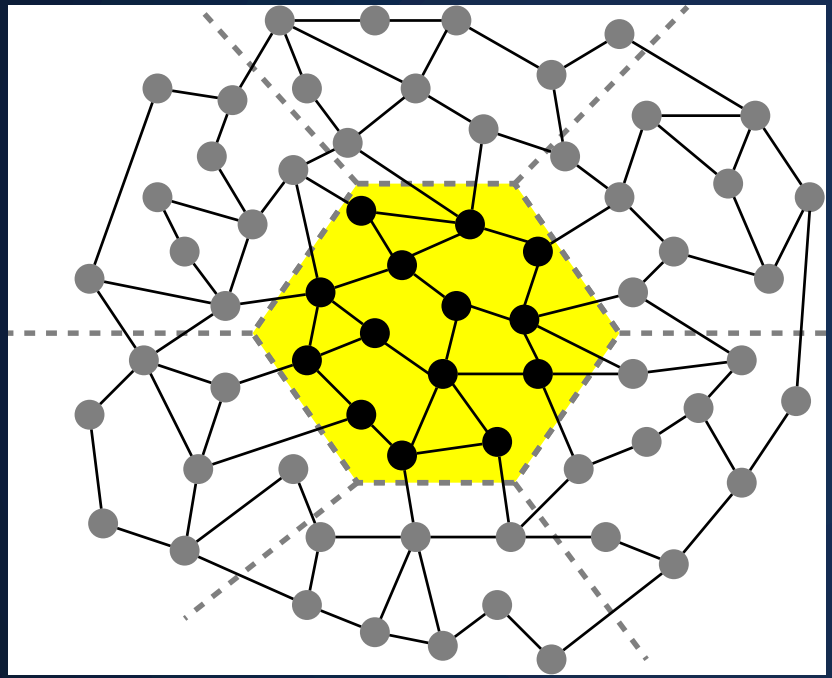


- Example: A tridiagonal, n=32, k=3

Parallel Algorithm:

- Each processor works on an overlapping trapezoid

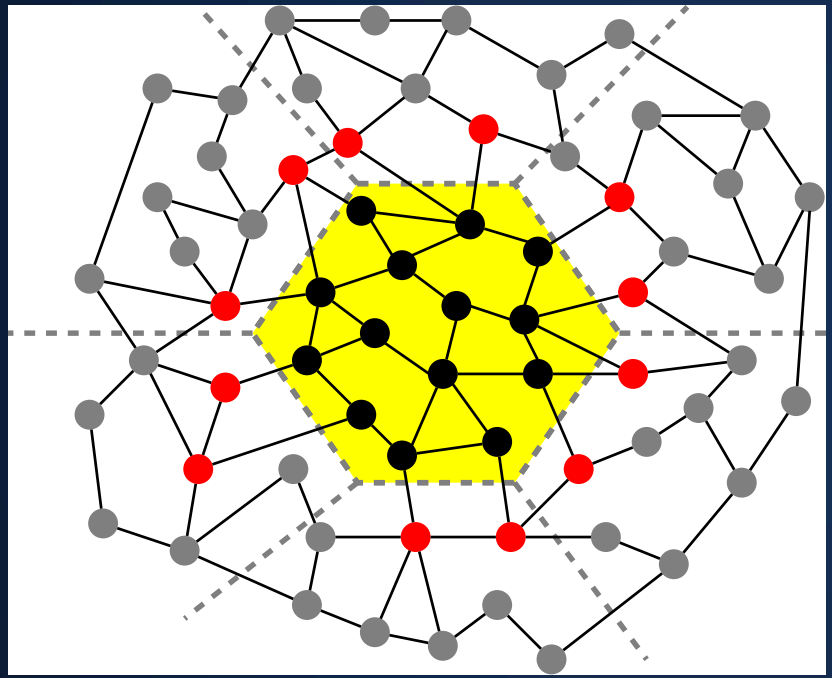
Matrix Powers Kernel on a General Matrix



Each processor (cache) starts with a localized set of points

See paper by Jim Demmel, Mark Hoemman, Marghoob Mohiyuddin, Kathy Yelick

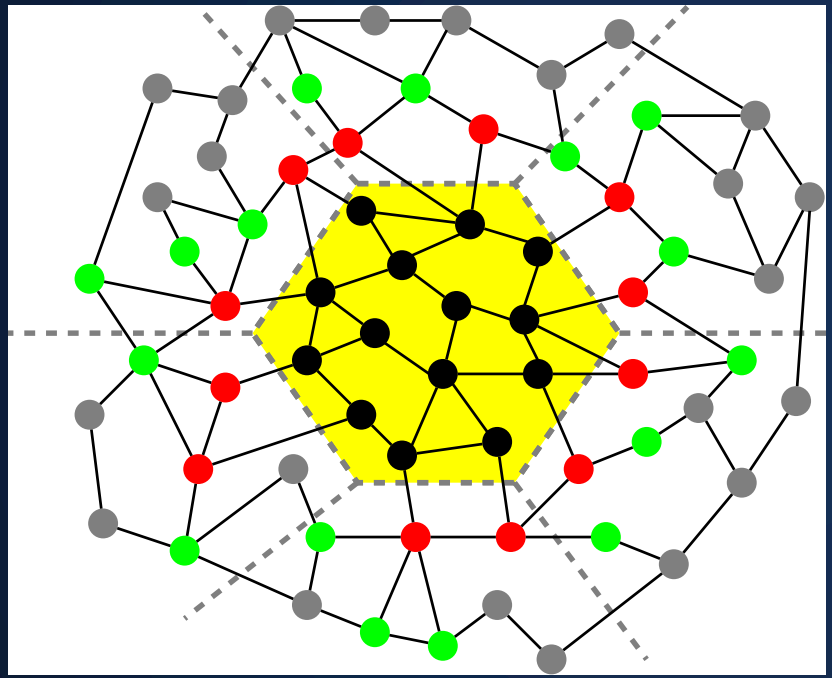
Matrix Powers Kernel on a General Matrix



Need nearest neighbors for Ax

See paper by Jim Demmel, Mark Hoemman, Marghoob Mohiyuddin, Kathy Yelick

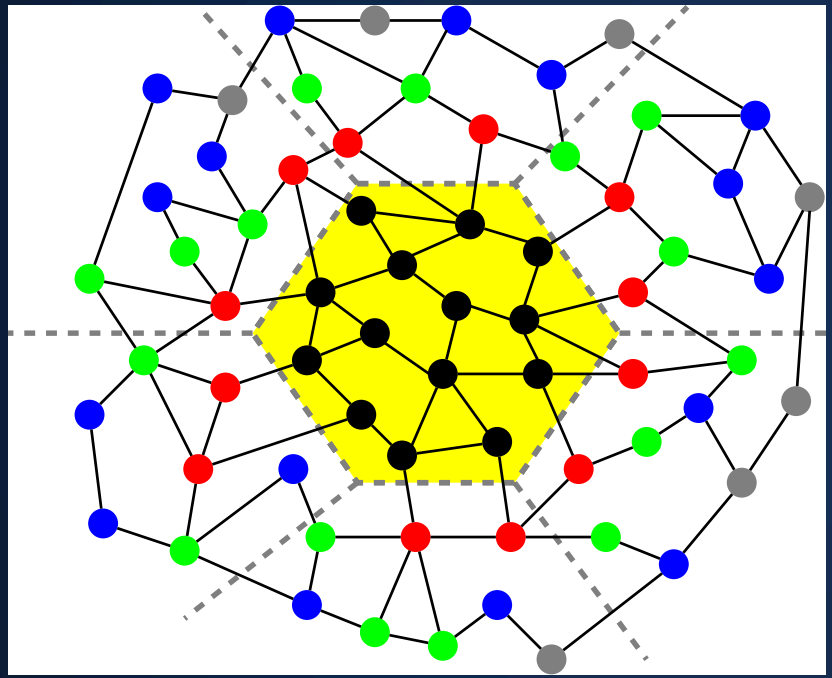
Matrix Powers Kernel on a General Matrix



And neighbor's neighbors for A^2x

See paper by Jim Demmel, Mark Hoemman, Marghoob Mohiyuddin, Kathy Yelick

Matrix Powers Kernel on a General Matrix

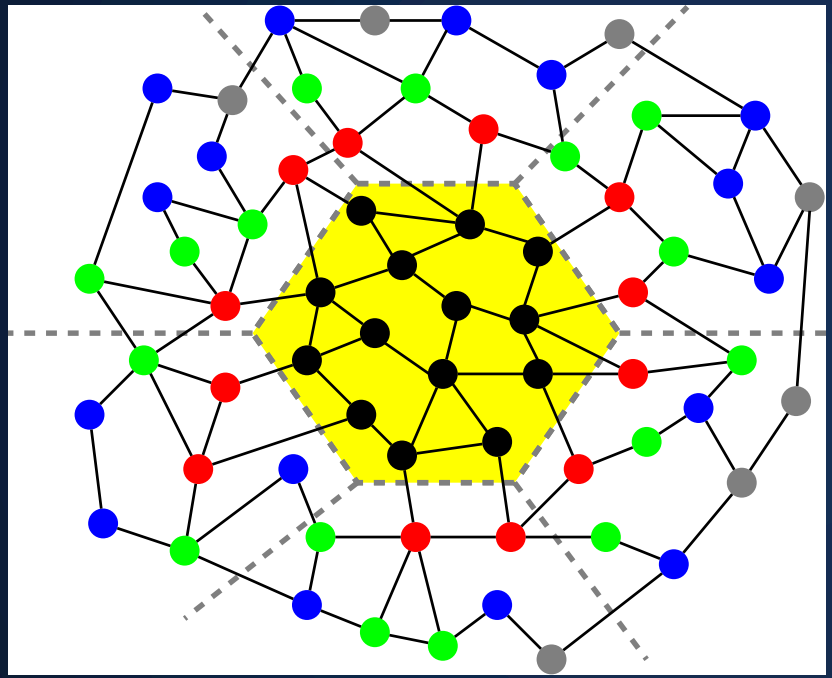


Saves communication for "well partitioned" matrices

- Serial memory bandwidth: O(1) moves of data moves vs. O(k)
- Parallel message latency: O(log p) messages vs. O(k log p)

See paper by Jim Demmel, Mark Hoemman, Marghoob Mohiyuddin, Kathy Yelick

Matrix Powers Kernel on a General Matrix



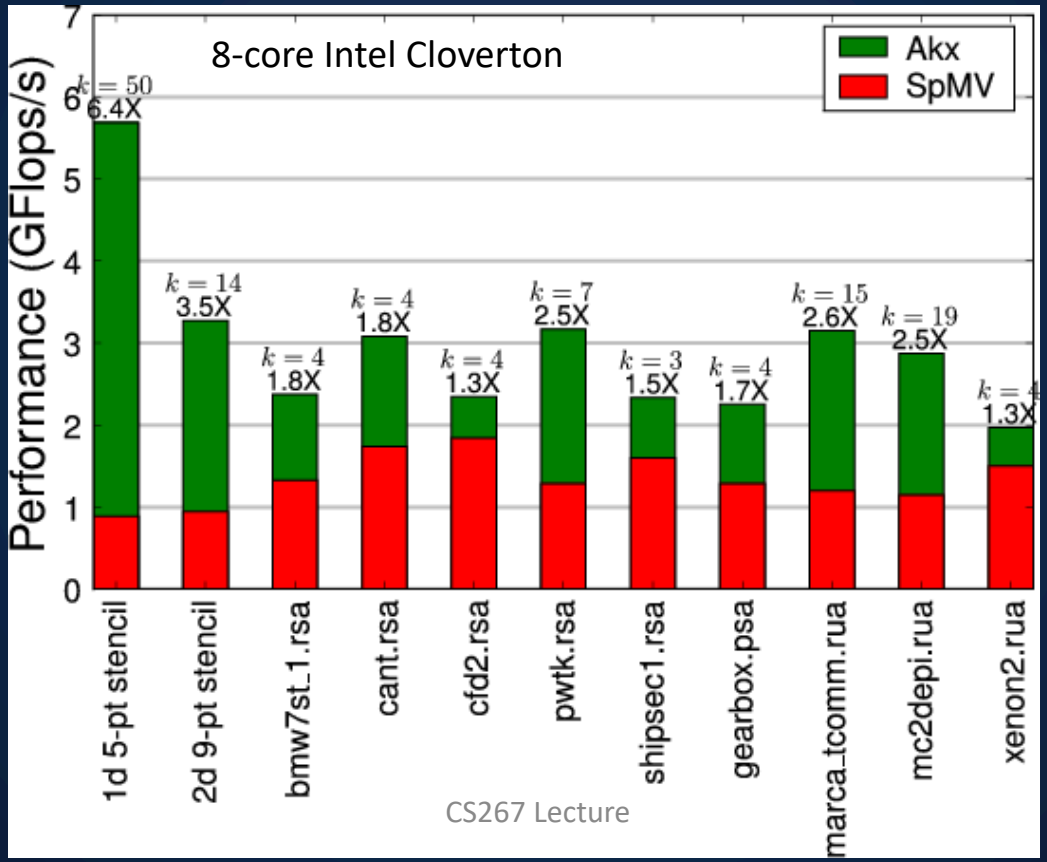
Saves communication for "well partitioned" matrices

- Serial memory bandwidth: O(1) moves of data moves vs. O(k)
- Parallel message latency: O(log p) messages vs. O(k log p)

- *Uses hypergraph partitioning*
- *For implicit memory management (caches) uses a TSP algorithm for layout*

See paper by Jim Demmel, Mark Hoemman, Marghoob Mohiyuddin, Kathy Yelick

Multicore Speedups



Performance Results

- Measured Multicore (Clovertown) speedups up to 6.4x
- Measured/Modeled sequential OOC speedup up to 3x
- Modeled parallel Petascale speedup up to 6.9x
- Modeled parallel Grid speedup up to 22x

- Sequential speedup due to bandwidth, works for many problem sizes
- Parallel speedup due to latency, works for smaller problems on many processors
- Multicore results used both techniques

Avoiding Communication in Iterative Linear Algebra

- Many "Krylov Subspace Methods" based on k SpMVs
  - Conjugate Gradient, GMRES, Lanczos, Arnoldi, …
- To minimize communication in Krylov Subspace Methods
  - Assume matrix "well-partitioned," modest surface-to-volume ratio
  - Parallel implementation
    - Conventional: O(k log p) messages, because k calls to SpMV
    - **New: O(log p) messages - optimal**
  - Serial implementation
    - Conventional: O(k) moves of data from slow to fast memory
    - **New**: **O(1) moves of data – optimal**
- Lots of speed up possible (modeled and measured)
  - Price: some redundant computation
- Prior work on CA iterraive methods
  - Theses of Mark Hoemmen, Erin Carson, other papers at bebop.cs.berkeley.edu

Minimizing Communication of GMRES to solve Ax=b

- GMRES: find x in span{b,Ab,…,A^kb} minimizing || Ax-b ||$_2$
- Cost of k steps of standard GMRES vs new GMRES

Standard GMRES

```
for i=1 to k
  w = A · v(i-1)
  MGS(w, v(0),…,v(i-1))
  update v(i), H
endfor
solve LSQ problem with H
```

Sequential: #words_moved =
O(k·nnz) from SpMV
+ O(k^2·n) from MGS
Parallel: #messages =
O(k) from SpMV
+ O(k^2 · log p) from MGS

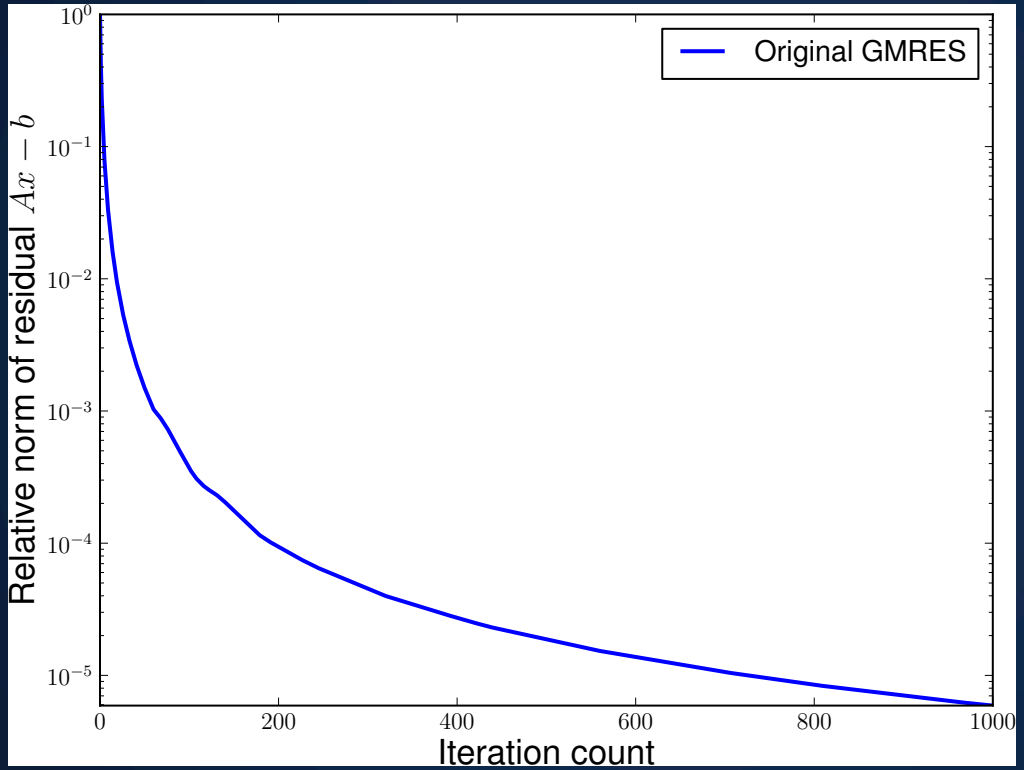
Communication-avoiding GMRES

```
W = [ v, Av, A²v, … , Aᵏv ]
[Q,R] = TSQR(W)  …  "Tall Skinny QR"
Build H from R, solve LSQ problem
```

Sequential: #words_moved =
O(nnz) from SpMV
+ O(k·n) from TSQR
Parallel: #messages =
O(1) from computing W
+ O(log p) from TSQR

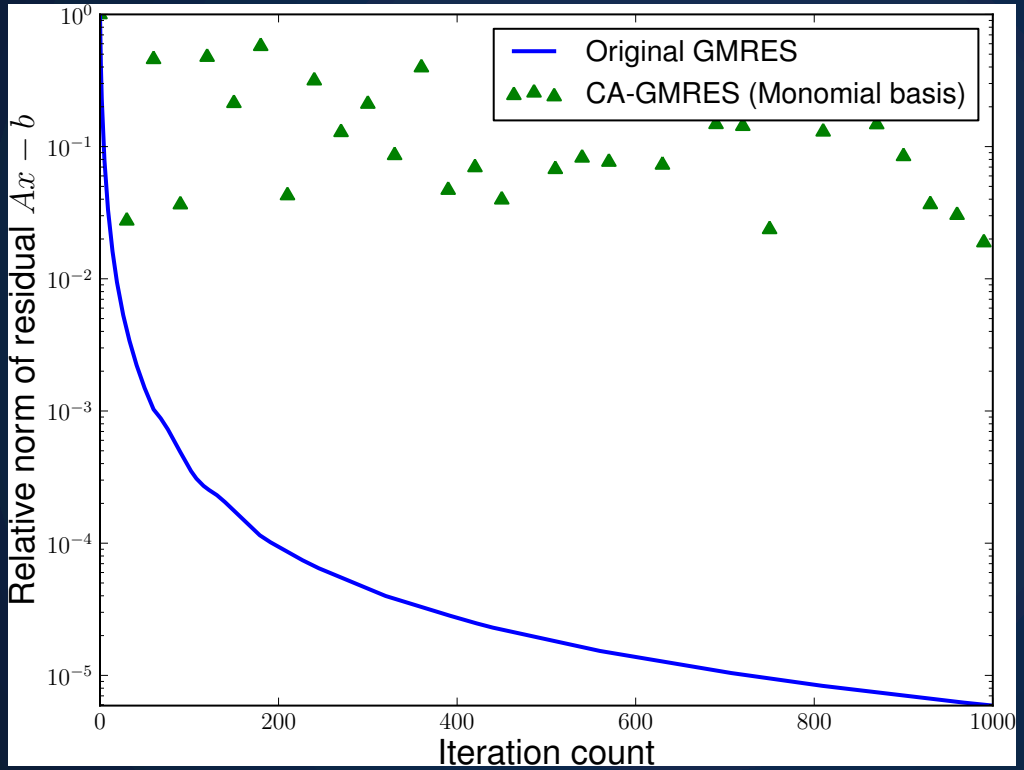
In exact arithmetic, these two algorithms are identical, but what about floating point?

Matrix Powers Kernel (and TSQR) in an iterative solver (GMRES)



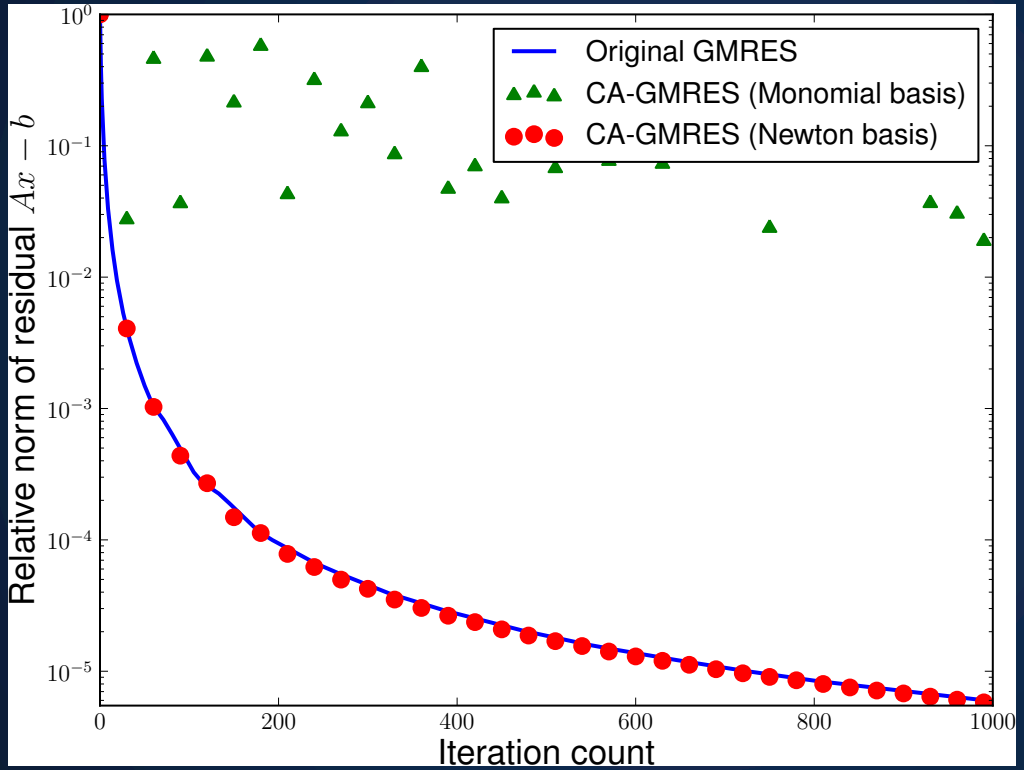
Jim Demmel, Mark Hoemmen, Marghoob Mohiyuddin, Kathy Yelick

Matrix Powers Kernel (and TSQR) in an iterative solver (GMRES)



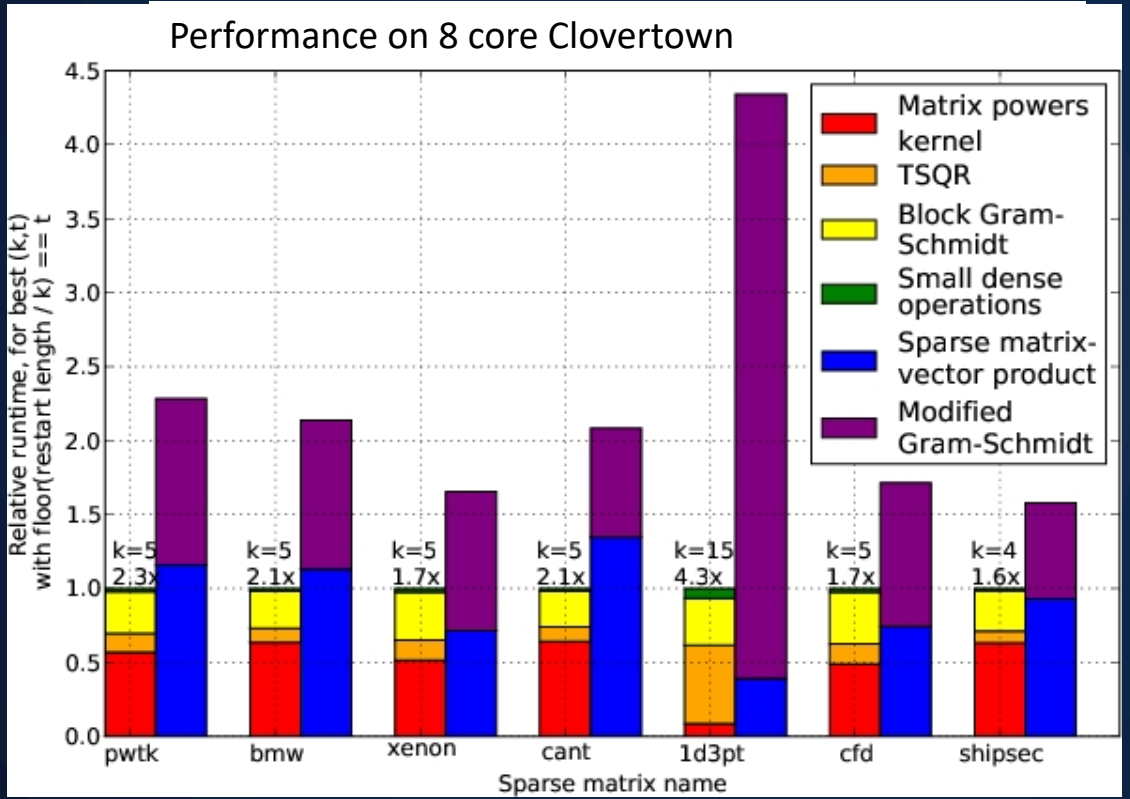
Jim Demmel, Mark Hoemmen, Marghoob Mohiyuddin, Kathy Yelick

Matrix Powers Kernel (and TSQR) in an iterative solver (GMRES)



Jim Demmel, Mark Hoemmen, Marghoob Mohiyuddin, Kathy Yelick

Communication-Avoiding Krylov Method (GMRES)



Takeaway messages

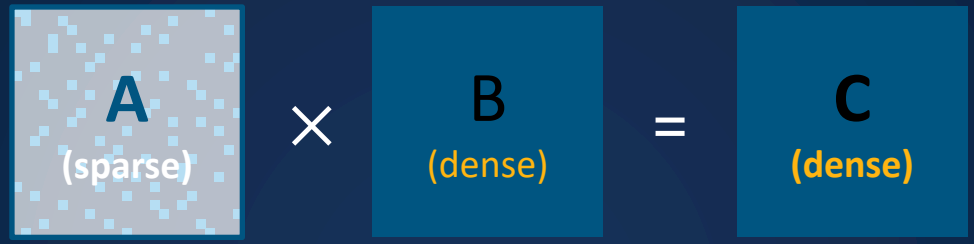
- Tuning for sparse matrices: harder than dense ones
- SpMV: benefits lower due to low Computational Intensity (read the matrix)
- Register blocking and other "compression" can be a big win
- Cache blocking less so; other low level tuning (e.g., prefetch) some
- For distribute memory, reordering (e.g.,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

Outline for today

- Sparse matrices in the world
- Sparse matrix formats and serial SpMV
- Parallel and distributed SpMV
- Register / cache blocking and autotuning SpMV
- CA iterative solvers
- Sparse matmul (SpGEMM, SPMM,...)

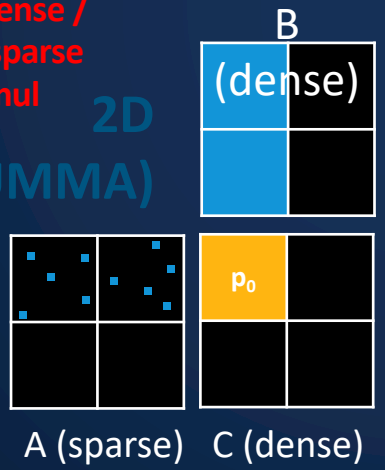
Sparse × Dense Matmul

These are not necessarily square

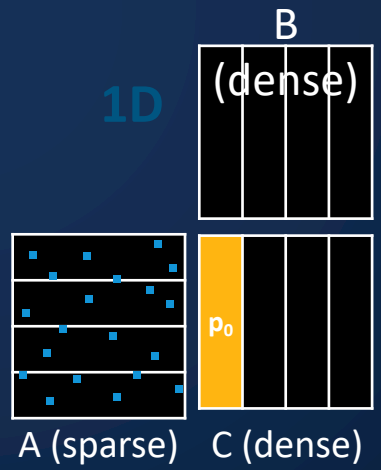


2D/2.5D/3D only optimal for dense-dense / sparse-sparse matmul

2D (SUMMA)



1D

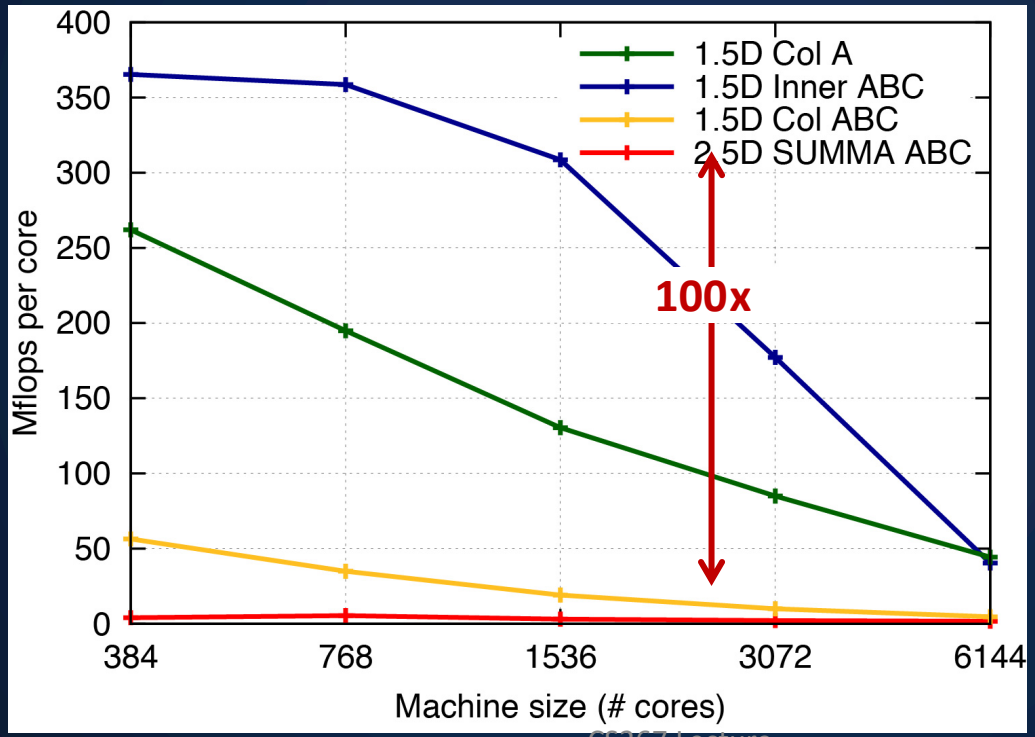


+ replication = 1.5D

[Koanantakool et al. 2016]

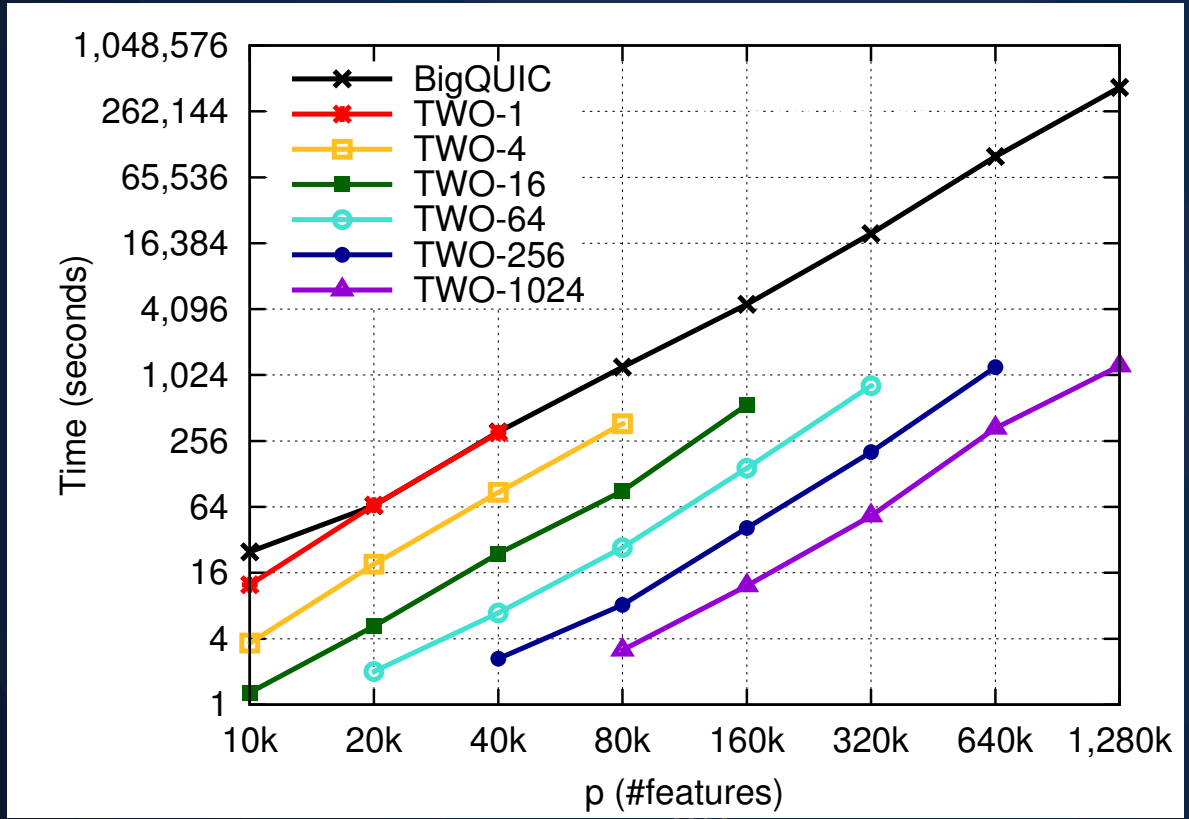
100x Improvement for the right algorithm

- $A^{66k \times 172k}$, $B^{172k \times 66k}$, 0.0038% nnz, Cray XC30



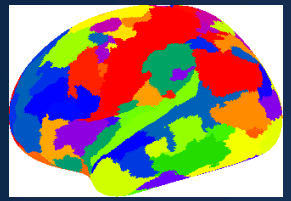
[Koanantakool et al. 2016]

Inverse Covariance Matrix Estimation (CONCORD) using fast SpDM^3 algorithm

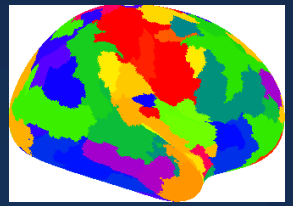


[Koanantakool et al. 2016]

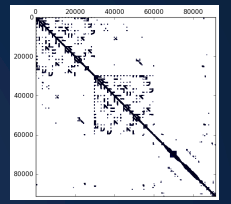
HP-CONCORD on Brain fMRI data



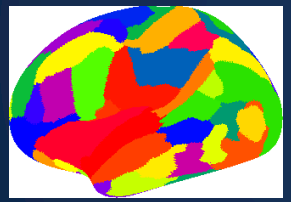
$\lambda_1 = 0.48,\ \lambda_2 = 0.39,\ \epsilon = 3,$
% of best score = 100



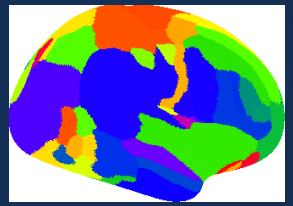
$\lambda_1 = 0.5,\ \lambda_2 = 0.39,\ \epsilon = 3,$
% of best score = 100



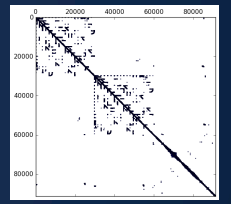
$\lambda_1 = 0.48,\ \lambda_2 = 0.39,\ \epsilon = 3,$
% of best score = 100



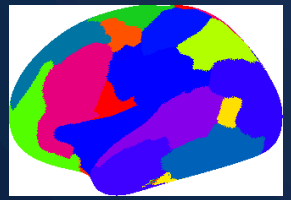
$\lambda_1 = 0.64,\ \lambda_2 = 0.13,\ k = 1,$
% of best score = 75.03



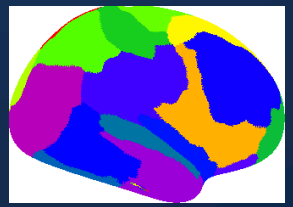
$\lambda_1 = 0.5425,\ \lambda_2 = 0.39,\ k = 0,$
% of best score = 73.45



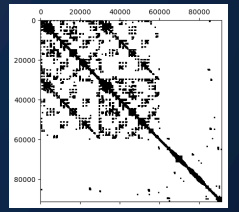
$\lambda_1 = 0.64,\ \lambda_2 = 0.13,\ k = 1,$
% of best score = 75.03



$t = 99.9,\ k = 4,$
% of best score = 32.24



$t = 99.9,\ k = 3,$
% of best score = 32.45



$t = 99.9,\ k = 4$
% of best score = 32.24

[Koanantakool et al. 2018]

Possible Class Projects

- Experiment with SpMV on modern architectures
  - Which optimizations are most effective on KNL? Haswell?
  - Update pOSKI (team effort)
- Speed up particular matrices of interest
  - Machine learning, "bottom solver" from AMR (done)
  - Matrices from your favorite application
- Explore tuning space of $[x, Ax, \ldots, A^k x]$ kernel
  - Different matrix representations
- Experiment with new frameworks (SPF, Halide)
- Other sparse matrix operations:
  - Triangular solve, matrix-matrix, sparse-dense matrix,…
  - See papers on Bebop pages or talk with us