Lecture 10:

Dense and Sparse Matrix Computation on GPUs

Modern Parallel Computing

John Owens

EEC 289Q, UC Davis, Winter 2018

Credits

- For their slides, thanks to Stan Tomov and Jack Dongarra, University of Tennessee; Kathy Yelick, UC Berkeley; Oded Green, Georgia Tech; Yao Zhang and Andrew Davidson, UC Davis; Carl Yang, UC Davis; and Duane Merrill, NVIDIA Research.

Project Proposals

- Preproposal Tue 20 Feb, proposal Tue 27 Feb—important!
- Single largest cause of problems in my experience
- What I want to see:
- Understanding of big-picture problem
  - With references as appropriate [I can help here]
- What you propose to do to help solve it
  - Small & complete beats big and vague
  - Convince me that your proposal is important
- How you will do that
  - Procedure
  - Schedule with milestones
  - Convince me that your proposal will work

EEC 277 Previous Projects (2013)

- Run-length Encoder and Huffman Encoder in CUDA
- Construction of Modified k-d trees on the GPU for Use in Particle-Based Flow Visualization
- Profiling Blowfish Encryption on GPU Architecture
- Ray casting parallelization with CUDA
- Solving the Transient Heat Conduction Equation With GPUs
- Physically-Based Material Interface Reconstruction
- OpenCL Implementation and Evaluation of the K-Nearest Neighbor Algorithm
- Network traffic aggregation on the GPU
- A GPU Implementation for Two-Dimensional Shallow Water Modeling
- Graph Clustering on CUDA by All Pairs Shortest Path Parallel Algorithms
- An Implementation of Connected Component Algorithm on GPU
- Parallel Implementation of Motion Graph
- Brute-Force MD5 Hash Cracking on the GPU
- Running CUDA on Ivy Bridge GPUs

Parallel Computing Patterns

Phillip Colella's "Seven dwarfs"

High-end simulation in the physical sciences = 7 numerical methods:

1. Structured Grids (including locally structured grids, e.g. Adaptive Mesh Refinement)
2. Unstructured Grids
3. Fast Fourier Transform
4. Dense Linear Algebra
5. Sparse Linear Algebra
6. Particles
7. Monte Carlo

- A dwarf is a pattern of computation and communication
- Dwarfs are well-defined targets from algorithmic, software, and architecture standpoints

Slide from "Defining Software Requirements for Scientific Computing", Phillip Colella, 2004

Do dwarfs work well outside HPC?

- Examine use of 7 dwarfs elsewhere

1. Embedded Computing (EEMBC benchmark)
2. Desktop/Server Computing (SPEC2006)
3. Data Base / Text Mining Software
   - Advice from Jim Gray of Microsoft and Joe Hellerstein of UC
4. Games/Graphics/Vision
5. Machine Learning
   - Advice from Mike Jordan and Dan Klein of UC Berkeley

- Result: Added 7 more dwarfs, revised 2 original dwarfs, renumbered list

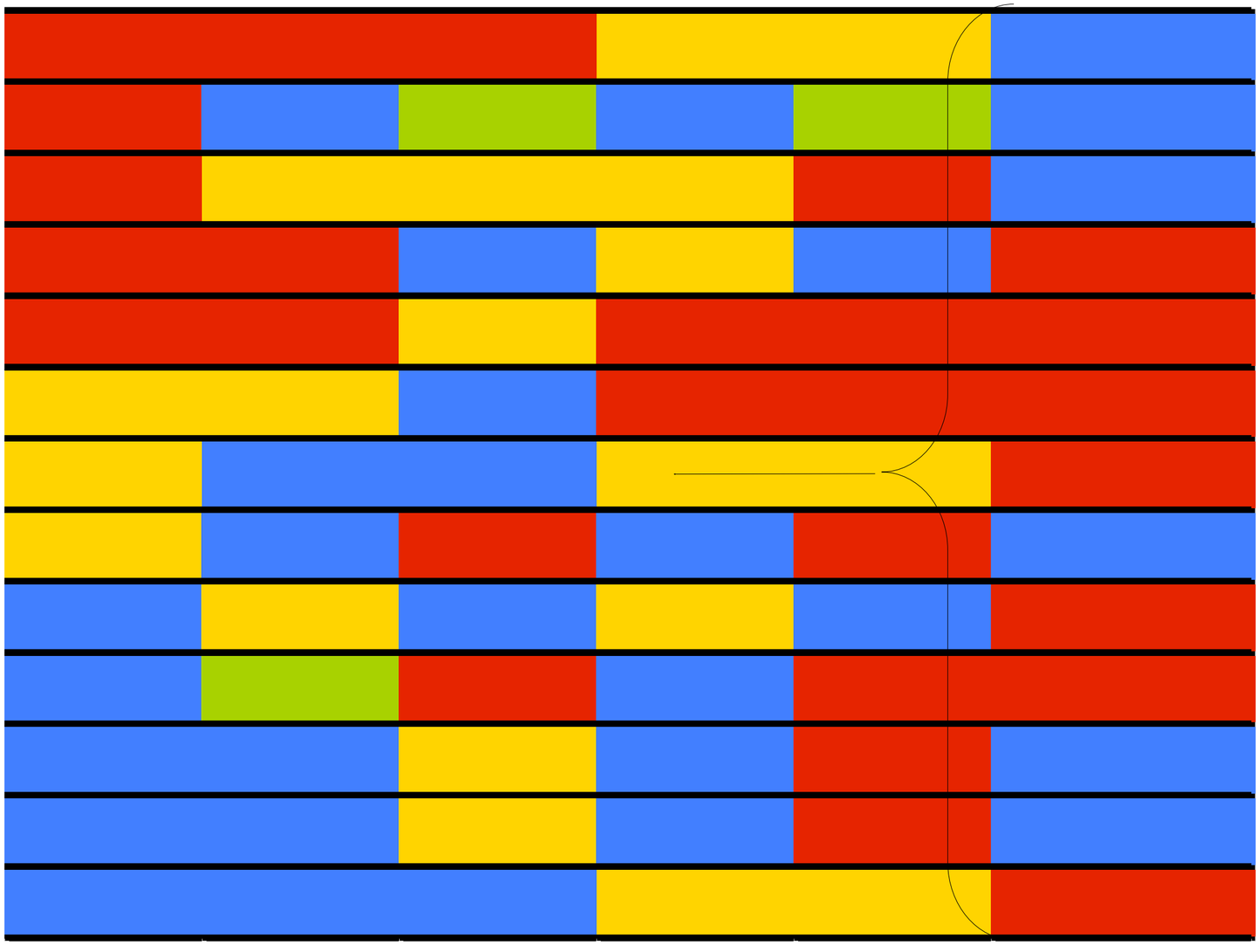
Dwarf Use (Red Important → Blue Not)







Embed SPEC DB Games ML HPC



1 Finite State Mach.
2 Combinational
3 Graph Traversal
4 Structured Grid
5 Dense Matrix
6 Sparse Matrix
7 Spectral (FFT)
8 Dynamic Prog
9 Particles
10 MapReduce
11 Backtrack/ B&B
12 Graphical Models
13 Unstructured Grid

Roles of Dwarfs

1. Give us a vocabulary/organization to talk across disciplinary boundaries
2. Define minimum set of necessary functionality for new hardware/software systems
3. Define building blocks for creating libraries that cut across application domains
4. “Anti-benchmarks” not tied to code or language artifacts ⇒ encourage innovation in algorithms, languages, data structures, and/or hardware
5. They decouple research, allowing analysis of HW & SW programming support without waiting years for full app development

Dense Matrices

What is dense linear algebra?

- Not just matmul!
- Linear Systems: Ax=b
- Least Squares: choose x to minimize $||Ax-b||_2$
  - Overdetermined or underdetermined
  - Unconstrained, constrained, weighted
- Eigenvalues and vectors of Symmetric Matrices
    - Standard ($Ax = \lambda x$), Generalized ($Ax=\lambda Bx$)
- Eigenvalues and vectors of Unsymmetric matrices
    - Eigenvalues, Schur form, eigenvectors, invariant subspaces
    - Standard, Generalized
- Singular Values and vectors (SVD)
  - Standard, Generalized
- Different matrix structures
  - Real, complex; Symmetric, Hermitian, positive definite; dense, triangular, banded …
- Level of detail
  - Simple Driver (“x=A\b”)
  - Expert Drivers with error bounds, extra-precision, other options
  - Lower level routines (“apply certain kind of orthogonal transformation”, matmul…)

Dense Linear Algebra

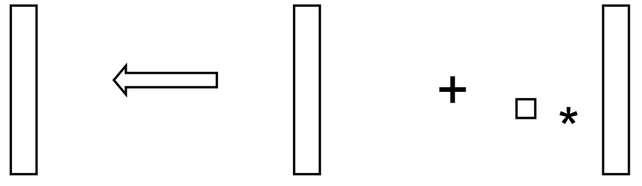
- Common Operations

$$Ax = b; \quad \min_x \| Ax - b \|; \quad Ax = \lambda x$$

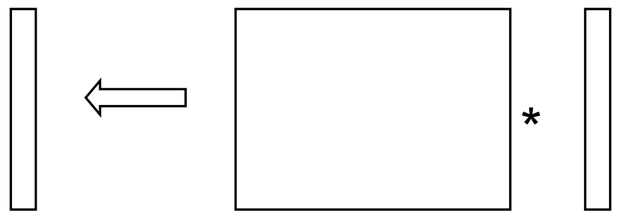
- A major source of large dense linear systems is problems involving the solution of boundary integral equations.
  - The price one pays for replacing three dimensions with two is that what started as a sparse problem in $O(n^3)$ variables is replaced by a dense problem in $O(n^2)$.
- Dense systems of linear equations are found in numerous other applications, including:
  - airplane wing design;
  - radar cross-section studies;
  - flow around ships and other off-shore constructions;
  - diffusion of solid bodies in a liquid;
  - noise reduction; and
  - diffusion of light through small particles.

Level 1, 2 and 3 BLAS

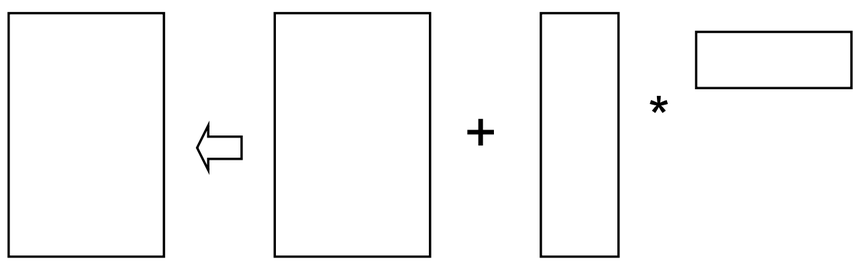
- **Level 1 BLAS**
  Vector-Vector operations



- **Level 2 BLAS**
  Matrix-Vector operations



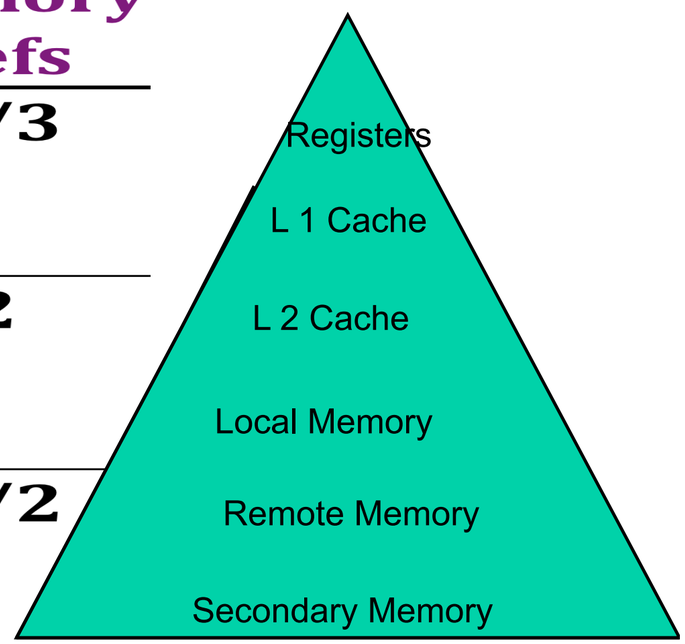
- **Level 3 BLAS**
  Matrix-Matrix operations



Why Higher Level BLAS?

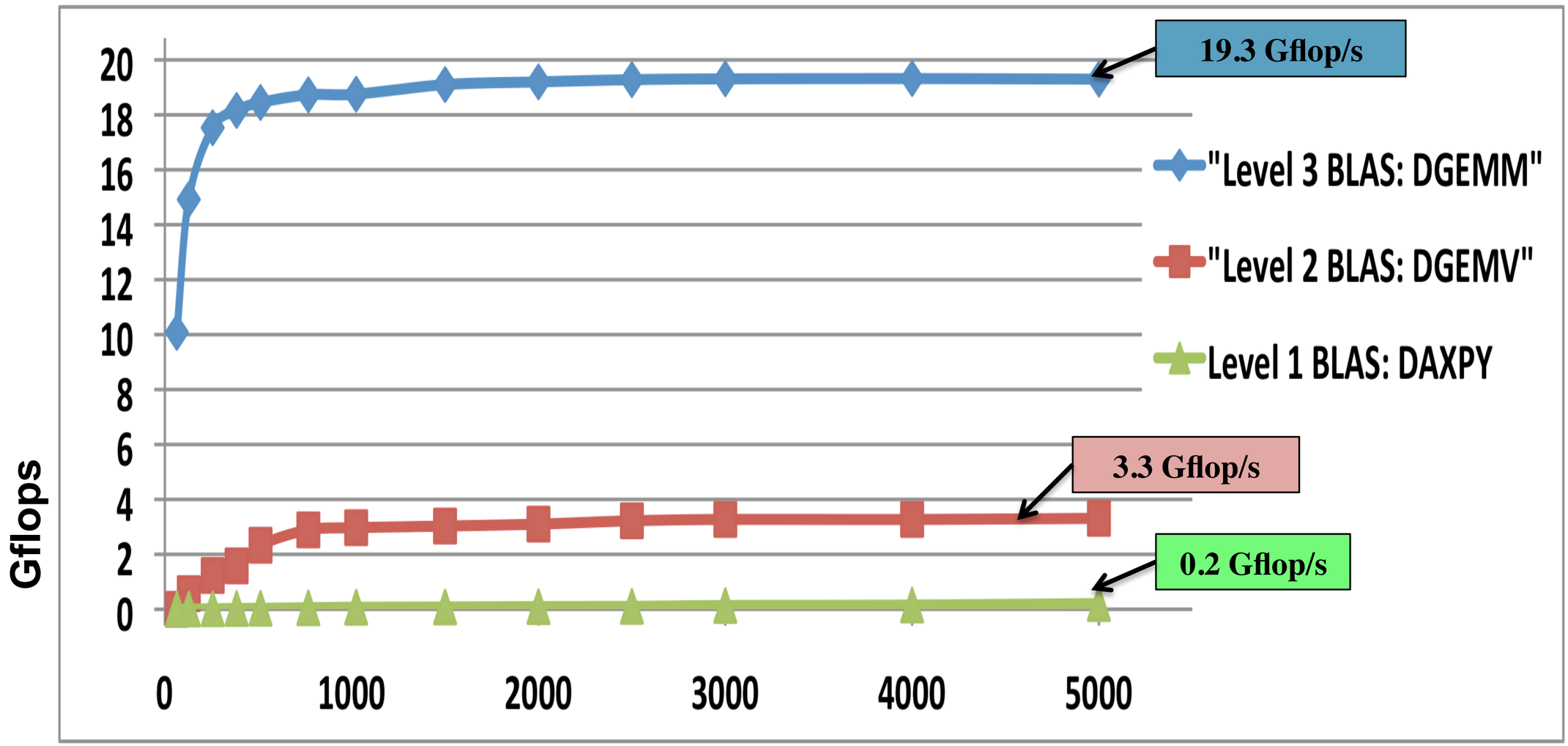
- Can only do arithmetic on data at the top of the hierarchy
- Higher level BLAS lets us do this

BLAS	Memory Refs	Flops	Flops/ Memory Refs
Level 1 $y=y+\alpha x$	$3n$	$2n$	$2/3$
Level 2 $y=y+Ax$	n^2	$2n^2$	2
Level 3 $C=C+AB$	$4n^2$	$2n^3$	$n/2$



Level 1, 2 and 3 BLAS

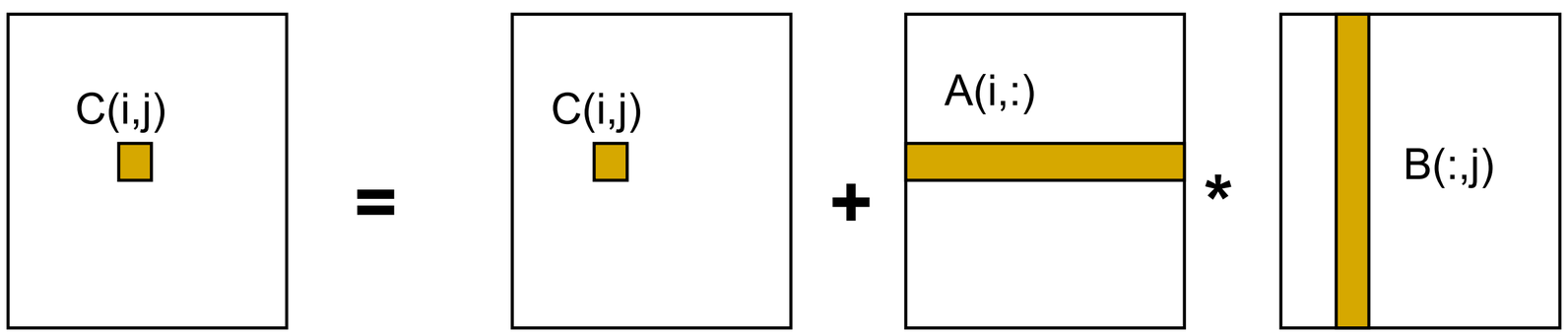
1 core Intel Xeon E5-2670 (Sandy Bridge); 2.6 GHz; Peak = 20.8 Gflop/s



1 core Intel Xeon E5-2670 (Sandy Bridge), 2.6 GHz.
24 MB shared L3 cache, and each core has a private 256 KB L2 and 64 KB L1.
The theoretical peak per core DP is 8 flop/cycle * 2.6 GHz = 20.8 Gflop/s per core.
Compiled with gcc 4.4.6 and using MKL_composer_xe_2013.3.163

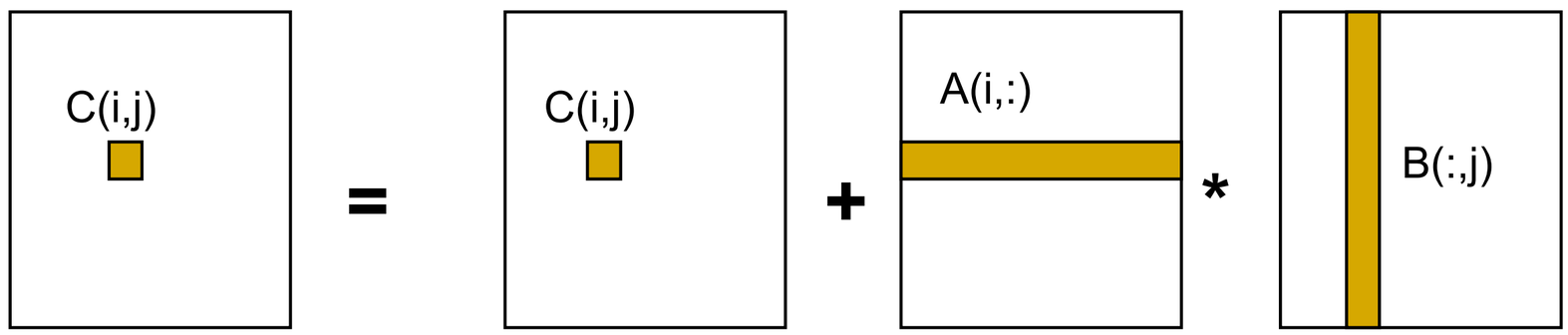
Matrix Multiply C=C+A*B

```
for i = 1 to n
        for j = 1 to n
           for k = 1 to n
                                C(i,j) = C(i,j) + A(i,k) * B(k,j)
```



Matrix Multiply C=C+A*B
(unblocked, or untiled)

```
for i = 1 to n
  {read row i of A into fast memory}
   for j = 1 to n
       {read C(i,j) into fast memory}
       {read column j of B into fast memory}
       for k = 1 to n
           C(i,j) = C(i,j) + A(i,k) * B(k,j)
       {write C(i,j) back to slow memory}
```



Matrix Multiply C=C+A*B (unblocked, or untiled)

q=ops/slow mem ref

Number of slow memory references on unblocked matrix multiply

m = n^3 read each column of B n times

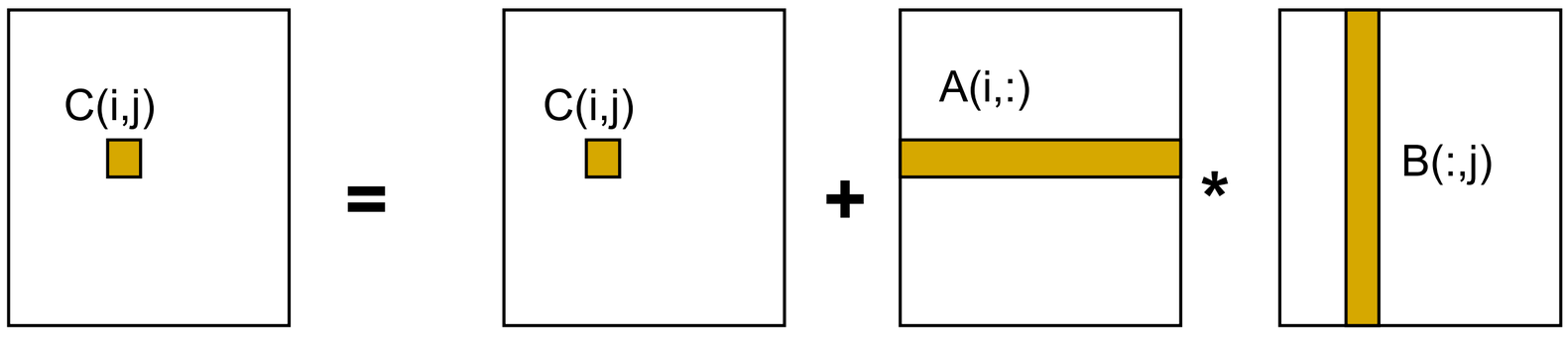
+ n^2 read each row of A once for each i

+ $2*n^2$ read and write each element of C once

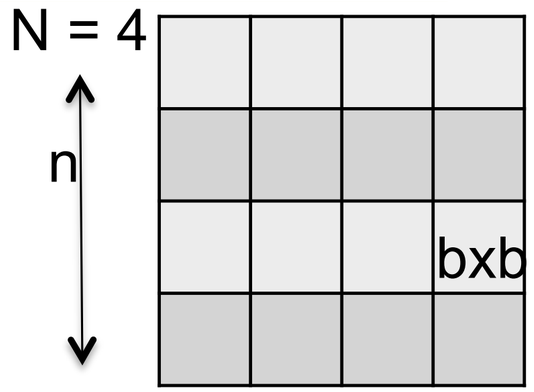
= $n^3 + 3*n^2$

So q = f/m = $(2*n^3)/(n^3 + 3*n^2)$

~= 2 for large n, no improvement over matrix-vector mult

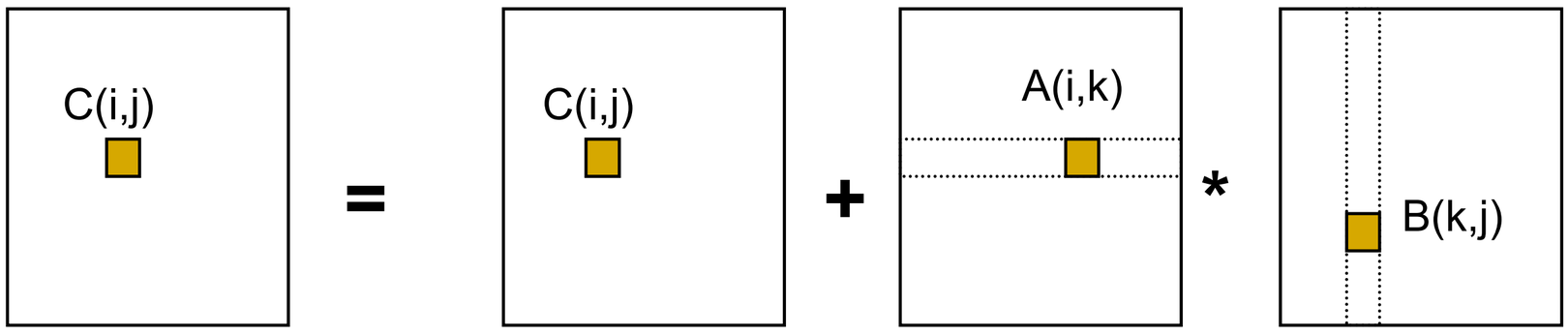


Matrix Multiply (blocked, or tiled)



Consider A,B,C to be N by N matrices of b by b subblocks where b=n/N is called the blocksize

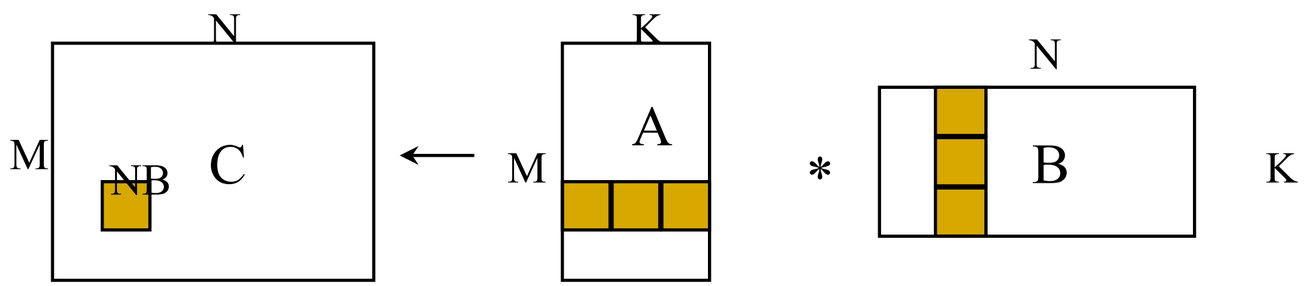
```
for i = 1 to N
    for j = 1 to N
        {read block C(i,j) into fast memory}                    N² times
        for k = 1 to N
            {read block A(i,k) into fast memory}                N³ times
            {read block B(k,j) into fast memory}                N³ times
            C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
        {write block C(i,j) back to slow memory}
```



Cache Blocking

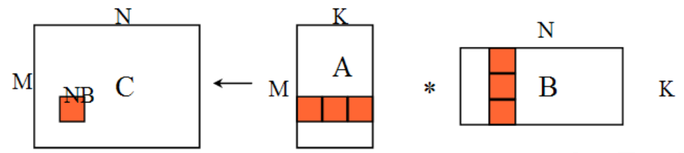
Looping over the blocks

```
do kk = 1,n,nblk
   do jj = 1,n,nblk
      do ii = 1,n,nblk
         do k = kk,kk+nblk-1
            do j = jj,jj+nblk-1
               do i = ii,ii+nblk-1
                  c(i,j) = c(i,j) + a(i,k) * b(k,j)
               end do
            end do
         end do
         . . .
      end do
   end do
end do
```



Matrix multiply of block

Matrix Multiply (blocked or tiled)



A, B, C made up of
NxN blocks of size b (n/N)

Why is this algorithm correct?

Number of slow memory references on blocked matrix multiply
m = $N*n^2$ read each block of B N^3 times (N^3 * n/N * n/N)
+ $N*n^2$ read each block of A N^3 times (N^3 * n/N * n/N)
+ $2*n^2$ read and write each block of C once
= $(2*N + 2)*n^2$
So q = f/m = $2*n^3$ / $((2*N + 2)*n^2)$
~= n/N = b for large n

So we can improve performance by increasing the blocksize b
Can be much faster than matrix-vector multiply (q=2)

q=ops/slow mem ref

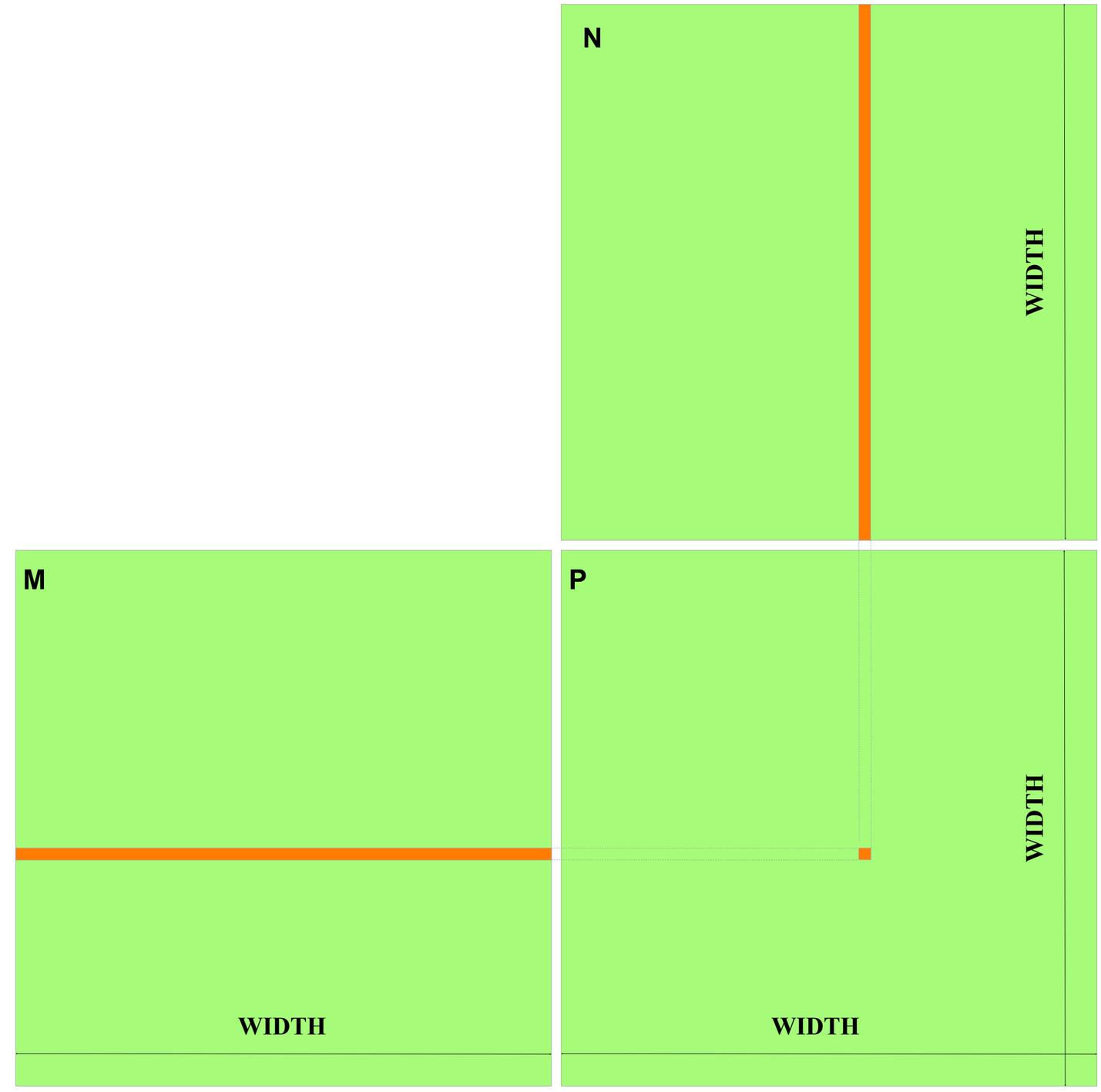
n size of matrix
b blocksize (n/N)
N number of blocks

Limit: All three blocks from A,B,C must fit in fast memory (cache), so we cannot make these blocks arbitrarily large: $3*b^2$ <= M, so q ~= b <= sqrt(M/3)

Theorem (Hong, Kung, 1981): Any reorganization of this algorithm (that uses only associativity) is limited to q =O(sqrt(M))

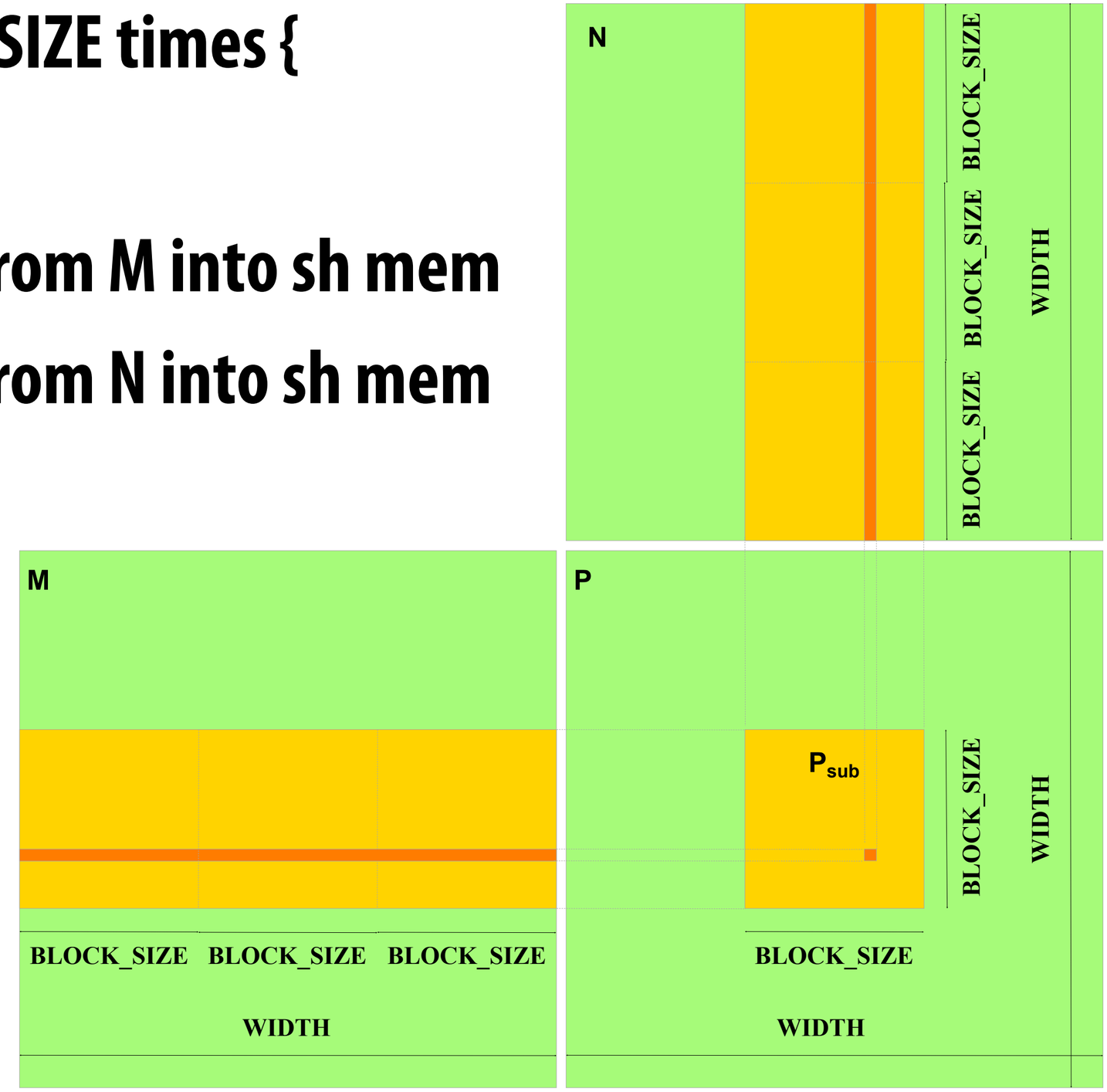
Dense Matrix Multiplication

- for all elements E in destination matrix P
  - $P_{r,c} = M_r \bullet N_c$



Dense Matrix Multiplication

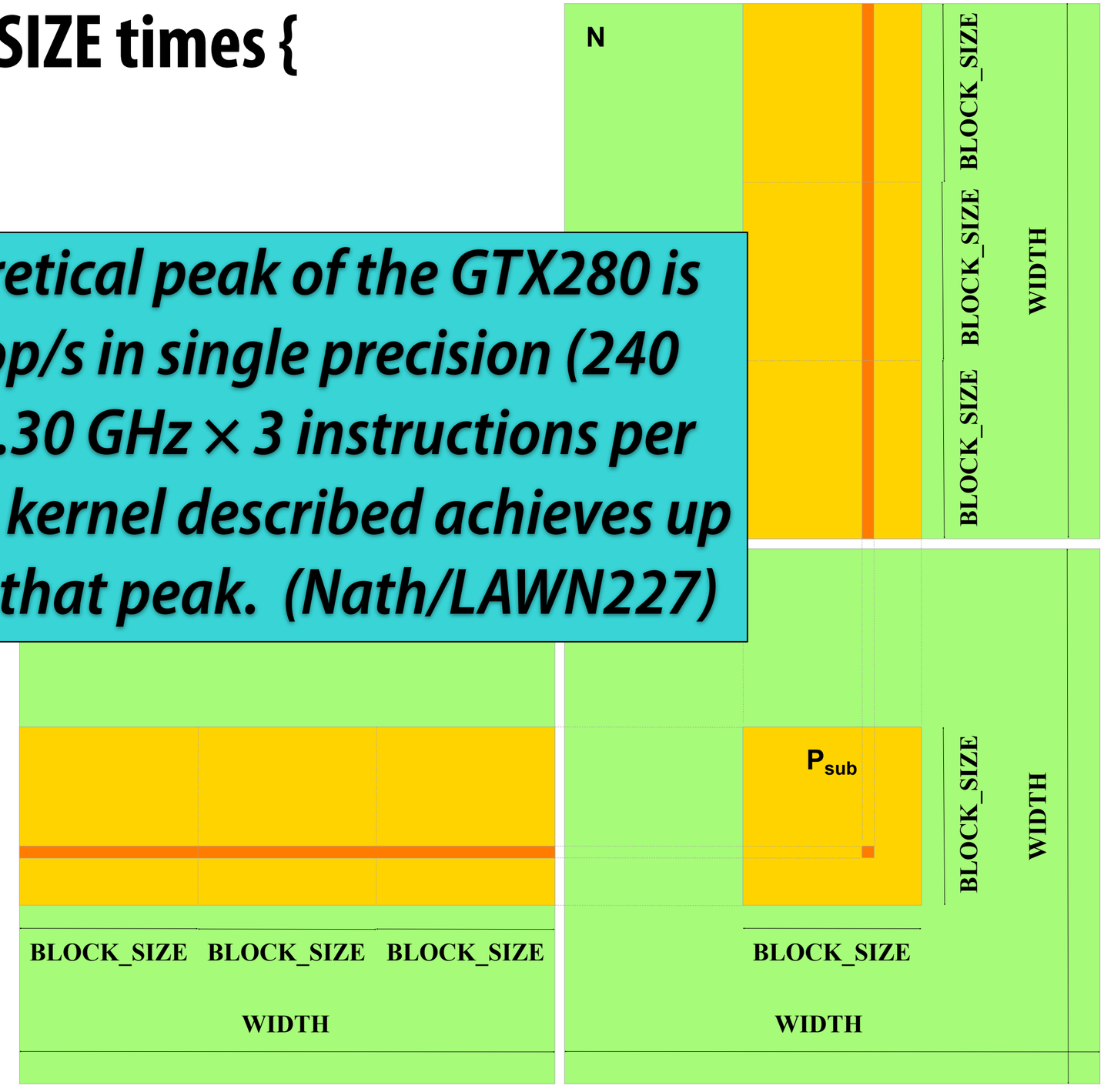
- P = M * N of size WIDTH x WIDTH
- Repeat WIDTH / BLOCK_SIZE times {
  - Each thread block:
    - Reads a subblock from M into sh mem
    - Reads a subblock from N into sh mem
  - Each thread:
    - Computes a mini-dot-product
- Output total sum



Dense Matrix Multiplication

- P = M * N of size WIDTH x WIDTH
- Repeat WIDTH / BLOCK_SIZE times {
  - Each thread block:
    - Reads a
    - Reads a
  - Each threa
    - Comput
      mini-dot-product
- Output total sum

The theoretical peak of the GTX280 is 936 GFlop/s in single precision (240 cores × 1.30 GHz × 3 instructions per cycle). The kernel described achieves up to 40% of that peak. (Nath/LAWN227)

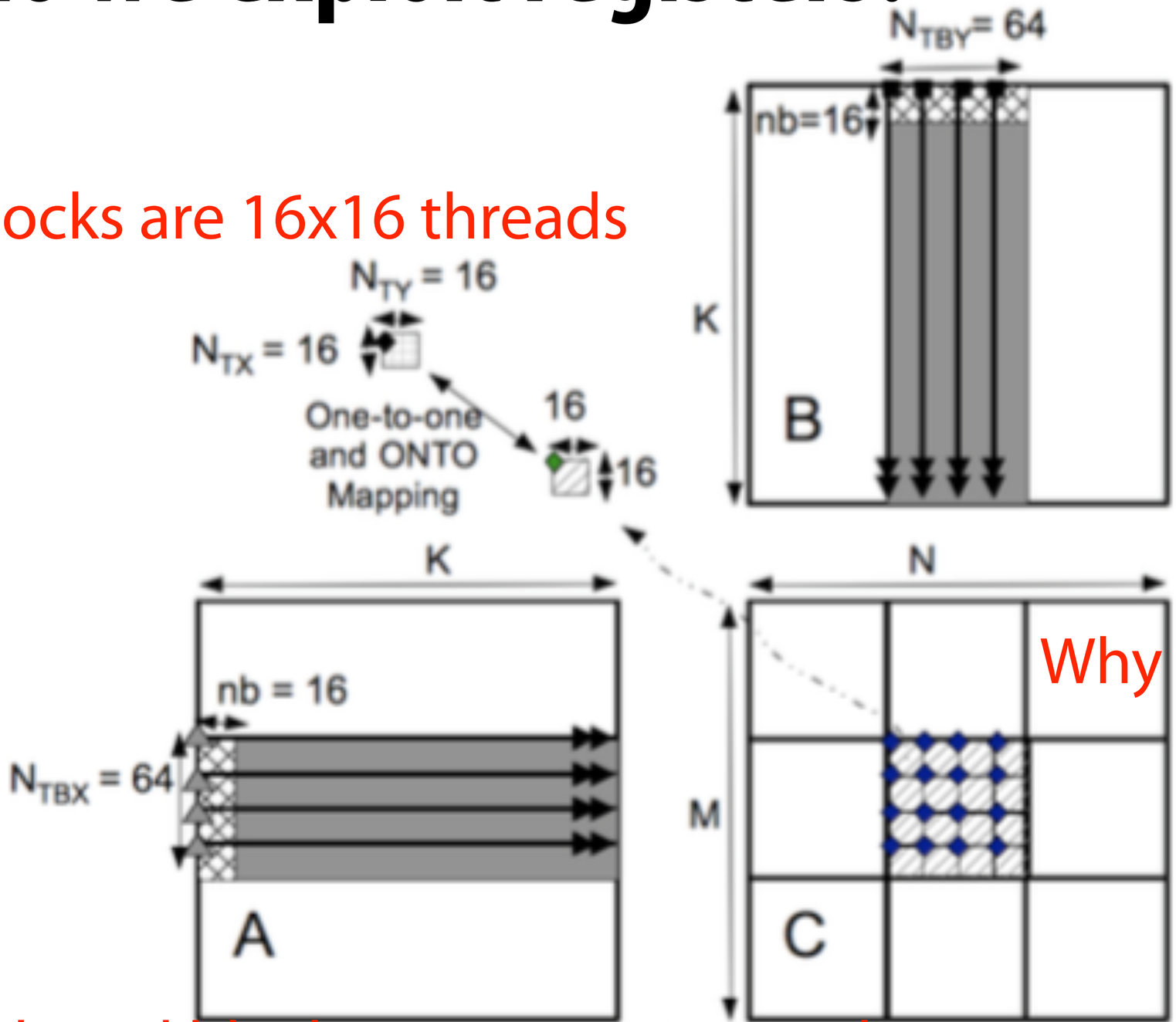


Moving from Tesla to Fermi

- "An Improved MAGMA GEMM for Fermi GPUs", Rajib Nath, Stanimire Tomov, and Jack Dongarra, LAPACK Working Note 227 (lawn227), 20 July 2010
- "Scaling up compute capabilities": "increased shared memory, number of registers, number of CUDA cores in a multiprocessor, etc."
  - Previous algorithm can be tuned to leverage these
- But: "changes that are related to added cache memories, and most importantly, that the latency to access register and shared memory were comparable in 1.x GPUs, but not in the Fermi (where accessing data from registers is much faster)"
  - Solution: "add one more level of blocking in the algorithm, namely register blocking, to account for the added memory hierarchy"

How do we exploit registers?

Thread blocks are 16x16 threads

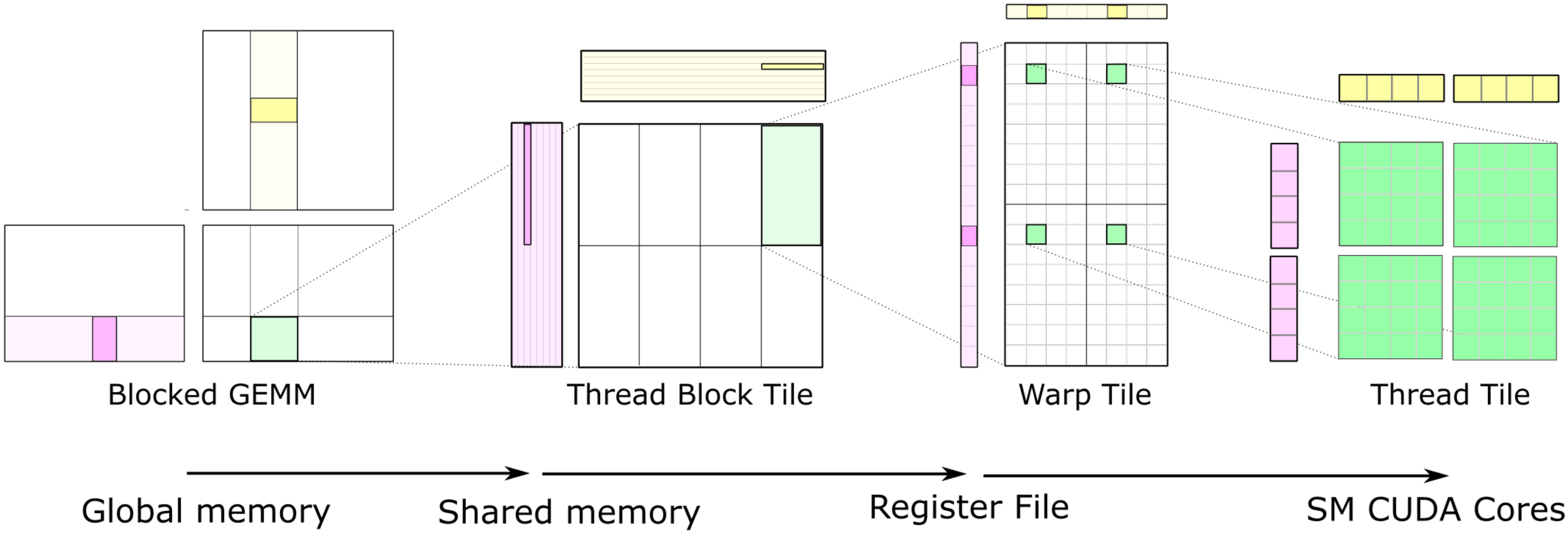


Why this layout?

Each thread block computes 64x64 elements; each thread computes a 4x4 output matrix with stride 16.

- **"Note that the theoretical peak of the Fermi, in this case a C2050, is 515 GFlop/s in double precision (448 cores × 1.15 GHz × 1 instruction per cycle). The kernel described achieves up to 58% of that peak."**

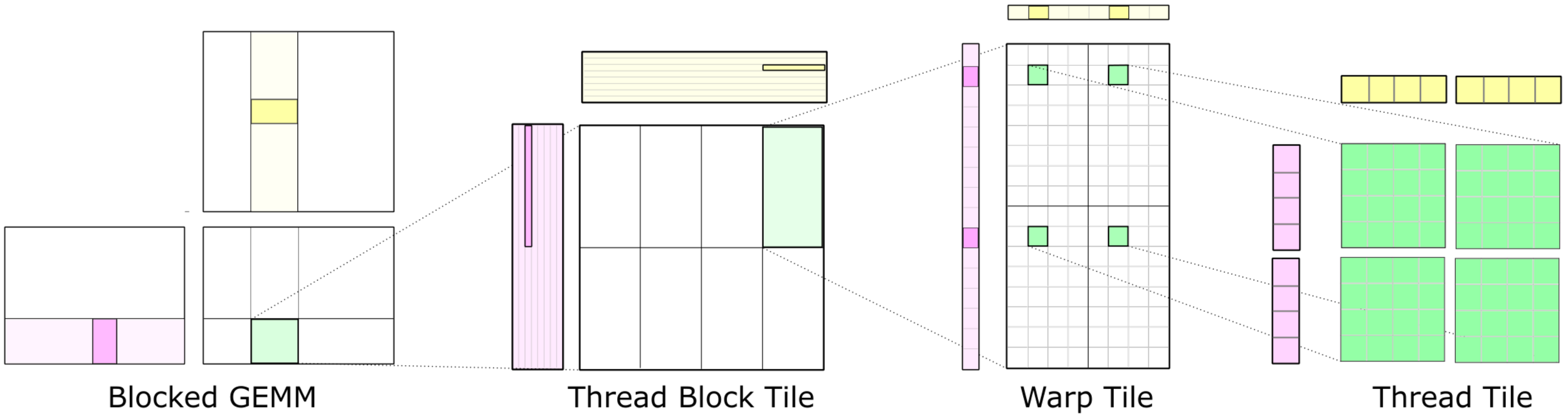
NVIDIA CUTLASS



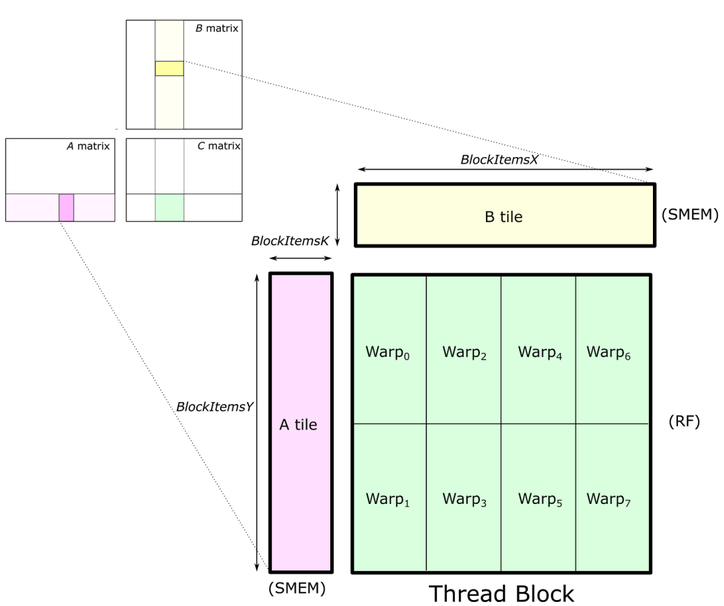
https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/

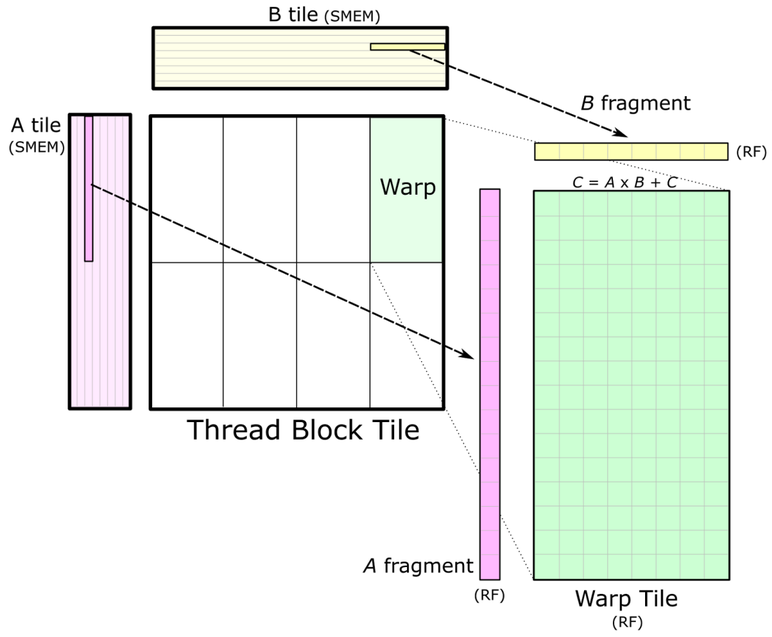
NVIDIA CUTLASS

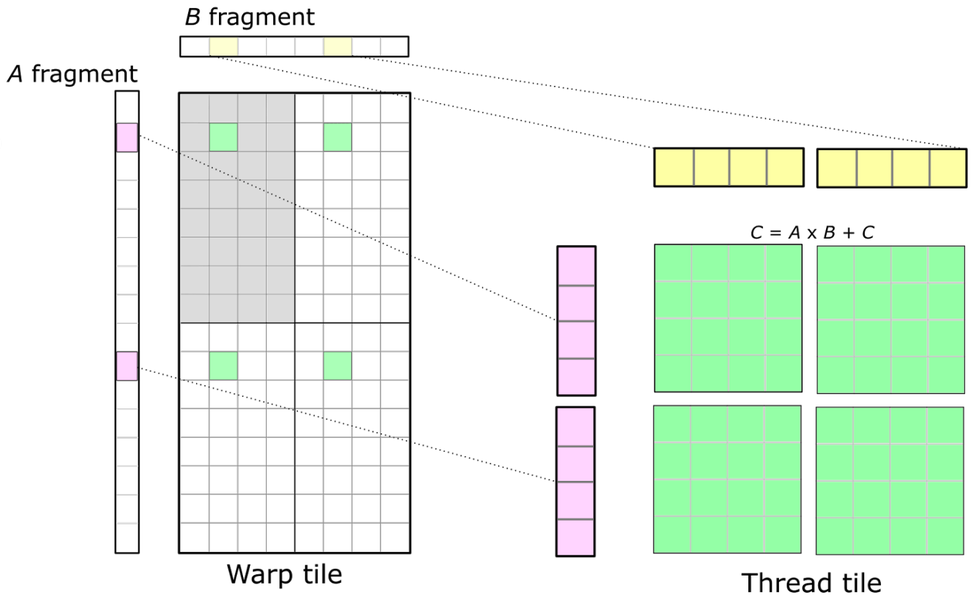
https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/



Global memory Shared memory Register File SM CUDA Cores



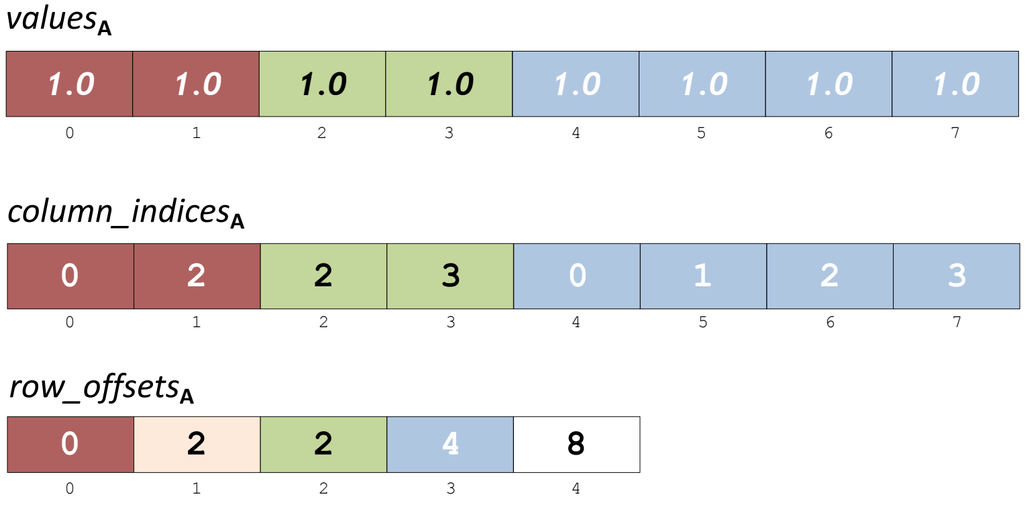




Sparse Matrix Basics

Sequential CSR Sparse-Matrix, Dense-Vector Multiplication

$$\begin{bmatrix} 1.0 & -- & 1.0 & -- \\ -- & -- & -- & -- \\ -- & -- & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 & 1.0 \end{bmatrix}$$

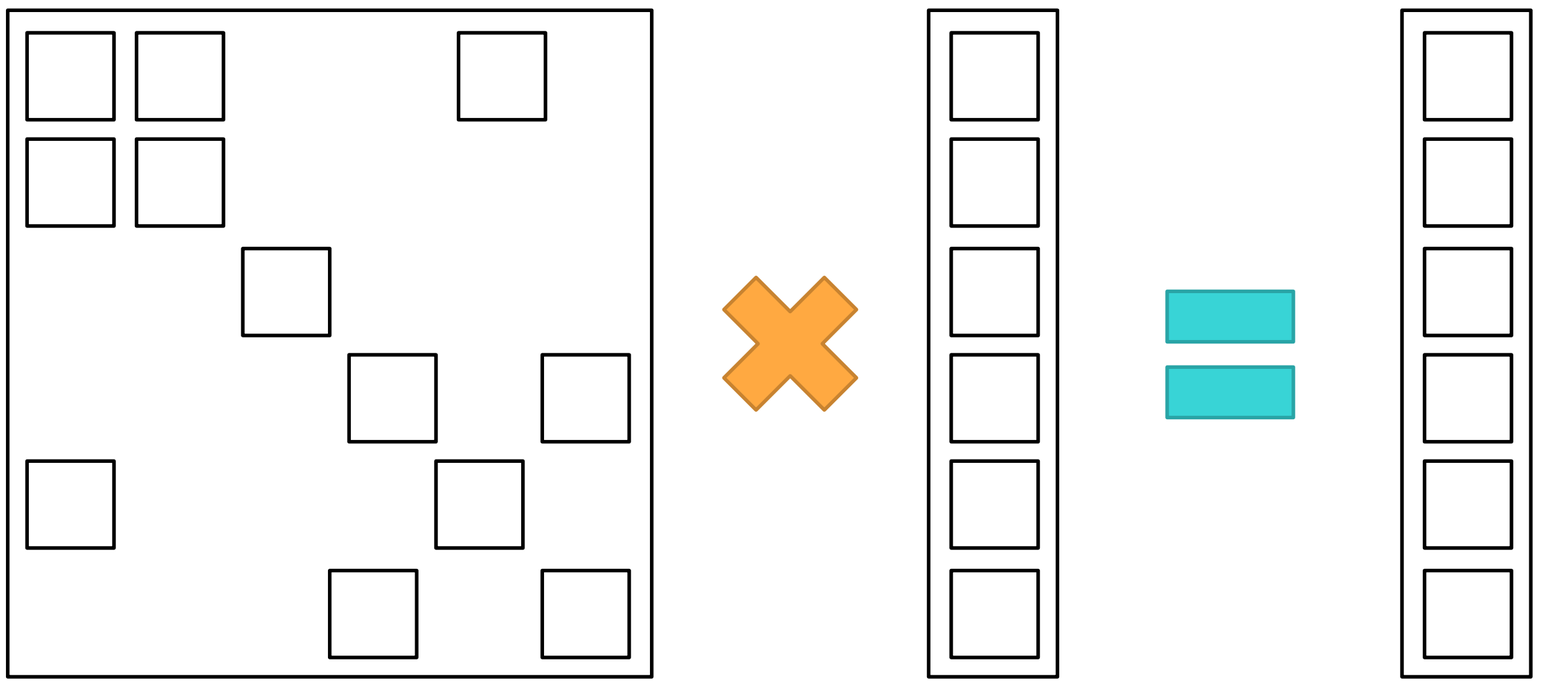


- **Input: CSR matrix A, dense vector x**
  Output: Dense vector y such that y <- Ax

```
for (int row = 0; row < A.m; ++row) {
  y[row] = 0.0;
  for (int nz = A.row_offsets[row]; nz < A.row_offsets[row + 1]; ++nz) {
    y[row] += A.values[nz] * x[A.column_indices[nz]];
  }
}
```

Sparse Matrix-Vector Multiply: What's Hard?

- Dense approach is wasteful
- Unclear how to map work to parallel processors
- Irregular data access
- Wildly varying input sizes/characteristics



Sparse Matrix Formats

(DIA) Diagonal

(ELL) ELLPACK

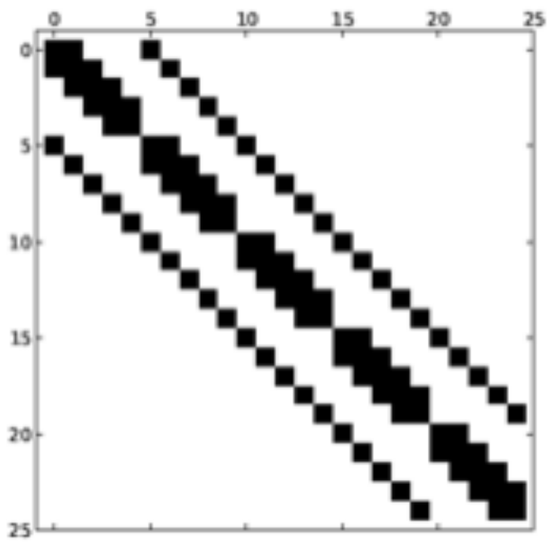
(CSR) Compressed Row

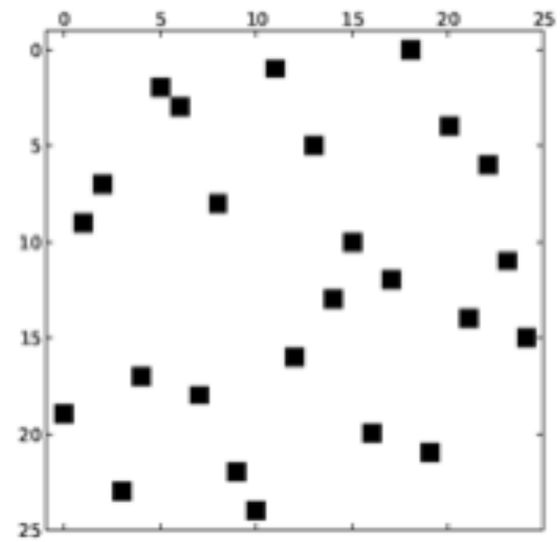
(HYB) Hybrid

(COO) Coordinate

Structured

Unstructured



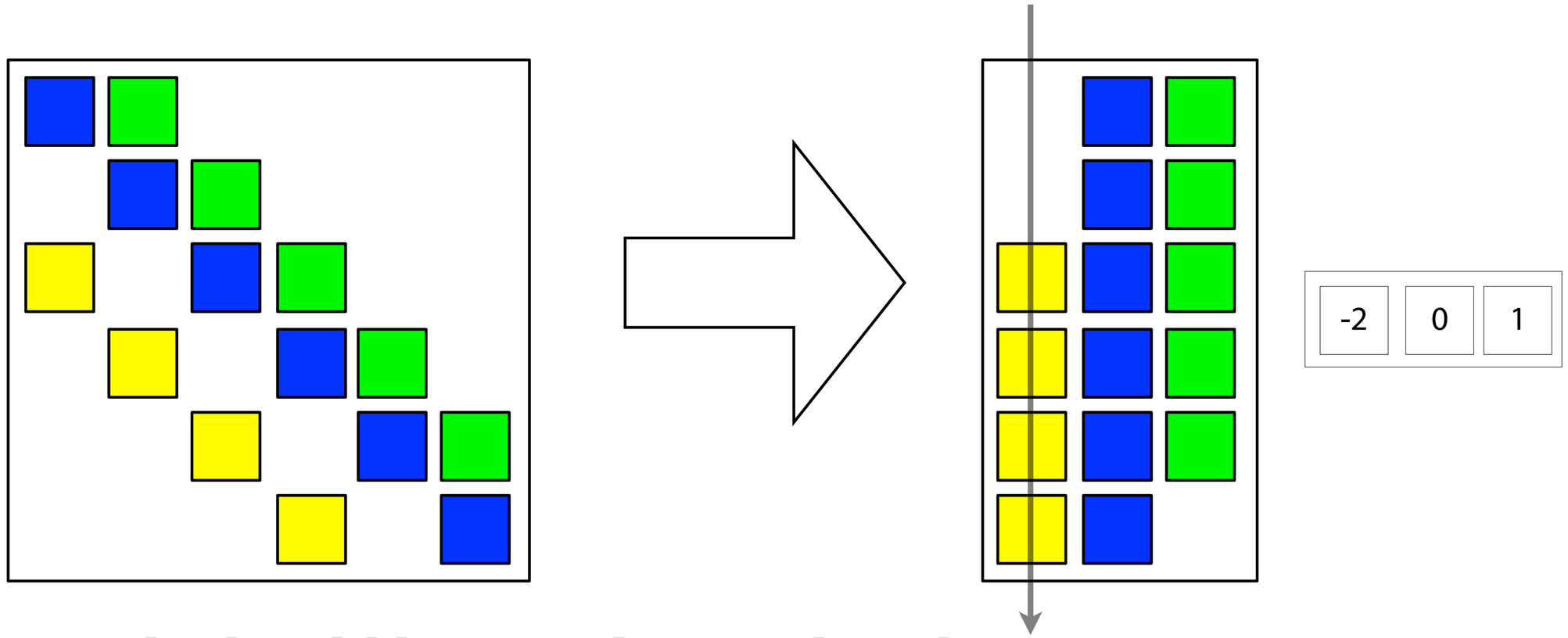


Tridiagonal Solvers

What is a tridiagonal system?

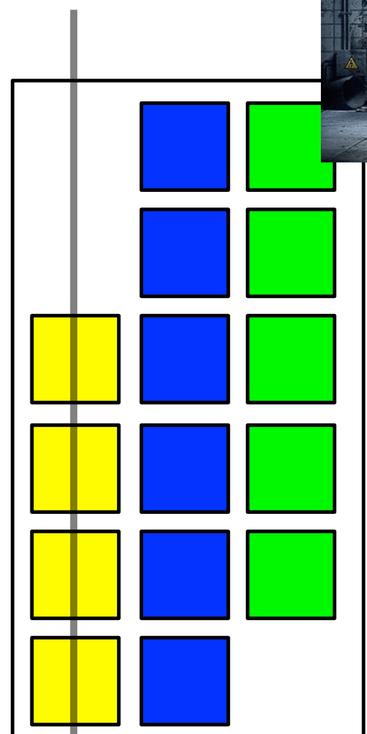
$$\begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & a_3 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ \vdots \\ d_n \end{pmatrix}$$

Diagonal Matrices



- Diagonals should be mostly populated
- Map one thread per row
  - Good parallel efficiency
  - Good memory behavior [column-major storage]

Diagonal Matrices



-2	0	1

[c

What is it used for?

- Scientific and engineering computing
  - Alternating direction implicit (ADI) methods
  - Numerical ocean models
  - Semi-coarsening for multi-grid solvers
  - Spectral Poisson Solvers
  - Cubic Spline Approximation
- Video games and computer-animated films
  - Depth of field blurs
  - Fluid simulation

Two Real-time Applications on GPU

Depth of field blur, Kass et al.

OpenGL and Shader language

Cyclic reduction

2006

Shallow water simulation, Sengupta et al.

CUDA

Cyclic reduction

2007

A Classic Sequential Algorithm

- Gaussian elimination in tridiagonal case (Thomas algorithm)

A Classic Sequential Algorithm

- Gaussian elimination in tridiagonal case (Thomas algorithm)

$$\begin{pmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & c_3 & \\ & & a_4 & b_4 & c_4 \\ & & & a_5 & b_5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \end{pmatrix}$$

A Classic Sequential Algorithm

- Gaussian elimination in tridiagonal case (Thomas algorithm)

$$\begin{pmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & c_3 & \\ & & a_4 & b_4 & c_4 \\ & & & a_5 & b_5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \end{pmatrix}$$

Phase 1: Forward Elimination

A Classic Sequential Algorithm

- Gaussian elimination in tridiagonal case (Thomas algorithm)

$$\begin{pmatrix} 1 & c'_1 & & & \\ 0 & 1 & c'_2 & & \\ & 0 & 1 & c'_3 & \\ & & 0 & 1 & c'_4 \\ & & & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} d'_1 \\ d'_2 \\ d'_3 \\ d'_4 \\ d'_5 \end{pmatrix}$$

Phase 1: Forward Elimination

A Classic Sequential Algorithm

- Gaussian elimination in tridiagonal case (Thomas algorithm)

$$\begin{pmatrix} 1 & c'_1 & & & \\ 0 & 1 & c'_2 & & \\ & 0 & 1 & c'_3 & \\ & & 0 & 1 & c'_4 \\ & & & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} d'_1 \\ d'_2 \\ d'_3 \\ d'_4 \\ d'_5 \end{pmatrix}$$

Phase 2: Backward Substitution

A Classic Sequential Algorithm

- Gaussian elimination in tridiagonal case (Thomas algorithm)

$$\begin{pmatrix} 1 & c'_1 & & & \\ 0 & 1 & c'_2 & & \\ & 0 & 1 & c'_3 & \\ & & 0 & 1 & c'_4 \\ & & & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} d'_1 \\ d'_2 \\ d'_3 \\ d'_4 \\ d'_5 \end{pmatrix}$$

Phase 2: Backward Substitution

Cyclic Reduction (CR)

Cyclic Reduction (CR)

8-unknown system

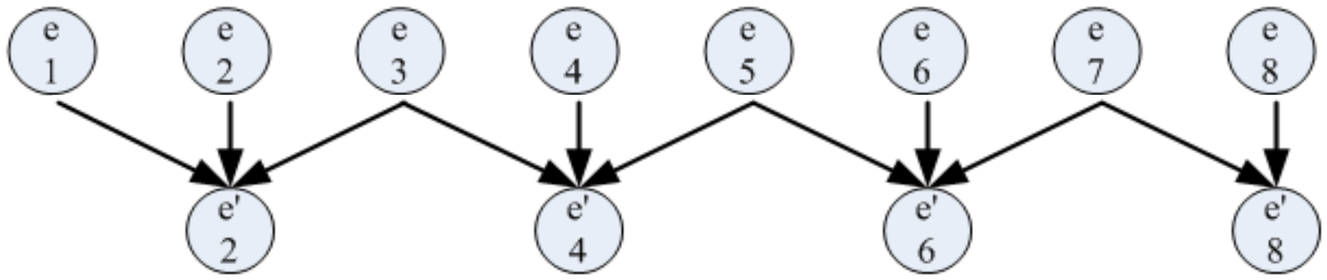
Cyclic Reduction (CR)

8-unknown system

Forward Reduction

Cyclic Reduction (CR)

4 threads working



Forward Reduction

8-unknown system

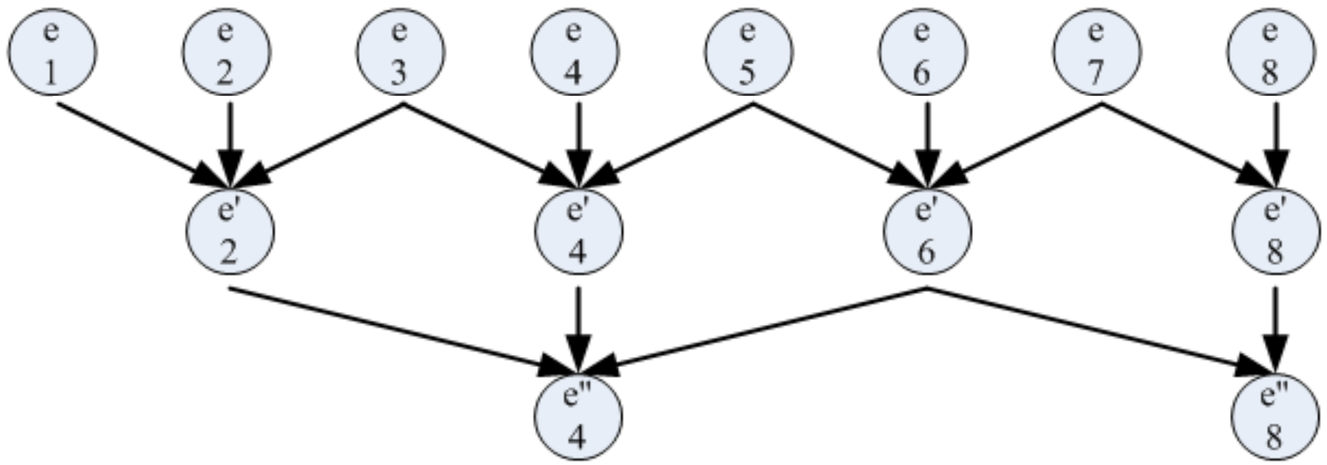
⇩

4-unknown system

Cyclic Reduction (CR)

2 threads working

Forward Reduction



8-unknown system

⇩

4-unknown system

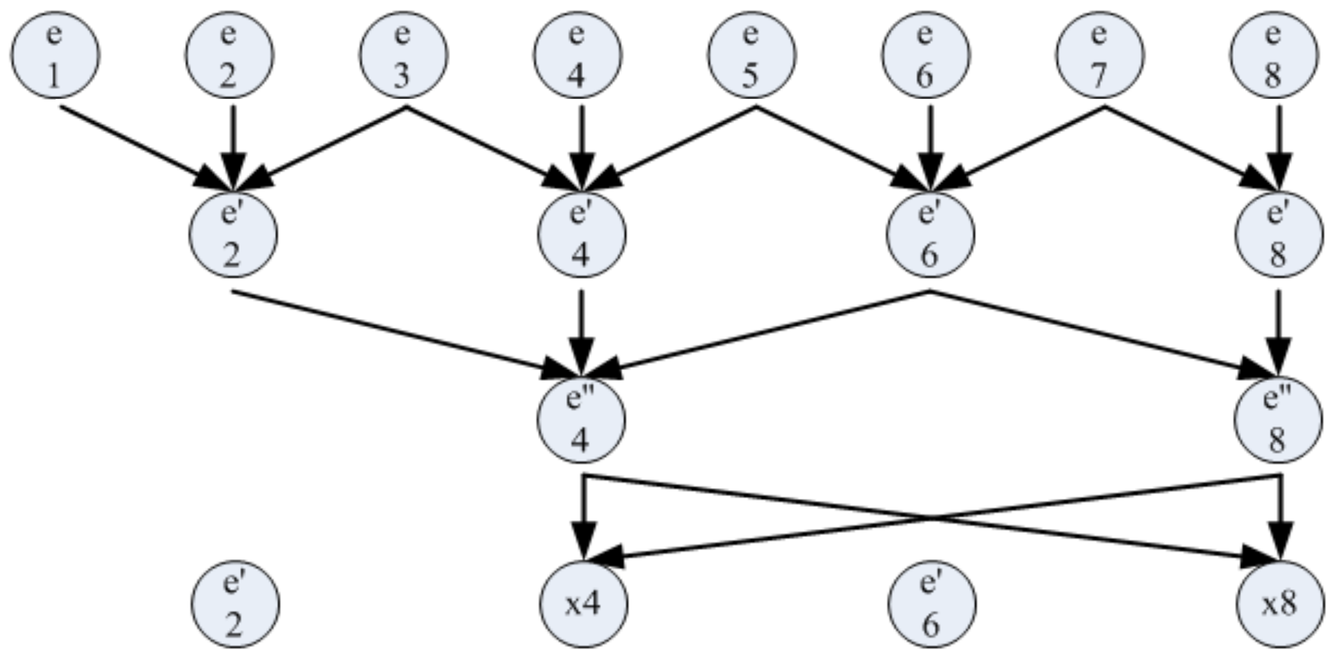
⇩

2-unknown system

Cyclic Reduction (CR)

1 thread working

Forward Reduction



8-unknown system

⇩

4-unknown system

⇩

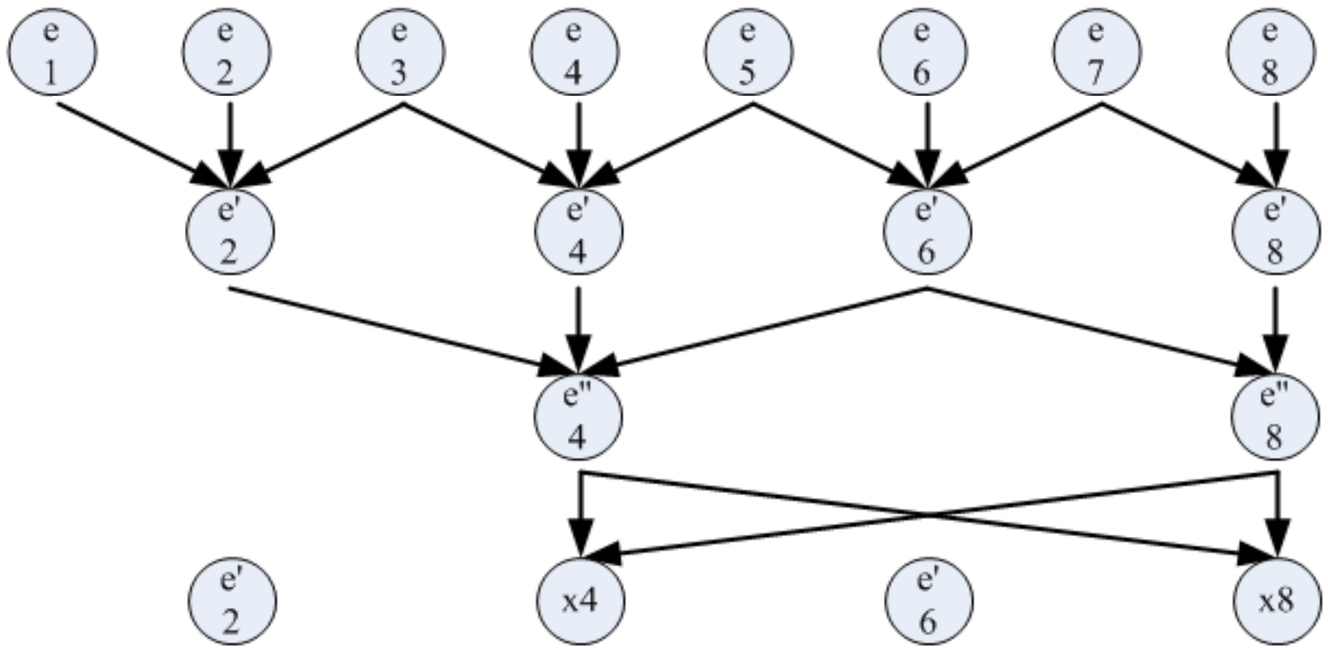
2-unknown system

⇩

Solve 2 unknowns

Cyclic Reduction (CR)

1 thread working



Forward Reduction

8-unknown system

⇩

4-unknown system

⇩

2-unknown system

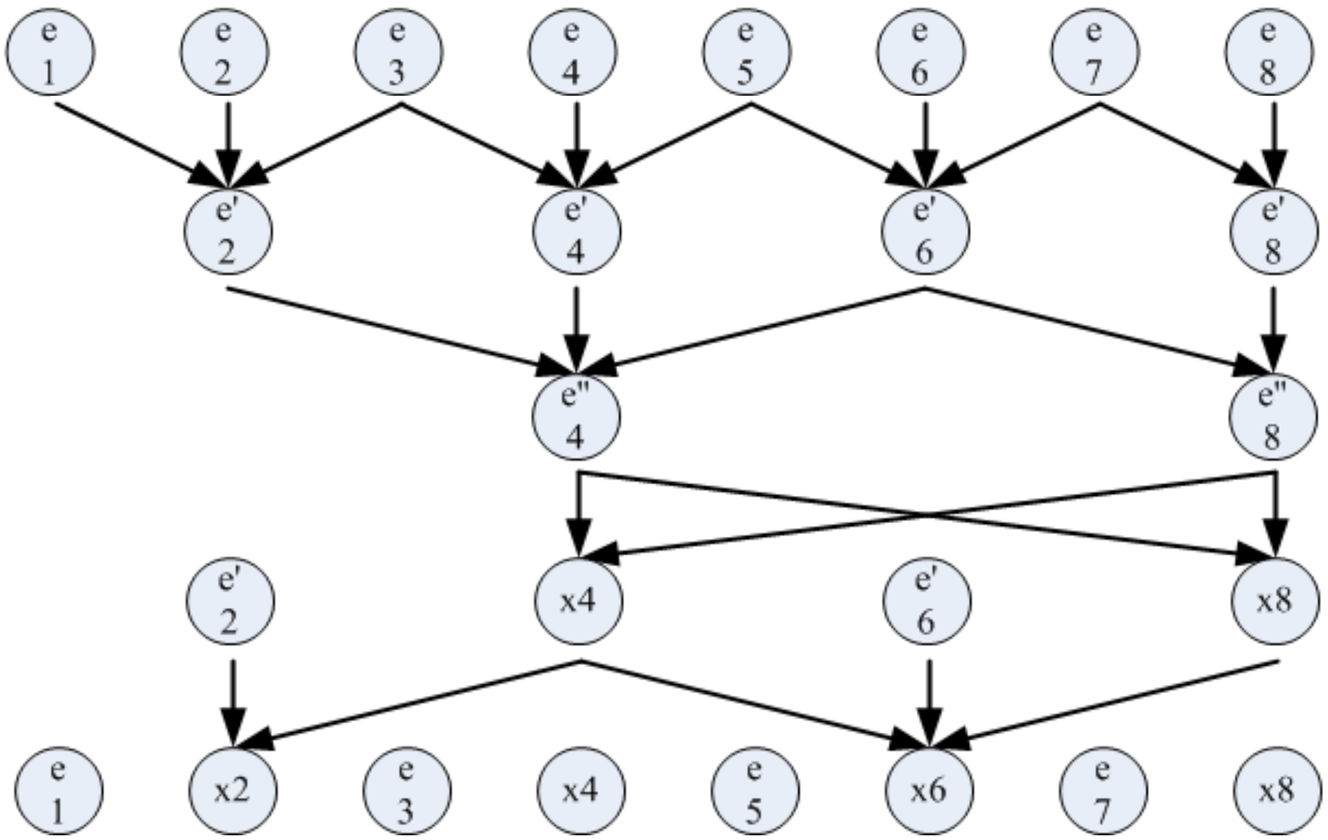
⇩

Solve 2 unknowns

Backward Substitution

Cyclic Reduction (CR)

Forward Reduction



Backward Substitution

8-unknown system

⇩

4-unknown system

⇩

2-unknown system

⇩

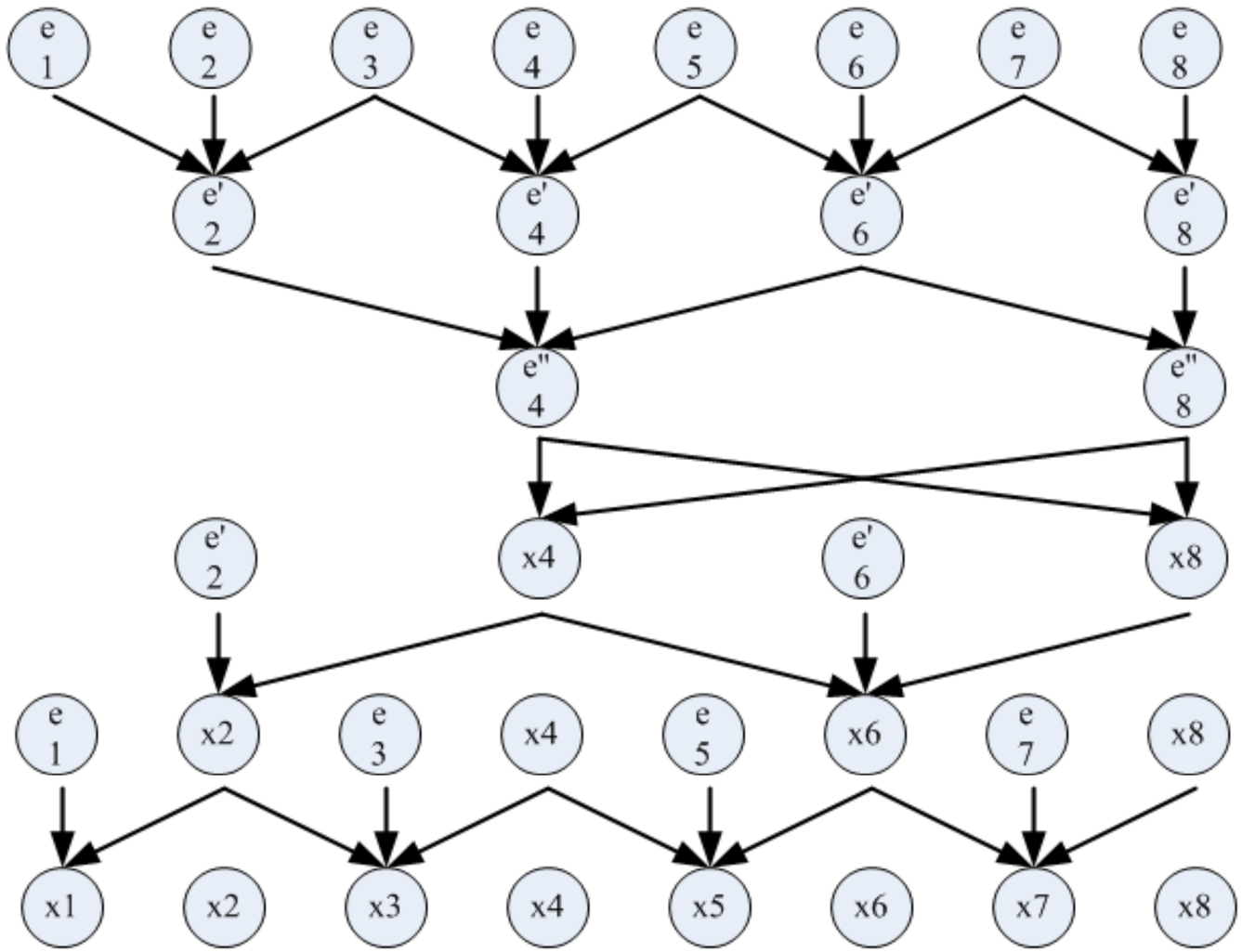
Solve 2 unknowns

⇩

Solve 4 unknowns

Cyclic Reduction (CR)

Forward Reduction



Backward Substitution

8-unknown system

⇩

4-unknown system

⇩

2-unknown system

⇩

Solve 2 unknowns

⇩

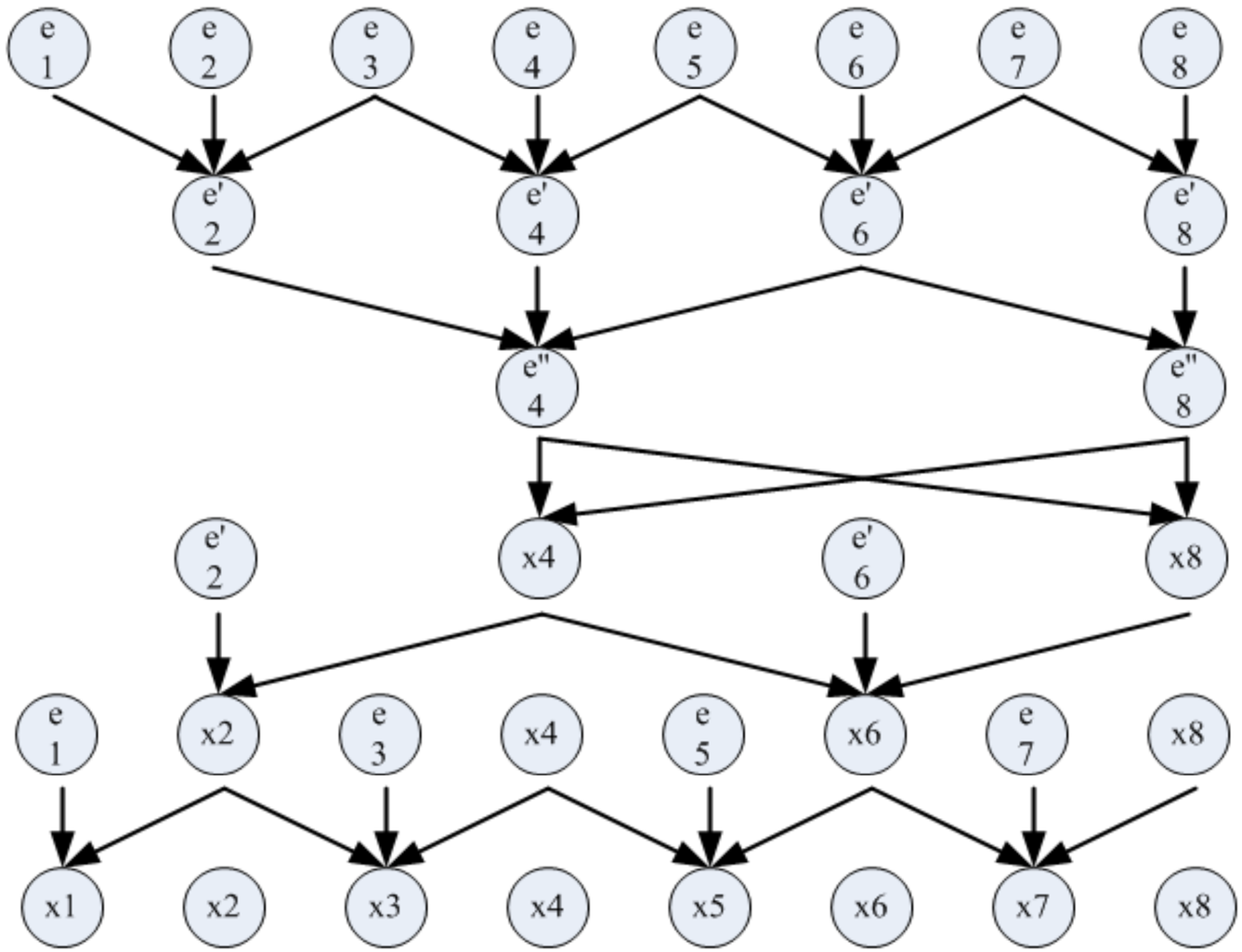
Solve 4 unknowns

⇩

Solve 8 unknowns

Cyclic Reduction (CR)

Forward Reduction



Backward Substitution

8-unknown system

⇩

4-unknown system

⇩

2-unknown system

⇩

Solve 2 unknowns

⇩

Solve 4 unknowns

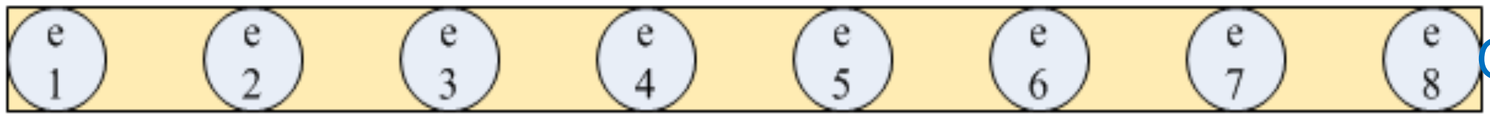
⇩

Solve 8 unknowns

$2*\log_2(8)-1 = 2*3-1 = 5$ steps

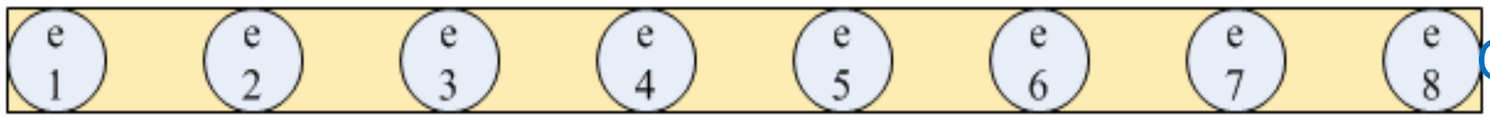
Parallel Cyclic Reduction (PCR)

Parallel Cyclic Reduction (PCR)



One 8-unknown system

Parallel Cyclic Reduction (PCR)

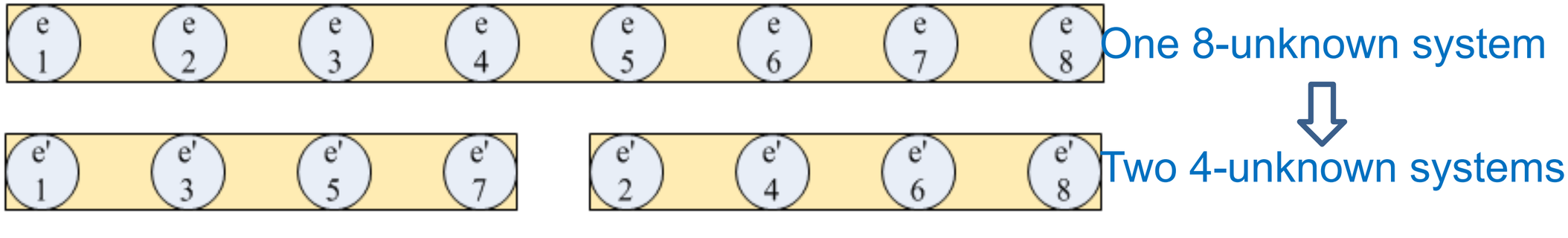


One 8-unknown system

Forward Reduction

Parallel Cyclic Reduction (PCR)

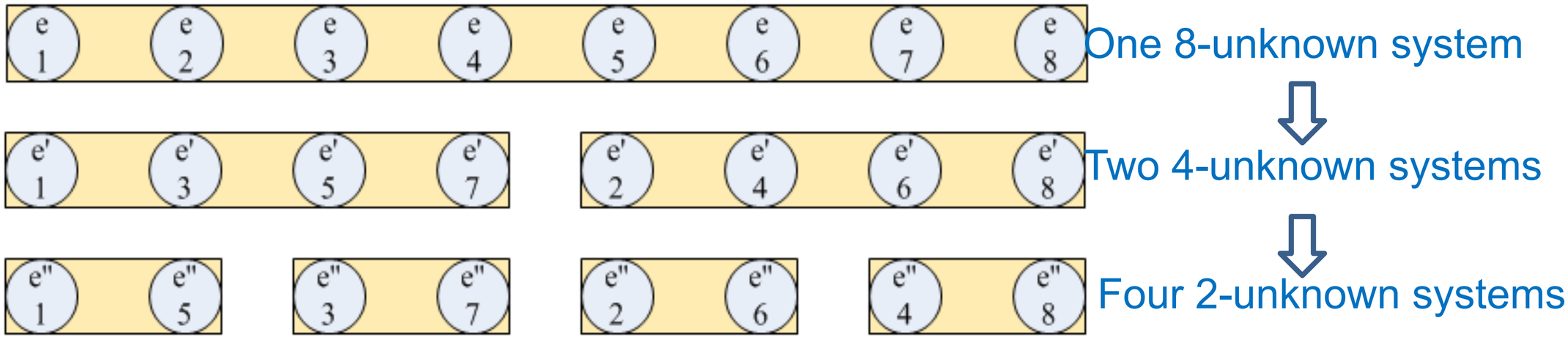
4 threads working



Forward Reduction

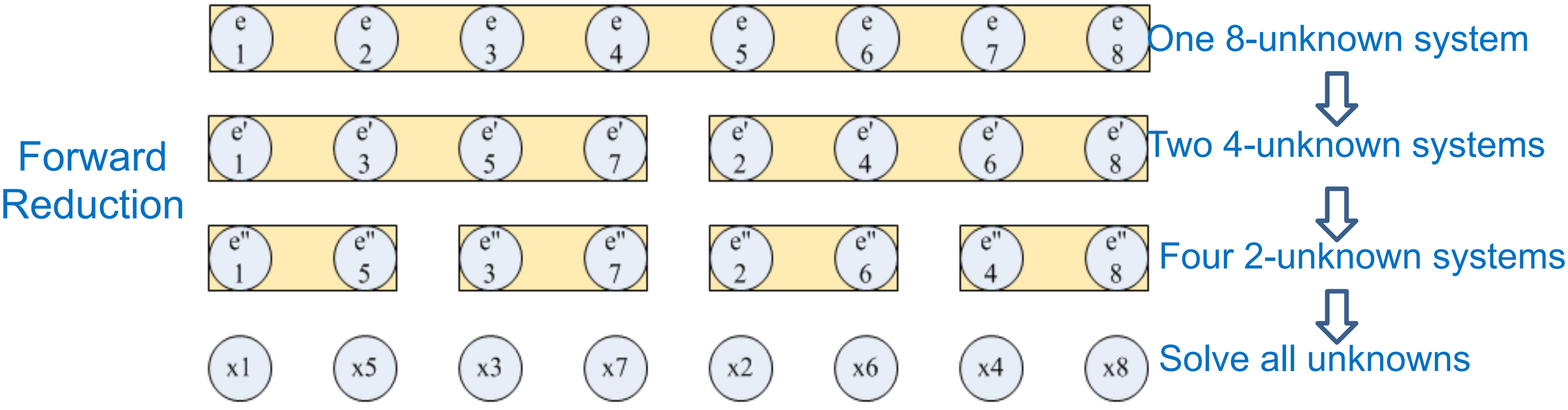
Parallel Cyclic Reduction (PCR)

4 threads working



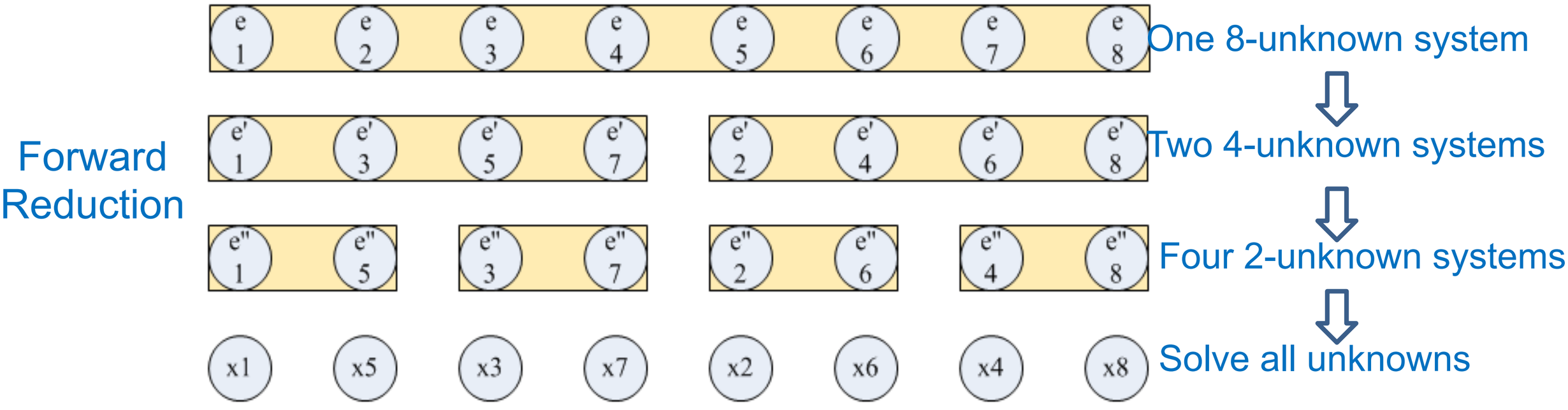
Parallel Cyclic Reduction (PCR)

4 threads working



Parallel Cyclic Reduction (PCR)

4 threads working

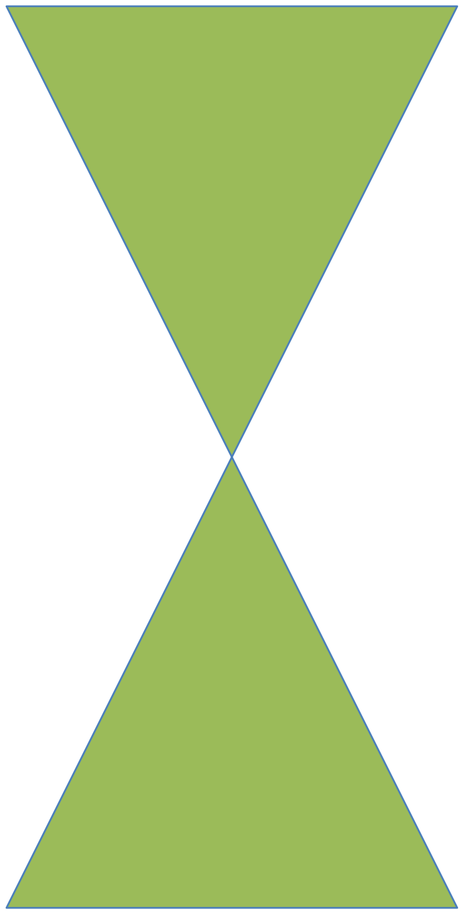


$\log_2 (8) = 3$ steps

Comparison

- CR
  - Every step we reduce the system size by half (Good)
  - Some processing cores stay **idle** if the system size is smaller than the number of cores (Bad)
  - Needs more steps to finish (Bad)
- PCR
  - Fewer steps required (Good)
  - Same amount of work for all steps (Bad)
    - More work overall (Bad)

Structure Comparison



CR
(Work-efficient)

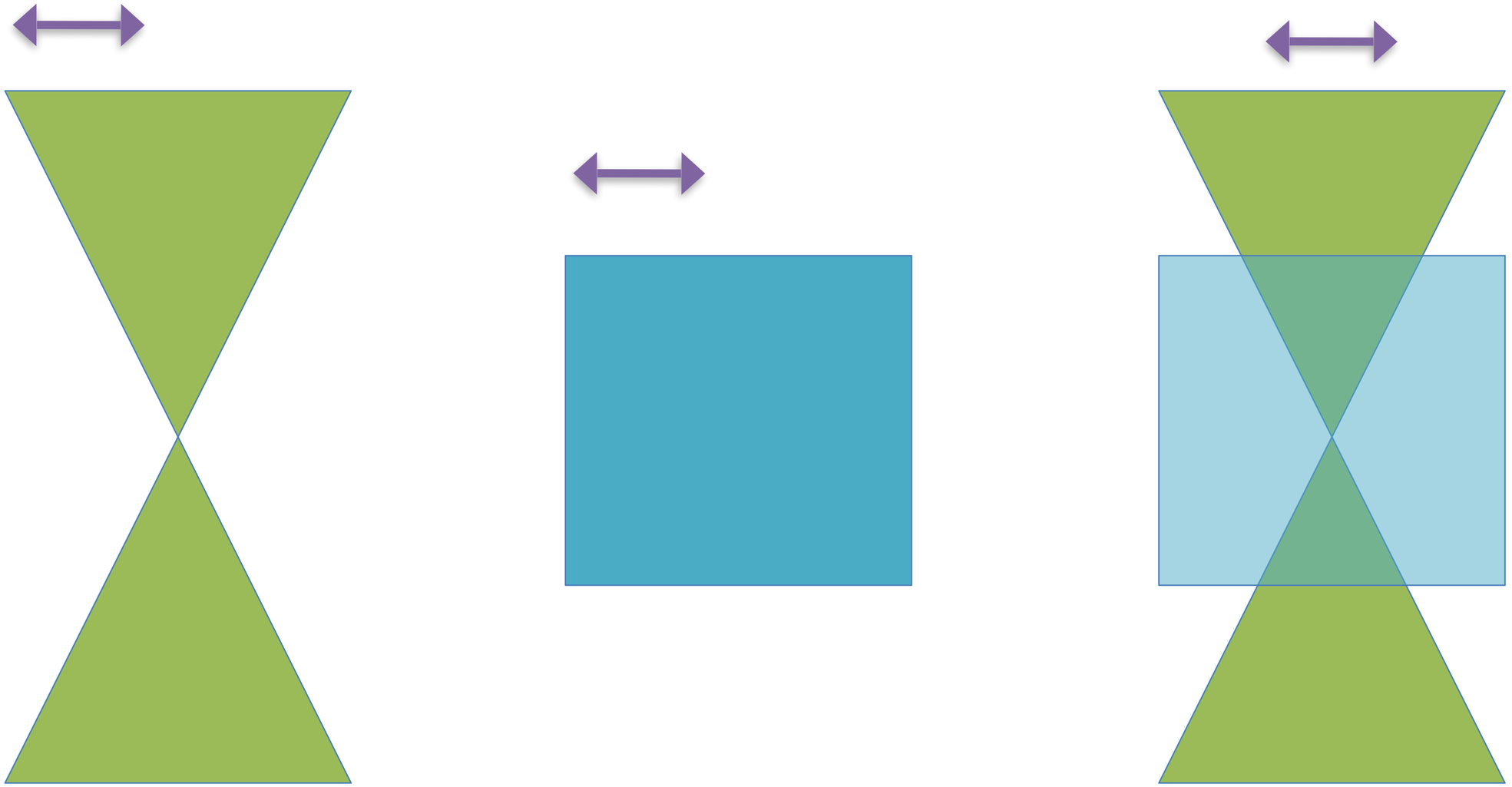
PCR
(Step-efficient)

Structure Comparison

CR
(Work-efficient)

PCR
(Step-efficient)

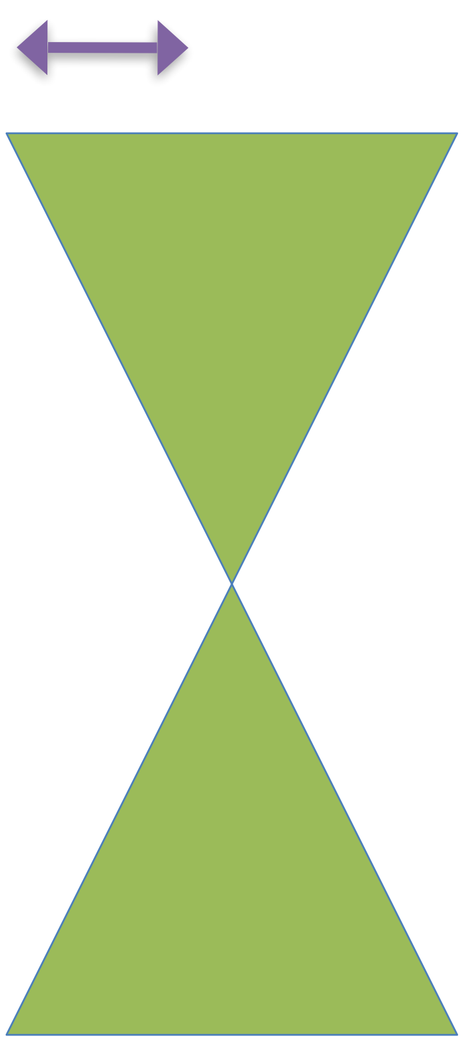
Structure Comparison



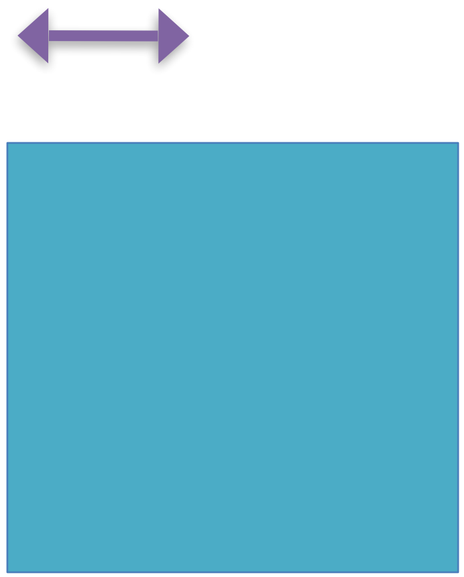
CR
(Work-efficient)

PCR
(Step-efficient)

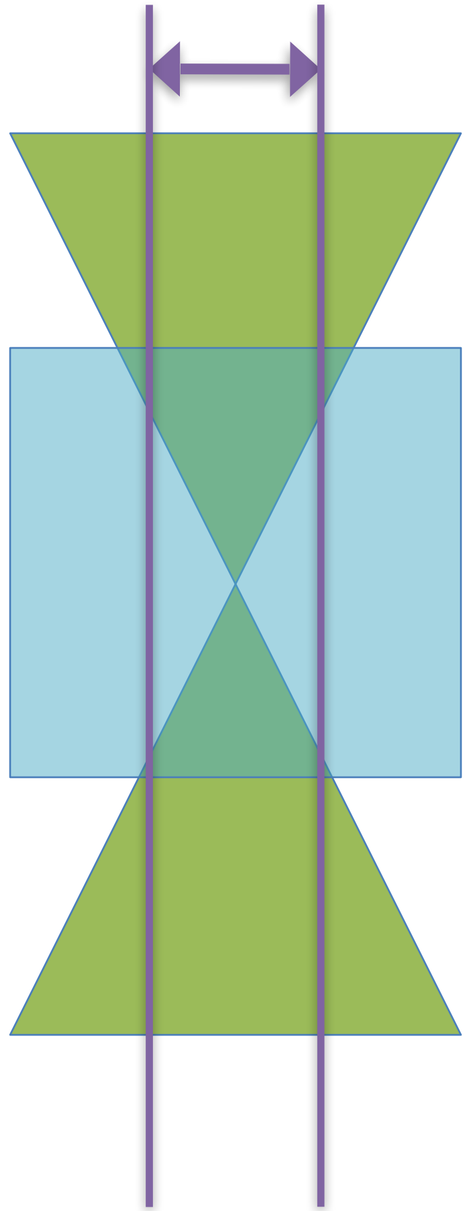
Structure Comparison



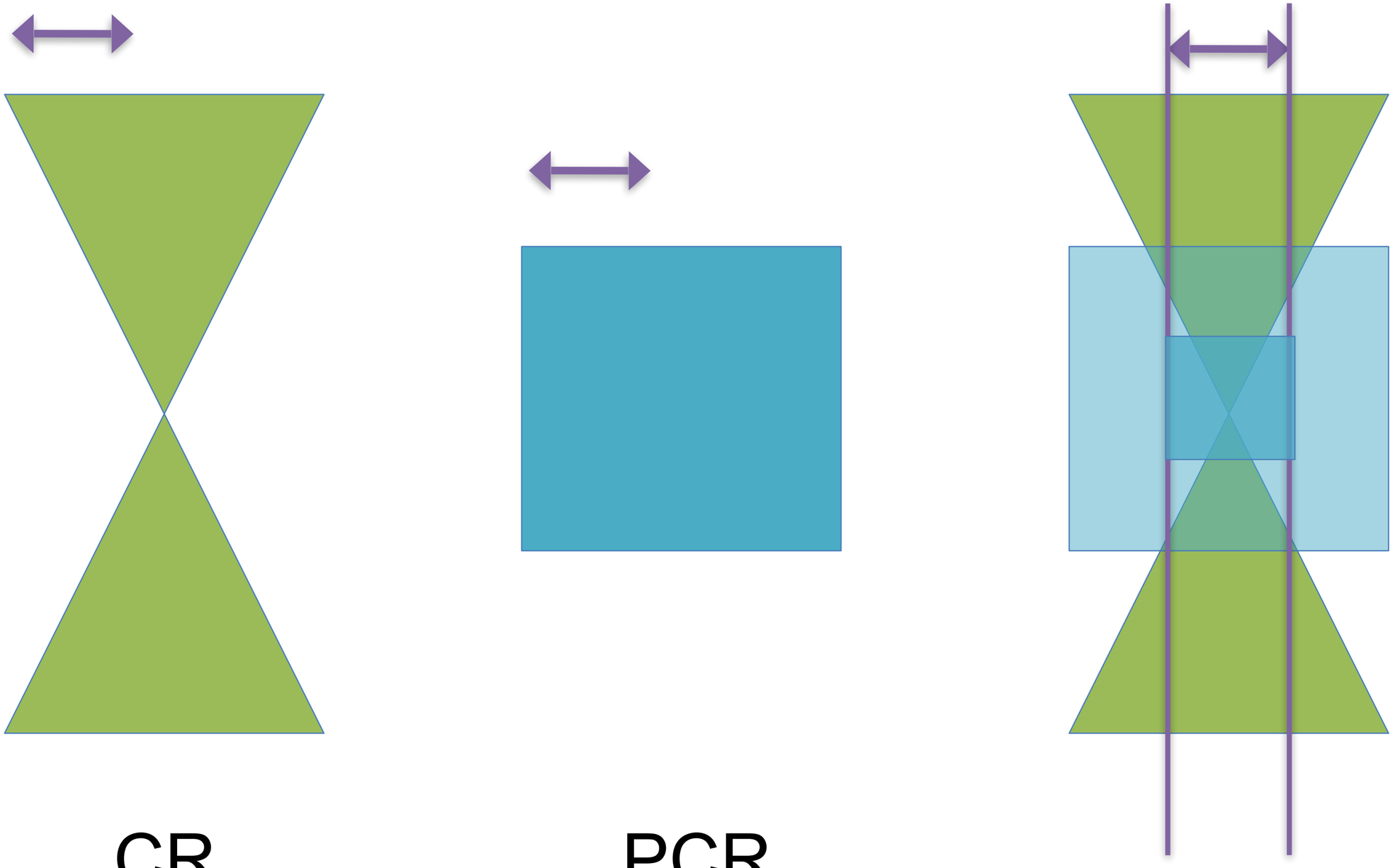
CR
(Work-efficient)



PCR
(Step-efficient)



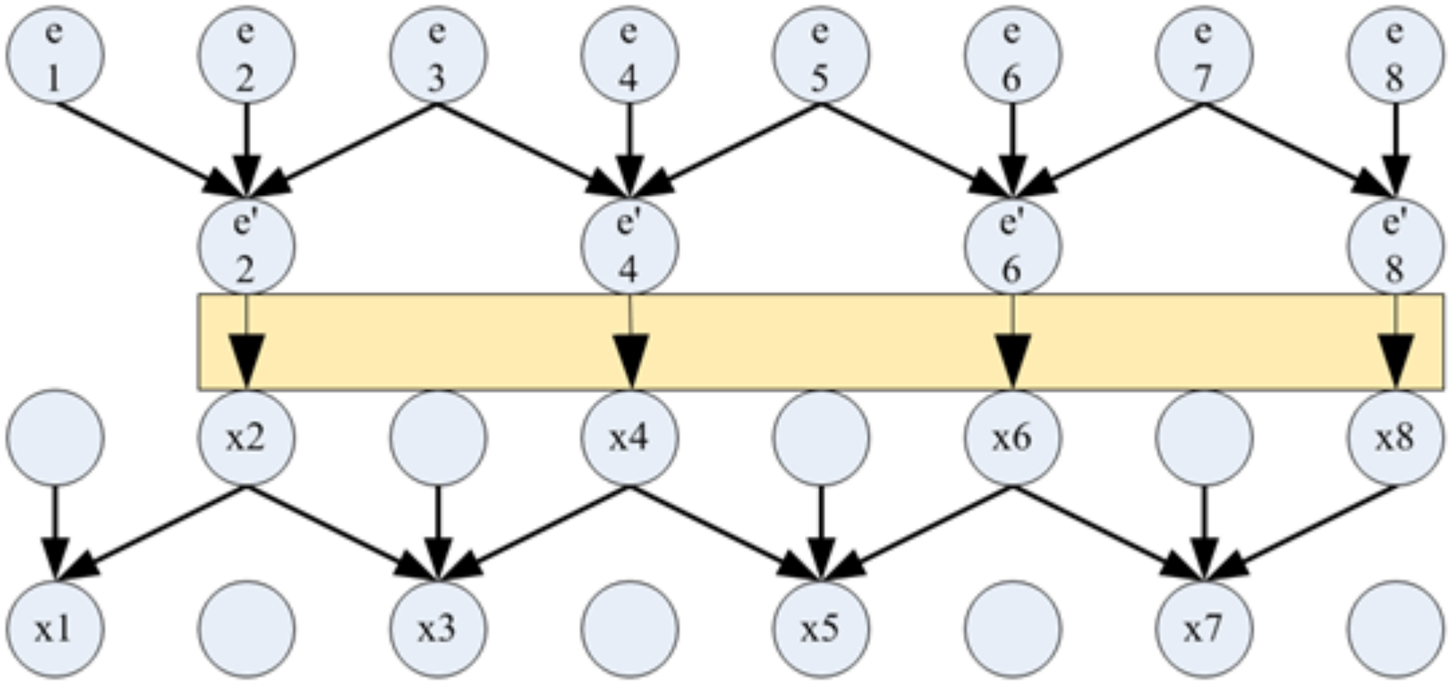
Structure Comparison



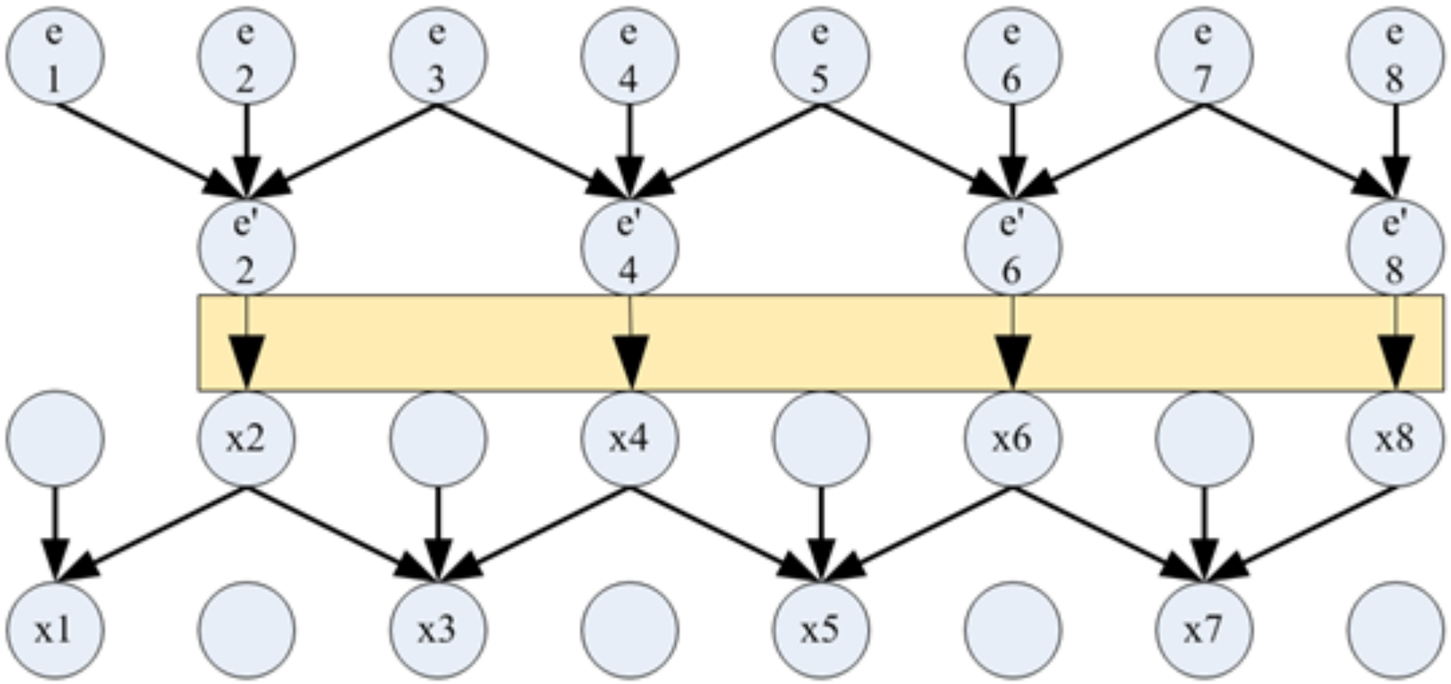
CR
(Work-efficient)

PCR
(Step-efficient)

Hybrid Algorithm

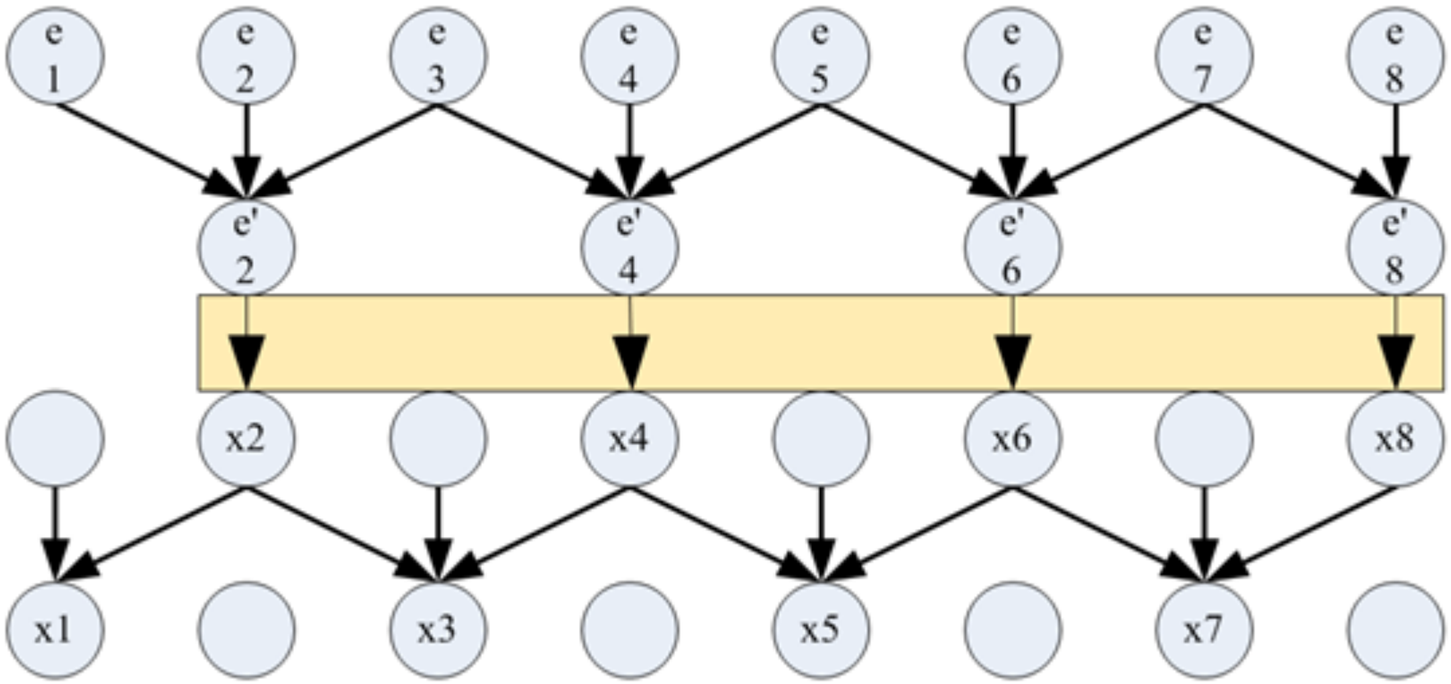


Hybrid Algorithm



Switch to PCR

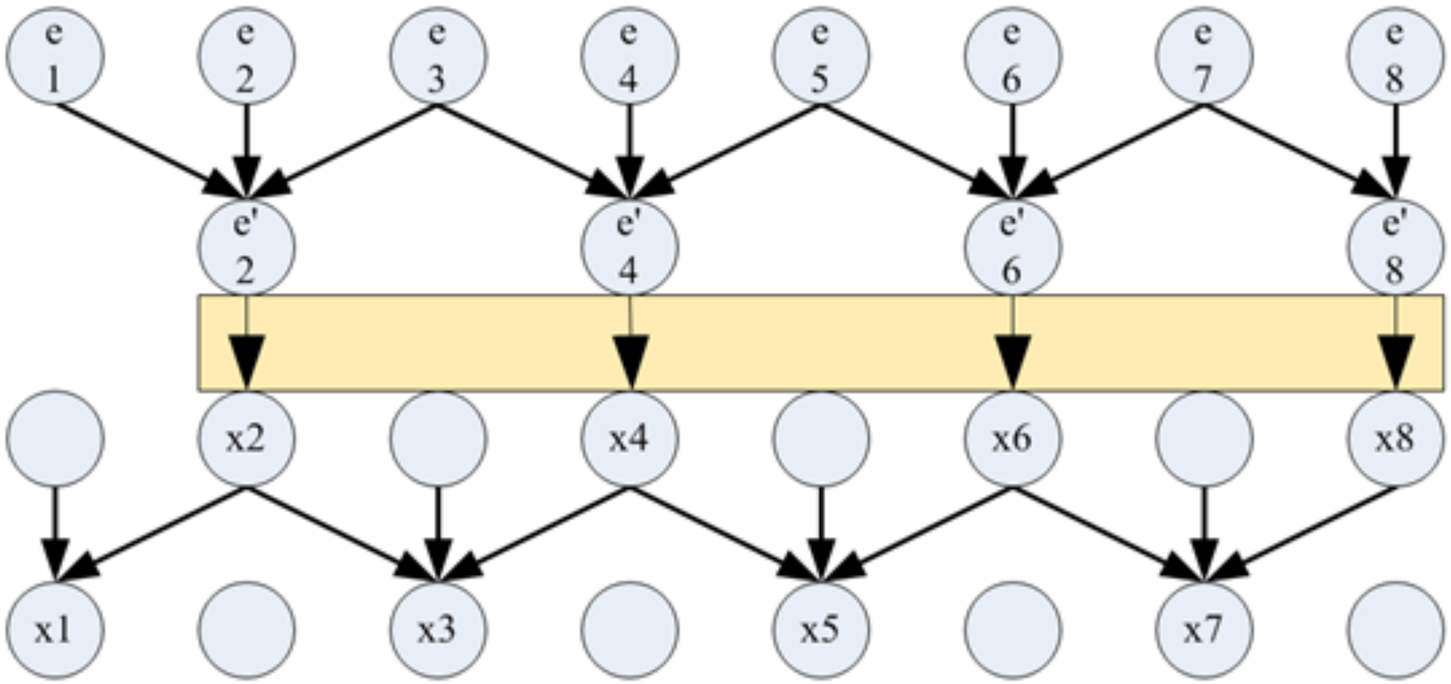
Hybrid Algorithm



Switch to PCR

Switch back to CR

Hybrid Algorithm

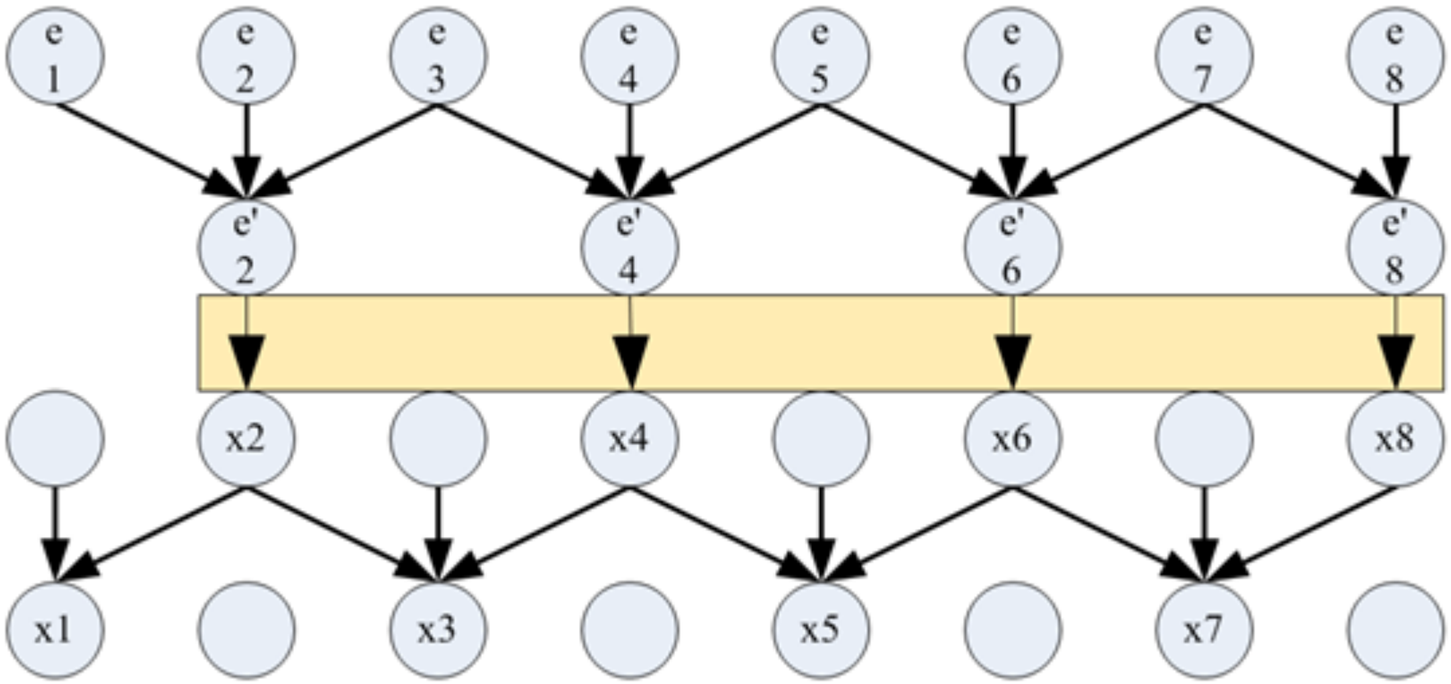


Switch to PCR

Switch back to CR

No idle processors
Fewer algorithmic steps

Hybrid Algorithm



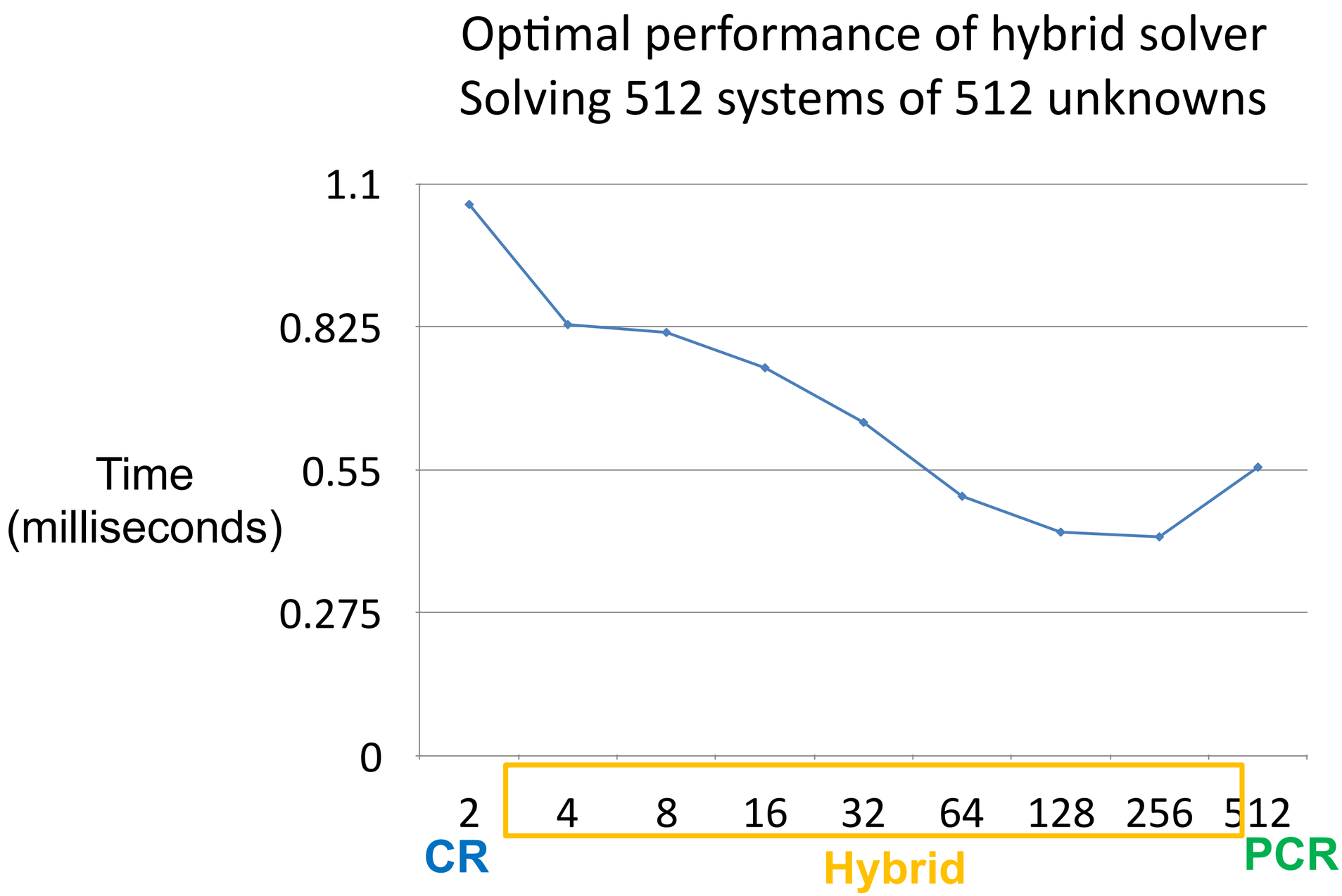
Switch to PCR

Switch back to CR

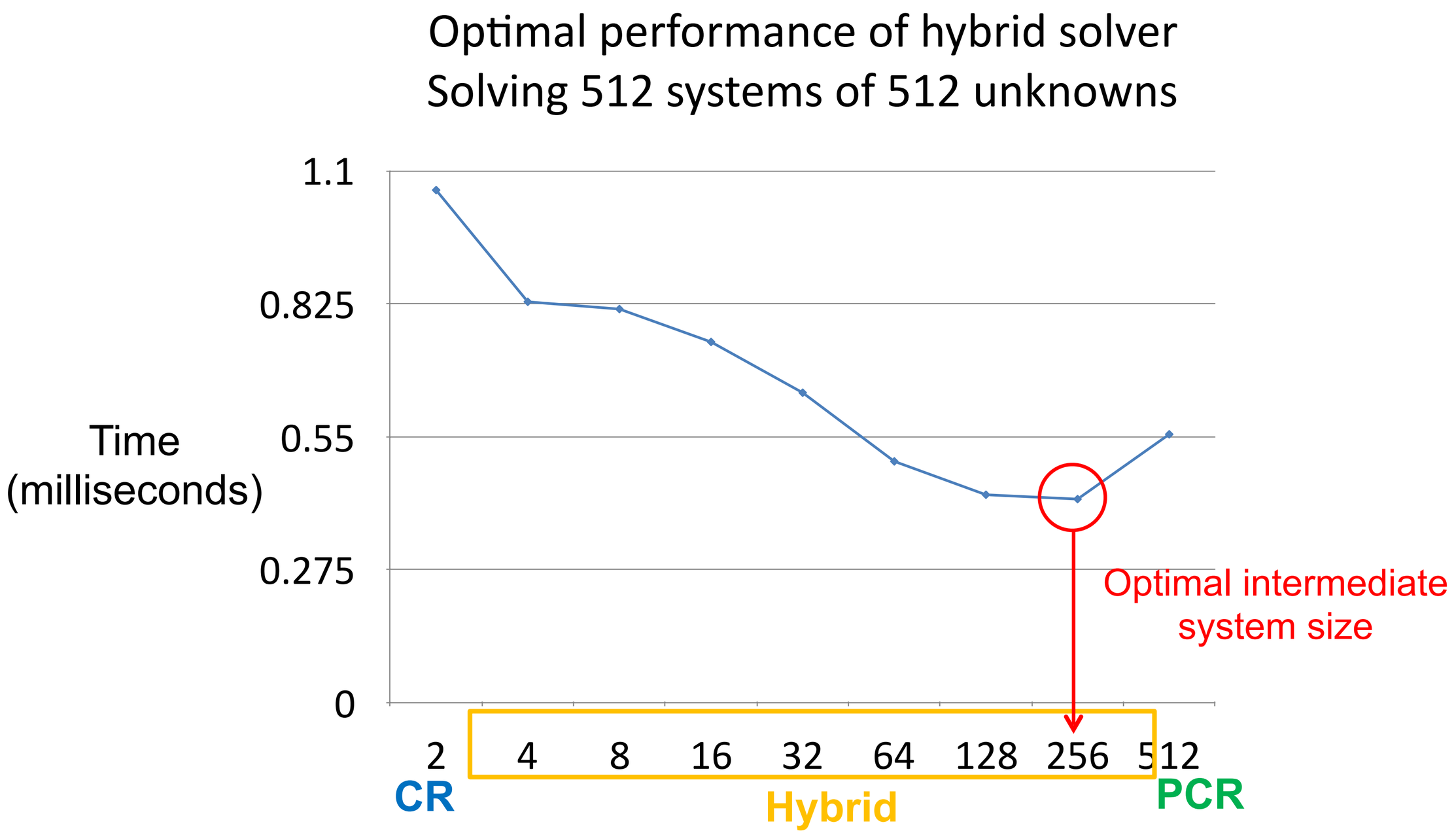
No idle processors
Fewer algorithmic steps

Even more beneficial because of:
bank conflicts
control overhead

Hybrid Solver – Sweet Spot



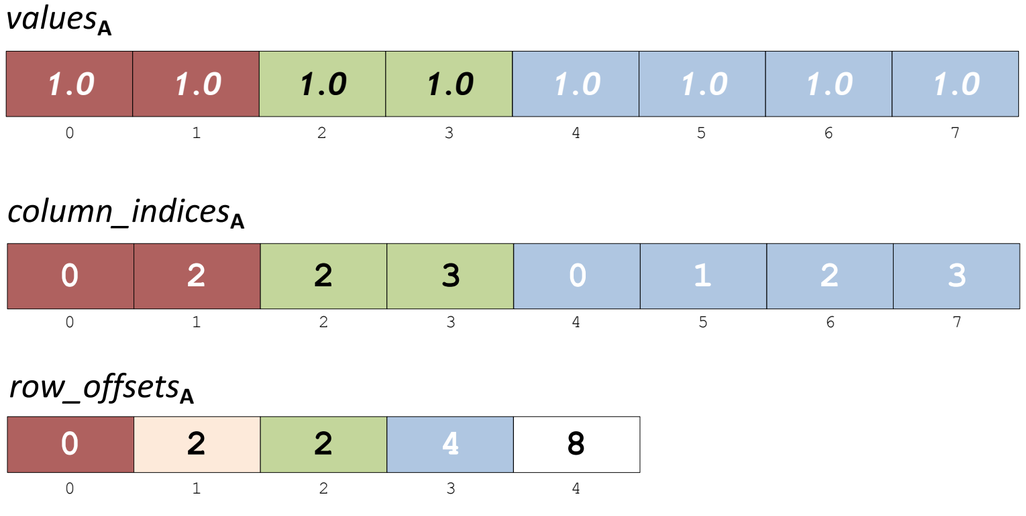
Hybrid Solver – Sweet Spot



Sparse Matrix-Vector

Recap: Sequential CSR Sparse-Matrix, Dense-Vector Multiplication

$$\begin{bmatrix} 1.0 & -- & 1.0 & -- \\ -- & -- & -- & -- \\ -- & -- & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 & 1.0 \end{bmatrix}$$

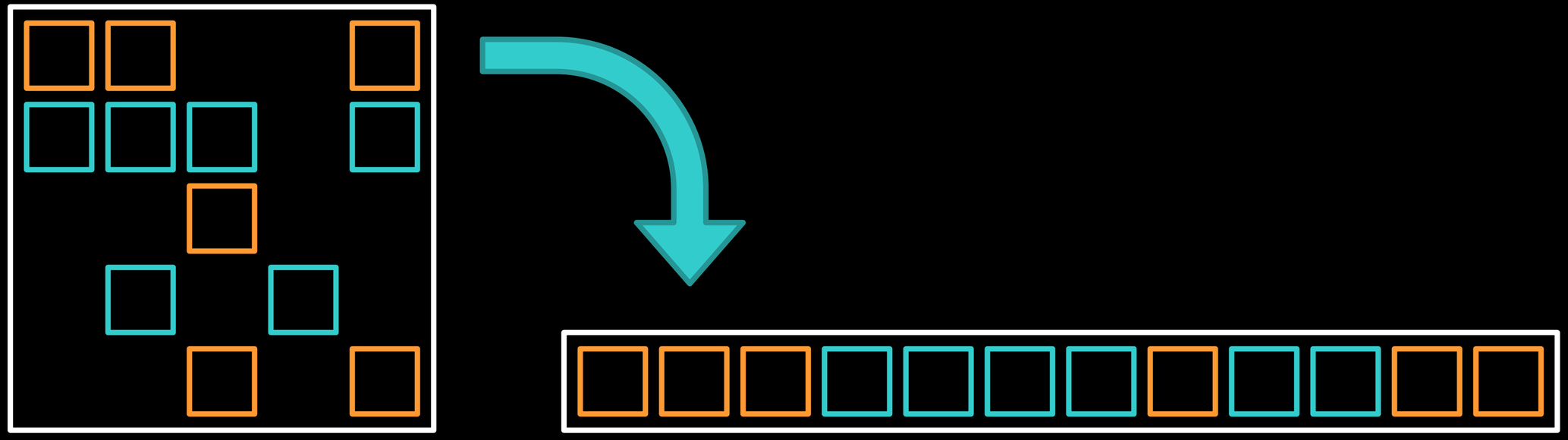


- **Input: CSR matrix A, dense vector x**
  Output: Dense vector y such that y <- Ax

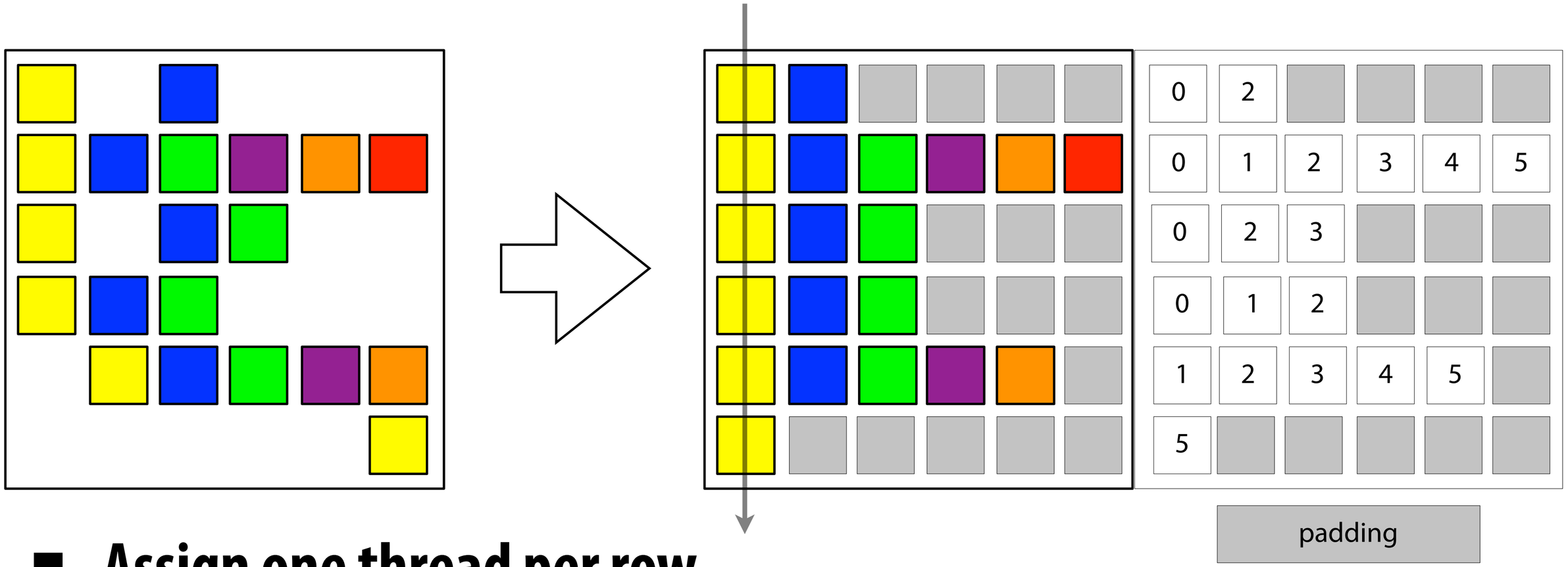
```
for (int row = 0; row < A.m; ++row) {
  y[row] = 0.0;
  for (int nz = A.row_offsets[row]; nz < A.row_offsets[row + 1]; ++nz) {
    y[row] += A.values[nz] * x[A.column_indices[nz]];
  }
}
```

Compressed Sparse Row (CSR)

- Rows laid out in sequence
- Parallelizing over rows is inconvenient for fine-grained parallelism

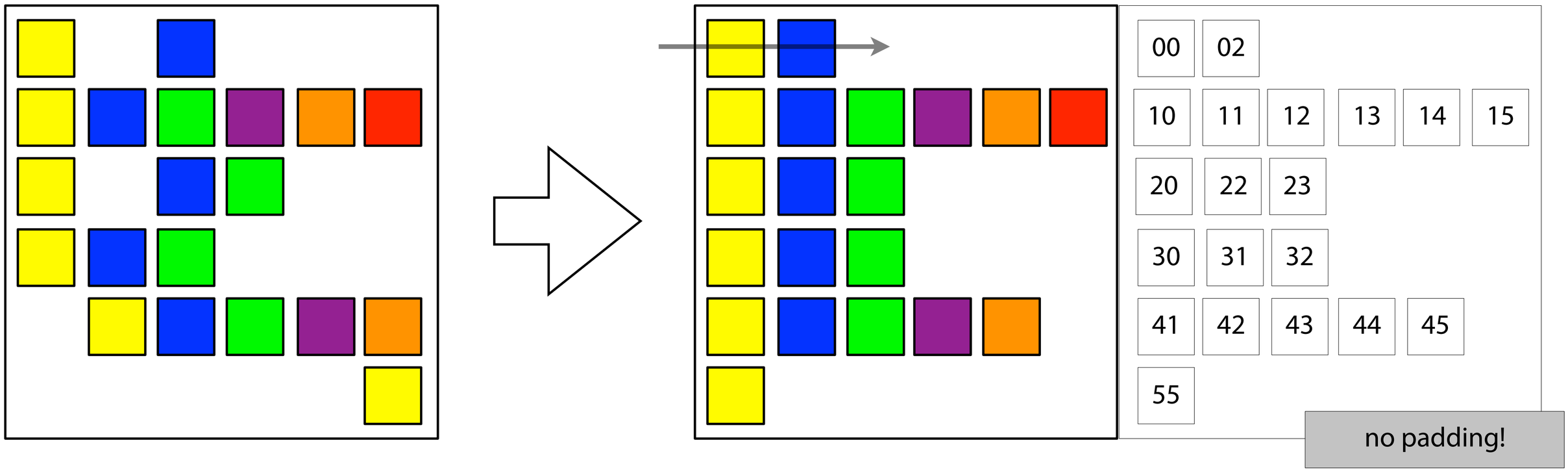


Irregular Matrices: ELL



- Assign one thread per row
- \+ Good computation rates
  \+ Multiple items per thread: registers for communication
  – Load imbalance hurts parallel efficiency

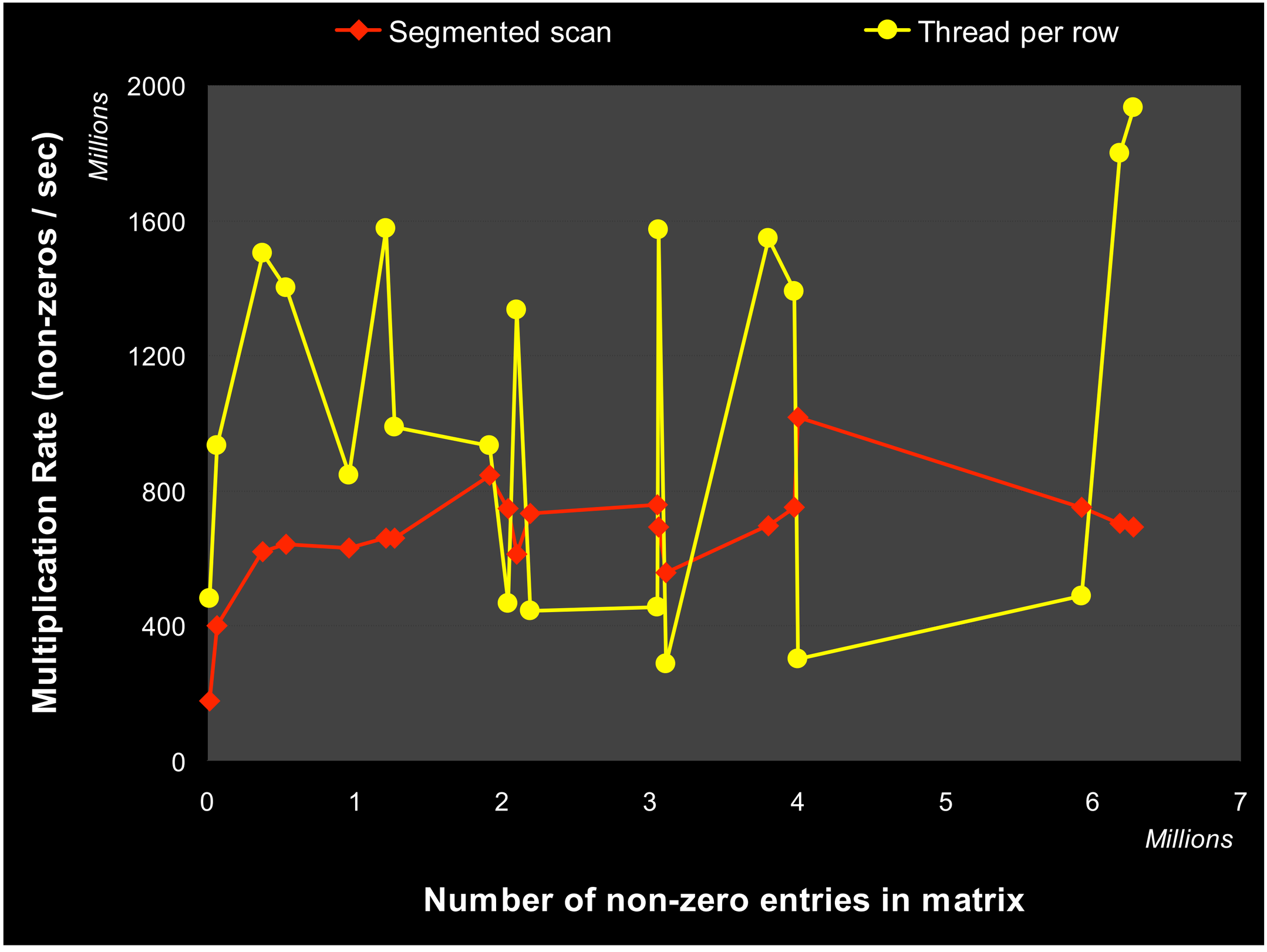
Irregular Matrices: COO



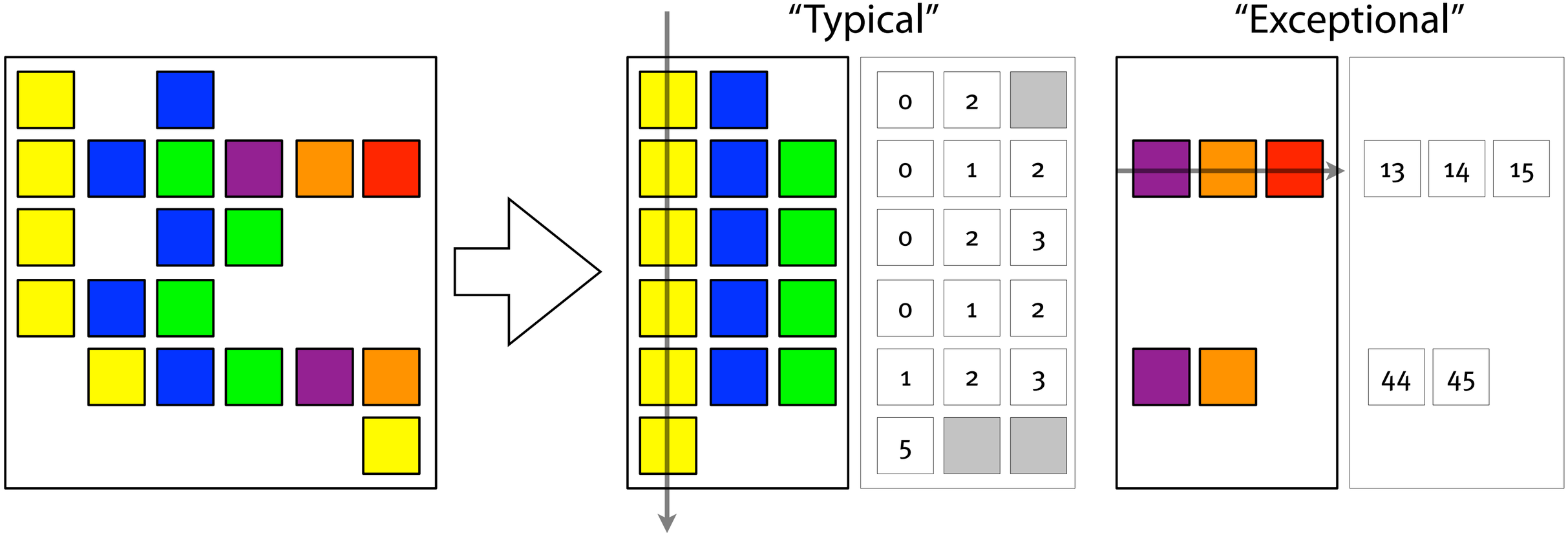
- General format; + insensitive to sparsity pattern (good load balance), but – ~3x slower than ELL
- Assign one thread per element, combine results from all elements in a row to get output element
  - Req segmented reduction, communication btwn threads

Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. **Scan Primitives for GPU Computing**. In *Graphics Hardware 2007*, pages 97–106, August 2007.

Thread-per-{element,row}



Irregular Matrices: HYB



- **Combine regularity of ELL + flexibility of COO**

SpMV: Summary

- Ample parallelism for large matrices
  - Structured matrices (dense, diagonal): straightforward
  - Sparse matrices: Tradeoff between parallel efficiency and load balance
- Take-home message: Use data structure appropriate to your matrix
  - Insight: Irregularity is manageable if you regularize the common case

SpMV: Big Idea

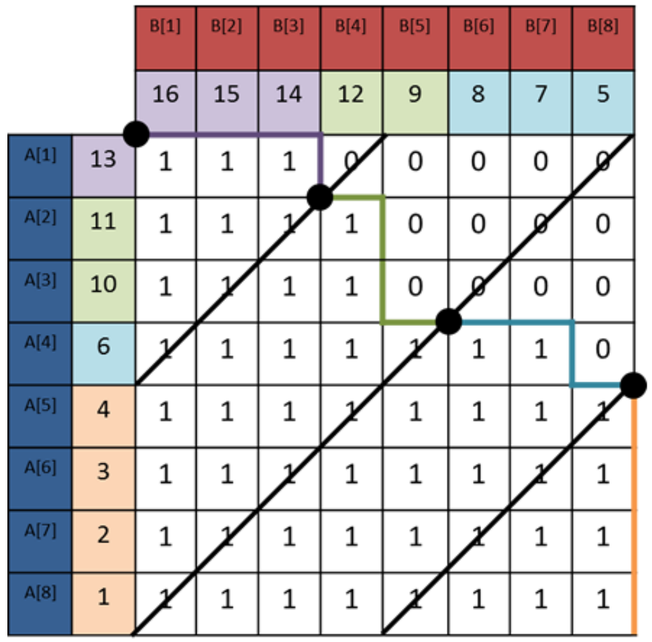
- Problem: "data-dependent underutilization from (1) an insufficient number of rows, and/or (2) irregular row lengths."
- "Ideal" efficiency is a result of two conflicting goals:
  - Localize computation to take advantage of the fastest memory/communication (key to the thread-per-row approach)
  - Load balance to make sure all threads have an equal amount of work (key to the thread-per-item approach)
- Because CSR lists all elements sequentially, we could divide up entries equally between threads, but that fails to account for different behavior at ends of rows; how do we know where rows end?

Merge Path

GPU merge path: a GPU merging algorithm, Oded Green, Robert McColl, and David A. Bader, ICS '12

Merge-Path and Its Impact on Irregular Algorithms

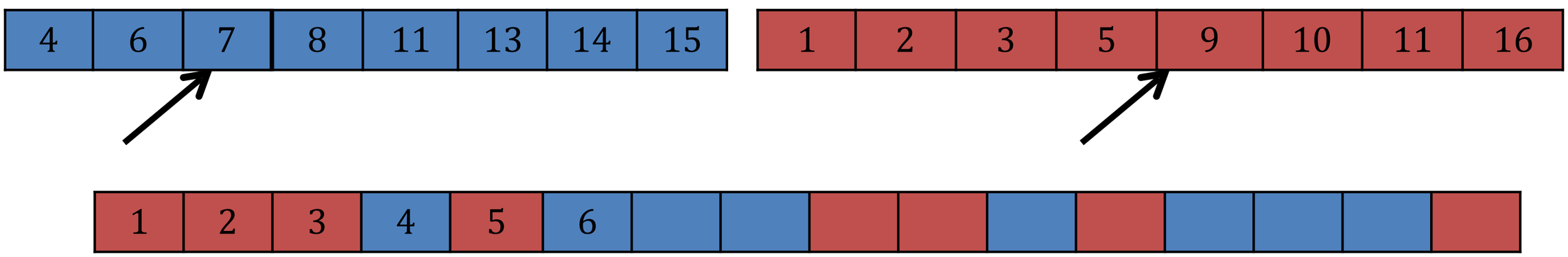
Oded Green



Lets start off by defining the “Merge” operation

- **Input:** Two sorted arrays A,B
- **Output:** Sorted array C
- |C| = |A| + |B|
- **Time:** O(n) = O(|C|)
- Simple

$$
\begin{aligned}
&if\ (A[a_i] < B[b_i])\ then\\
&\quad C[c_i++] \leftarrow A[a_i++]\\
&else\\
&\quad C[c_i++] \leftarrow B[b_i++]
\end{aligned}
$$

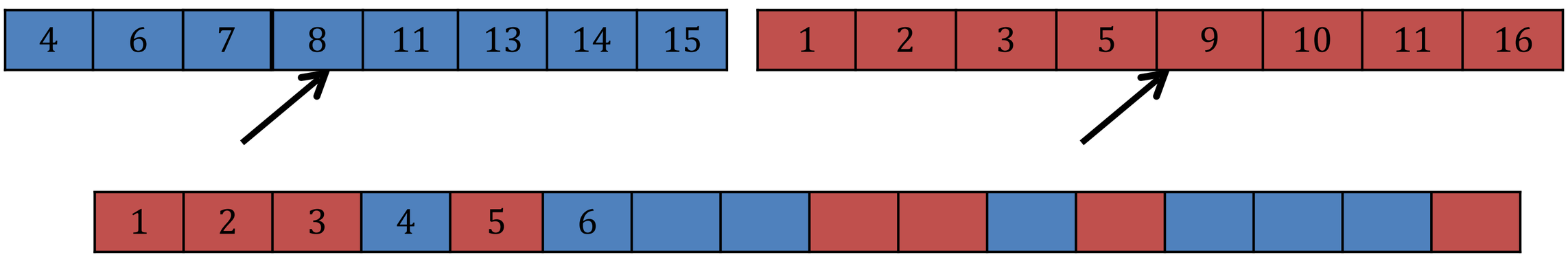


Q: How can we split the output into two halves so we can process each half independently?

Lets start off by defining the “Merge” operation

- **Input:** Two sorted arrays A,B
- **Output:** Sorted array C
- |C| = |A| + |B|
- **Time:** O(n) = O(|C|)
- Simple

$$
\begin{aligned}
&if\ (A[a_i] < B[b_i])\ then\\
&\quad C[c_i++] \leftarrow A[a_i++]\\
&else\\
&\quad C[c_i++] \leftarrow B[b_i++]
\end{aligned}
$$

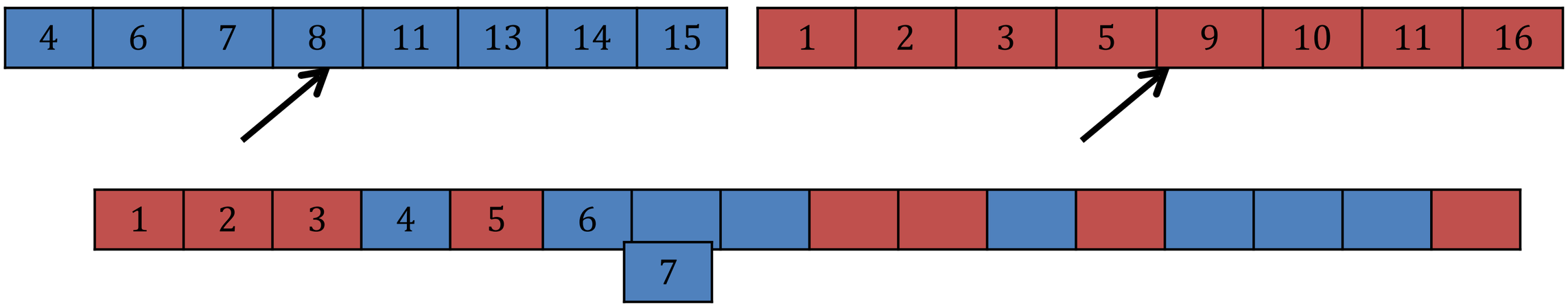


Q: How can we split the output into two halves so we can process each half independently?

Lets start off by defining the “Merge” operation

- **Input:** Two sorted arrays A,B
- **Output:** Sorted array C
- |C| = |A| + |B|
- **Time:** O(n) = O(|C|)
- Simple

$$
\begin{aligned}
&if\ (A[a_i] < B[b_i])\ then\\
&\quad C[c_i++] \leftarrow A[a_i++]\\
&else\\
&\quad C[c_i++] \leftarrow B[b_i++]
\end{aligned}
$$



Q: How can we split the output into two halves so we can process each half independently?

Multi-way Partitioning Merge

- Pick every 256th element of A & B as splitter

- Apply merge recursively to merge splitter sets
  - recursively apply merge procedure

not perfectly balanced

- Split A & B with merged splitters

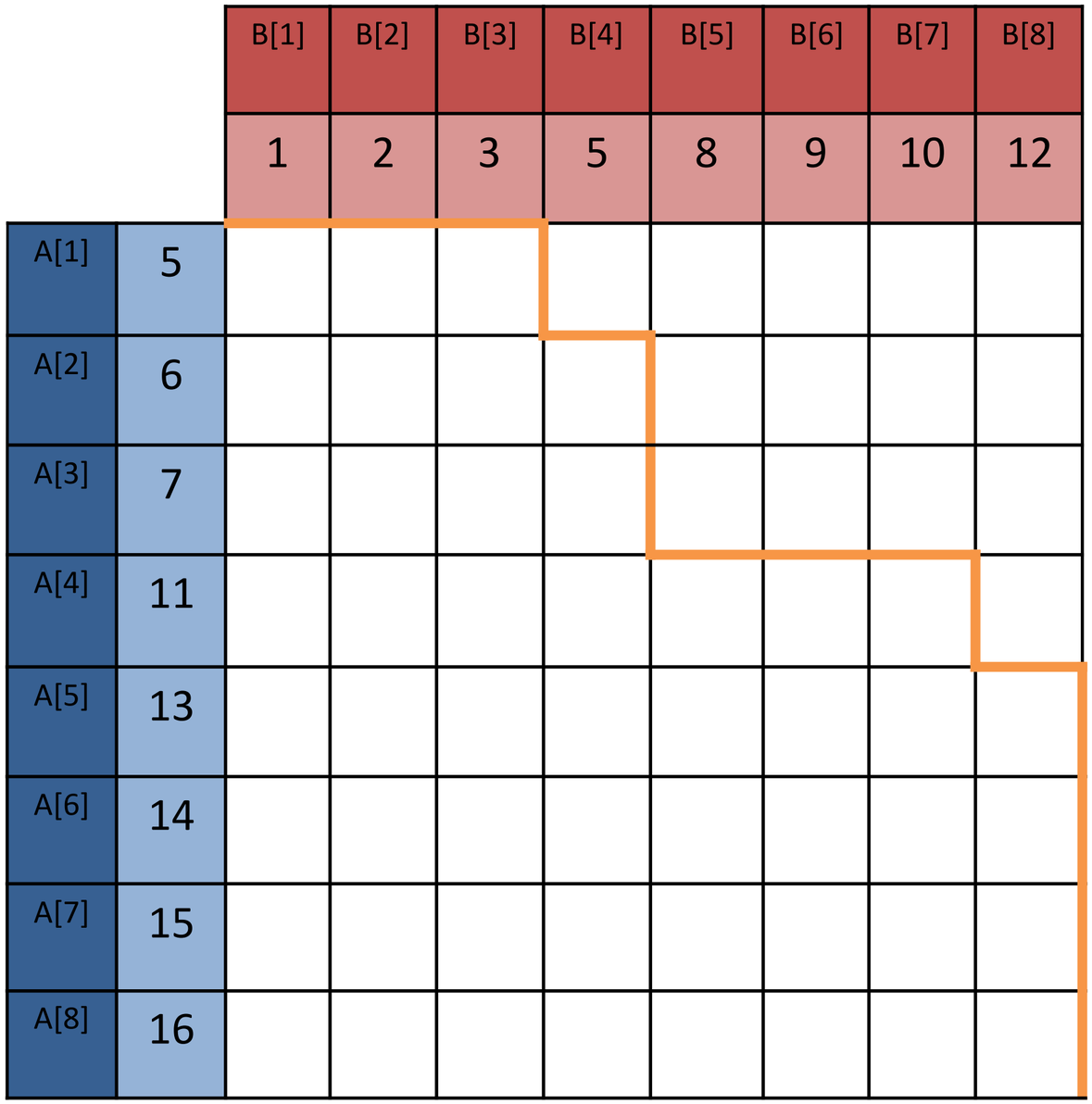
- Merge resulting pairs of tiles (at most 256 elements)

Parallel Merge Challenges

- Partitioning / Load balancing
  - Must be computationally cheap (less than O(n))
  - Must be parallel (high utilization)
- Low synchronization/communication
- Scalable
- Simple (preferably)

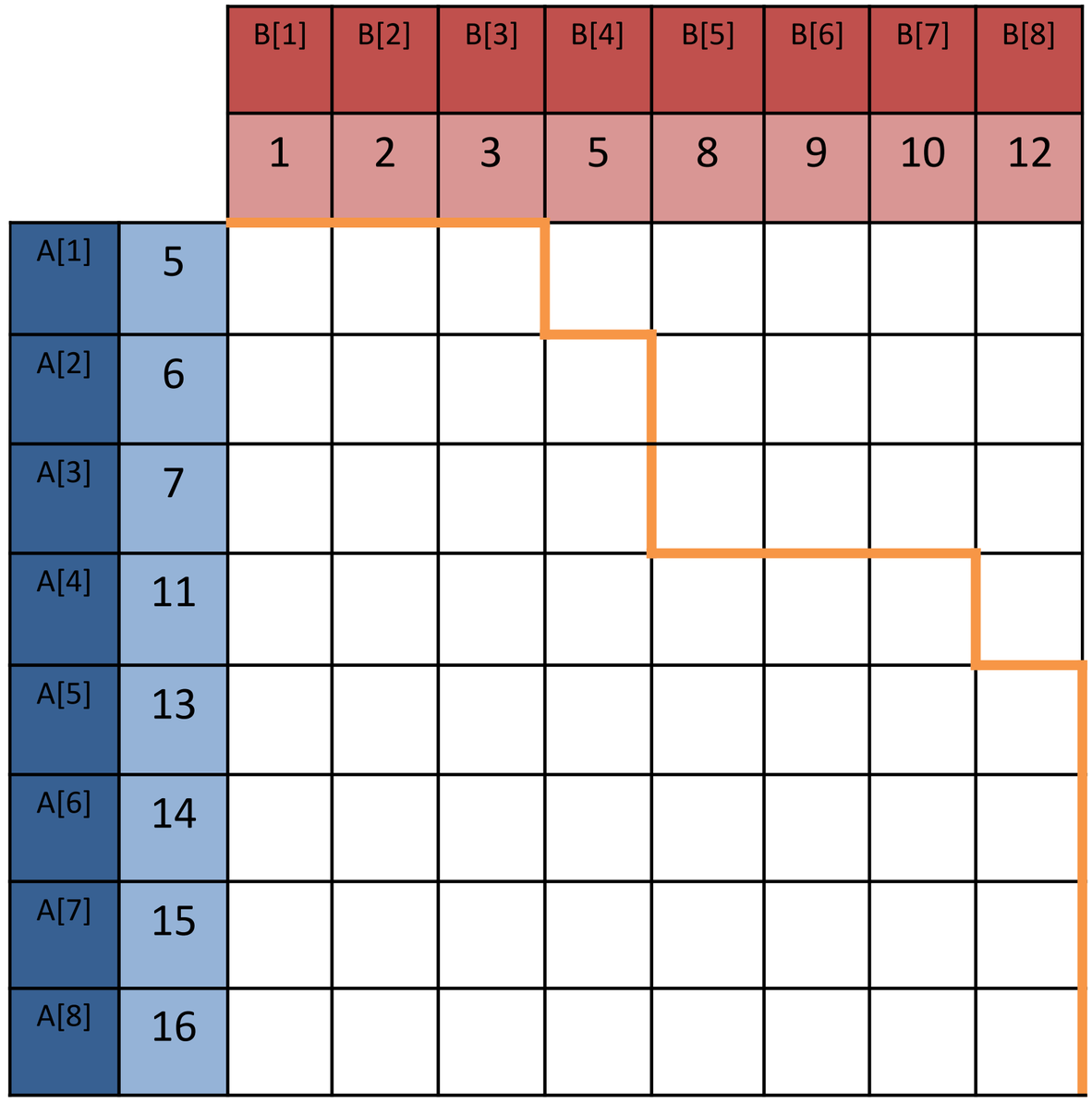
Merge Path

- if (A[i]>B[j])
  - Select A[i]
  - Next i
  - **Move down**
- else
  - Select B[j]
  - Next j
  - **Move right**



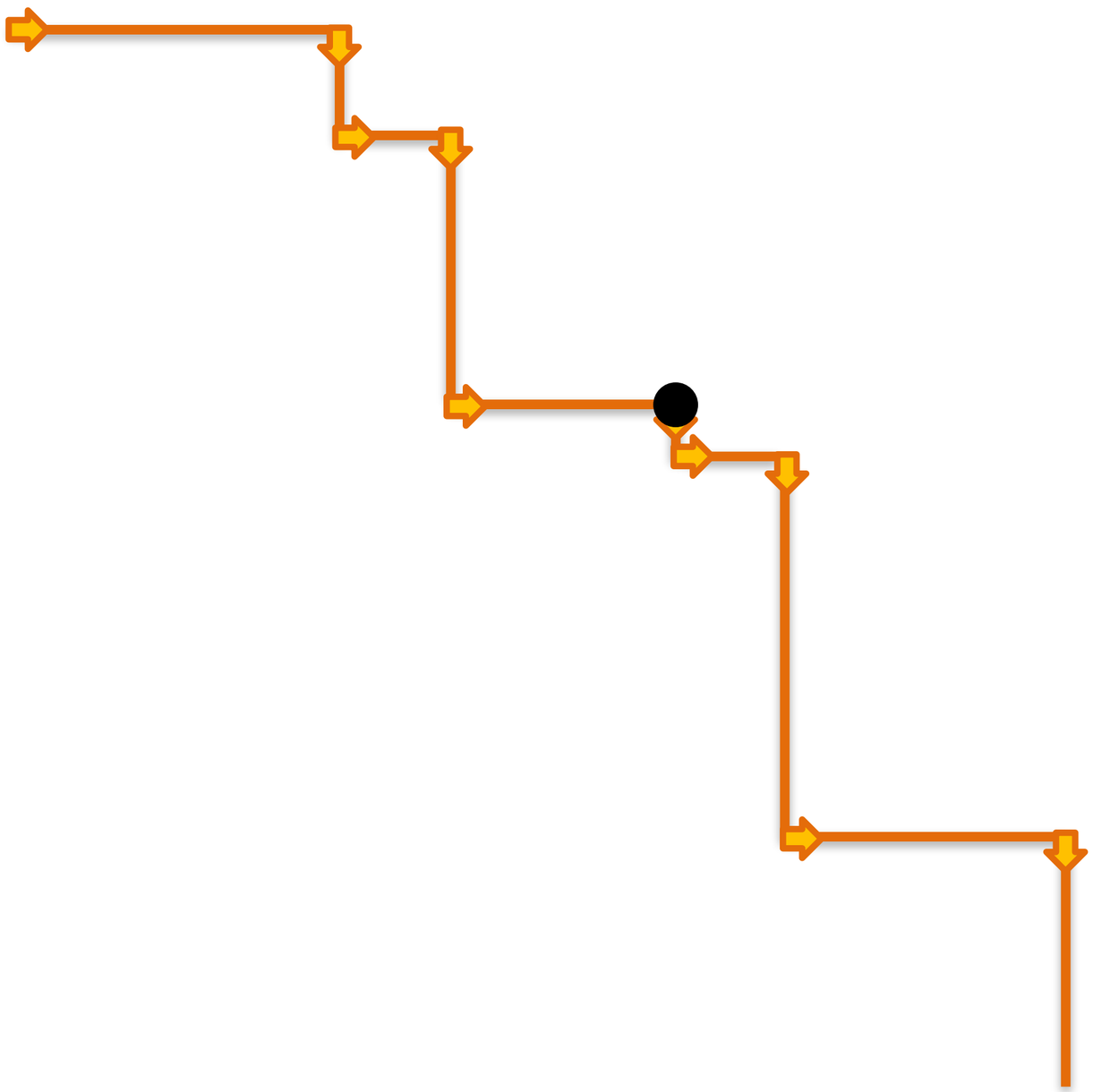
Merge Path

- if (A[i]>B[j])
  - Select A[i]
  - Next i
  - **Move down**
- else
  - Select B[j]
  - Next j
  - **Move right**



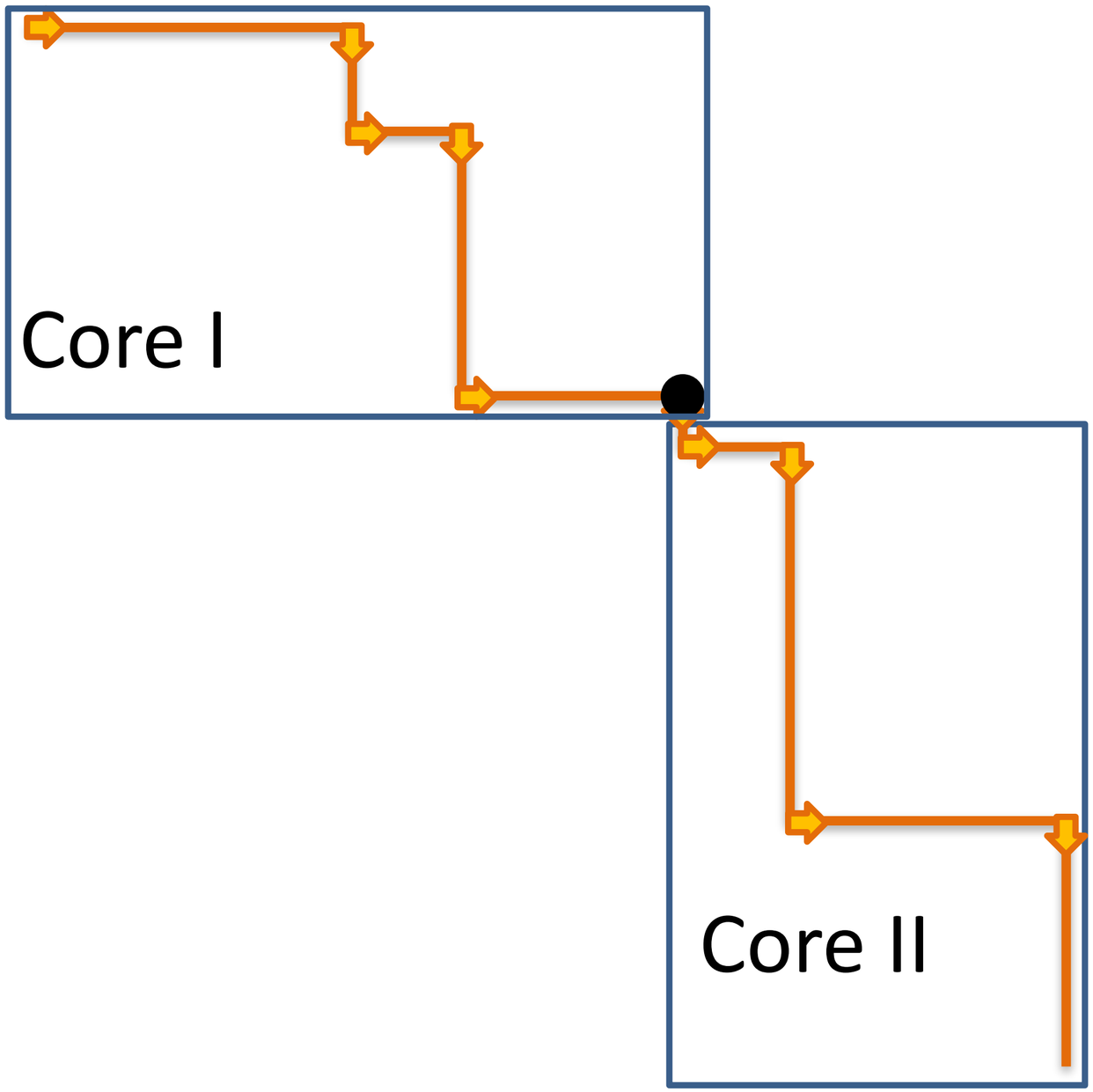
[*Odeh, Green, Birk*; 2012; *MTAAP*]
[*Green*; 2012; *ICS*]

Intuition – 2 Cores



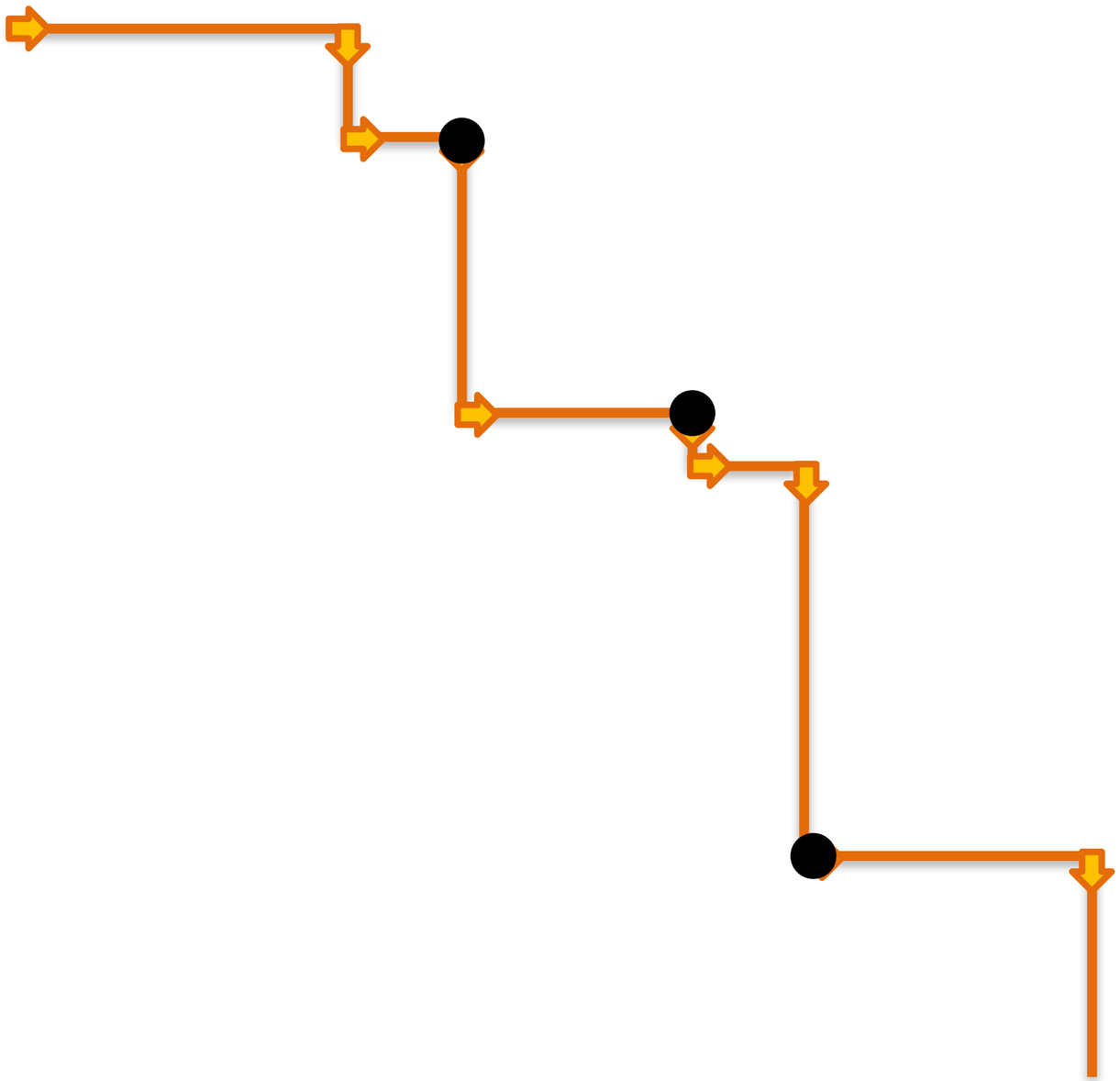
- Each core will merge a sub-path
- For perfect load balancing, divide path into equal length sub-paths

Intuition – 2 Cores



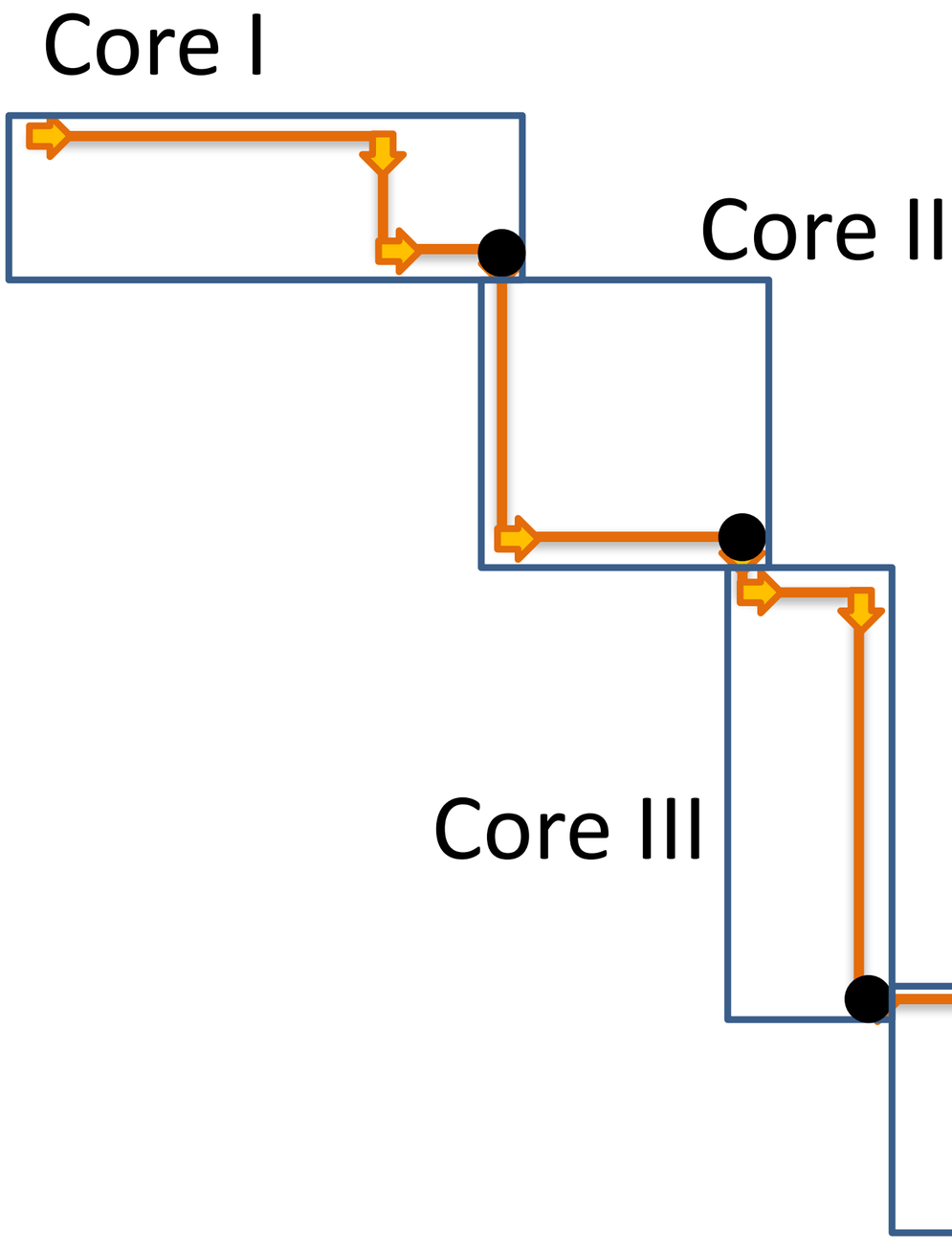
- Each core will merge a sub-path
- For perfect load balancing, divide path into equal length sub-paths

Intuition – 4 Cores



- Each core will merge a sub-path
- For perfect load balancing, divide path into equal length sub-paths

Intuition – 4 Cores



- Each core will merge a sub-path
- For perfect load balancing, divide path into equal length sub-paths

Can the entire path be computed in parallel?

Oded Green, NVIDIA Research, 2015

Can the entire path be computed in parallel?

- Yes. However,
  - Computation complexity will be O(n log(n))
    - Worse than the sequential merge

- Alternative: compute p points in the path
  - *This divides work into p+1 chunks*
  - *Each chunk can be independently processed (e.g., by a thread block)*
  - *"Two-phase decomposition"*

Merge Matrix

- Size |A|x|B|
- M[i,j]=1 if A[i]<B[j]
- M[i,j]=0 if A[i]≥B[j]

		B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]
		1	2	3	5	8	9	10	12
A[1]	5	1	1	1	0	0	0	0	0
A[2]	6	1							
A[3]	7	1							
A[4]	11	1							
A[5]	13	1							
A[6]	14	1							
A[7]	15	1							
A[8]	16	1							

Merge Matrix

- O(|A||B|) space and time to compute

		B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]
		1	2	3	5	8	9	10	12
A[1]	5	1	1	1	0	0	0	0	0
A[2]	6	1	1	1	1	0	0	0	0
A[3]	7	1	1	1	1	0	0	0	0
A[4]	11	1	1	1	1	1	1	1	0
A[5]	13	1	1	1	1	1	1	1	1
A[6]	14	1	1	1	1	1	1	1	1
A[7]	15	1	1	1	1	1	1	1	1
A[8]	16	1	1	1	1	1	1	1	1

Oded Green, NVIDIA Research, 2015

Merge Matrix

- O(|A| |B|) space and time to compute
- Matrix is conceptual
- Values implicitly computed

		B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]
		1	2	3	5	8	9	10	12
A[1]	5	1	1	1	0	0	0	0	0
A[2]	6	1	1	1	1	0	0	0	0
A[3]	7	1	1	1	1	0	0	0	0
A[4]	11	1	1	1	1	1	1	1	0
A[5]	13	1	1	1	1	1	1	1	1
A[6]	14	1	1	1	1	1	1	1	1
A[7]	15	1	1	1	1	1	1	1	1
A[8]	16	1	1	1	1	1	1	1	1

Merge Matrix

- Path goes between “1”s and “0”s.

		B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]
		1	2	3	5	8	9	10	12
A[1]	5	1	1	1	0	0	0	0	0
A[2]	6	1	1	1	1	0	0	0	0
A[3]	7	1	1	1	1	0	0	0	0
A[4]	11	1	1	1	1	1	1	1	0
A[5]	13	1	1	1	1	1	1	1	1
A[6]	14	1	1	1	1	1	1	1	1
A[7]	15	1	1	1	1	1	1	1	1
A[8]	16	1	1	1	1	1	1	1	1

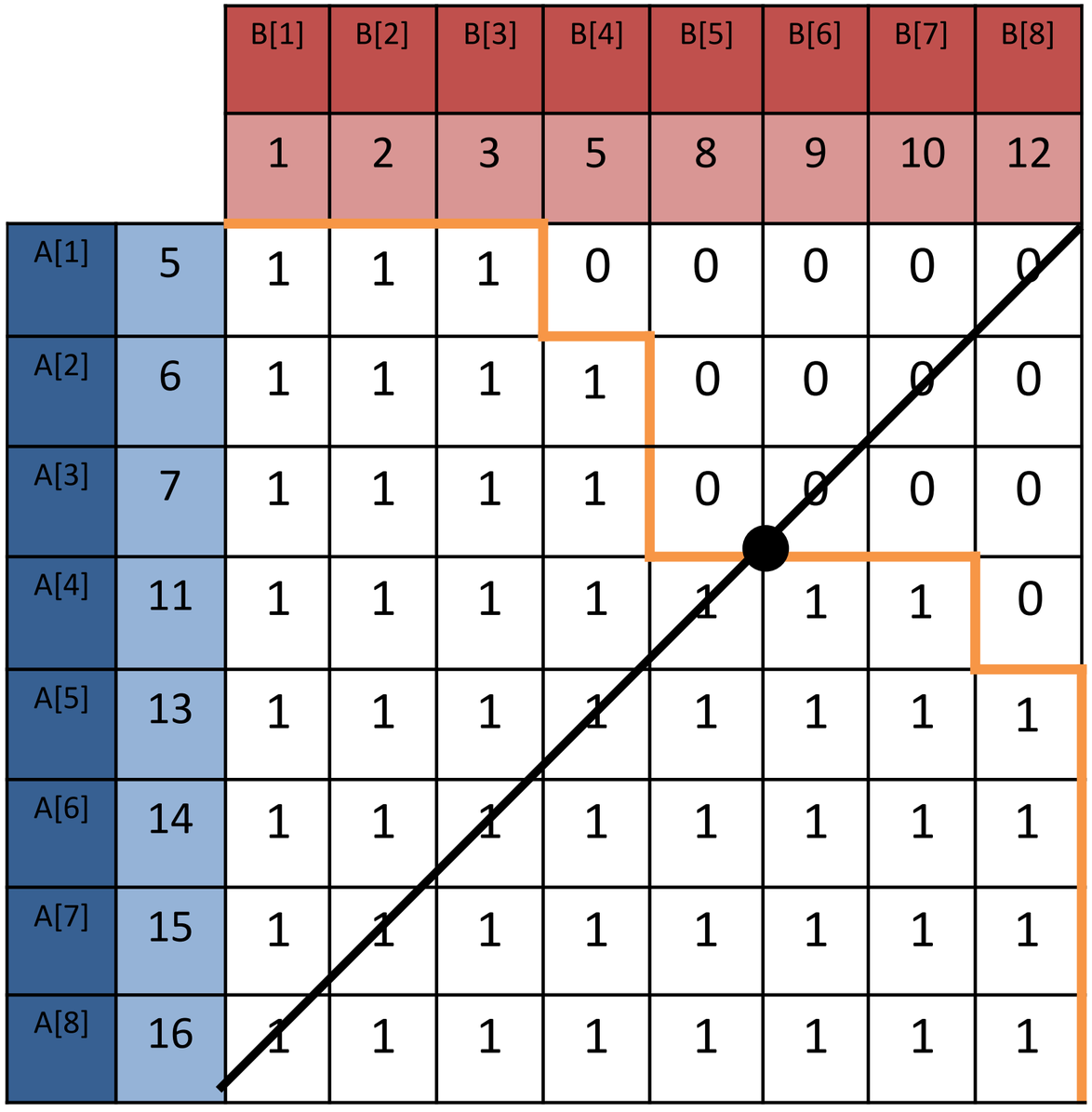
Finding Partitioning Points

		B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]
		1	2	3	5	8	9	10	12
A[1]	5	1	1	1	0	0	0	0	0
A[2]	6	1	1	1	1	0	0	0	0
A[3]	7	1	1	1	1	0	0	0	0
A[4]	11	1	1	1	1	1	1	1	0
A[5]	13	1	1	1	1	1	1	1	1
A[6]	14	1	1	1	1	1	1	1	1
A[7]	15	1	1	1	1	1	1	1	1
A[8]	16	1	1	1	1	1	1	1	1

Finding Partitioning Points

- Path
  - start = top-left
  - stop = bottom-right
- Cross Diagonals
  - start = top-right
  - stop = bottom-left

=>Intersection



Why use the cross diagonals?

	1	2	3	5	8	9	10	12
5	1	1	1	0	0	0	0	0
6	1	1	1	1	0	0	0	0
7	1	1	1	1	0	0	0	0
11	1	1	1	1	1	1	1	0
13	1	1	1	1	1	1	1	1
14	1	1	1	1	1	1	1	1
15	1	1	1	1	1	1	1	1
16	1	1	1	1	1	1	1	1

- Distance from starting point to diagonal is equal on all possible paths

Why use the cross diagonals?

	1	2	3	5	8	9	10	12
5	1	1	1	0	0	0	0	0
6	1	1	1	1	0	0	0	0
7	1	1	1	1	0	0	0	0
11	1	1	1	1	1	1	1	0
13	1	1	1	1	1	1	1	1
14	1	1	1	1	1	1	1	1
15	1	1	1	1	1	1	1	1
16	1	1	1	1	1	1	1	1

Length = 8 blocks

- Distance from starting point to diagonal is equal on all possible paths

Why use the cross diagonals?

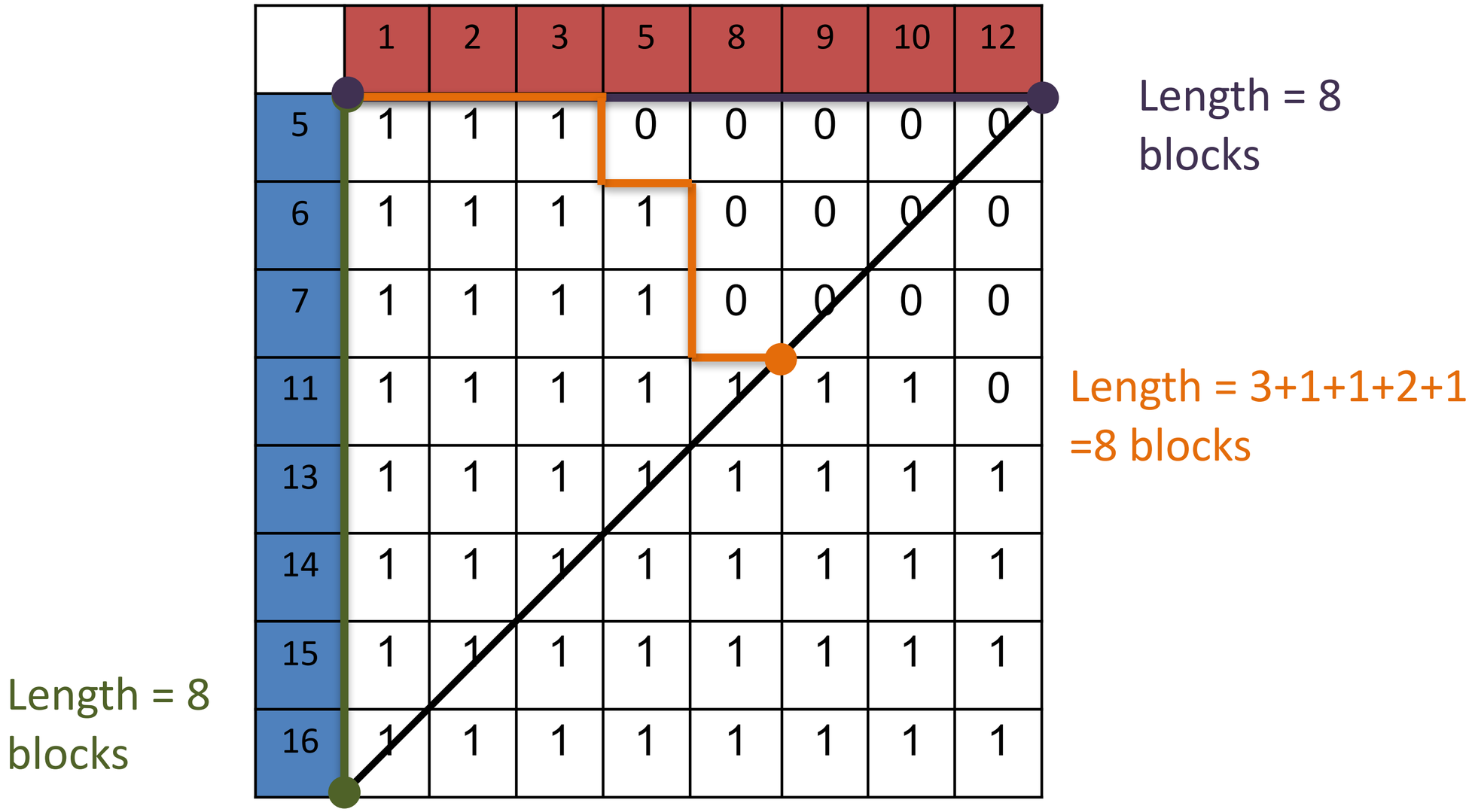
	1	2	3	5	8	9	10	12
5	1	1	1	0	0	0	0	0
6	1	1	1	1	0	0	0	0
7	1	1	1	1	0	0	0	0
11	1	1	1	1	1	1	1	0
13	1	1	1	1	1	1	1	1
14	1	1	1	1	1	1	1	1
15	1	1	1	1	1	1	1	1
16	1	1	1	1	1	1	1	1

Length = 8 blocks

Length = 8 blocks

- Distance from starting point to diagonal is equal on all possible paths

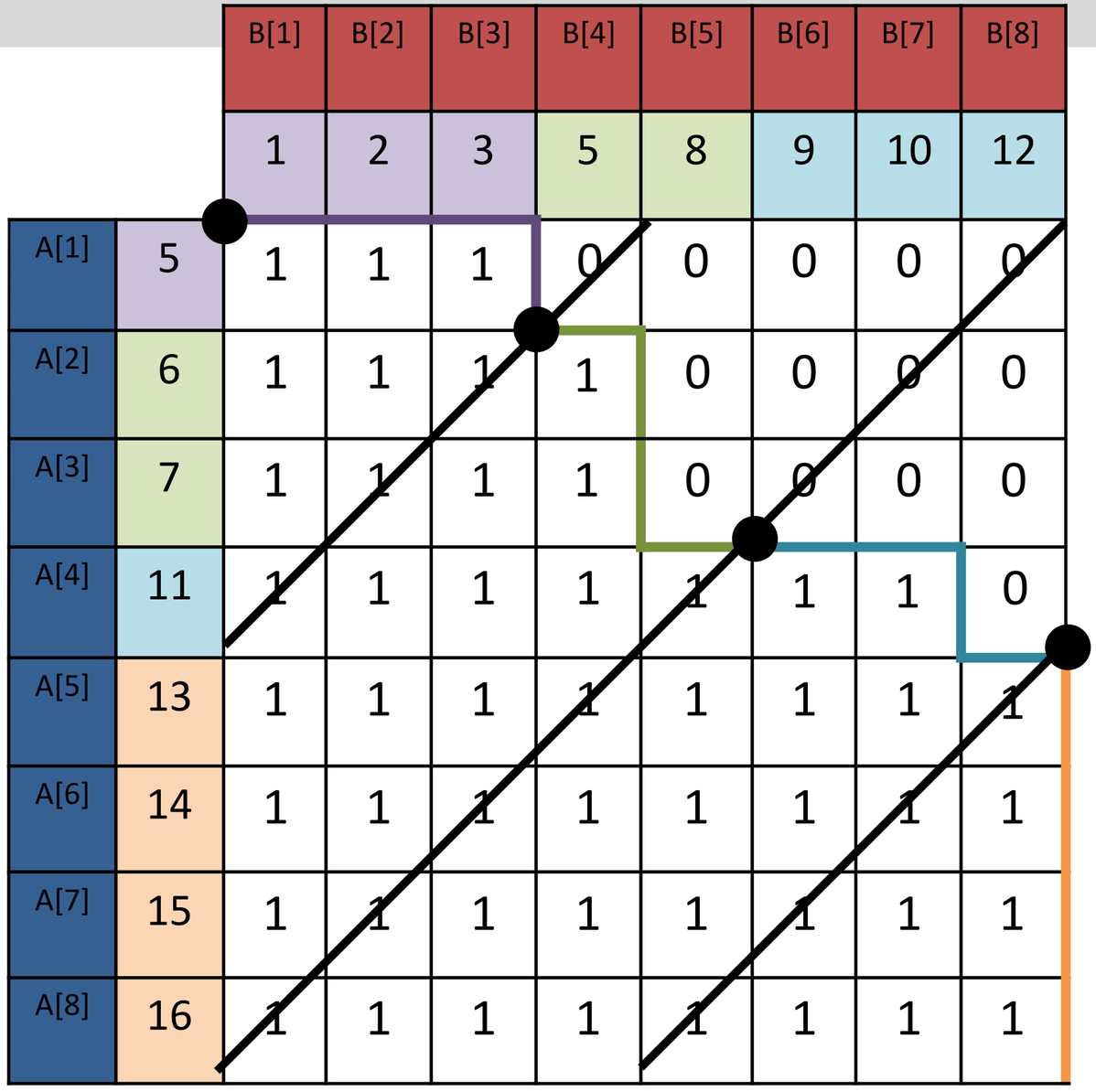
Why use the cross diagonals?



- Distance from starting point to diagonal is equal on all possible paths

P – Cross Diagonals

- P points for P processors
- How do we select the points?
- Binary search on equidistant diagonals



Output:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Merge-Path-Based SpMV

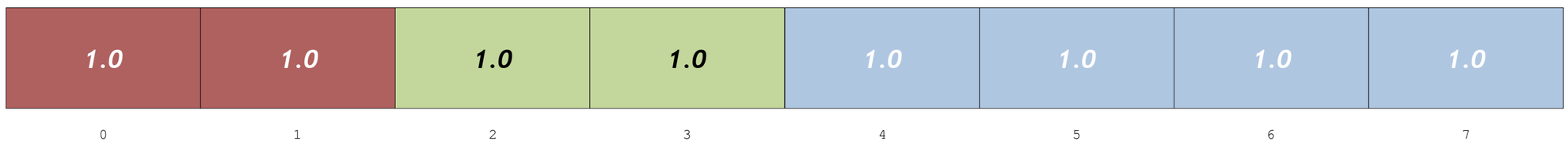
Merge-based Parallel Sparse Matrix-Vector Multiplication using the CSR Storage Format, Duane Merrill and Michael Garland, NVIDIA Research

UCD EEC 289Q, Winter 2018

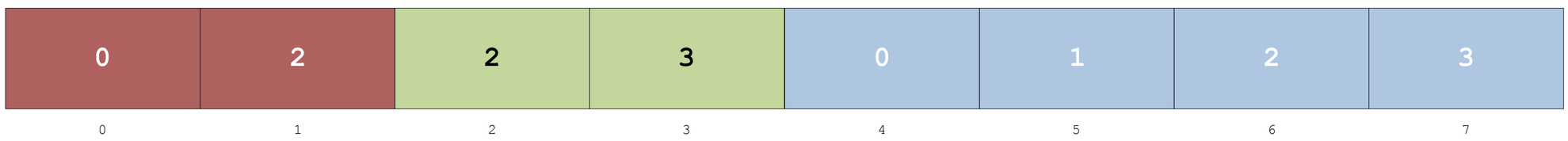
CSR sparse matrix **A**

```
1.0  --  1.0  --
 --   --   --   --
 --   --  1.0  1.0
1.0  1.0  1.0  1.0
```

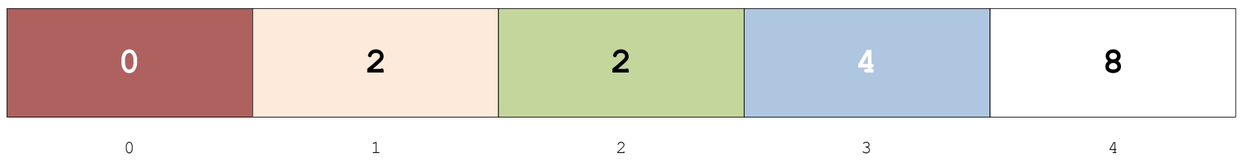
values$_{\mathbf{A}}$



column_indices$_{\mathbf{A}}$

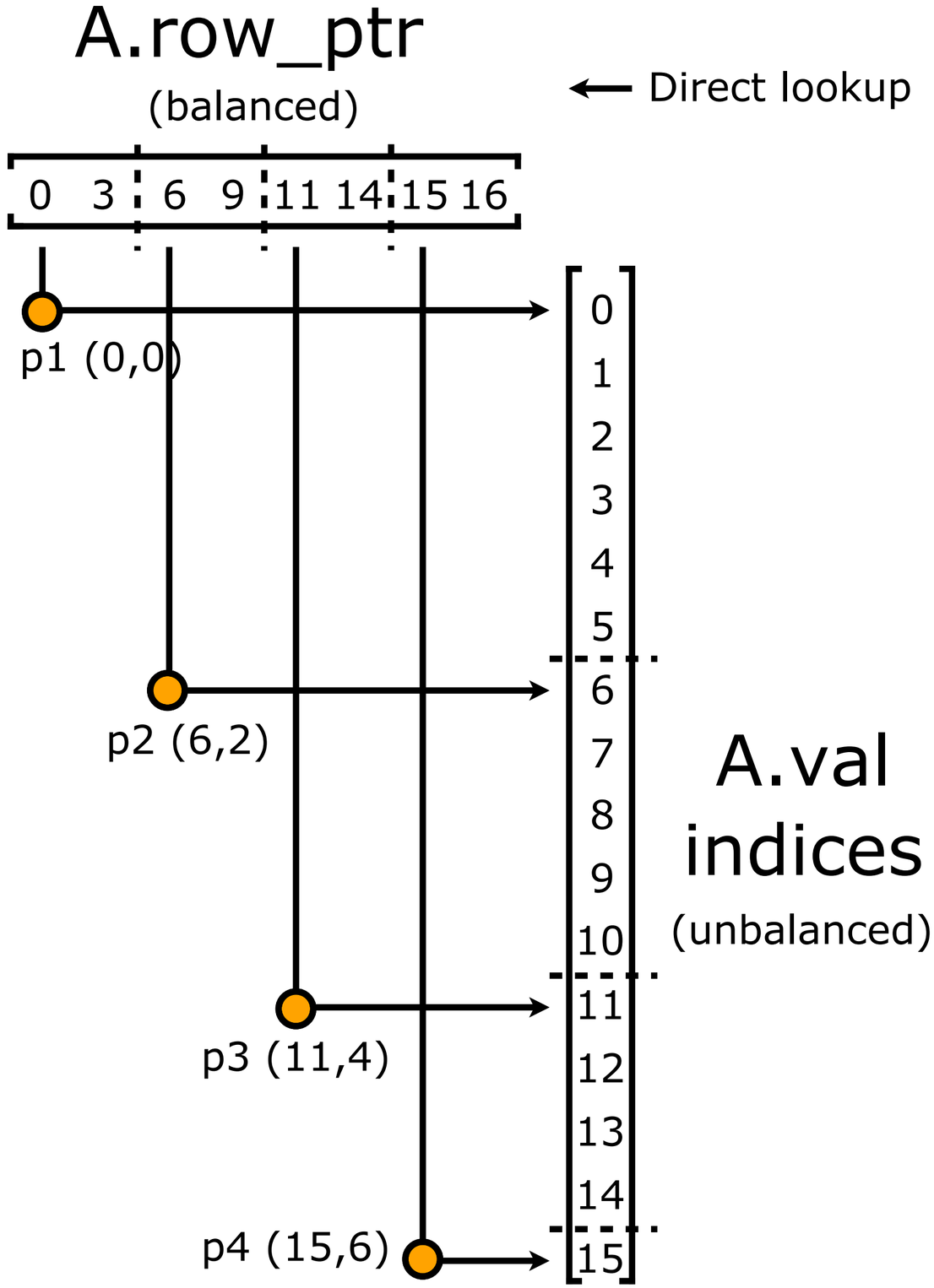


row_offsets$_{\mathbf{A}}$



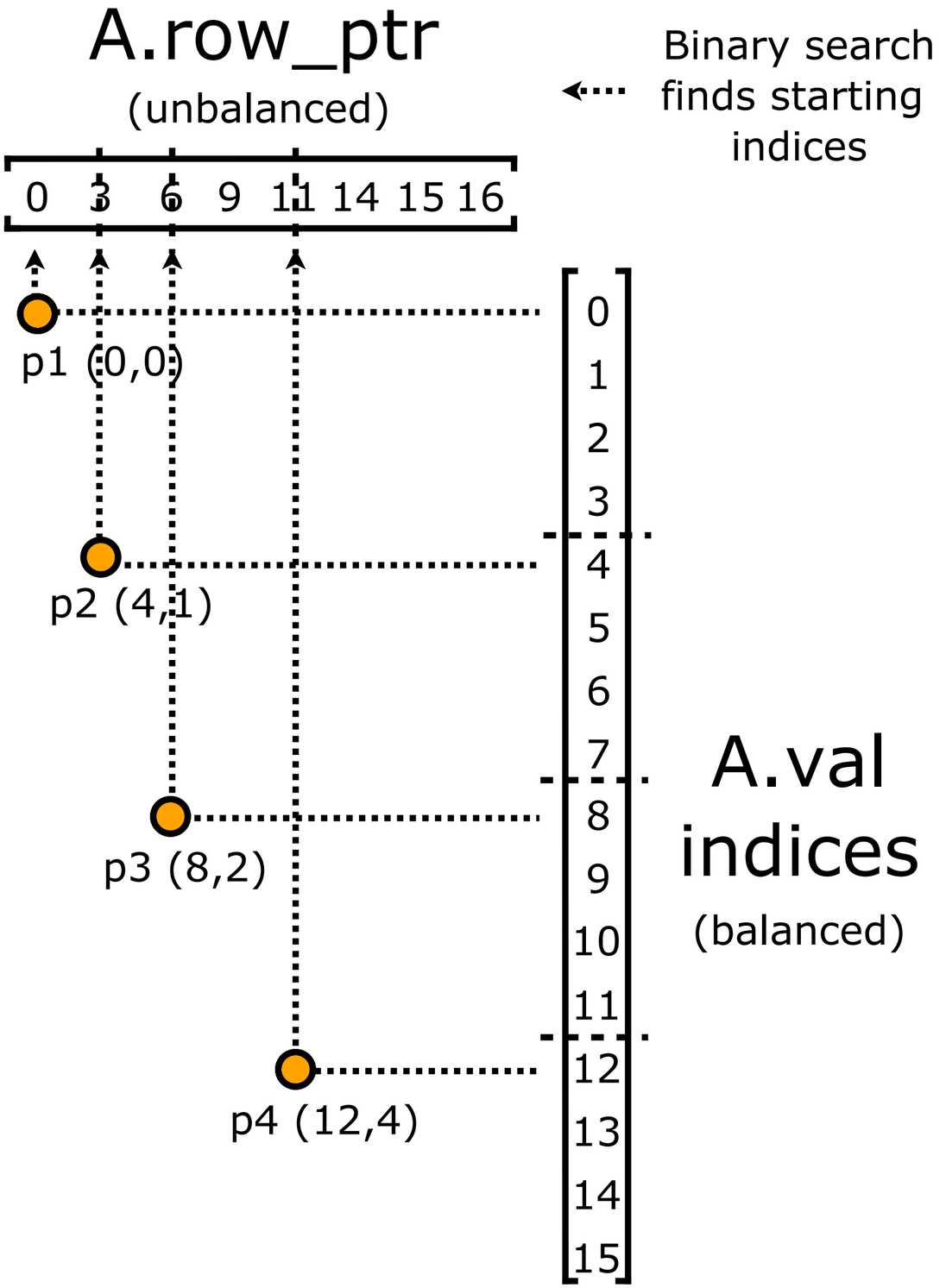
Row split

- Balanced number of rows per parallel unit
- Unbalanced number of indices



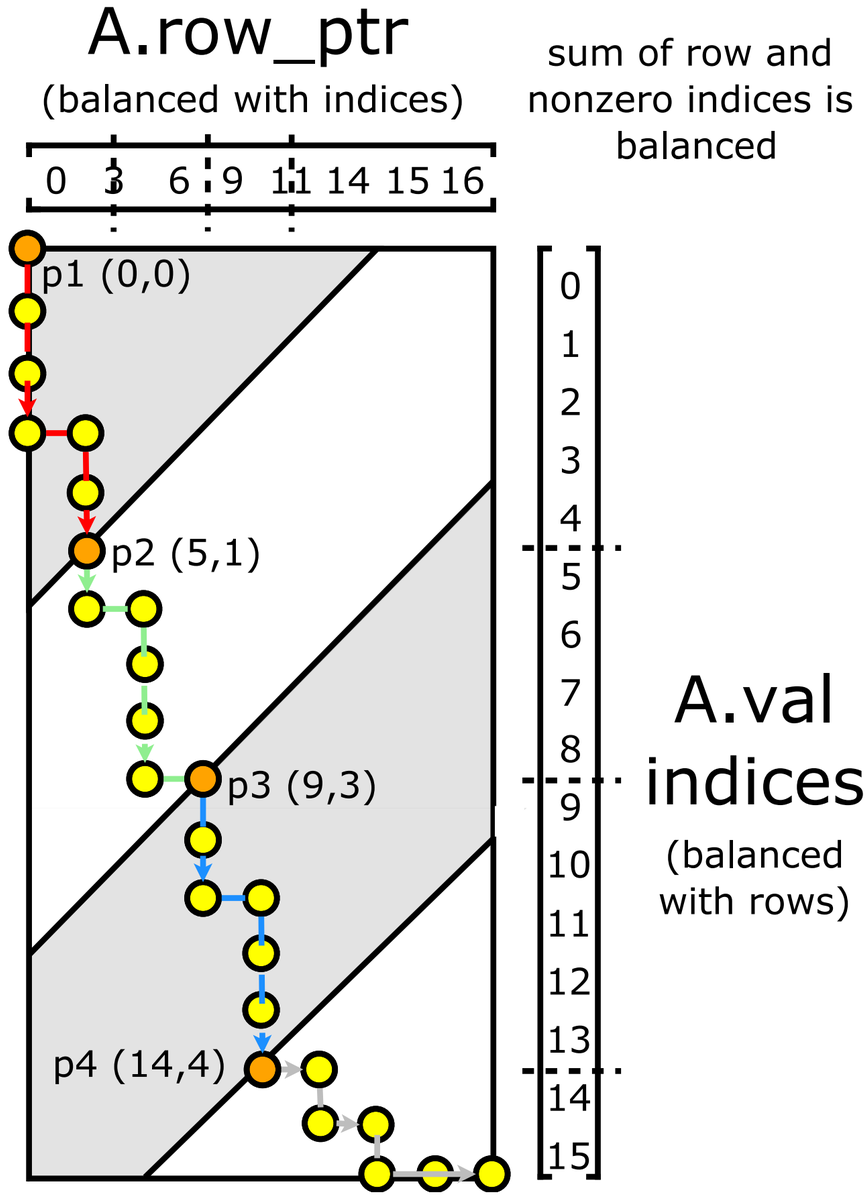
Non-zero splitting

- **Balanced number of indices per parallel unit**
- **Unbalanced number of rows**



Key idea for next formulation

- Every time we hit a new row we have to do something
- Every time we hit a new non-zero we have to do something
- Let's load-balance (rows + non-zeroes)



CSR sparse matrix **A**

$$\begin{bmatrix} 1.0 & -- & 1.0 & -- \\ -- & -- & -- & -- \\ -- & -- & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 & 1.0 \end{bmatrix}$$

values$_{\mathbf{A}}$

1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	1	2	3	4	5	6	7

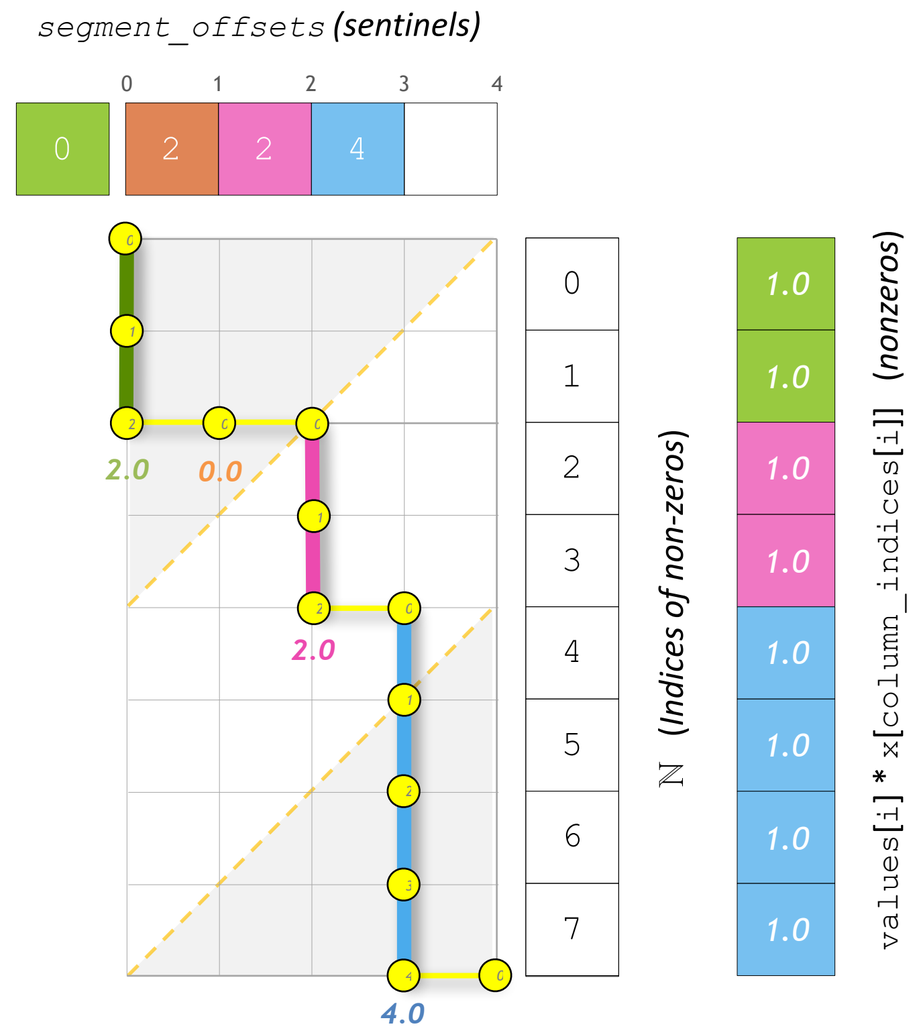
column_indices$_{\mathbf{A}}$

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

row_offsets$_{\mathbf{A}}$

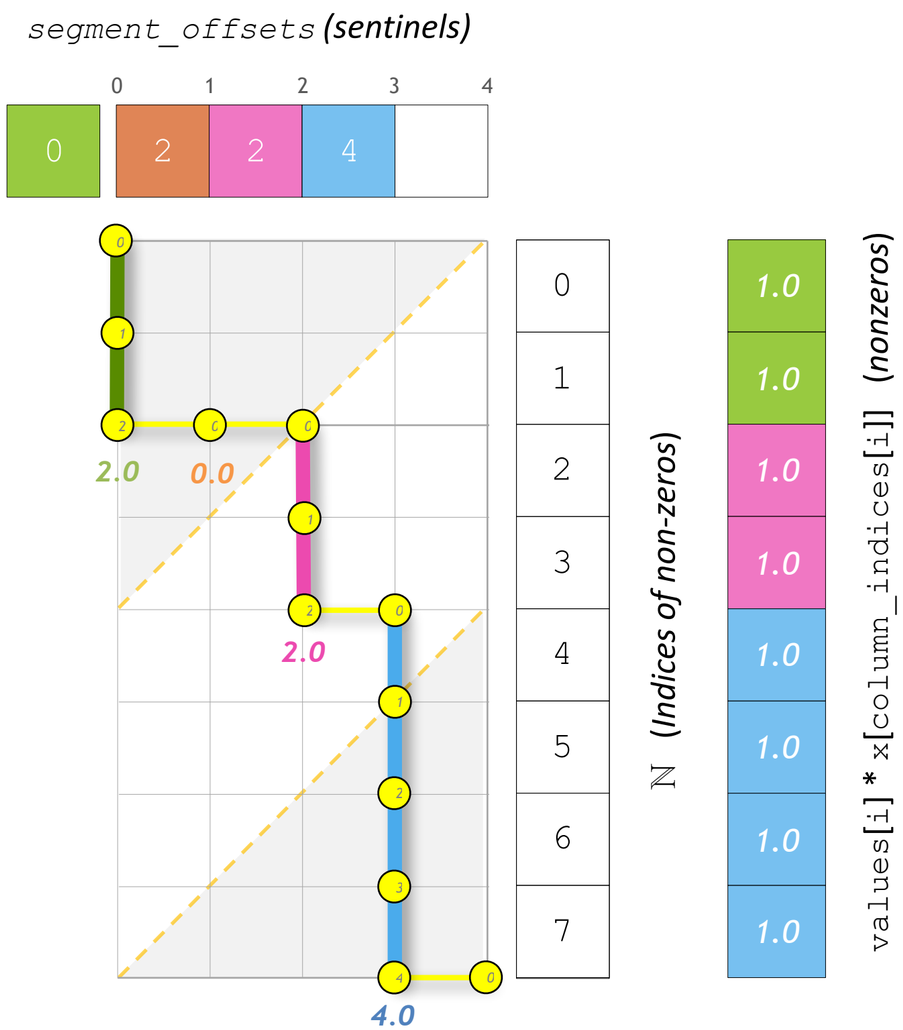
0	2	2	4	8
0	1	2	3	4

Merge-based CsrMV



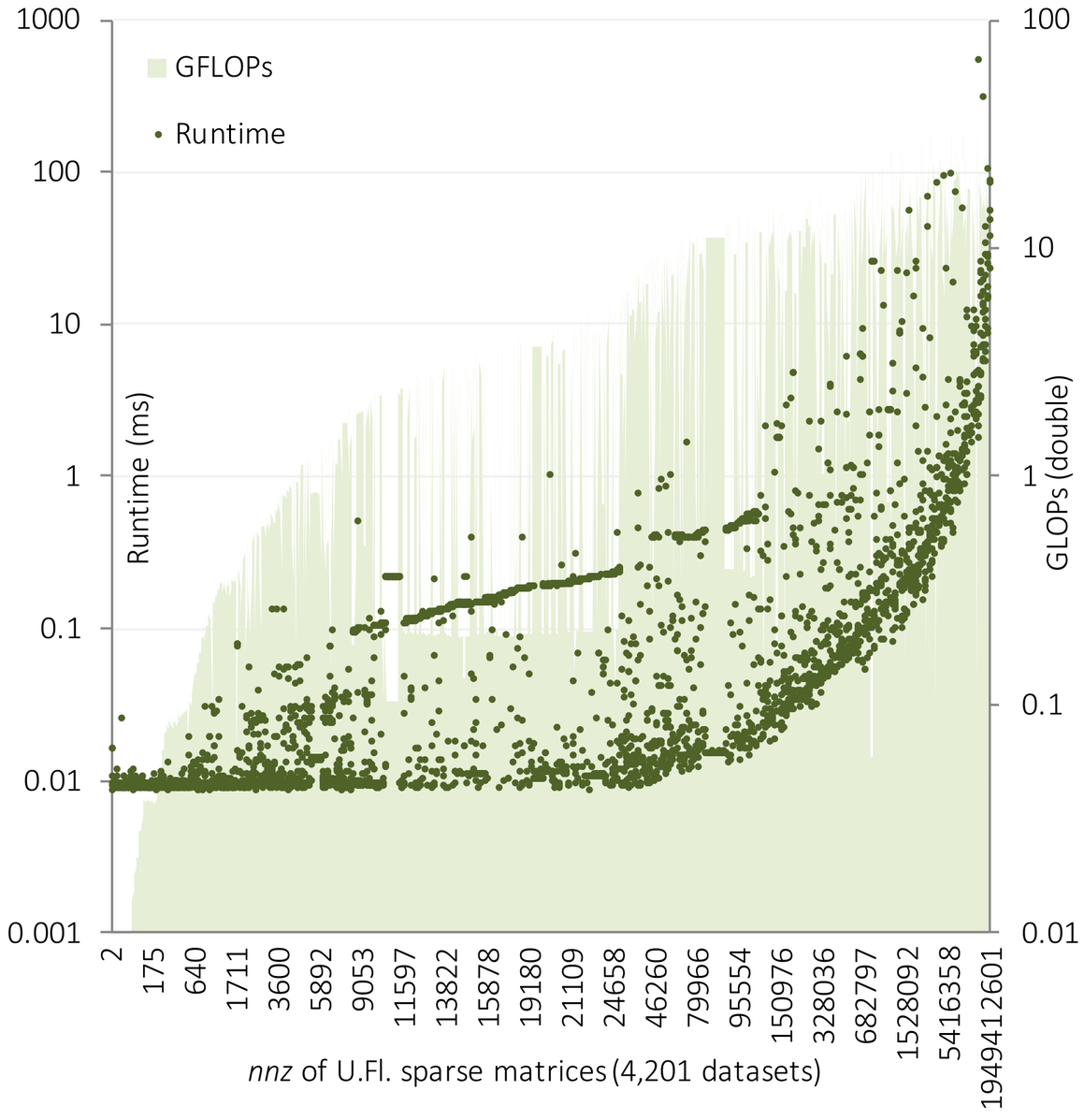
Key idea: Logically merge the row end-offsets with the indices of the matrix non-zeros

Merge-based CsrMV

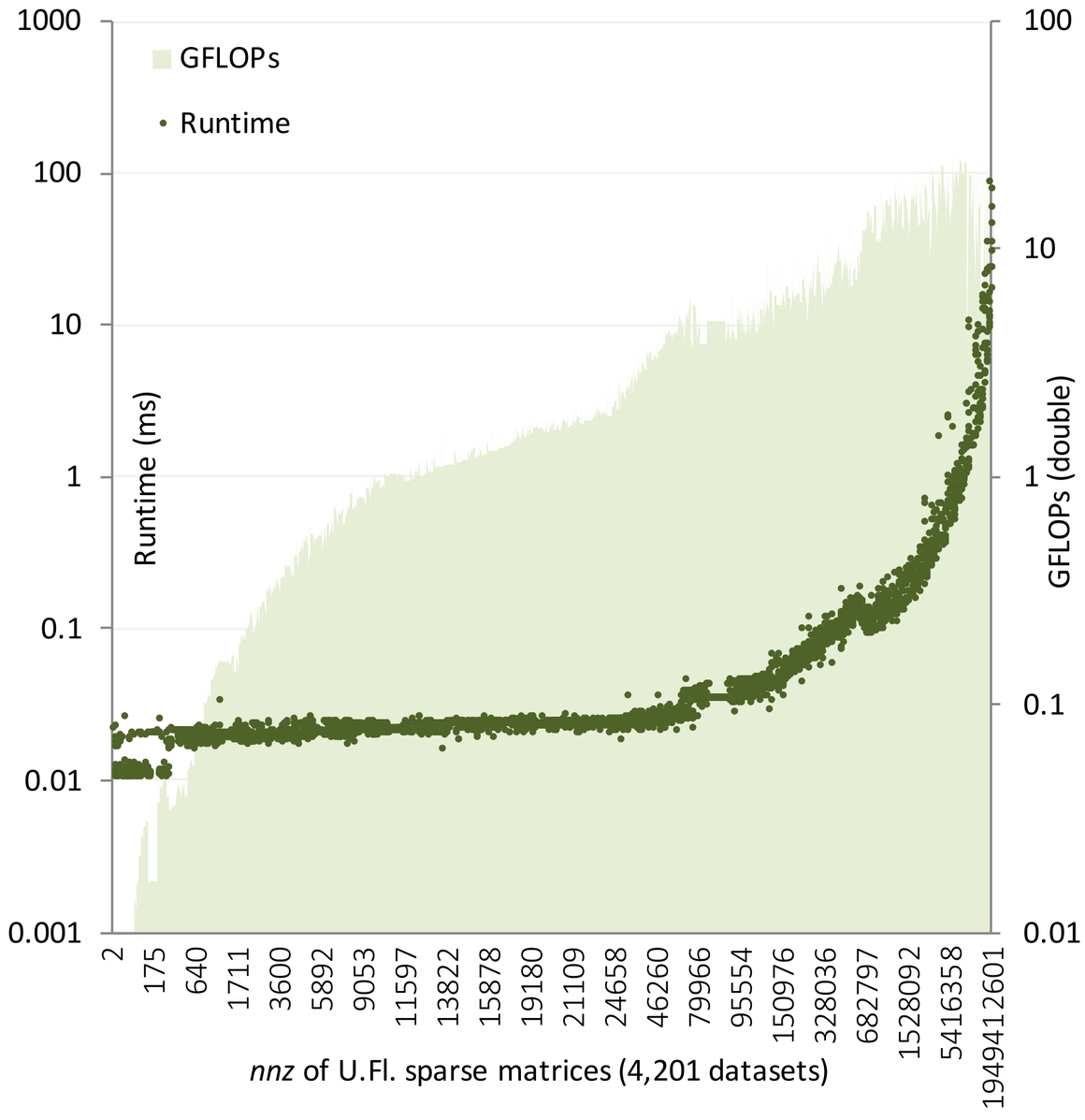



```
// read all relevant data into shared memory
...
// Consume exactly items-per-thread merge items
double running_total = 0.0;
for (int i = 0; i < items_per_thread; ++i) {

if (nz_indices[thread_coord.y] <
    row_end_offsets[thread_coord.x]) {

  // Move down (accumulate)

  running_total += A.values[thread_coord.y] *
          x[A.column_indices[thread_coord.y]];

  ++thread_coord.y;

} else {

  // Move right (output row-total and reset)
  y[thread_coord.x] = running_total;

  running_total; = 0.0;

 ++thread_coord.x;

}

// there's a fix-up step after this
```

Merrill/Garland SpMV performance



(a) NVIDIA cuSPARSE CsrMV



(b) Our merge-based CsrMV