MERGE-BASED SpMV

PERFECT WORKLOAD BALANCE. GUARANTEED.

Duane Merrill, NVIDIA Research

NVIDIA

SPARSE MATRIX-VECTOR MULTIPLICATION

SpMV ($Ax = y$)

$$\begin{bmatrix} 1.0 & \text{--} & 1.0 & \text{--} \\ \text{--} & \text{--} & \text{--} & \text{--} \\ \text{--} & \text{--} & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 & 1.0 \end{bmatrix} * \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 0.0 \\ 2.0 \\ 4.0 \end{bmatrix}$$

sparse matrix **A**

dense vector *x*

dense vector *y*

SPARSE MATRIX-VECTOR MULTIPLICATION

Lots of available parallelism

$$\begin{bmatrix} 1.0 & -- & 1.0 & -- \\ -- & -- & -- & -- \\ -- & -- & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 & 1.0 \end{bmatrix} * \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} (1.0)(1.0) + (1.0)(1.0) \\ 0.0 \\ (1.0)(1.0) + (1.0)(1.0) \\ (1.0)(1.0) + (1.0)(1.0) + (1.0)(1.0) + (1.0)(1.0) \end{bmatrix}$$

sparse matrix **A**

dense vector *x*

dense vector *y*

PARALLEL DECOMPOSITION

COMPRESSED SPARSE ROW (CSR) FORMAT

3-array representation

$$
\begin{array}{c|c}
 & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \left[\begin{array}{cccc} 1.0 & \text{--} & 1.0 & \text{--} \\ \text{--} & \text{--} & \text{--} & \text{--} \\ \text{--} & \text{--} & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 & 1.0 \end{array}\right]
\end{array}
$$

A

	0	1	2	3	4	5	6	7
values	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
column indices	0	2	2	3	0	1	2	3

	0	1	2	3	4
row offsets	0	2	2	4	8

CSR PARALLEL DECOMPOSITION

a) Row-based

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

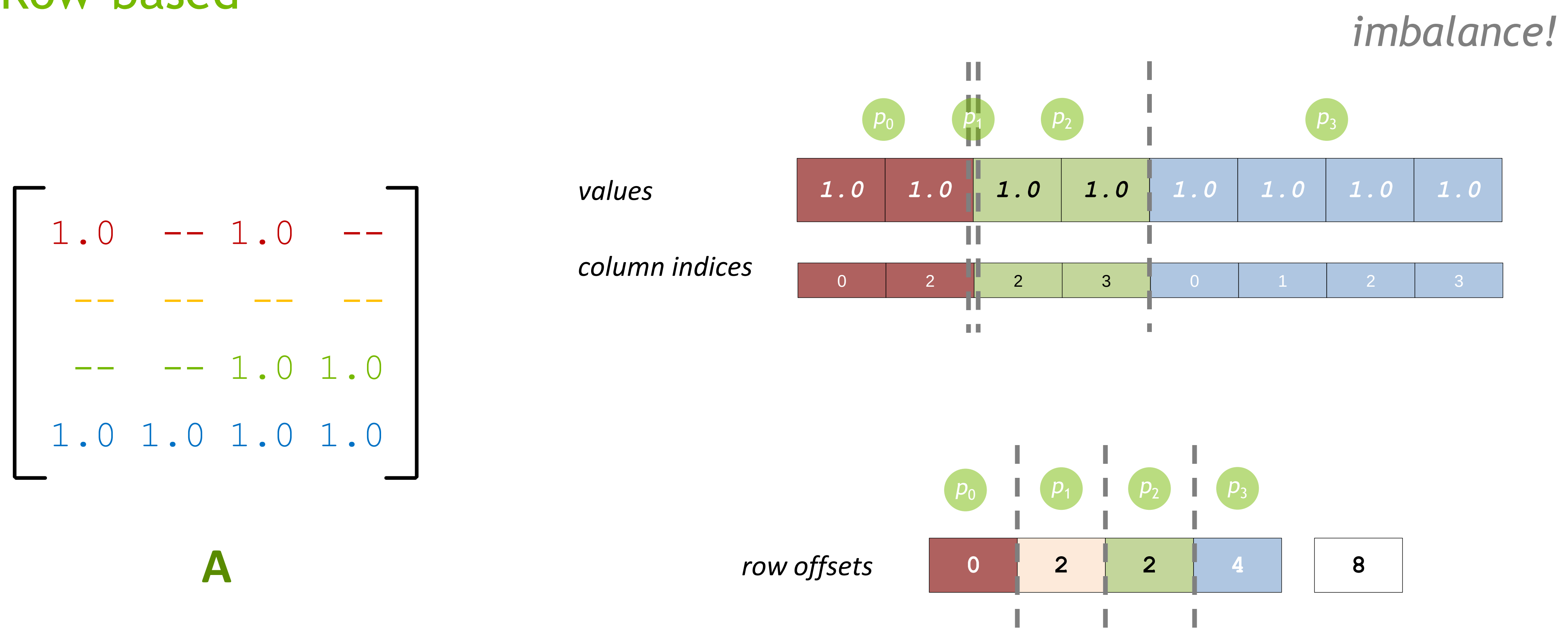
A

values	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
column indices	0	2	2	3	0	1	2	3

	p_0	p_1	p_2	p_3	
row offsets	0	2	2	4	8

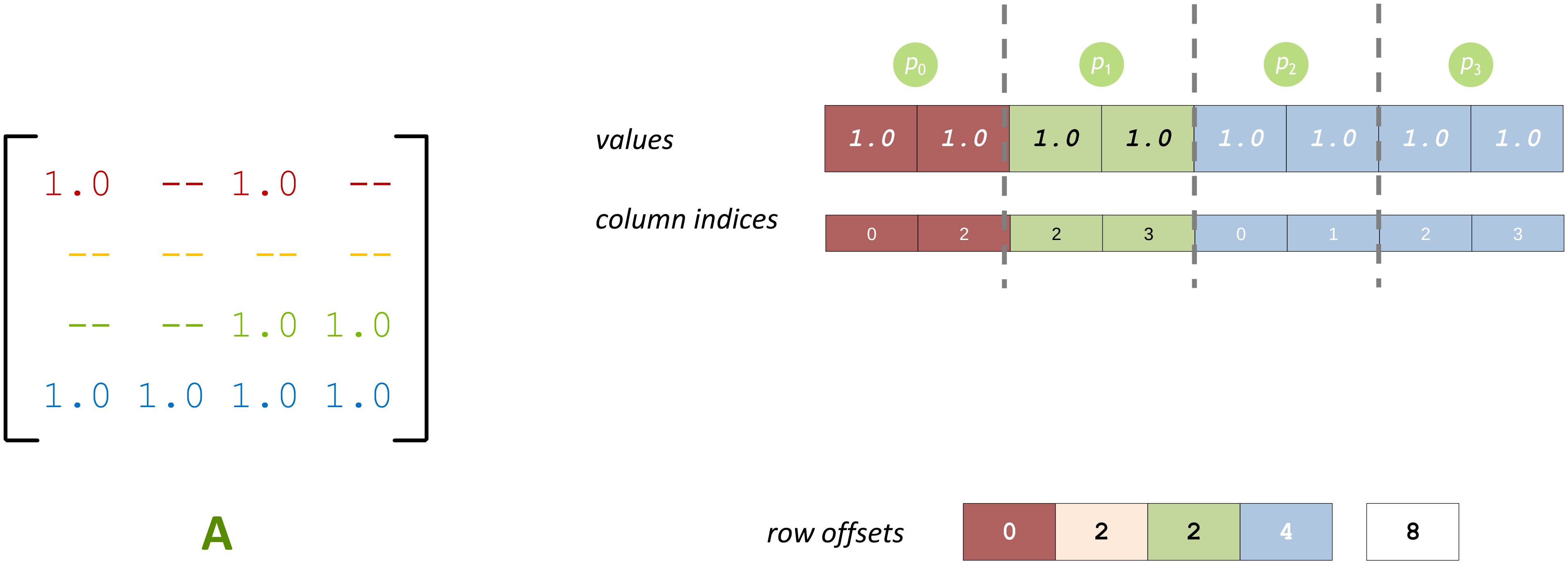
CSR PARALLEL DECOMPOSITION

a) Row-based



CSR PARALLEL DECOMPOSITION

b) Nonzero splitting

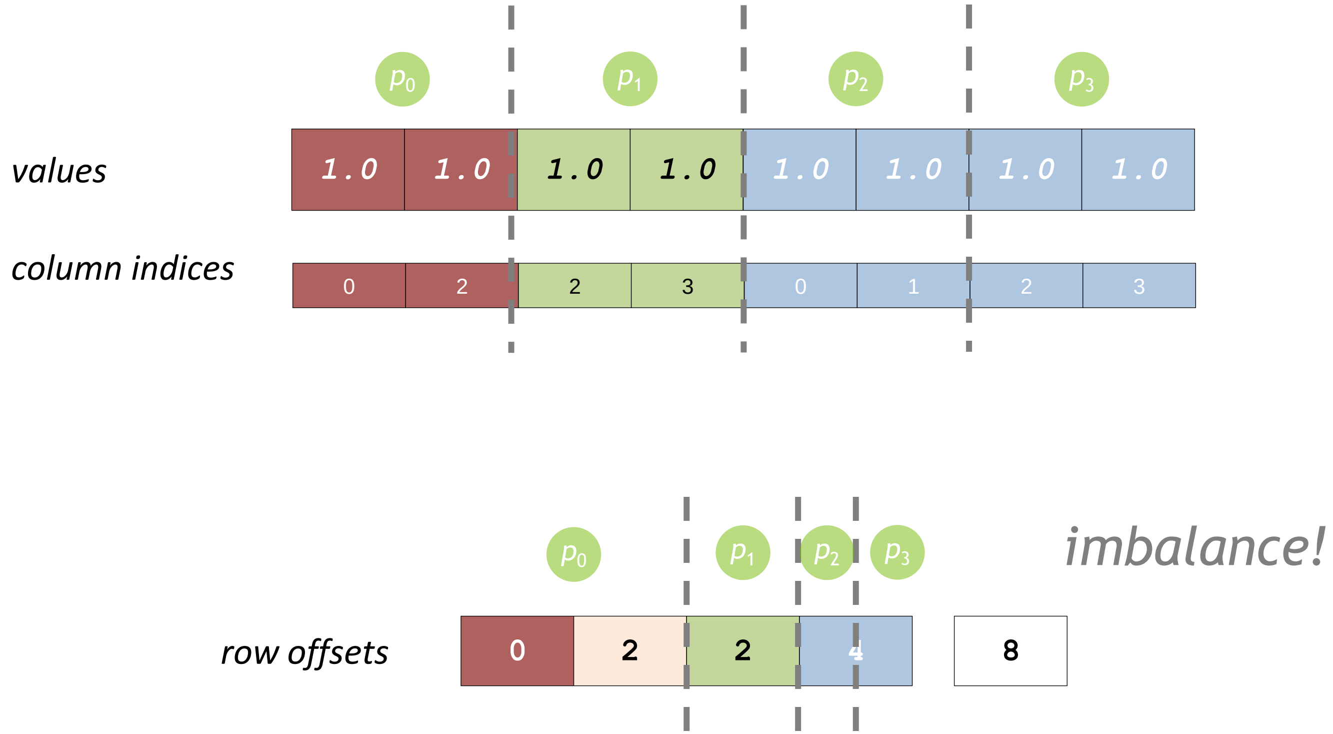


CSR PARALLEL DECOMPOSITION

b) Nonzero splitting

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

A

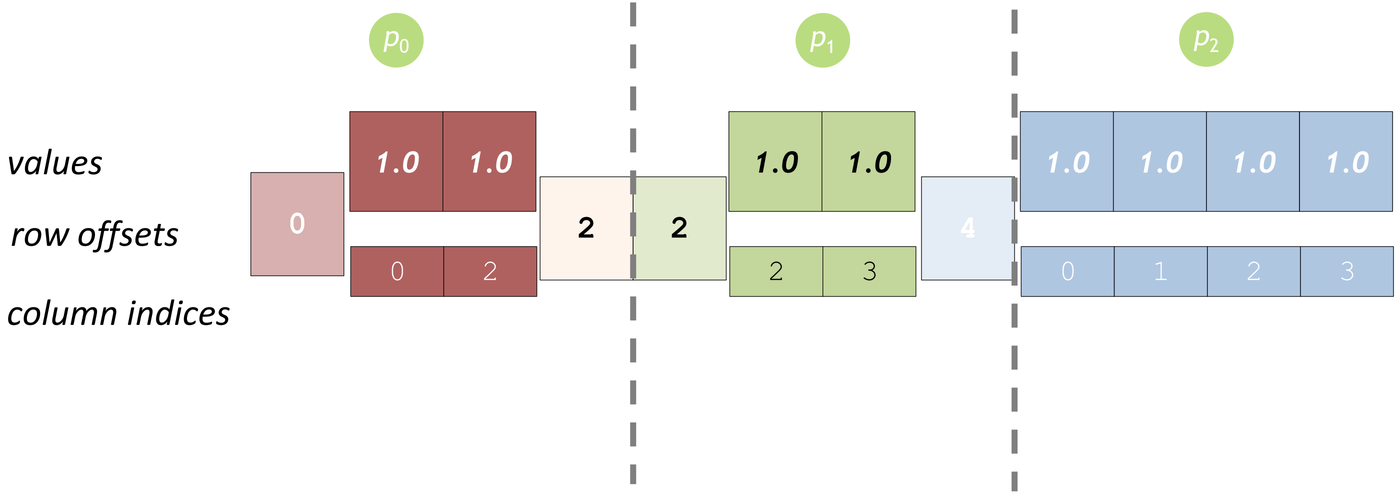


CSR PARALLEL DECOMPOSITION

c) Merge-based (logical)

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

A



NVIDIA

PERFORMANCE CONSISTENCY

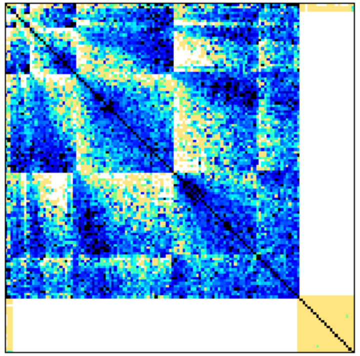
"Consistency is far better than rare moments of greatness"

-Scott Ginsberg

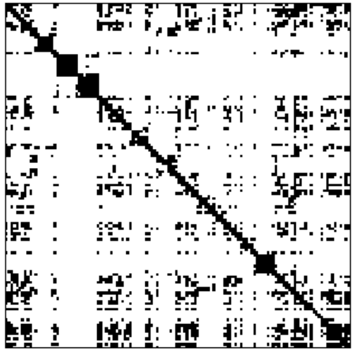
Examples

`csrmv`() with ~35M non-zeros (K40, fp64)

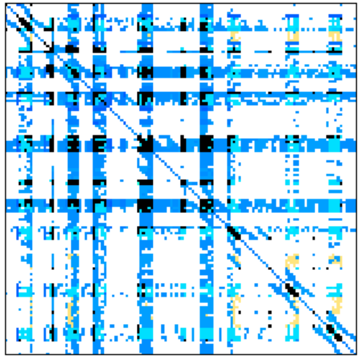
thermomech_dK
(temperature deformation)



cnr-2000
(Web connectivity)



ASIC_320k
(circuit simulation)



cuSPARSE *(row-based, vectorized parallel decomposition):*

12.4 GFLOPS

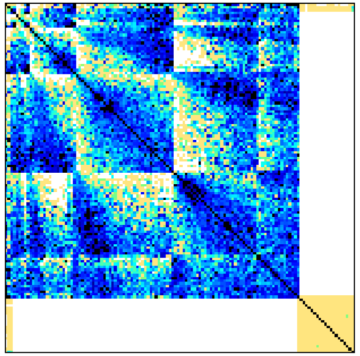
5.9 GFLOPS

0.12 GFLOPS

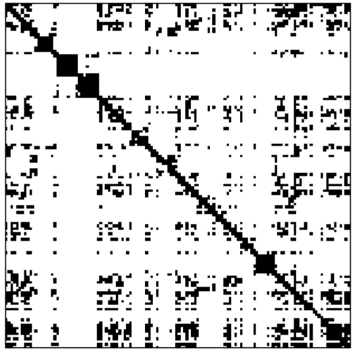
Examples

`csrmv`() with ~35M non-zeros (K40, fp64)

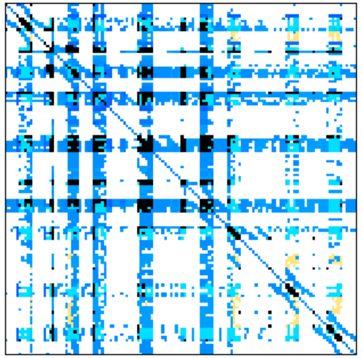
thermomech_dK
(temperature deformation)



cnr-2000
(Web connectivity)



ASIC_320k
(circuit simulation)



cuSPARSE *(row-based, vectorized parallel decomposition):*

12.4 GFLOPS	5.9 GFLOPS	0.12 GFLOPS

Merge-based:

15.5 GFLOPS	16.7 GFLOPS	14.1 GFLOPS

PERFORMANCE (IN)CONSISTENCY

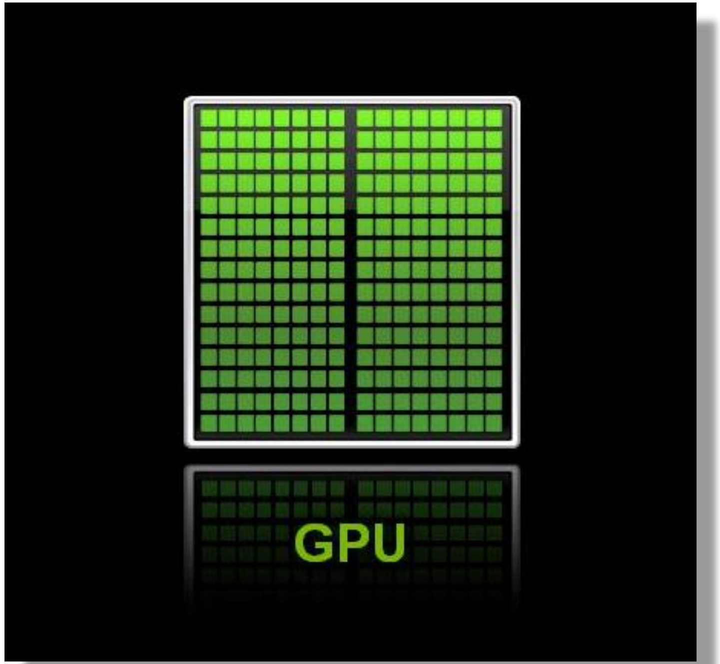
Parallelization shouldn't introduce artifacts in the performance landscape

Sources of data-dependent performance artifacts:

- Contention
- Workload imbalance

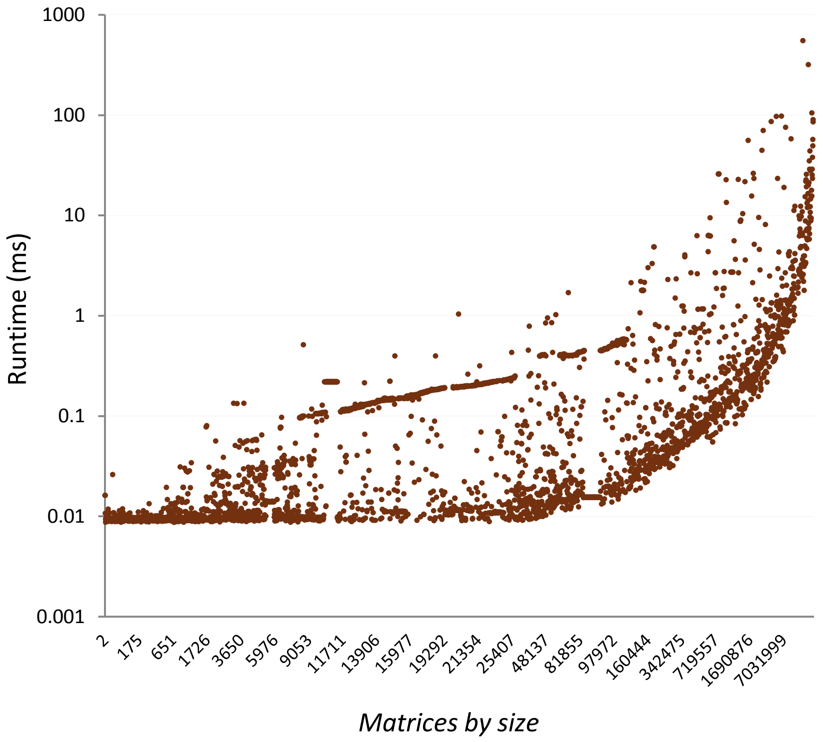
Exacerbated on massively parallel GPUs

>60 specialized GPU-specific SpMV algorithms and sparse matrix formats!

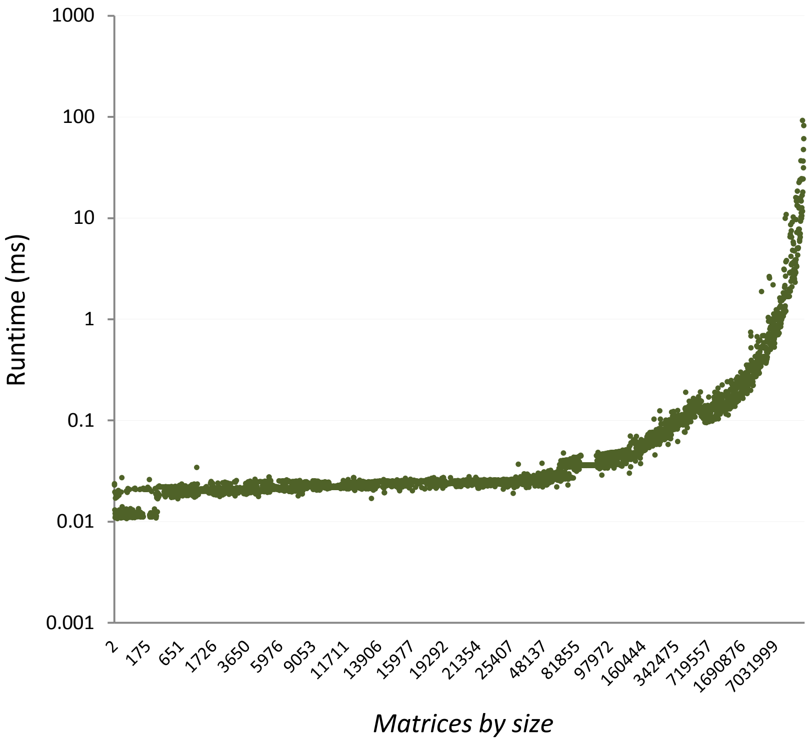


SPMV PERFORMANCE LANDSCAPE

The entire Florida Sparse Matrix Collection: 4.2K datasets (K40, fp64 CsrMV)



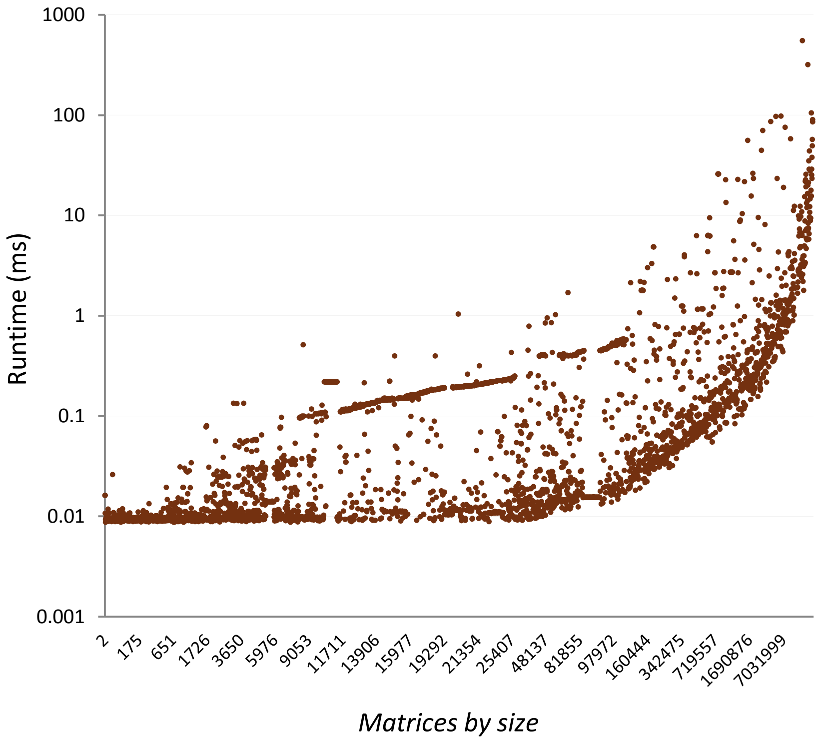
cuSPARSE



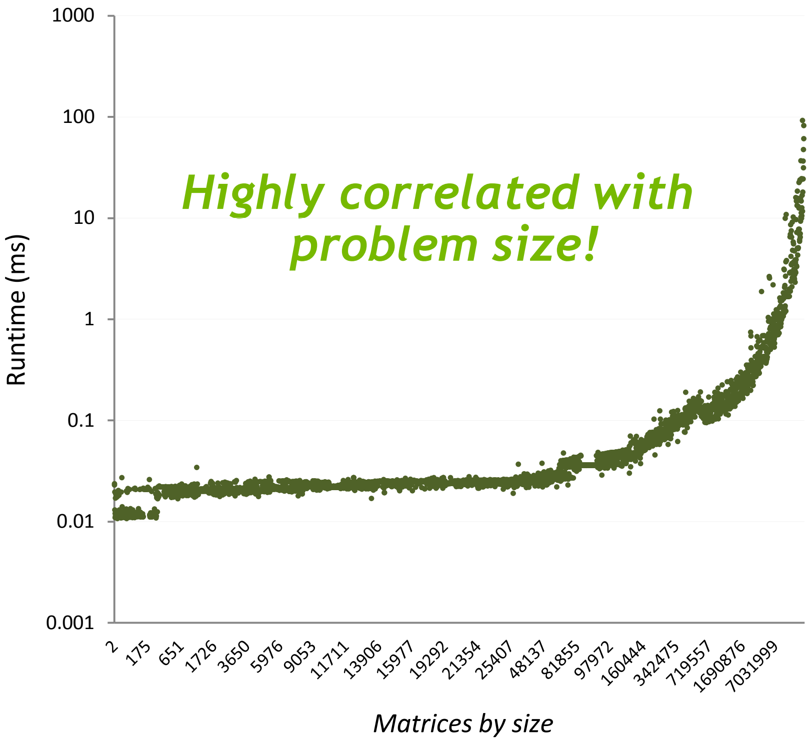
Merge-based

SPMV PERFORMANCE LANDSCAPE

The entire Florida Sparse Matrix Collection: 4.2K datasets (K40, fp64 CsrMV)



cuSPARSE



Merge-based

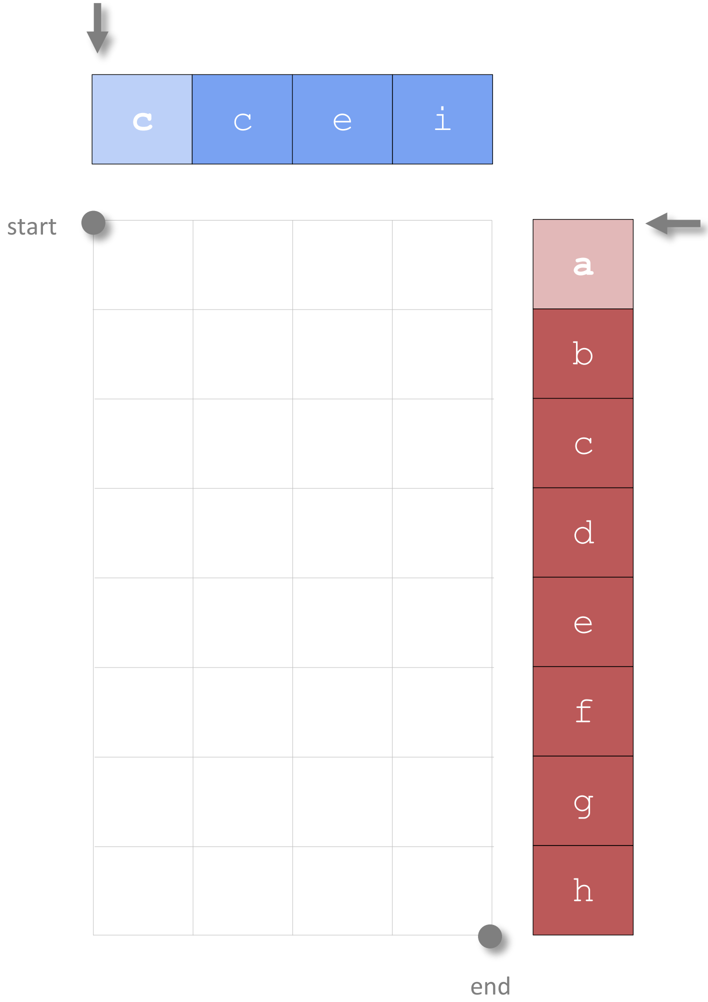
2D "MERGE PATH" DECOMPOSITION

Narsingh Deo, Amit Jain, and Muralidhar Medidi. 1994. *An optimal parallel algorithm for merging using multiselection*. Inf. Process. Lett. 50, 2 (April 1994), 81-87.

Odeh, S. et al. 2012. *Merge Path - Parallel Merging Made Simple*. Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (Washington, DC, USA, 2012), 1611–1618

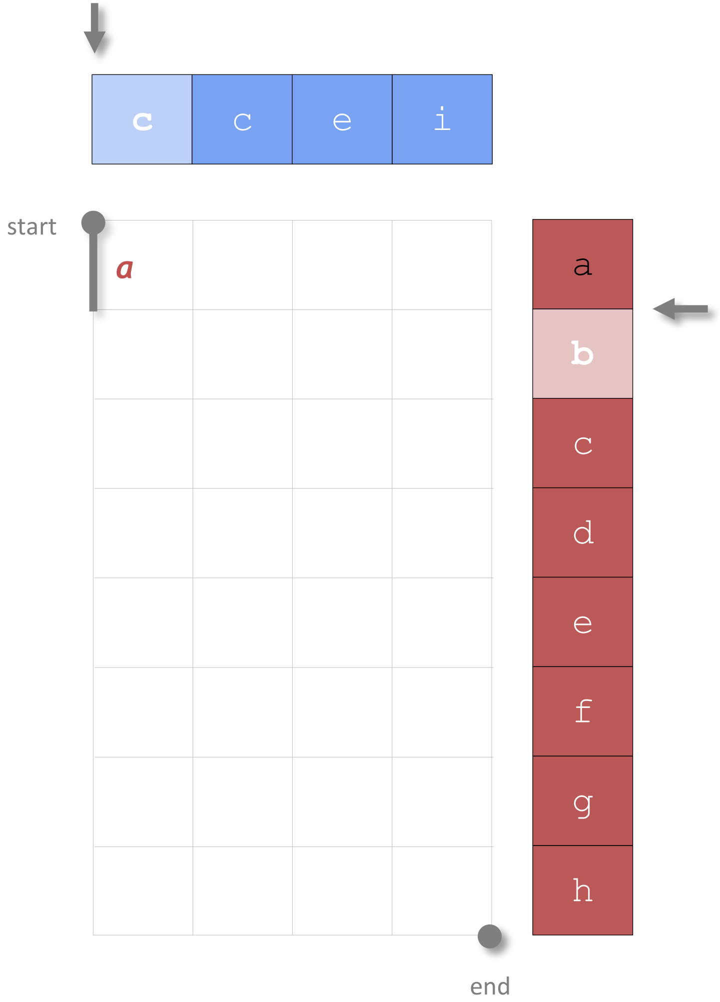
The 2D “merge-path” visualization

- The decision path runs from top-left to bottom-right:
  - Moves right when consuming from *list*$_A$
  - Moves down when consuming from *list*$_B$
  - Each step produces an output
  - Break ties by always preferring the element from *list*$_A$



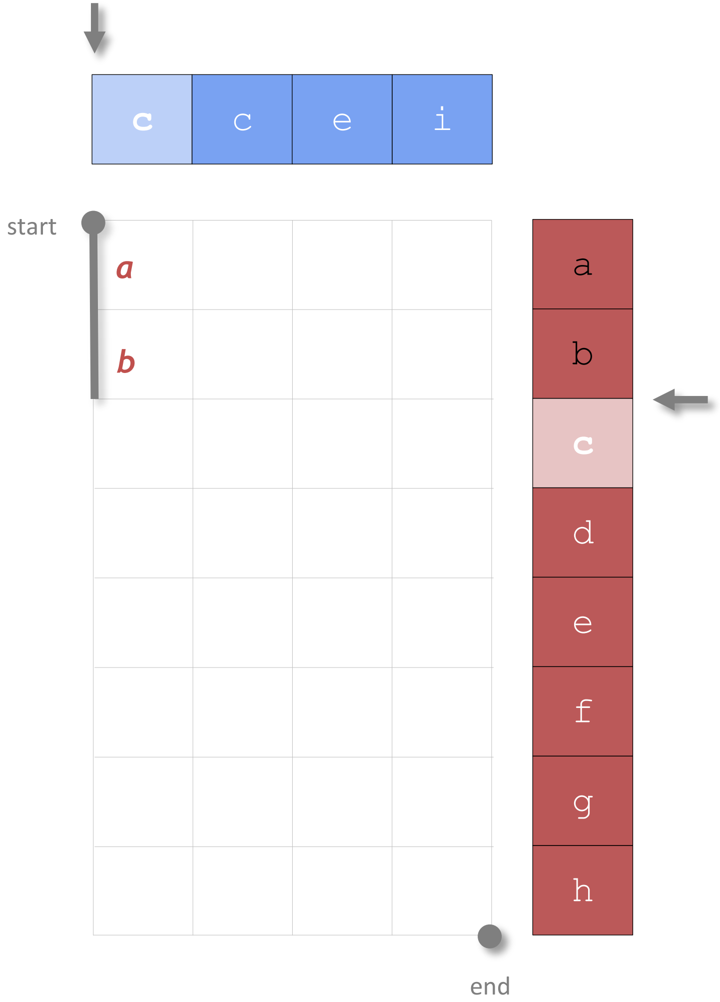
The 2D "merge-path" visualization

- The decision path runs from top-left to bottom-right:
  - Moves right when consuming from *list*$_A$
  - Moves down when consuming from *list*$_B$
  - Each step produces an output
  - Break ties by always preferring the element from *list*$_A$



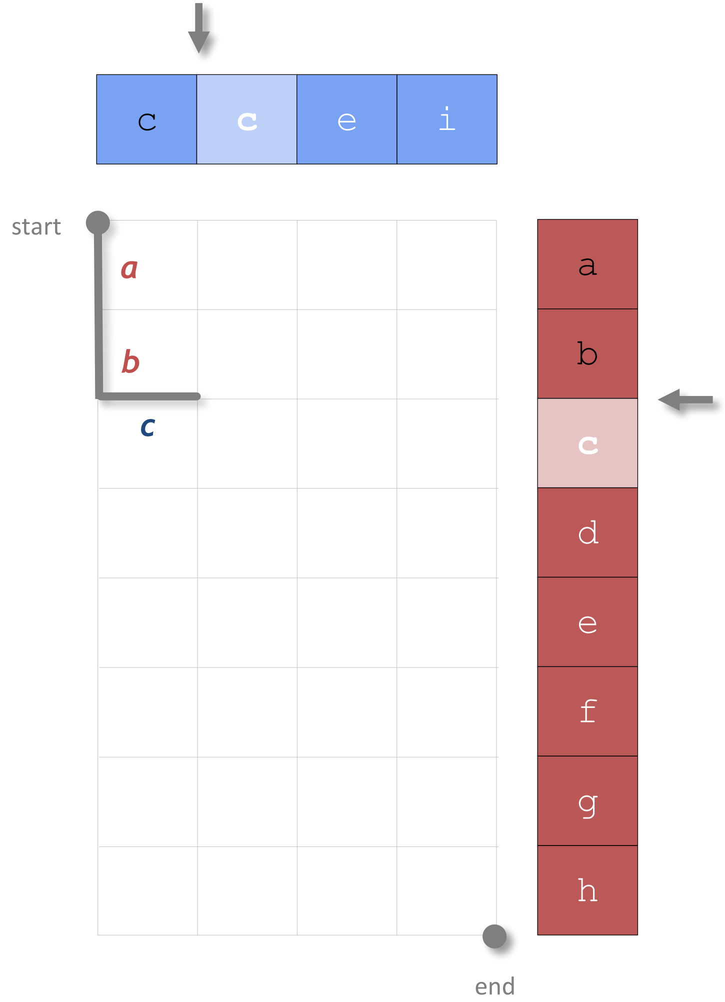
The 2D "merge-path" visualization

- The decision path runs from top-left to bottom-right:
  - Moves right when consuming from $list_A$
  - Moves down when consuming from $list_B$
  - Each step produces an output
  - Break ties by always preferring the element from $list_A$



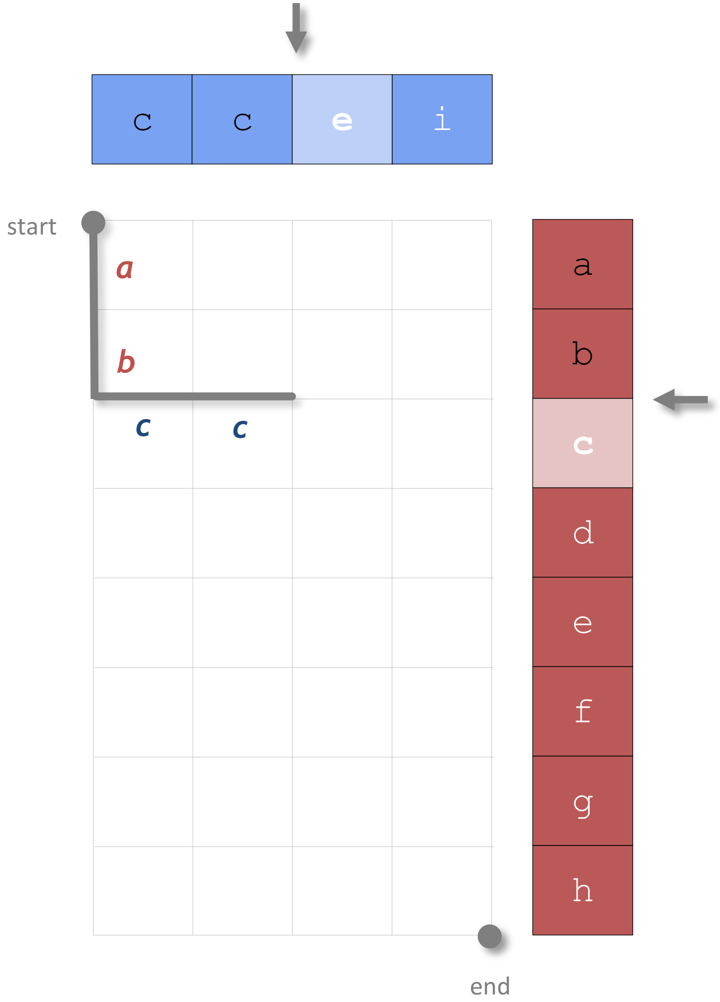
The 2D “merge-path” visualization

- The decision path runs from top-left to bottom-right:
  - Moves right when consuming from $list_A$
  - Moves down when consuming from $list_B$
  - Each step produces an output
  - Break ties by always preferring the element from $list_A$



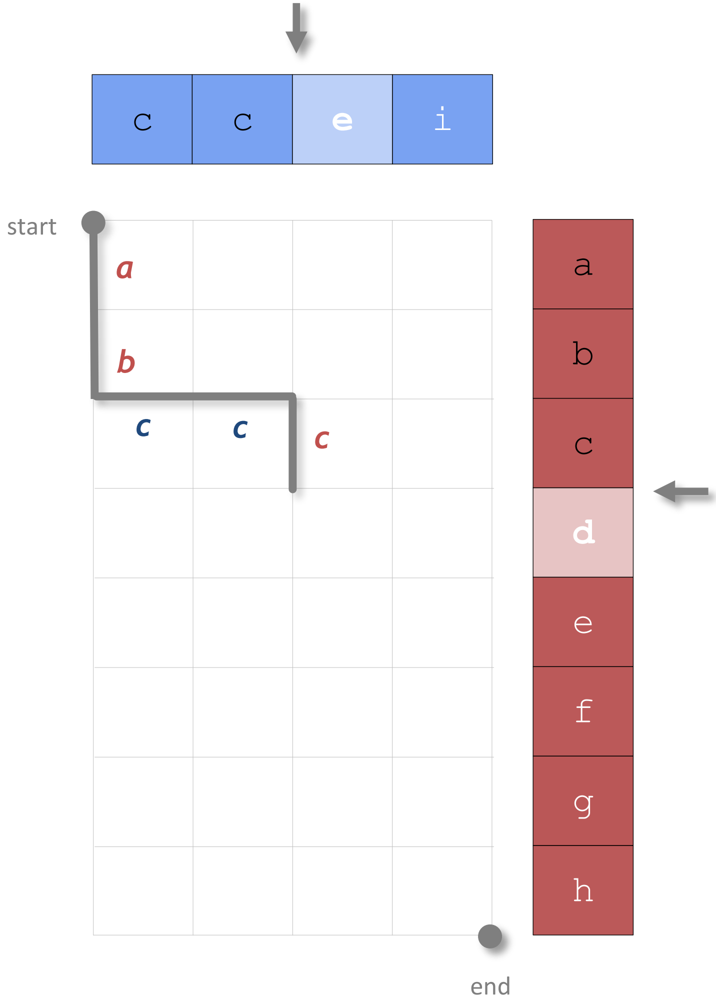
The 2D "merge-path" visualization

- The decision path runs from top-left to bottom-right:
  - Moves right when consuming from *list*$_A$
  - Moves down when consuming from *list*$_B$
  - Each step produces an output
  - Break ties by always preferring the element from *list*$_A$



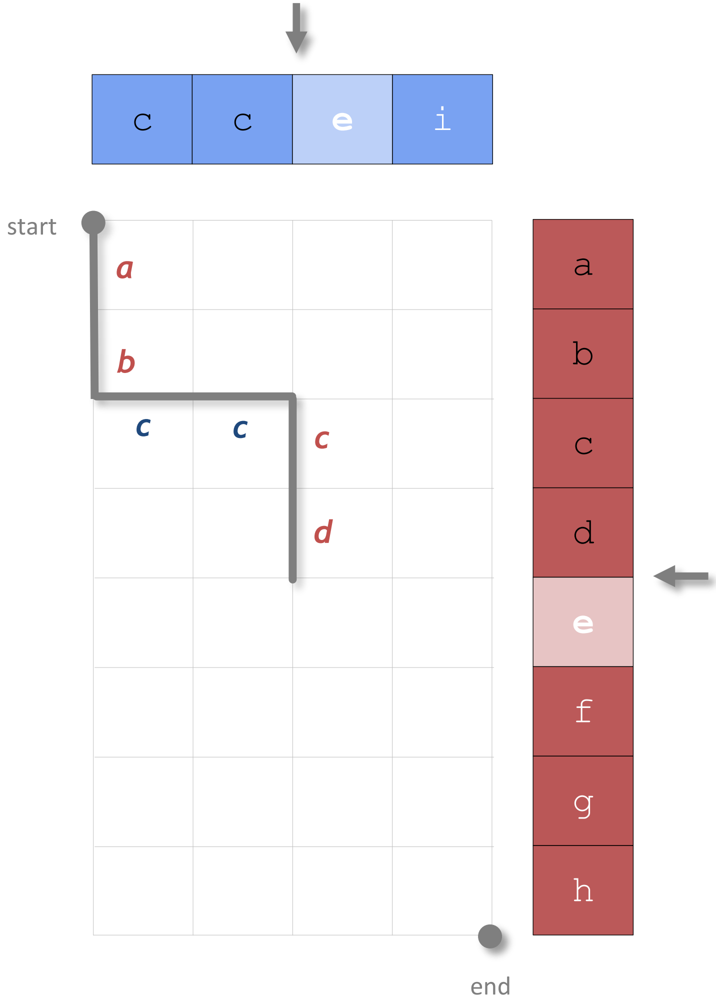
The 2D “merge-path” visualization

- The decision path runs from top-left to bottom-right:
  - Moves right when consuming from *list*$_A$
  - Moves down when consuming from *list*$_B$
  - Each step produces an output
  - Break ties by always preferring the element from *list*$_A$



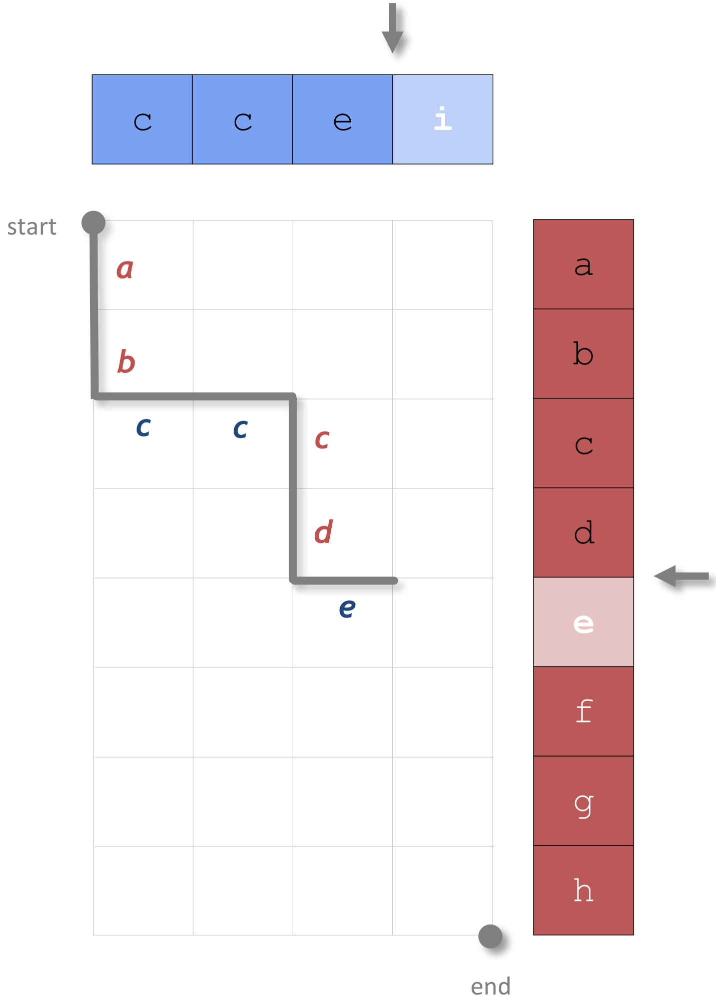
The 2D "merge-path" visualization

- The decision path runs from top-left to bottom-right:
  - Moves right when consuming from *list*$_A$
  - Moves down when consuming from *list*$_B$
  - Each step produces an output
  - Break ties by always preferring the element from *list*$_A$



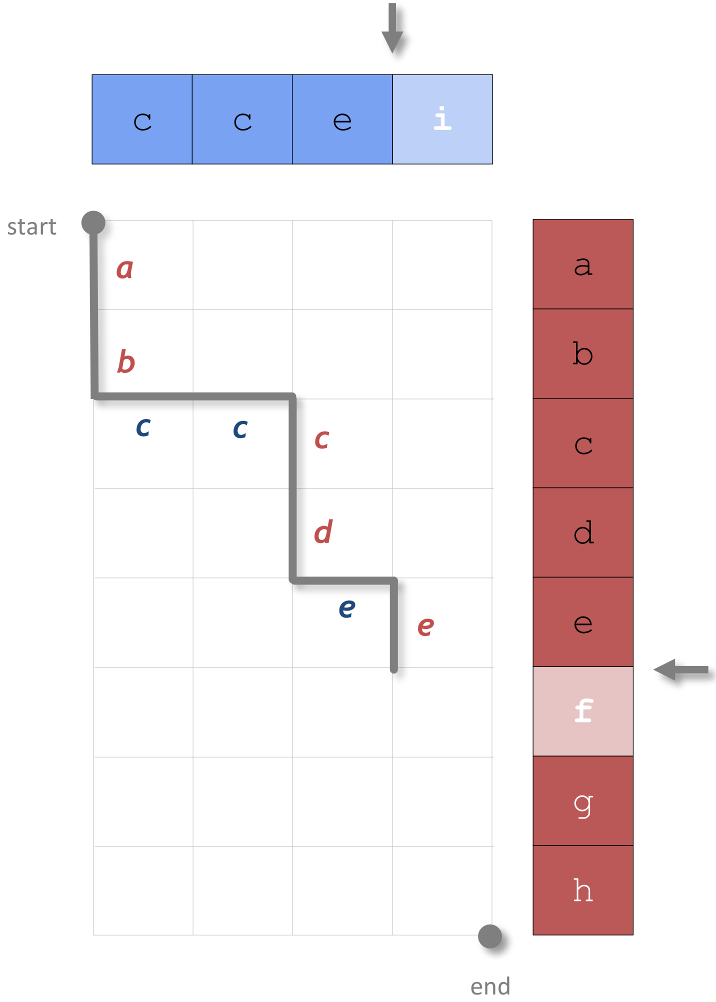
The 2D "merge-path" visualization

- The decision path runs from top-left to bottom-right:
  - Moves right when consuming from $list_A$
  - Moves down when consuming from $list_B$
  - Each step produces an output
  - Break ties by always preferring the element from $list_A$



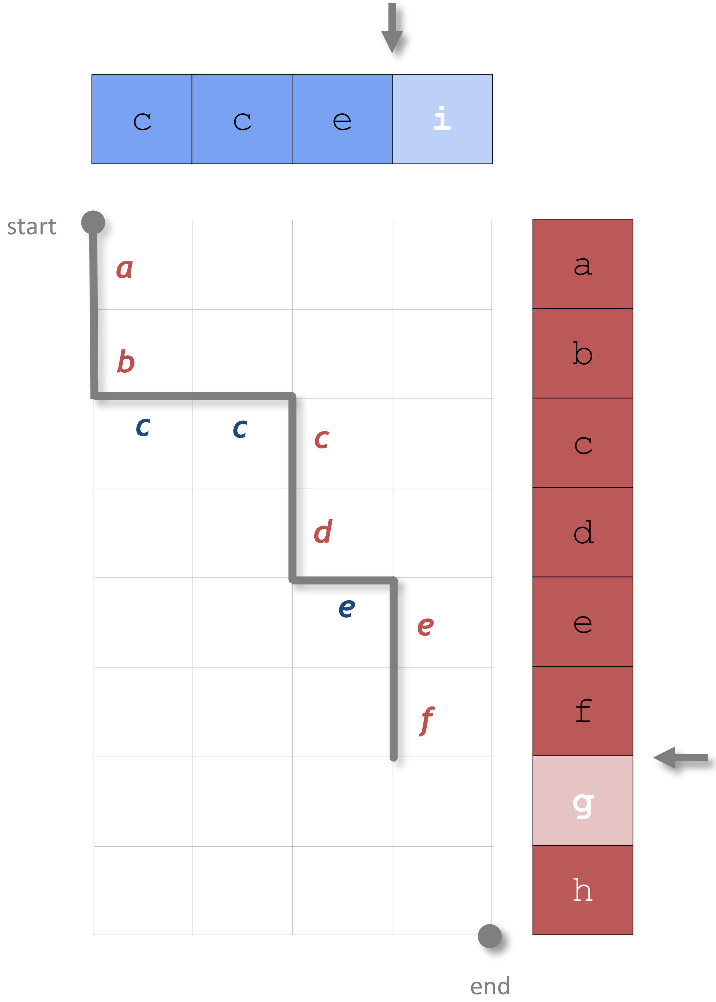
The 2D “merge-path” visualization

- The decision path runs from top-left to bottom-right:
    - Moves right when consuming from $list_A$
    - Moves down when consuming from $list_B$
    - Each step produces an output
    - Break ties by always preferring the element from $list_A$



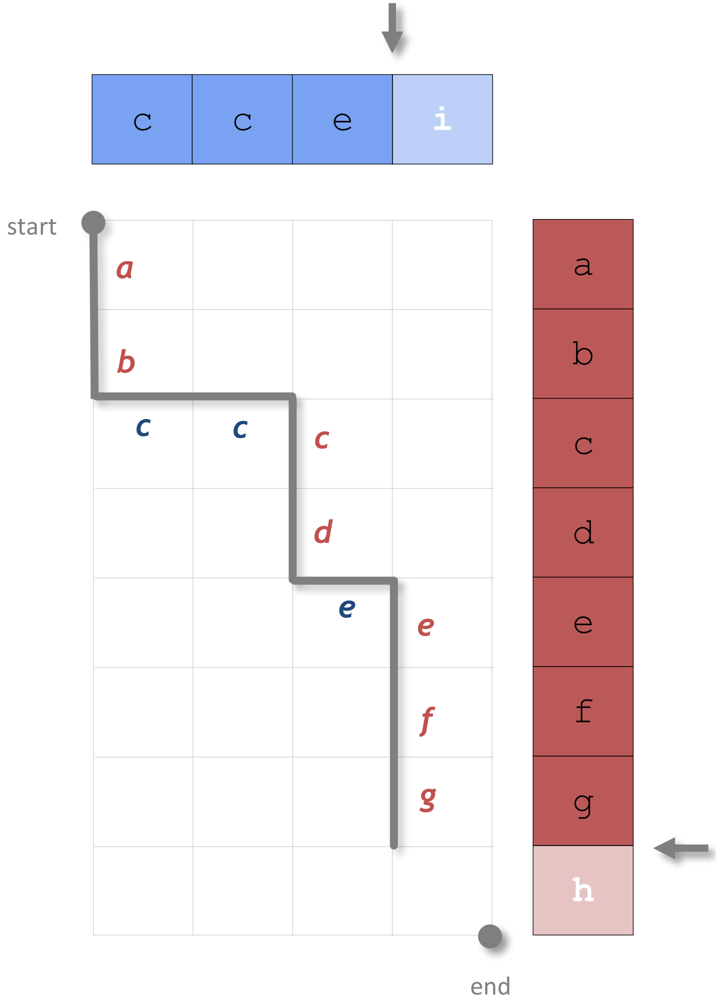
The 2D “merge-path” visualization

- The decision path runs from top-left to bottom-right:
    - Moves right when consuming from *$list_A$*
    - Moves down when consuming from *$list_B$*
    - Each step produces an output
    - Break ties by always preferring the element from *$list_A$*



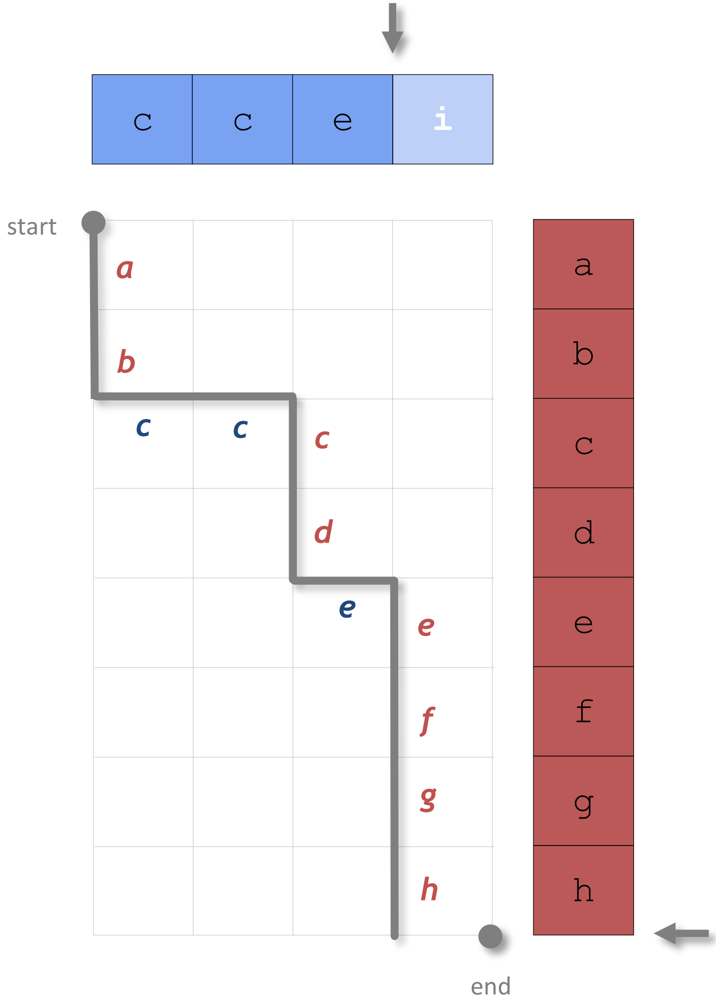
The 2D “merge-path” visualization

- The decision path runs from top-left to bottom-right:
  - Moves right when consuming from $list_A$
  - Moves down when consuming from $list_B$
  - Each step produces an output
  - Break ties by always preferring the element from $list_A$



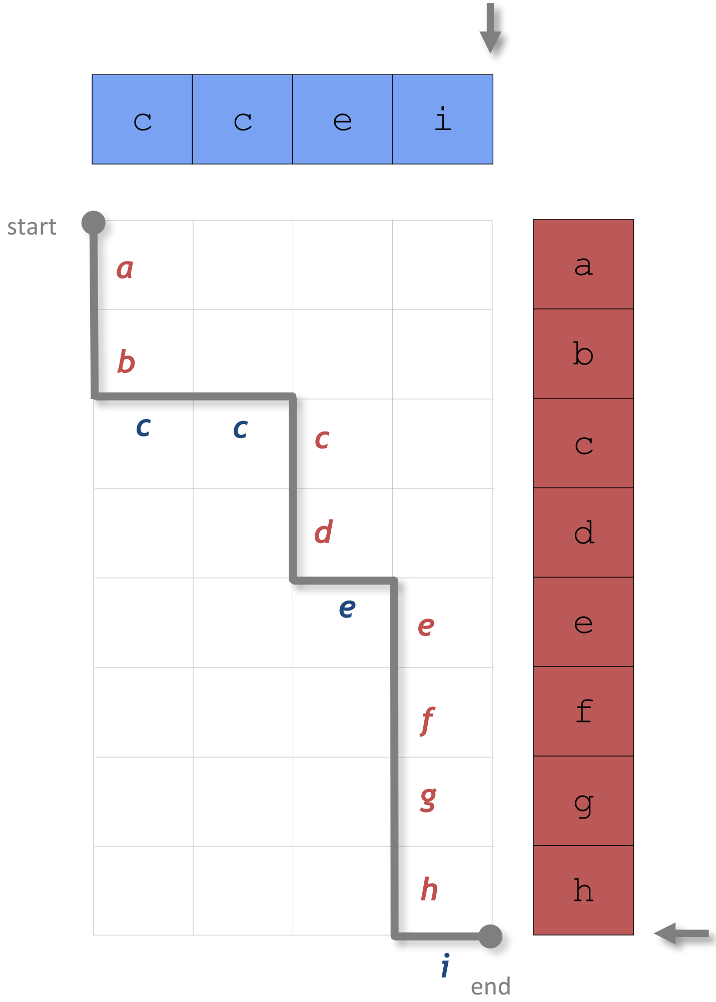
The 2D “merge-path” visualization

- The decision path runs from top-left to bottom-right:
  - Moves right when consuming from *$list_A$*
  - Moves down when consuming from *$list_B$*
  - Each step produces an output
  - Break ties by always preferring the element from *$list_A$*

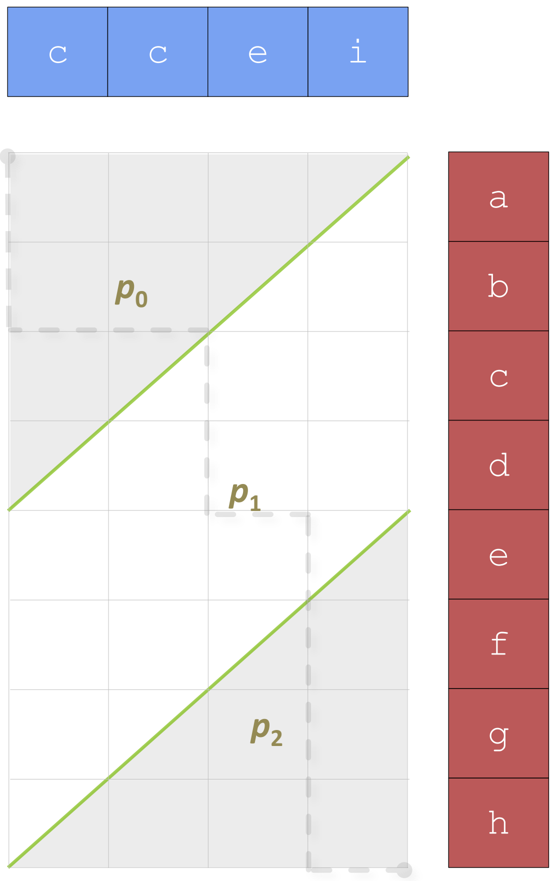


The 2D "merge-path" visualization

- The decision path runs from top-left to bottom-right:
  - Moves right when consuming from *list*$_A$
  - Moves down when consuming from *list*$_B$
  - Each step produces an output
  - Break ties by always preferring the element from *list*$_A$



Partitioning the "merge path"



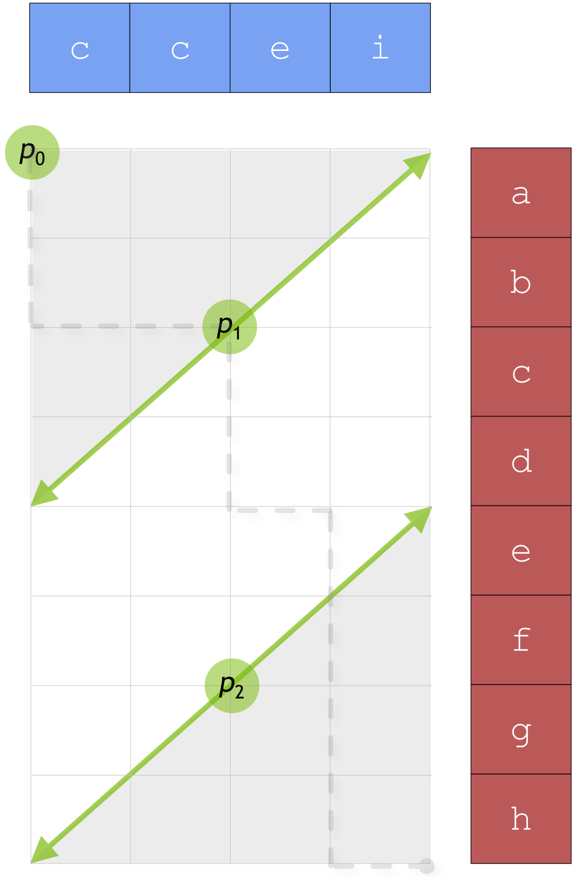
1. Partition the grid into *P* equally-sized diagonal regions (one thread per region)

2. Threads search along diagonals for 2D starting coordinates

   - I.e., Find the first `(i,j)` where X_i is greater than all of the items consumed before Y_j

3. Threads run the serial merge algorithm from their starting points

Partitioning the “merge path”



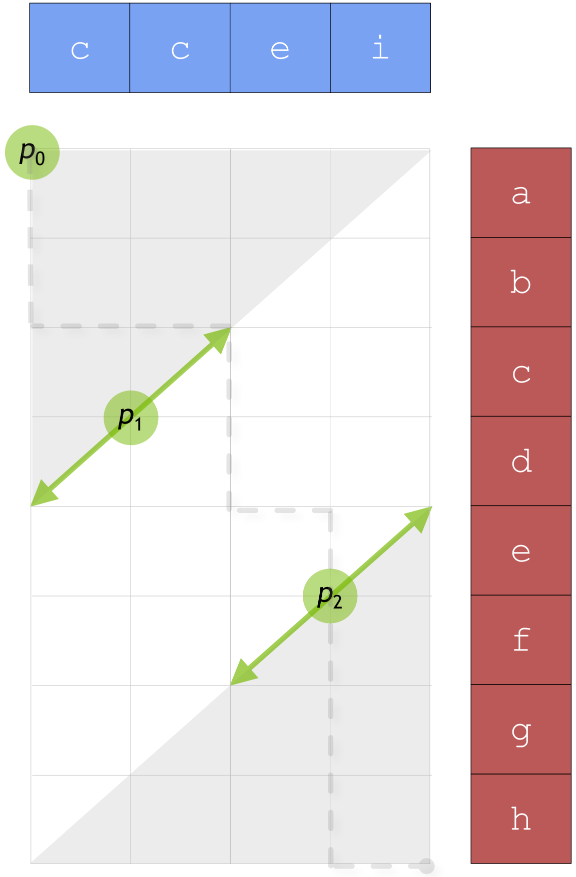
1. Partition the grid into *P* equally-sized diagonal regions (one thread per region)

2. Threads search along diagonals for 2D starting coordinates

   - I.e., Find the first `(i,j)` where X_i is greater than all of the items consumed before Y_j

3. Threads run the serial merge algorithm from their starting points

Partitioning the "merge path"

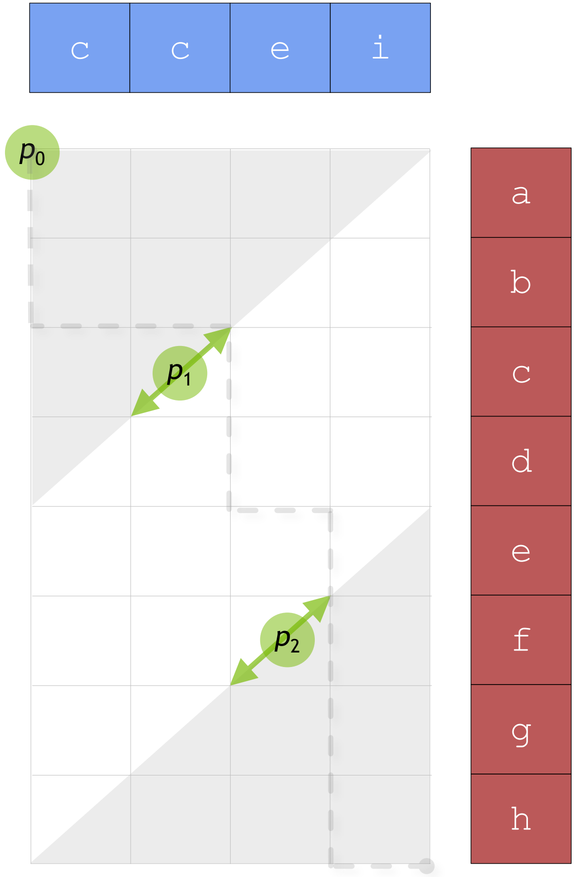


1. Partition the grid into *P* equally-sized diagonal regions (one thread per region)

2. Threads search along diagonals for 2D starting coordinates
   - I.e., Find the first `(i,j)` where X_i is greater than all of the items consumed before Y_j

3. Threads run the serial merge algorithm from their starting points

Partitioning the "merge path"

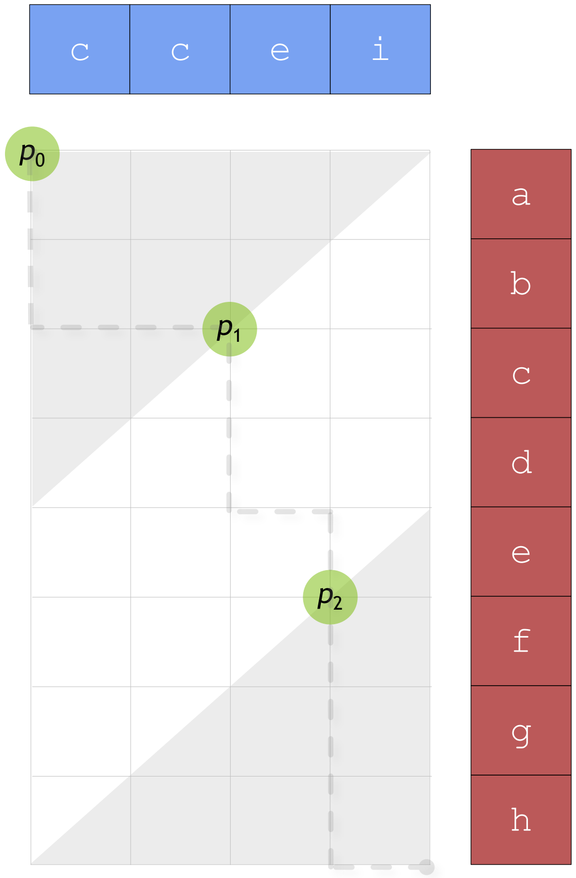


1. Partition the grid into P equally-sized diagonal regions (one thread per region)

2. Threads search along diagonals for 2D starting coordinates
   - I.e., Find the first `(i,j)` where X_i is greater than all of the items consumed before Y_j

3. Threads run the serial merge algorithm from their starting points

Partitioning the "merge path"

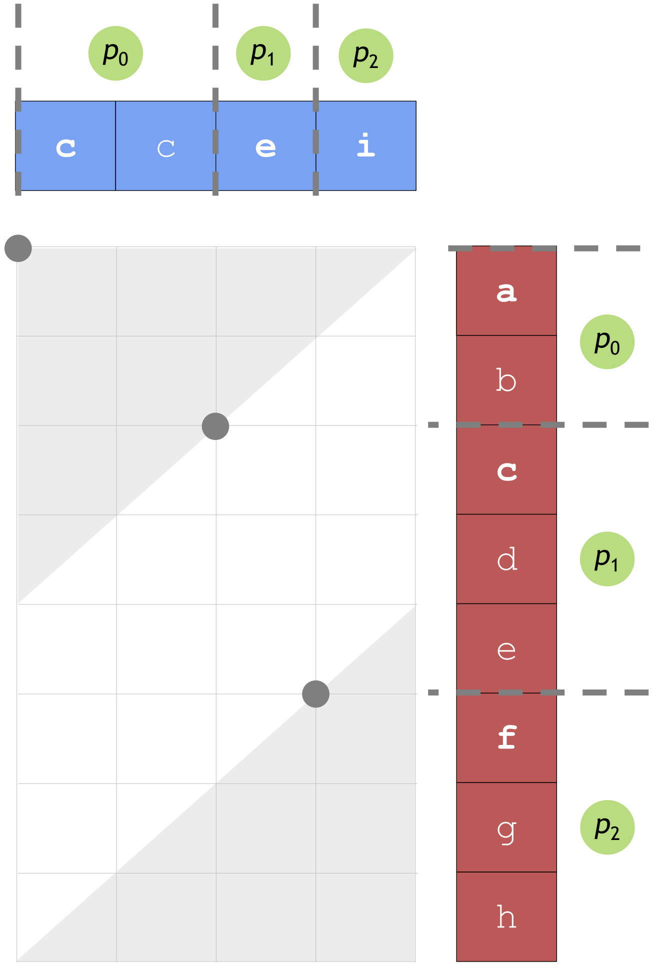


1. Partition the grid into *P* equally-sized diagonal regions (one thread per region)

2. Threads search along diagonals for 2D starting coordinates
   - I.e., Find the first `(i,j)` where X_i is greater than all of the items consumed before Y_j

3. Threads run the serial merge algorithm from their starting points

Partitioning the "merge path"



1. Partition the grid into P equally-sized diagonal regions (one thread per region)

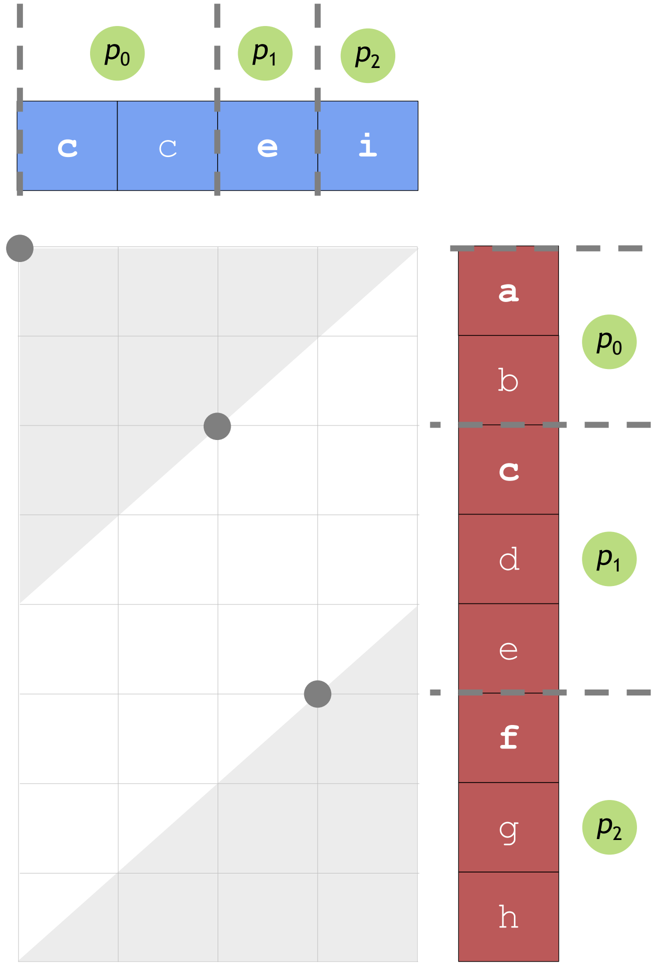
2. Threads search along diagonals for 2D starting coordinates

   - I.e., Find the first `(i,j)` where X_i is greater than all of the items consumed before Y_j

3. Threads run the serial merge algorithm from their starting points

   - Work is perfectly load-balanced!

Consuming the "merge path"



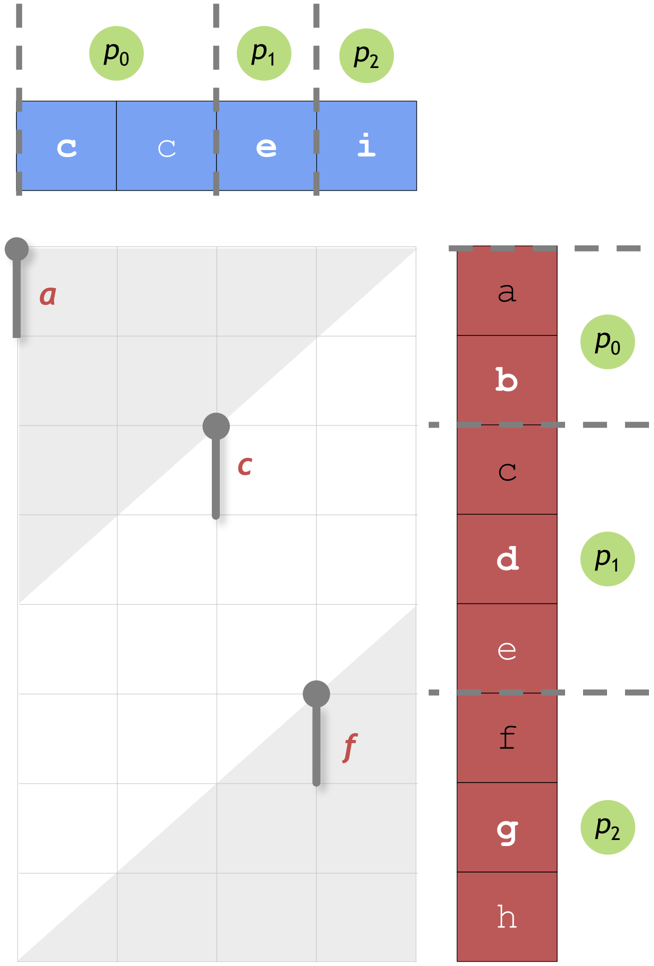
1. Partition the grid into P equally-sized diagonal regions (one thread per region)

2. Threads search along diagonals for 2D starting coordinates

   - I.e., Find the first `(i,j)` where X_i is greater than all of the items consumed before Y_j

3. Threads run the serial merge algorithm from their starting points

Consuming the "merge path"



1. Partition the grid into P equally-sized diagonal regions (one thread per region)

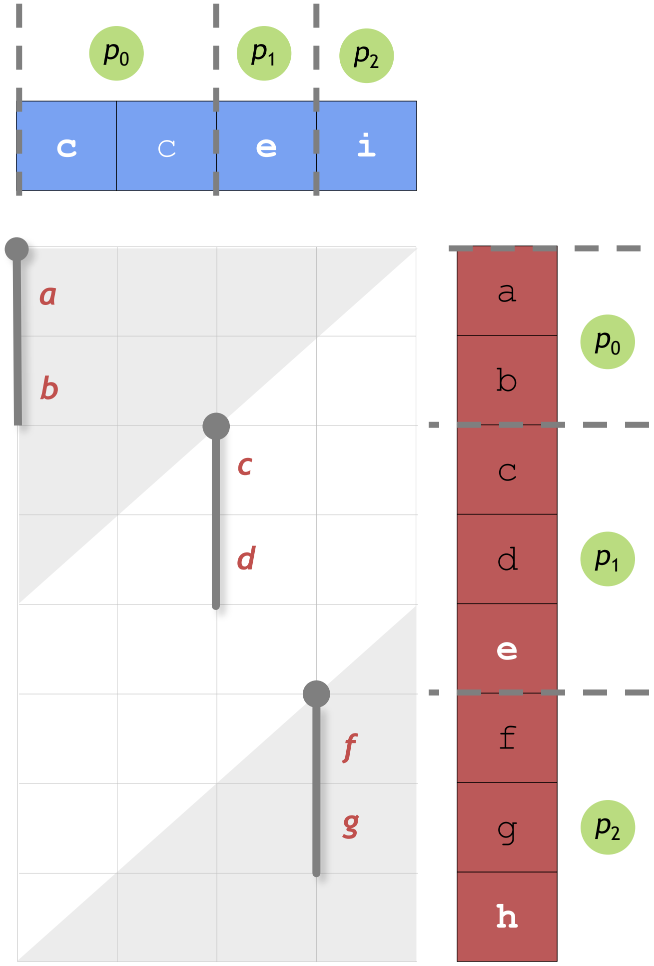
2. Threads search along diagonals for 2D starting coordinates

   - I.e., Find the first `(i,j)` where X_i is greater than all of the items consumed before Y_j

3. Threads run the serial merge algorithm from their starting points

   - Work is perfectly load-balanced!

Consuming the "merge path"



1. Partition the grid into P equally-sized diagonal regions (one thread per region)

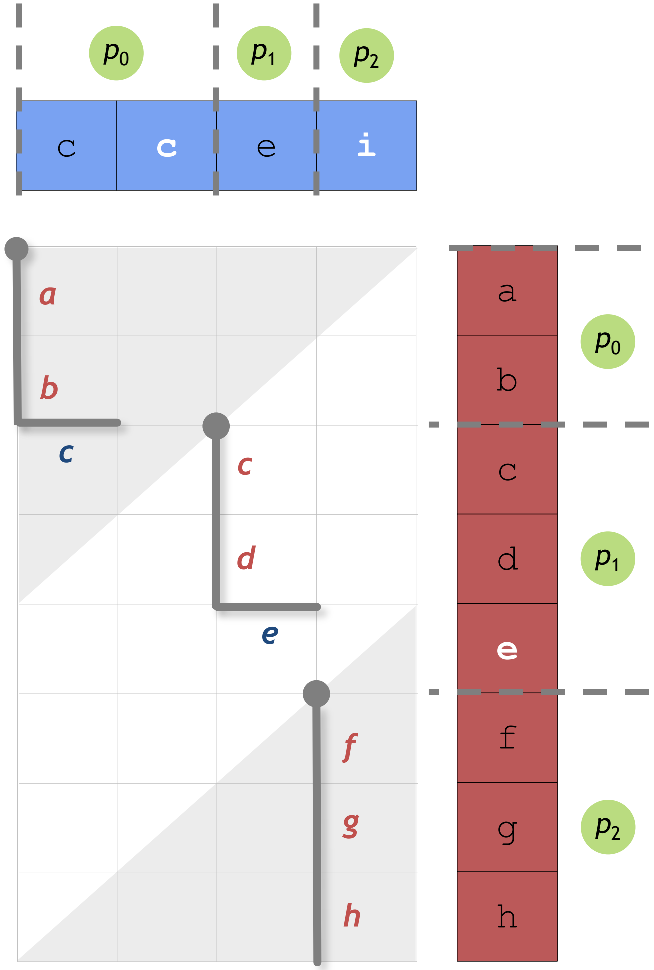
2. Threads search along diagonals for 2D starting coordinates

   - I.e., Find the first `(i,j)` where X_i is greater than all of the items consumed before Y_j

3. Threads run the serial merge algorithm from their starting points

   - Work is perfectly load-balanced!

Consuming the "merge path"



1. Partition the grid into P equally-sized diagonal regions (one thread per region)

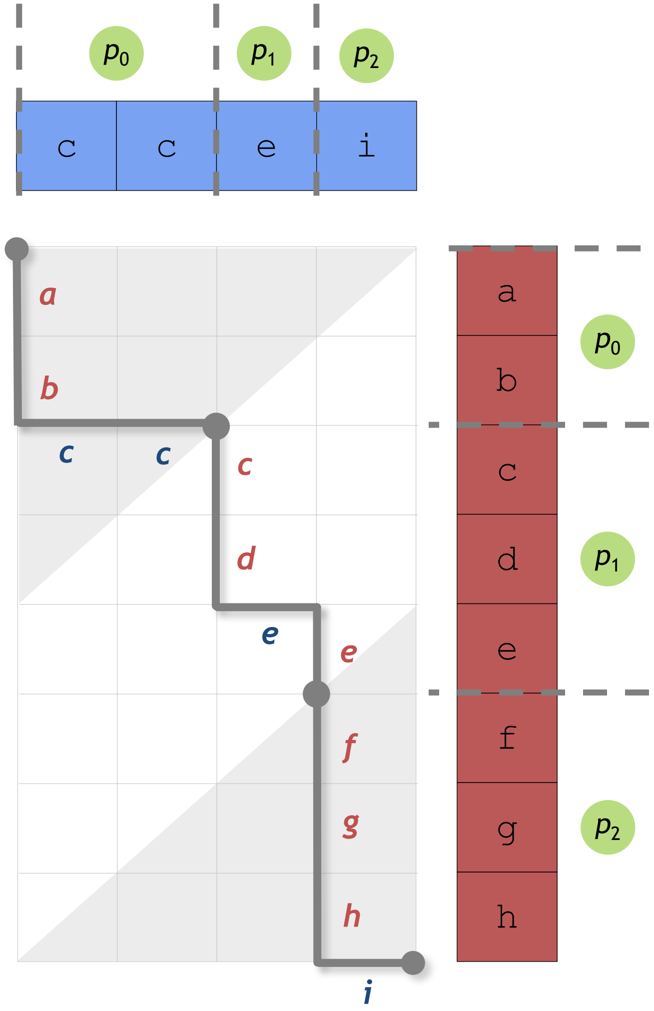
2. Threads search along diagonals for 2D starting coordinates

   - I.e., Find the first `(i,j)` where X_i is greater than all of the items consumed before Y_j

3. Threads run the serial merge algorithm from their starting points

   - Work is perfectly load-balanced!

Consuming the "merge path"



1. Partition the grid into *P* equally-sized diagonal regions (one thread per region)

2. Threads search along diagonals for 2D starting coordinates

   - I.e., Find the first `(i,j)` where X_i is greater than all of the items consumed before Y_j

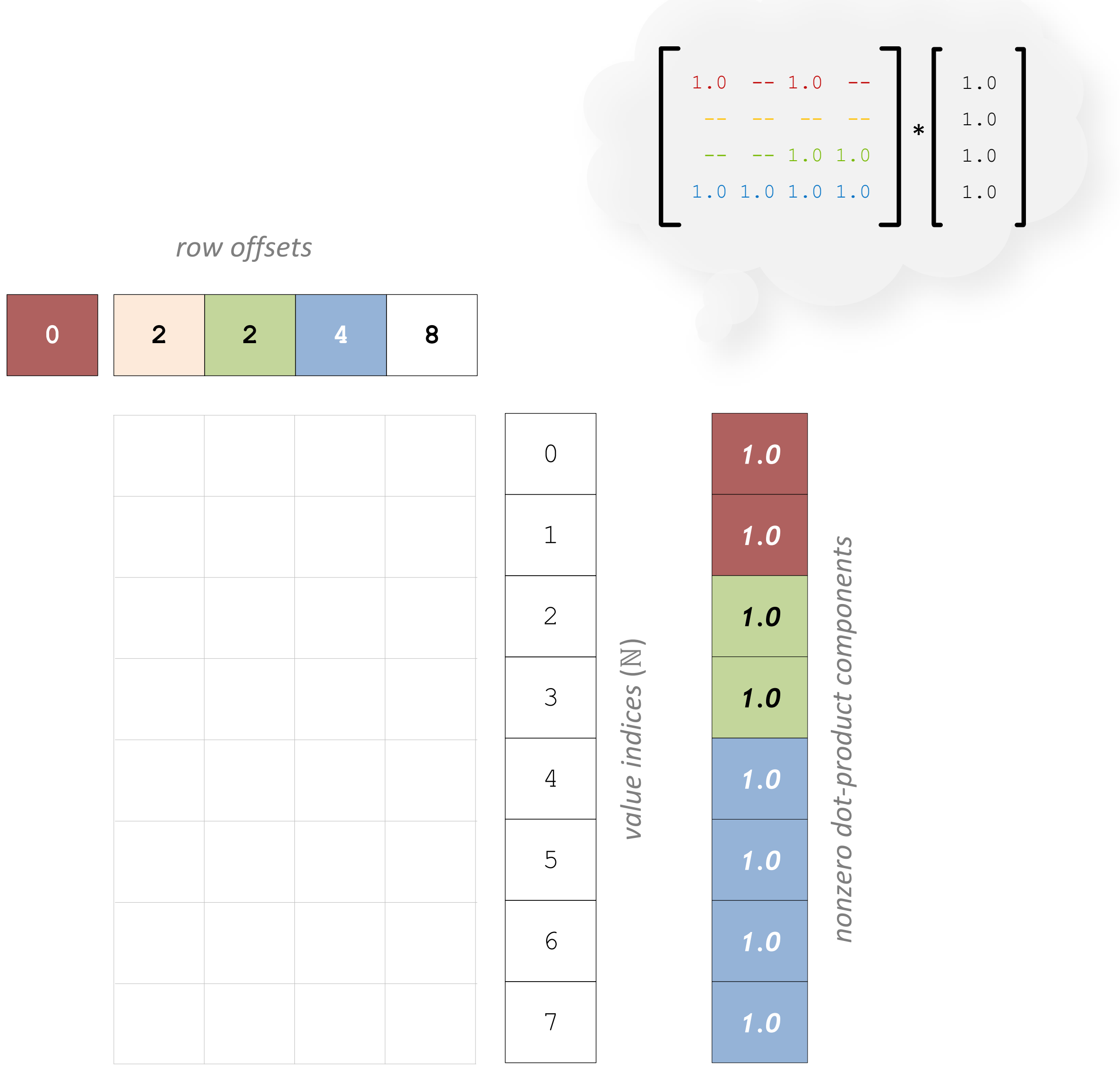
3. Threads run the serial merge algorithm from their starting points

   - Work is perfectly load-balanced!

MERGE-BASED CsrMV

MERGE-BASED CsrMV

1. Logically merge CSR *row-offsets* *vs.* ℕ (the nonzero indices)
2. Partition the path into *P* regions
3. Path processing:
   - Accumulate values when moving down
   - Flush and reset accumulator when right
4. "Fixup" for partial-sums from rows that cross partitions



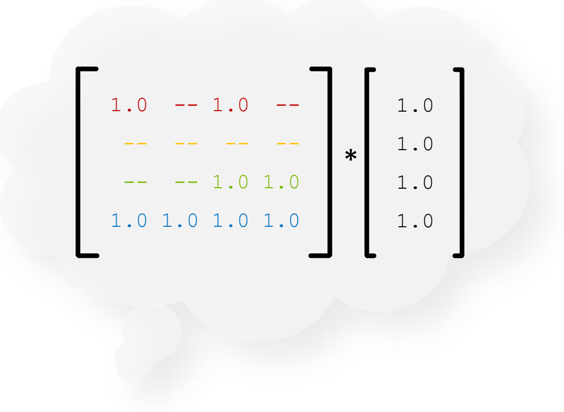
MERGE-BASED CsrMV

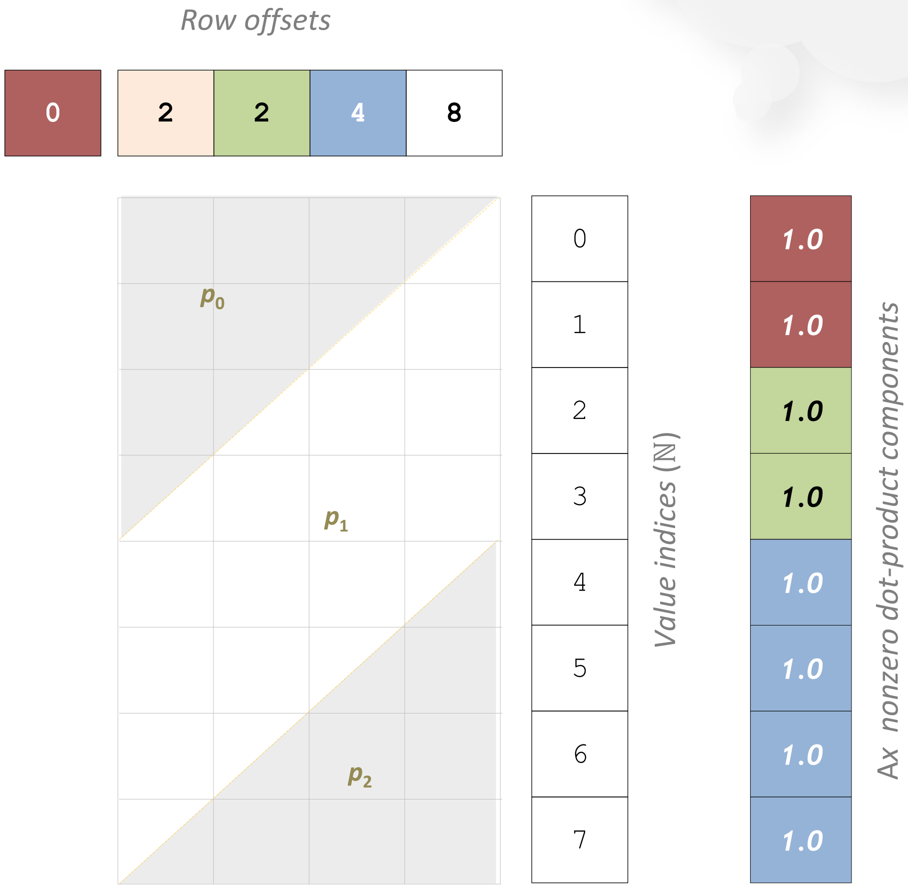
1. Logically merge *row-offsets vs.* ℕ (the nonzero indices)

2. Partition the path into *P* regions

3. Path processing:
   - Accumulate values when moving down
   - Flush and reset accumulator when right

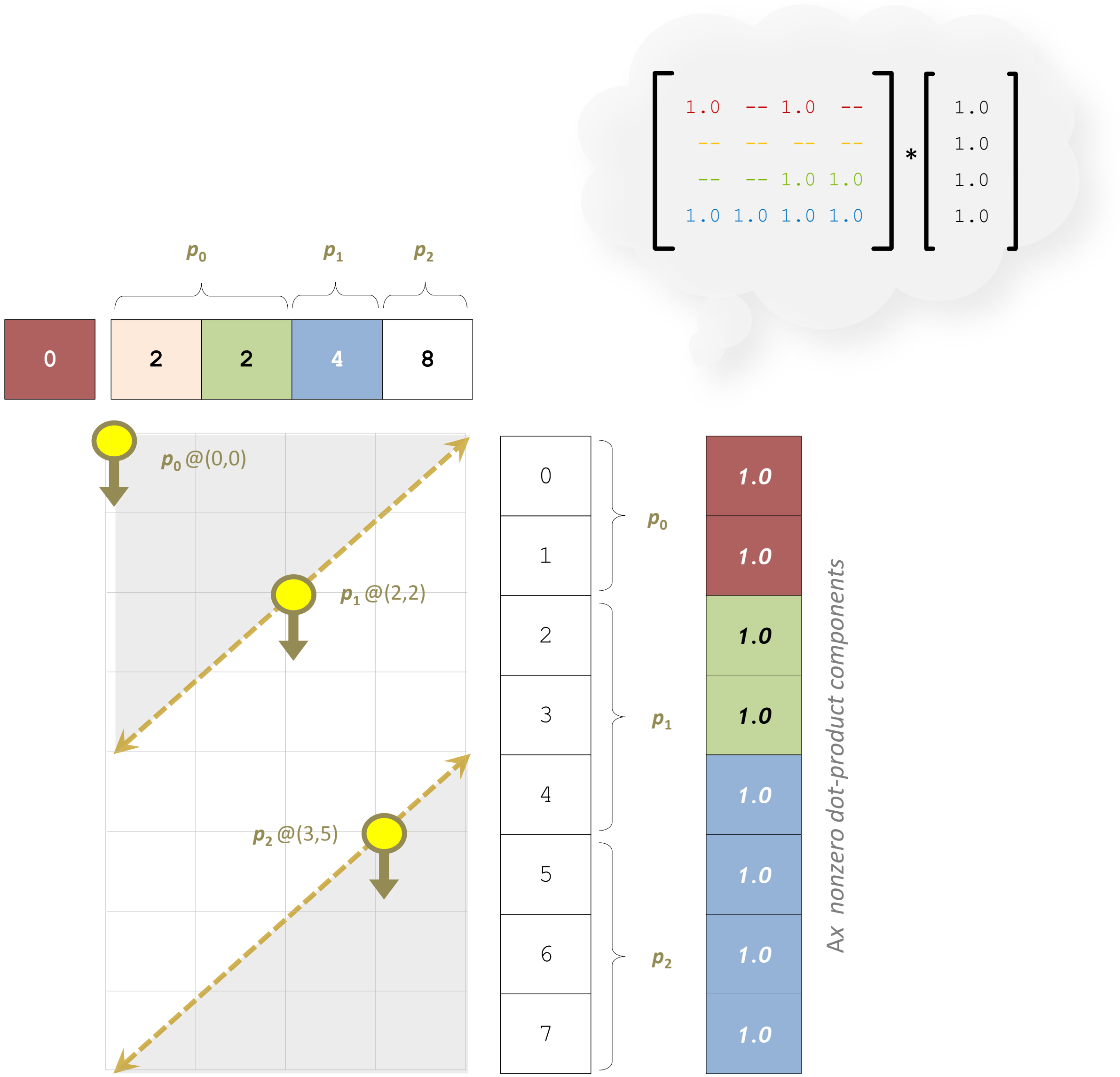
4. "Fixup" for partial-sums from rows that cross partitions





MERGE-BASED CsrMV

1. Logically merge *row-offsets vs.* ℕ (the nonzero indices)
2. Partition the path into *P* regions
3. Path processing:
   - Accumulate values when moving down
   - Flush and reset accumulator when right
4. "Fixup" for partial-sums from rows that cross partitions



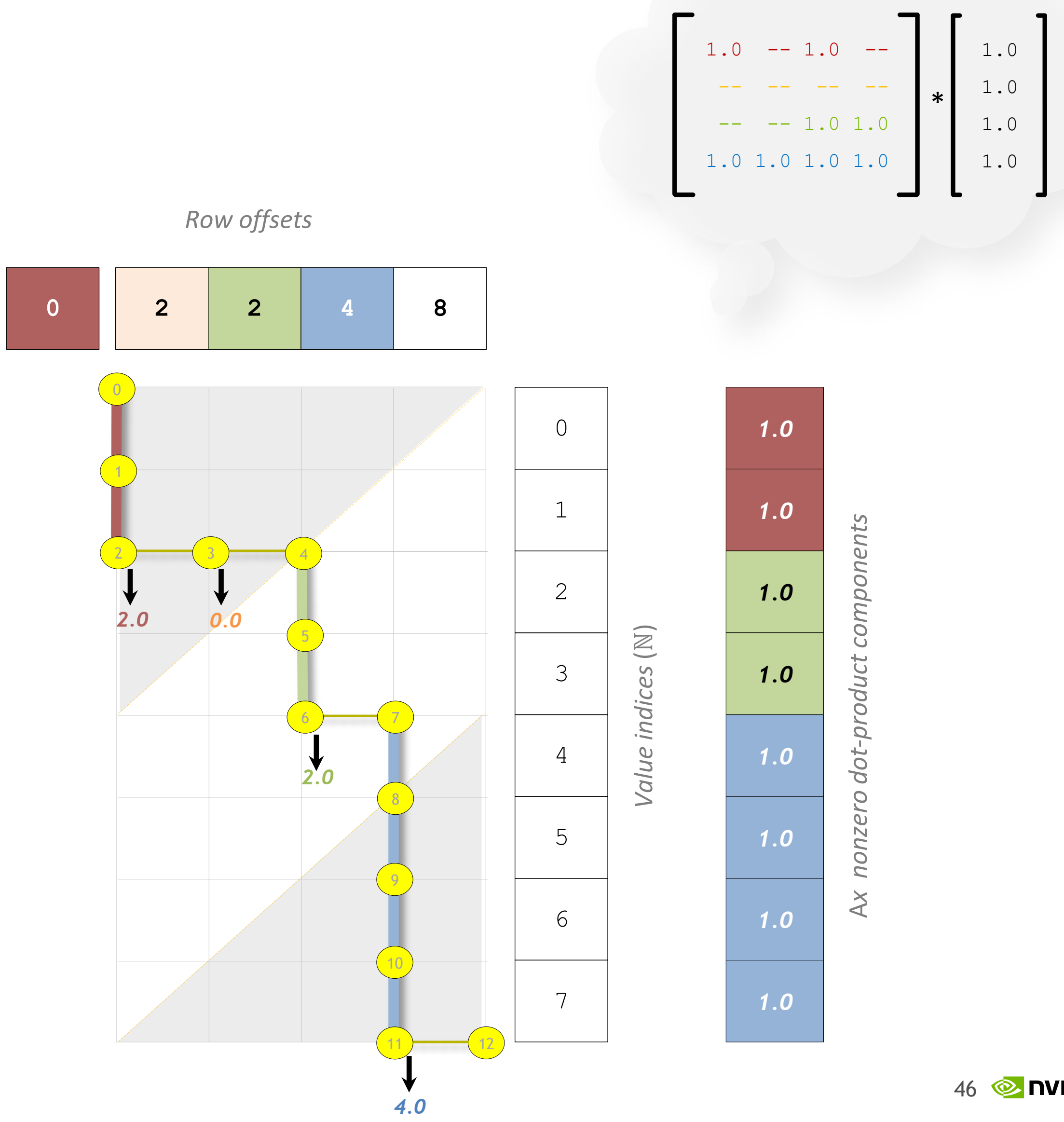
MERGE-BASED CsrMV

1. Logically merge *row-offsets vs.* ℕ (the nonzero indices)

2. Partition the path into *P* regions

3. Path processing:
   - Accumulate nonzero values when moving down
   - Flush and reset accumulator when right

4. "Fixup" for partial-sums from rows that cross partitions



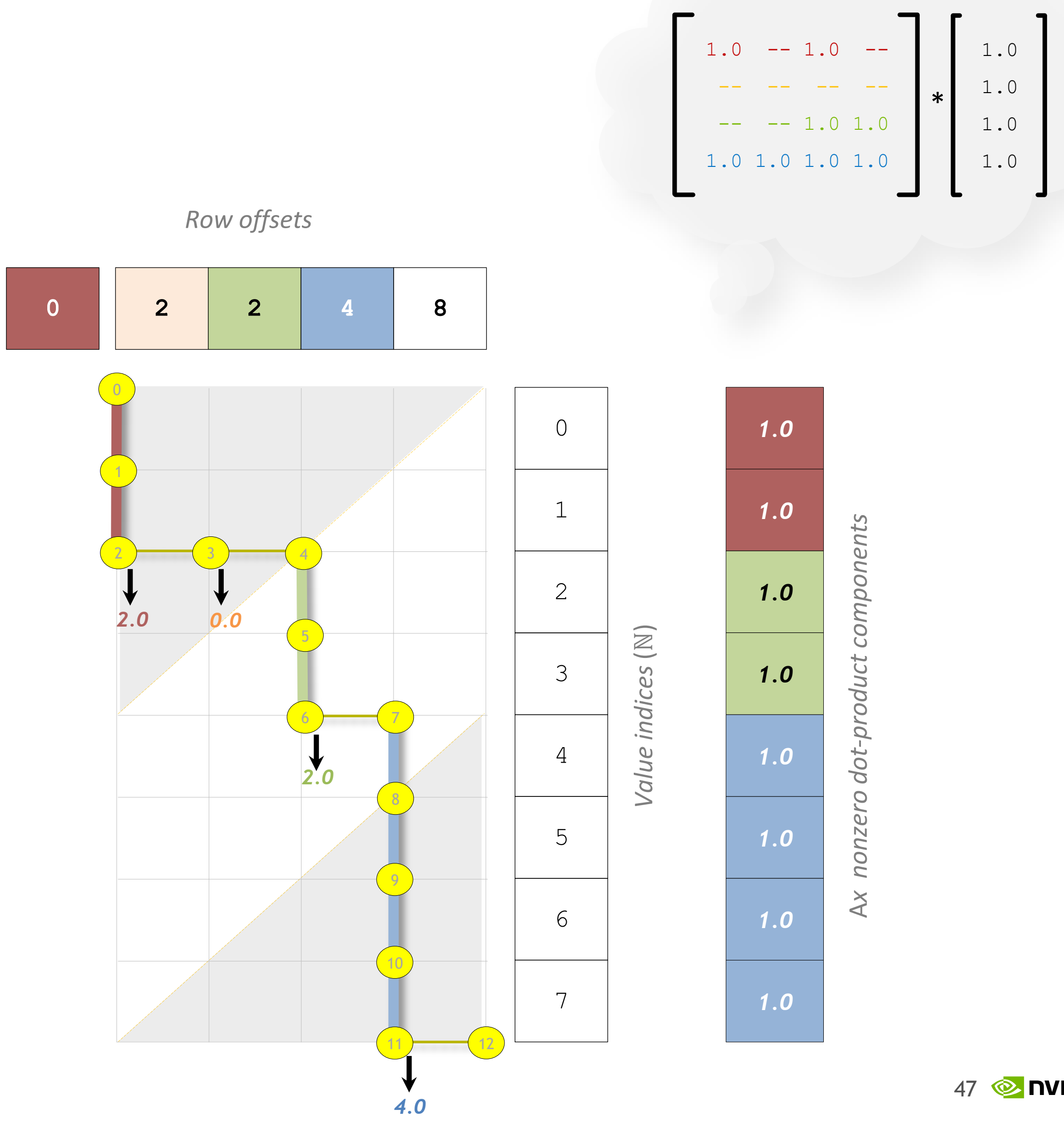
MERGE-BASED CsrMV

1. Logically merge *row-offsets vs.* ℕ (the nonzero indices)

2. Partition the path into *P* regions

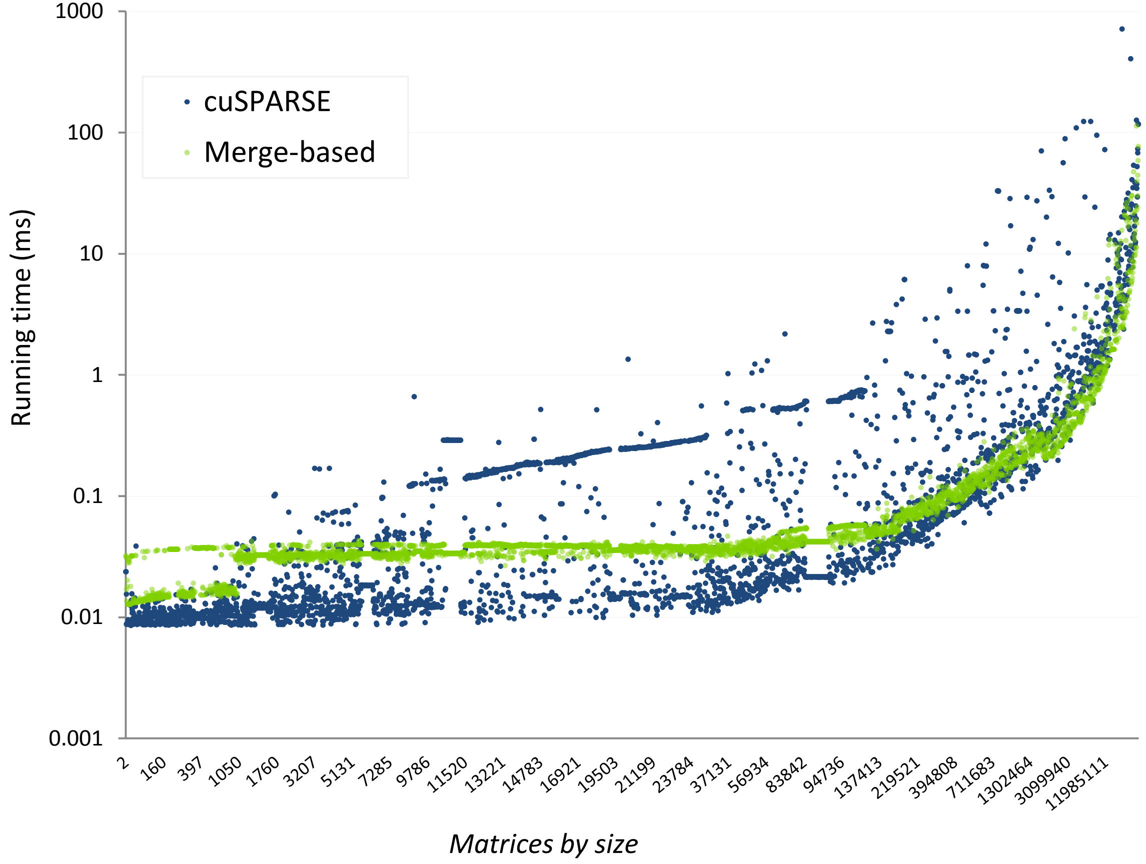
3. Path processing:
   - Accumulate values when moving down
   - Flush and reset accumulator when right

4. "Fixup" for partial-sums from rows that cross partitions



SPMV PERFORMANCE LANDSCAPE

The entire Florida Sparse Matrix Collection: 4.2K datasets (K40, fp64 CsrMV)



- Much higher correlation of runtime to problem size (0.79 versus 0.31)
- Much lower correlation of FLOPS to row-length variation (-0.02 versus -0.24)
- Much lower correlation of FLOPS to row-length skew (-0.07 versus -0.23)

QUESTIONS?

Merrill, D. and Garland, M. 2015. *Merge-based Parallel Sparse Matrix-Vector Multiplication using the CSR Storage Format*. Tech. Rep. NVR-2015-002, NVIDIA Corp.

- Further thanks and appreciation for support from DARPA PERFECT,

  Sean Baxter, Michael Garland

NVIDIA

Los Alamos
NATIONAL LABORATORY
EST. 1943