Supporting Structured Data

Saman Amarasinghe

Fredrik Kjolstad (Stanford)
Stephen Chou (MIT)
Rawn Henry (MIT)
Olivia Hsu (Stanford)
Rohan Yadav (Stanford)
Changwan Hong (MIT)
Ryan Senanayake (MIT)
Daniel Donenfeld (MIT)
Peter Ahrens (MIT)
Shoaib Kamil (Adobe)
David Lugato (CEA)

Massachusetts Institute of Technology

MIT CSAIL
compute. collaborate. create

World Is Built For Dense

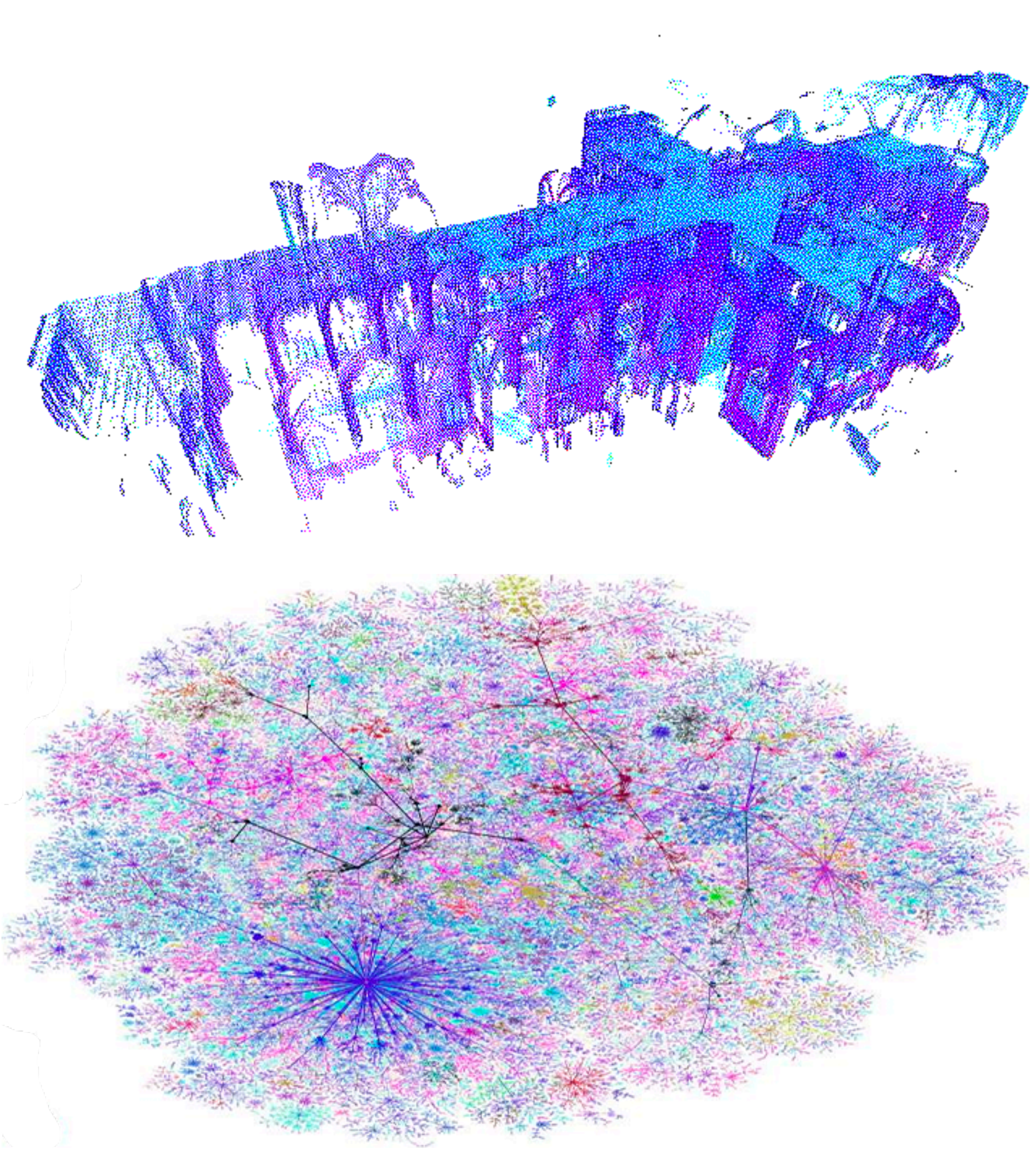
Hardware Utilization

- Peak Performance (GEMM)
  - 70-80% of CPU
  - 80-90% of GPU
- Optimizations
  - Prefetching, Branch Predictions, TLB, cache, ..

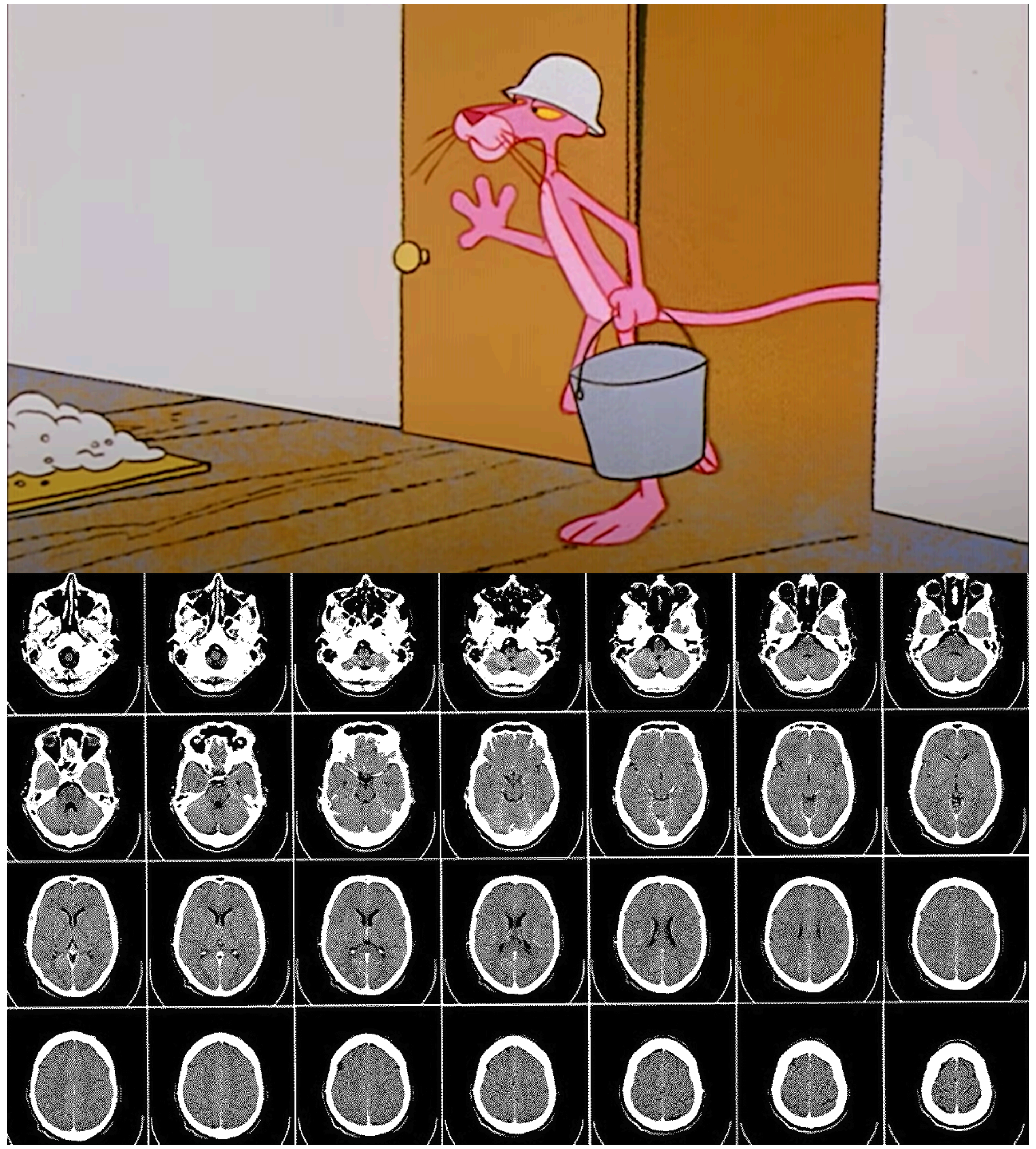
Programming Systems

- Abstractions that Work across Different Algorithms (Dense Linear Algebra, Image Processing, Deep Learning, ..)
  - BLAS, Halide, TensorFlow, ...
  - Optimizing Compilers
    - Tiling, Vectorization, Unrolling, ..

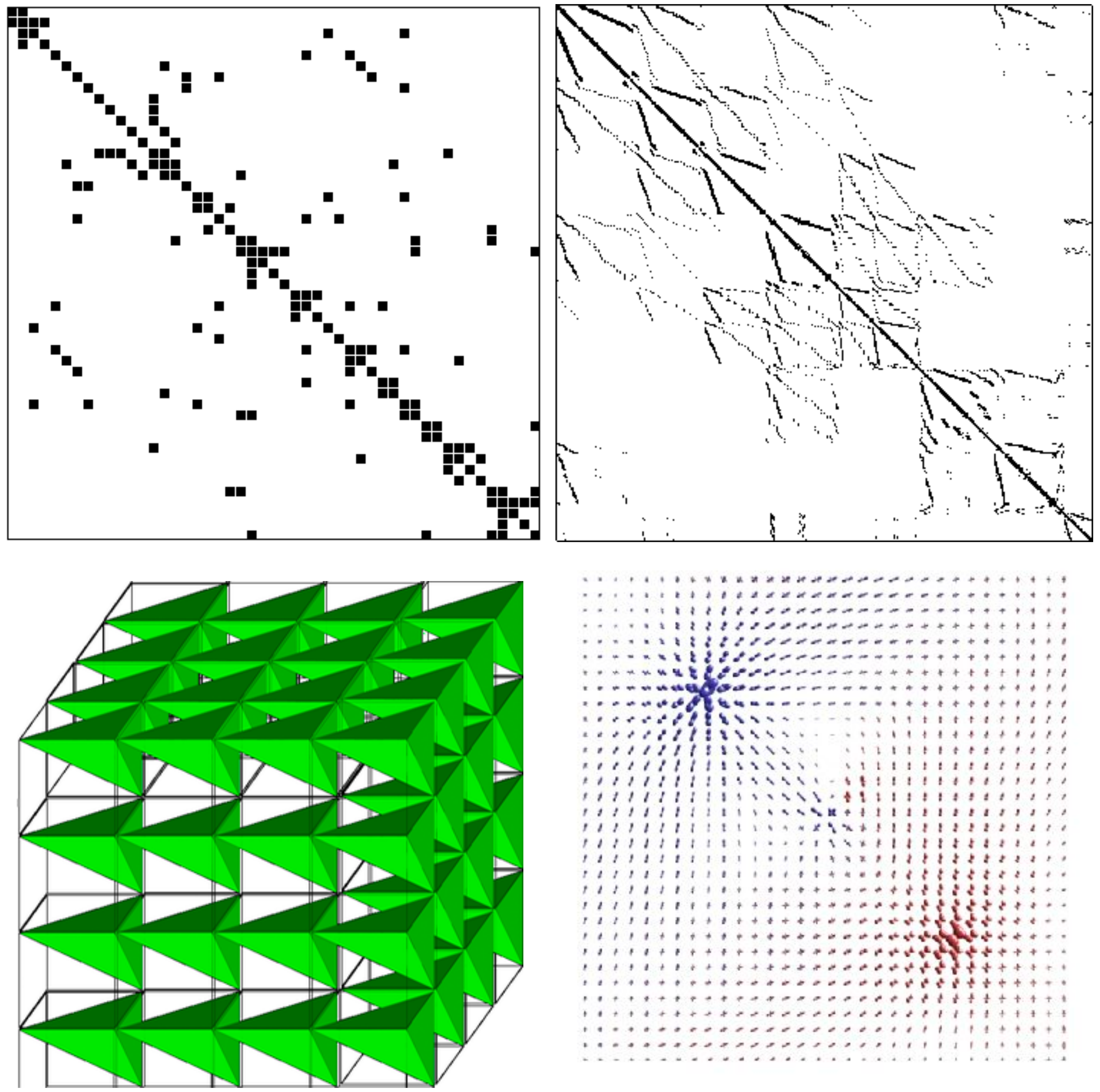
But...Most Data Has Structure



Sparsity



Replicated

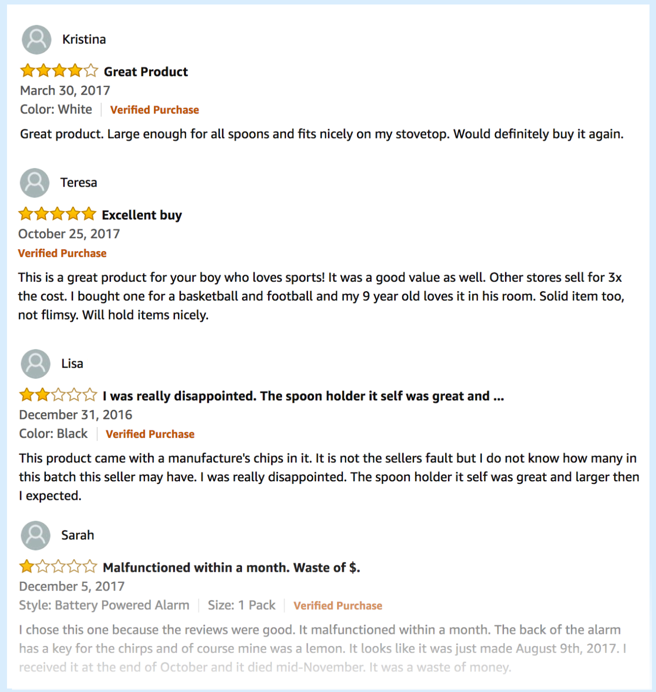


Symmetry

For Example, Sparse Tensors Are Everywhere

Data Analytics

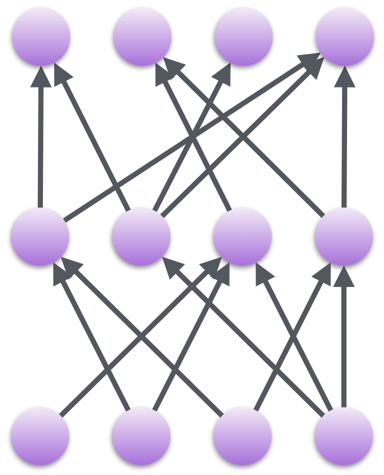
Movies



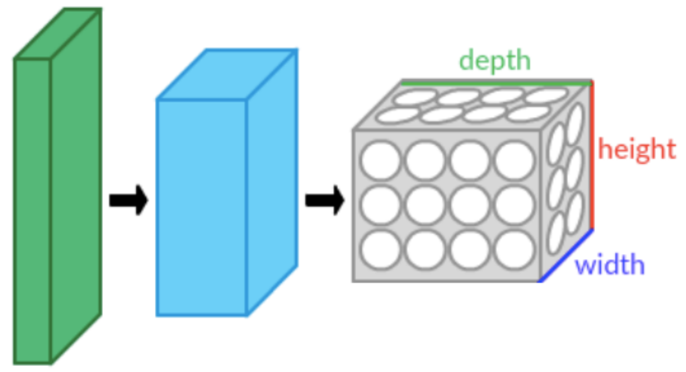
Product Reviews

Social Networks

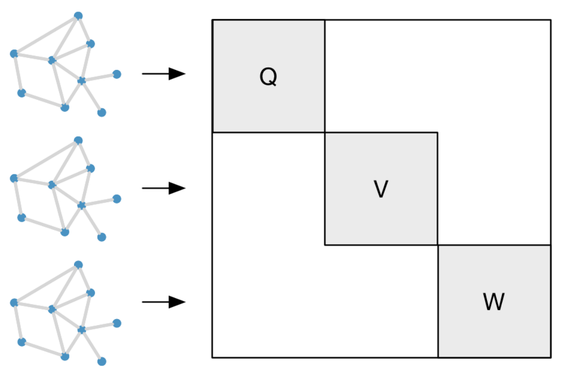
Machine Learning



Sparse Networks

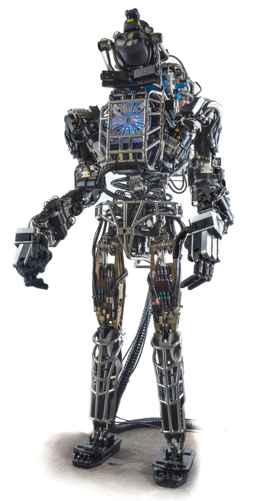


Sparse Convolutional Networks

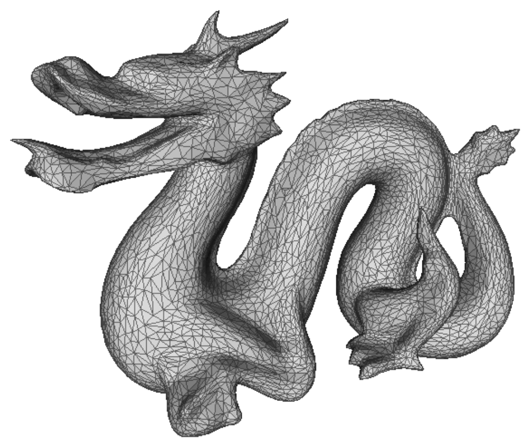


Graph Convolutional Network

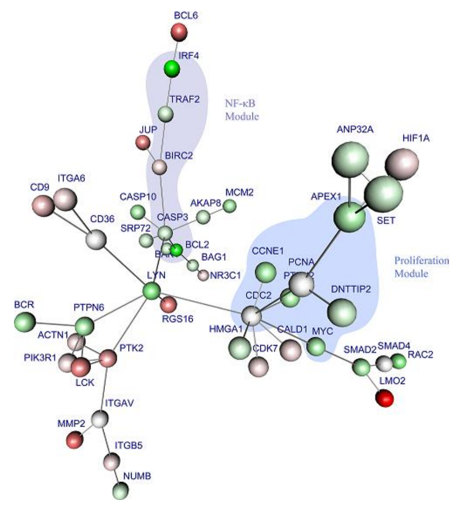
Science and Engineering



Robotics

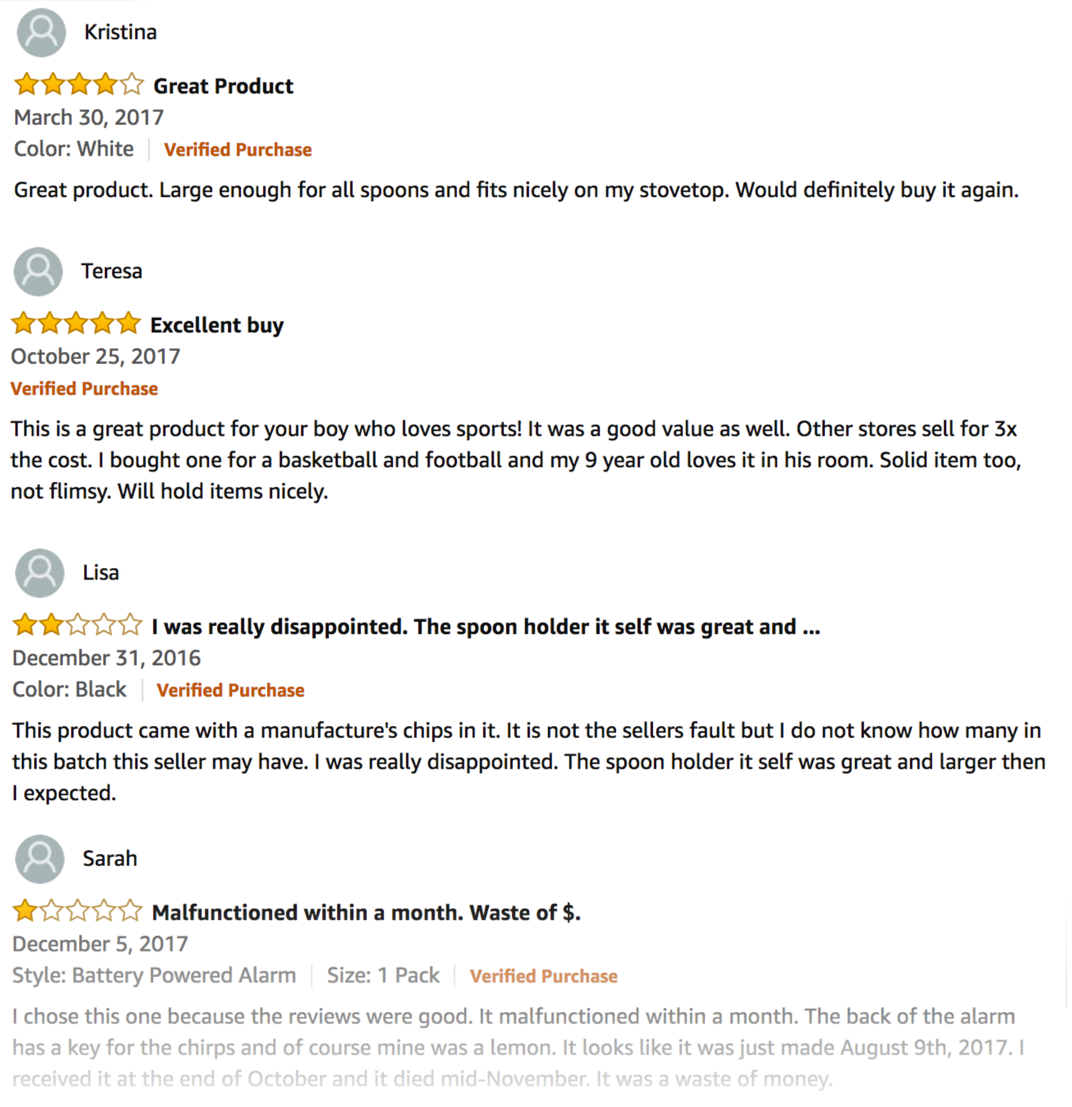


Simulations



Computational Biology

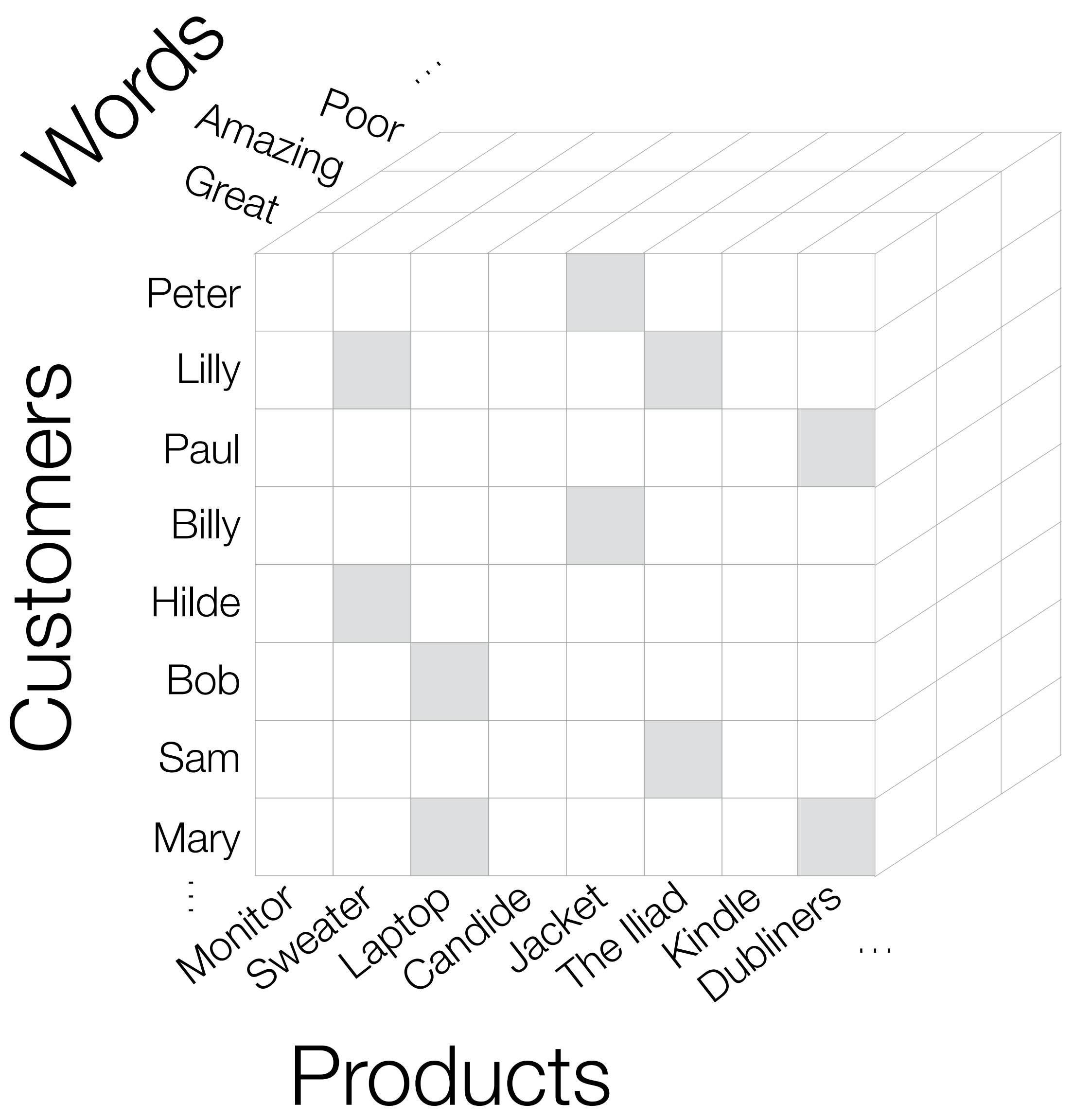
amazon


Kristina
Great Product
March 30, 2017
Color: White | Verified Purchase
Great product. Large enough for all spoons and fits nicely on my stovetop. Would definitely buy it again.

Teresa
Excellent buy
October 25, 2017
Verified Purchase
This is a great product for your boy who loves sports! It was a good value as well. Other stores sell for 3x the cost. I bought one for a basketball and football and my 9 year old loves it in his room. Solid item too, not flimsy. Will hold items nicely.

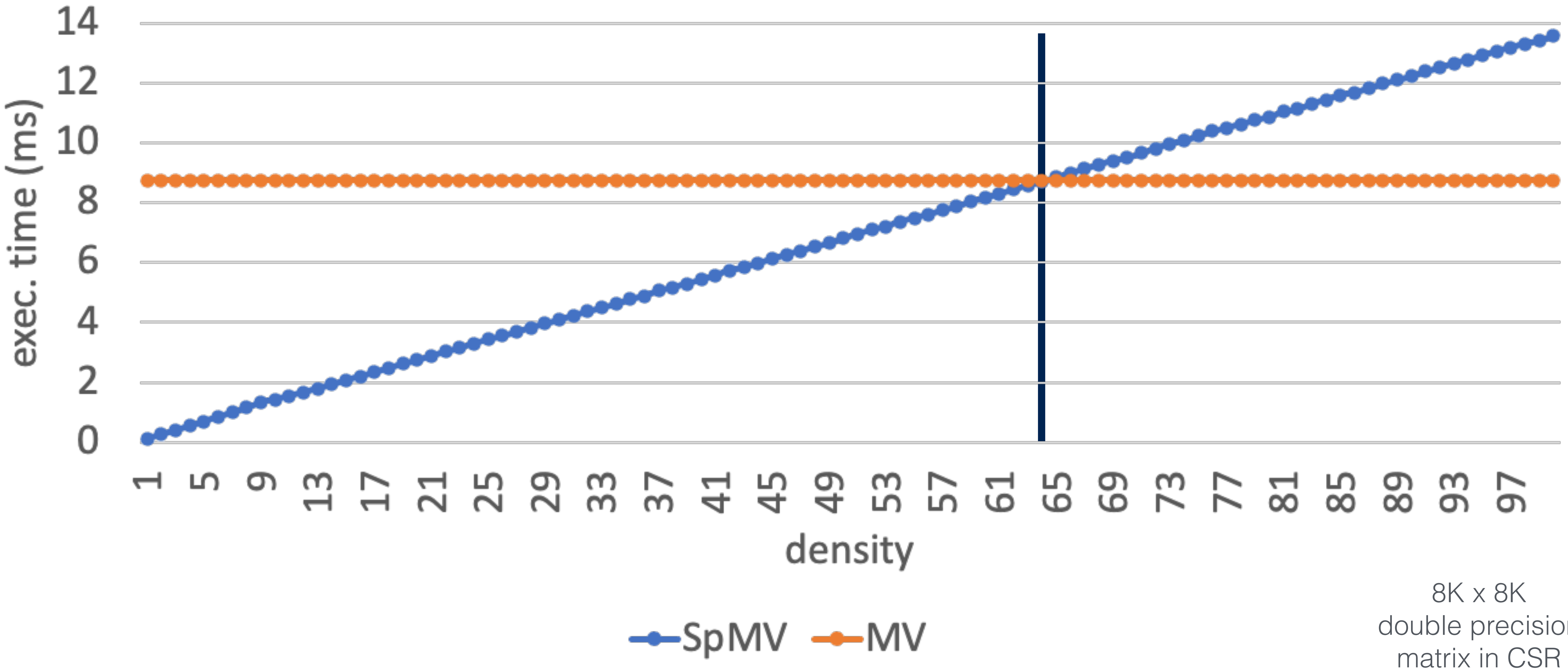
Lisa
I was really disappointed. The spoon holder it self was great and ...
December 31, 2016
Color: Black | Verified Purchase
This product came with a manufacture's chips in it. It is not the sellers fault but I do not know how many in this batch this seller may have. I was really disappointed. The spoon holder it self was great and larger then I expected.

Sarah
Malfunctioned within a month. Waste of $.
December 5, 2017
Style: Battery Powered Alarm | Size: 1 Pack | Verified Purchase
I chose this one because the reviews were good. It malfunctioned within a month. The back of the alarm has a key for the chirps and of course mine was a lemon. It looks like it just made August 9th, 2017. I received it at the end of October and it died mid-November. It was a waste of money.



Extremely sparse
Dense storage: 107 Exabytes
Sparse storage: 13 Gigabytes

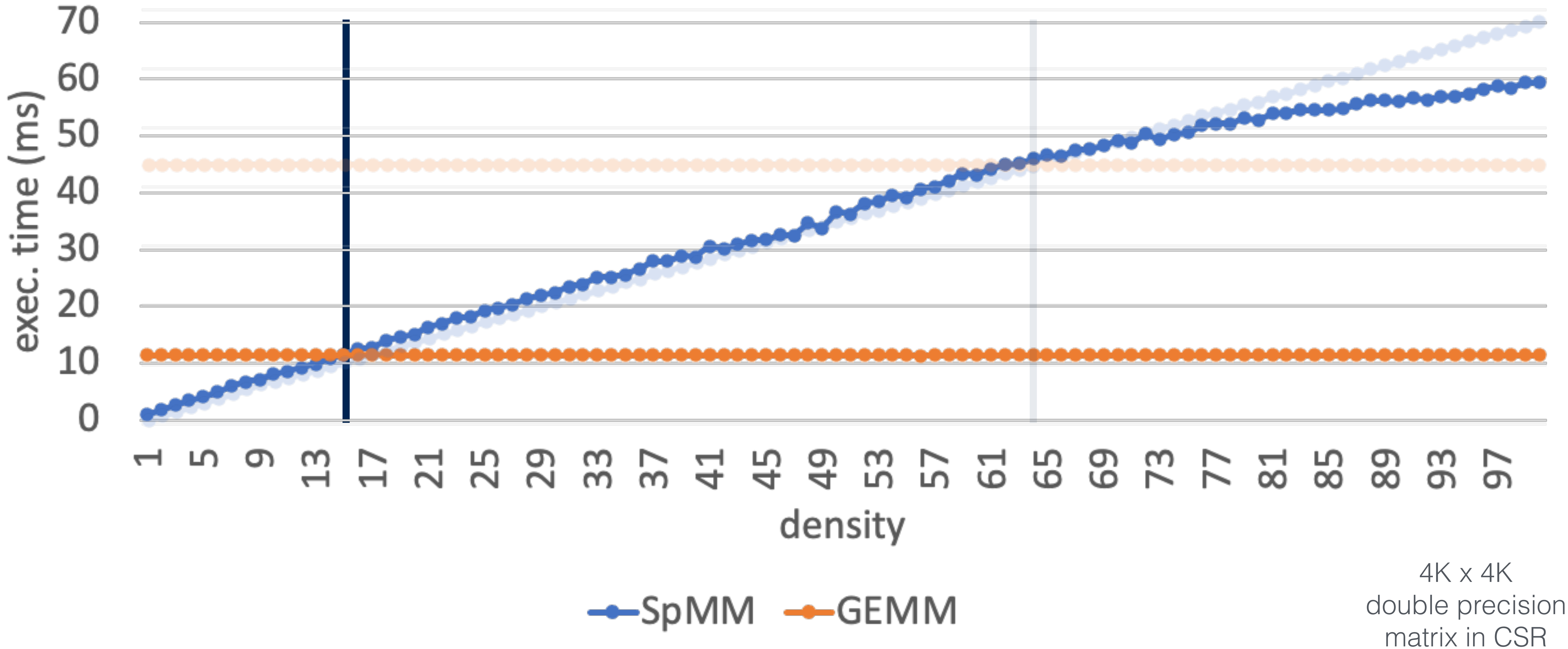
Ignoring Sparsity Is Throwing Away Performance



8K x 8K
double precision
matrix in CSR

Sparse Matrix Vector Multiplication (SpMV)

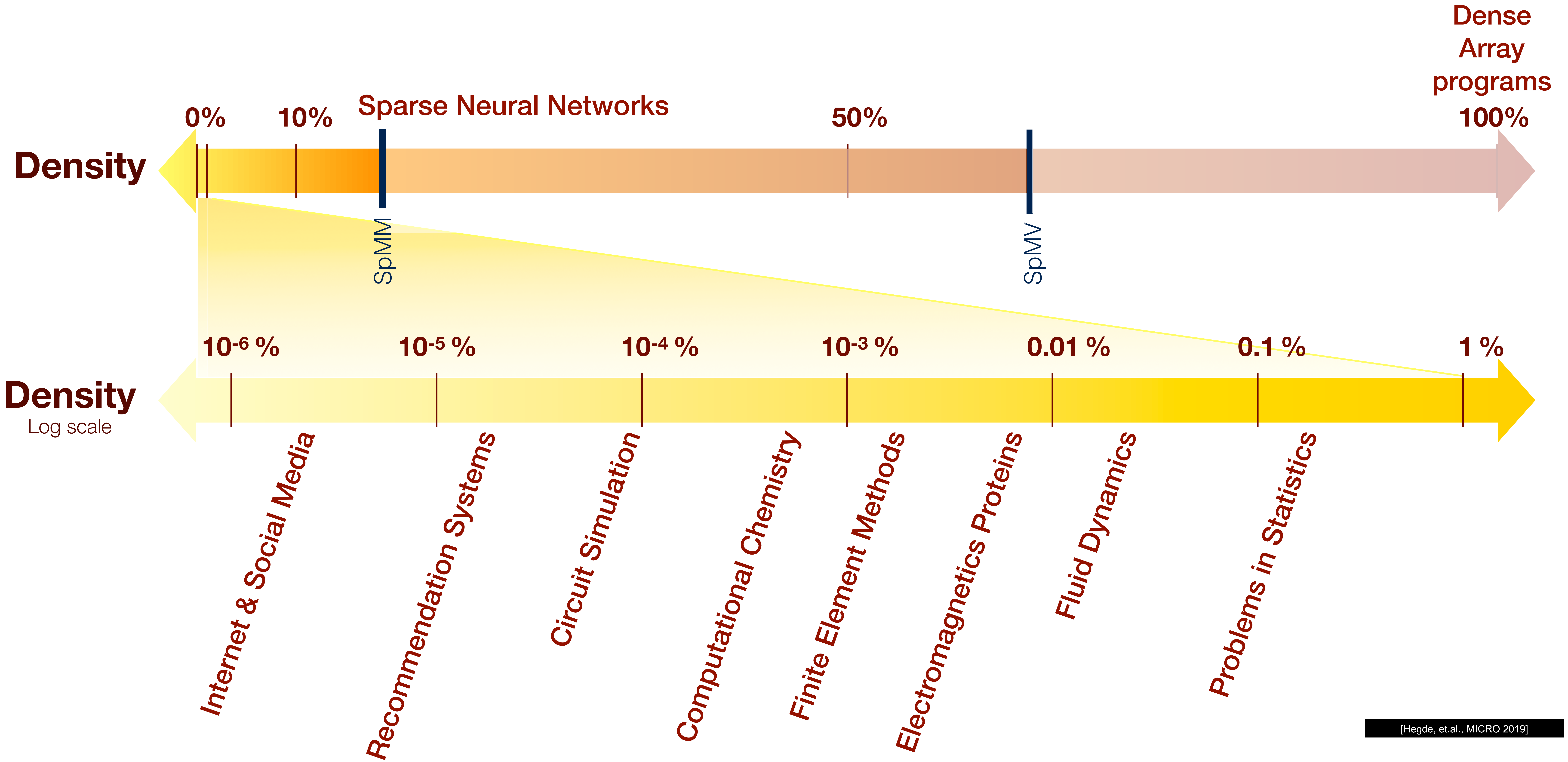
Ignoring Sparsity Is Throwing Away Performance



4K x 4K
double precision
matrix in CSR

Sparse Matrix Matrix Multiplication (SpMV)

Sparse Problems Are Everywhere



Complexity Of Sparse Code

$$A_{ijk} = B_{ijk} + C_{ijk}$$

CSF   COO

```
for (int i = 0; i < m; i++) {
  for (int j = 0; j < n; j++) {
    int pB2 = i*n + j;
    int pA2 = i*n + j;
    double t = 0.0;
    for (int k = 0; k < o; k++) {
      int pB3 = pB2*o + k;
      t += B[pB3] * c[k];
    }
    A[pA2] = t;
  }
}
```

```
int iB = 0;
int C0_pos = C0_pos[0];
while (C0_pos < C0_pos[1]) {
  int iC = C0_crd[C0_pos];
  int C0_end = C0_pos + 1;
  if (iC == iB)
    while ((C0_end < C0_pos[1]) && (C0_crd[C0_end] == iB)) {
      C0_end++;
    }
  if (iC == iB) {
    int B1_pos = B1_pos[iB];
    int C1_pos = C0_pos;
    while ((B1_pos < B1_pos[iB + 1]) && (C1_pos < C0_end)) {
      int jB = B1_crd[B1_pos];
      int jC = C1_crd[C1_pos];
      int j = min(jB, jC);
      int A1_pos = (iB * A1_size) + j;
      int C1_end = C1_pos + 1;
      if (jC == j)
        while ((C1_end < C0_end) && (C1_crd[C1_end] == j)) {
          C1_end++;
        }
      if ((jB == j) && (jC == j)) {
        int B2_pos = B2_pos[B1_pos];
        int C2_pos = C1_pos;
        while ((B2_pos < B2_pos[B1_pos + 1]) && (C2_pos < C1_end)) {
          int kB = B2_crd[B2_pos];
          int kC = C2_crd[C2_pos];
          int k = min(kB, kC);
          int A2_pos = (A1_pos * A2_size) + k;
          if ((kB == k) && (kC == k)) {
            A[A2_pos] = B[B2_pos] + C[C2_pos];
          } else if (kB == k) {
            A[A2_pos] = B[B2_pos];
          } else {
            A[A2_pos] = C[C2_pos];
          }
          if (kB == k) B2_pos++;
          if (kC == k) C2_pos++;
        }
        while (B2_pos < B2_pos[B1_pos + 1]) {
          int kB0 = B2_crd[B2_pos];
          int A2_pos0 = (A1_pos * A2_size) + kB0;
          A[A2_pos0] = B[B2_pos];
          B2_pos++;
        }
        while (C2_pos < C1_end) {
          int kC0 = C2_crd[C2_pos];
          int A2_pos1 = (A1_pos * A2_size) + kC0;
          A[A2_pos1] = C[C2_pos];
          C2_pos++;
        }
      } else if (jB == j) {
        for (int B2_pos0 = B2_pos[B1_pos];
                 B2_pos0 < B2_pos[B1_pos + 1]; B2_pos0++) {
          int kB1 = B2_crd[B2_pos0];
          int A2_pos2 = (A1_pos * A2_size) + kB1;
          A[A2_pos2] = B[B2_pos0];
        }
      } else {
        for (int C2_pos0 = C1_pos; C2_pos0 < C1_end; C2_pos0++) {
          int kC1 = C2_crd[C2_pos0];
          int A2_pos3 = (A1_pos * A2_size) + kC1;
          A[A2_pos3] = C[C2_pos0];
        }
      }
      if (jB == j) B1_pos++;
      if (jC == j) C1_pos = C1_end;
    }
    while (B1_pos < B1_pos[iB + 1]) {
      int jB0 = B1_crd[B1_pos];
      int A1_pos0 = (iB * A1_size) + jB0;
      for (int B2_pos1 = B2_pos[B1_pos];
               B2_pos1 < B2_pos[B1_pos + 1]; B2_pos1++) {
        int kB2 = B2_crd[B2_pos1];
        int A2_pos4 = (A1_pos0 * A2_size) + kB2;
        A[A2_pos4] = B[B2_pos1];
      }
      B1_pos++;
    }
    while (C1_pos < C0_end) {
      int jC0 = C1_crd[C1_pos];
      int A1_pos1 = (iB * A1_size) + jC0;
      int C1_end0 = C1_pos + 1;
      while ((C1_end0 < C0_end) && (C1_crd[C1_end0] == jC0)) {
        C1_end0++;
      }
      for (int C2_pos1 = C1_pos; C2_pos1 < C1_end0; C2_pos1++) {
        int kC2 = C2_crd[C2_pos1];
        int A2_pos5 = (A1_pos1 * A2_size) + kC2;
        A[A2_pos5] = C[C2_pos1];
      }
      C1_pos = C1_end0;
    }
  } else {
    for (int B1_pos0 = B1_pos[iB];
             B1_pos0 < B1_pos[iB + 1]; B1_pos0++) {
      int jB1 = B1_crd[B1_pos0];
      int A1_pos2 = (iB * A1_size) + jB1;
      for (int B2_pos2 = B2_pos[B1_pos0];
               B2_pos2 < B2_pos[B1_pos0 + 1]; B2_pos2++) {
        int kB3 = B2_crd[B2_pos2];
        int A2_pos6 = (A1_pos2 * A2_size) + kB3;
        A[A2_pos6] = B[B2_pos2];
      }
    }
  }
  if (iC == iB) C0_pos = C0_end;
  iB++;
}
while (iB < B0_size) {
  for (int B1_pos1 = B1_pos[iB];
           B1_pos1 < B1_pos[iB + 1]; B1_pos1++) {
    int jB2 = B1_crd[B1_pos1];
    int A1_pos3 = (iB * A1_size) + jB2;
    for (int B2_pos3 = B2_pos[B1_pos1];
             B2_pos3 < B2_pos[B1_pos1 + 1]; B2_pos3++) {
      int kB4 = B2_crd[B2_pos3];
      int A2_pos7 = (A1_pos3 * A2_size) + kB4;
      A[A2_pos7] = B[B2_pos3];
    }
  }
  iB++;
}
```

Sparsity Is Currently Addressed One-Problem-At-A-Time

Eigen (SpMV)

CSparse

$a = Bc$

$a = Bc + a$

OSKI

OSKI has 282 specialized variants of this expression

$a = Bc + b$ $\quad A = B + C$ $\quad a = \alpha Bc + \beta a$

PETSc $\quad a = B^T c$ $\quad A = \alpha B$ $\quad a = B(c + d)$

$a = B^T c + d$ $\quad A = B + C + D$ $\quad A = BC$

$A = B \odot C$ $\quad a = b \odot c$ $\quad A = 0$ $\quad A = B \odot (CD)$

$A = BCd$ $\quad A = B^T$ $\quad a = B^T Bc$

$a = b + c$ $\quad A = B$ $\quad K = A^T CA$

$$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \qquad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$$

$$A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj} \qquad A_{ij} = \sum_{k} B_{ijk} c_k$$

$$A_{ijk} = \sum_{l} B_{ikl} C_{lj} \qquad A_{ik} = \sum_{j} B_{ijk} c_j$$

$$A_{jk} = \sum_{i} B_{ijk} c_i \qquad A_{ijl} = \sum_{k} B_{ikl} C_{kj}$$

$$\tau = \sum_i z_i \Big(\sum_j z_j \theta_{ij}\Big)\Big(\sum_k z_k \theta_{ik}\Big)$$

$$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$$

$$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$$

Linear Algebra

Data analytics (tensor factorization)

Quantum Chromodynamics

$\times$

Dense Matrix

CSR DCSR BCSR

COO ELLPACK CSB

Blocked COO CSC

DIA Blocked DIA DCSC

Sparse vector Hash Maps

Coordinates

CSF

Dense Tensors

Blocked Tensors

Finite Elements Method,
Block-Sparse NN Weights [GRK 2017]

Data Analytics

Convolutions, Image Processing

$\times$

CPU

GPUs

TPUs

FPGA

Sparse Tensor Hardware

Cloud Computers

Supercomputers

World Is Built For Dense...What About Sparse?

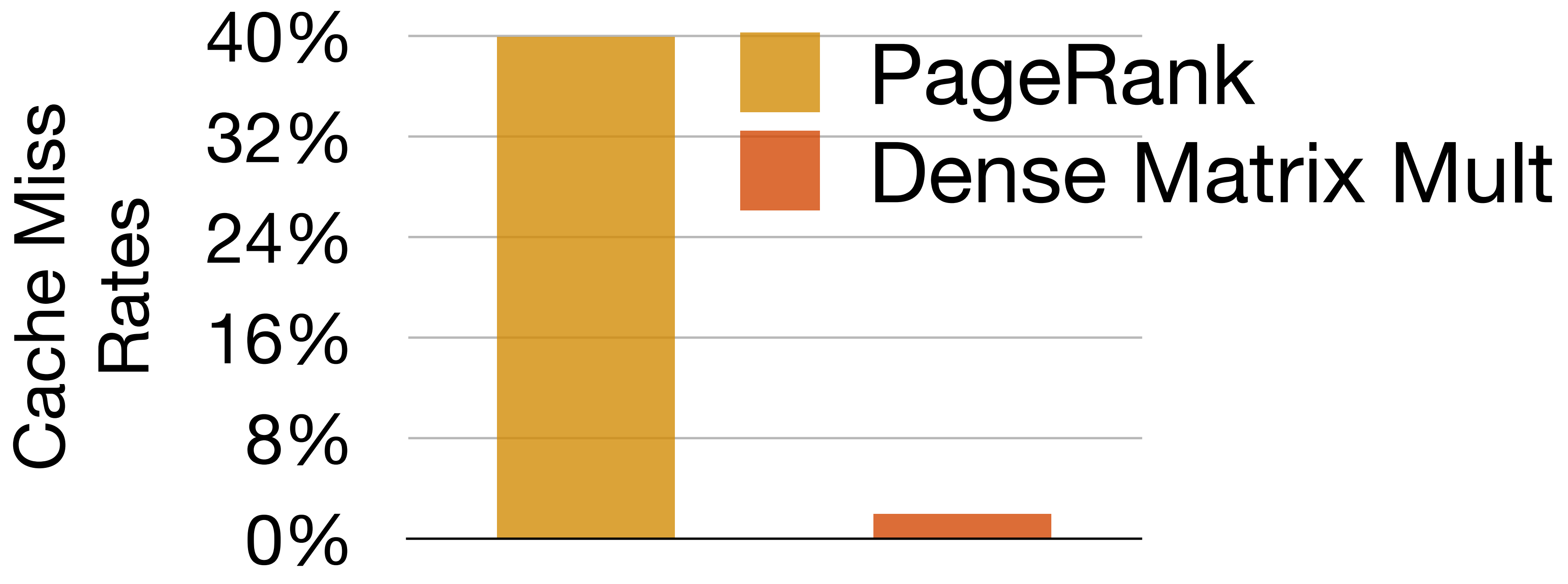
Hardware Utilization

- Peak Performance (GEMM)
  - 70-80% of CPU
  - 80-90% of GPU
- Optimizations
  - Prefetching, Branch Predictions, TLB, cache, ..

Programming Systems

- Abstractions that Work across Different Algorithms
  - BLAS, Halide, TensorFlow, ...
  - Optimizing Compilers
    - Tiling, Vectorization, Unrolling, ..

- Peak Performance (PageRank, SpMv)
  - < 10% Peak of CPU and GPU



- What abstraction??

```
template<typename APPLY_FUNC>
void edgeset_apply_pull_parallel(Graph &g, APPLY_FUNC apply_func) {
    int64_t numVertices = g.num_nodes(), numEdges = g.num_edges();
    parallel_for(int n = 0; n < numVertices; n++) {
        for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
            local_new_rank[socketId][n] = new_rank[n]; } }
    int numPlaces = omp_get_num_places();
    int numSegments = g.getNumSegments("s1");
    int segmentsPerSocket = (numSegments + numPlaces - 1) / numPlaces;
        #pragma omp parallel num_threads(numPlaces) proc_bind(spread){
        int socketId = omp_get_place_num();
        for (int i = 0; i < segmentsPerSocket; i++) {
            int segmentId = socketId + i * numPlaces;
            if (segmentId >= numSegments) break;
            auto sg = g.getSegmentedGraph(std::string("s1"), segmentId);
            #pragma omp parallel num_threads(omp_get_place_num_procs(socketId)) proc_bind(close){
                #pragma omp for schedule(dynamic, 1024)
                for (NodeID localId = 0; localId < sg->numVertices; localId++) {
                    NodeID d = sg->graphId[localId];
                    for (int64_t ngh = sg->vertexArray[localId]; ngh < sg->vertexArray[localId + 1]; ngh++) {
                        NodeID s = sg->edgeArray[ngh];
                        local_new_rank[socketId][d] += contrib[s]; }}}}}
    parallel_for(int n = 0; n < numVertices; n++) {
        for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
            new_rank[n] += local_new_rank[socketId][n]; }}}
struct updateVertex {
    void operator() (NodeID v) {
        double old_score = old_rank[v];
        new_rank[v] = (base_score + (damp * new_rank[v]));
        error[v] = fabs((new_rank[v] - old_rank[v])) ;
        old_rank[v] = new_rank[v];
        new_rank[v] = ((float) 0) ; }; };
void pagerank(Graph &g, double *new_rank, double *old_rank, int *out_degree, int max_iter) {
    for (int i = (0); i < (max_iter); i++) {
        parallel_for(int v_iter = 0; v_iter < builtin_getVertices(edges); v_iter ++) {
            contrib[v] = (old_rank[v] / out_degree[v]);};
        edgeset_apply_pull_parallel(edges, updateEdge());
        parallel_for(int v_iter = 0; v_iter < builtin_getVertices(edges); v_iter ++) {
            updateVertex()(v_iter); }; }
```

Optimized PageRank for Multi-Core CPU

Sparse Tensor Compiler (Taco)

Expression Language

$A = Bc + a$ $\quad a = Bc$

$A = B \odot C$ $\quad A = B + C$ $\quad a = \alpha Bc + \beta a$

$A = BCd$ $\quad A = \alpha B$ $\quad A = 0$ $\quad A = BC$

$a = b \odot c$ $\quad A = B \odot (CD)$

$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj}$ $\quad A = B^T$ $\quad a = B^T Bc$

$A_{ik} = \sum_{j} B_{ijk} c_j$ $\quad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$

$A_{ijk} = \sum_{l} B_{ikl} C_{lj}$ $\quad A_{ij} = (\sum_{k} B_{ijk} C_{ijk}) + D_{ij}$

$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$ $\quad \tau = \sum_{i} z_i (\sum_{j} z_j \theta_{ij})(\sum_{k} z_k \theta_{ik})$

$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$

Format Language

Dense Matrix, DCSR, CSR, BCSR, COO, CSF, DIA, ELLPACK, CSB, Hash Maps, Blocked COO, CSC, DCSC, Sparse vector, Blocked DIA, Dense Tensors, Blocked Tensors

Schedule Language

pos, reorder, vectorize, precompute, divide, split, parallelize

The Sparse Tensor Compiler (TACO)

THE C PROGRAMMING LANGUAGE

NVIDIA CUDA

Structured Data Tensor Compiler

Expression Language

$A = Bc + a$ $\quad a = Bc$

$A = B \odot C$ $\quad A = B + C$ $\quad a = \alpha Bc + \beta a$

$A = BCd$ $\quad A = \alpha B$ $\quad A = 0$ $\quad A = BC$

$a = b \odot c$ $\quad A = B \odot (CD)$

$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj}$ $\quad A = B^T$ $\quad a = B^T Bc$

$A_{ik} = \sum_{j} B_{ijk} c_j$ $\quad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$

$A_{ijk} = \sum_{l} B_{ikl} C_{lj}$ $\quad A_{ij} = (\sum_{k} B_{ijk} C_{ijk}) + D_{ij}$

$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$ $\quad \tau = \sum_{i} z_i (\sum_{j} z_j \theta_{ij})(\sum_{k} z_k \theta_{ik})$

$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$

Format Language

Dense Matrix DCSR CSR BCSR
COO CSF DIA ELLPACK CSB
Hash Maps Blocked COO CSC
DCSC Sparse vector Blocked DIA
Dense Tensors Blocked Tensors
PackBITS Banded VBR Ragged
LZ77 RLE Definite Symmetric

Schedule Language

pos reorder vectorize
precompute divide split
parallelize

Looplet Language

Lookup Run Spike
Pipeline Stepper Jumper
Switch Shift

Structured Data Tensor Compiler

THE C PROGRAMMING LANGUAGE

NVIDIA CUDA

Generated Sparse Code Performance Matches Hand-Optimized Libraries

$a = Bc + a$ $\quad a = Bc$

$a = Bc + b$ $\quad A = B + C$ $\quad a = \alpha Bc + \beta a$

$a = B^T c$ $\quad A = \alpha B$ $\quad a = B(c + d)$

$a = B^T c + d$ $\quad A = B + C + D$ $\quad A = BC$

$A = 0$

$A = B \odot C$ $\quad a = b \odot c$ $\quad A = B \odot (CD)$

$A = BCd$ $\quad A = B^T$ $\quad a = B^T Bc$

$a = b + c$ $\quad A = B$ $\quad K = A^T C A$

$$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \qquad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$$

$$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \qquad A_{ij} = \sum_{k} B_{ijk} c_k$$

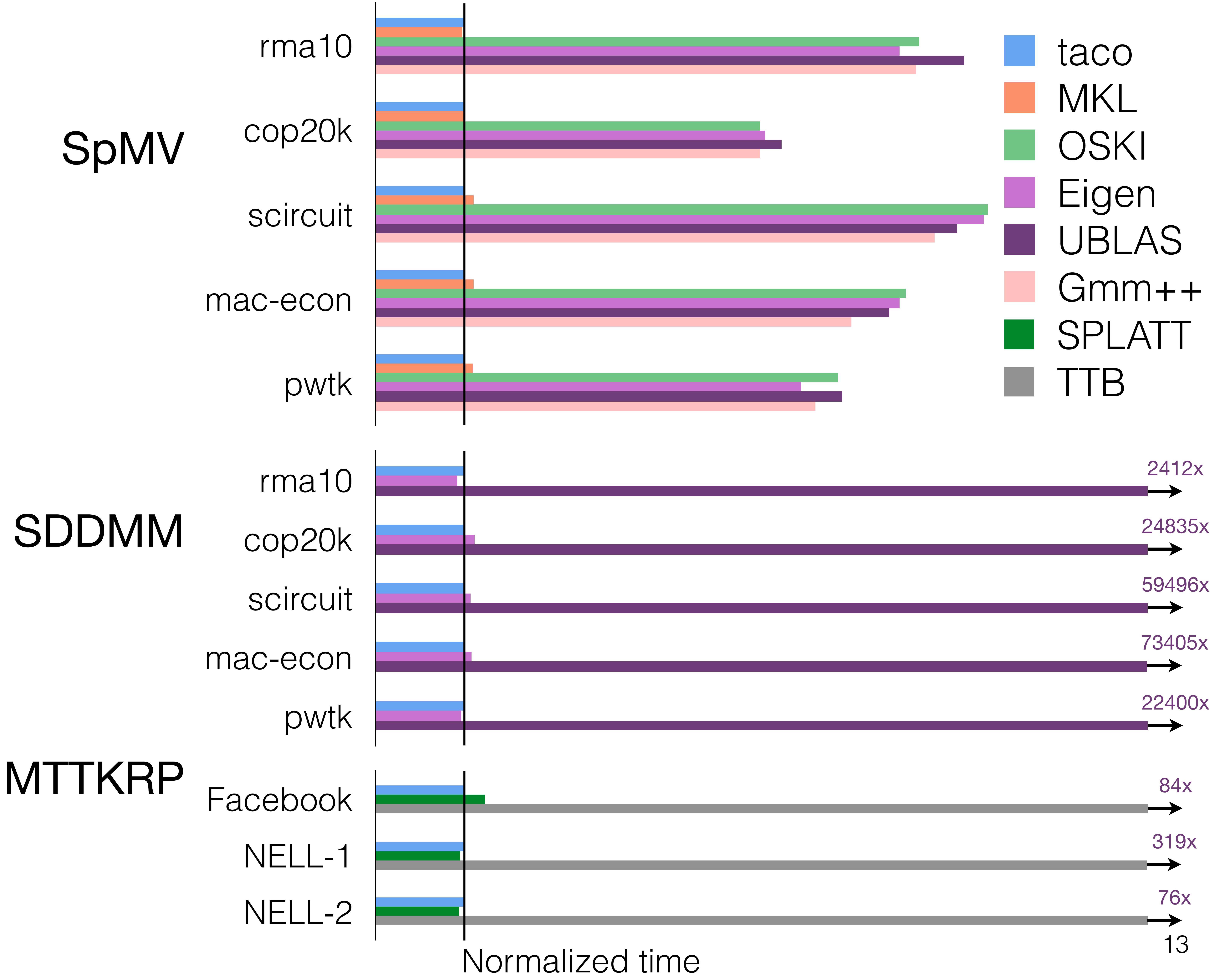
$$A_{ijk} = \sum_{l} B_{ikl} C_{lj} \qquad A_{ik} = \sum_{j} B_{ijk} c_j$$

$$A_{jk} = \sum_{i} B_{ijk} c_i \qquad A_{ijl} = \sum_{k} B_{ikl} C_{kj}$$

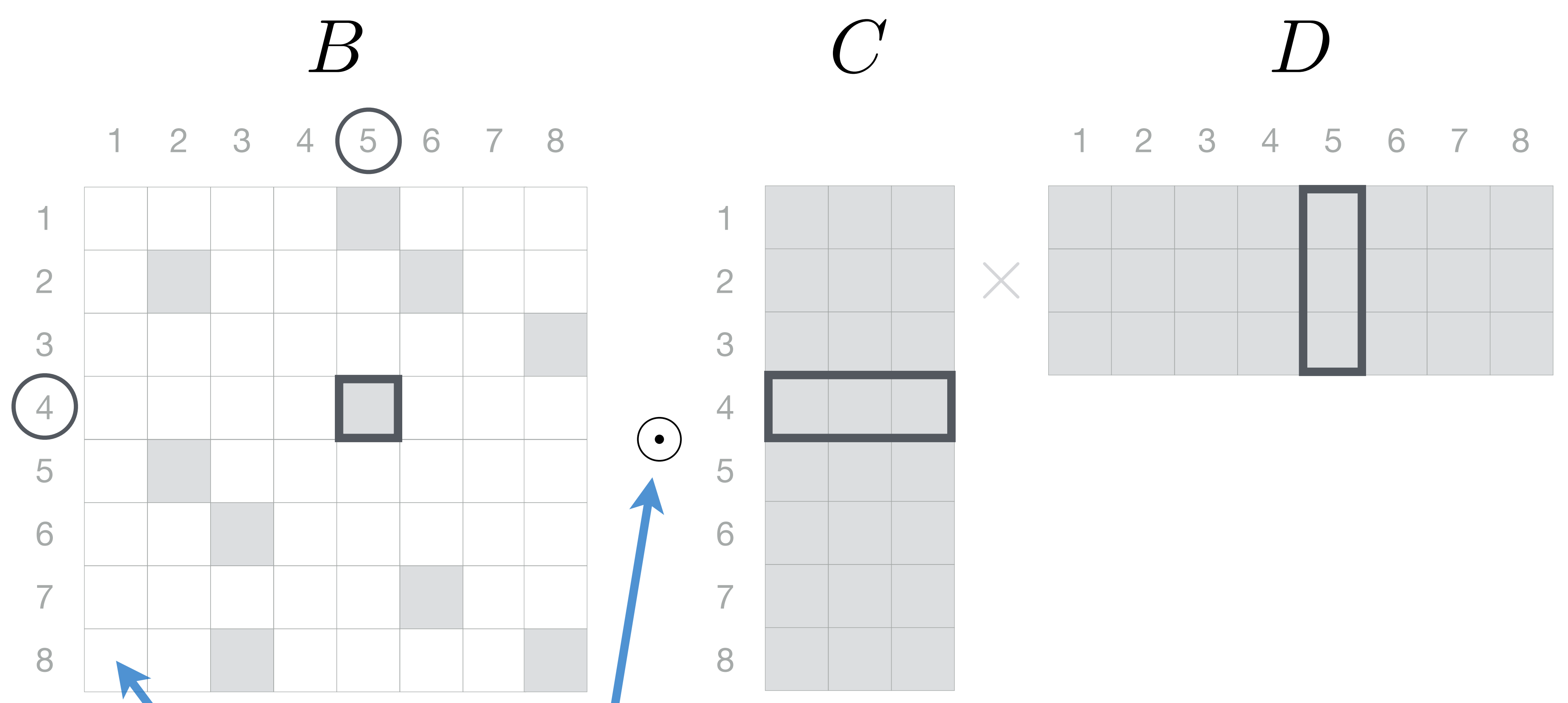
$$\tau = \sum_{i} z_i \Big(\sum_{j} z_j \theta_{ij}\Big)\Big(\sum_{k} z_k \theta_{ik}\Big)$$

$$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$$

$$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$$



Generated Sparse Code Performance Matches Hand-Optimized Libraries



Element-wise multiplication

This dot product need not be computed

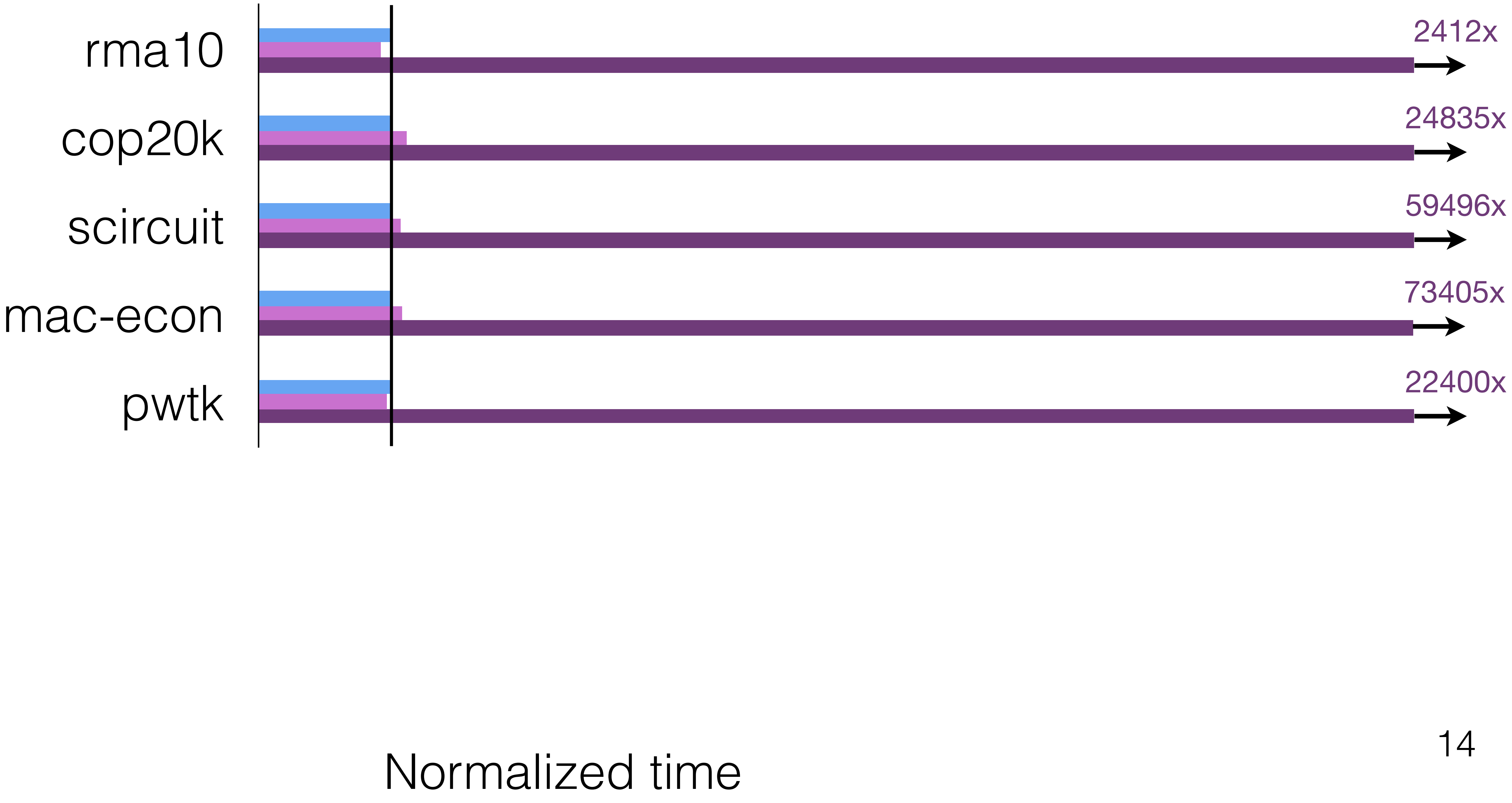
~~64 inner product~~
10 inner product

We will generate fused operations

SDDMM

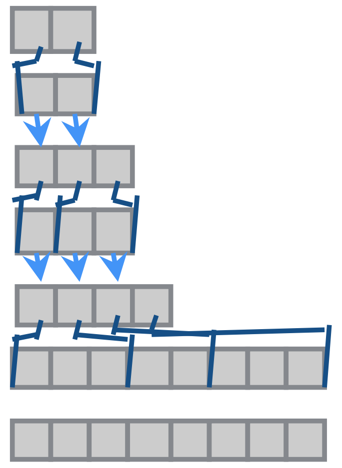
$$A = B \odot (CD)$$

Sampled Dense-Dense Matrix Multiplication

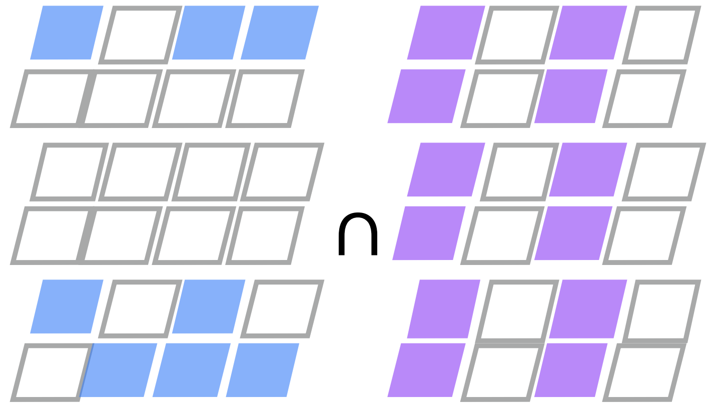


Challenges Of Sparse Array Compilation

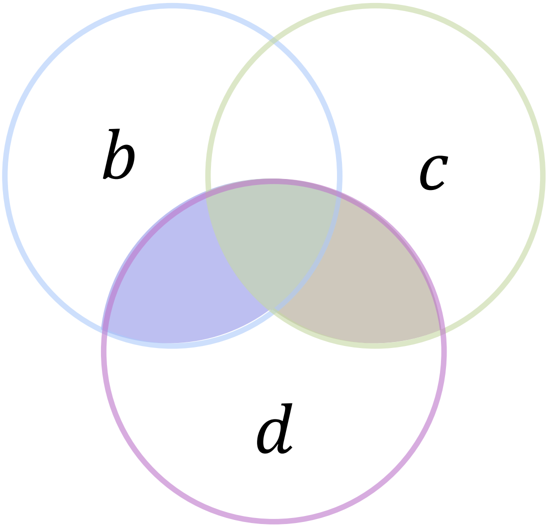
Irregular Data Structures



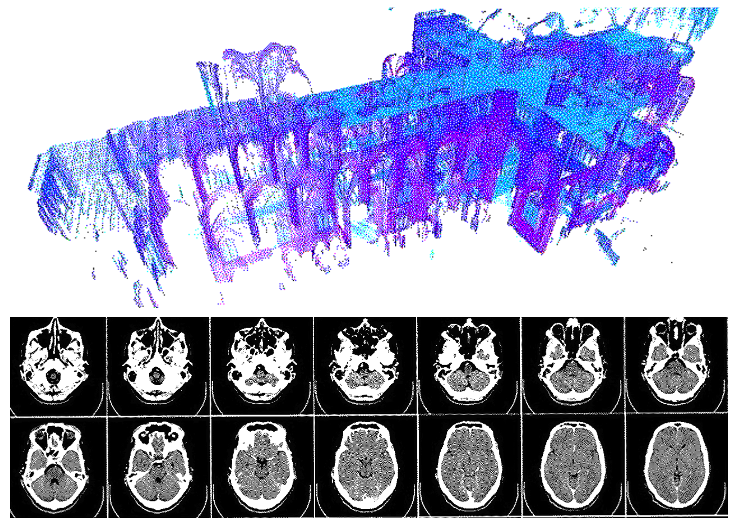
Sparse Iteration with limited O(1) access



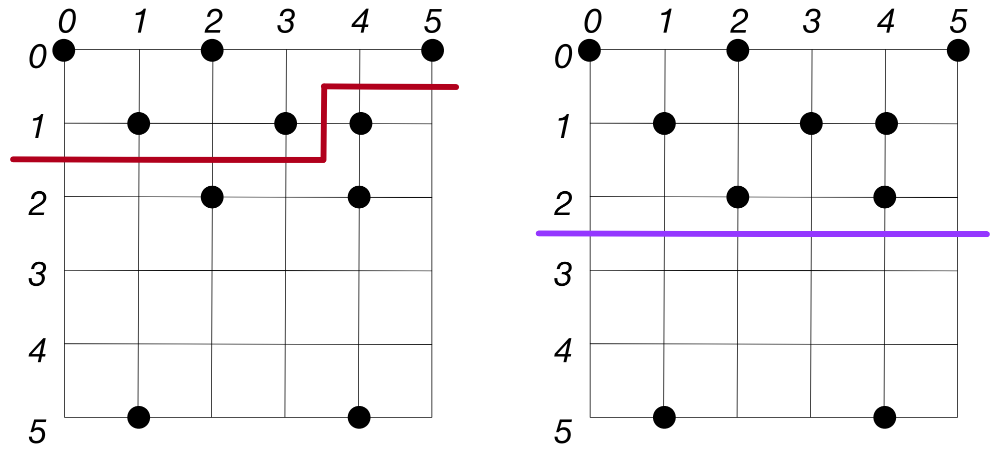
Avoid wasted work and iterations



Coiteration Over Complex Data

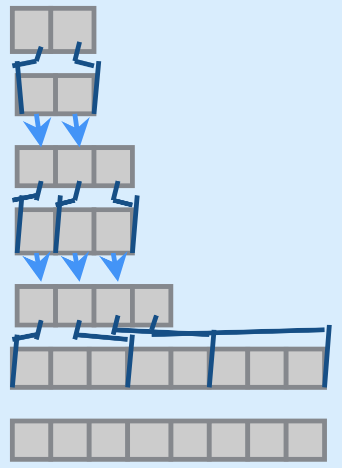


Optimize Parallelism and Locality



Challenges Of Sparse Array Compilation

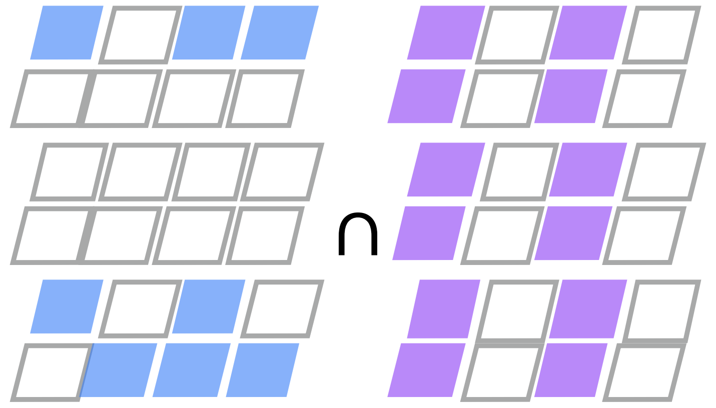
Irregular Data Structures



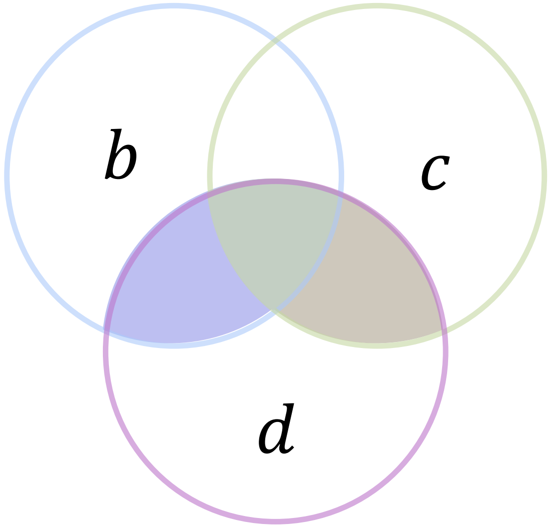
Format Language

CSF
Dense
Compressed
Compressed

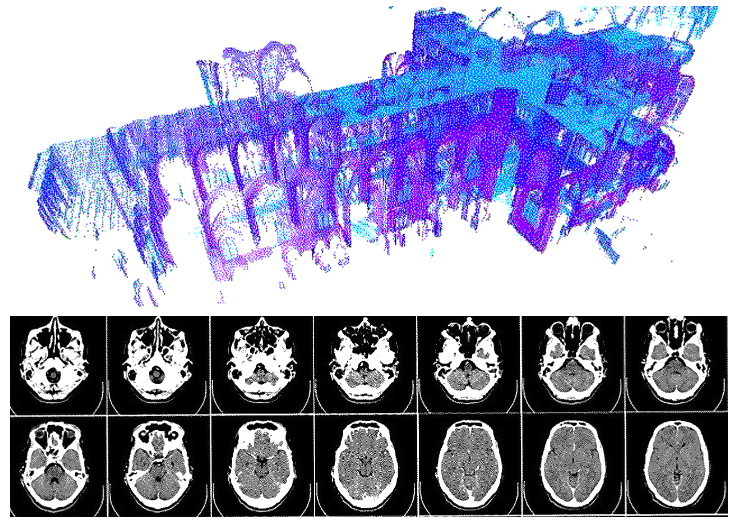
Sparse Iteration with limited O(1) access



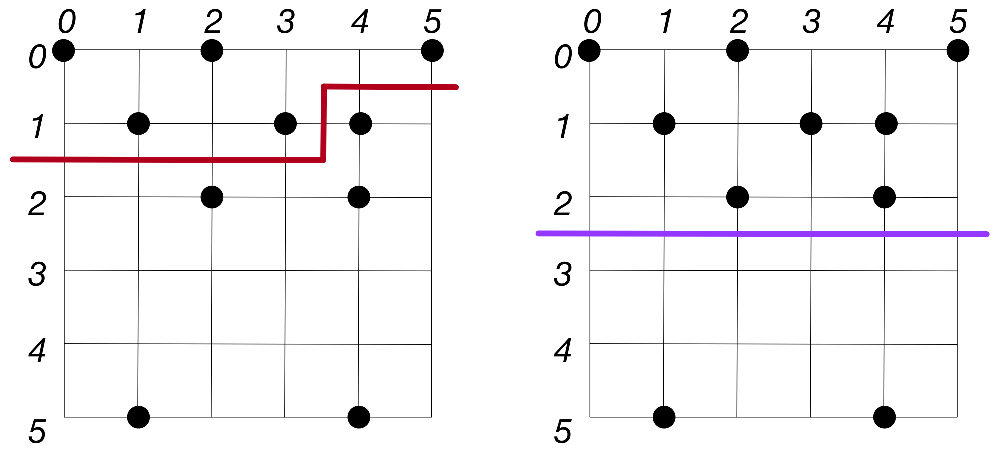
Avoid wasted work and iterations



Coiteration Over Complex Data



Optimize Parallelism and Locality



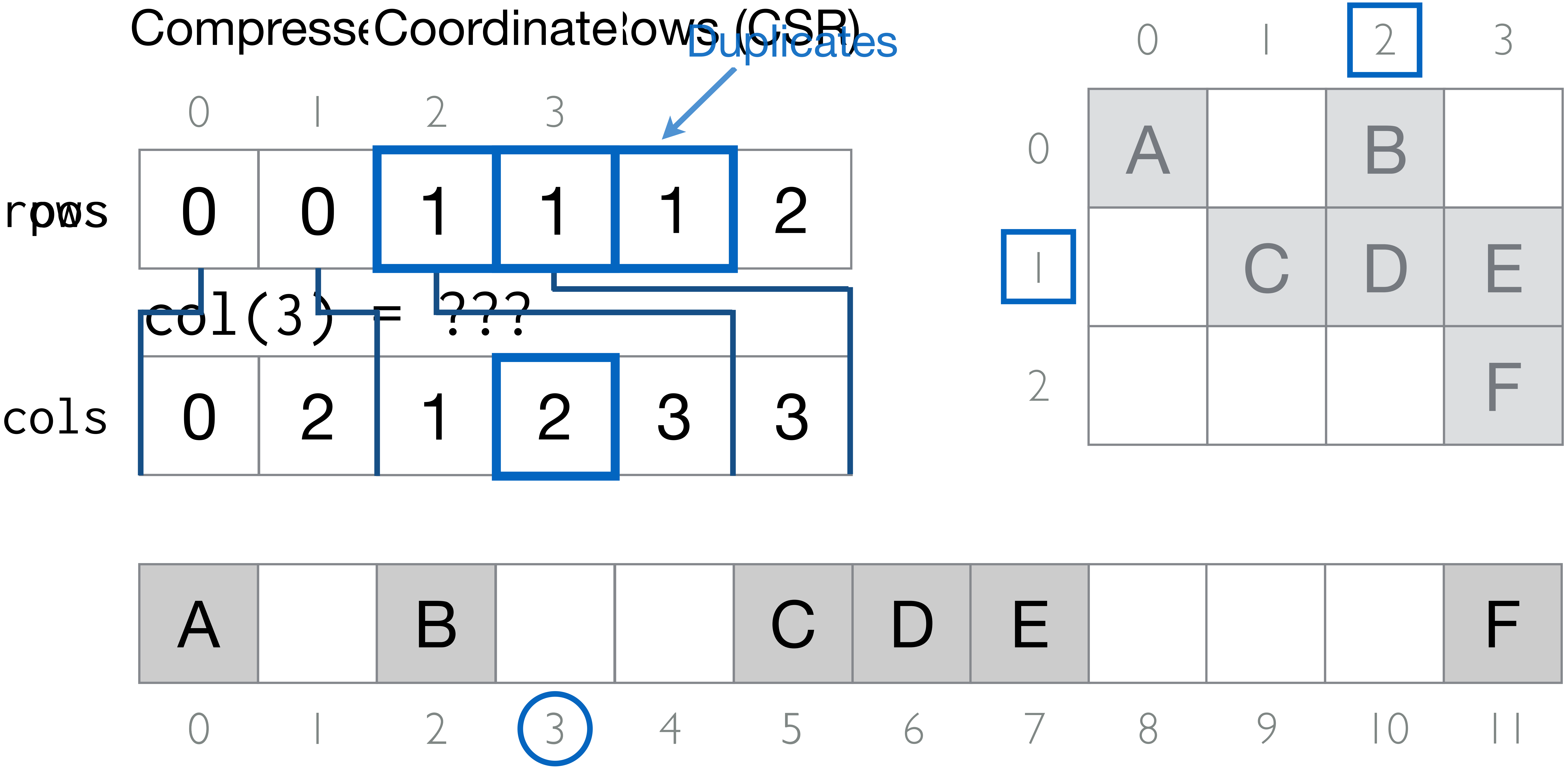
Dense Tensors Are Flexible But Can Waste Memory

```
locate(1,2) = 1*4 + 2
            = 6
```

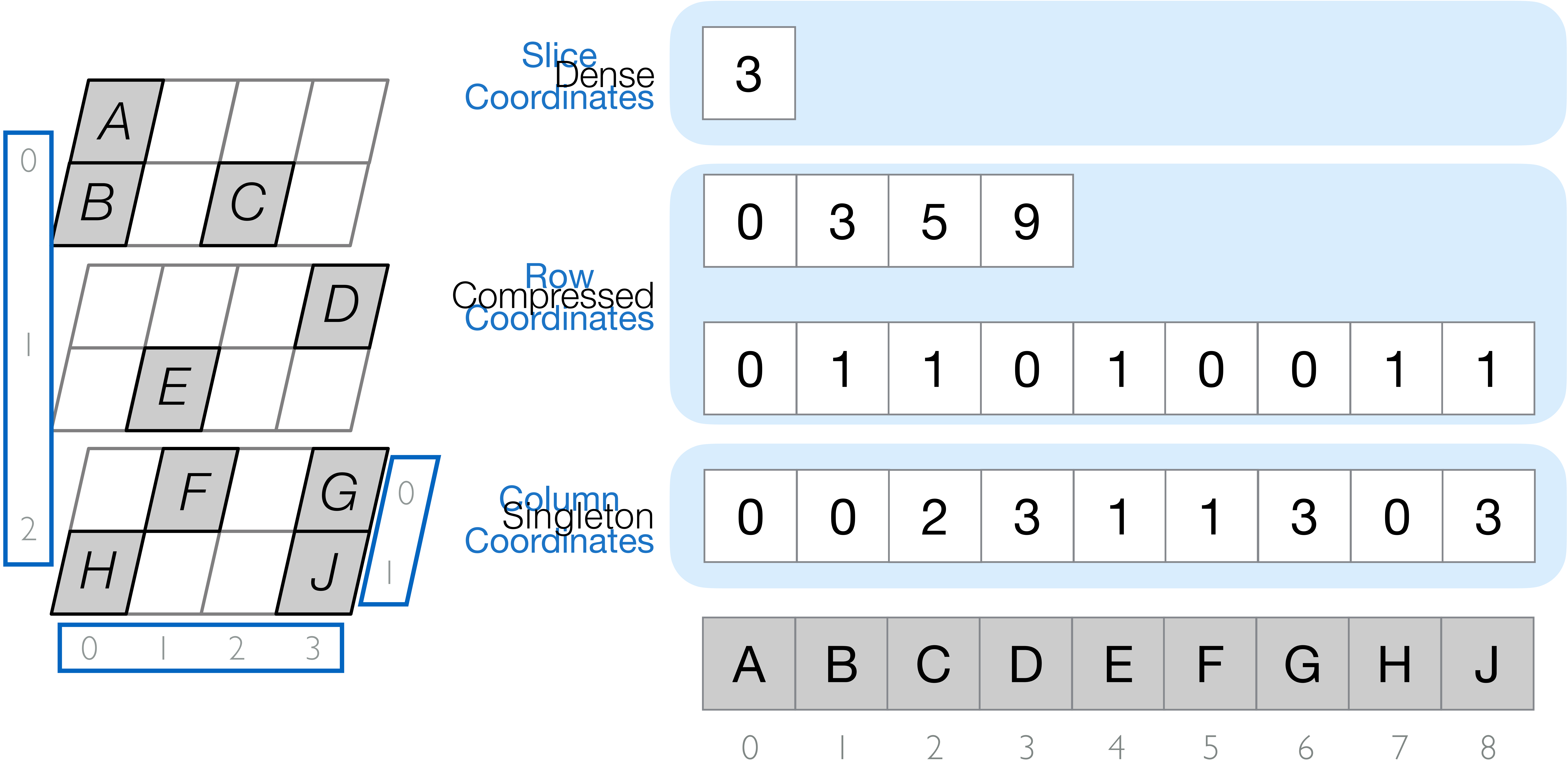
	0	1	2	3
0	A		B	
1		C	D	E
2				F

0 1 2 3 4 5 $\boxed{6}$ 7 8 9 10 11

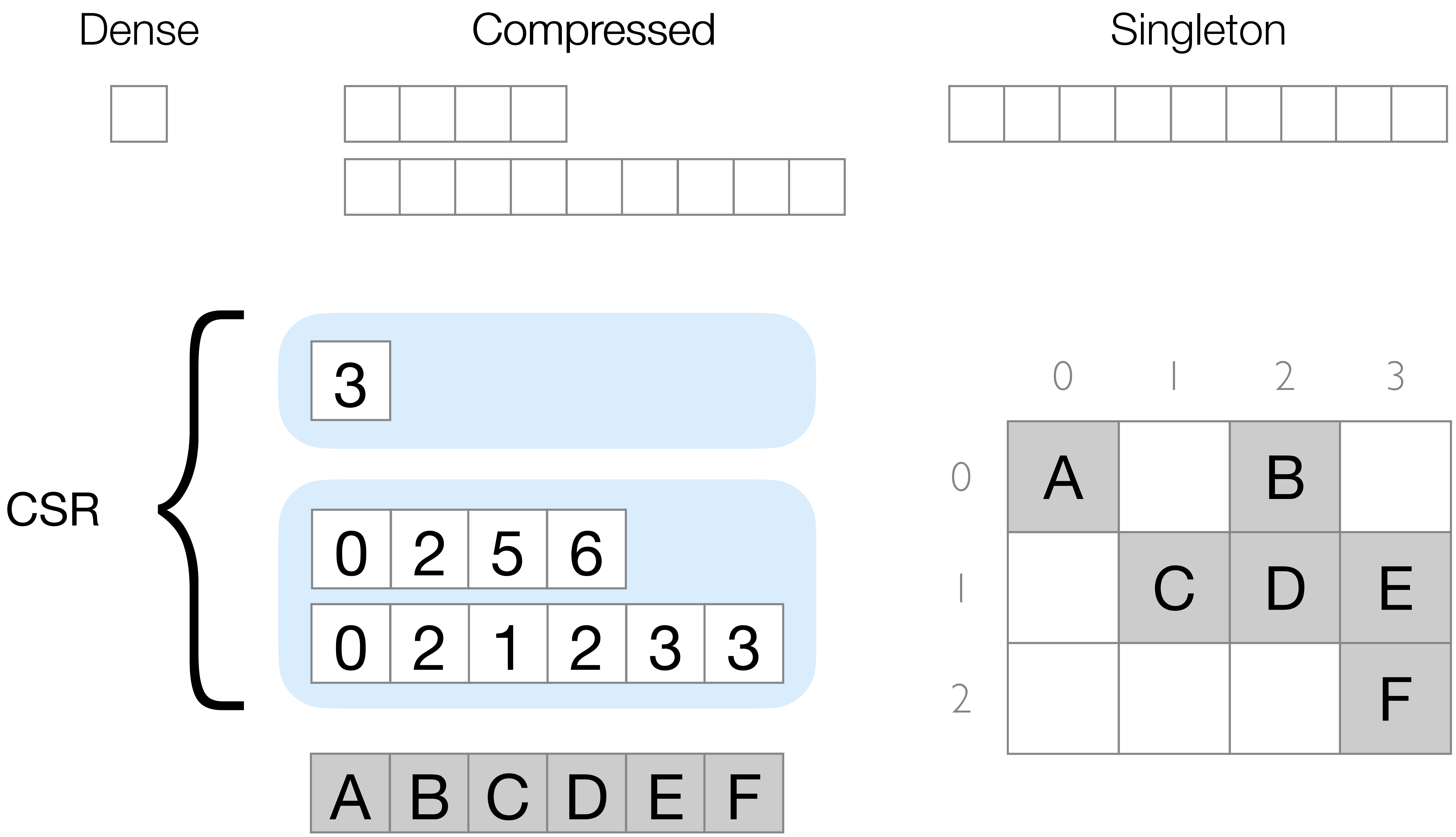
Sparse Tensors Can Be Compressed By Adding Metadata



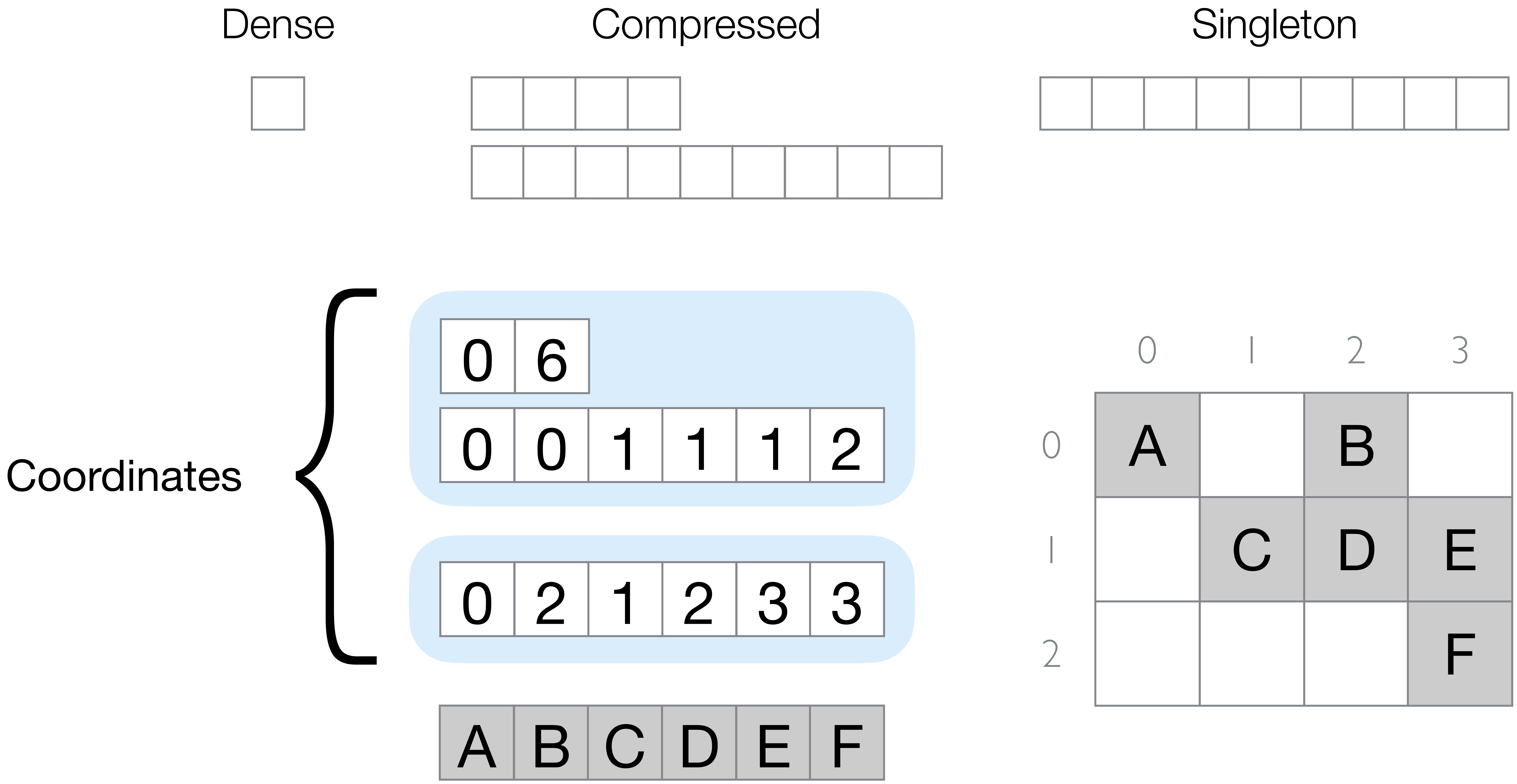
We Model Tensor Formats As A Hierarchy Of Per-Dimension Formats



Per-Dimension Formats Can Be Composed In Many Ways



Per-Dimension Formats Can Be Composed In Many Ways



Level Formats Can Be Composed In Many Ways

Level formats

Dense, Compressed, Singleton, Hashed, Range, Offset

Tensor formats

Coordinate matrix
Compressed
Singleton

CSR
Dense
Compressed

[Tinney and Walker, 1967]

Dense array tensor
Dense
Dense
Dense

Coordinate tensor
Compressed
Singleton
Singleton

Mode-generic tensor
Compressed
Singleton
Dense
Dense

[Baskaran et al. 2012]

BCSR
Dense
Compressed
Dense
Dense

[Im and Yelick 1998]

CSB
Dense
Dense
Compressed
Singleton

[Buluç et al. 2009]

ELLPACK
Dense
Dense
Singleton

[Kincaid et al. 1989]

Hash map vector
Hashed

[Patwary et al. 2015]

Hash map matrix
Hashed
Hashed

DIA
Dense
Range
Offset

[Saad 2003]

Block DIA
Dense
Range
Offset
Dense
Dense

Sparsity Beyond Zero Fill Values

	0	1	2	3	4	5	6	7
0	1	1	1	0	0	0	5	5
1	6	6	6	6	1	1	1	1
2	3	3	3	0	0	0	0	0
3	1	1	1	8	8	8	2	2

Sparsity Beyond Zero Fill Values

Compressed Level Format

pos	0	5	13	16	23																			
coord	0	1	2	6	7	0	1	2	3	4	5	6	7	0	1	2	0	1	2	3	4	5	7	7
vals	1	1	1	5	5	6	6	6	6	1	1	1	1	3	3	3	1	1	1	8	8	8	2	2

	0	1	2	3	4	5	6	7
0	1	1	1	0	0	0	5	5
1	6	6	6	6	1	1	1	1
2	3	3	3	0	0	0	0	0
3	1	1	1	8	8	8	2	2

Sparsity Beyond Zero Fill Values

Compressed Level Format

pos	0	5	13	16	23																			
coord	0	1	2	6	7	0	1	2	3	4	5	6	7	0	1	2	0	1	2	3	4	5	7	7
vals	1	1	1	5	5	6	6	6	6	1	1	1	1	3	3	3	1	1	1	8	8	8	2	2

	0	1	2	3	4	5	6	7
0	1	1	1	0	0	0	5	5
1	6	6	6	6	1	1	1	1
2	3	3	3	0	0	0	0	0
3	1	1	1	8	8	8	2	2

Compressed Level Format with a Fill Value

pos	0	5	9	17	21																	
coord	3	4	5	6	7	0	1	2	3	0	1	2	3	4	5	6	7	3	4	5	6	7
vals	0	0	0	5	5	6	6	6	6	3	3	3	0	0	0	0	0	8	8	8	2	2
Fill	1																					

Sparsity Beyond Zero Fill Values

Compressed Level Format

pos	0	5	13	16	23																			
coord	0	1	2	6	7	0	1	2	3	4	5	6	7	0	1	2	0	1	2	3	4	5	7	7
vals	1	1	1	5	5	6	6	6	6	1	1	1	1	3	3	3	1	1	1	8	8	8	2	2

Compressed Level Format with a Fill Value

pos	0	5	9	17	21																	
coord	3	4	5	6	7	0	1	2	3	0	1	2	3	4	5	6	7	3	4	5	6	7
vals	0	0	0	5	5	6	6	6	6	3	3	3	0	0	0	0	0	8	8	8	2	2
Fill	1																					

	0	1	2	3	4	5	6	7
0	1	1	1	0	0	0	5	5
1	6	6	6	6	1	1	1	1
2	3	3	3	0	0	0	0	0
3	1	1	1	8	8	8	2	2

Run Length Encoding (RLE) Level Format

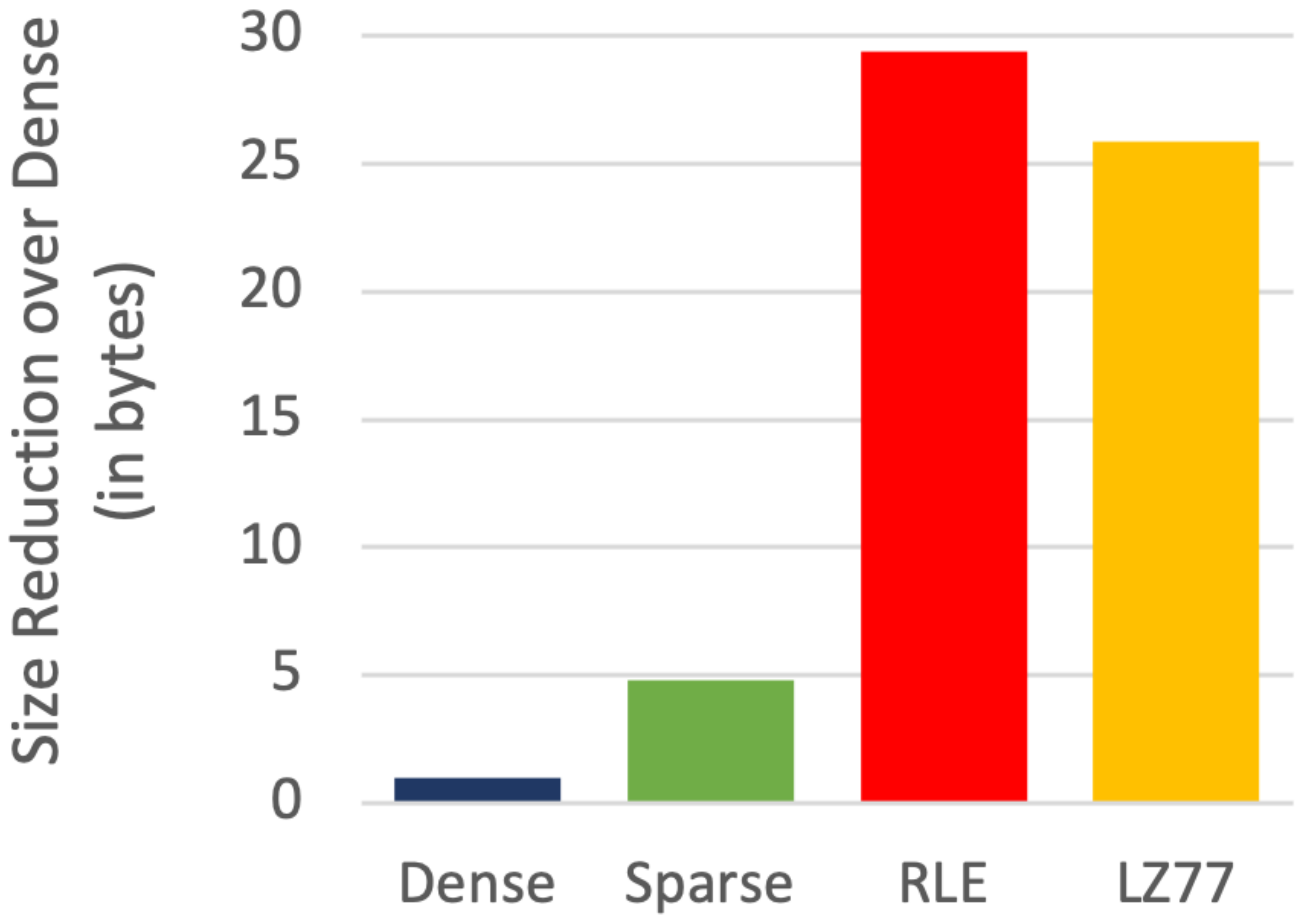
pos	0	3	5	7	9					
coord	0	3	6	0	4	0	3	0	3	6
vals	1	0	5	6	1	3	0	1	8	2

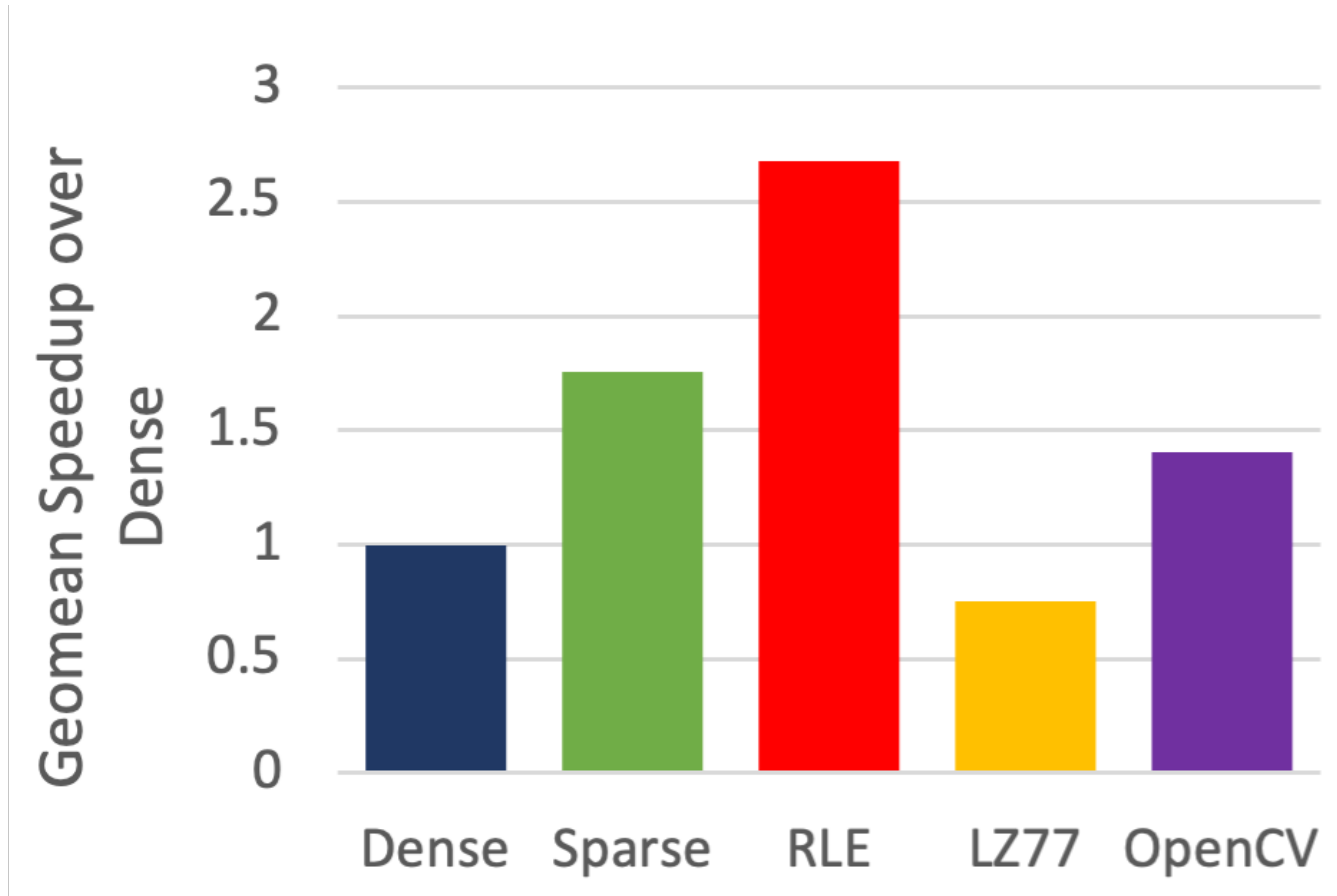
- Extension of the Compressed Format
- Last value is the Fill Value

Unifying Sparsity and Lossless Compression

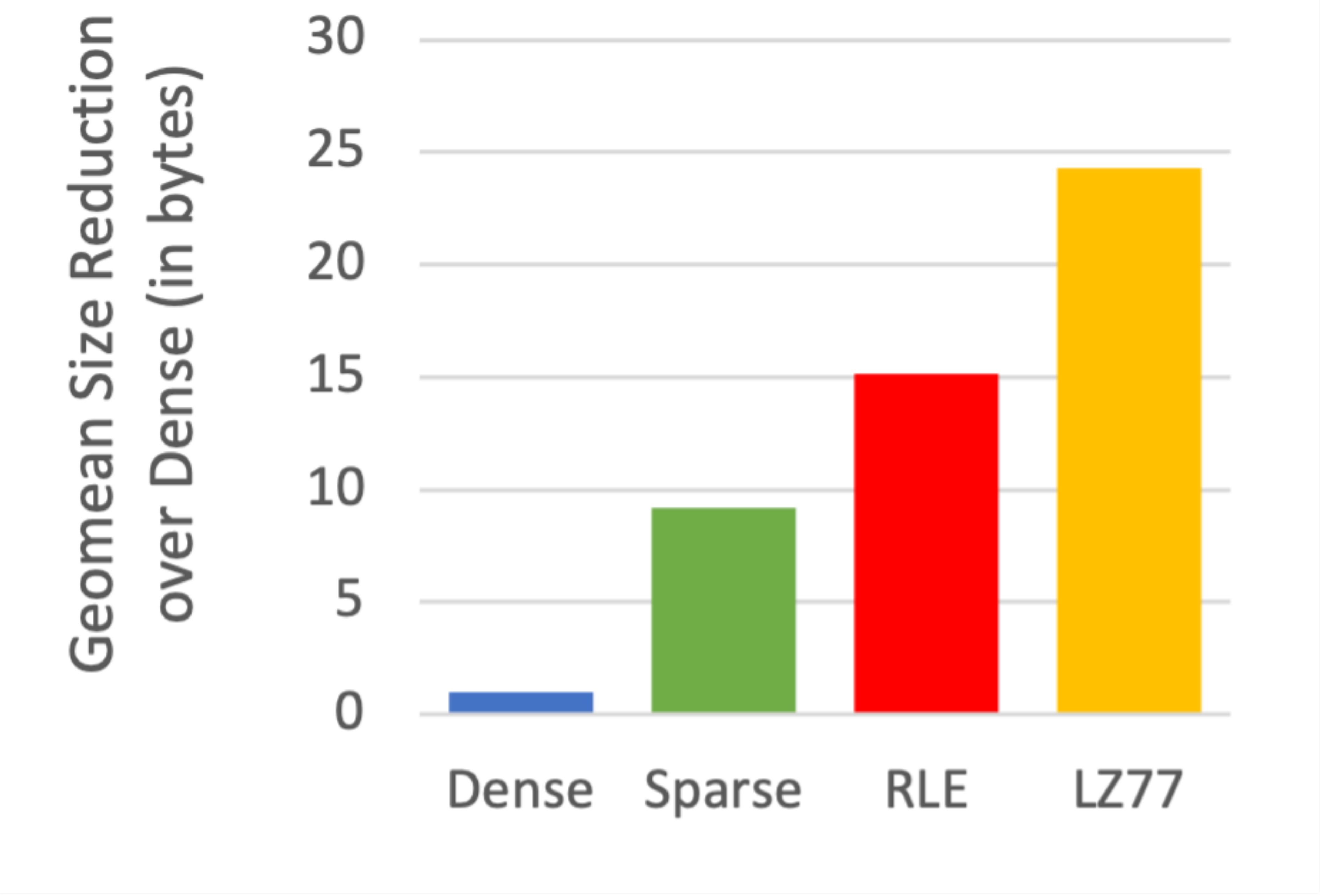
Performance Advantage In Lossless Compression

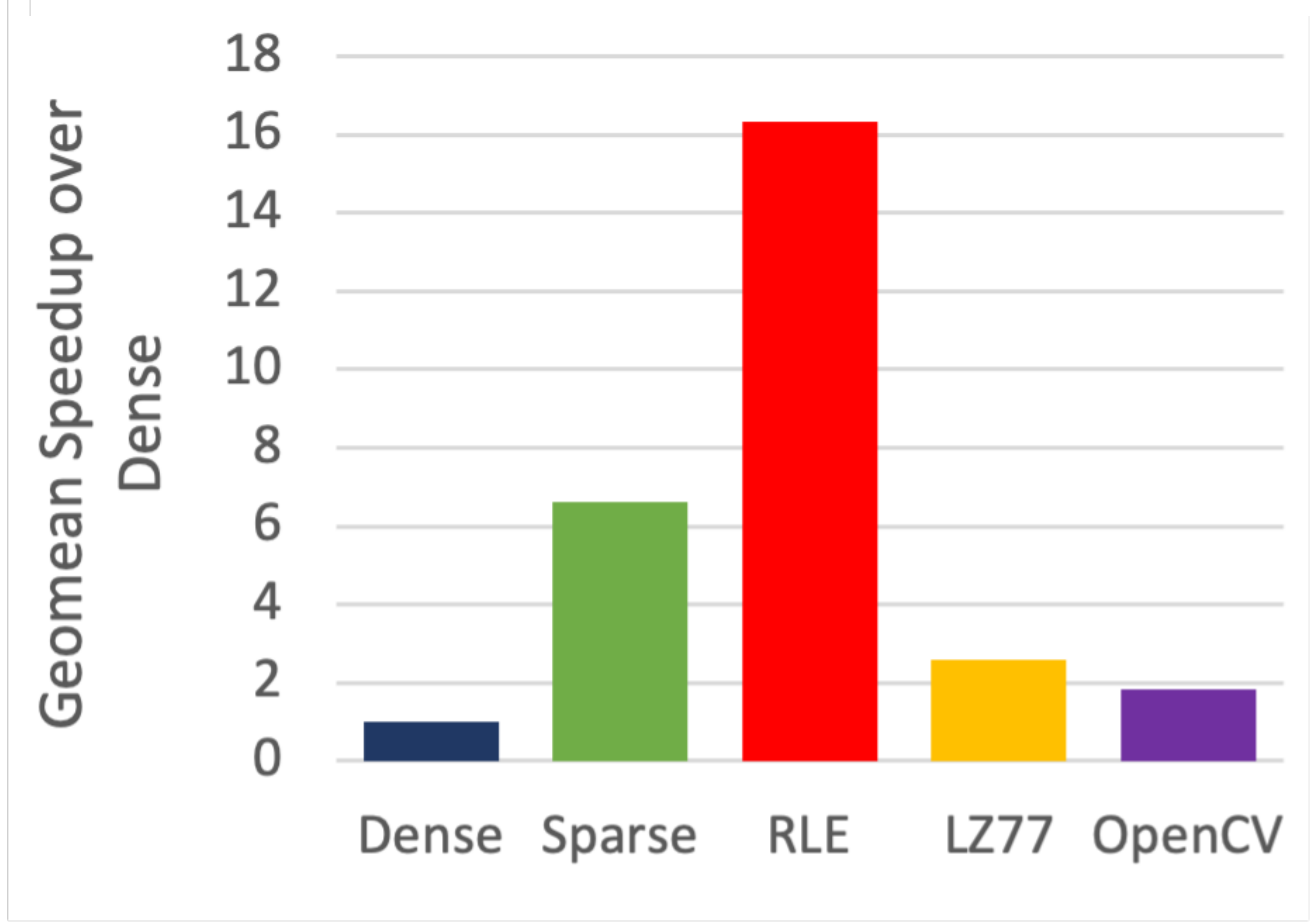
Edge Detection of MRI Image





Alpha Blending of Two Videos



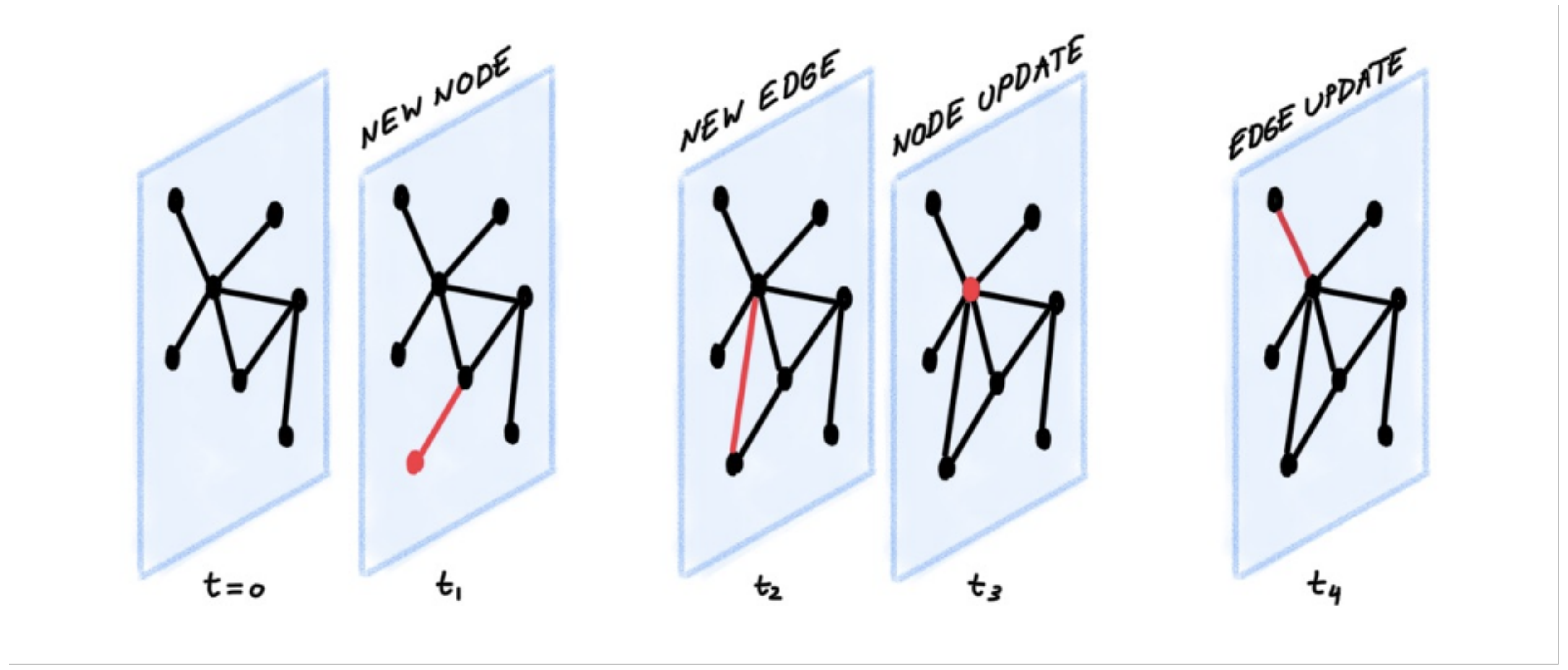


Dynamic Sparse Tensors

- All formats so far (CSR, COO, DIA, ELLPACK, RLE etc.) are static
  - Computing on them can be very fast
  - But…inserting or deleting an element can be (asymptotically) slow

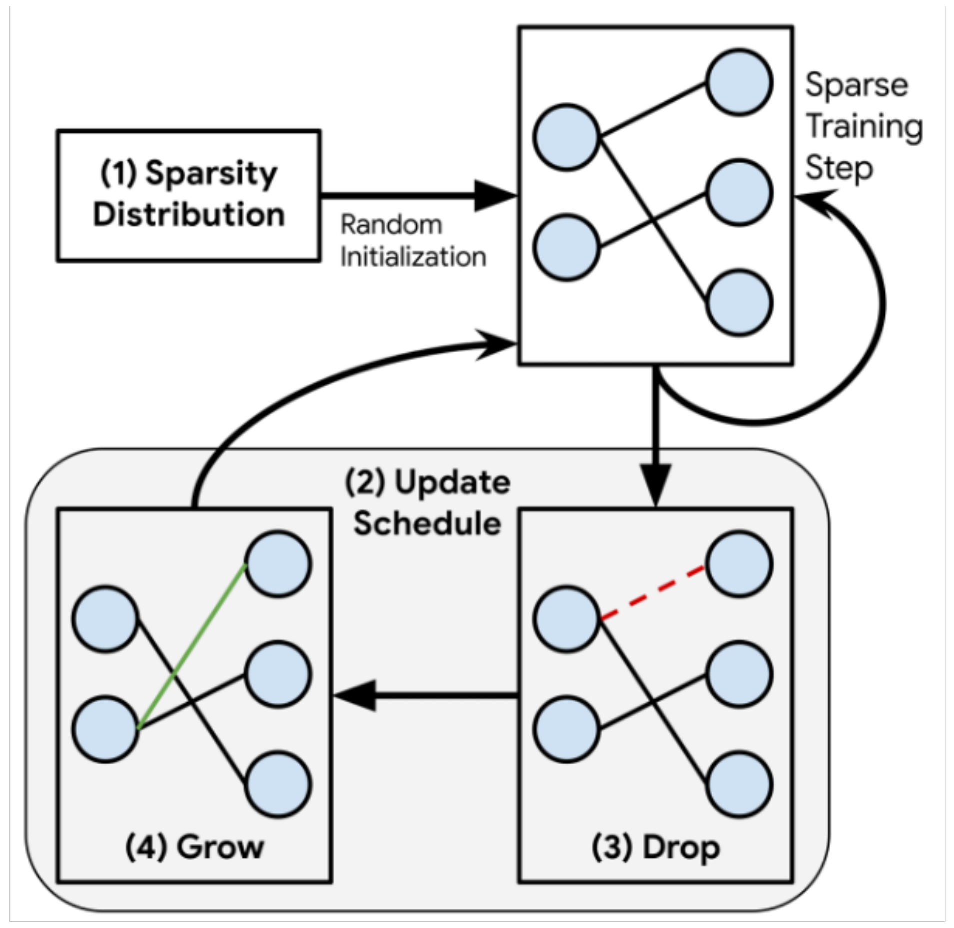
- Many real world Applications are dynamic

Dynamic Graph Processing



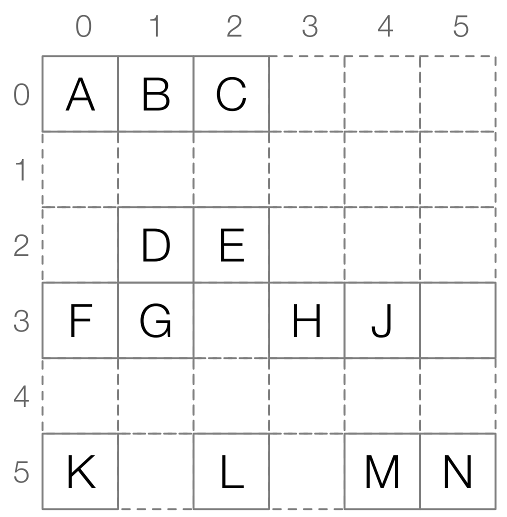
https://blog.twitter.com/engineering/en_us/topics/insights/2021/temporal-graph-networks

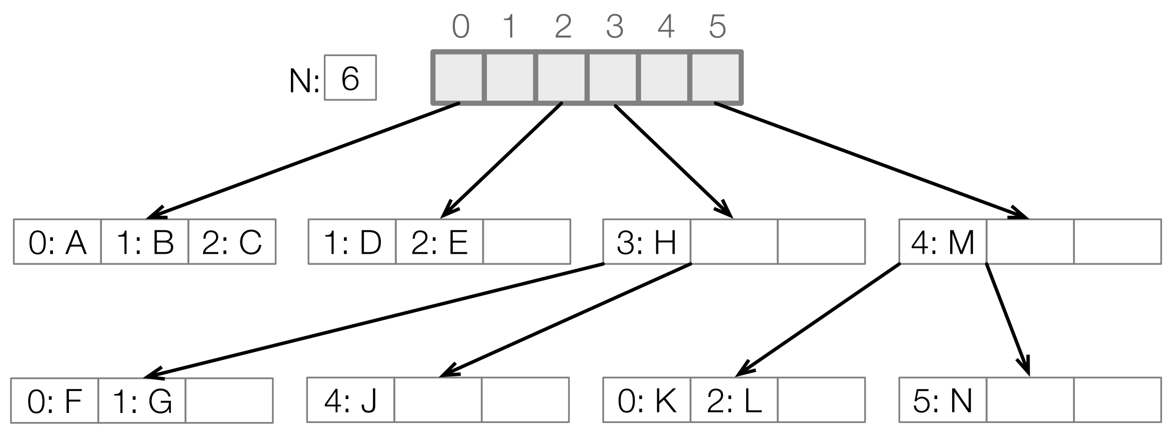
Sparse Neural Network Training

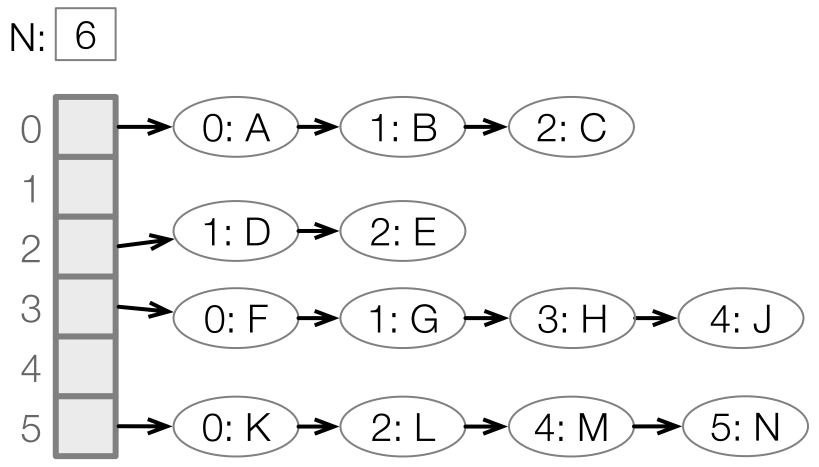


Dynamic Sparse Tensors

- Need pointer-based, recursive data structures
- Novel Node Schema Language
  - Automatically generate the data structures
  - Automatically Generate the code for iteration

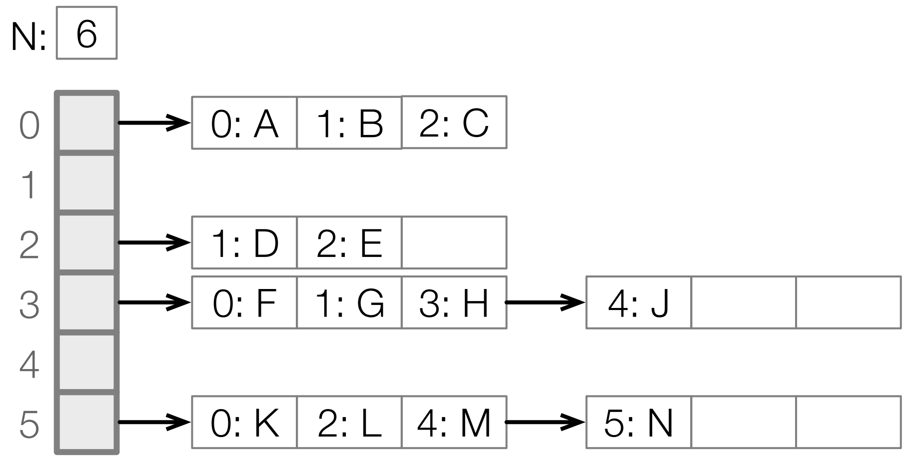






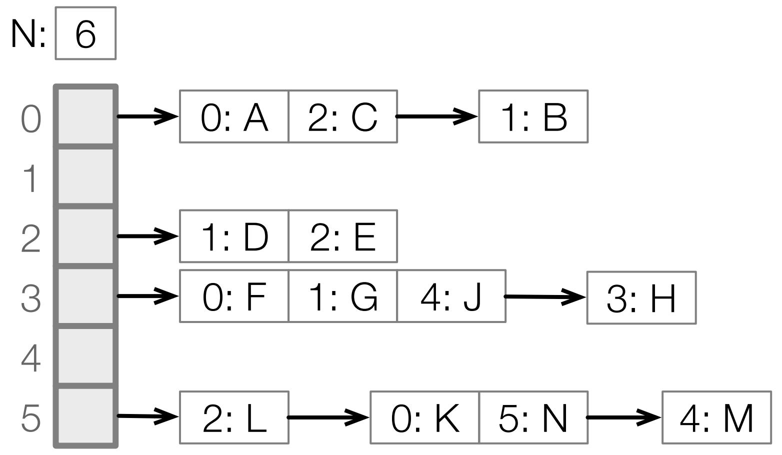

```
def list {
  e : elem nonempty
  n : list
  seq = {e}, n
}

def list_head {
  h : list
}
```

Linked List



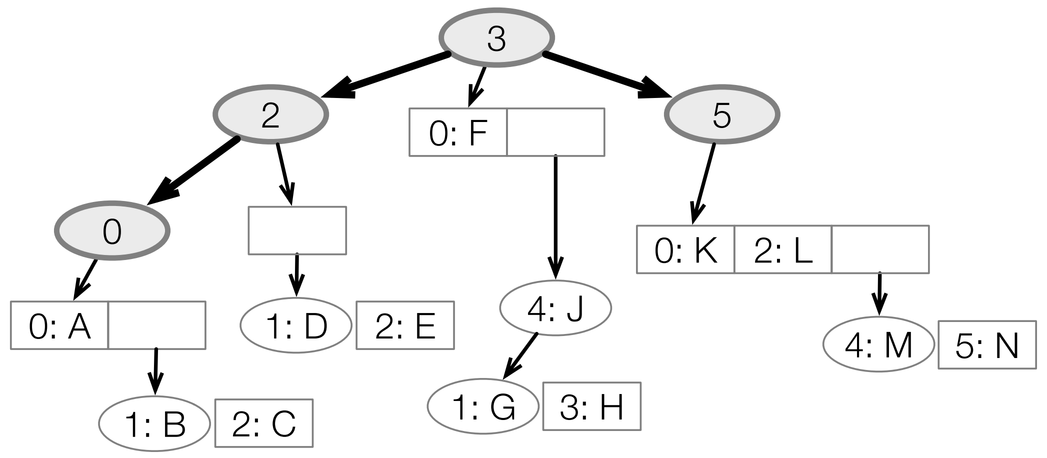

```
def blist {
  e : elem[B] nonempty
  n : blist
  B : size in [0, 3]
  seq = {e}, n
}

def blist_head {
  h : blist
}
```

Block Linked List




```
def vblist {
  e : elem[B] nonempty
  n : vblist
  B : size
  seq = {e}, n
}

def vblist_head {
  h : vblist
}
```

Variable Block Linked List




```
def ctree {
  h : elem nonempty
  t : elem[N] nonempty
  l : ctree
  r : ctree
  N : size
  seq = l, h, {t}, r
}

def prefix {
  e : elem[N] nonempty
  r : ctree
  N : size
  seq = {e}, r
}
```

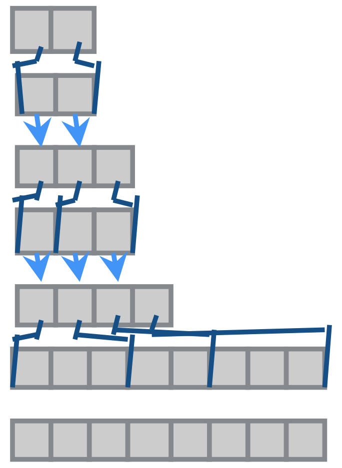
C-Tree

```
def supertype btree

def btree_internal : btree {
  e  : elem[B] nonempty
  c  : btree[B] nonempty
  cl : btree nonempty
  B  : size in [1, 3]
  seq = {c, e}, cl
}

def btree_leaf : btree {
  e : elem[B] nonempty
  B : size in [1, 3]
  seq = {e}
}

def btree_root {
  r : btree
}
```

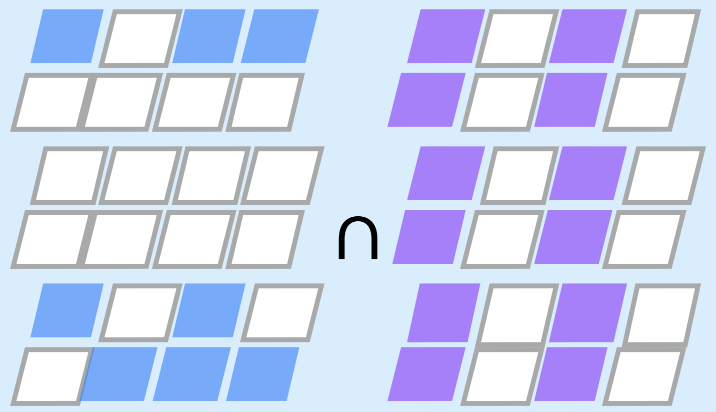
B-Tree

Challenges Of Sparse Array Compilation

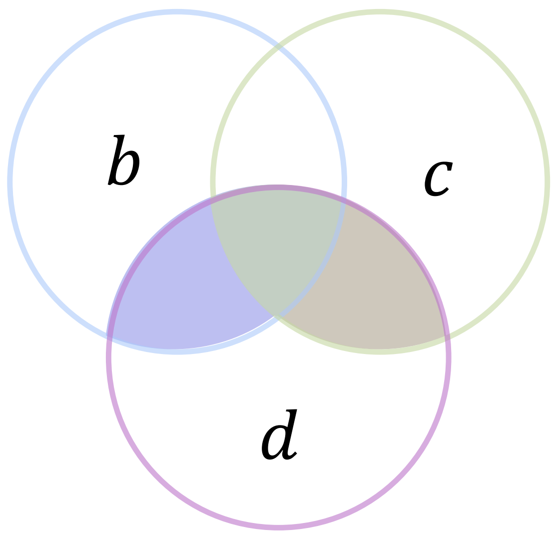
Irregular Data Structures



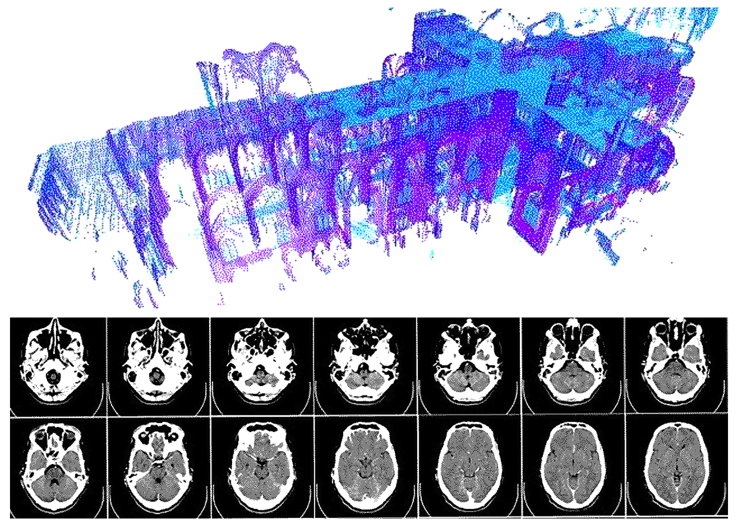
Sparse Iteration with limited O(1) access



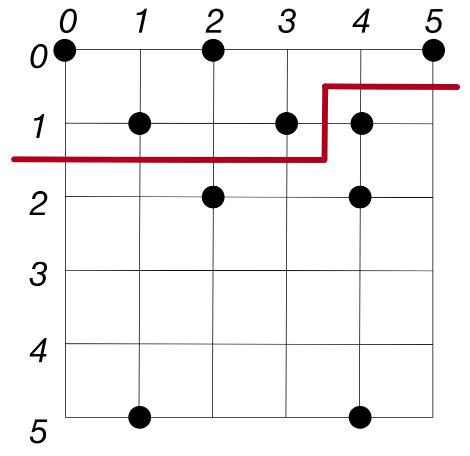
Avoid wasted work and iterations

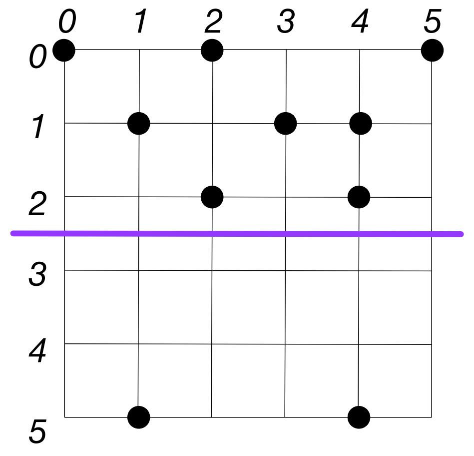


Coiteration Over Complex Data



Optimize Parallelism and Locality

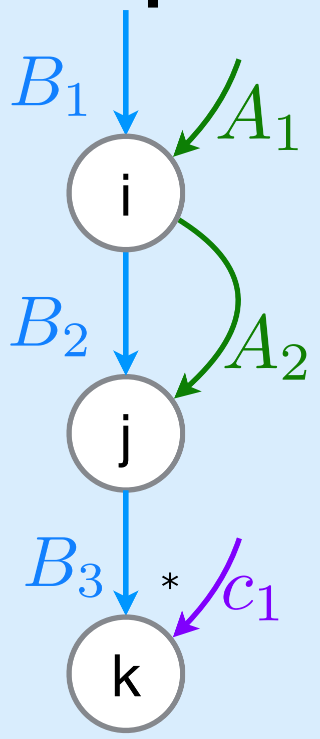




Format Language

CSF
Dense
Compressed
Compressed

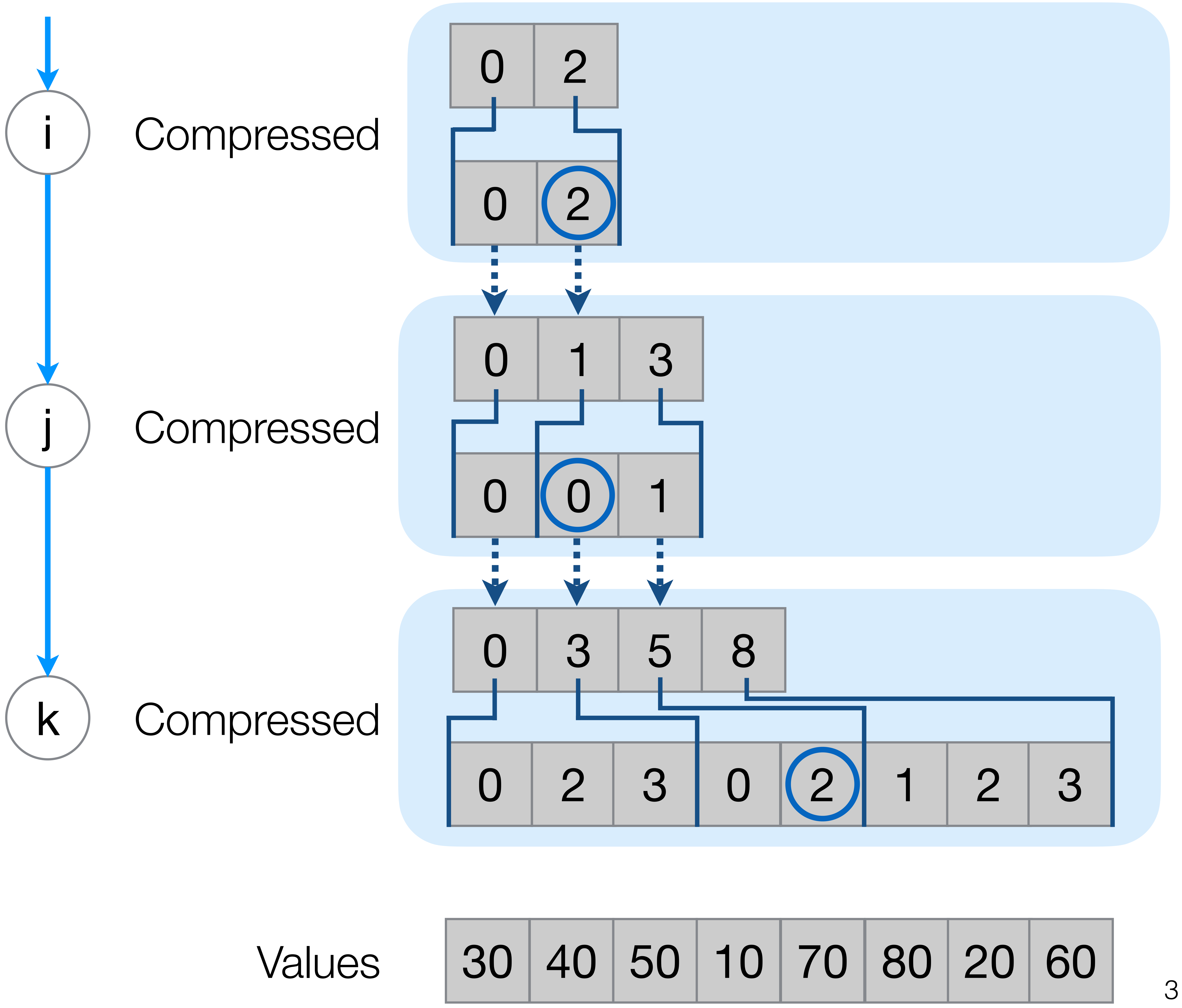
Sparse Iteration Graphs



Compressed Storge Formats

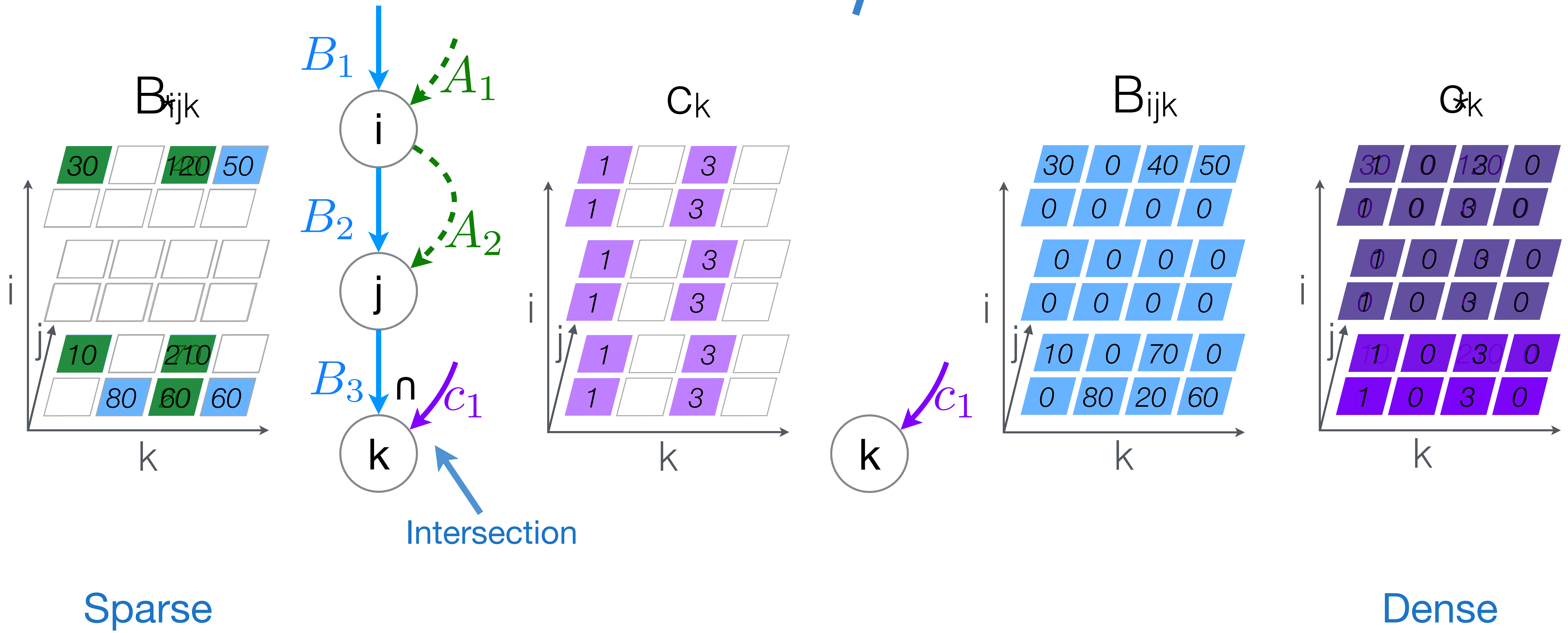
No O(1) random access

`locate(2, 0, 2)`



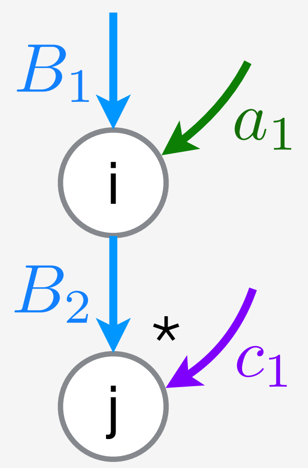
Sparse Iteration Spaces And Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * c_k$$

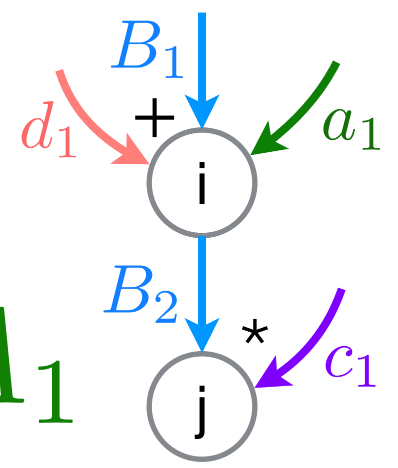


Sparse Iteration Graph Examples

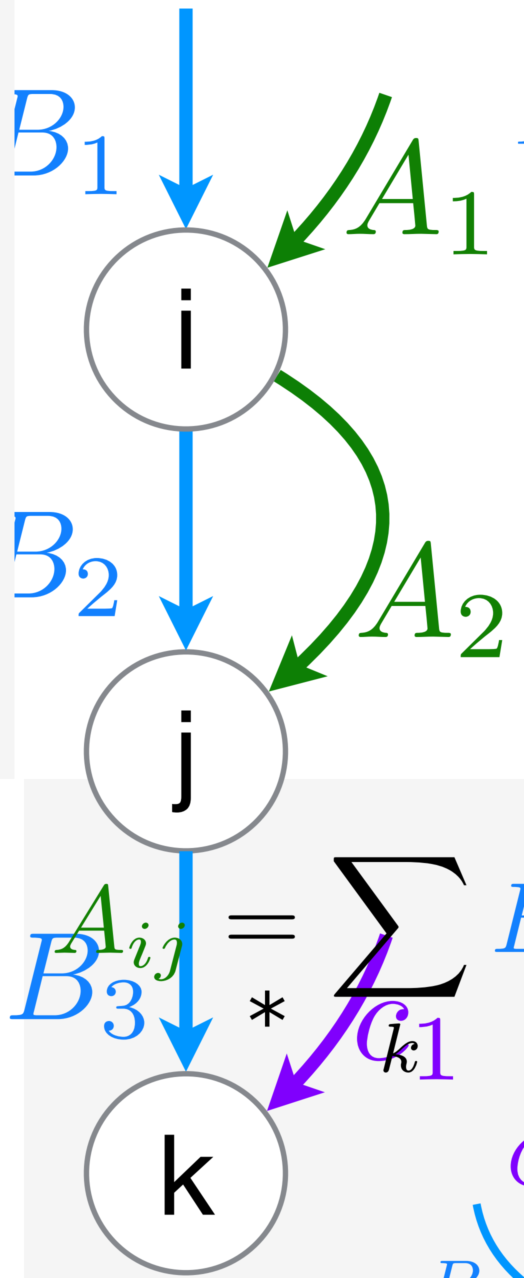
$$a_i = \sum_j B_{ij}\, c_j$$



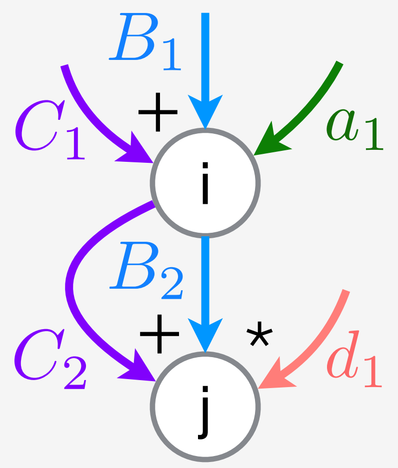
$$a_i = \sum_j \alpha B_{ij}\, c_j + \beta d_i$$



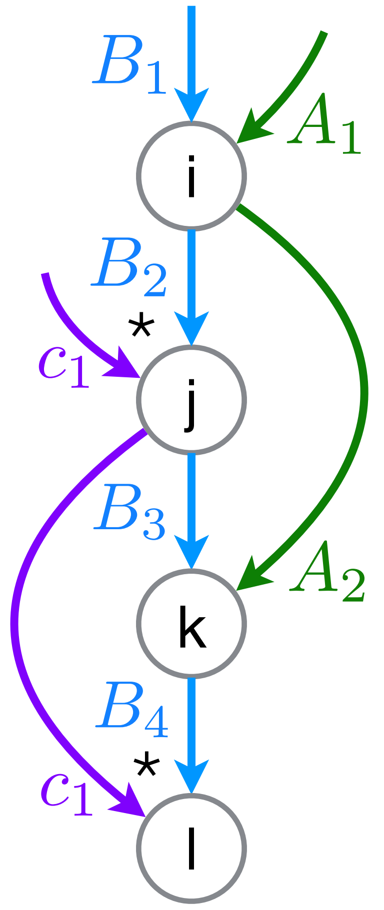
$$A_{ij} = \sum_k B_{ijk} * c_k$$



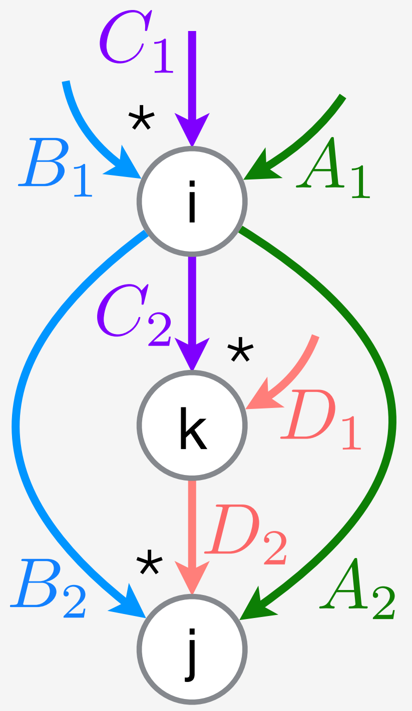
$$a_i = \sum_j (B_{ij} + C_{ij})\, d_j$$



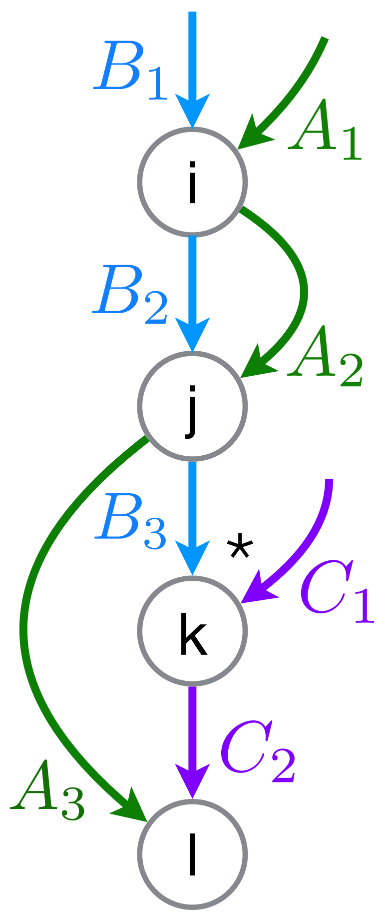
$$a_{ik} = \sum_j \sum_l B_{ijkl}\, c_{jl}$$



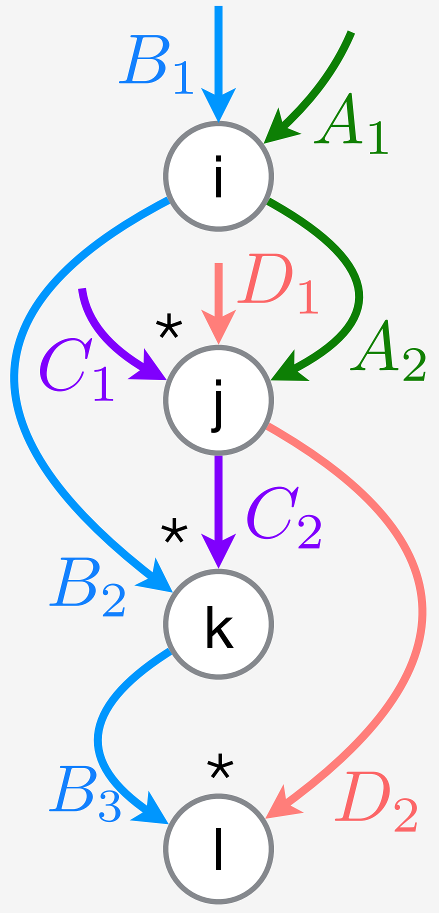
$$A_{ij} = \sum_k B_{ij}(C_{ik} D_{kj})$$



$$A_{ijl} = \sum_k B_{ijk} C_{lk}$$



$$A_{ij} = \sum_k \sum_l B_{ikl} C_{kj} D_{lj}$$



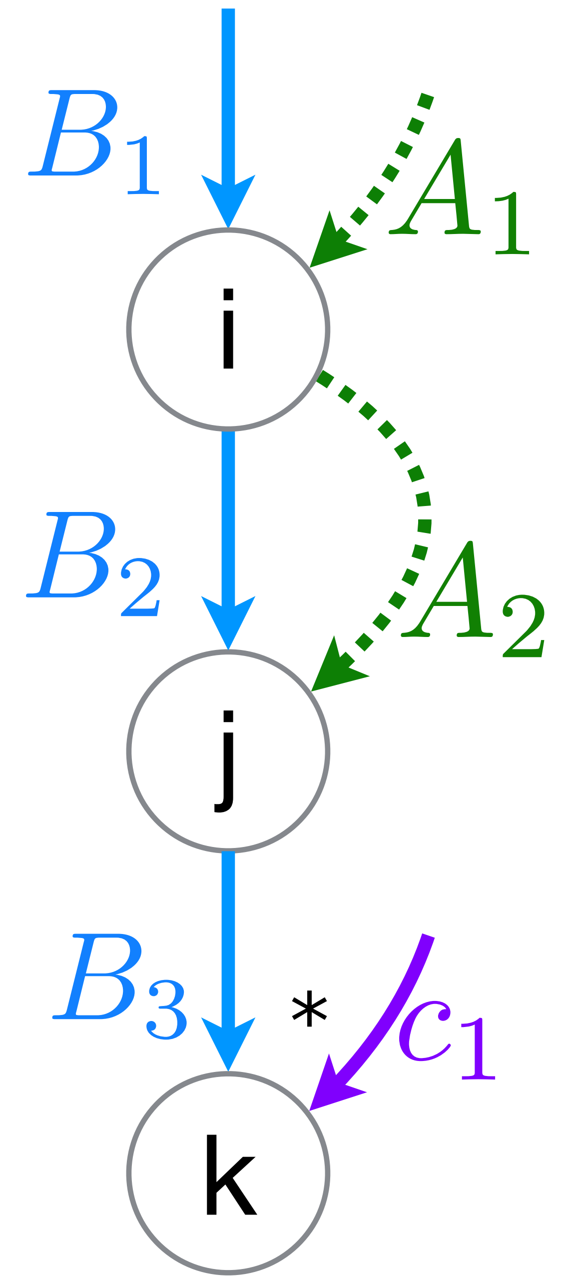
Code Generation From Iteration Graph

$$A_{ij} = \sum_k B_{ijk} * c_k$$

A
Dense
Dense

B
Dense
Compressed
Compressed

c
Compressed



dense dimension

compressed dimension

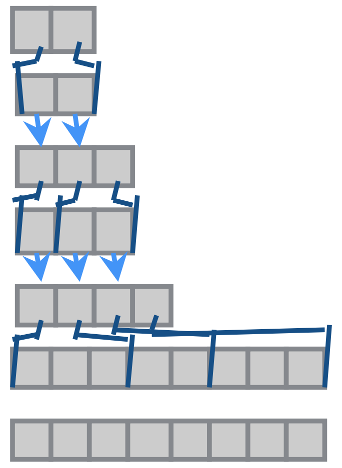
merged dimensions

compute statement

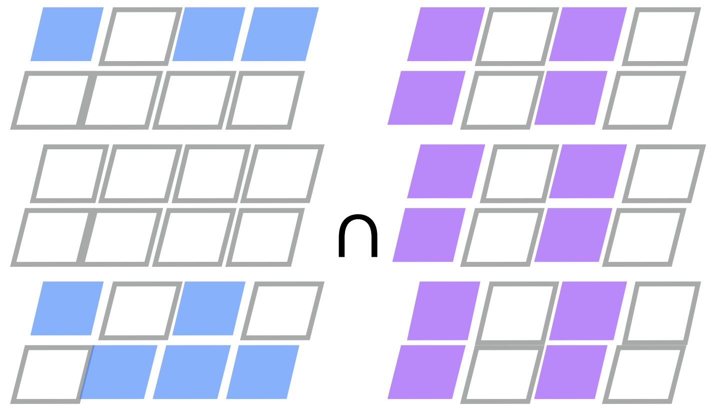
```
for (int i = 0; i < m; i++) {
  for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1 + 1]; pB2++) {
    int j = B2_crd[pB2];
    int pA2 = (i * n) + j;
    int pB3 = B3_pos[pB2];
    int pc1 = c1_pos[0];
    while (pB3 < B3_pos[pB2 + 1] && pc1 < c1_pos[1]) {
      int kB = B3_crd[pB3];
      int kc = c1_crd[pc1];
      int k = min(kB, kc);
      if (kB == k && kc == k) {
        A[pA2] += B[pB3] * c[pc1];
      }
      if (kB == k) pB3++;
      if (kc == k) pc1++;
    }
  }
}
```

Challenges Of Sparse Array Compilation

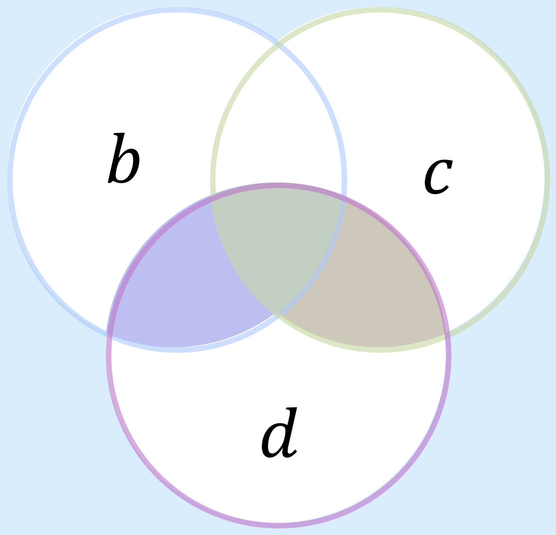
Irregular Data Structures



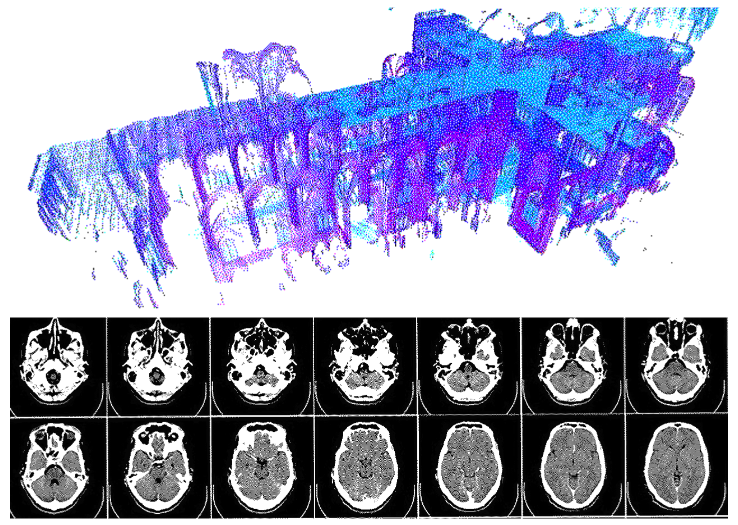
Sparse Iteration with limited O(1) access



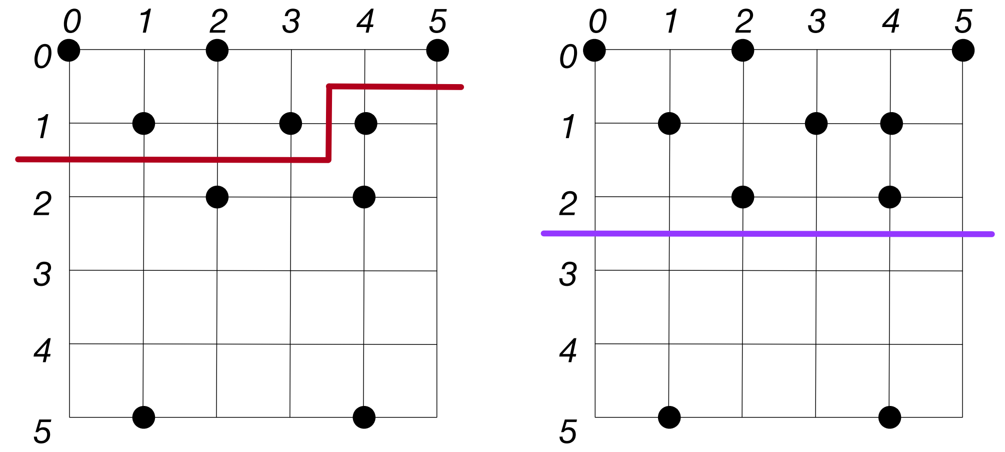
Avoid wasted work and iterations



Coiteration Over Complex Data



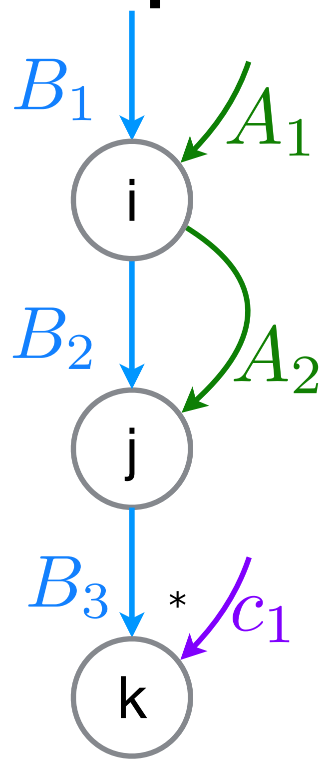
Optimize Parallelism and Locality



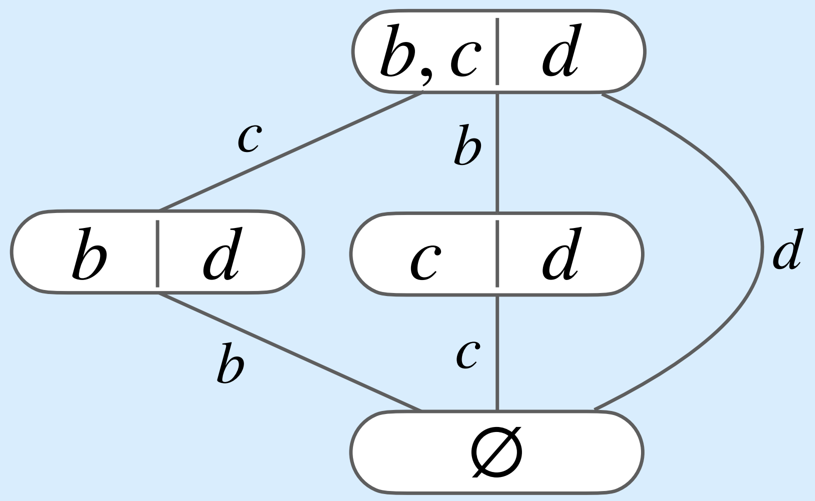
Format Language

CSF
Dense
Compressed
Compressed

Sparse Iteration Graphs



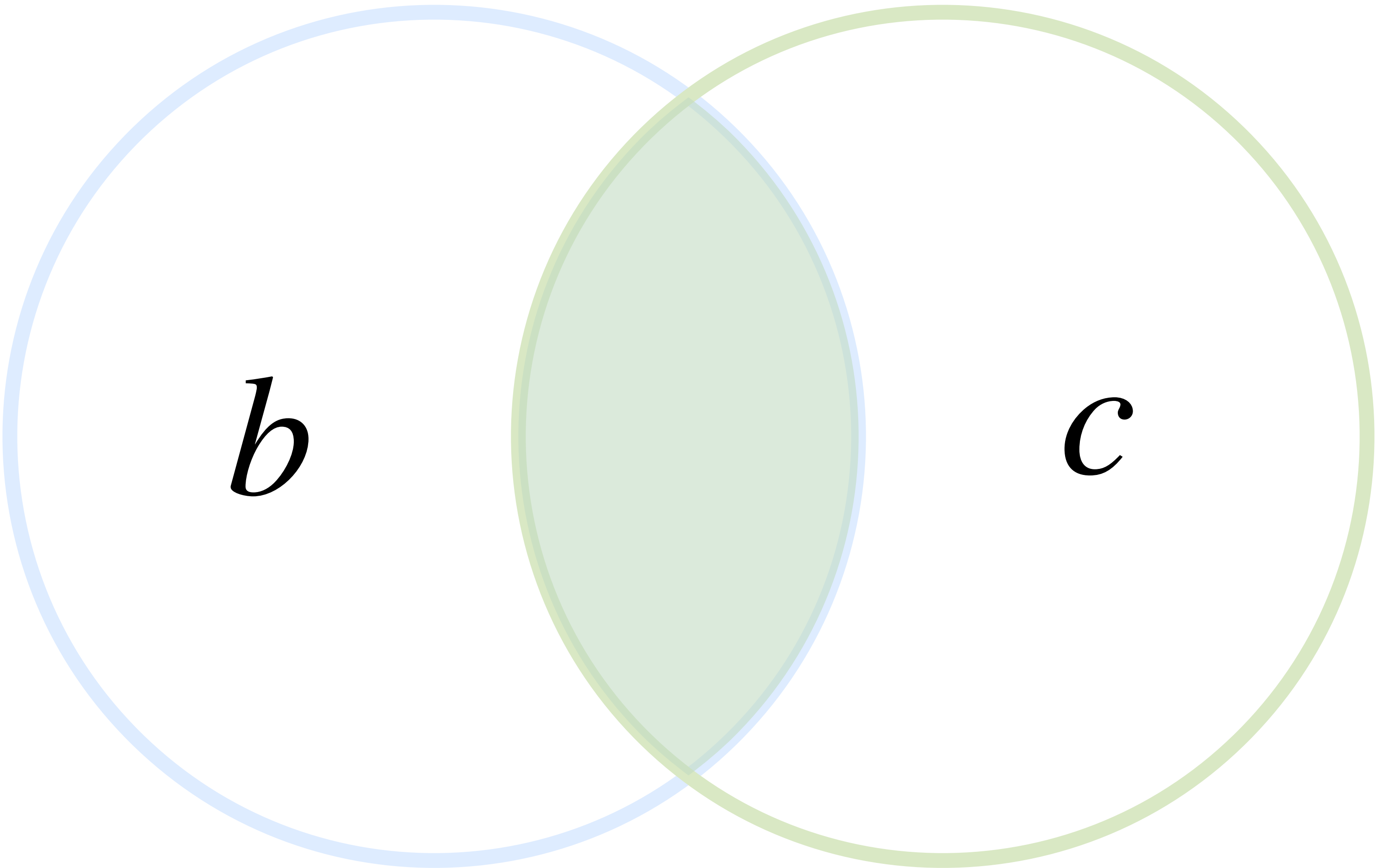
Coiteration Code Generation

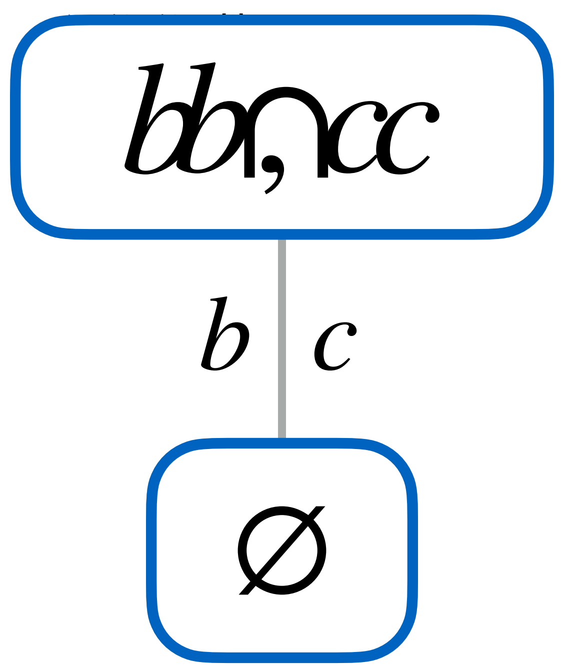


Merge Lattice For Multiplications

$$a_i = b_i c_i$$

Multiplication requires intersection



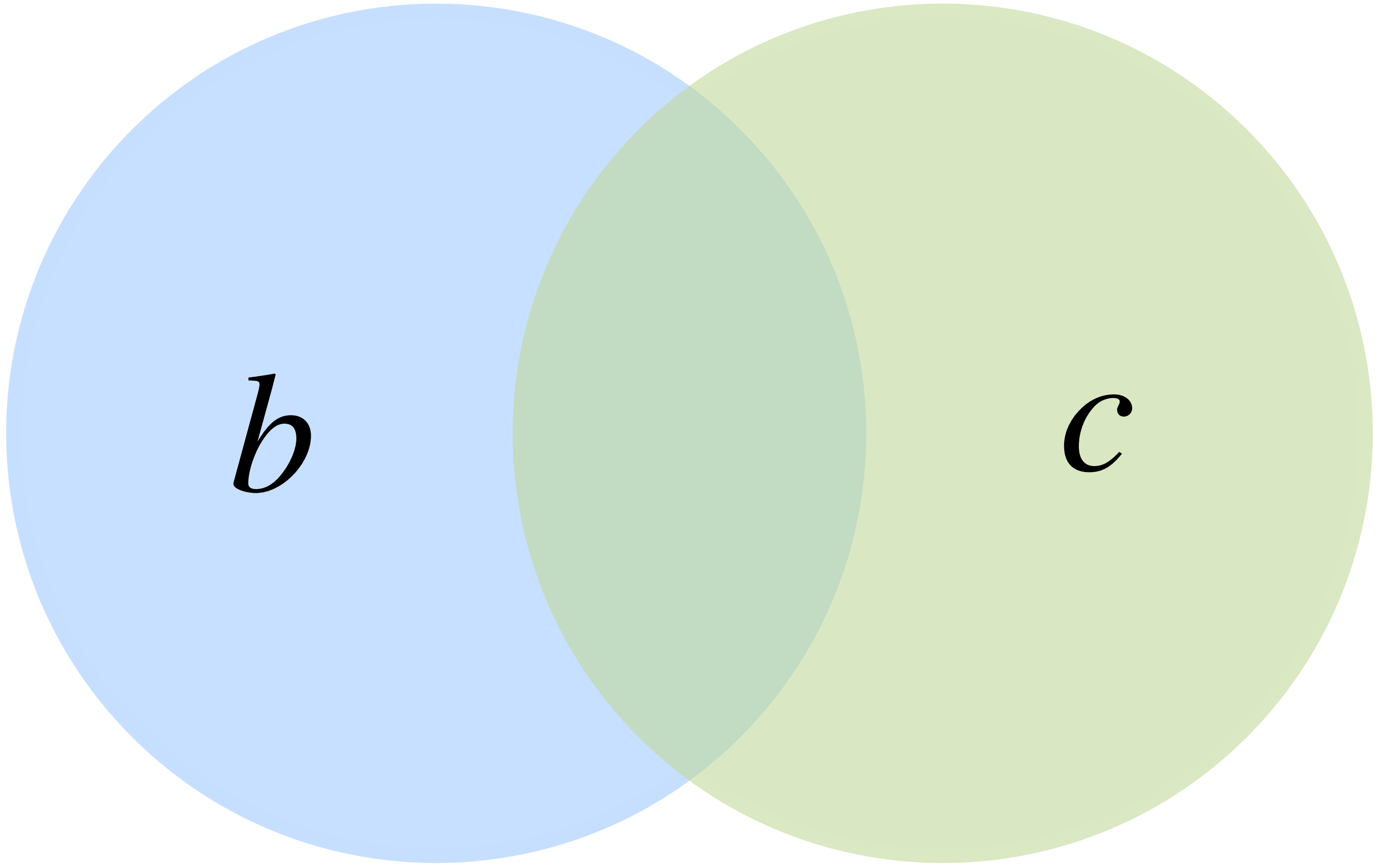


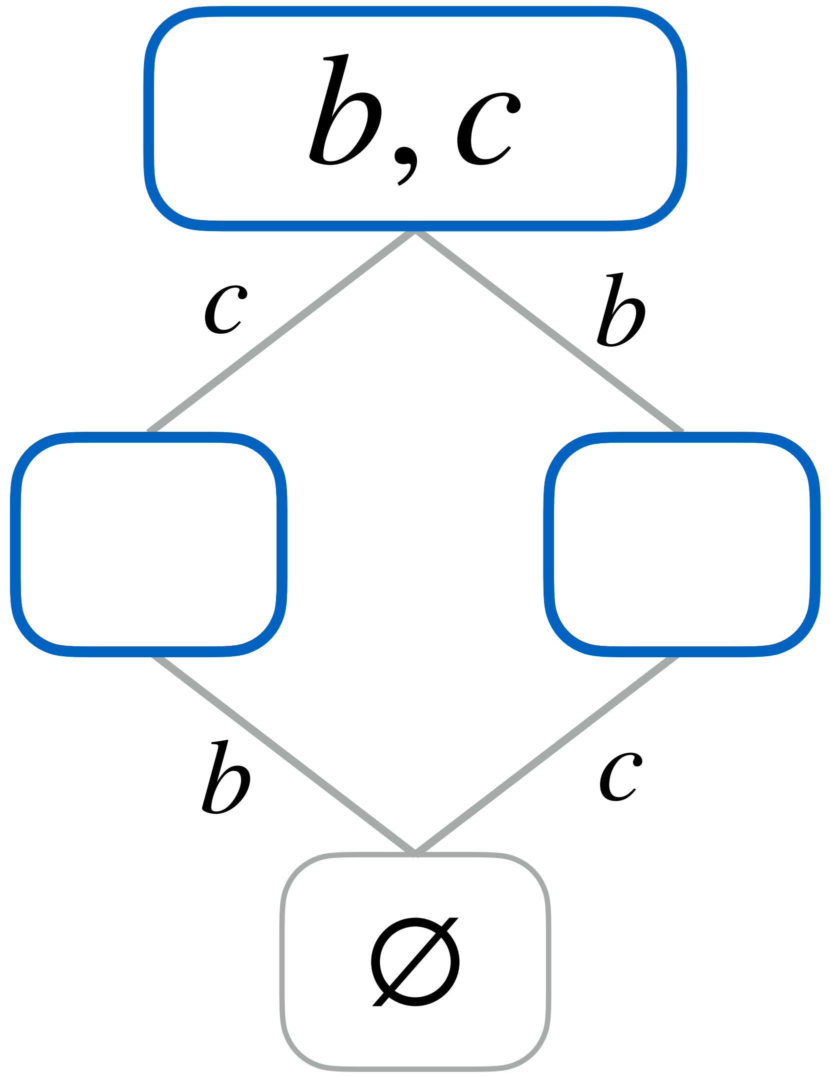

```c
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    a[i] = b[pb1] * c[pc1];
  }
  if (ib == i) pb1++;
  if (ic == i) pc1++;
}
```

Merge Lattice For Additions

$$a_i = b_i + c_i$$

Addition requires union

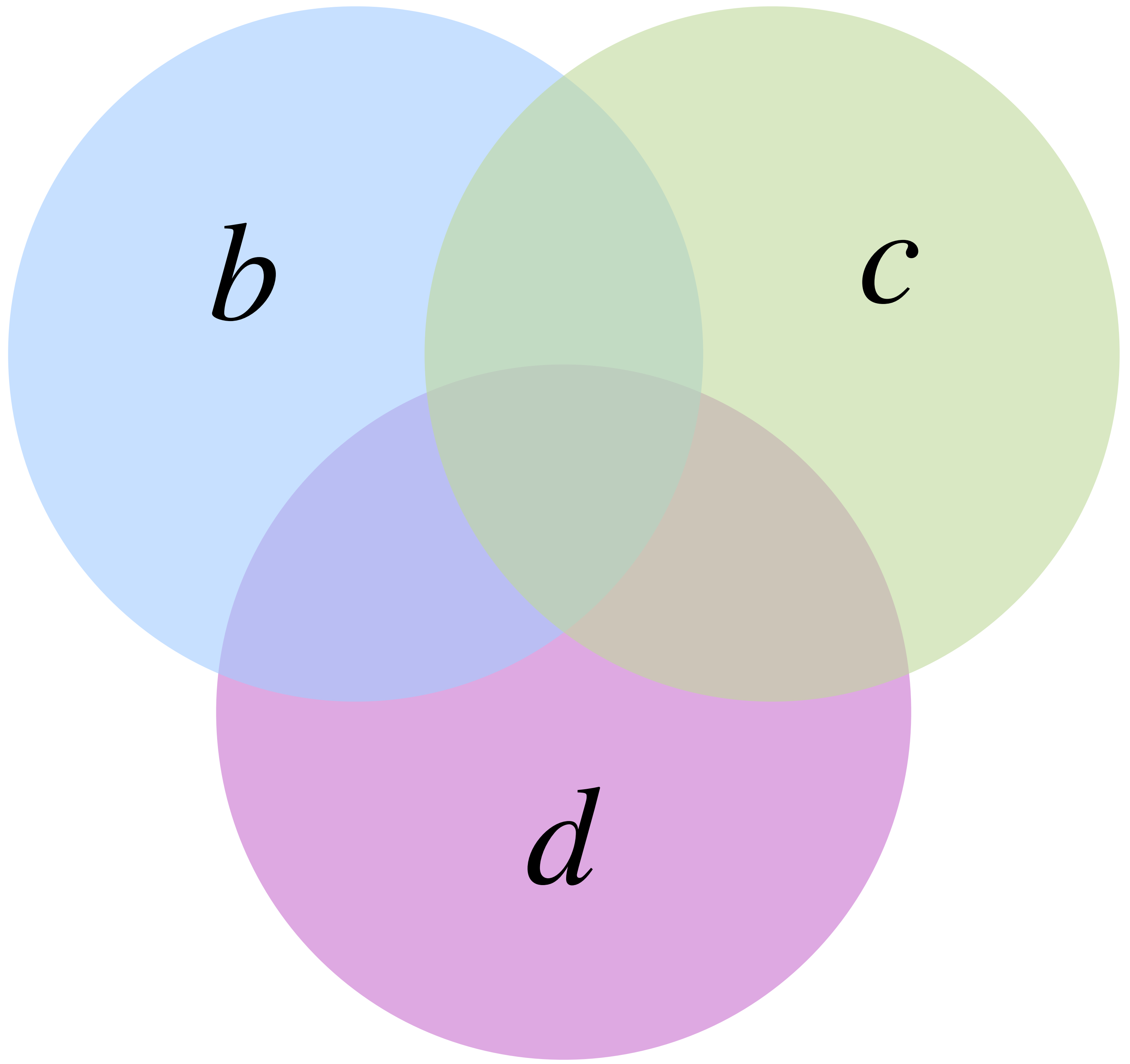


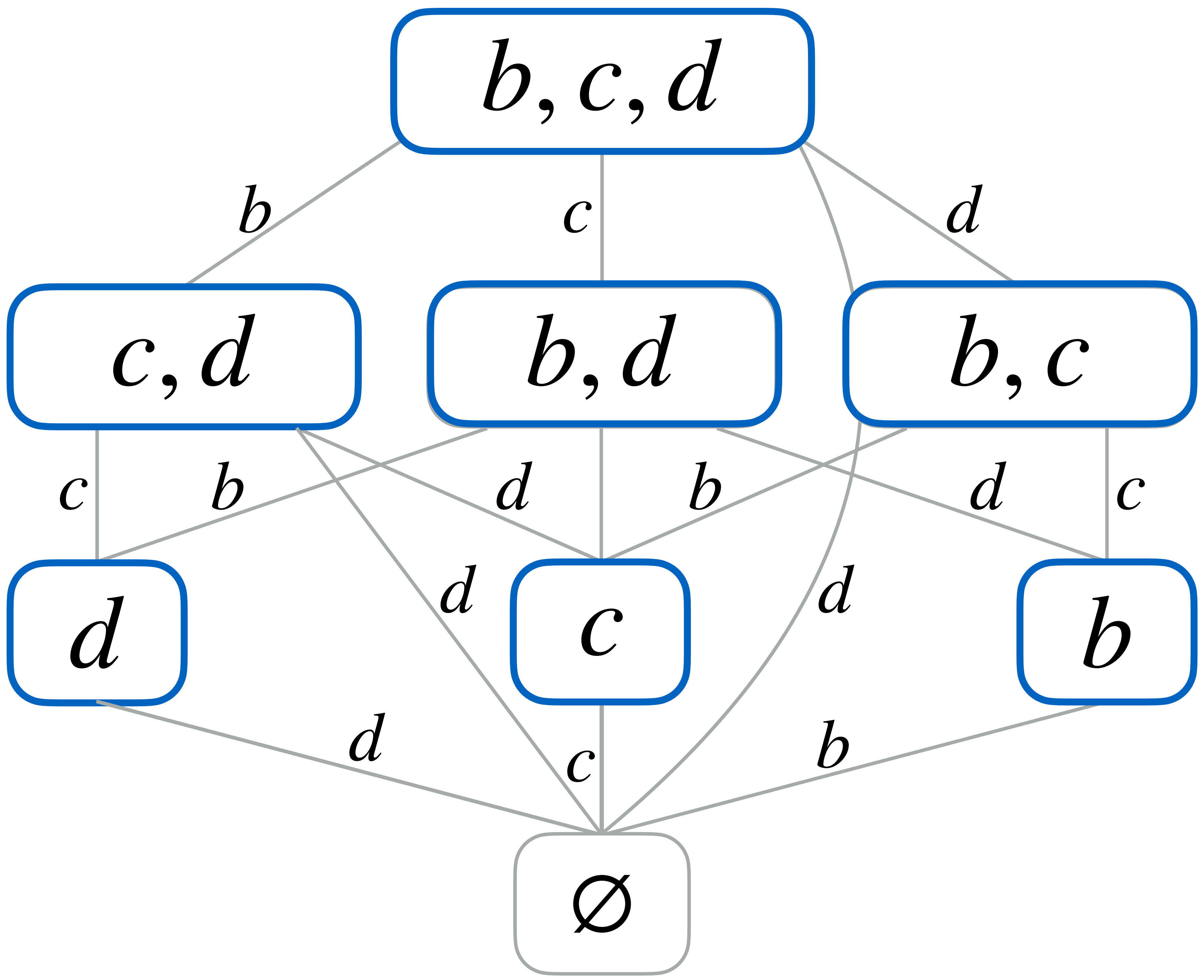



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    a[i] = b[pb1] + c[pc1];
  }
  else if (ib == i) {
    a[i] = b[pb1];
  }
  else {
    a[i] = c[pc1];
  }
  if (ib == i) pb1++;
  if (ic == i) pc1++;
}

while (pb1 < b1_pos[1]) {
  int i = b1_crd[pb1];
  a[i] = b[pb1++];
}

while (pc1 < c1_pos[1]) {
  int i = c1_crd[pc1];
  a[i] = c[pc1++];
}
```

Merge Lattice For A Compound Expression

$$a_i = b_i + c_i + d_i$$

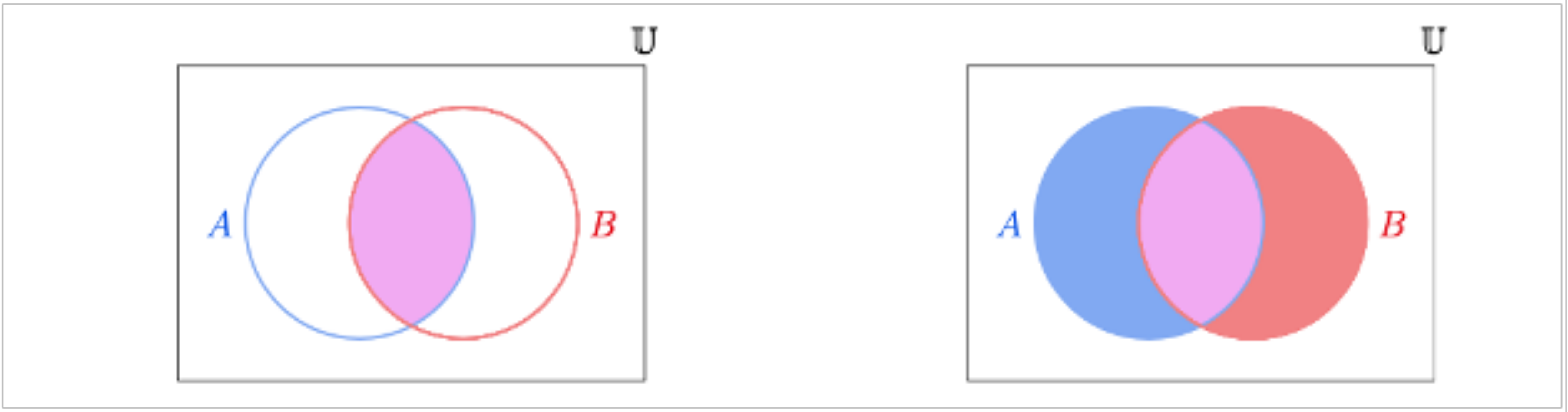


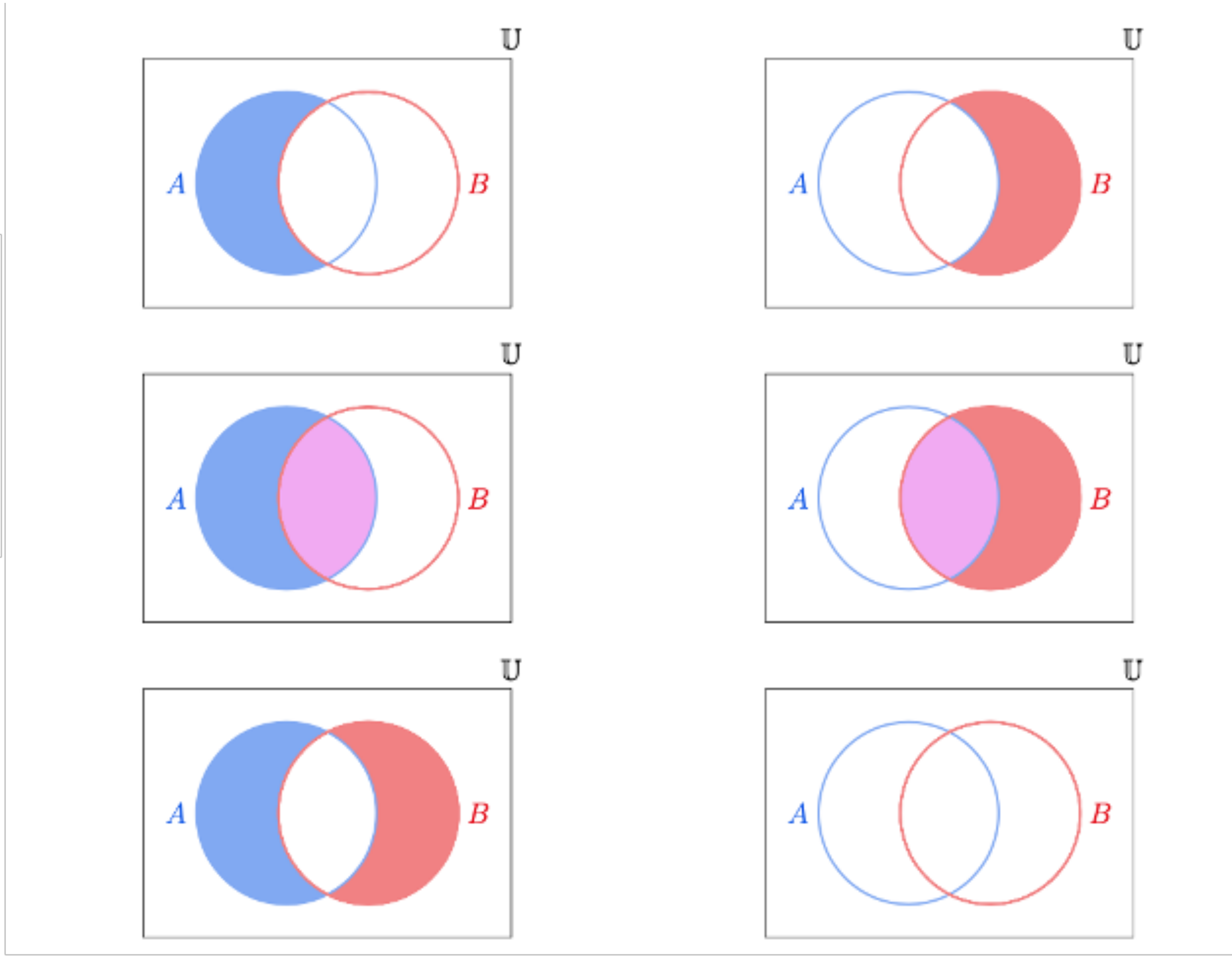



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int id = d1_crd[pd1];
  int i = min(ib, ic, id);
  if (ib == i && ic == i && id == i) {
    a[i] = b[pb1] + c[pc1] + d[pd1];
  }
  else if (ib == i && id == i) {
    a[i] = b[pb1] + d[pd1];
  }
  else if (ic == i && id == i) {
    a[i] = c[pc1] + d[pd1];
  }
  else if (ib == i && ic == i) {
    a[i] = b[pb1] + c[pc1];
  }
  else if (ib == i) {
    a[i] = b[pb1];
  }
  else if (ic == i) {
    a[i] = c[pc1];
  }
  else {
    a[i] = d[pd1];
  }
  if (ib == i) pb1++;
  if (ic == i) pc1++;
  if (id == i) pd1++;
}
```

```
while (pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
  int ic = c1_crd[pc1];
  int id = d1_crd[pd1];
  int i = min(ic, id);
  if (ic == i && id == i) {
    a[i] = c[pc1] + d[pd1];
  }
  else if (ic == i) {
    a[i] = c[pc1];
  }
  else {
    a[i] = d[pd1];
  }
  if (ic == i) pc1++;
  if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1]) {
  int ib = b1_crd[pb1];
  int id = d1_crd[pd1];
  int i = min(ib, id);
  if (ib == i && id == i) {
    a[i] = b[pb1] + d[pd1];
  }
  else if (ib == i) {
    a[i] = b[pb1];
  }
  else {
    a[i] = d[pd1];
  }
  if (ib == i) pb1++;
  if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    a[i] = b[pb1] + c[pc1];
  }
  else if (ib == i) {
    a[i] = b[pb1];
  }
  else {
    a[i] = c[pc1];
  }
  if (ib == i) pb1++;
  if (ic == i) pc1++;
}

  while (pd1 < d1_pos[1]) {
    int id = d1_crd[pd1];
    a[id] = d[pd1];
    pd1++;
  }

  while (pc1 < c1_pos[1]) {
    int ic = c1_crd[pc1];
    a[ic] = c[pc1];
    pc1++;
  }

  while (pb1 < b1_pos[1]) {
    int ib = b1_crd[pb1];
    a[ib] = b[pb1];
    pb1++;
  }
```

Beyond Multiply And Add

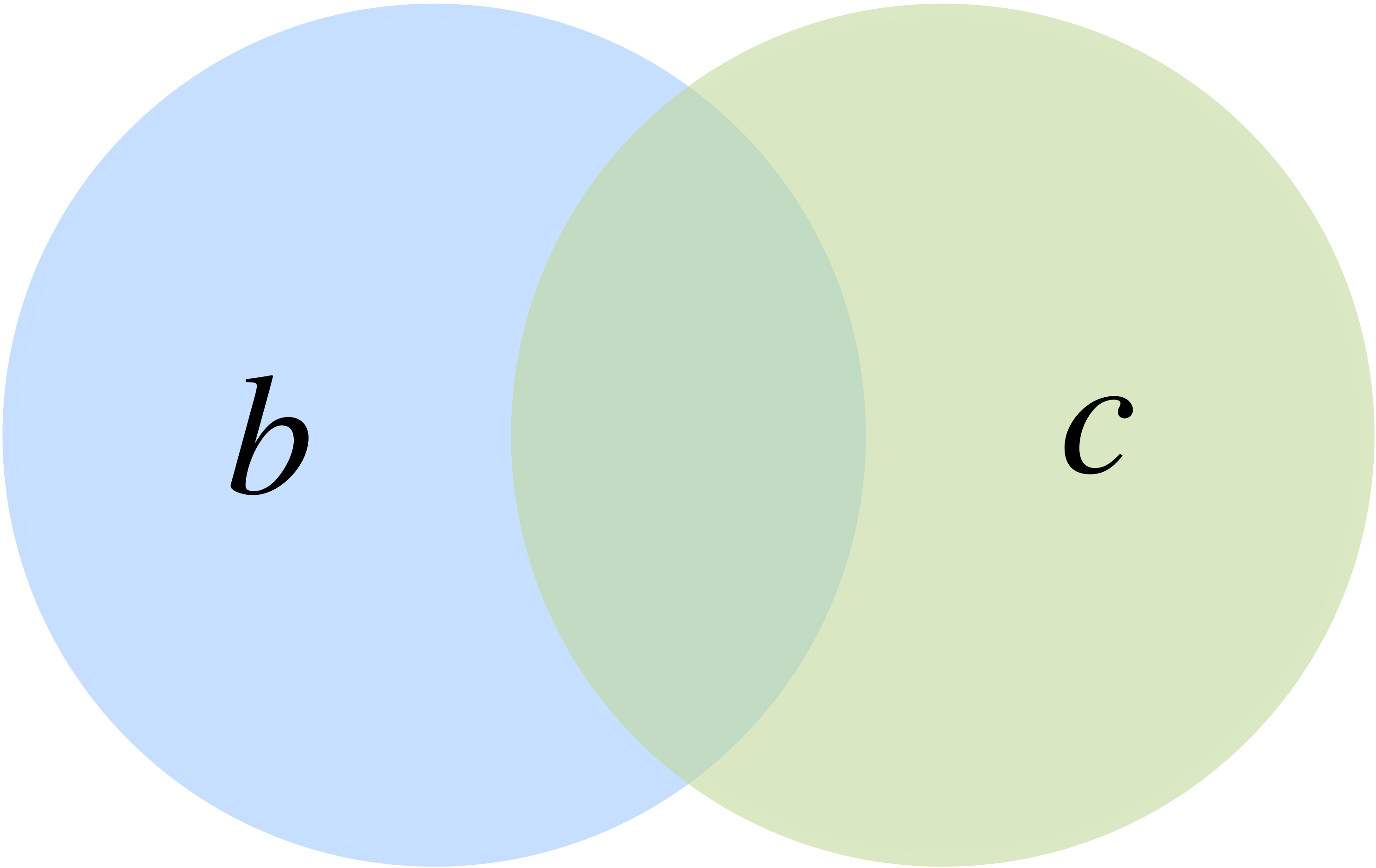


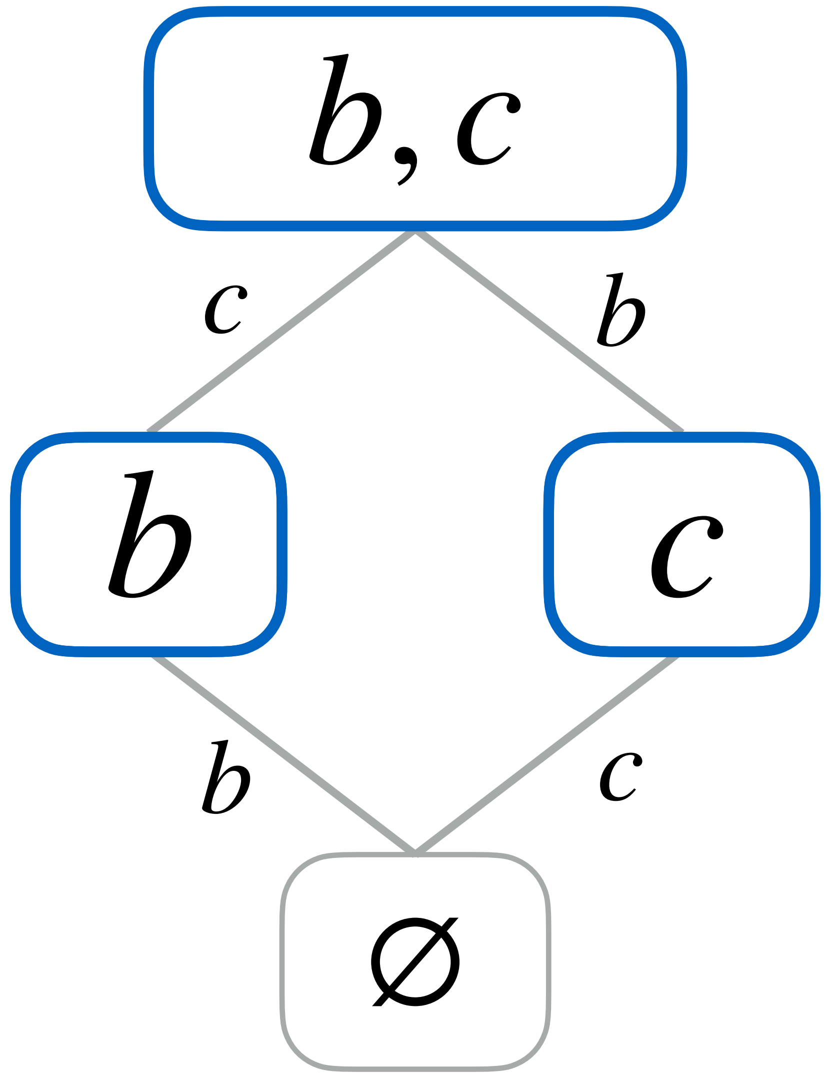


Merge Lattice For Additions

$$a_i = b_i + c_i$$

Addition requires union

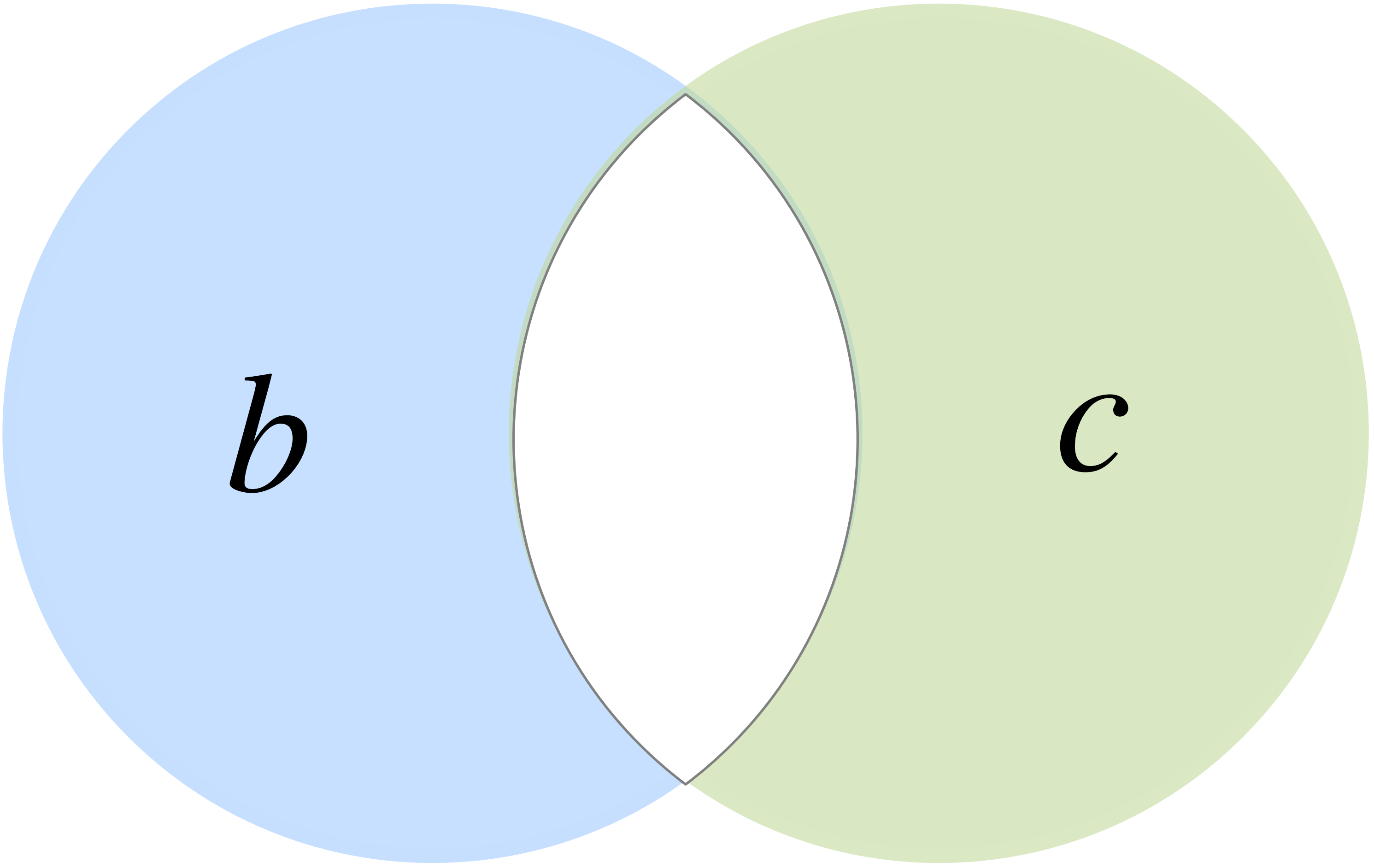


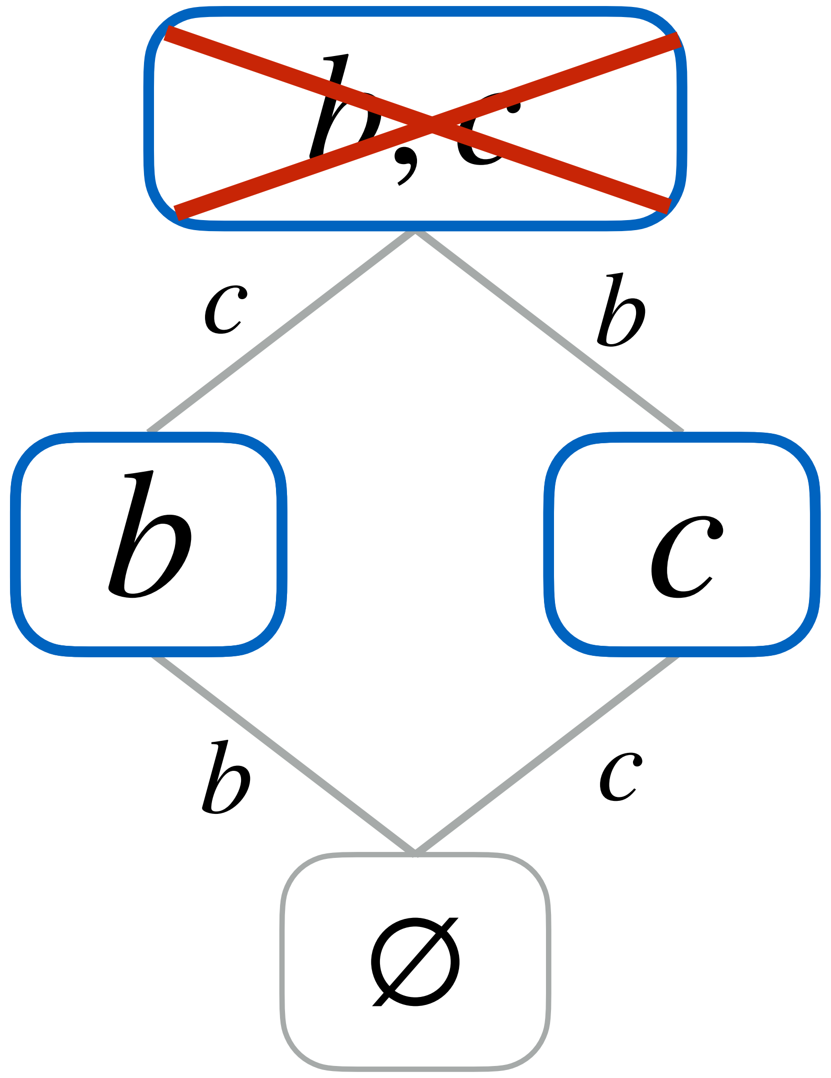



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    a[i] = b[pb1] + c[pc1];
  }
  else if (ib == i) {
    a[i] = b[pb1];
  }
  else {
    a[i] = c[pc1];
  }
  if (ib == i) pb1++;
  if (ic == i) pc1++;
}

while (pb1 < b1_pos[1]) {
  int i = b1_crd[pb1];
  a[i] = b[pb1++];
}

while (pc1 < c1_pos[1]) {
  int i = c1_crd[pc1];
  a[i] = c[pc1++];
}
```

Merge Lattice For Xor

$$a_i = b_i \bigoplus c_i$$






```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    a[i] = b[pb1] + c[pc1];
  }
  else if (ib == i) {
    a[i] = b[pb1];
  }
  else {
    a[i] = c[pc1];
  }
  if (ib == i) pb1++;
  if (ic == i) pc1++;
}

while (pb1 < b1_pos[1]) {
  int i = b1_crd[pb1];
  a[i] = b[pb1++];
}

while (pc1 < c1_pos[1]) {
  int i = c1_crd[pc1];
  a[i] = c[pc1++];
}
```

User Defined Functions

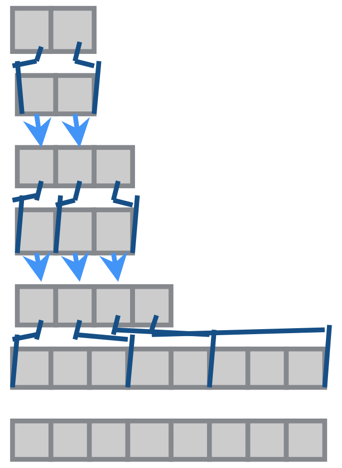
```
def bitwise_and(x, y):
  x, y => {
            return x & y;
          }

  properties:
      commutative
      annihilator=0
```

```
def gcd(x, y):
  x, 0 => { return abs(x); }
  0, y => { return abs(y); }
  x, y => { x = abs(x);
            y = abs(y);
            while (x != 0) {
                int t = x;
                x = y % x;
                y = t;
            }
            return y;
          }

  iteration_space:
```

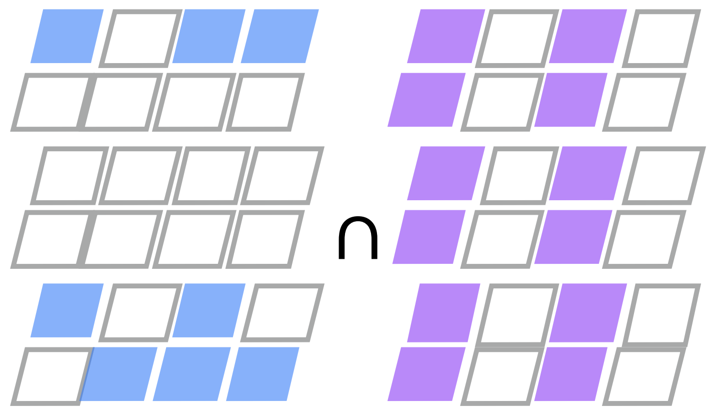
$$\{x \neq 0\} \cup \{y \neq 0\}$$

Challenges Of Sparse Array Compilation

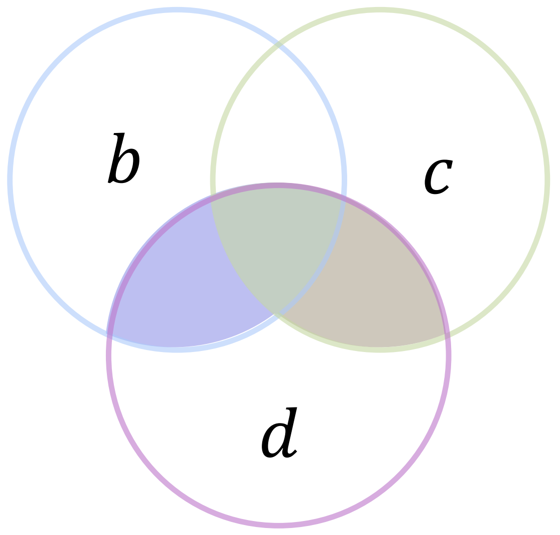
Irregular Data Structures



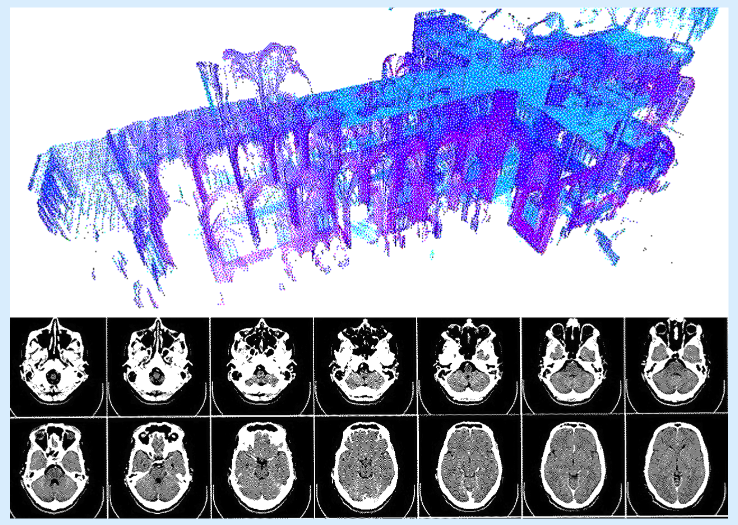
Sparse Iteration with limited O(1) access



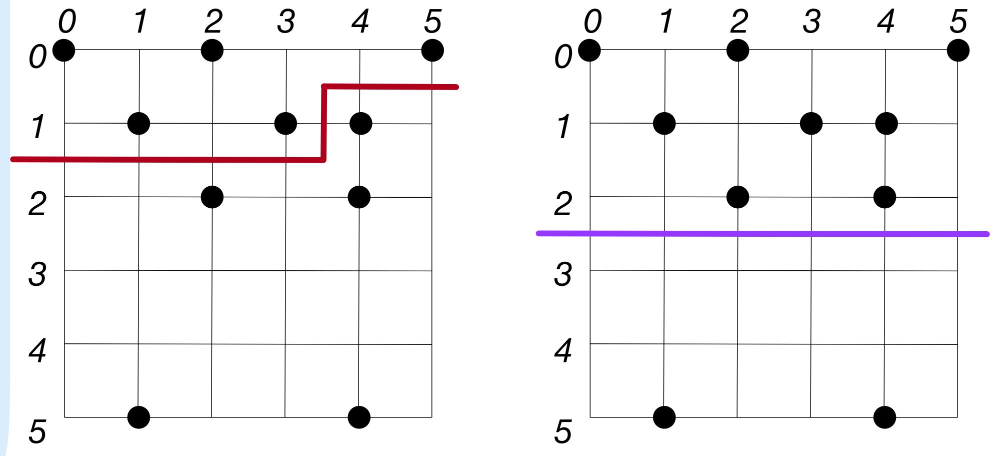
Avoid wasted work and iterations



Coiteration Over Complex Data



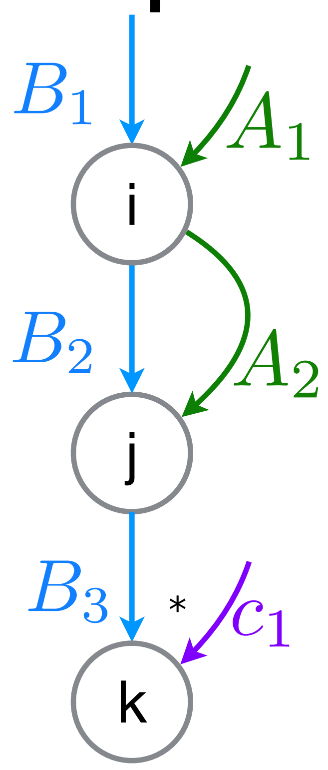
Optimize Parallelism and Locality



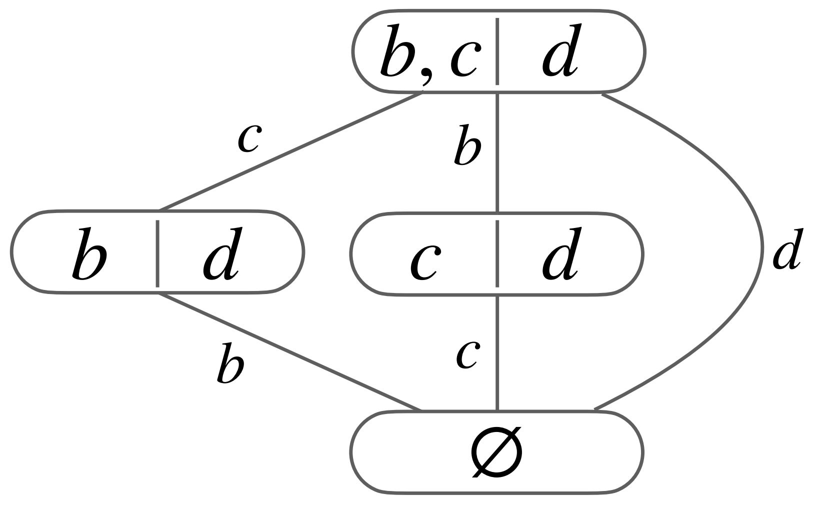
Format Language

CSF
Dense
Compressed
Compressed

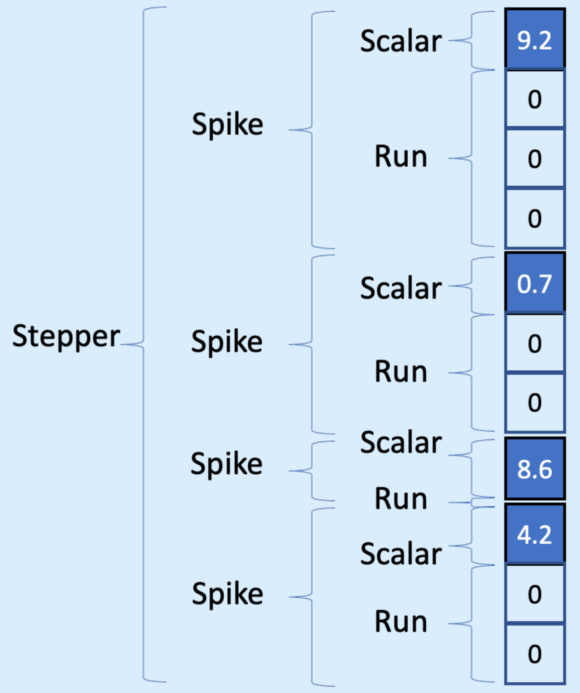
Sparse Iteration Graphs



Coiteration Code Generation



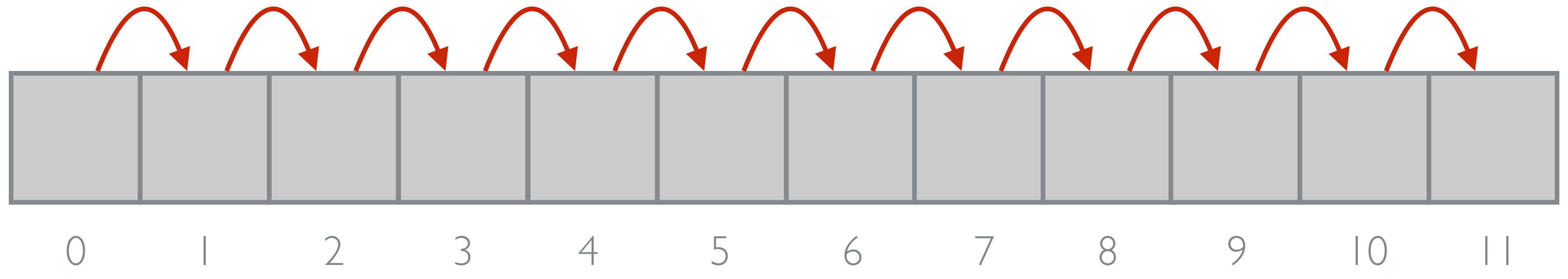
Looplet Language



Iteration And Coiteration

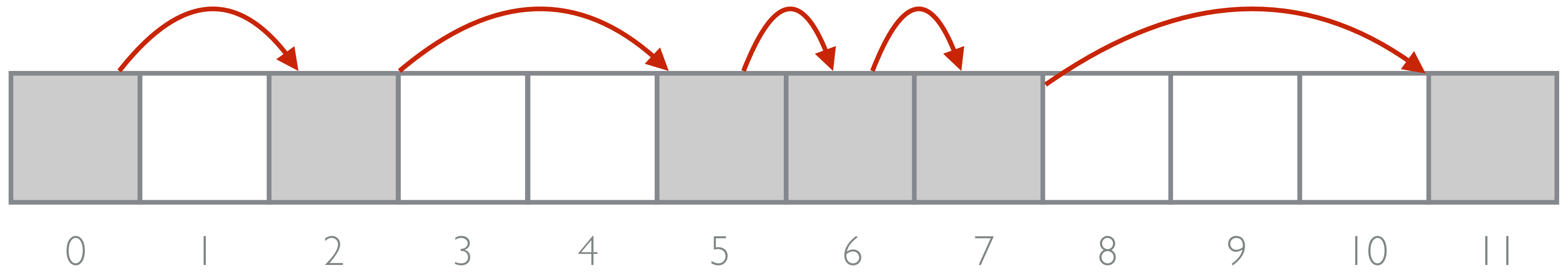
Dense

Increment



Compressed

Next



	Dense	Compressed	Run Length	Blocked
Dense	Increment	Next/lookup	…..	….
Compresed	Next/lookup	Next/next two-finger merge	……	….
Run Length	…..	…..	…..	….
Blocked	….	….	….	….

Variable blocks

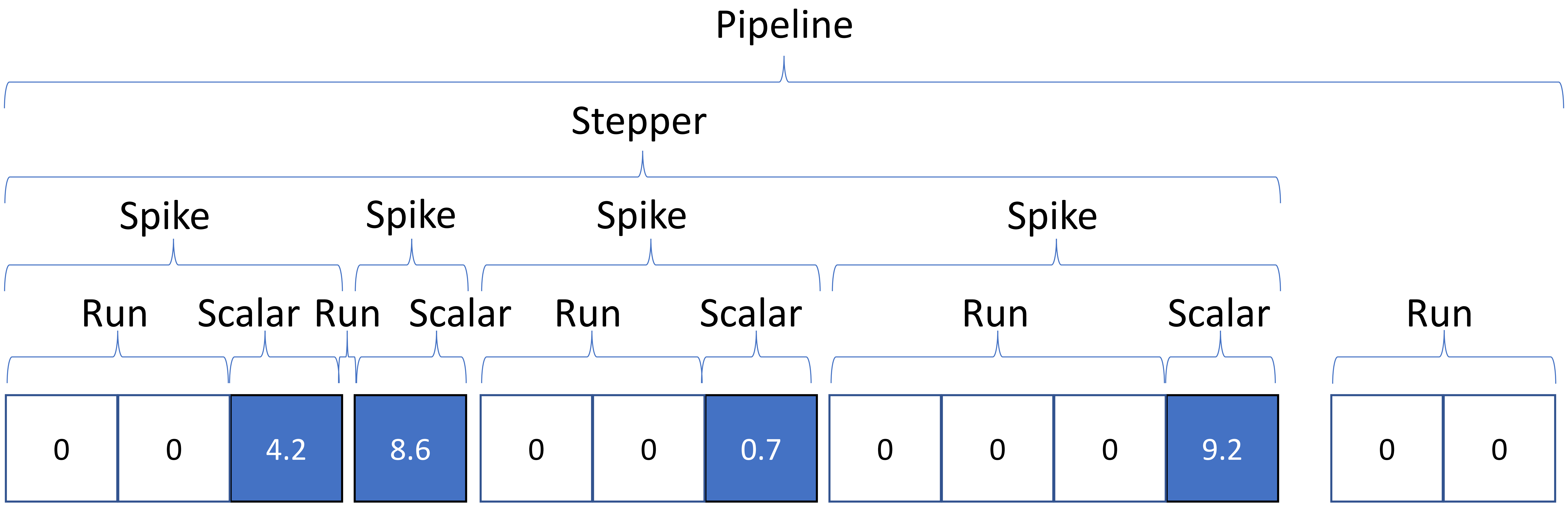
Pack BITS

Ragged

Variable blocks

Looplet Language

- A general language to iterate over structured data
  - Iterating over complex structured data expressed using a language of a few primitives
    - Lookup
    - Run
    - Spike
    - Pipeline
    - Stepper
    - Jumper
    - Shift
    - Switch



Looplet Language

- A general language to iterate over structured data
  - Iterating over complex structured data expressed using a language of a few primitives
    - Lookup
    - Spike
    - Stepper
    - Shift
    - Run
    - Pipeline
    - Jumper
    - Switch

- Code generation from the iteration protocols is simple and mechanical

Pipeline

Run

Lookup

Run

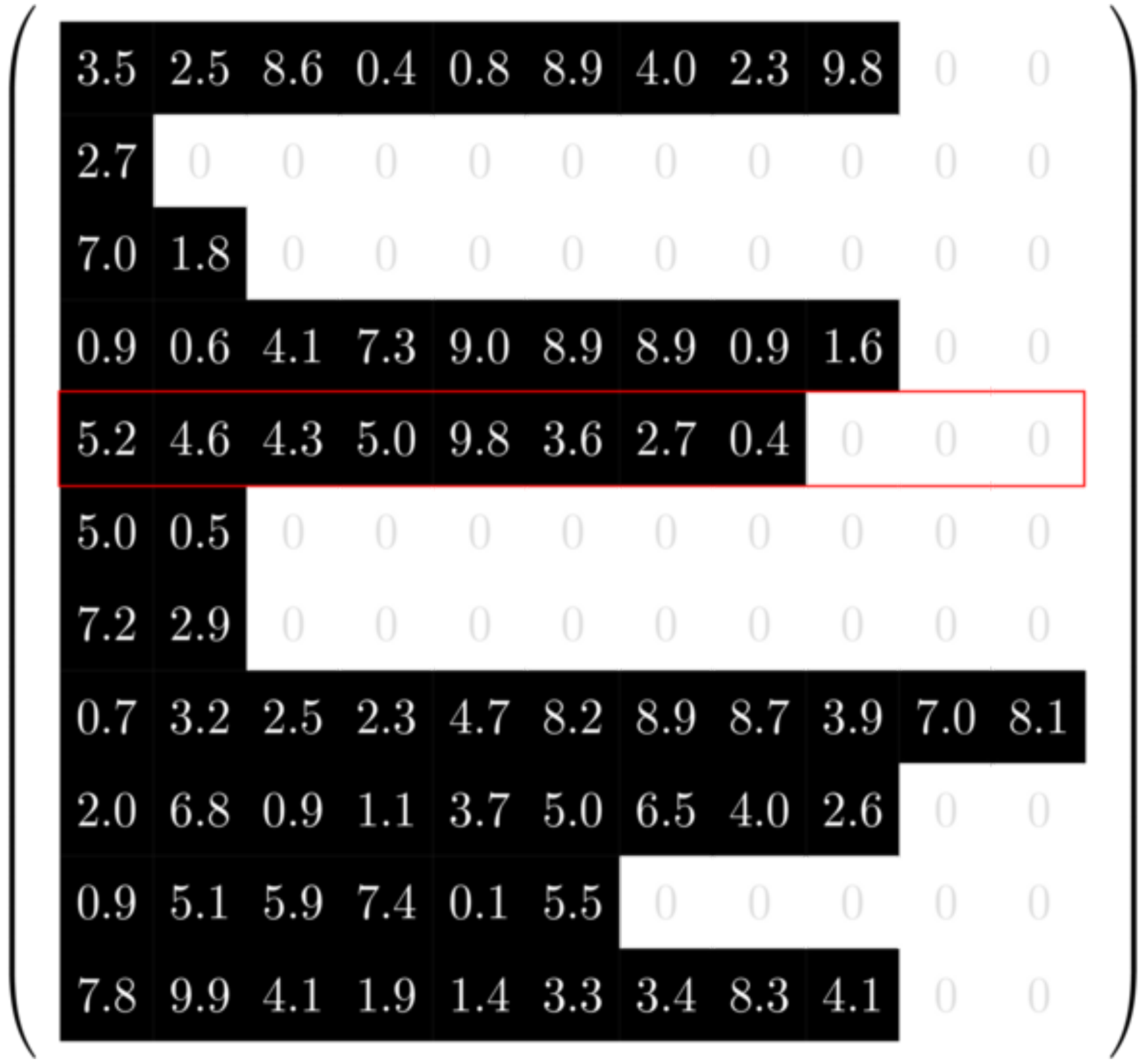
```
for i = 1:y.start-1
  visit(i, 0)
for i = y.start:y.stop
  visit(i, y.val[i + 1 - start])
for i = y.stop + 1:end
  visit(i, 0)
```

Looplet Language

- A general language to iterate over structured data
  - Iterating over complex structured data expressed using a language of a few primitives
    - Lookup
    - Run
    - Spike
    - Pipeline
    - Stepper
    - Jumper
    - Shift
    - Switch
- Code generation from the iteration protocols is simple and mechanical
- To coiterate, merge the individual iteration protocols
  - Use rewrite rules to simplify

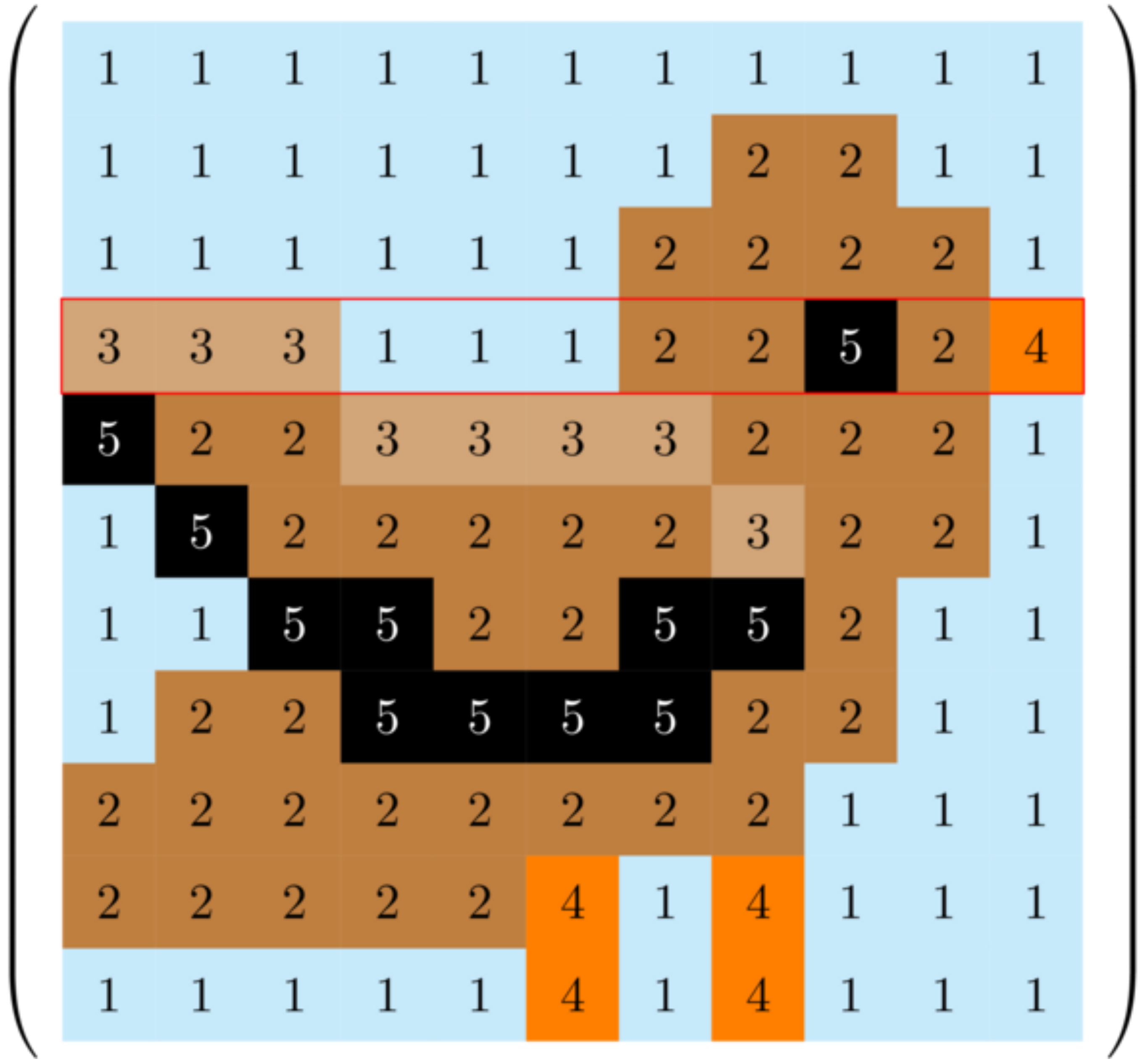
Looplet Language Supports Many Types Of Structured Data

Ragged Matrix



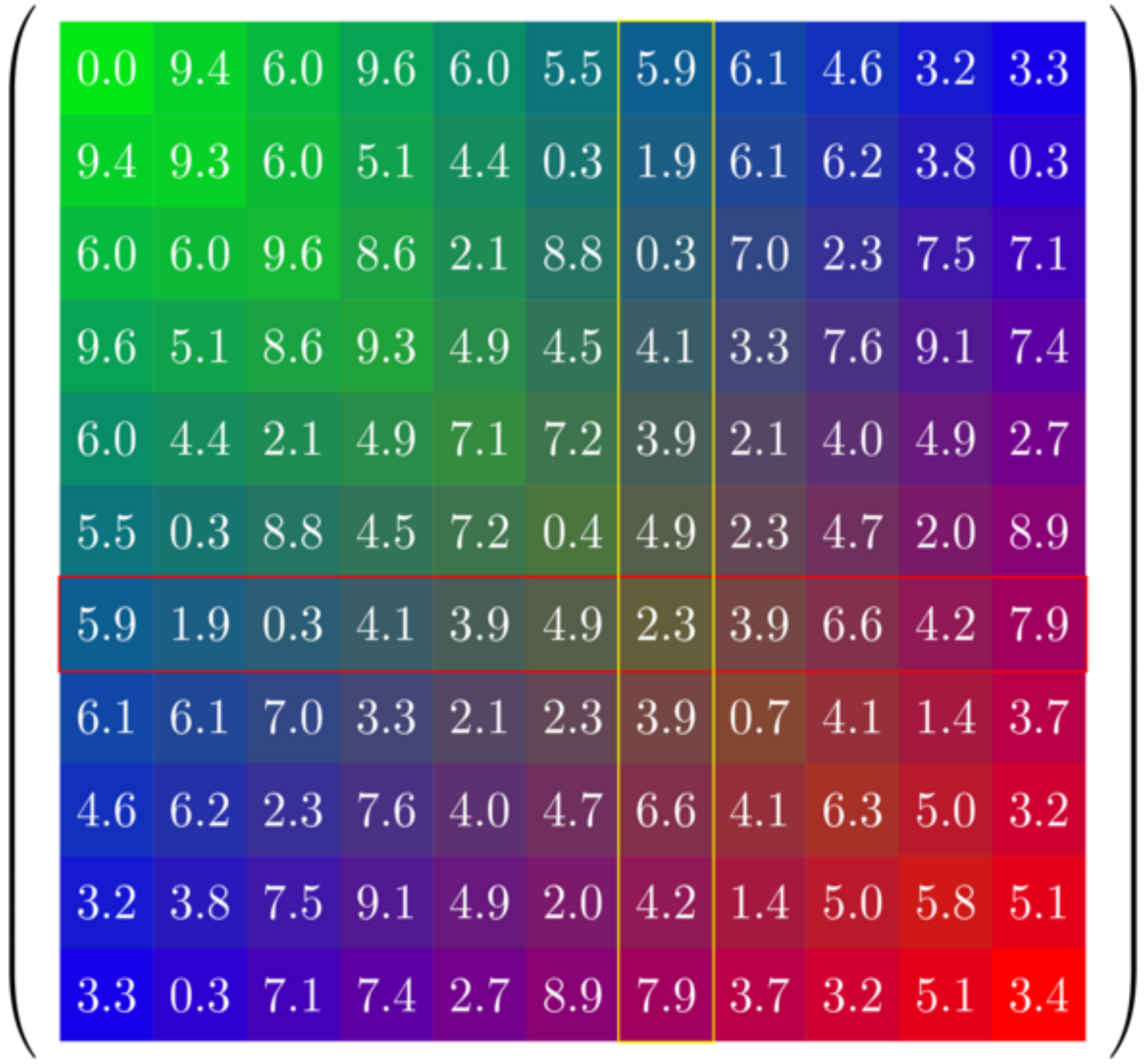
P. Fegade, T. Chen, P. B. Gibbons, and T. C. Mowry, “The CoRa Tensor Compiler: Compilation for Ragged Tensors with Minimal Padding”

Run Length Matrix



D. Donenfeld, S. Chou, and S. Amarasinghe, “Unified Compilation for Lossless Compression and Sparse Computing”

Symmetric Matrix

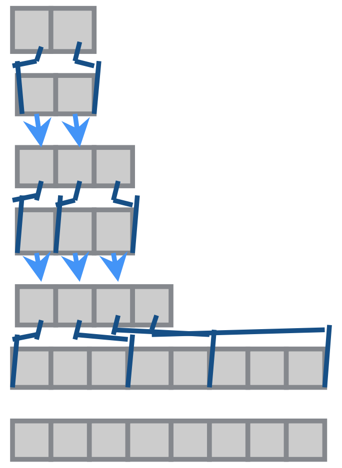


J. Shi, S. Chou, F. Kjolstad, and S. Amarasinghe, “An Attempt to Generate Code for Symmetric Tensor Computations”

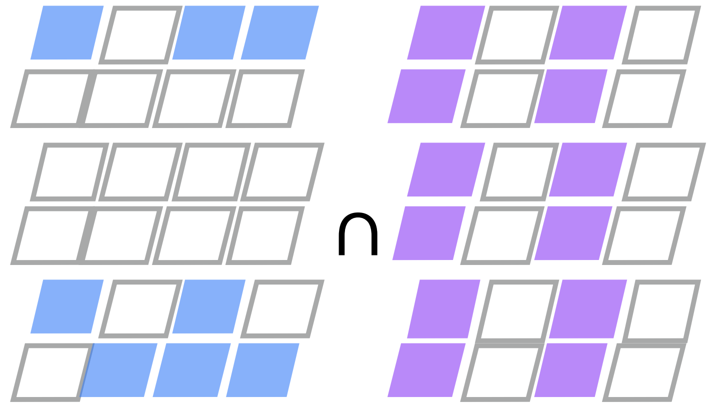
- Unifying what is currently done by multiple compilers
  - Hybrid “have-it-all” formats
  - Expanding into other types of structures

Challenges Of Sparse Array Compilation

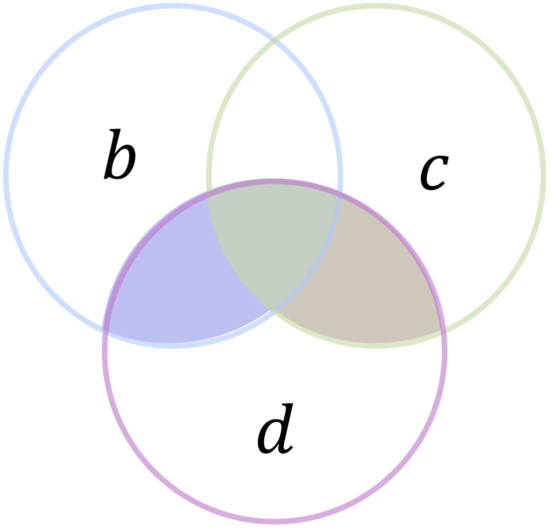
Irregular Data Structures



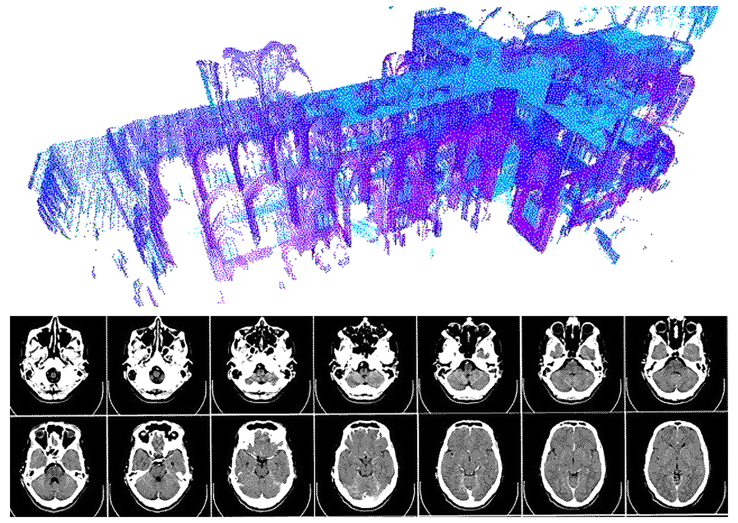
Sparse Iteration with limited O(1) access



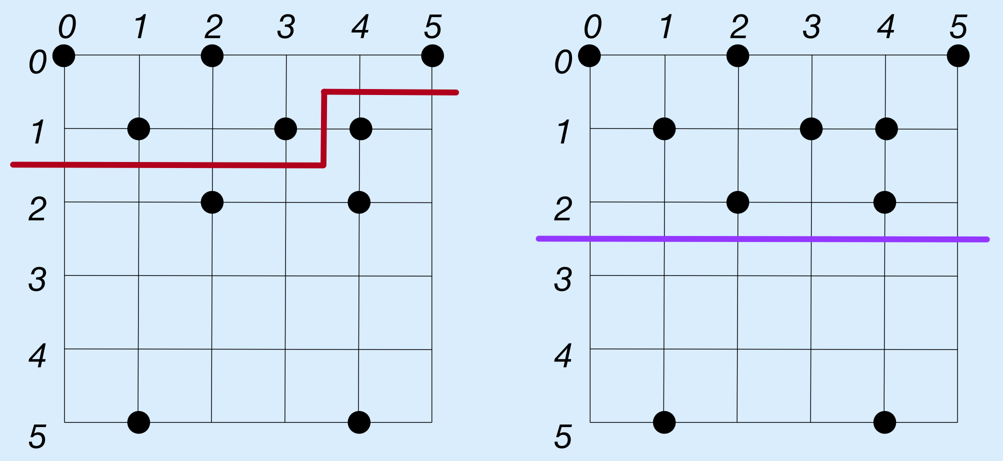
Avoid wasted work and iterations



Coiteration Over Complex Data



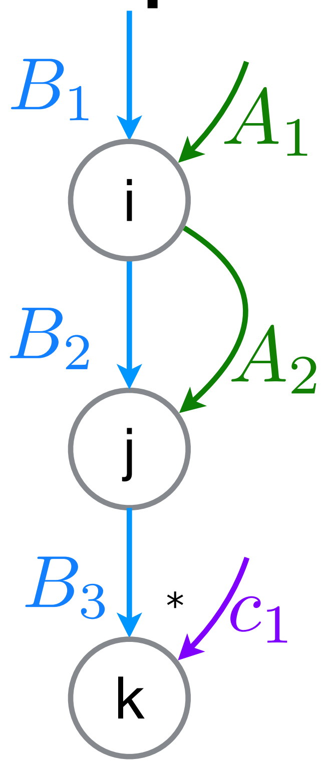
Optimize Parallelism and Locality



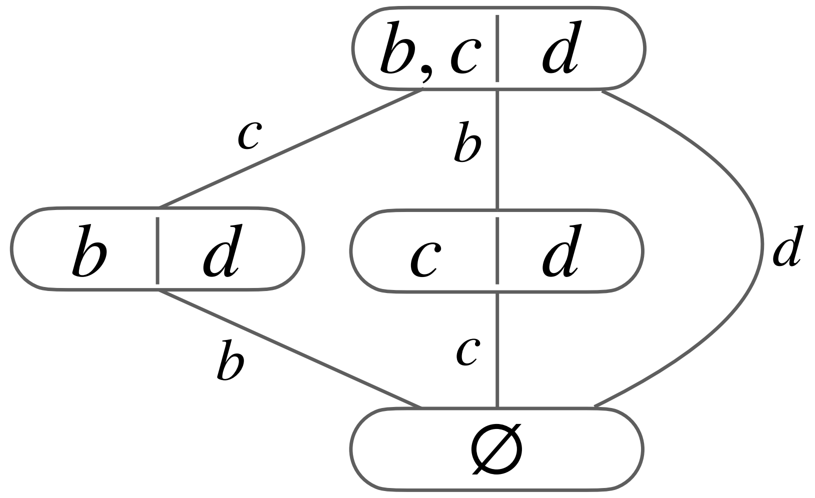
Format Language

CSF
Dense
Compressed
Compressed

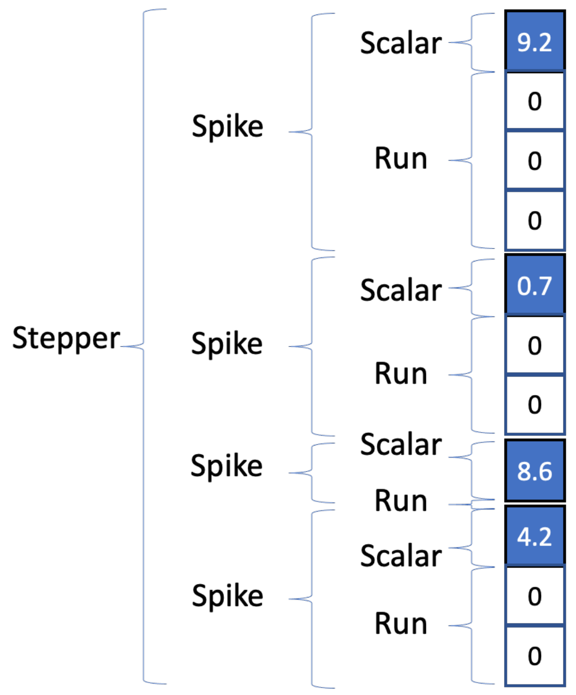
Sparse Iteration Graphs



Coiteration Code Generation



Looplet Language



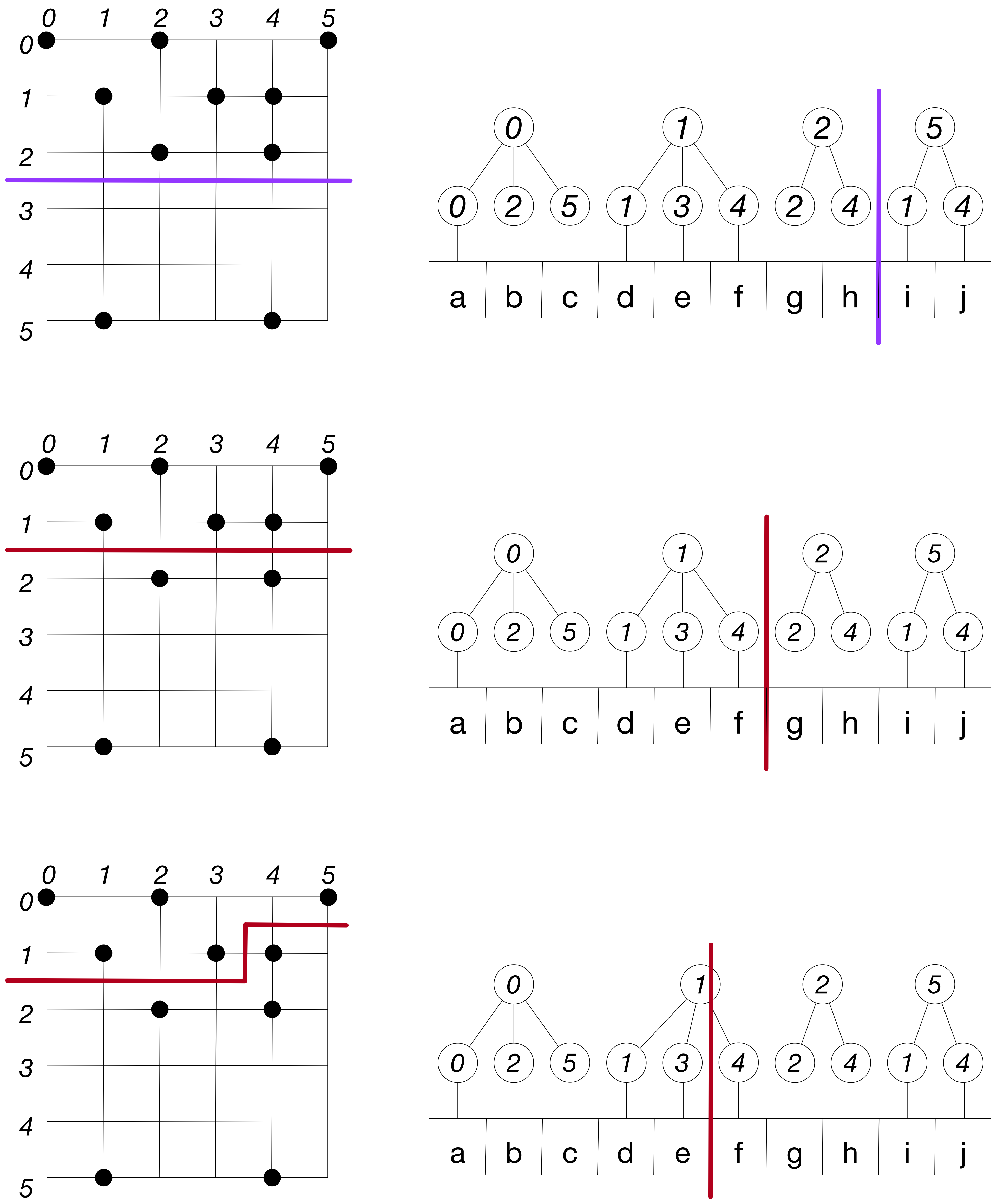
Scheduling Language

```
void reorder(IndexVar i1, IndexVar i2);
void fuse(IndexVar i, IndexVar j, IndexVar f)
void split(IndexVar i, IndexVar i1, IndexVar i2, int size)
void divide(IndexVar i, IndexVar i1, IndexVar i2, int size)
void pos(IndexVar i, IndexVar p, Access a)
void coord(IndexVar p, IndexVar i)
void parallelize(IndexVar i, ParallelUnit pu,
                 ReductionStrategy rs)
void unroll(IndexVar i, int unrollFactor)
void bound(IndexVar i, int min, int max)
void precompute(IndexExpr e, vector<IndexVar> vars,
                Tensor w);
```

Sparse Iteration Space Transformations

Scheduling API

```
void reorder(IndexVar i1, IndexVar i2);
void fuse(IndexVar i, IndexVar j, IndexVar f)
void split(IndexVar i, IndexVar i1, IndexVar i2, int size)
void divide(IndexVar i, IndexVar i1, IndexVar i2, int size)
void pos(IndexVar i, IndexVar p, Access a)
void coord(IndexVar p, IndexVar i)
void parallelize(IndexVar i, ParallelUnit pu,
                 ReductionStrategy rs)
void unroll(IndexVar i, int unrollFactor)
void bound(IndexVar i, int min, int max)
void precompute(IndexExpr e, vector<IndexVar> vars,
                Tensor w);
```



Sparse Iteration Space Transformations

```
int compute(taco_tensor_t *y, taco_tensor_t *A, taco_tensor_t *x) {
  ...
  for (int32_t i = 0; i < A1_dimension; i++) {
    for (int32_t jA = A2_pos[i]; jA < A2_pos[(i + 1)]; jA++) {
      int32_t j = A2_crd[jA];
      y_vals[i] = y_vals[i] + A_vals[jA] * x_vals[j];
    }
  }
  return 0;
}
```

Sparse Iteration Space Transformations

```
int compute(taco_tensor_t *y, taco_tensor_t *A, taco_tensor_t *x) {
  ...
  int32_t i_pos = 0;
  int32_t i = 0;
  for (int32_t fposA = 0; fposA < A2_pos[A1_dimension]; fposA++) {
    if (fposA >= A2_pos[A1_dimension])
      break;

    int32_t f = A2_crd[fposA];
    while (fposA == A2_pos[(i_pos + 1)]) {
      i_pos = i_pos + 1;
      i = i_pos;
    }
    y_vals[i] = y_vals[i] + A_vals[fposA] * x_vals[f];
  }
  return 0;
}
```

Scheduling Commands

.fuse(i, j, f)
.pos(f, fpos, A(i, j))

Sparse Iteration Space Transformations

```
int compute(taco_tensor_t *y, taco_tensor_t *A, taco_tensor_t *x) {
  ...
  int32_t i_pos = 0;
  int32_t i = 0;
  for (int32_t block = 0; block < ((A2_pos[A1_dimension] + 2047) / 2048); block++){
    for (int32_t warp = 0; warp < 8; warp++) {
      for (int32_t thread = 0; thread < 32; thread++) {
        for (int32_t thread_nz = 0; thread_nz < 8; thread_nz++) {
          int32_t fpos2 = thread * 8 + thread_nz;
          int32_t fpos1 = warp * 256 + fpos2;
          int32_t fposA = block * 2048 + fpos1;
          if (fposA >= A2_pos[A1_dimension])
            break;

          int32_t f = A2_crd[fposA];
          while (fposA == A2_pos[(i_pos + 1)]) {
            i_pos = i_pos + 1;
            i = i_pos;
          }
          y_vals[i] = y_vals[i] + A_vals[fposA] * x_vals[f];
        }
      }
    }
  }
  return 0;
}
```

Scheduling Commands

.fuse(i, j, f)
.pos(f, fpos, A(i, j))
.split(fpos, block, fpos1, NNZ_PER_TB)
.split(fpos1, warp, fpos2, NNZ_PER_WARP)
.split(fpos2, thread, thread_nz, NNZ_PER_THREAD)
.reorder({block, warp, thread, thread_nz})

Sparse Iteration Space Transformations

```
__global__
void computeDeviceKernel0(taco_tensor_t * __restrict__ A, int32_t* i_blockStarts,
taco_tensor_t * __restrict__ x, taco_tensor_t * __restrict__ y){
  ...
  int32_t block = blockIdx.x;
  int32_t thread = (threadIdx.x % (32));
  int32_t warp = (threadIdx.x / 32);

  double tthread_nz_val = 0.0;
  int32_t pA2_begin = i_blockStarts[block];
  int32_t pA2_end = i_blockStarts[(block + 1)];
  int32_t fpos2 = thread * 8;
  int32_t fpos1 = warp * 256 + fpos2;
  int32_t fposA = block * 2048 + fpos1;
  int32_t i_pos = taco_binarySearchBefore(A2_pos, pA2_begin, pA2_end, fposA);
  int32_t i = i_pos;
  for (int32_t thread_nz = 0; thread_nz < 8; thread_nz++) {
    int32_t fpos2 = thread * 8 + thread_nz;
    int32_t fpos1 = warp * 256 + fpos2;
    int32_t fposA = block * 2048 + fpos1;
    if (fposA >= A2_pos[A1_dimension])
      break;

    int32_t f = A2_crd[fposA];
    while (fposA == A2_pos[(i_pos + 1)]) {
      i_pos = i_pos + 1;
      i = i_pos;
    }
    tthread_nz_val = tthread_nz_val + A_vals[fposA] * x_vals[f];
    if (fposA + 1 == A2_pos[(i_pos + 1)]) {
      atomicAdd(&y_vals[i], tthread_nz_val);
      tthread_nz_val = 0.0;
    }
  }
  atomicAddWarp<double>(y_vals, i, tthread_nz_val);
}
```

Scheduling Commands

.fuse(i, j, f)
.pos(f, fpos, A(i, j))
.split(fpos, block, fpos1, NNZ_PER_TB)
.split(fpos1, warp, fpos2, NNZ_PER_WARP)
.split(fpos2, thread, thread_nz, NNZ_PER_THREAD)
.reorder({block, warp, thread, thread_nz})
.parallelize(block, ParallelUnit::GPUBlock, OutputRaceStrategy::IgnoreRaces)
.parallelize(warp, ParallelUnit::GPUWarp, OutputRaceStrategy::IgnoreRaces)
.parallelize(thread, ParallelUnit::GPUThread, OutputRaceStrategy::Atomics)

```
int compute(taco_tensor_t *y, taco_tensor_t *A, taco_tensor_t *x) {
  ...
  gpuErrchk(cudaMallocManaged((void**)&i_blockStarts, sizeof(int32_t) *
((A2_pos[A1_dimension] + 2047) / 2048 + 1)));
  i_blockStarts = taco_binarySearchBeforeBlockLaunch(A2_pos, i_blockStarts,
(int32_t) 0, A1_dimension, (int32_t) 2048, (int32_t) 256, ((A2_pos[A1_dimension] +
2047) / 2048));

  computeDeviceKernel0<<<(A2_pos[A1_dimension] + 2047) / 2048, 32 * 8>>>(A,
i_blockStarts, x, y);
  cudaDeviceSynchronize();
  ...
}
```

Sparse Iteration Space Transformations

```
__global__
void computeDeviceKernel0(taco_tensor_t * __restrict__ A, int32_t* i_blockStarts,
taco_tensor_t * __restrict__ x, taco_tensor_t * __restrict__ y){
 ....
  int32_t block = blockIdx.x;
  int32_t thread = (threadIdx.x % (32));
  int32_t warp = (threadIdx.x / 32);

  double precomputed[8];
  for (int32_t pprecomputed = 0; pprecomputed < 8; pprecomputed++) {
    precomputed[pprecomputed] = 0.0;
  }
  for (int32_t thread_nz_pre = 0; thread_nz_pre < 8; thread_nz_pre++) {
    int32_t thread_nz = thread_nz_pre;
    int32_t fpos2 = thread * 8 + thread_nz;
    int32_t fpos1 = warp * 256 + fpos2;
    int32_t fposA = block * 2048 + fpos1;
    if (fposA >= A2_pos[A1_dimension])
      break;

    int32_t f = A2_crd[fposA];
    precomputed[thread_nz_pre] = A_vals[fposA] * x_vals[f];
  }
  double tthread_nz_val = 0.0;
  ...
  for (int32_t thread_nz = 0; thread_nz < 8; thread_nz++) {
    ...
    tthread_nz_val = tthread_nz_val + precomputed[thread_nz];
    if (fposA + 1 == A2_pos[(i_pos + 1)]) {
      atomicAdd(&y_vals[i], tthread_nz_val);
      tthread_nz_val = 0.0;
    }
  }
  atomicAddWarp<double>(y_vals, i, tthread_nz_val);
}
```

Scheduling Commands

.fuse(i, j, f)
.pos(f, fpos, A(i, j))
.split(fpos, block, fpos1, NNZ_PER_TB)
.split(fpos1, warp, fpos2, NNZ_PER_WARP)
.split(fpos2, thread, thread_nz, NNZ_PER_THREAD)
.reorder({block, warp, thread, thread_nz})
.precompute(precomputedExpr, thread_nz,
thread_nz_pre, precomputed)
.parallelize(block, ParallelUnit::GPUBlock,
OutputRaceStrategy::IgnoreRaces)
.parallelize(warp, ParallelUnit::GPUWarp,
OutputRaceStrategy::IgnoreRaces)
.parallelize(thread, ParallelUnit::GPUThread,
OutputRaceStrategy::Atomics)

```
int compute(taco_tensor_t *y, taco_tensor_t *A, taco_tensor_t *x) {
  ...
  gpuErrchk(cudaMallocManaged((void**)&i_blockStarts, sizeof(int32_t) *
((A2_pos[A1_dimension] + 2047) / 2048 + 1)));
  i_blockStarts = taco_binarySearchBeforeBlockLaunch(A2_pos, i_blockStarts,
(int32_t) 0, A1_dimension, (int32_t) 2048, (int32_t) 256, ((A2_pos[A1_dimension] +
2047) / 2048));

  computeDeviceKernel0<<<(A2_pos[A1_dimension] + 2047) / 2048, 32 * 8>>>(A,
i_blockStarts, x, y);
  cudaDeviceSynchronize();
  ...
}
```

Sparse Iteration Space Transformations

```
__global__
void computeDeviceKernel0(taco_tensor_t * __restrict__ A, int32_t* i_blockStarts,
taco_tensor_t * __restrict__ x, taco_tensor_t * __restrict__ y){
  ...
  int32_t thread_nz = 0;
  int32_t fpos2 = thread * 8 + thread_nz;
  int32_t fpos1 = warp * 256 + fpos2;
  int32_t fposA = block * 2048 + fpos1;
  int32_t f = A2_crd[fposA];
  if (block * 2048 + fpos1 + 8 >= A2_pos[A1_dimension]) {
    for (int32_t thread_nz_pre = 0; thread_nz_pre < 8; thread_nz_pre++) {
      int32_t thread_nz = thread_nz_pre;
      int32_t fpos2 = thread * 8 + thread_nz;
      int32_t fpos1 = warp * 256 + fpos2;
      int32_t fposA = block * 2048 + fpos1;
      if (fposA >= A2_pos[A1_dimension])
        break;

      int32_t f = A2_crd[fposA];
      precomputed[thread_nz_pre] = A_vals[fposA] * x_vals[f];
    }
  }
  else {
    #pragma unroll 8
    for (int32_t thread_nz_pre = 0; thread_nz_pre < 8; thread_nz_pre++) {
      int32_t thread_nz = thread_nz_pre;
      int32_t fpos2 = thread * 8 + thread_nz;
      int32_t fpos1 = warp * 256 + fpos2;
      int32_t fposA = block * 2048 + fpos1;
      int32_t f = A2_crd[fposA];
      precomputed[thread_nz_pre] = A_vals[fposA] * x_vals[f];
    }
  }
  ...
}
```

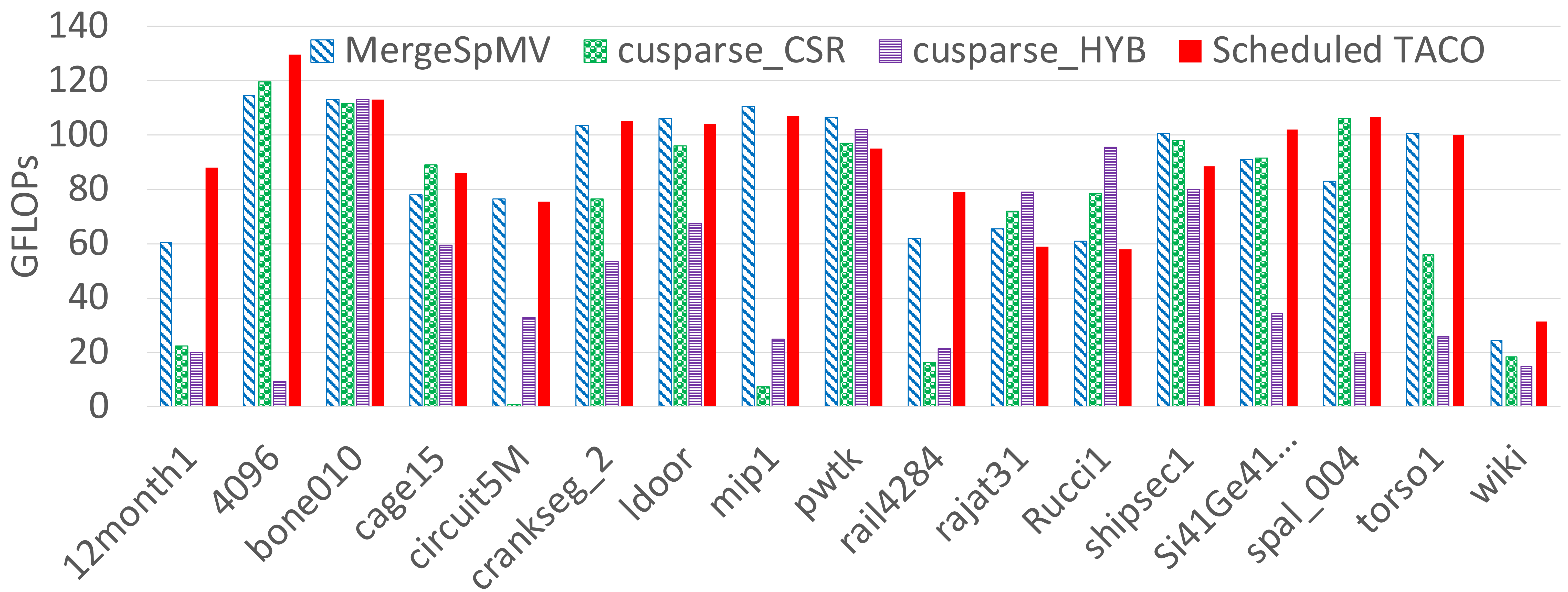
Scheduling Commands

.fuse(i, j, f)
.pos(f, fpos, A(i, j))
.split(fpos, block, fpos1, NNZ_PER_TB)
.split(fpos1, warp, fpos2, NNZ_PER_WARP)
.split(fpos2, thread, thread_nz, NNZ_PER_THREAD)
.reorder({block, warp, thread, thread_nz})
.precompute(precomputedExpr, thread_nz,
thread_nz_pre, precomputed)
.unroll(thread_nz_pre, NNZ_PER_THREAD)
.parallelize(block, ParallelUnit::GPUBlock,
OutputRaceStrategy::IgnoreRaces)
.parallelize(warp, ParallelUnit::GPUWarp,
OutputRaceStrategy::IgnoreRaces)
.parallelize(thread, ParallelUnit::GPUThread,
OutputRaceStrategy::Atomics)

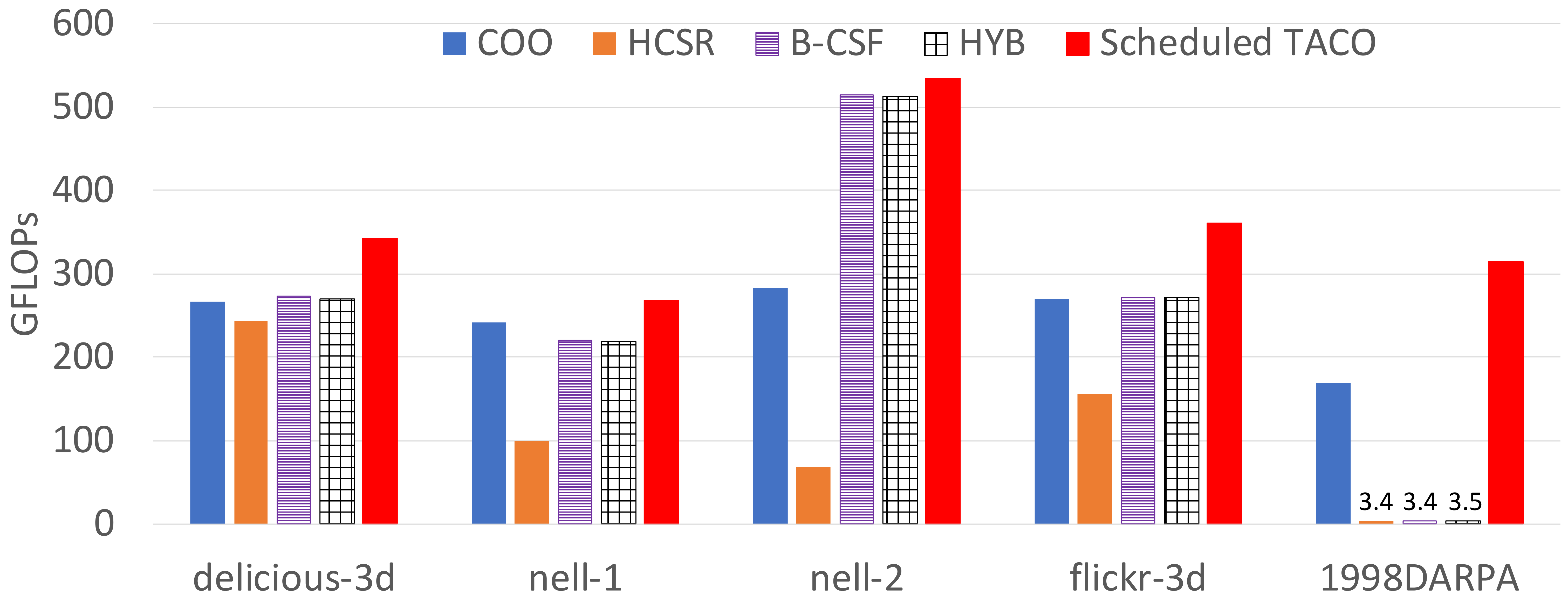
```
int compute(taco_tensor_t *y, taco_tensor_t *A, taco_tensor_t *x) {
  ...
  gpuErrchk(cudaMallocManaged((void**)&i_blockStarts, sizeof(int32_t) *
((A2_pos[A1_dimension] + 2047) / 2048 + 1)));
  i_blockStarts = taco_binarySearchBeforeBlockLaunch(A2_pos, i_blockStarts,
(int32_t) 0, A1_dimension, (int32_t) 2048, (int32_t) 256, ((A2_pos[A1_dimension] +
2047) / 2048));

  computeDeviceKernel0<<<(A2_pos[A1_dimension] + 2047) / 2048, 32 * 8>>>(A,
i_blockStarts, x, y);
  cudaDeviceSynchronize();
  ...
}
```

With Sparse Iteration Space Transformations We Can Target GPUS

SpMV on GPU
(NVIDIA V100)



MTTKRP on GPU
(NVIDIA V100)



Availability of TACO

C++ Tensor Library

```
Format  CSR({Dense,Sparse});
Format  CSF({Sparse,Sparse,Sparse});
Format SVEC({Sparse});
```

```
Tensor<double> A({1024,1024}, CSR);
Tensor<double> B = read("B.tns", CSF);
Tensor<double> c = read("c.tns", SVEC);
```

```
Var i, j, k;
A(i,j) = B(i,j,k) * c(k);
```

```
A.compile();
A.assemble();
A.compute();
```

Some Rights Reserved

(Vanilla) MIT License

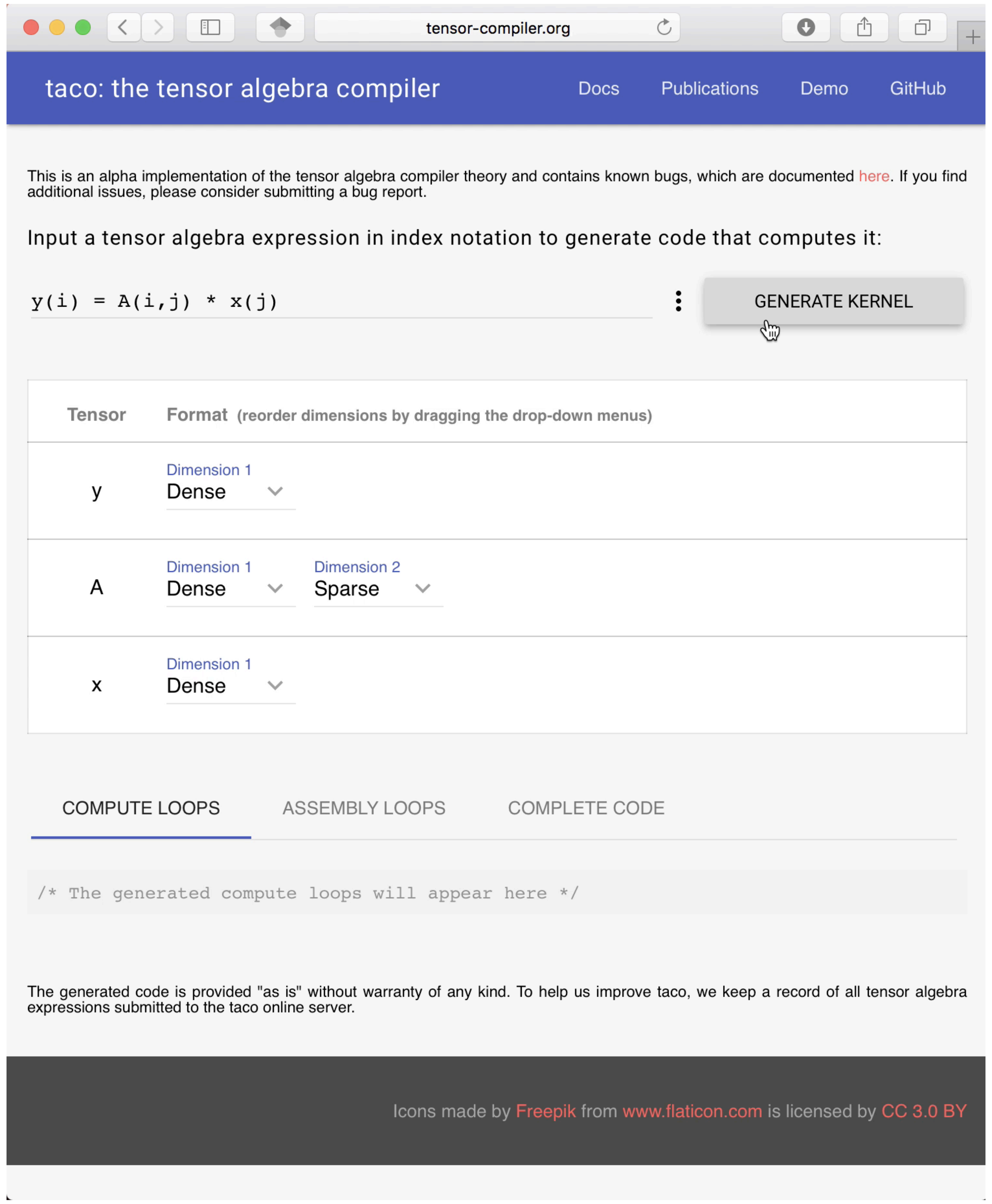
tensor-compiler.org

github.com/tensor-compiler/taco

PyTACO

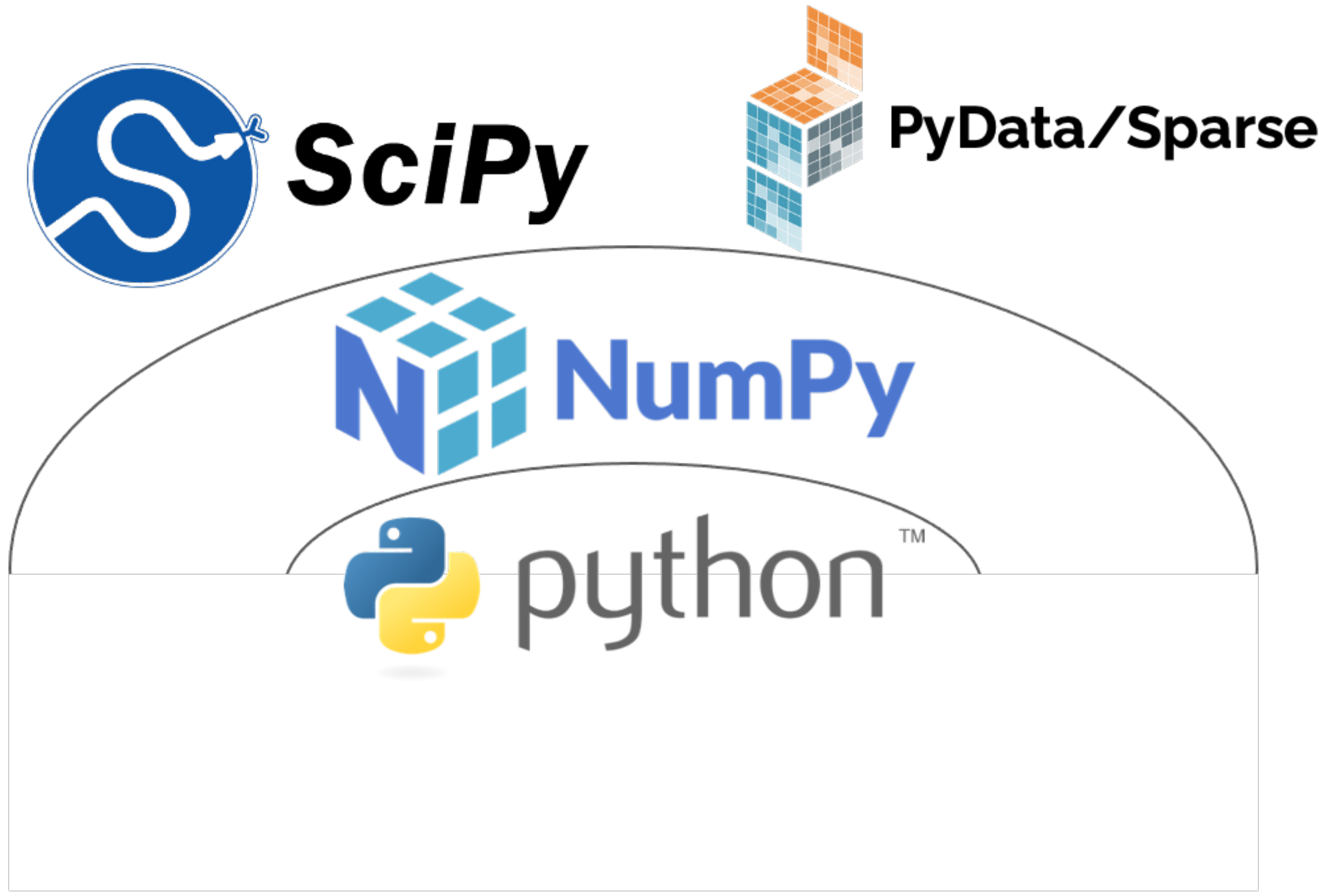
```python
import taco
from taco import sparse, dense

csr = taco.format([dense, sparse])
csf = taco.format([sparse, sparse, sparse])
sv  = taco.format([sparse])

A = taco.tensor([2, 3], csr, dtype=taco.float32)
B = taco.tensor([2, 3, 4], csf, dtype=taco.float32)
C = taco.tensor([4], sv, dtype=taco.float32)

B[0,0,0] = 1.0
B[1,2,0] = 2.0
B[1,2,1] = 3.0
C[0] = 4.0
C[1] = 5.0

# Using C++ style API
i, j, k  = taco.get_index_vars(3)
A(i,j).assign(B(i,j,k) * C(k))
print(A)

# Using Python Style API
print(taco.tensor_dot(B, C))

# Using taco parser
print(taco.parse("A(i, j) = B(i, j, k) * C(k)", B, C))

# Using numpy style einsum
print(taco.einsum("ijk,k->ij", B, C))
```

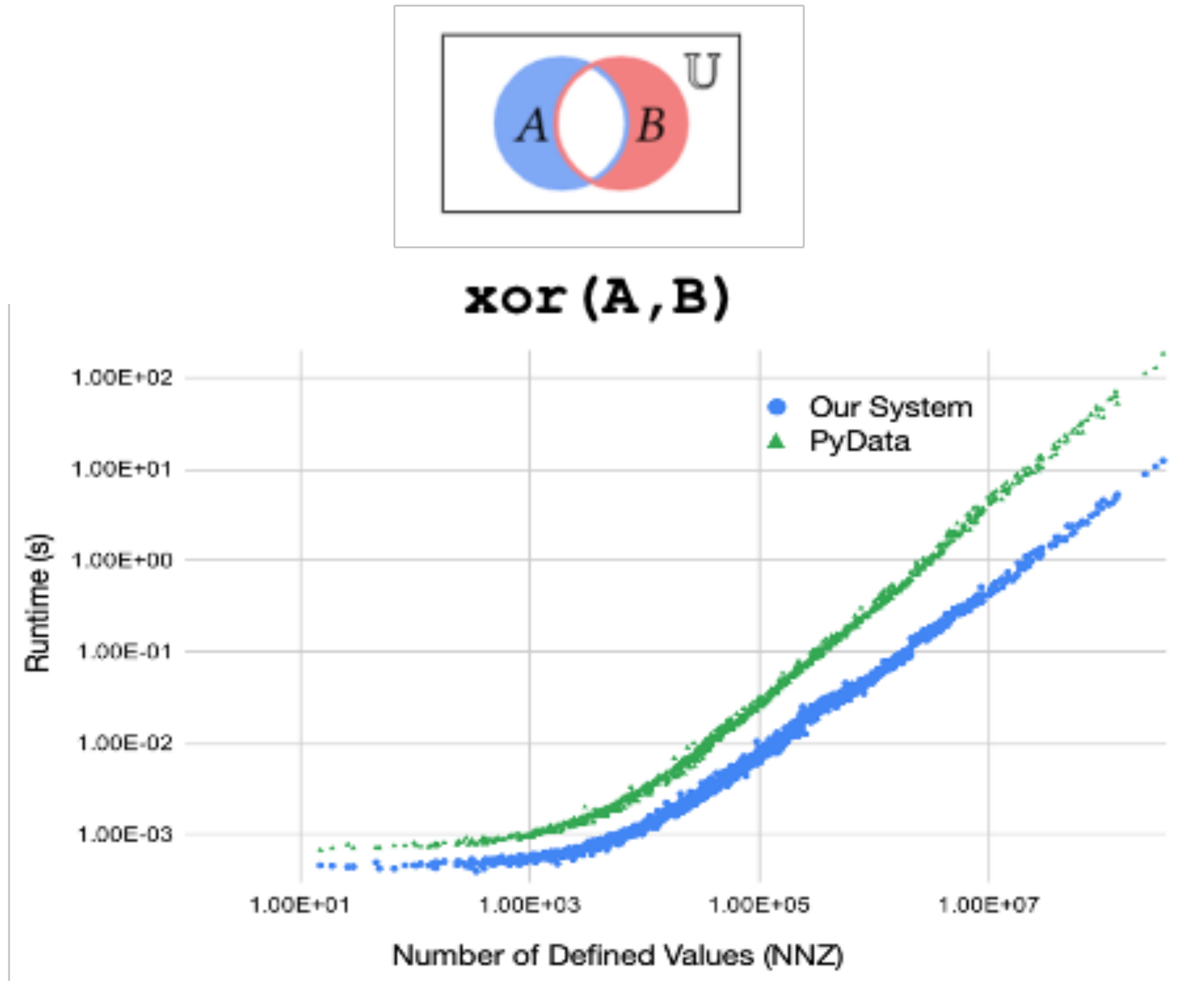
Online Code Generator

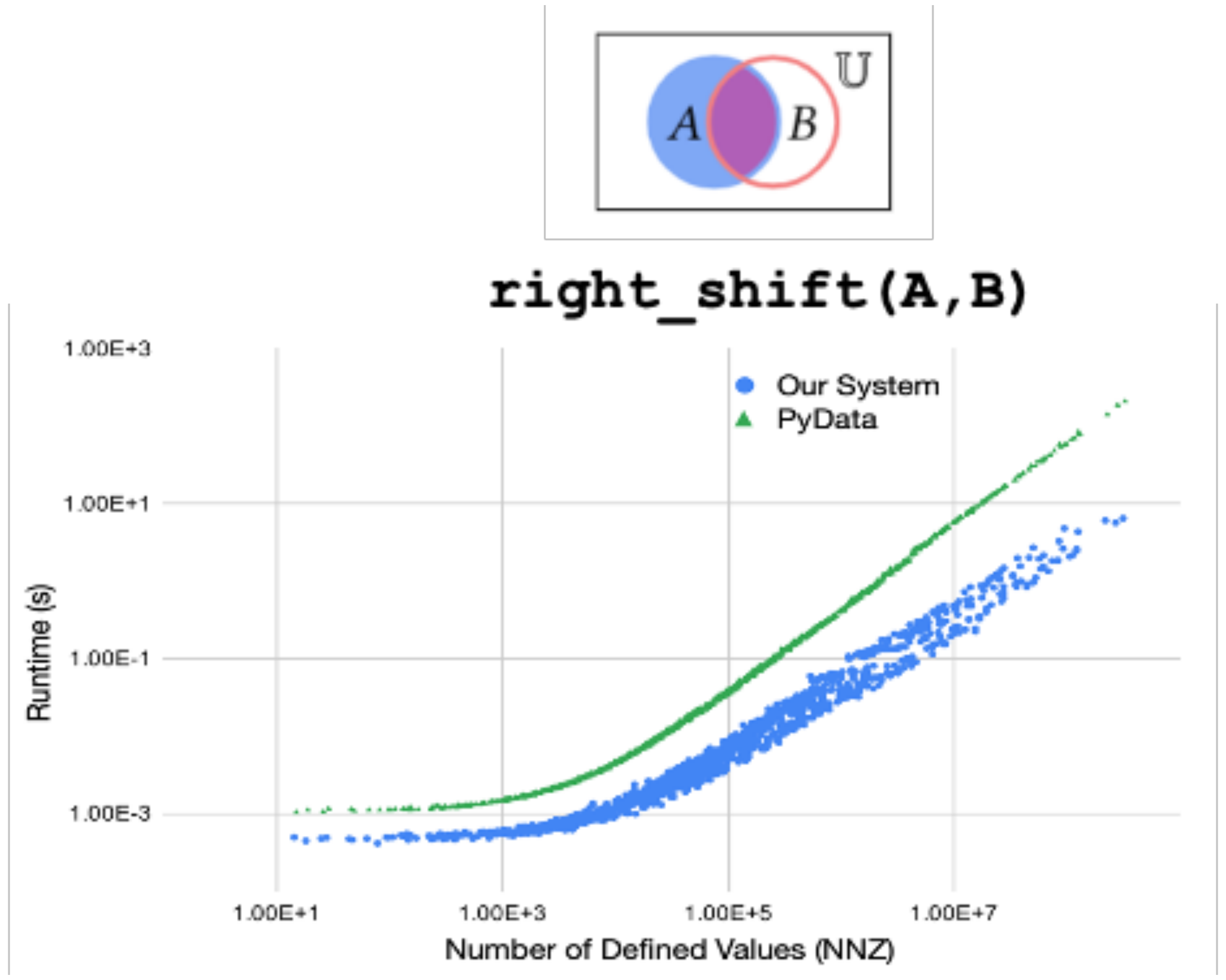


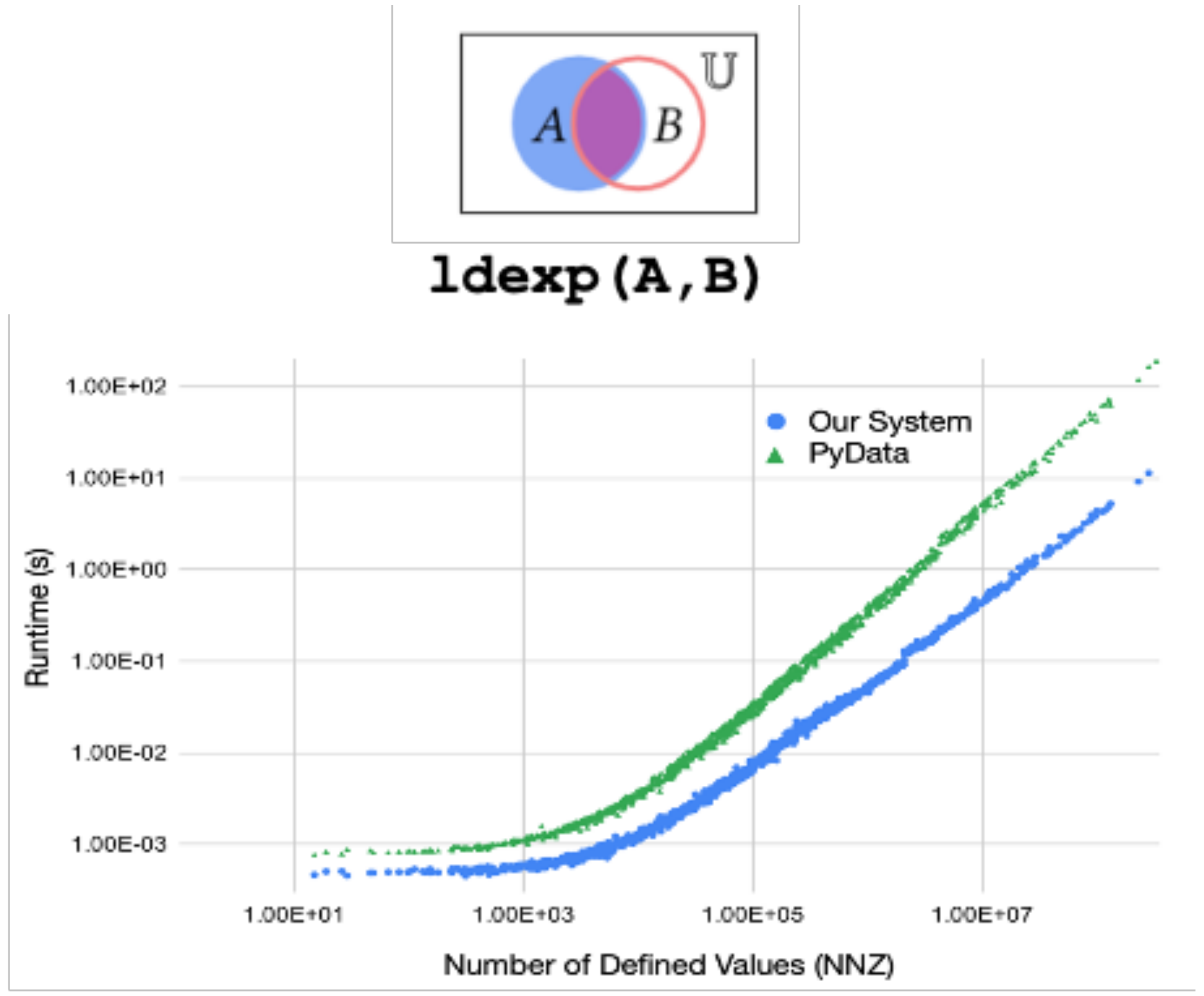
tensor-compiler.org/codegen/

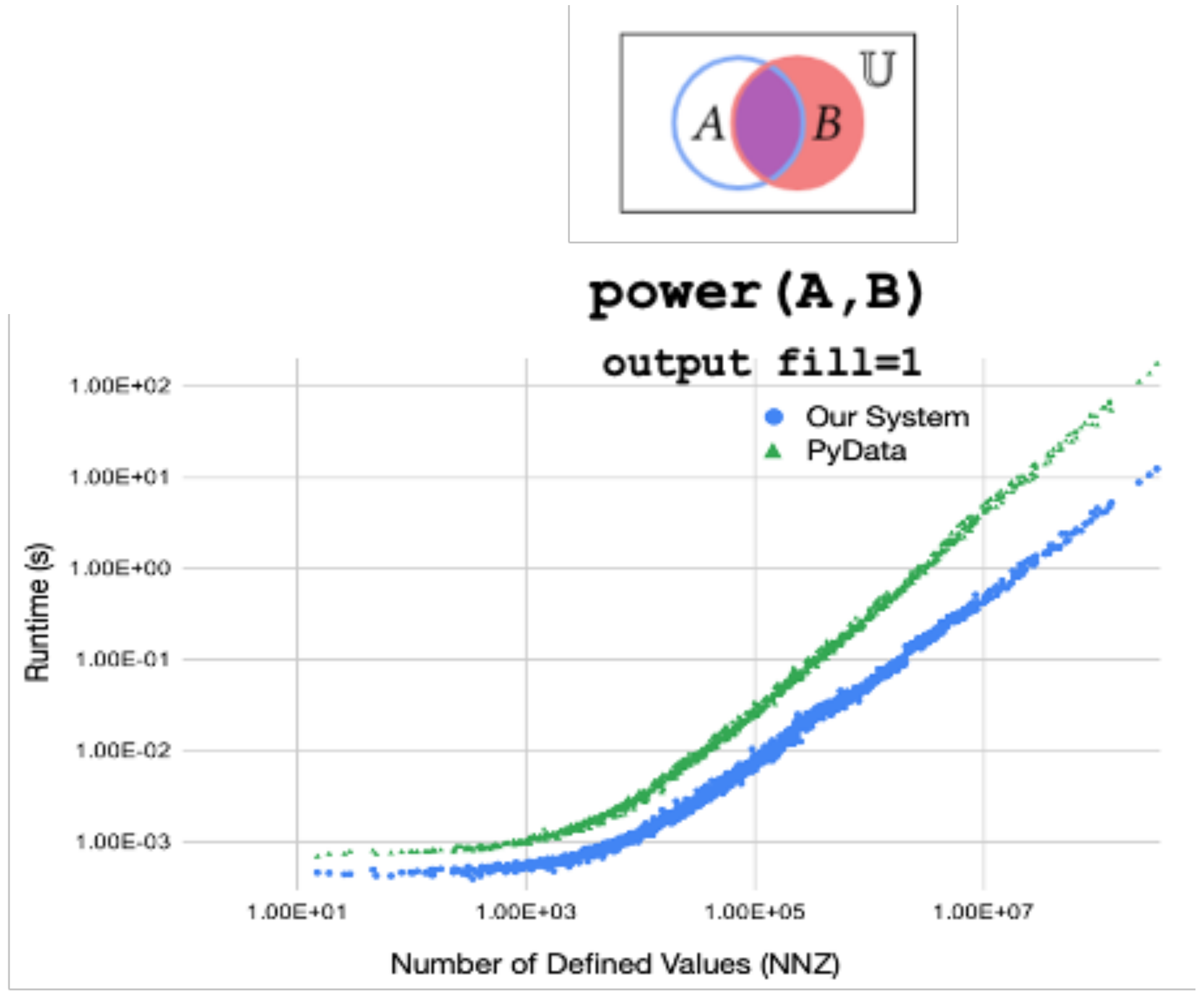
Speeding Up Sparse Array Programming In The Python Ecosystem











Publications On TACO

1. WACO: Learning workload-aware co-optimization of the format and schedule of a sparse tensor program.
   Jaeyeon Won, Charith Mendis, Joel Emer, Saman Amarasinghe.
   International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).
   Apr, 2023. Bibtex.
2. Compilation of Dynamic Sparse Tensor Algebra.
   Stephen Chou, Saman Amarasinghe.
   Proceedings of the ACM on Programming Languages.
   New York, NY, USA. Oct, 2022. Bibtex.
3. Format Abstractions for the Compilation of Sparse Tensor Algebra.
   Stephen Chou.
   PhD Thesis, Massachusetts Institute of Technology.
   Cambridge, MA. Aug, 2022. Bibtex.
4. Compilation of Sparse Array Programming Models.
   Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, Fredrik Kjolstad.
   Proceedings of the ACM on Programming Languages.
   Chicago, IL, USA. Oct, 2021. Bibtex.
5. Automatic Generation of Efficient Sparse Tensor Format Conversion Routines.
   Stephen Chou, Fredrik Kjolstad, Saman Amarasinghe.
   Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.
   New York, NY, USA. Jun, 2020. Bibtex.
6. A Framework for Computing on Sparse Tensors based on Operator Properties.
   Rawn Henry.
   MEng Thesis, Massachusetts Institute of Technology.
   Cambridge, MA. May, 2020. Bibtex.
7. Automatic Optimization of Sparse Tensor Algebra Programs.
   Ziheng Wang.
   MEng Thesis, Massachusetts Institute of Technology.
   Cambridge, MA. May, 2020. Bibtex.
8. A Unified Iteration Space Transformation Framework for Sparse and Dense Tensor Algebra.
   Ryan Senanayake.
   MEng Thesis, Massachusetts Institute of Technology.
   Cambridge, MA. Feb, 2020. Bibtex.
   Charles Jennifer Johnson Computer Science MEng Thesis Award - 1st place, 2020.
9. Sparse Tensor Transpositions in the Tensor Algebra Compiler.
   Suzanne Mueller.
   MEng Thesis, Massachusetts Institute of Technology.
   Cambridge, MA. Feb, 2020. Bibtex.
10. Sparse Tensor Algebra Compilation.
    Fredrik Kjolstad.
    PhD Thesis, Massachusetts Institute of Technology.
    Cambridge, MA. Feb, 2020. Bibtex.
    George M. Sprowls PhD Thesis Award in MIT Computer Science, 1st place, 2021.
11. A Tensor Algebra Compiler Library Interface and Runtime.
    Patricio Noyola.
    MEng Thesis, Massachusetts Institute of Technology.
    Cambridge, MA. May, 2019. Bibtex.
12. Tensor Algebra Compilation with Workspaces.
    Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, Saman Amarasinghe.
    International Symposium on Code Generation and Optimization.
    Feb, 2019. Bibtex.
13. SuperTaco: Taco Tensor Algebra Kernels on Distributed Systems Using Legion.
    Sachin Dilip Shinde.
    MEng Thesis, Massachusetts Institute of Technology.
    Cambridge, MA. Feb, 2019. Bibtex.
14. Format Abstraction for Sparse Tensor Algebra Compilers.
    Stephen Chou, Fredrik Kjolstad, Saman Amarasinghe.
    Proceedings of the ACM on Programming Languages.
    New York, NY, USA. Oct, 2018. Bibtex.
15. Unified Sparse Formats for Tensor Algebra Compilers.
    Stephen Chou.
    SM Thesis, Massachusetts Institute of Technology.
    Cambridge, MA. Feb, 2018. Bibtex.
16. The Tensor Algebra Compiler.
    Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, Saman Amarasinghe.
    Proceedings of the ACM on Programming Languages.
    New York, NY, USA. Oct, 2017. Bibtex.
    OOPSLA Distinguished Paper Award.
17. Taco: A Tool to Generate Tensor Algebra Kernels.
    Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, Saman Amarasinghe.
    Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering.
    Piscataway, NJ, USA. 2017. Bibtex.
18. An Investigation of Sparse Tensor Formats for Tensor Libraries.
    Parker Allen Tew.
    MEng Thesis, Massachusetts Institute of Technology.
    Cambridge, MA. Jun, 2016. Bibtex.

http://groups.csail.mit.edu/commit/?page=publications

What Is Next...

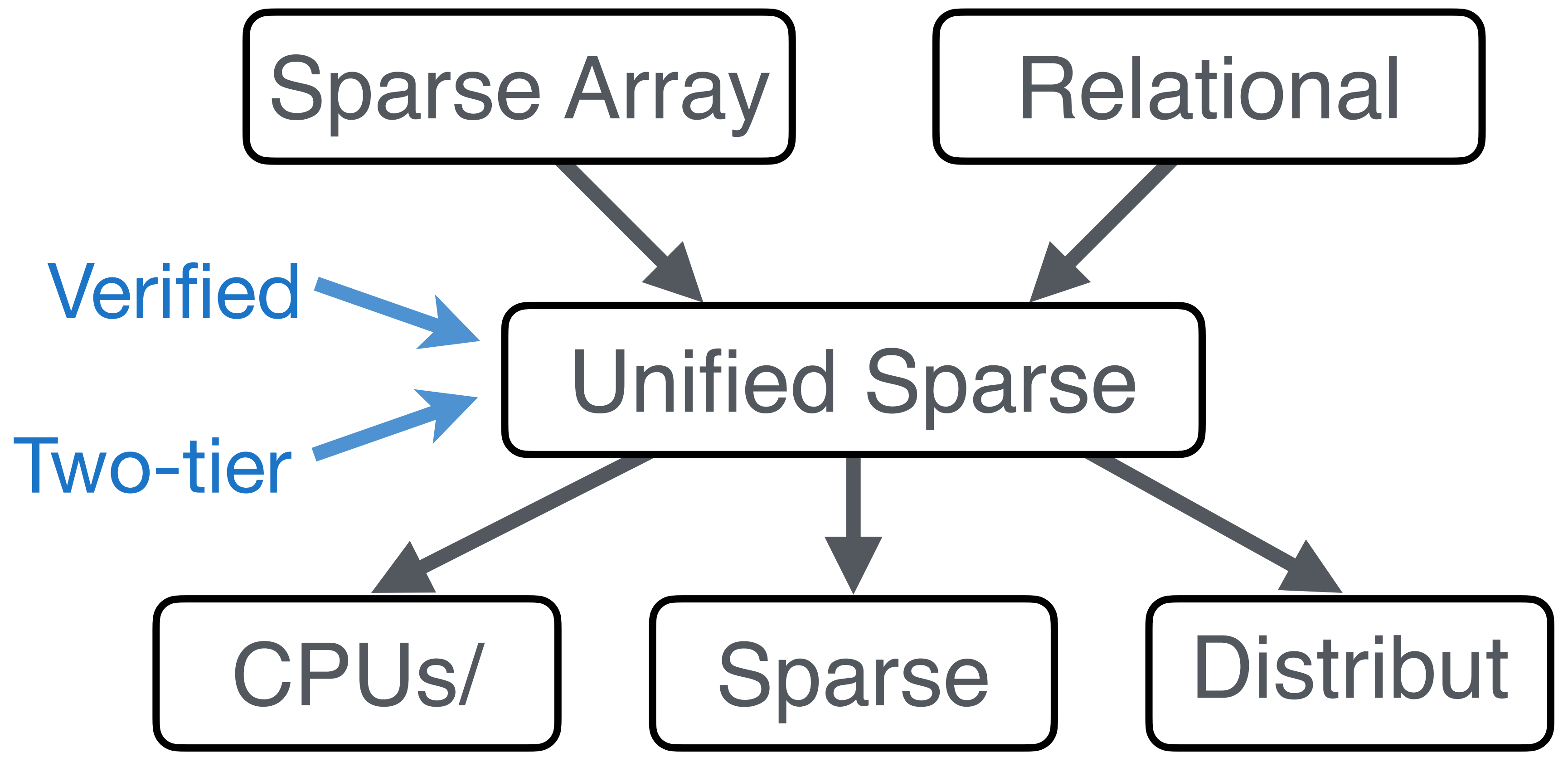
Commit Group

- Novel formats for
  - Faster computation
  - More compactness
  - More types of structured data
- Scheduling
  - Algorithmic Auto-scheduling
  - Learned Auto-scheduling
- Unifying Sparsity
  - Tensors: TACO
  - Graphs: GraphIt
- Hardware Support for Sparsity

Google

- Reimplementation of TACO sparse tensor theory in MLDIR

Stanford Compiler and Language Group



Quansight YOUR DATA EXPERTS

- Integrating TACO into Pydata Sparse

Commit Group

- Current & recent projects
  - SEQ: A DSL for bio informatics
  - TACO: A DSL for sparse tensor algebra
  - GraphIt: A DSL for graph analysts
  - BuildIt: A Multistage programming framework in C++
  - CoLa: A DSL for data compression
  - SimIt: A DSL for sparse systems
  - MILK: A DSL for Optimizing indirect memory references
  - Cimple: A DSL for Instruction and Memory Level Parallelism
  - Codon: A Pythonic DSL framework
  - Tiramisu: A polyhedral compiler for data parallel algorithms
  - Ithemal: Performance prediction using machine learning
  - VeGen: Generating Vectorizers for vector instructions beyond SIMD
  - Vemal: Vectorization using machine learning
  - goSLP & Revec: Modernizing vectorization technology
  - OpenTuner: An extensible framework for program autotuning

Thank You

http://tensor-compiler.org/

This Work Supported By:

ada
Applications Driving Architectures

DARPA

NSF

Semiconductor
Research
Corporation