Sparse Computations on GPUs

Changwan Hong

04/25/24

* Disclaimer: The methods described here are simplified/modified for easier understanding and are not highly accurate.

Recap on GPU Computing

Properties of GPUs:

- Many lightweight cores (high degree of parallelism)
- Extremely high computing power (FLOP/s)
- SIMD* execution mode (all threads of a warp follow the same execution path)
- High memory bandwidth
- No sophisticated memory hierarchy, small caches

Why?

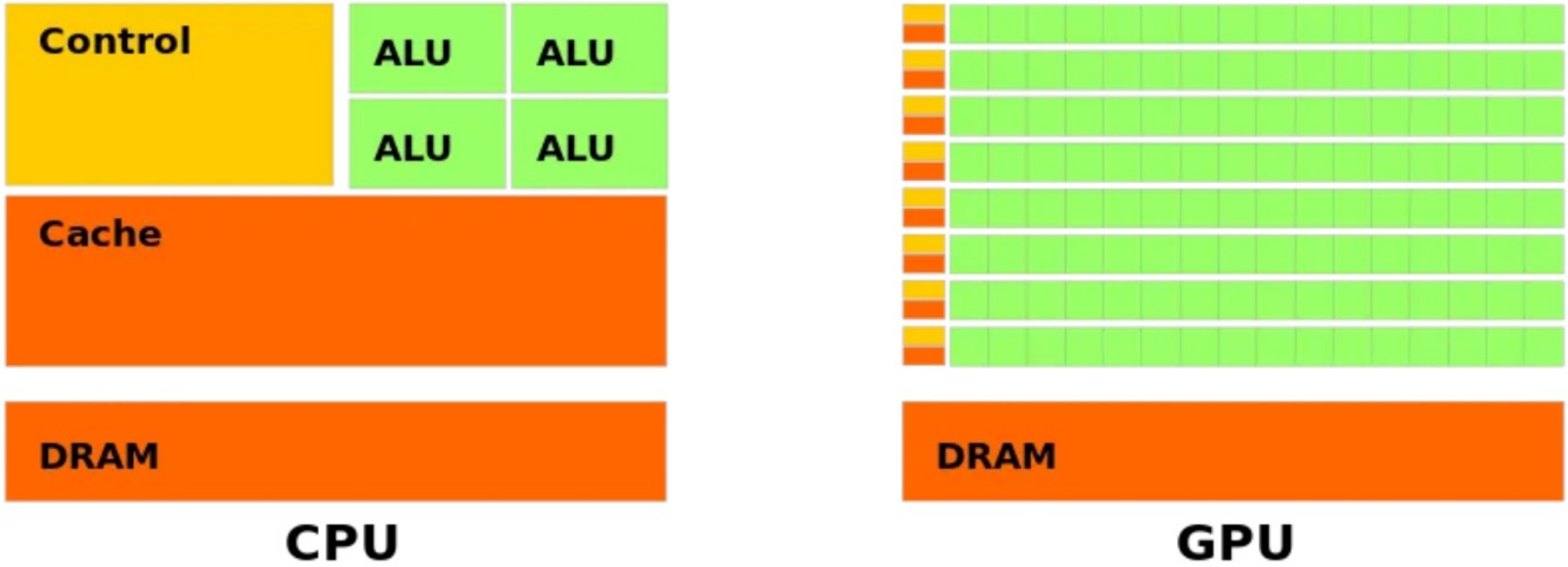
- Originally designed for graphics use
- No dependencies when updating pixels on a screen
- Many pixels need to be updated at the same time

How is the data accessed for general purpose computing?

- On CPUs: sophisticated memory hierarchy (L1, L2, L3…)
- On GPU: fast switching between contexts to hide latencies: execute operation on data that is present

* https://icl.utk.edu/~hanzt/talks/SparseMatricesAndParallelProcessingOnGPUs.pdf

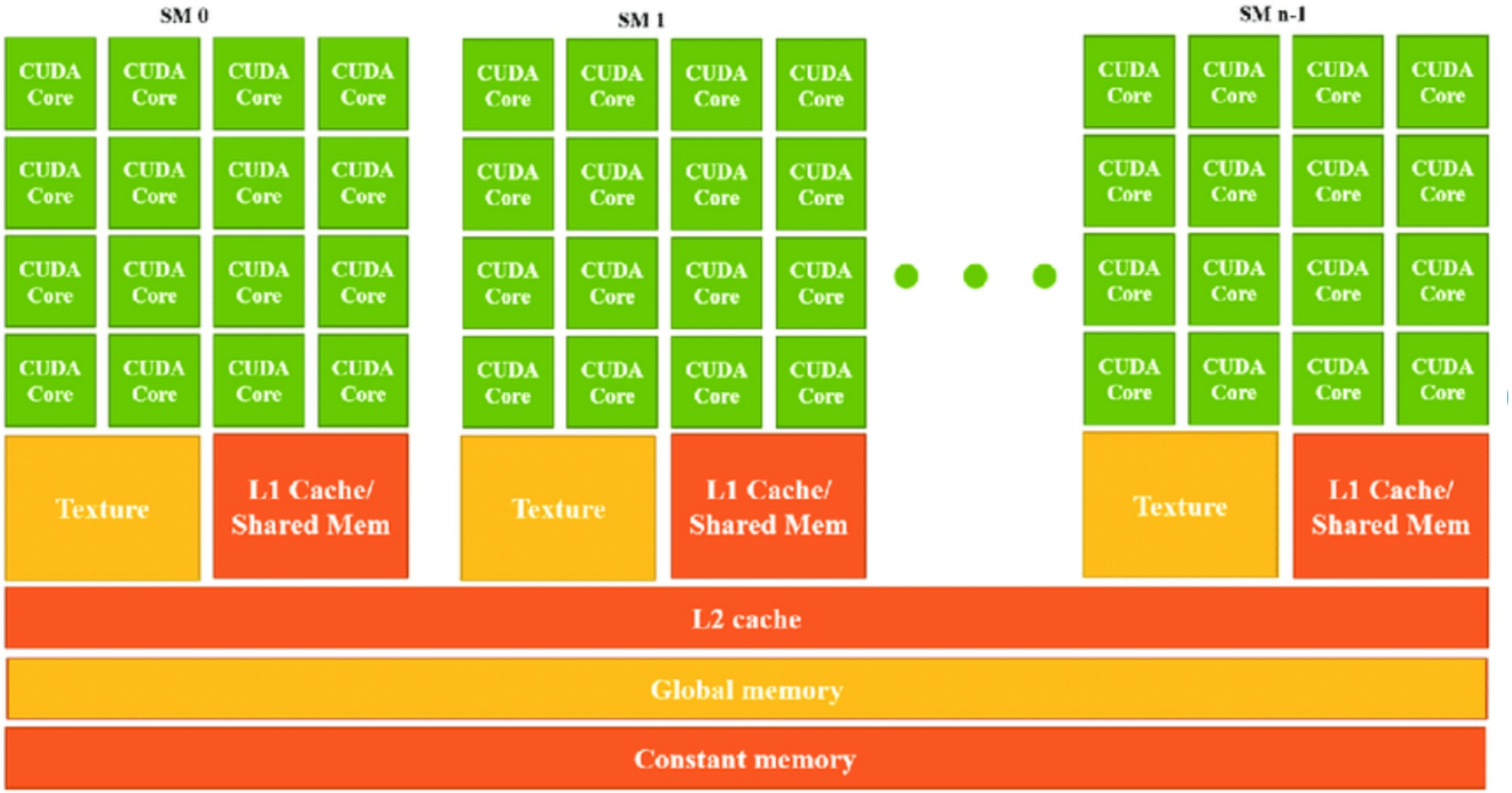
GPU Component



Left: CPU architecture; right: GPU architecture. Source: https://www.omnisci.com/technical-glossary/cpu-vs-gpu.

* https://medium.com/codex/understanding-the-architecture-of-a-gpu-d5d2d2e8978b

GPU Structure



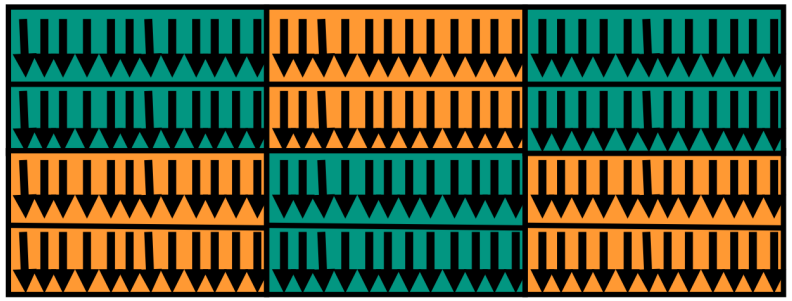
* https://www.researchgate.net/figure/Schematic-of-NVIDIA-GPU-architecture-where-SM-refers-to-streaming-multiprocessor_fig2_321958738

Programming GPUs

Fundamentally different from programming general purpose CPUs!

- On **GPU**s, we create a huge amount of independent **threads**.
- We organize the threads in **thread blocks** arranged as a **grid**.

threads

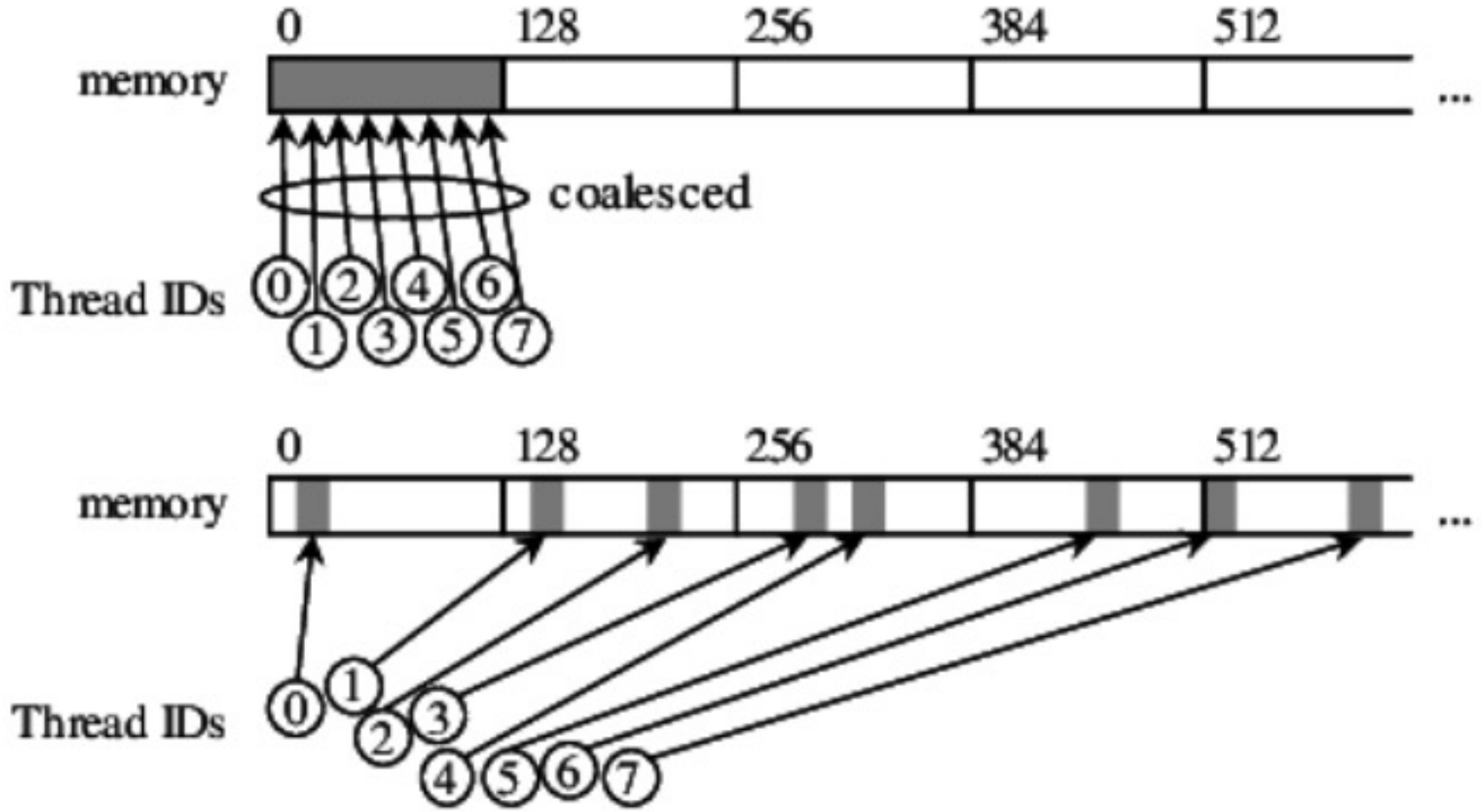


```
__global__ void kernel(int* a)
{
   a[blockIdx.x*blockDim.x +
threadIdx.x]=0;
}

int main() {
  . . .
  dim3 block(4);
  dim3 grid(n/block.x);
  kernel<<<grid,block>>>(d_a);
  . . .
  return 0;

}
```

- One warp consists of 32 threads on Nvidia GPUs.
  - You can think of the threads in a warp as being executed in single-instruction multiple-thread (SIMT) fashion.
- One thread block consists of a few warps, which users can configure."

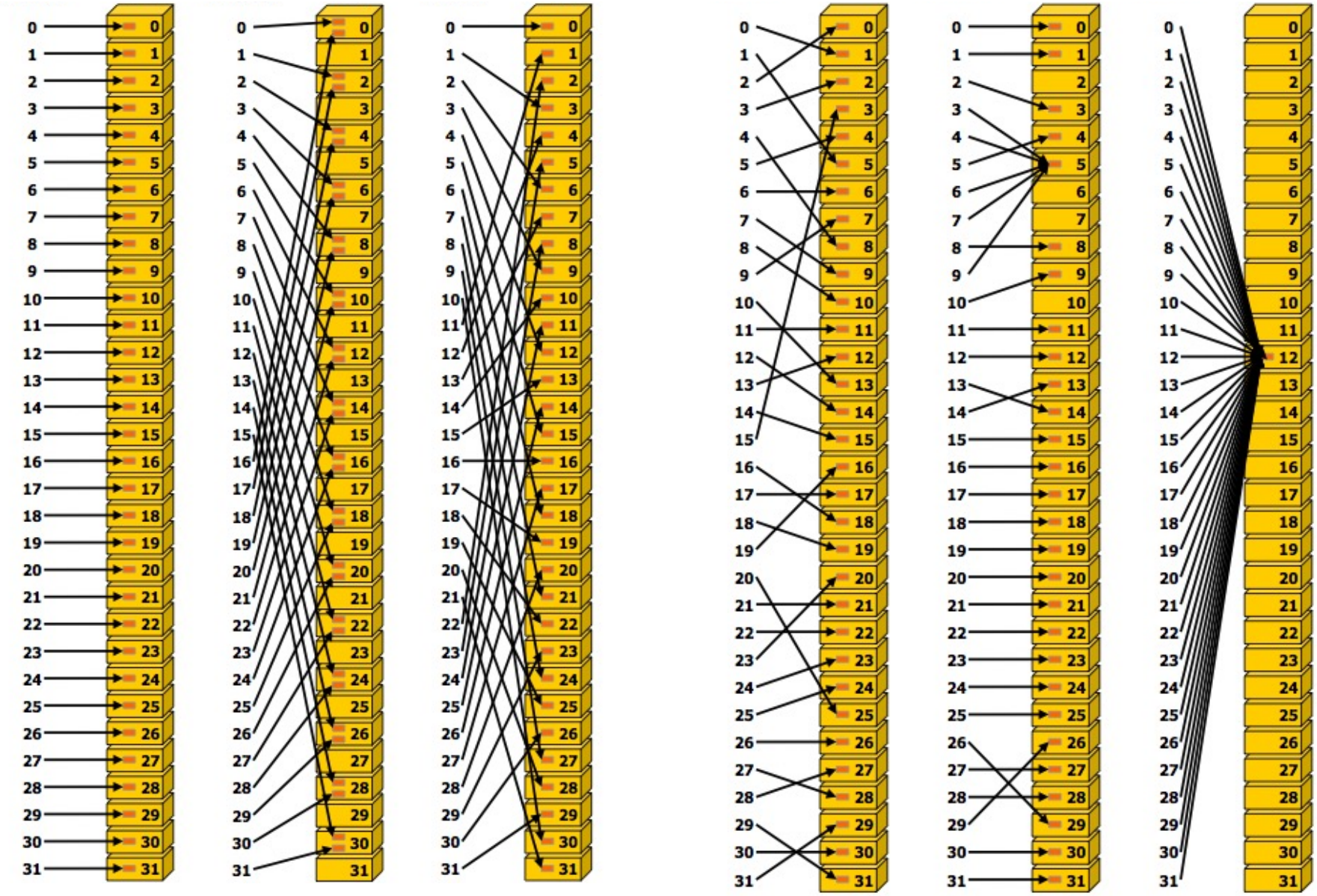
* https://icl.utk.edu/~hanzt/talks/SparseMatricesAndParallelProcessingOnGPUs.pdf

GPU Memory coalescing



* https://www.researchgate.net/figure/Memory-coalescing-fast-access-and-not-coalesced-slow-access-representation_fig2_286446838

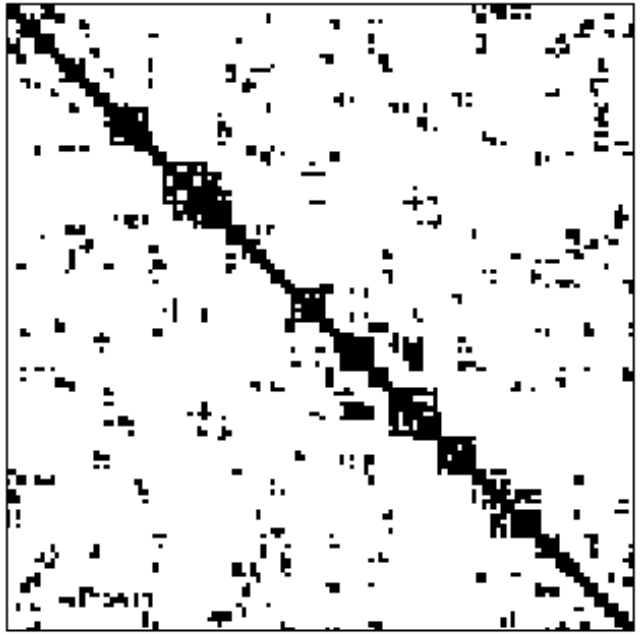
GPU Memory coalescing



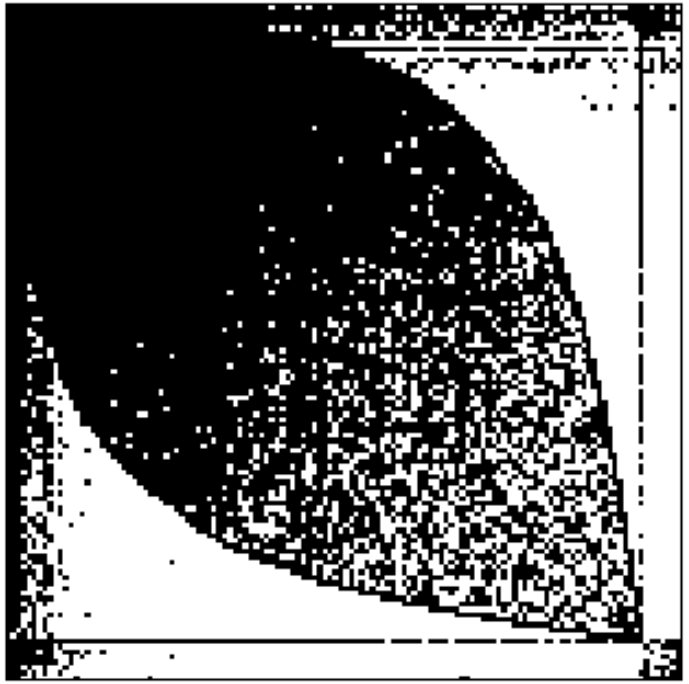
* http://homepages.math.uic.edu/~jan/mcs572f16/mcs572notes/lec35.html

Sparse matrix

- A matrix in which the majority of its elements are zero.
- Many real-world matrices exhibit sparsity.
- We need a compact representation that only stores the non-zero elements and their indices.
  - The asymptotic time and space complexity for sparse matrix computations can be orders of magnitude worse otherwise (i.e., infeasible to compute and store).



roadNet-CA
(Road network of California)

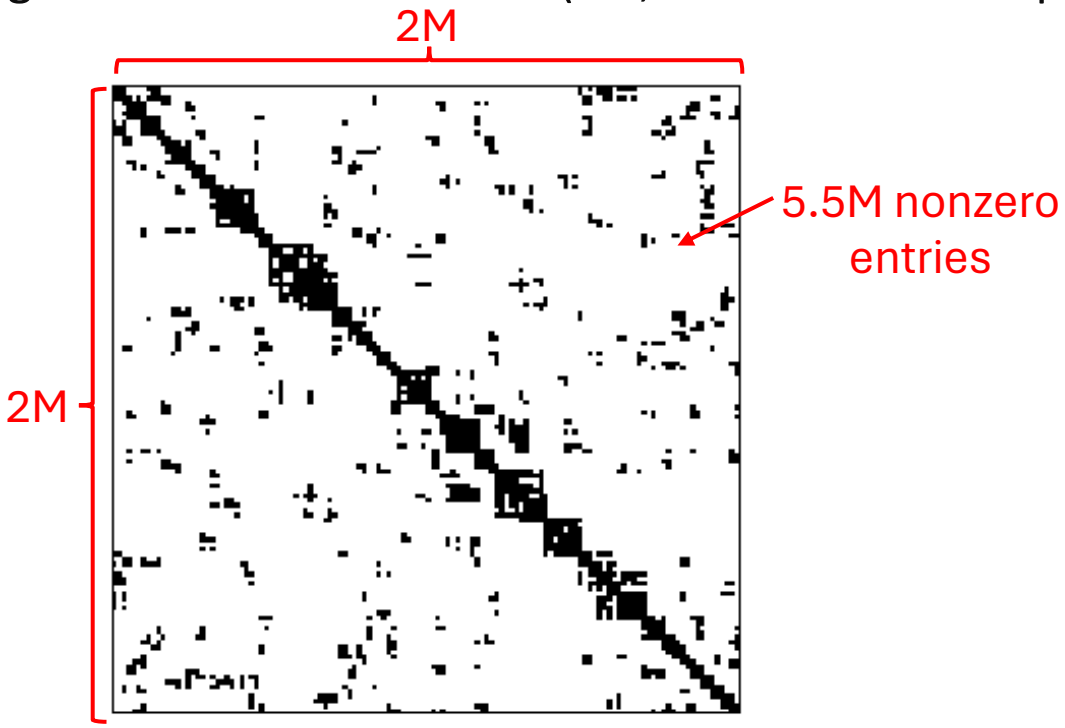


Soc-Epinions1
(Who-trusts-whom network of Epinions.com)

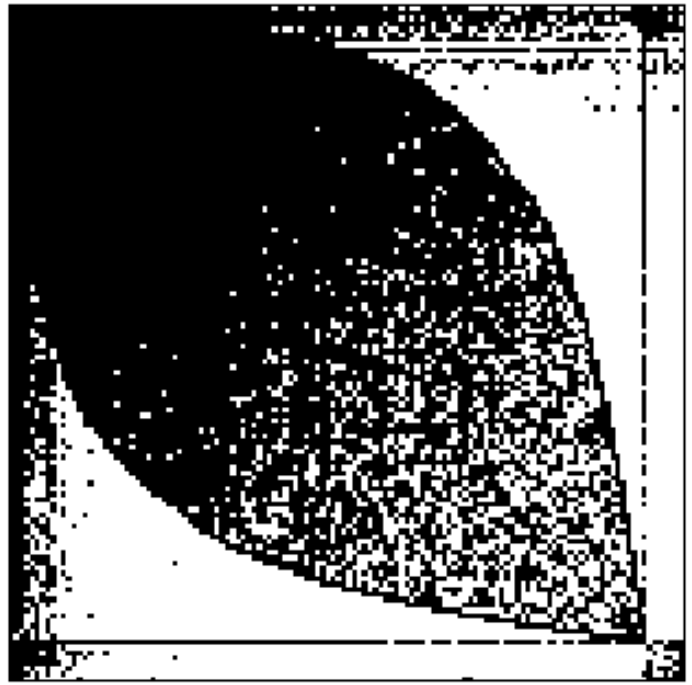
* http://sparse.tamu.edu

Sparse matrix

- A matrix in which the majority of its elements are zero.
- Many real-world matrices exhibit sparsity.
- We need a compact representation that only stores the non-zero elements and their indices.
  - The asymptotic time and space complexity for sparse matrix computations can be orders of magnitude worse otherwise (i.e., infeasible to compute and store).



roadNet-CA
(Road network of California)

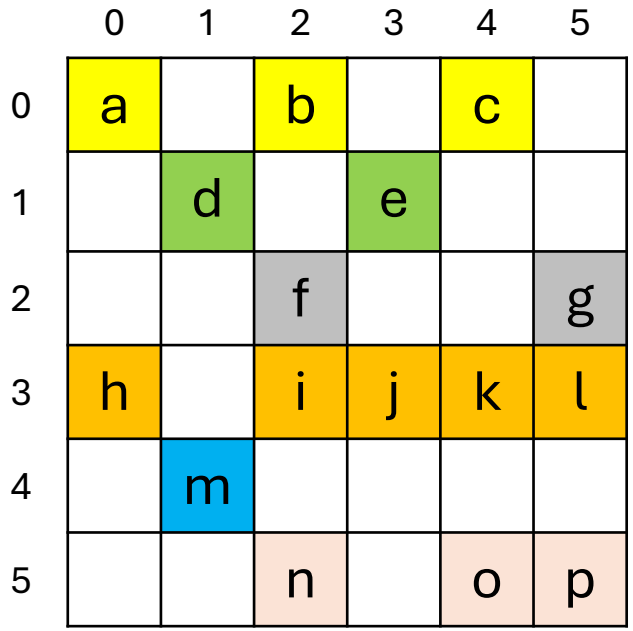


Soc-Epinions1
(Who-trusts-whom network of Epinions.com)

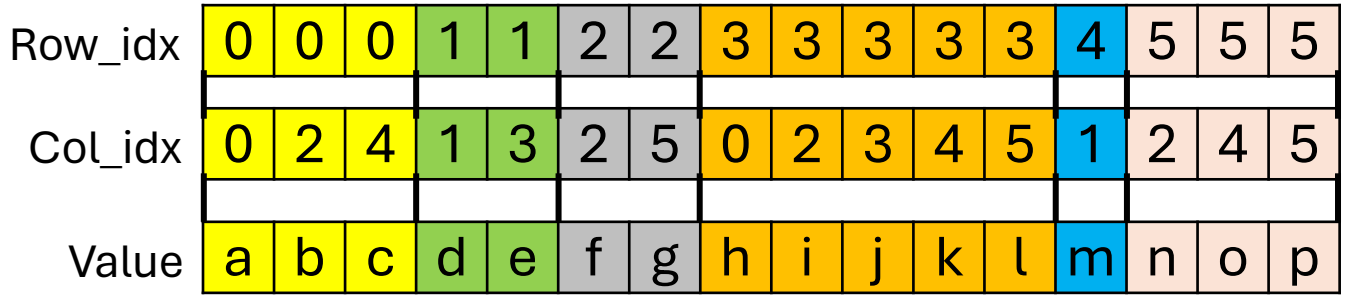
* http://sparse.tamu.edu

Compressed Sparse Row (CSR)

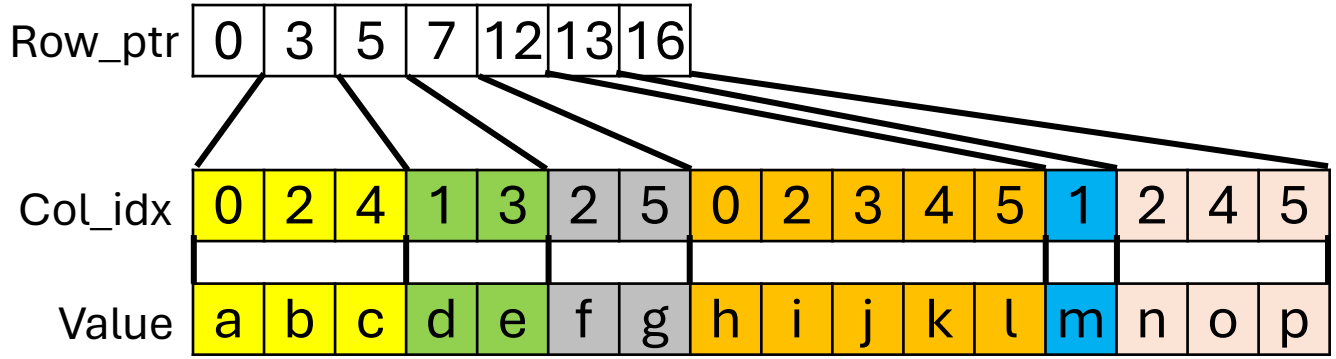
- Compressed Sparse Row (CSR) and Coordinate format (COO) are the most popular data representations for sparse matrices.
- The row pointers indicate where each row starts in the other two arrays
- Column indices of each row is usually sorted in ascending order.



Sparse Matrix

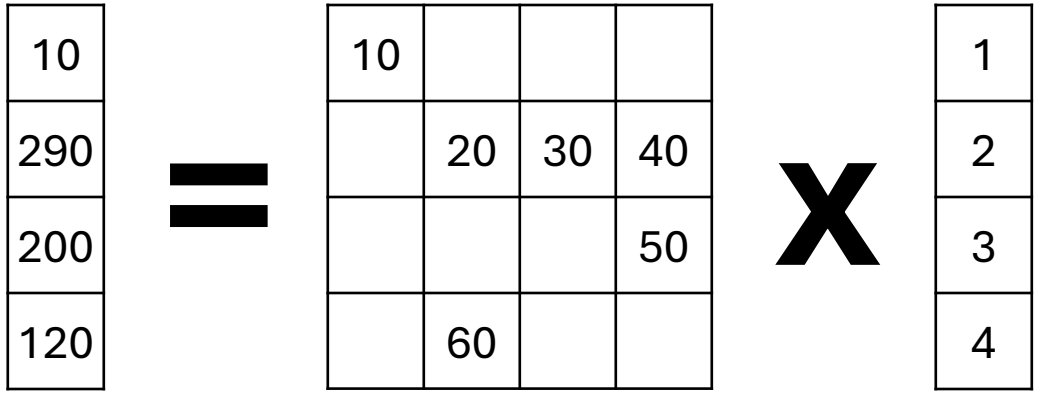


Corresponding COO



Corresponding CSR

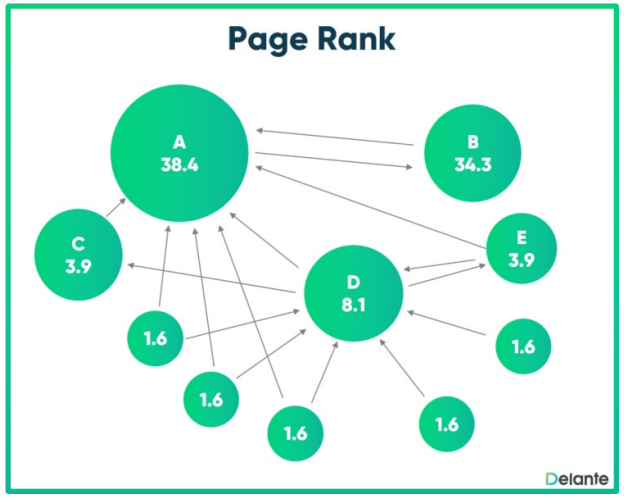
What is SpMV?



y **A** **x**

Sparse Matrix-Vector Multiplication (SpMV) is computed by multiplying each row of the sparse matrix with the dense vector and summing the results to obtain the elements of the output vector.

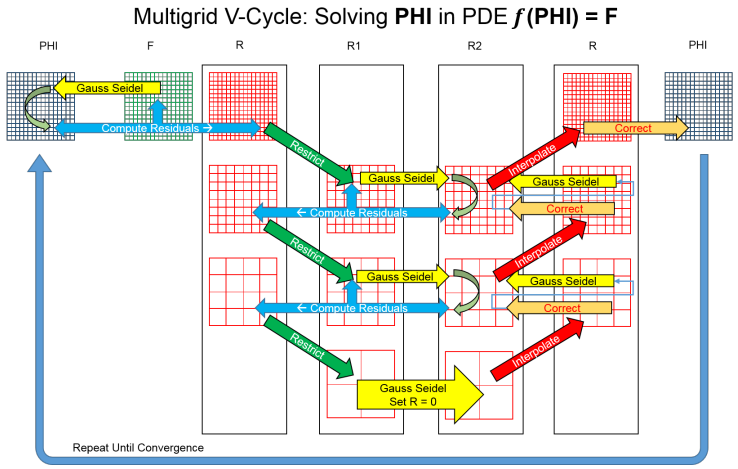
SpMV is ubiquitous



Graph algorithms

$$A X = \lambda X$$

Eiganvalue Systems



Multigrid method

* https://delante.co/definitions/pagerank/

* https://mathworld.wolfram.com/Eigenvalue.html

* https://en.wikipedia.org/wiki/Multigrid_method

Optimizing SpMV on GPUs is challenging

- Load-balanced execution is challenging.
- Achieving coalesced access is nontrivial.
- The performance of SpMV is constrained by the size of sparse matrix
- Efficiently reusing the input and output vectors

Optimizing SpMV on GPUs is challenging

- Load-balanced execution is challenging.
- Achieving coalesced access is nontrivial.

Our focus in this talk

- The performance of SpMV is constrained by the size of sparse matrix
- Efficiently reusing the input and output vectors

Beyond the scope of this talk

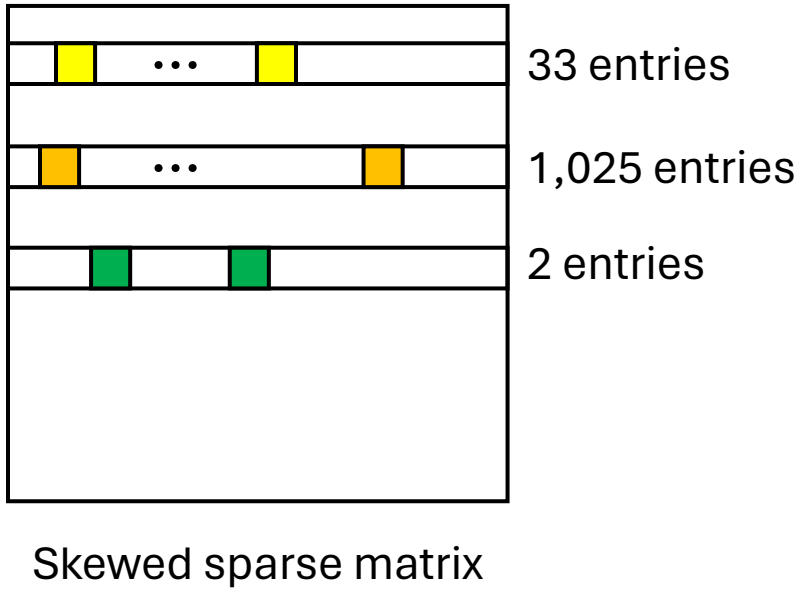
Done with non-standard sparse matrix representation.

How to assign threads to work?

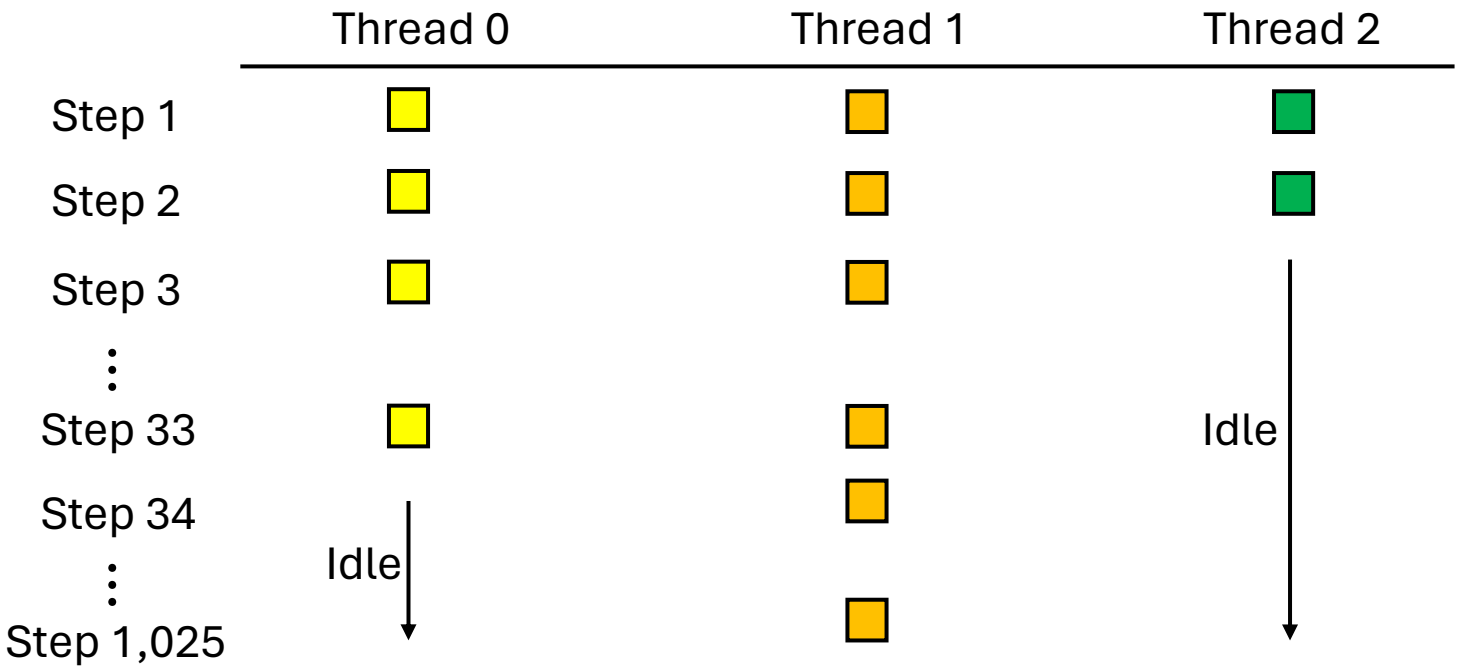
- One thread per row
- One warp (=32 threads) per row
- One thread block (assuming 256 threads) per row

One thread per row

- No reduction cost
- Can result in significant load-imbalance
- Can lead to uncoalesced memory accesses

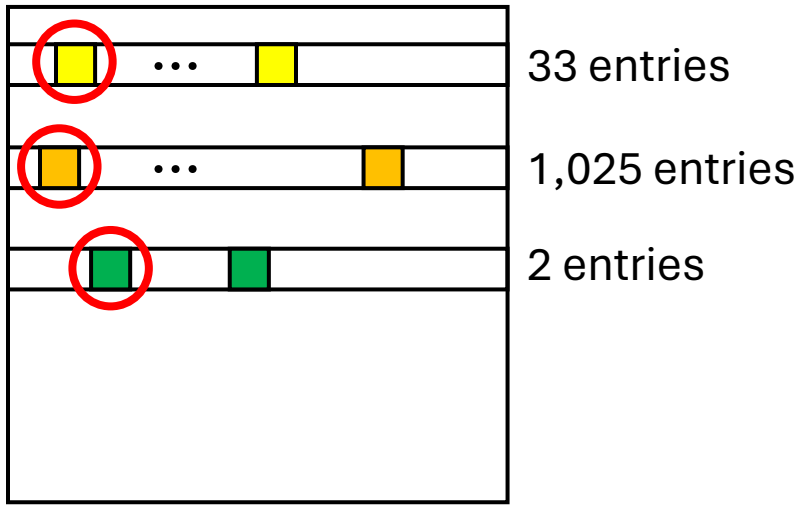


Skewed sparse matrix

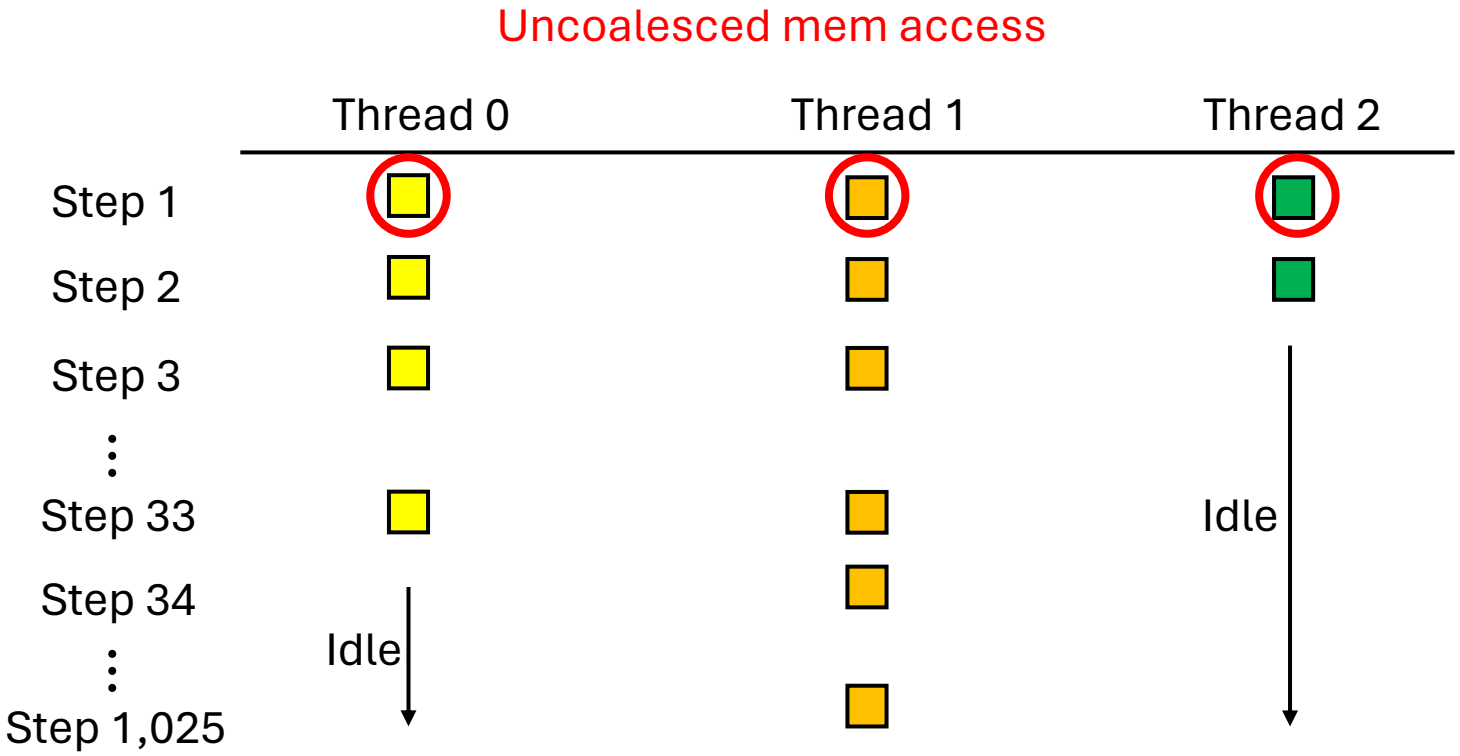


One thread per row

- No reduction cost
- Can result in significant load-imbalance
- Can lead to uncoalesced memory accesses

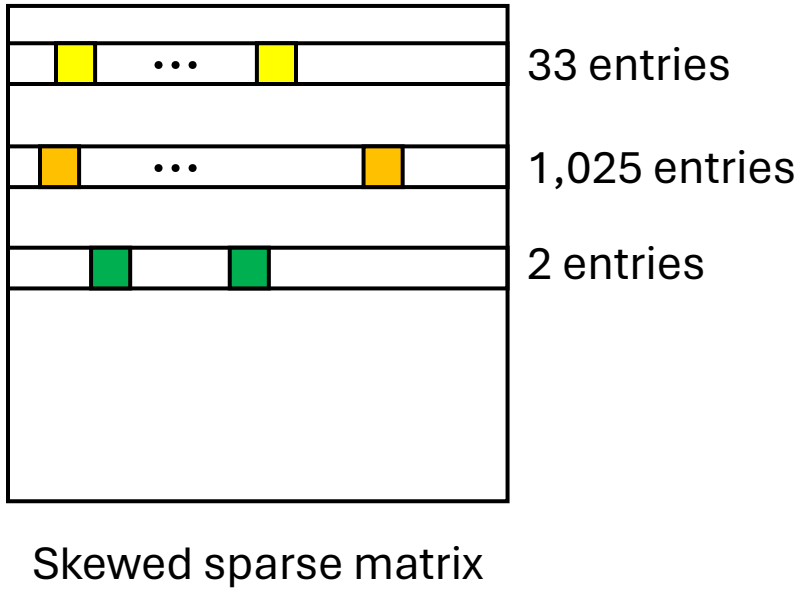


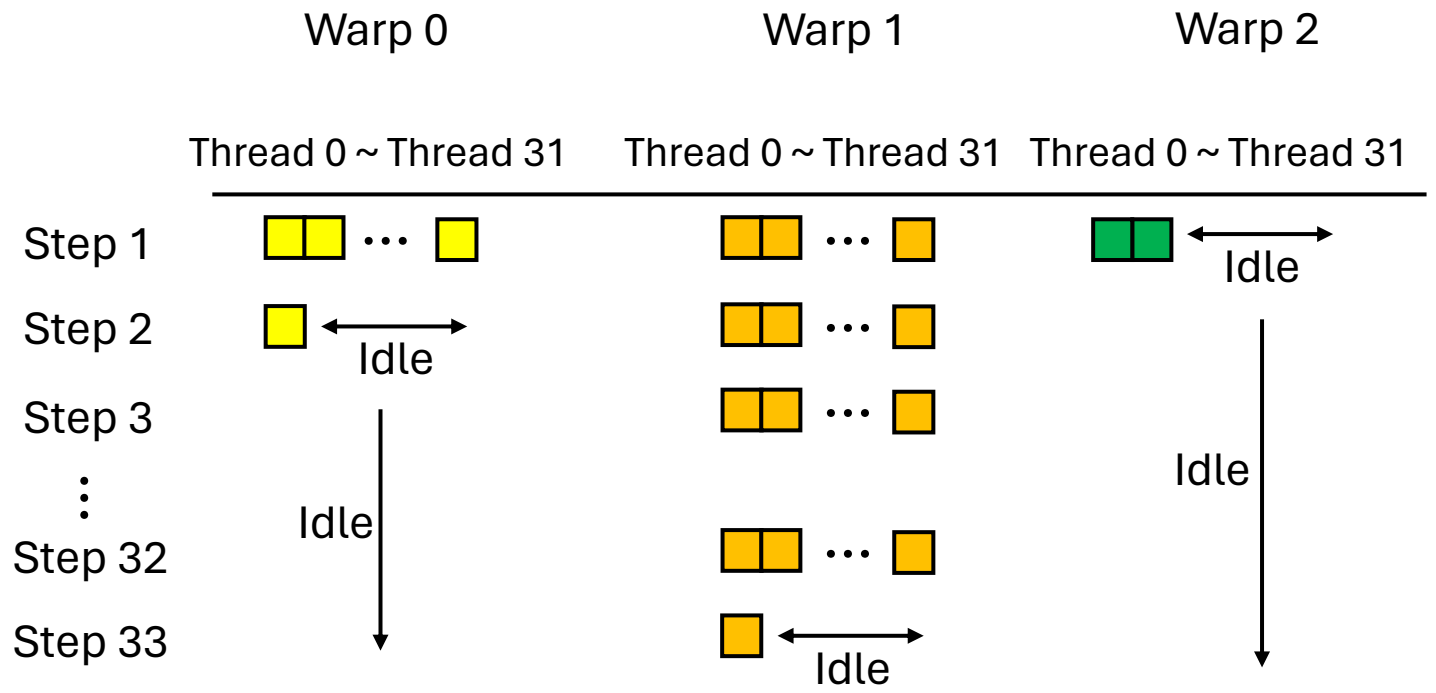
Skewed sparse matrix



One warp per row

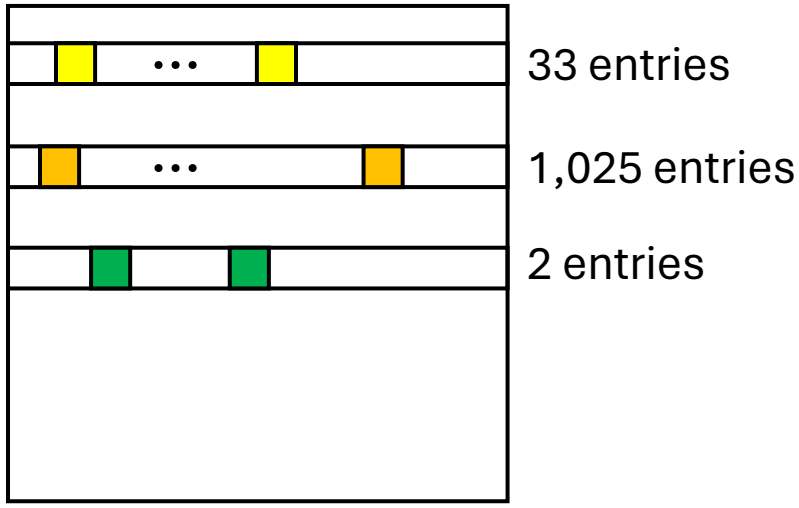
- Result in reduction cost
- Can result in significant load-imbalance



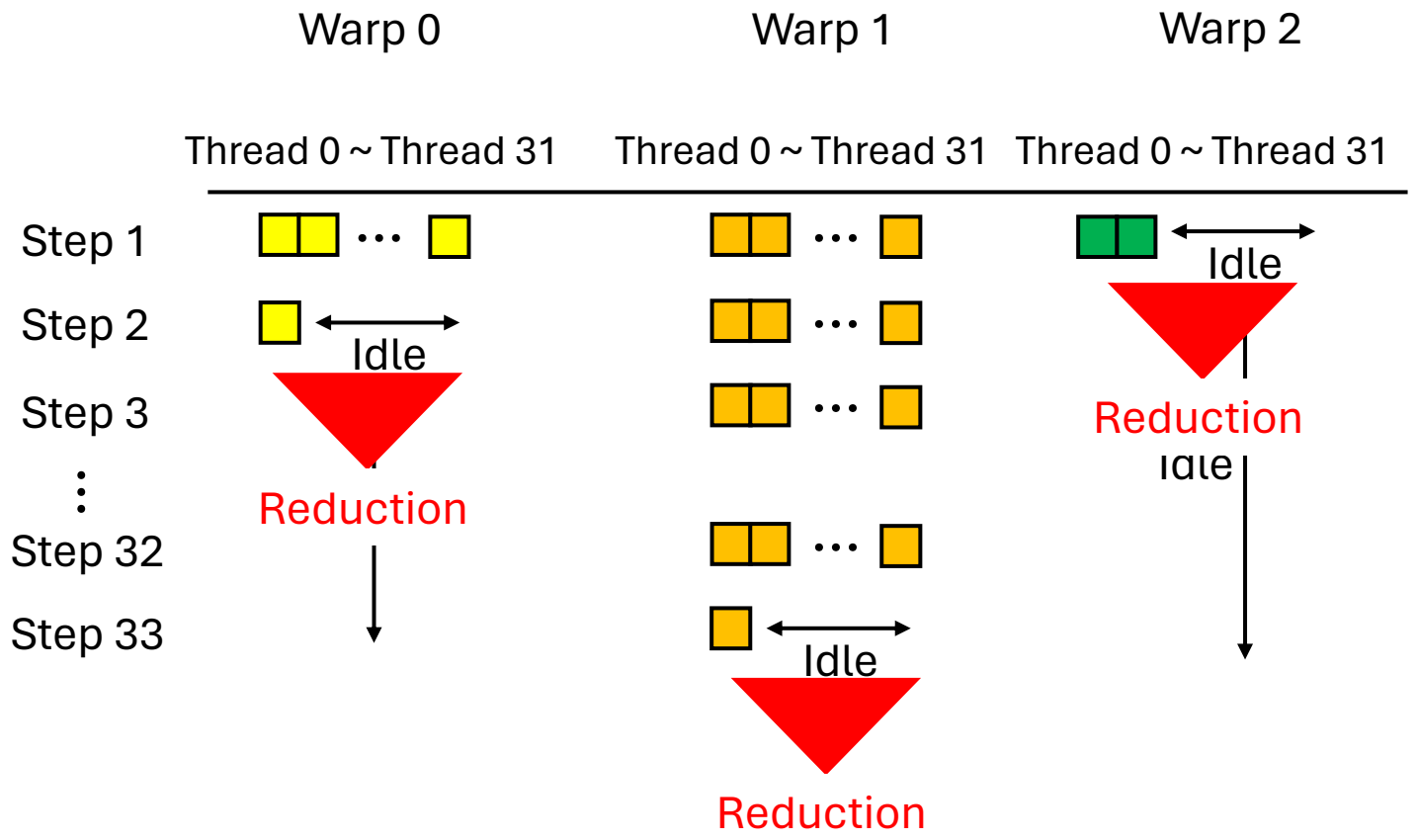


One warp per row

- Result in reduction cost
- Can result in significant load-imbalance

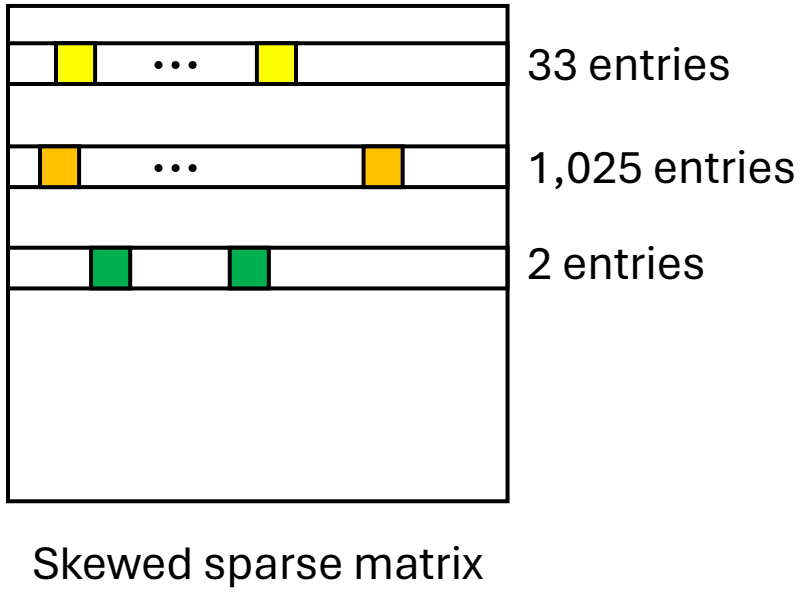


Skewed sparse matrix

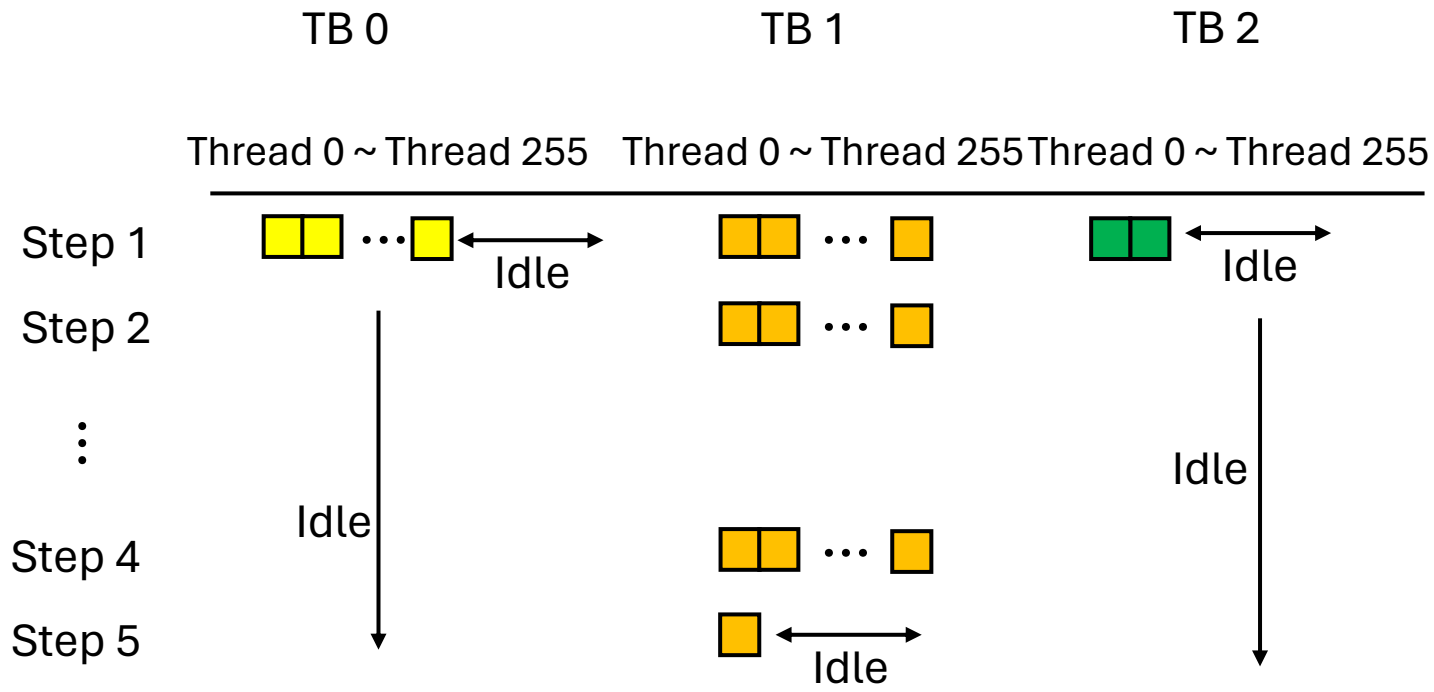


One thread block per row

- Result in more reduction cost
- Can result in significant load-imbalance

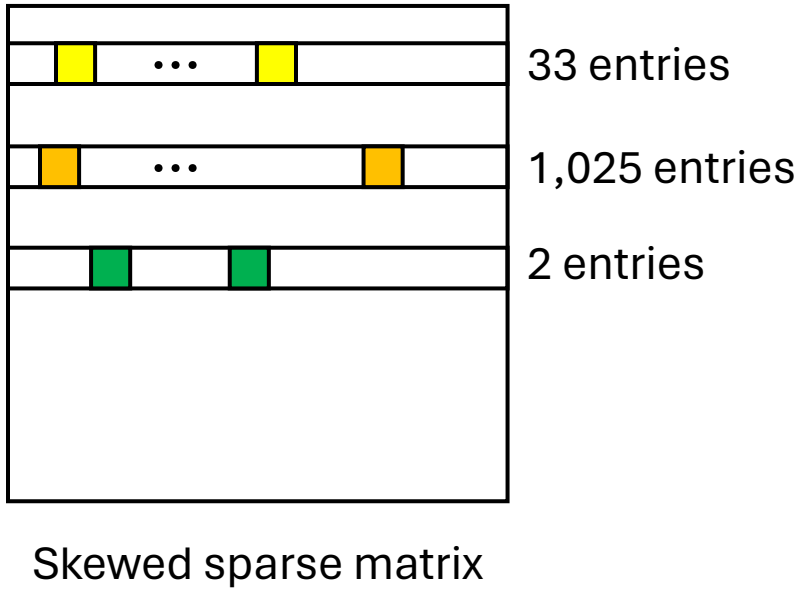


Skewed sparse matrix

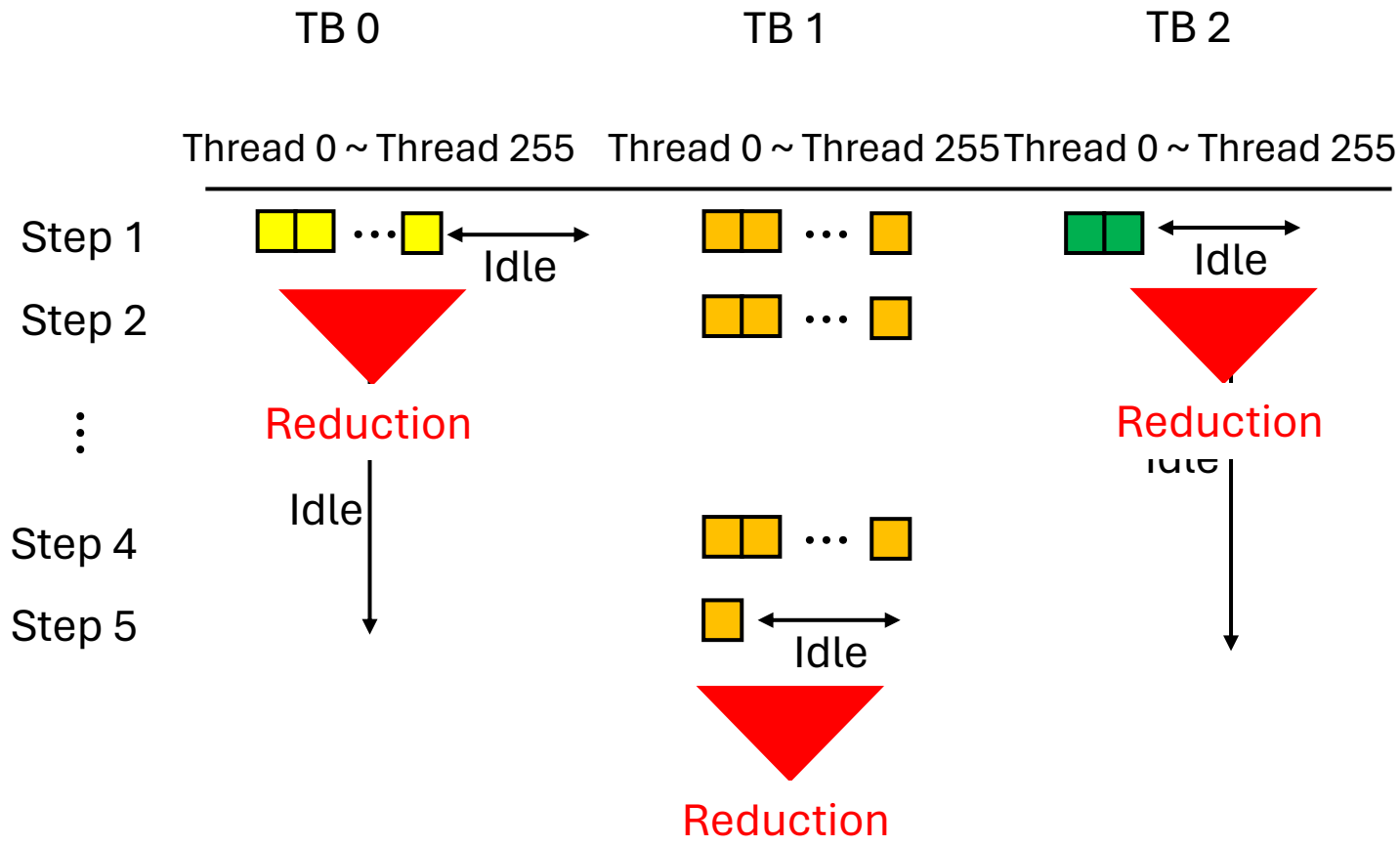


One thread block per row

- Result in more reduction cost
- Can result in significant load-imbalance

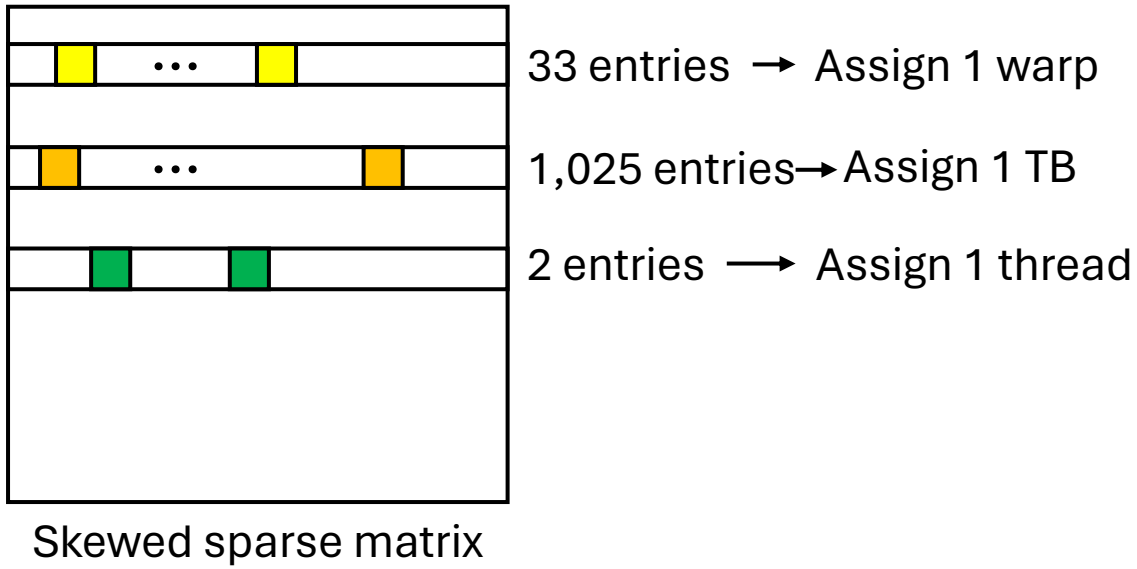


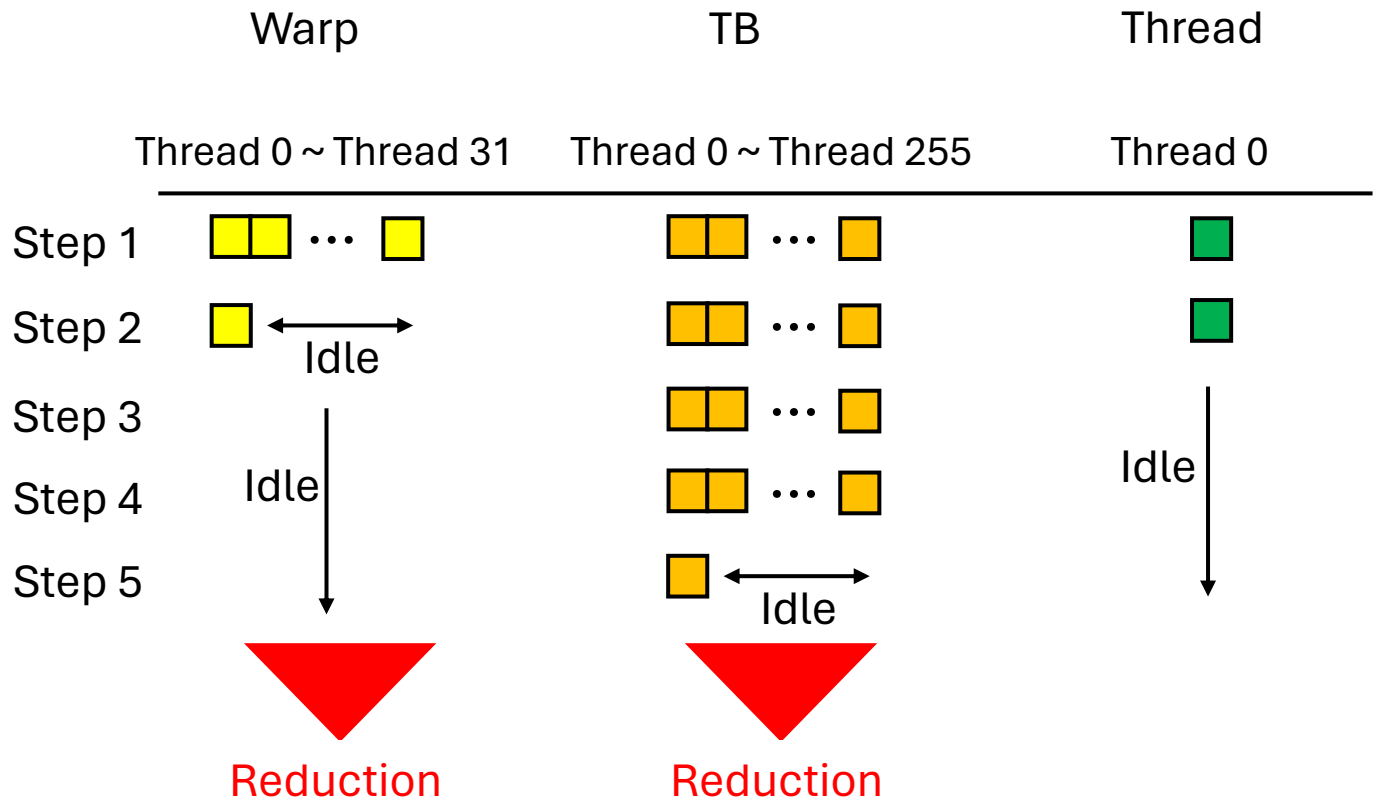
Skewed sparse matrix



Binning can partially resolve this problem

- A thread, warp, or thread block is assigned based on the number of entries in the row
- Still suffers from load-imbalance and reduction cost





* Ashari, Arash, et al. "Fast sparse matrix-vector multiplication on GPUs for graph applications." *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014.

Load-balanced SpMV with coalesced memory access

- Load imbalanced execution
  - Ensure that each thread processes the same number of nonzero entries of the sparse matrix (i.e., strict nonzero splitting).

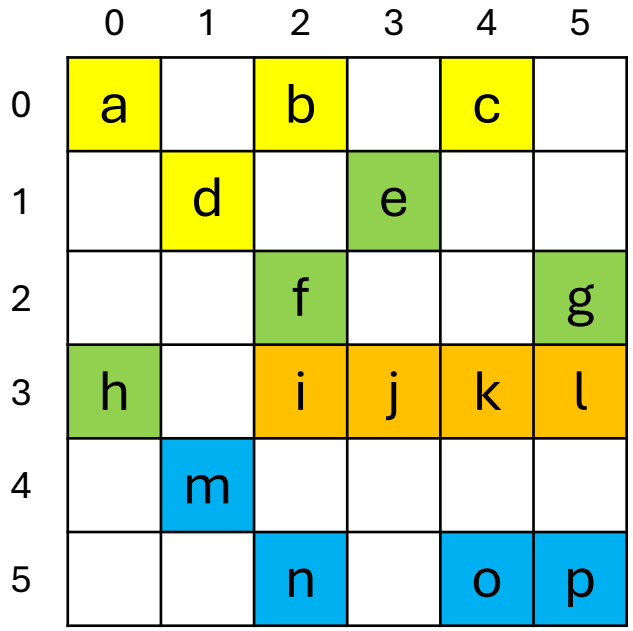
- Achieving coalesced memory access of the sparse matrix
  - First load the sparse matrix into shared memory

- Achieving coalesced memory access of the output
  - Use a shared memory buffer for the output

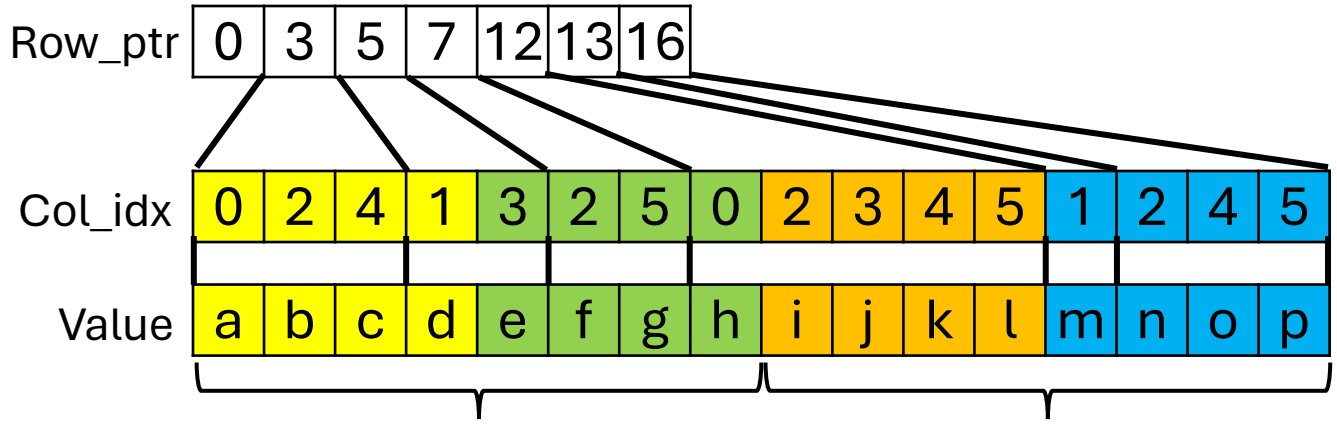
* Steinberger, Markus, Rhaleb Zayer, and Hans-Peter Seidel. "Globally homogeneous, locally adaptive sparse matrix-vector multiplication on the GPU." Proceedings of the International Conference on Supercomputing. 2017.

Strict Nonzero Splitting

- Assuming one thread block consists of two threads, each thread is tasked with processing four nonzero entries of the sparse matrix.



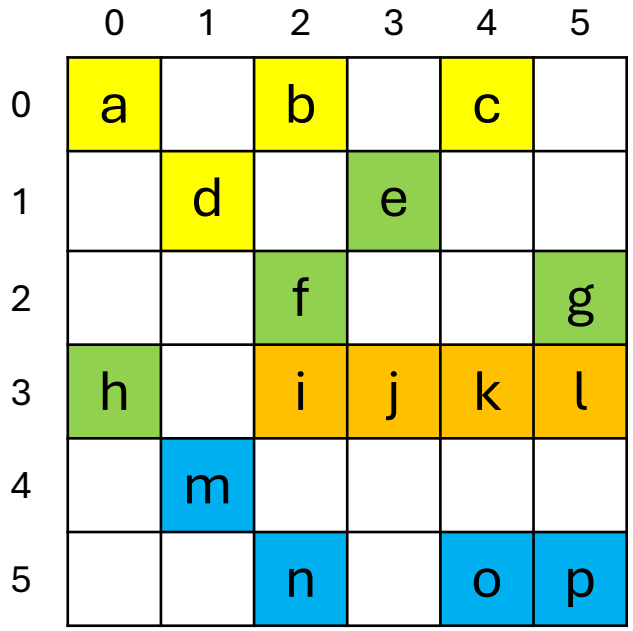
Sparse Matrix



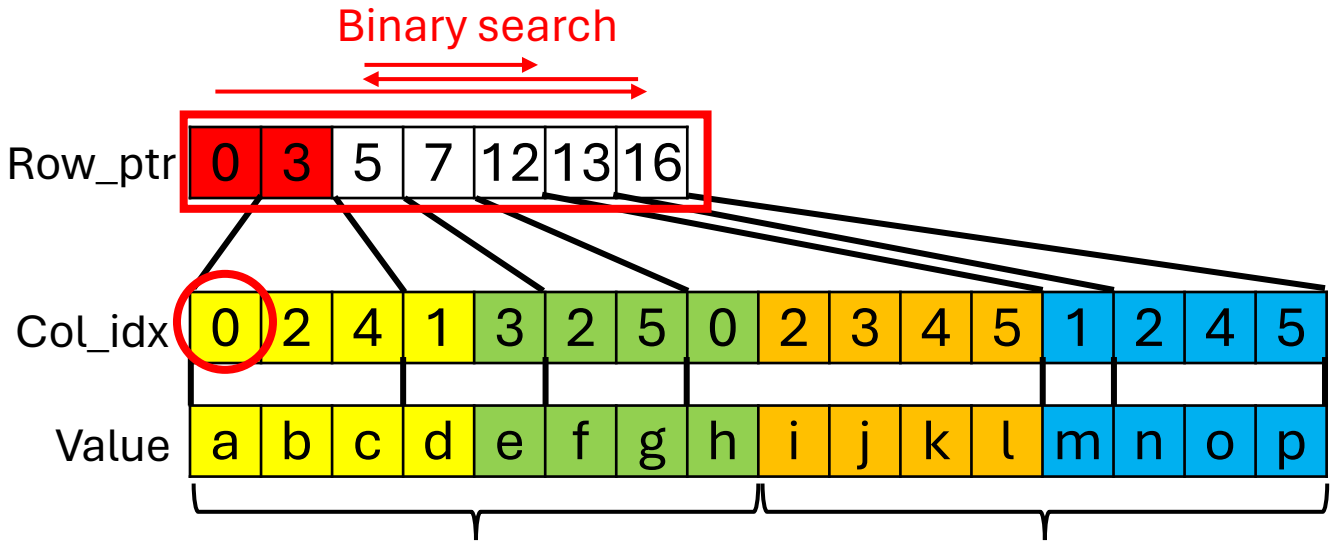
Corresponding CSR

Strict Nonzero Splitting

- Assuming one thread block consists of two threads, each thread is tasked with processing four nonzero entries of the sparse matrix.



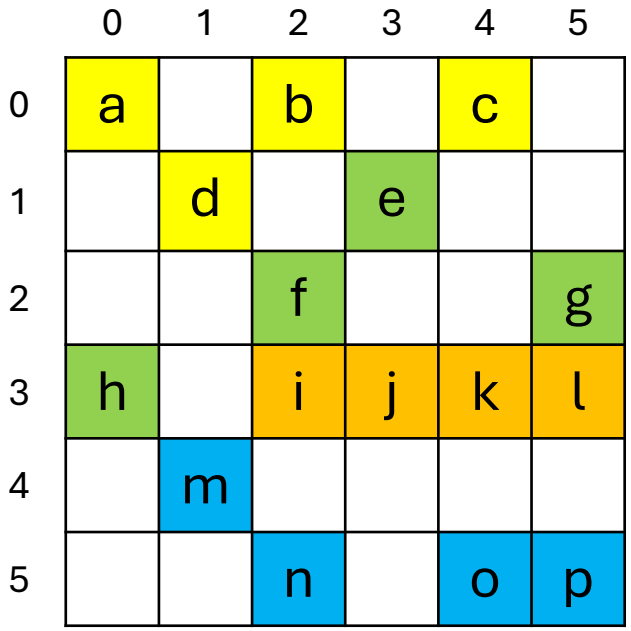
Sparse Matrix



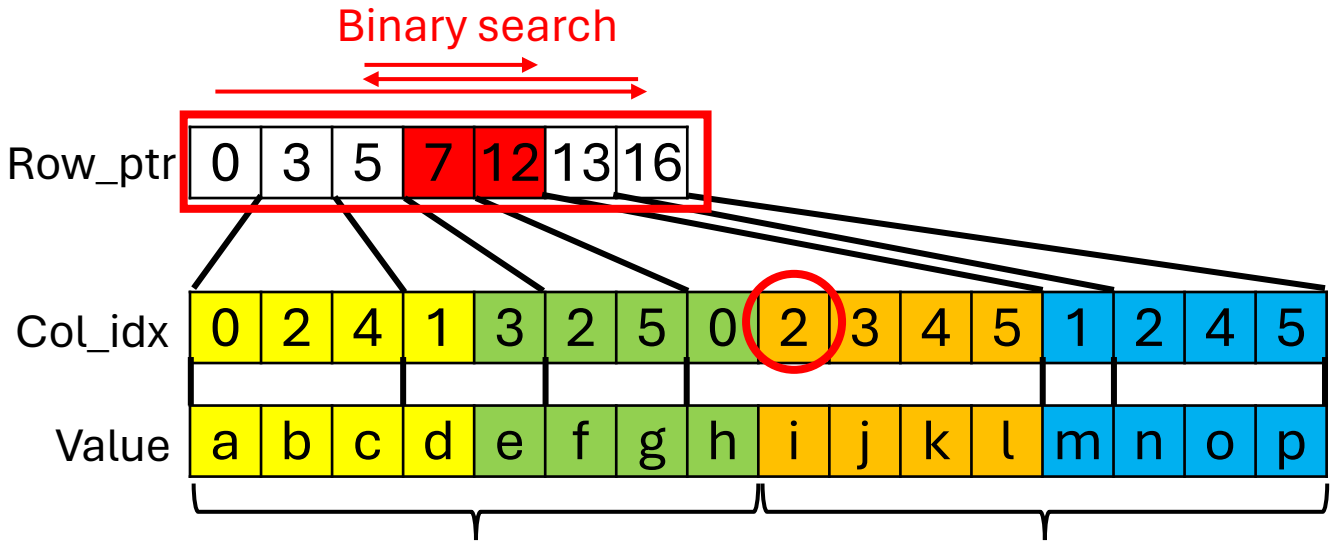
Corresponding CSR

Strict Nonzero Splitting

- Assuming one thread block consists of two threads, each thread is tasked with processing four nonzero entries of the sparse matrix.



Sparse Matrix

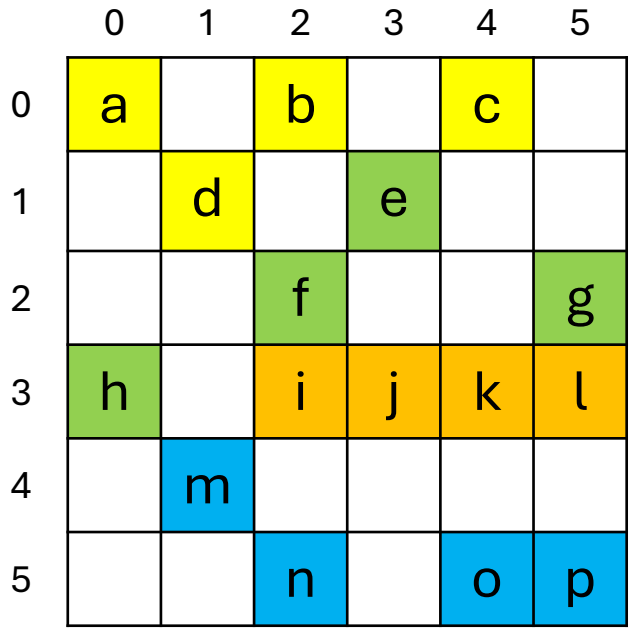


Corresponding CSR

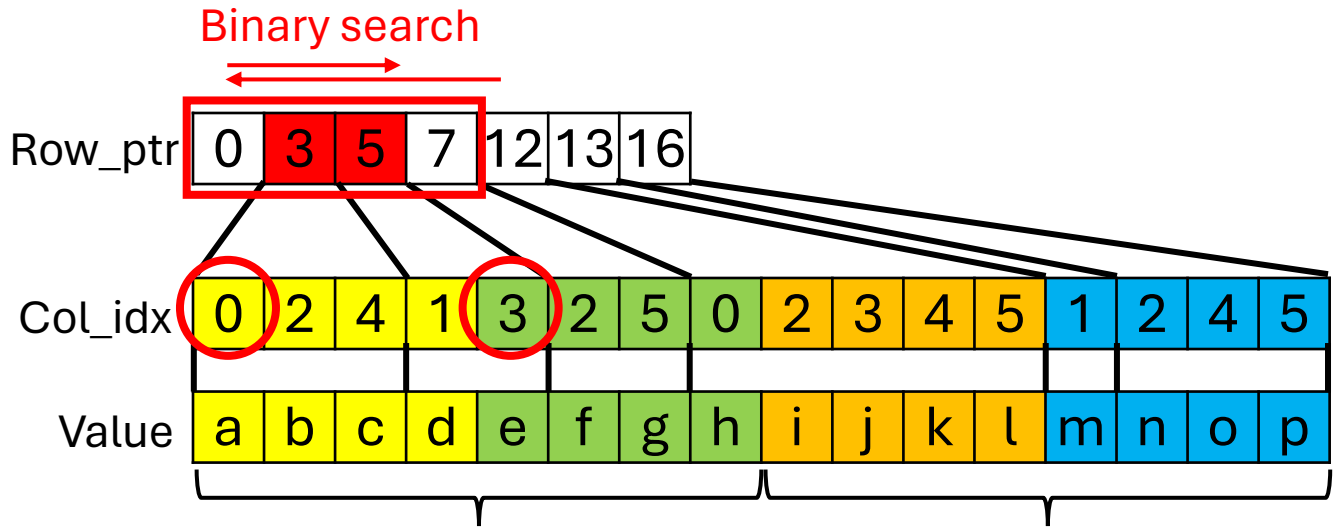
TB_row_start | 0 | 3 |

Strict Nonzero Splitting

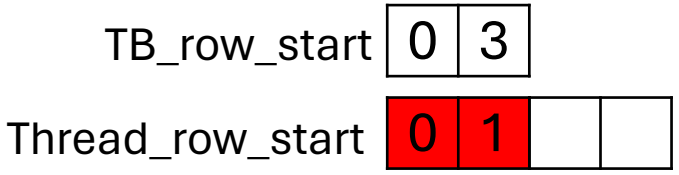
- Assuming one thread block consists of two threads, each thread is tasked with processing four nonzero entries of the sparse matrix.



Sparse Matrix

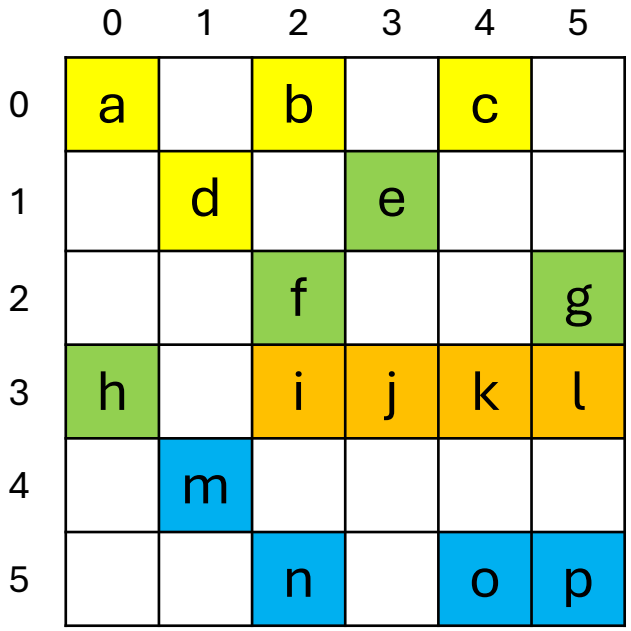


Corresponding CSR

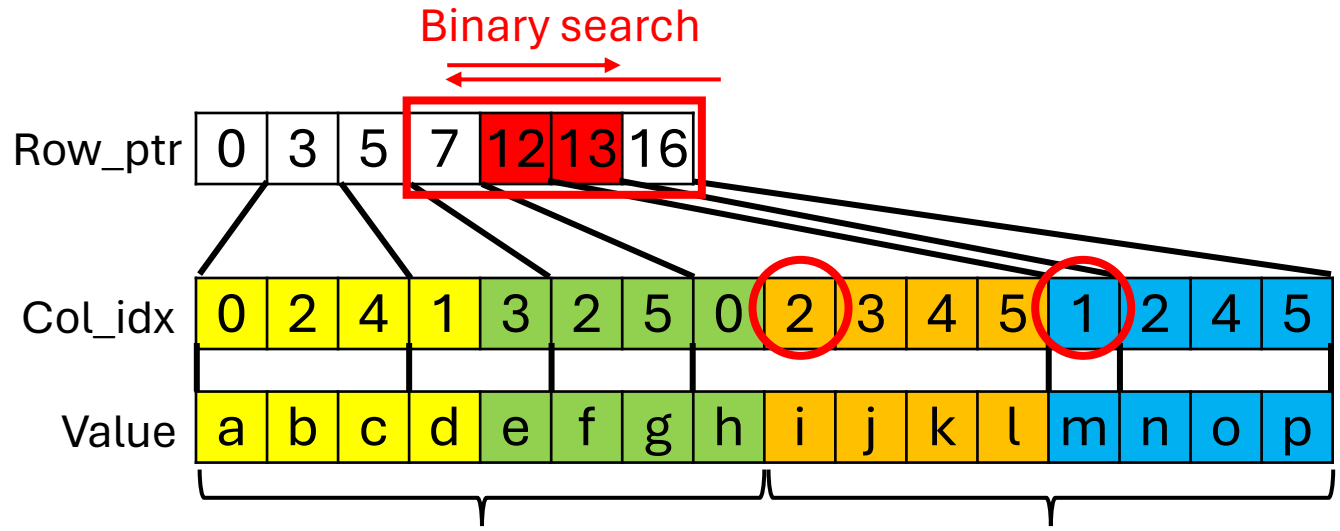


Strict Nonzero Splitting

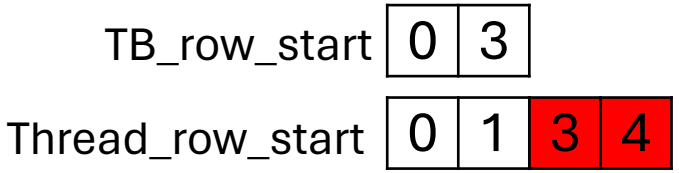
- Assuming one thread block consists of two threads, each thread is tasked with processing four nonzero entries of the sparse matrix.



Sparse Matrix

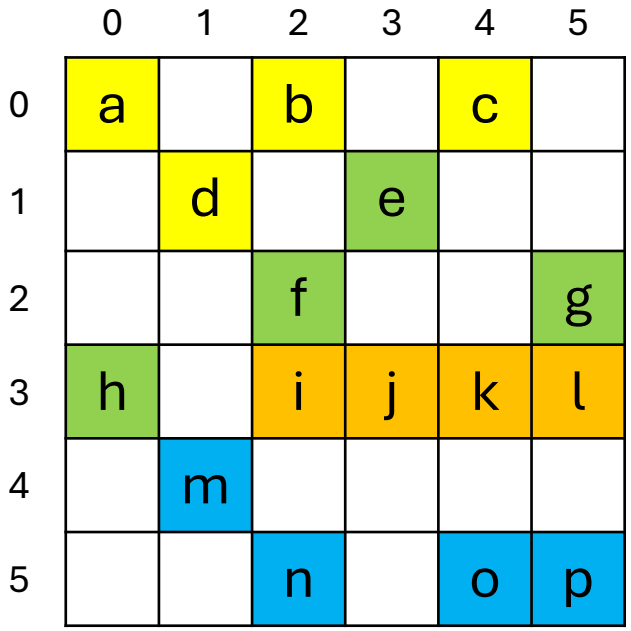


Corresponding CSR

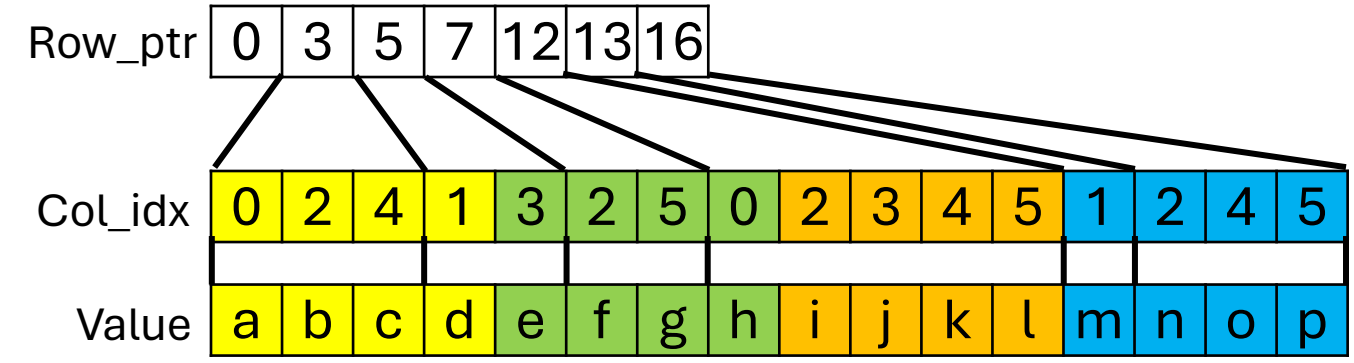


Strict Nonzero Splitting

- Assuming one thread block consists of two threads, each thread is tasked with processing four nonzero entries of the sparse matrix.



Sparse Matrix



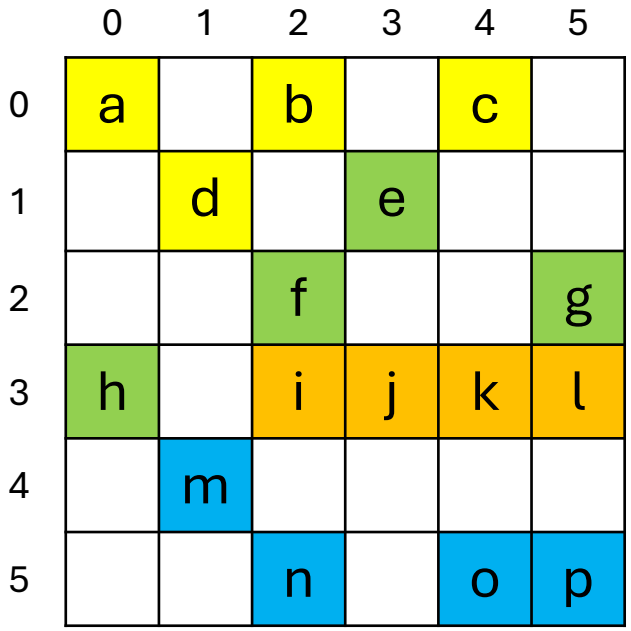
Corresponding CSR

TB_row_start	0	3

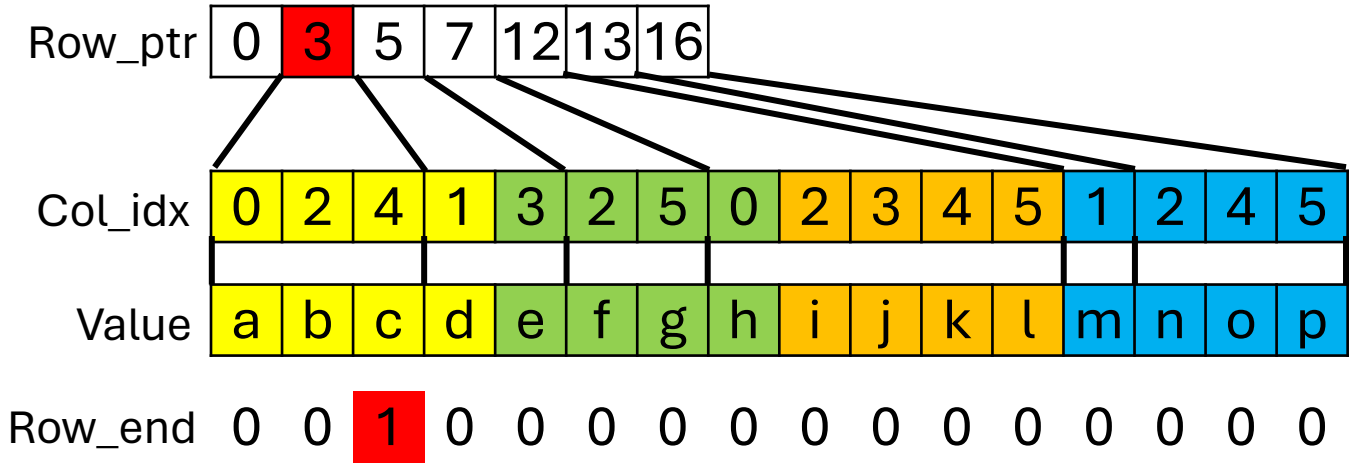
Thread_row_start	0	1	3	4

Strict Nonzero Splitting

- Assuming one thread block consists of two threads, each thread is tasked with processing four nonzero entries of the sparse matrix.



Sparse Matrix



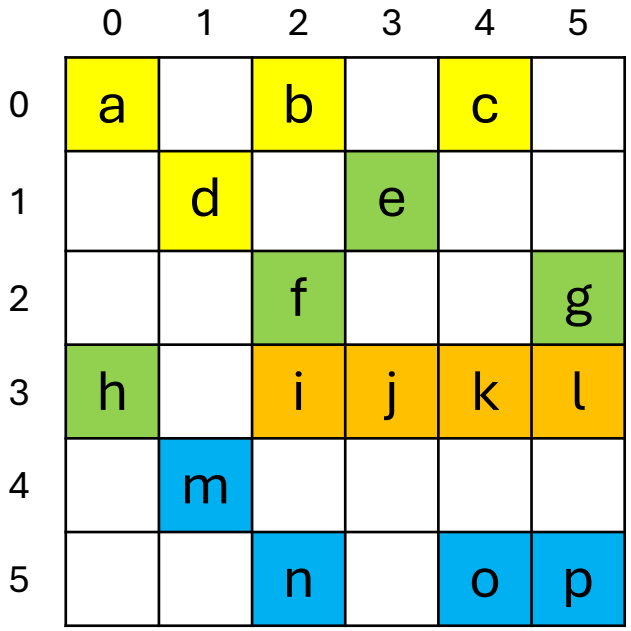
Corresponding CSR

TB_row_start	0	3

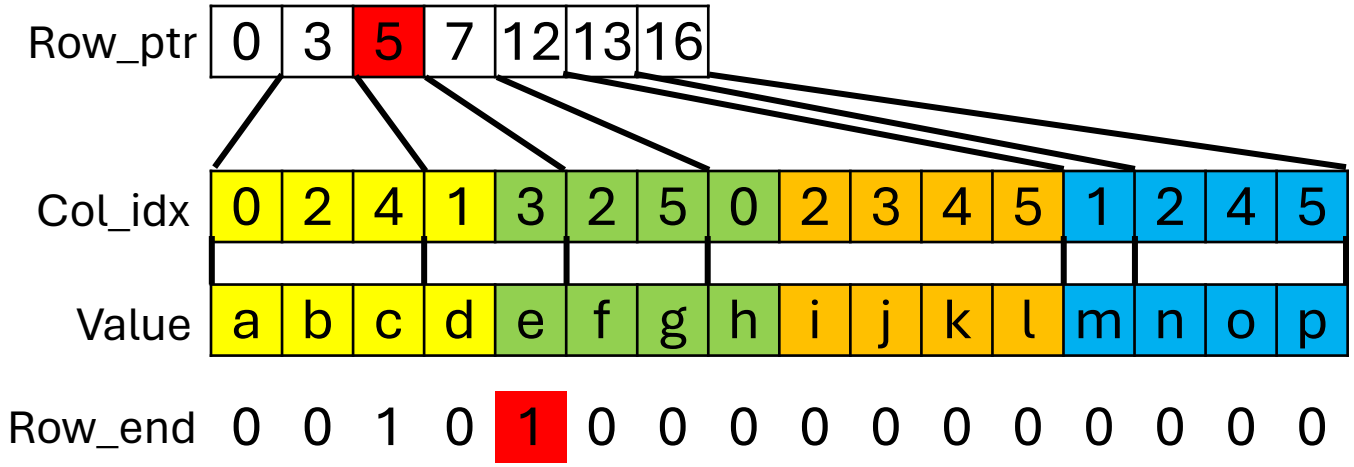
Thread_row_start	0	1	3	4

Strict Nonzero Splitting

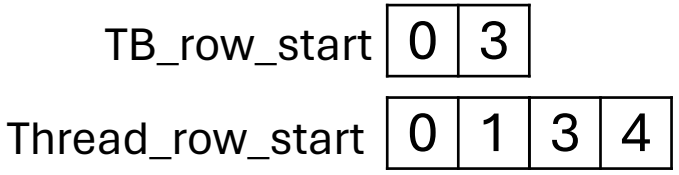
- Assuming one thread block consists of two threads, each thread is tasked with processing four nonzero entries of the sparse matrix.



Sparse Matrix

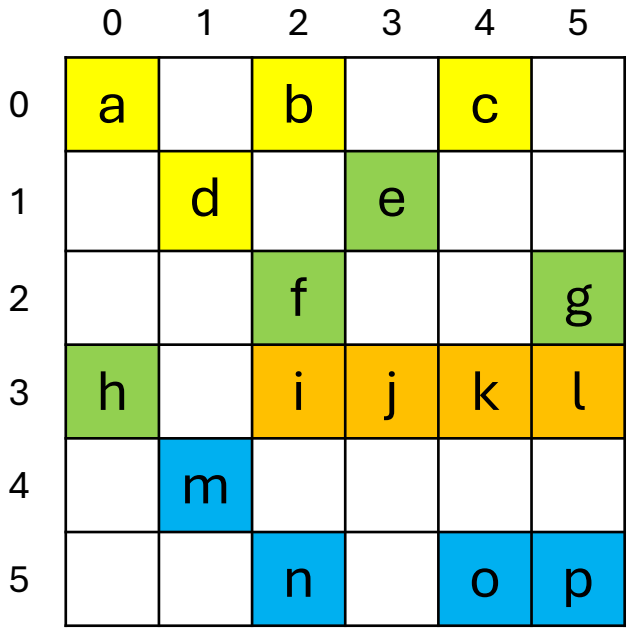


Corresponding CSR

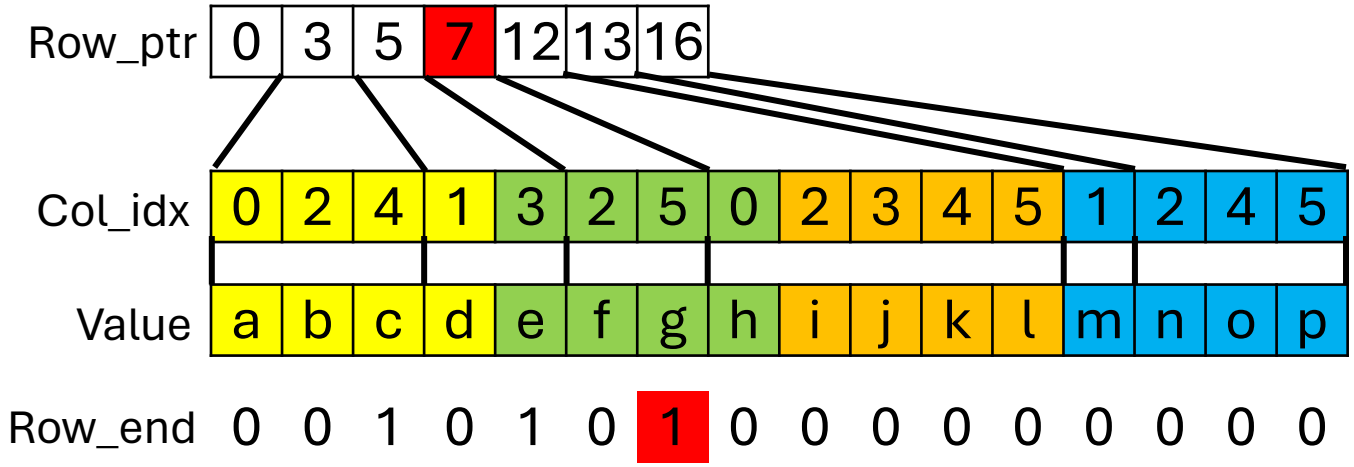


Strict Nonzero Splitting

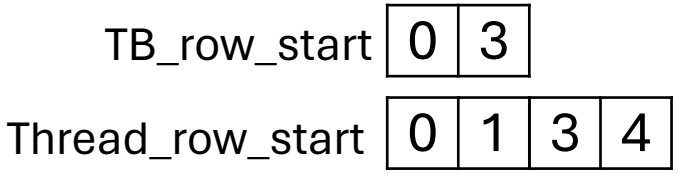
- Assuming one thread block consists of two threads, each thread is tasked with processing four nonzero entries of the sparse matrix.



Sparse Matrix

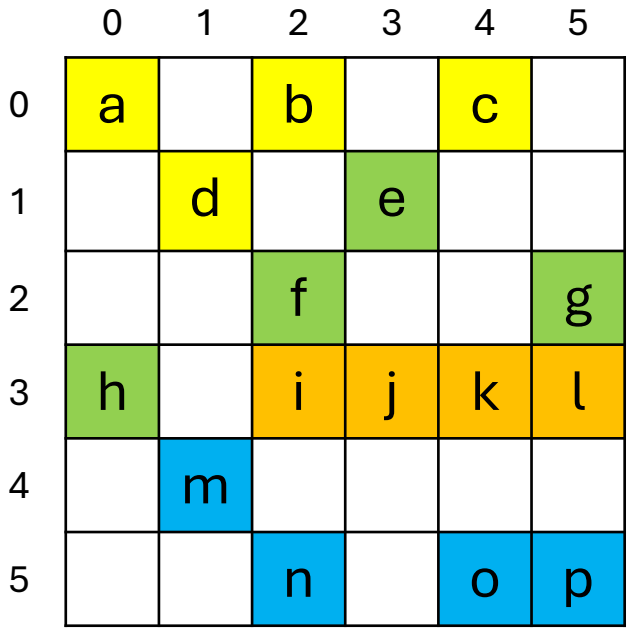


Corresponding CSR

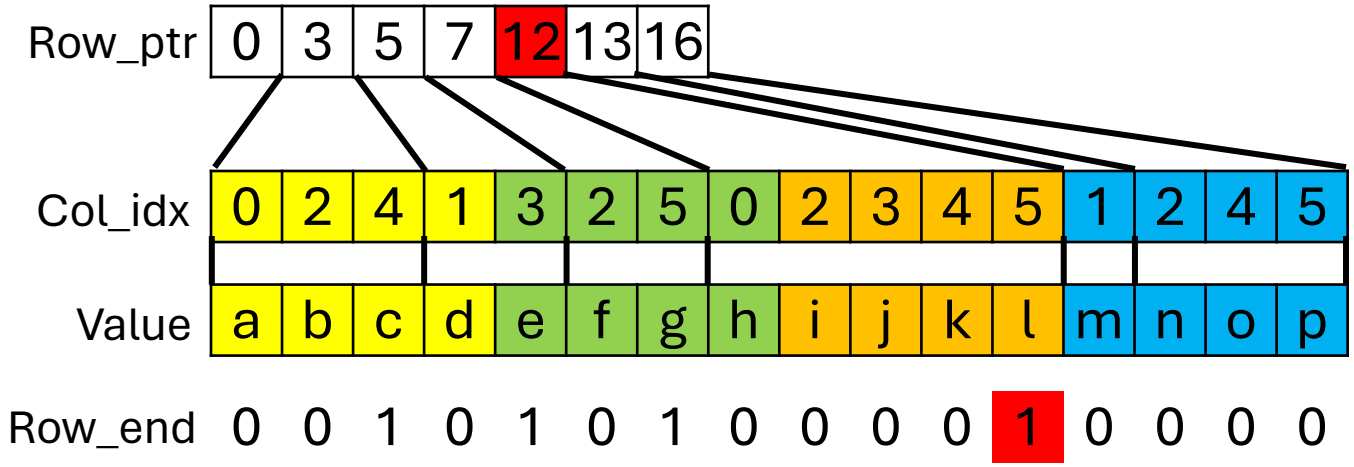


Strict Nonzero Splitting

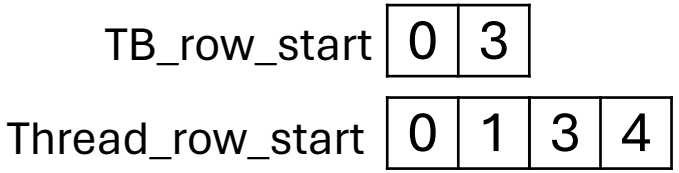
- Assuming one thread block consists of two threads, each thread is tasked with processing four nonzero entries of the sparse matrix.



Sparse Matrix

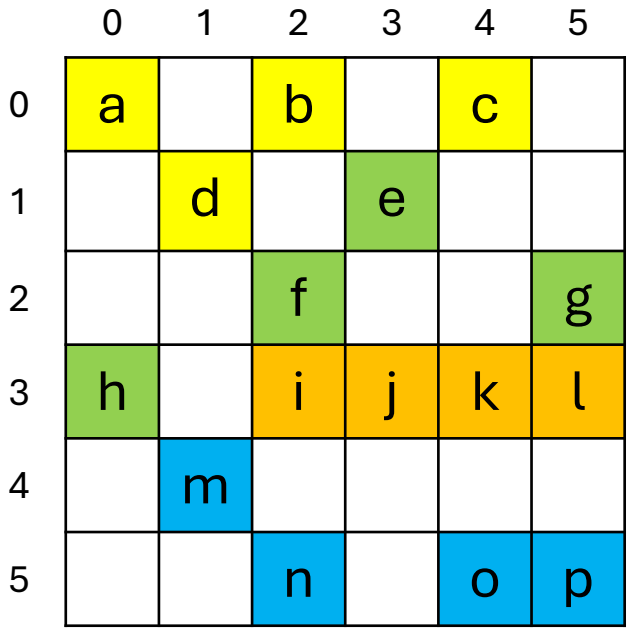


Corresponding CSR

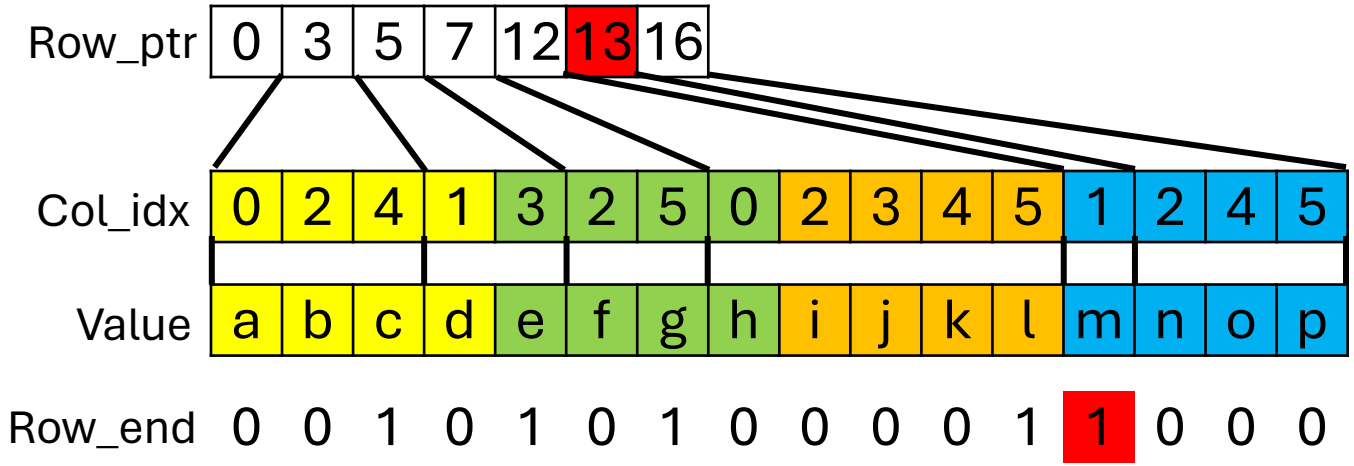


Strict Nonzero Splitting

- Assuming one thread block consists of two threads, each thread is tasked with processing four nonzero entries of the sparse matrix.



Sparse Matrix



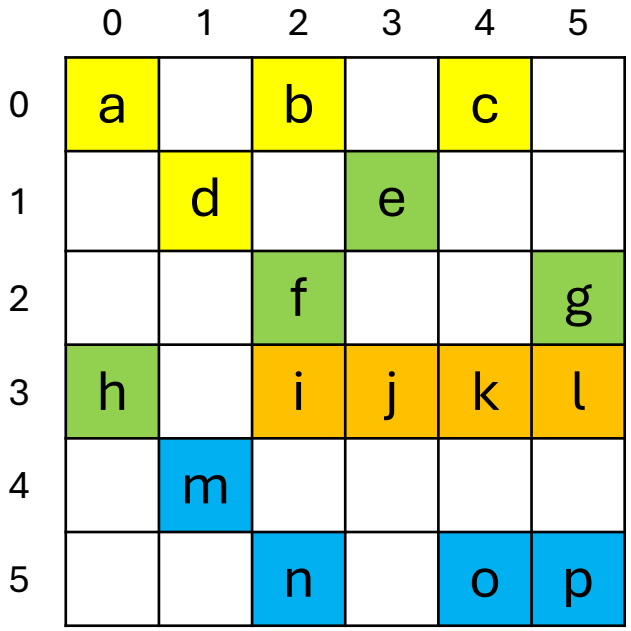
Corresponding CSR

TB_row_start	0	3

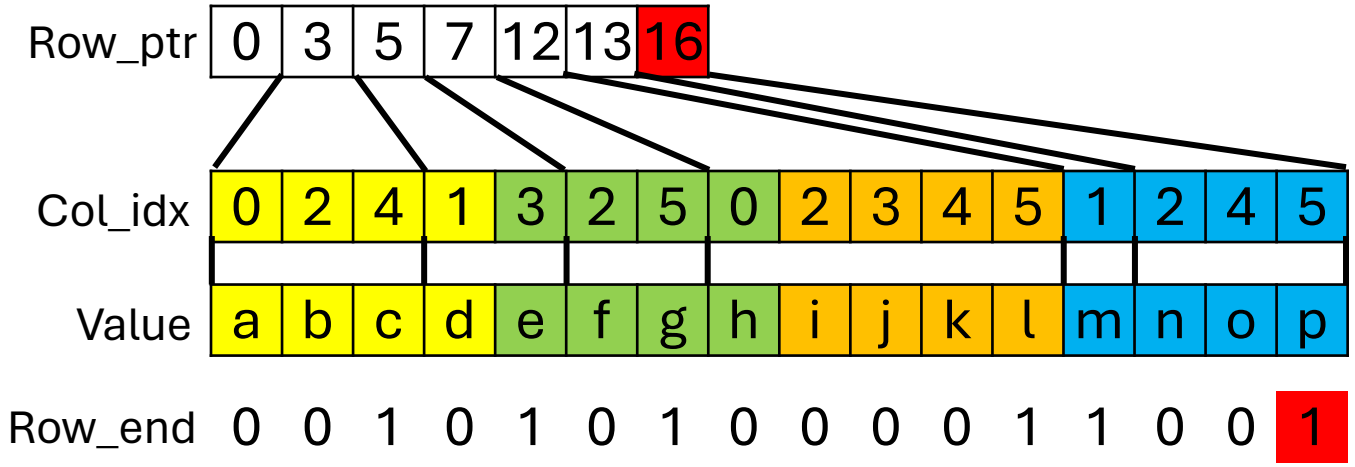
Thread_row_start	0	1	3	4

Strict Nonzero Splitting

- Assuming one thread block consists of two threads, each thread is tasked with processing four nonzero entries of the sparse matrix.



Sparse Matrix



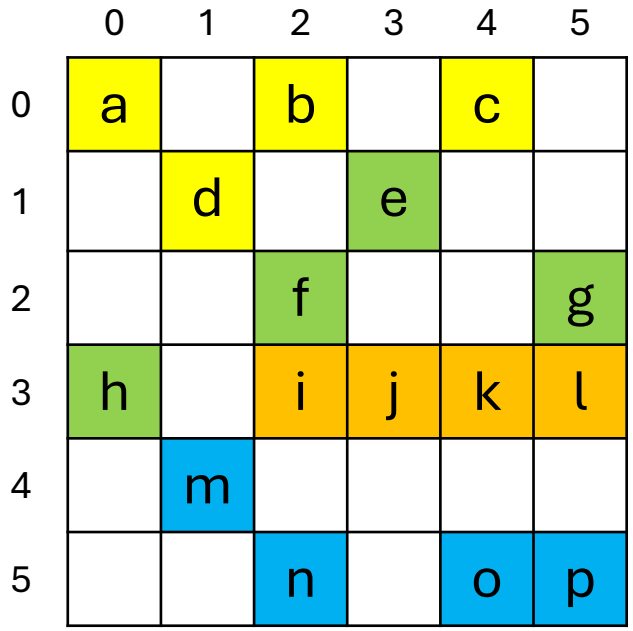
Corresponding CSR

TB_row_start	0	3

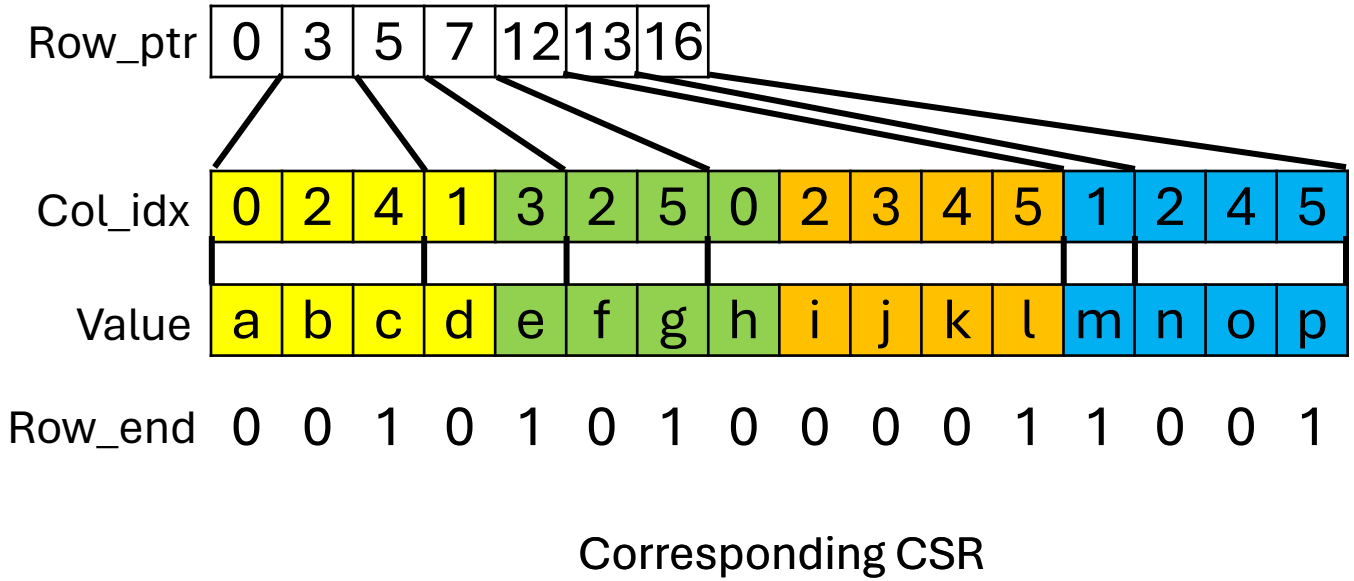
Thread_row_start	0	1	3	4

Strict Nonzero Splitting

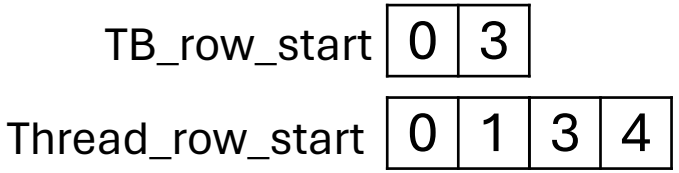
- Assuming one thread block consists of two threads, each thread is tasked with processing four nonzero entries of the sparse matrix.



Sparse Matrix

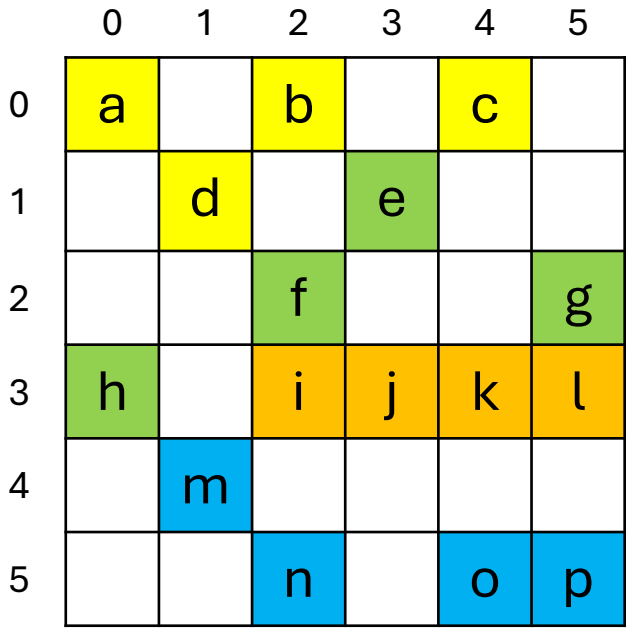


Corresponding CSR

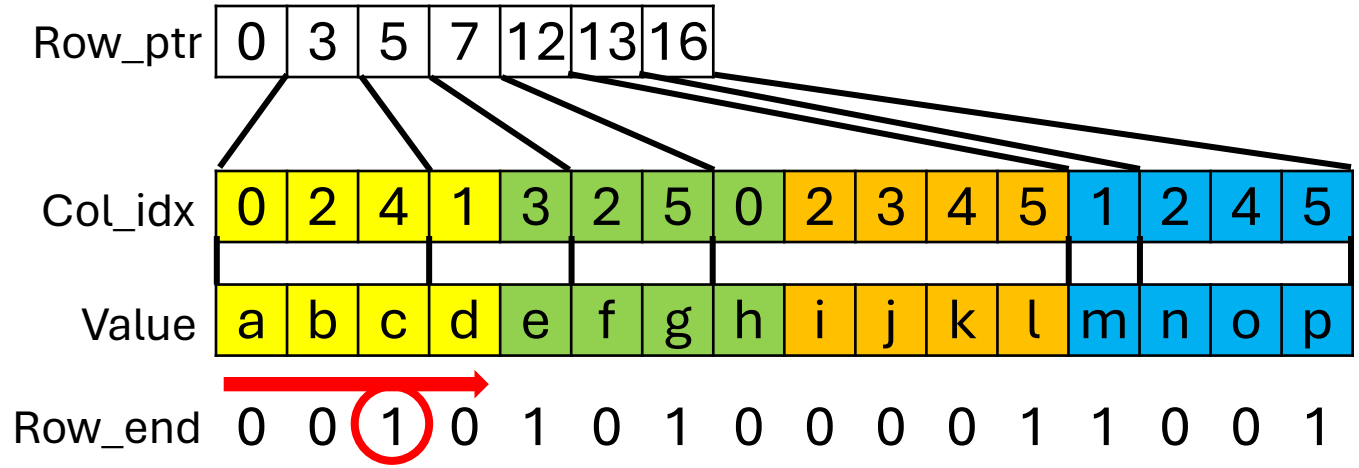


Strict Nonzero Splitting

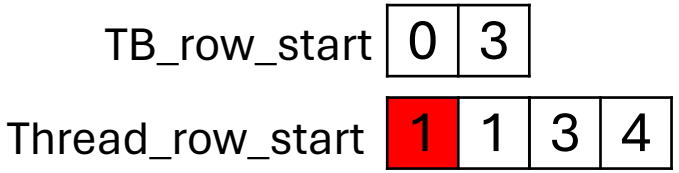
- Assuming one thread block consists of two threads, each thread is tasked with processing four nonzero entries of the sparse matrix.



Sparse Matrix

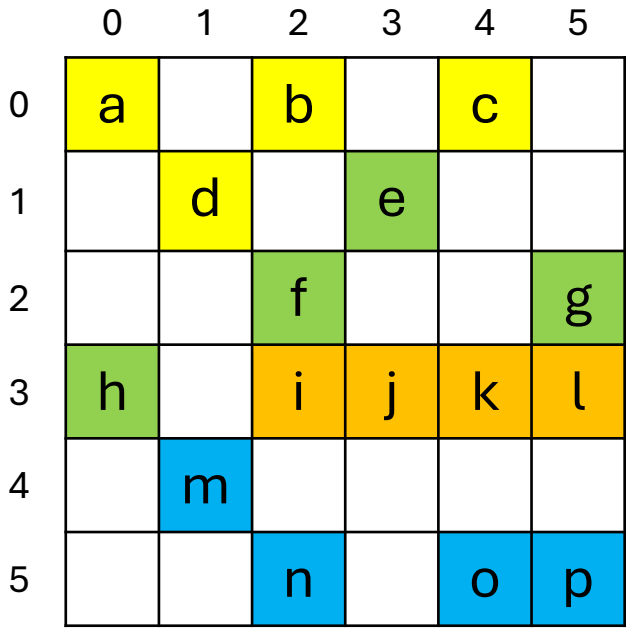


Write the accumulated result to the output, and increment Thread_row_start by 1

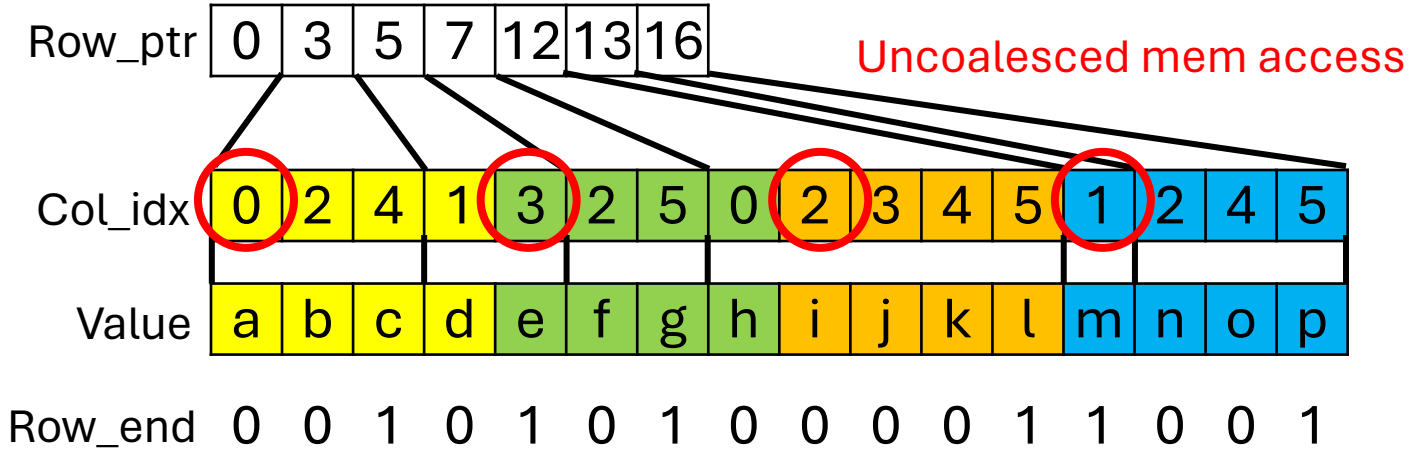


Uncoalesced Access to the Sparse Matrix

- Assuming one warp consists of four threads, each thread is tasked with processing four nonzero entries of the sparse matrix.



Sparse Matrix

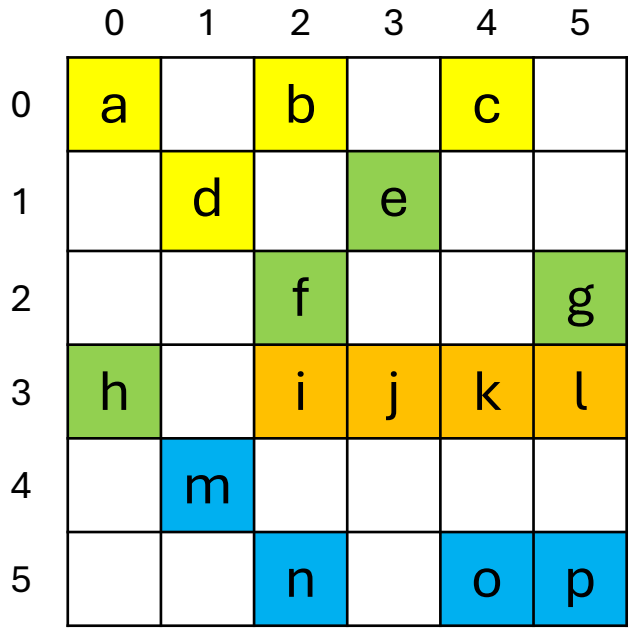


TB_row_start	0	3

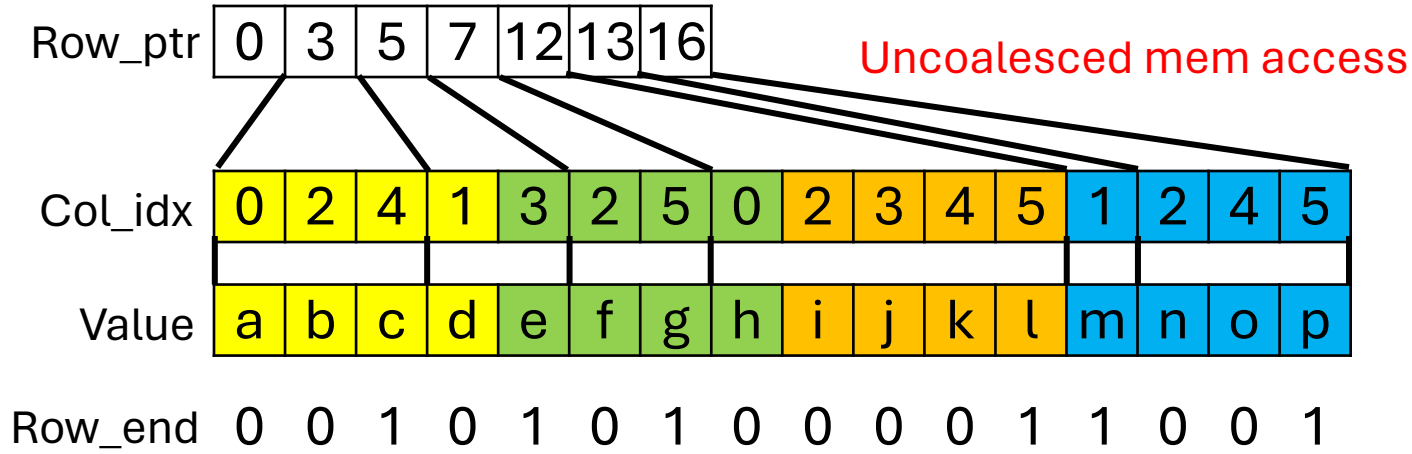
Thread_row_start	0	1	3	4

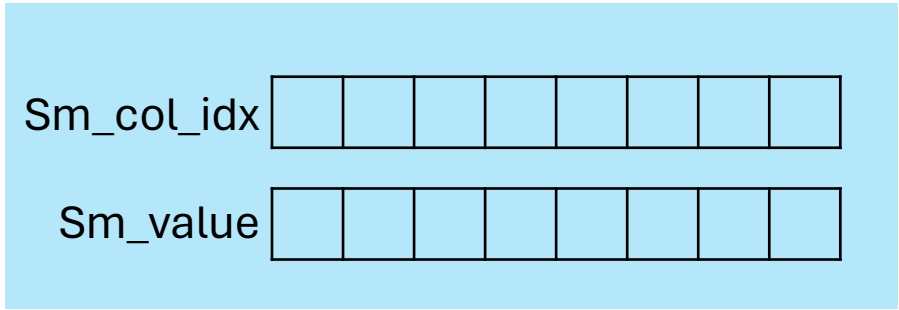
Uncoalesced Access to the Sparse Matrix

- Assuming one warp consists of four threads, each thread is tasked with processing four nonzero entries of the sparse matrix.



Sparse Matrix

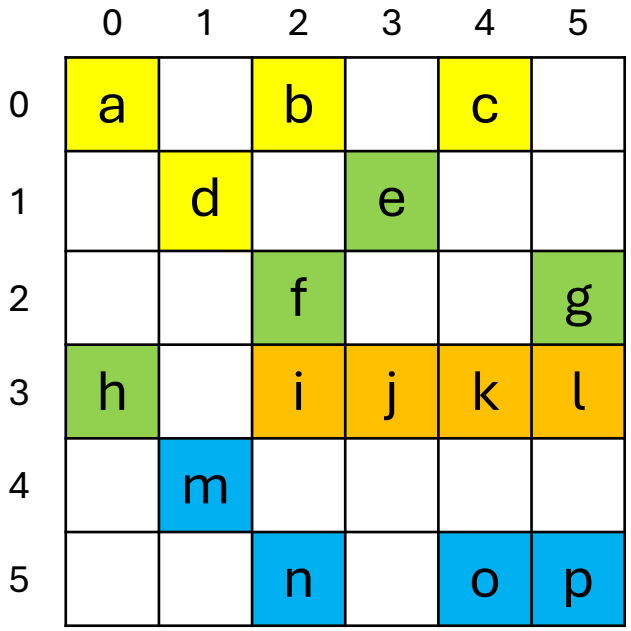




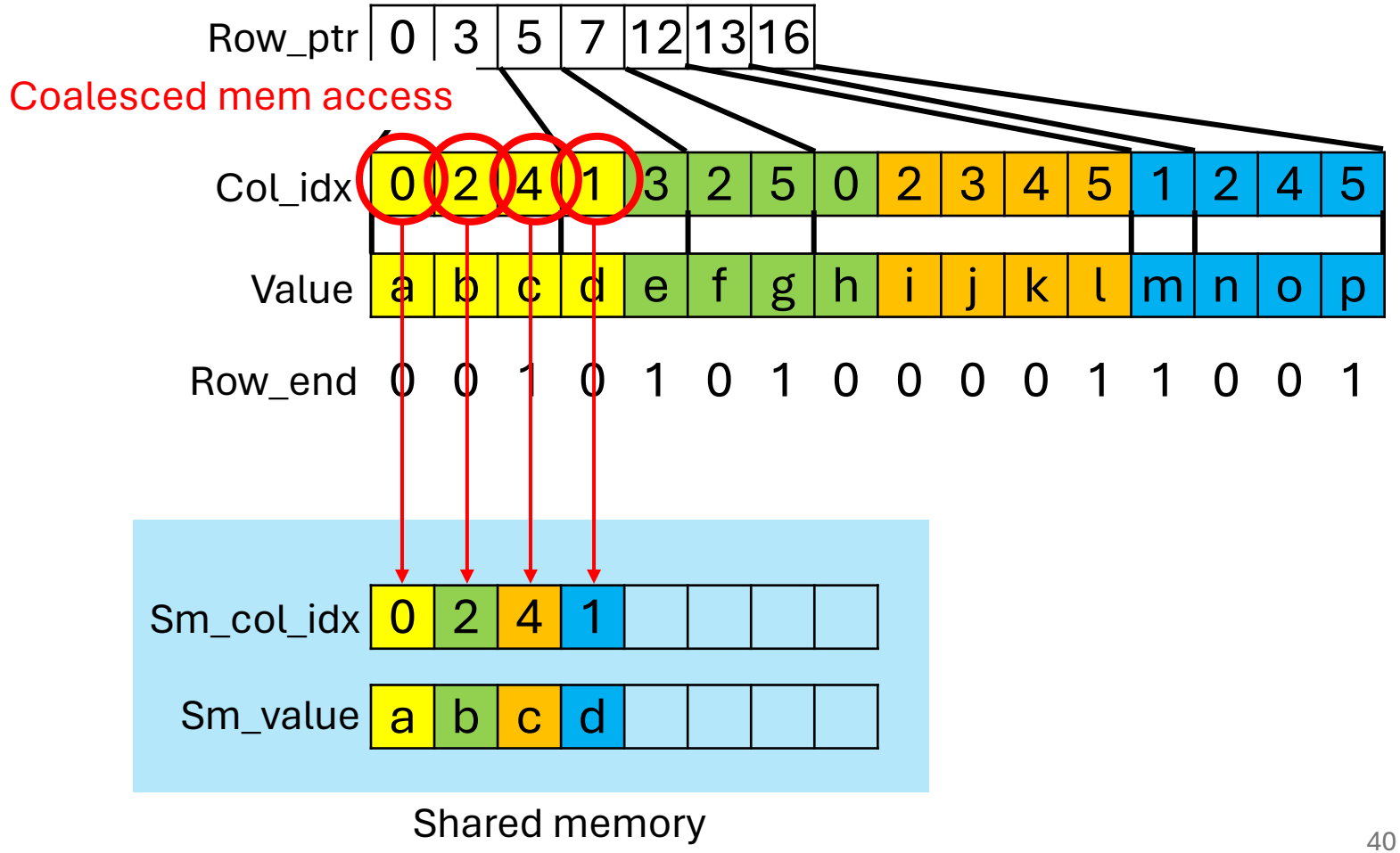
Shared memory

Uncoalesced Access to the Sparse Matrix

- Assuming one warp consists of four threads, each thread is tasked with processing four nonzero entries of the sparse matrix.



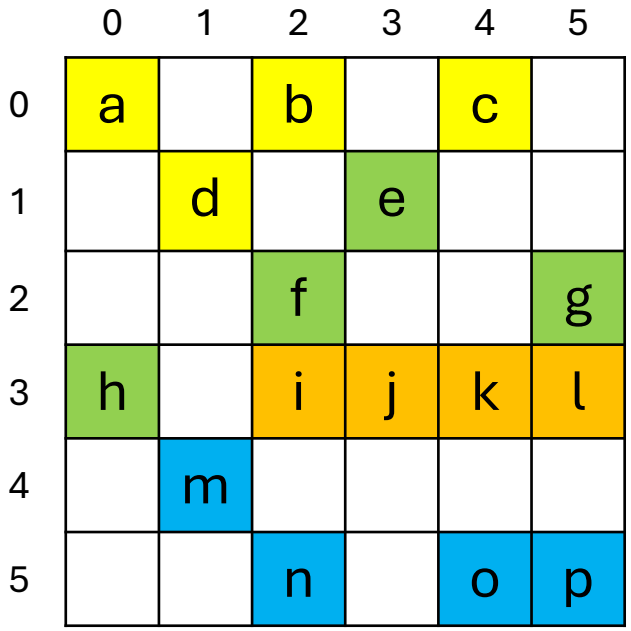
Sparse Matrix



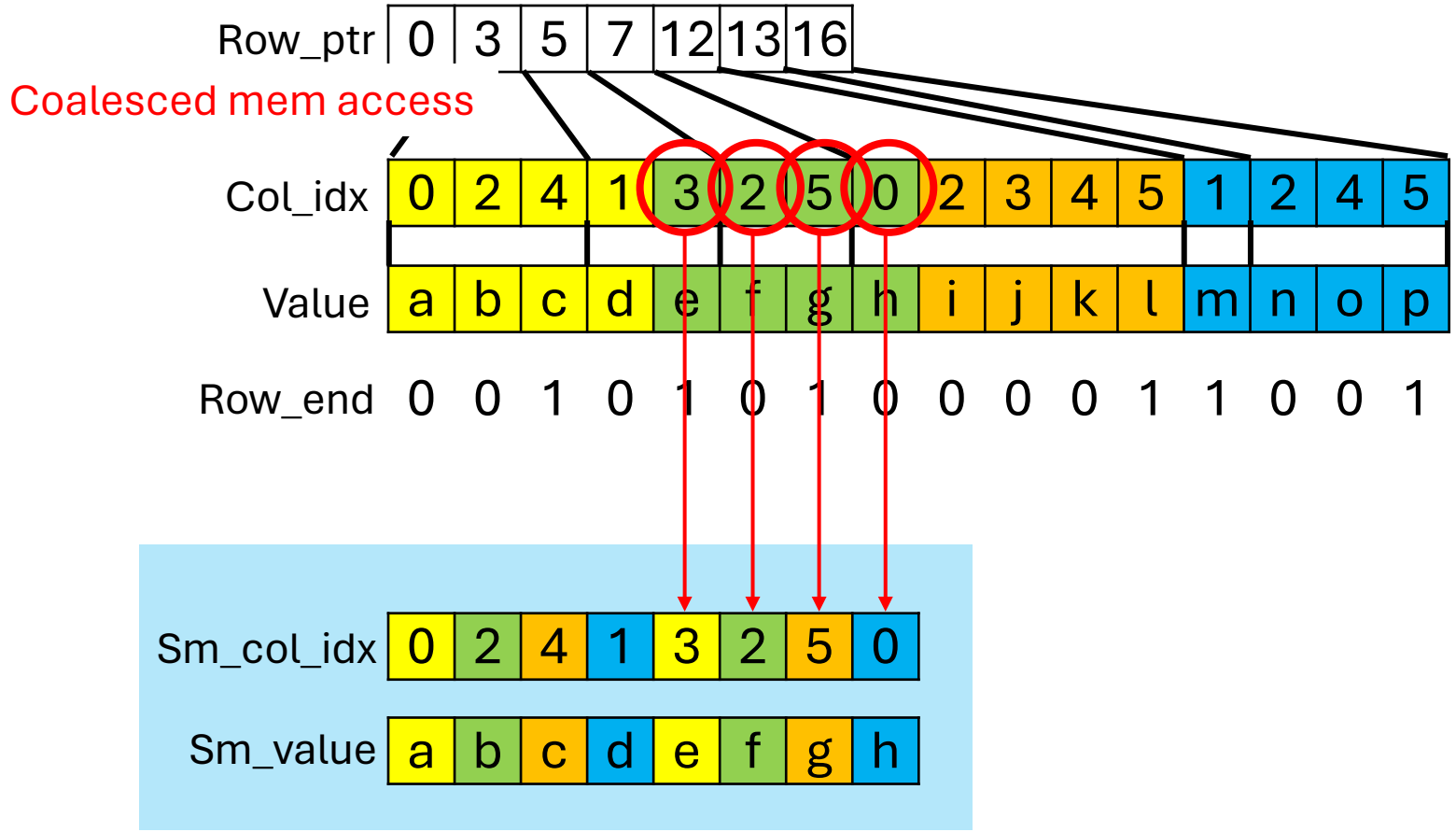
Shared memory

Uncoalesced Access to the Sparse Matrix

- Assuming one warp consists of four threads, each thread is tasked with processing four nonzero entries of the sparse matrix.

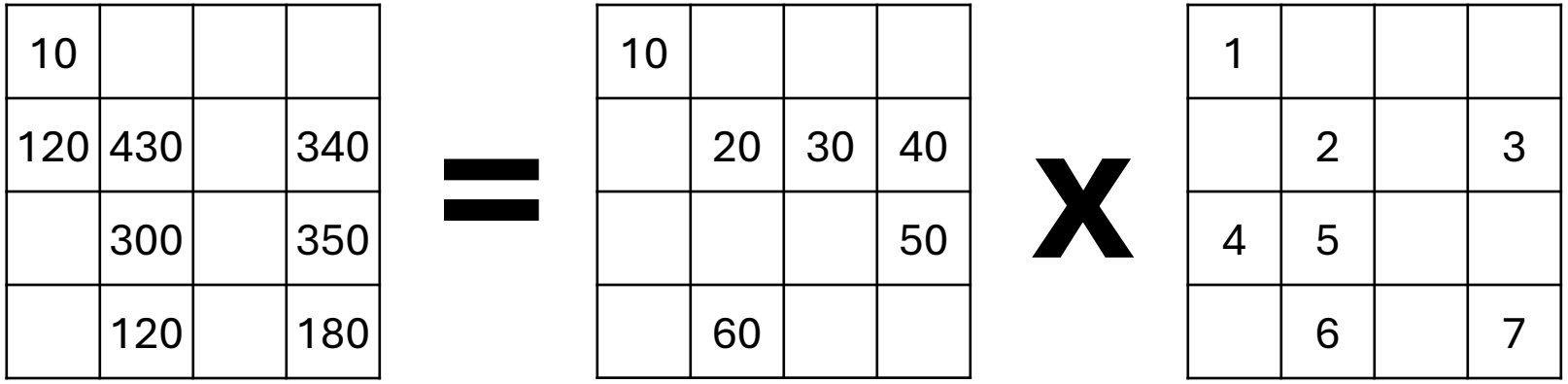


Sparse Matrix



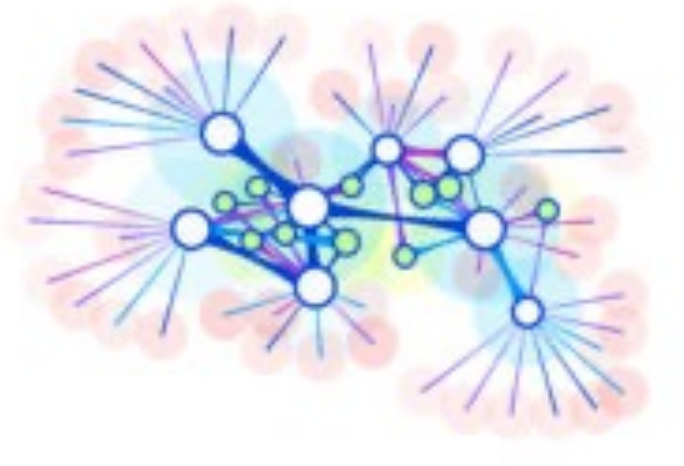
Shared memory

What is SpGEMM?

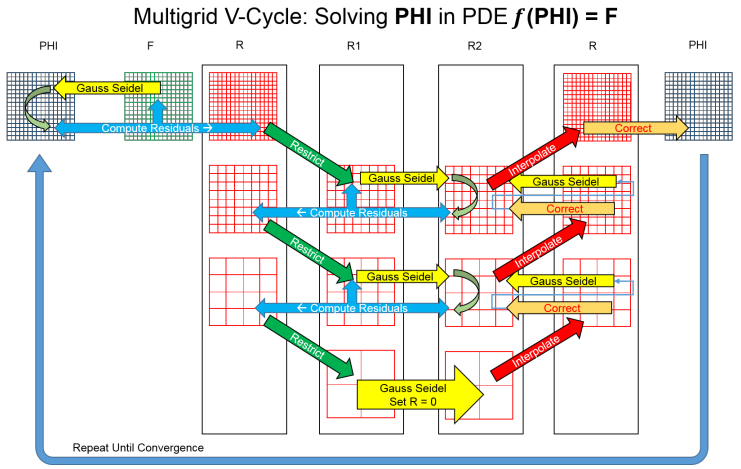


In sparse matrix-matrix multiplication (SpGEMM), all three matrices are sparse

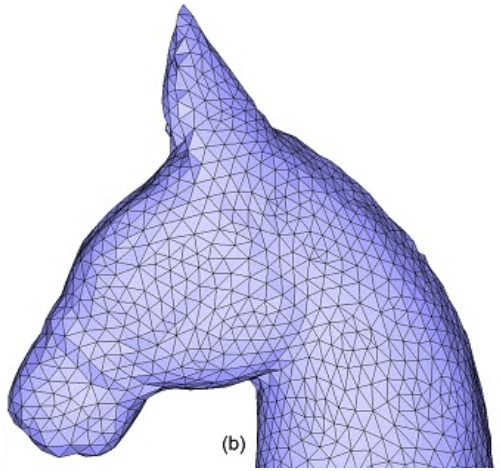
SpGEMM is ubiquitous



Graph processing



Multigrid method



Mesh operation

* https://www.datanami.com/2018/12/10/graphit-promises-big-speedup-in-graph-processing/

* https://en.wikipedia.org/wiki/Multigrid_method

* https://doc.cgal.org/latest/Polygon_mesh_processing/index.html

Three approaches for SpGEMM

- Inner-product SpGEMM
- Outer-product SpGEMM
- Gustavson’s SpGEMM

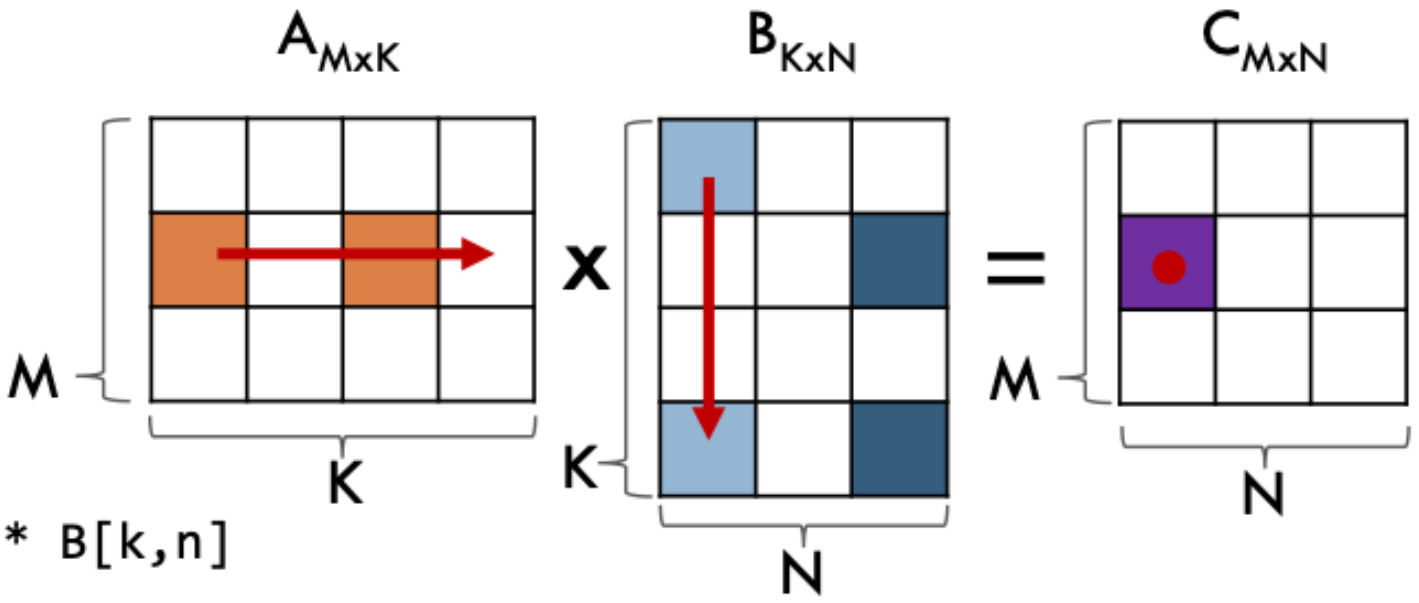
* Zhang, Guowei, et al. "Gamma: Leveraging Gustavson’s algorithm to accelerate sparse matrix multiplication." Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2021.

Inner-product SpGEMM

- Compute the output matrix one element at a time.
  - Requiring the intersection between a row of A, and a column of B.
- Offers good output reuse, but poor input reuse.
- Asymptotically very inefficient
  - For each row of A, an intersection operation is necessary for every nonzero column of B.
    - But, most intersections will result in an empty set.

Inner-product dataflow

```
for m in [0, M)
 for n in [0, N)
  for k in [0, K)
   C[m,n] += A[m,k] * B[k,n]
```



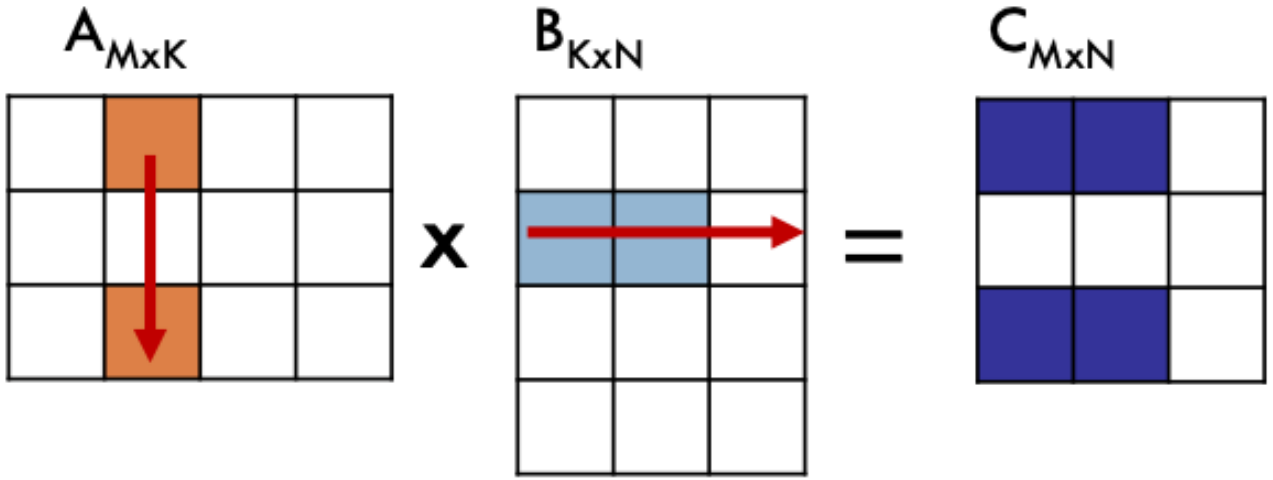
* Zhang, Guowei, et al. "Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication." Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2021.

Outer-product SpGEMM

- Compute the output one partial matrix at a time by traversing each column of A and each row of B.
  - Sums the outer products of corresponding columns and rows.
- Offers good input reuse, but poor output reuse.
  - Unordered access across different rows and columns of the output matrix.
    - The output is primarily required to be in GPU global memory due to poor locality.
    - Global memory access for each partial product.

Outer-product dataflow

```
for k in [0, K)
 for m in [0, M)
  for n in [0, N)
   C[m,n] += A[m,k] * B[k,n]
```



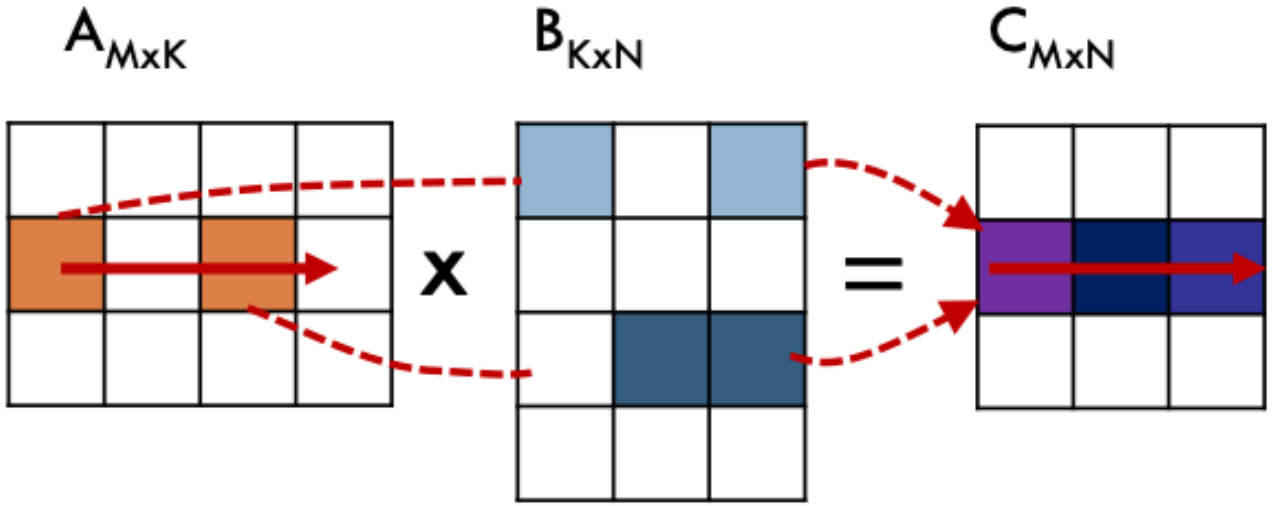
* Zhang, Guowei, et al. "Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication." Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2021.

Gustavson's SpGEMM

- Computes the output matrix one row at a time by summing the rows of B scaled by the corresponding columns in each row of A.
- Requires combining partial output rows instead of partial output matrices, as in outer-product SpGEMM.
- Modest reuse of input and output
- Allows for consistency in the format for both inputs and outputs, meaning all formats are CSR.
  - Inner- or outer-product requires one input to be transposed (i.e., transposed CSR).

Gustavson dataflow

```
for m in [0, M)
 for k in [0, K)
  for n in [0, N)
   C[m,n] += A[m,k] * B[k,n]
```



* Zhang, Guowei, et al. "Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication." Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2021.

Three approaches for SpGEMM

- Inner-product SpGEMM
- Outer-product SpGEMM
- Gustavson's SpGEMM

Adopted in all state-of-the-art GPU implementations

* Zhang, Guowei, et al. "Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication." Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2021.

Optimizing SpGEMM on GPUs is challenging

- Concurrent access of the output.
  - The sparse output matrix needs to be constructed in parallel.
  - The output size is unknown a priori.

- Accumulating partial products.
  - Accumulating partial products in global memory significantly hurts performance.
    - Causes uncoalesced atomic memory accesses.

- Load balancing.
  - All matrices are irregular.
  - Achieving load-balanced execution in SpGEMM is significantly more challenging compared to SpMV.

* Parger, Mathias, et al. "Speck: Accelerating gpu sparse matrix-matrix multiplication through lightweight analysis." Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2020.

Optimizing SpGEMM on GPUs is challenging

- Concurrent access of the output.
  - The sparse output matrix needs to be constructed in parallel.
  - The output size is unknown a priori.

- Accumulating partial products.
  - Accumulating partial products in global memory significantly hurts performance.
    - Causes uncoalesced atomic memory accesses.

- Load balancing.
  - All matrices are irregular.
  - Achieving load-balanced execution in SpGEMM is significantly more challenging compared to SpMV.

* Parger, Mathias, et al. "Speck: Accelerating gpu sparse matrix-matrix multiplication through lightweight analysis." Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2020.

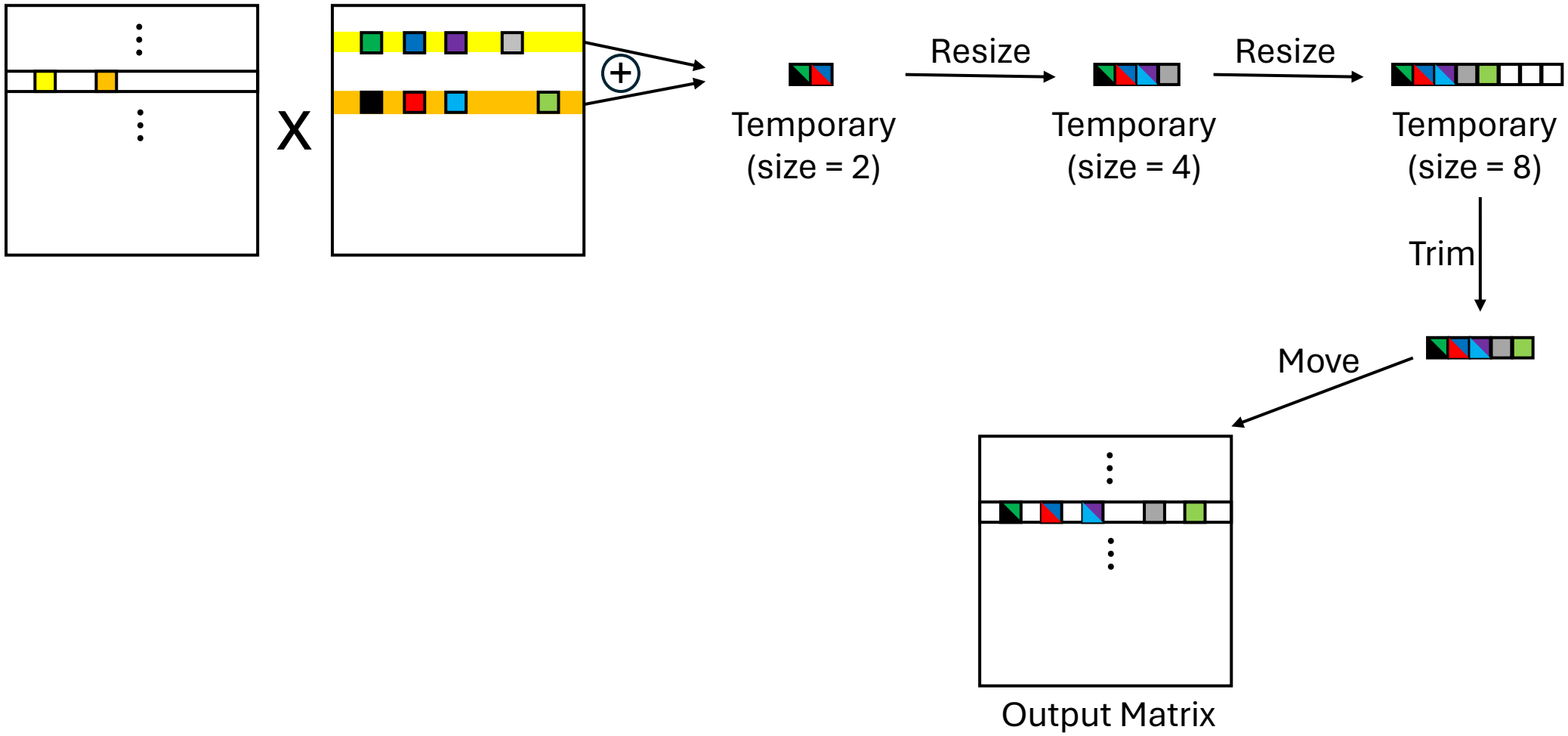
Methods for Memory Pre-allocation for the Output Matrix*

- Progressive method
- Upper-bound method
- Probabilistic method
- Two-phase method (Precise method)

* Liu, Weifeng, and Brian Vinter. "An efficient GPU general sparse matrix-matrix multiplication for irregular data." *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014.

Progressive Method

- First allocates memory of a proper size, starts sparse matrix computation and reallocates the buffer if larger space is required.



Progressive Method

- First allocates memory of a proper size, starts sparse matrix computation and reallocates the buffer if larger space is required.

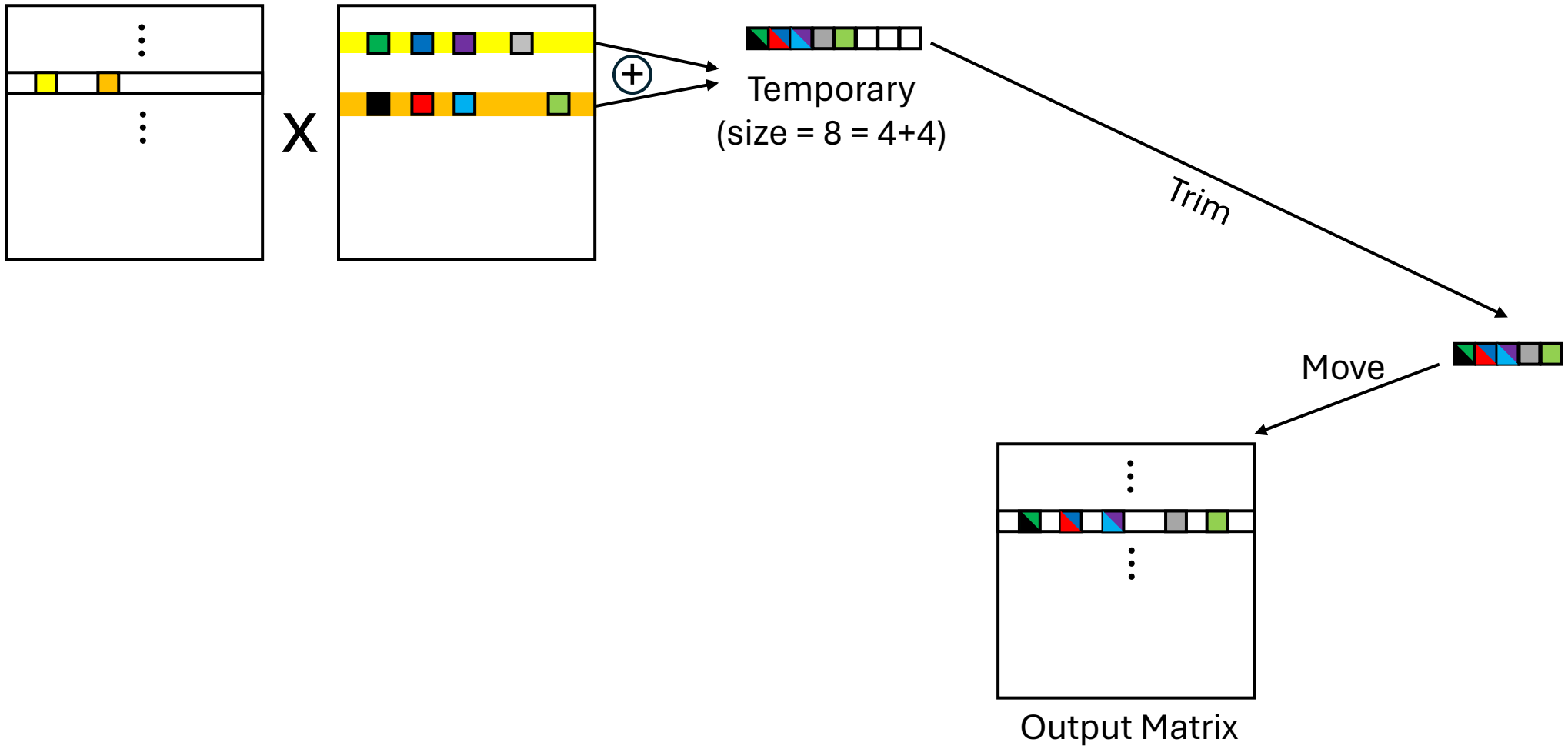
- Concurrent memory management over hundreds of thousands of threads is challenging.

- Reallocation of device memory on the fly during computations is difficult.

- Memory space is wasted up to k times (k is an expansion factor).

Upper-bound Method

- Computes an upper bound of the number of the nonzero entries in the output matrix and allocates corresponding memory space.



Upper-bound Method

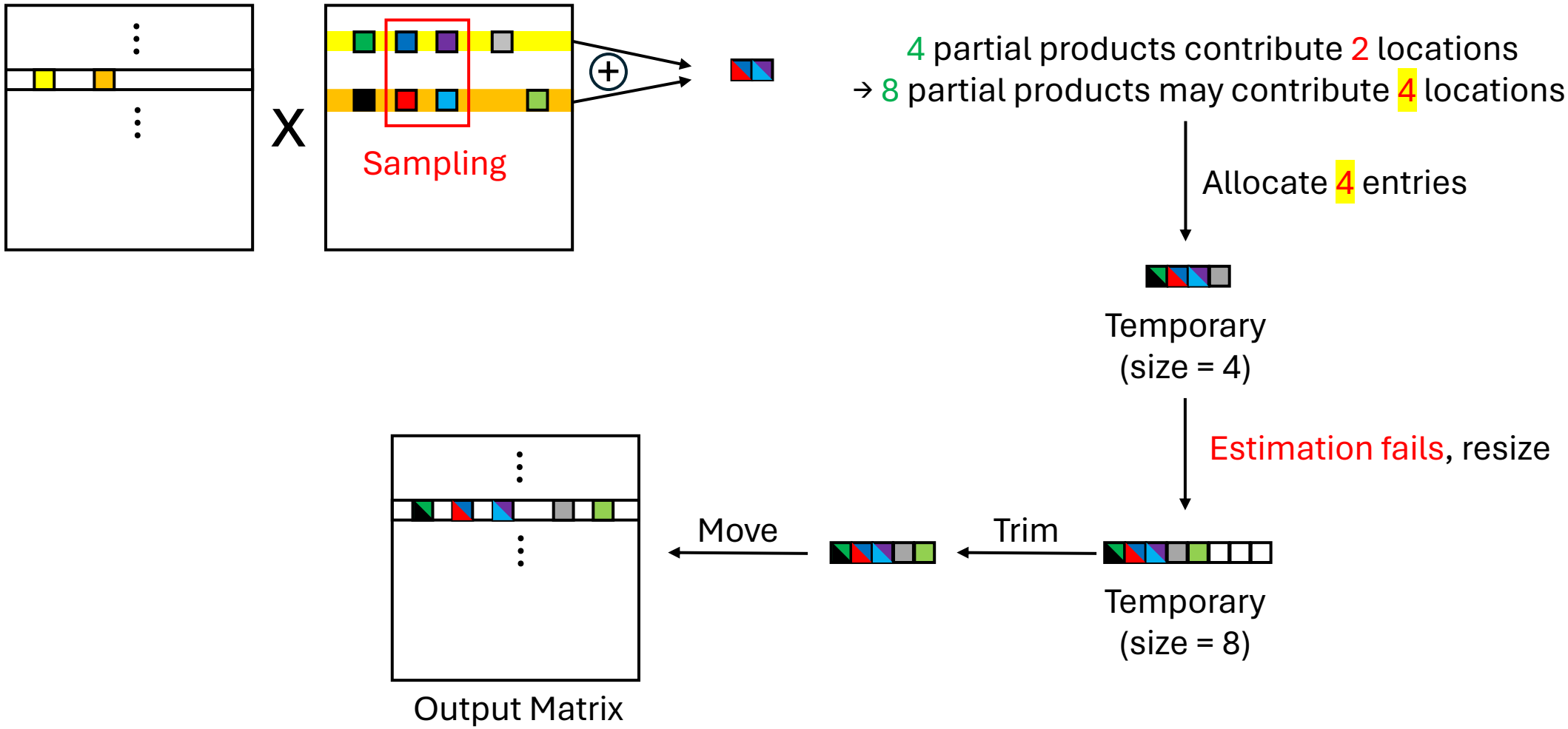
- Computes an upper bound of the number of the nonzero entries in the output matrix and allocates corresponding memory space.

- Can significantly waste memory space
  - The size of GPU memory is relatively small (e.g., 80GB for an A100 GPU)

- Memory bandwidth can be wasted
  - GPU transaction granularity is 32/64 bytes

Probabilistic Method

- Estimates an imprecise size of the output based on random sampling and probability analysis on the input matrices.



Probabilistic Method

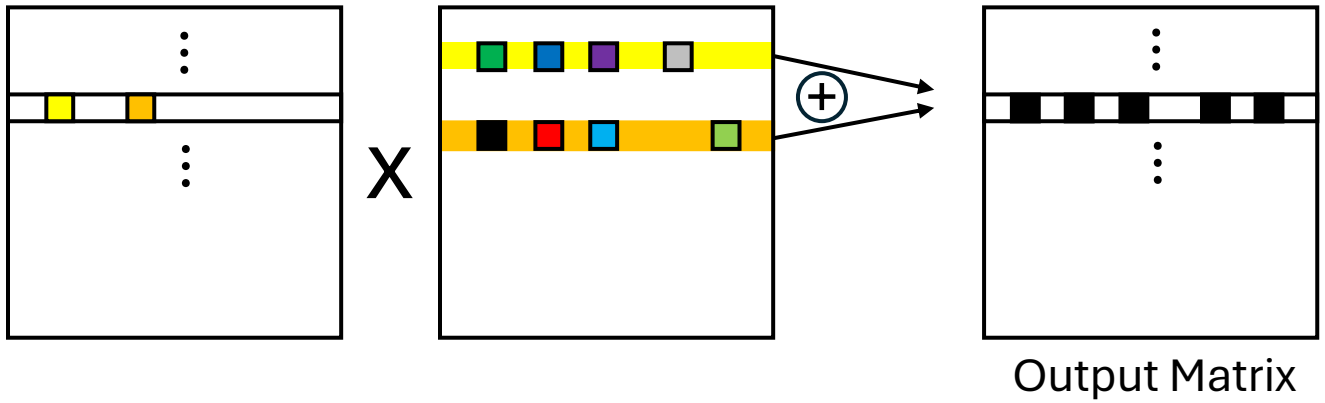
- Estimates an imprecise size of the output based on random sampling and probability analysis on the input matrices.

- Precisely estimating the upper bound of the output size is very challenging.
  - Interactions between A and B are complicated: estimating which intermediate results contribute to the same entry of the output is difficult.

- Extra memory has to be allocated while the estimation fails.

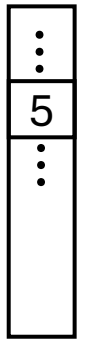
Two-phase method

- In the symbolic phase, count the number of nonzero entries of each row of the output
- In the numeric phase, compute the column indices and values of the entries of the output

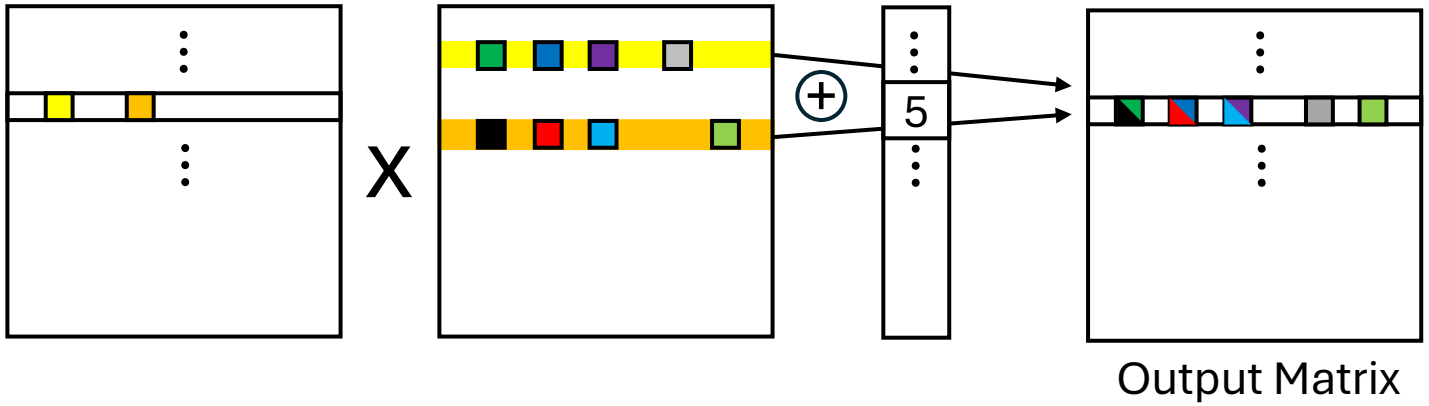


Count the number of entries of the row in the output
w/o actual computations

Generate



Allocate output



Two-phase method

- In the symbolic phase, count the number of nonzero entries of each row of the output
- In the numeric phase, compute the column indices and values of the entries of the output

- The significant overhead arises from having the same pattern of computations twice.

- Memory-space efficiency is achieved, ensuring that memory space is not wasted.

- All state-of-the-art SpGEMM implementations on GPUs utilize this method.

Progressive Method

- Progressive method

- Upper-bound method

- Probabilistic method

- Two-phase method (Precise method)

  Adopted in all state-of-the-art GPU implementations

* Liu, Weifeng, and Brian Vinter. "An efficient GPU general sparse matrix-matrix multiplication for irregular data." *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014.

Progressive Method

- Progressive method

  This may outperform the two-phase method on GPUs if we have enough GPU memory

- Upper-bound method

- Probabilistic method

- Two-phase method (Precise method)

  Adopted in all state-of-the-art GPU implementations

* Liu, Weifeng, and Brian Vinter. "An efficient GPU general sparse matrix-matrix multiplication for irregular data." *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014.

Optimizing SpGEMM on GPUs is challenging

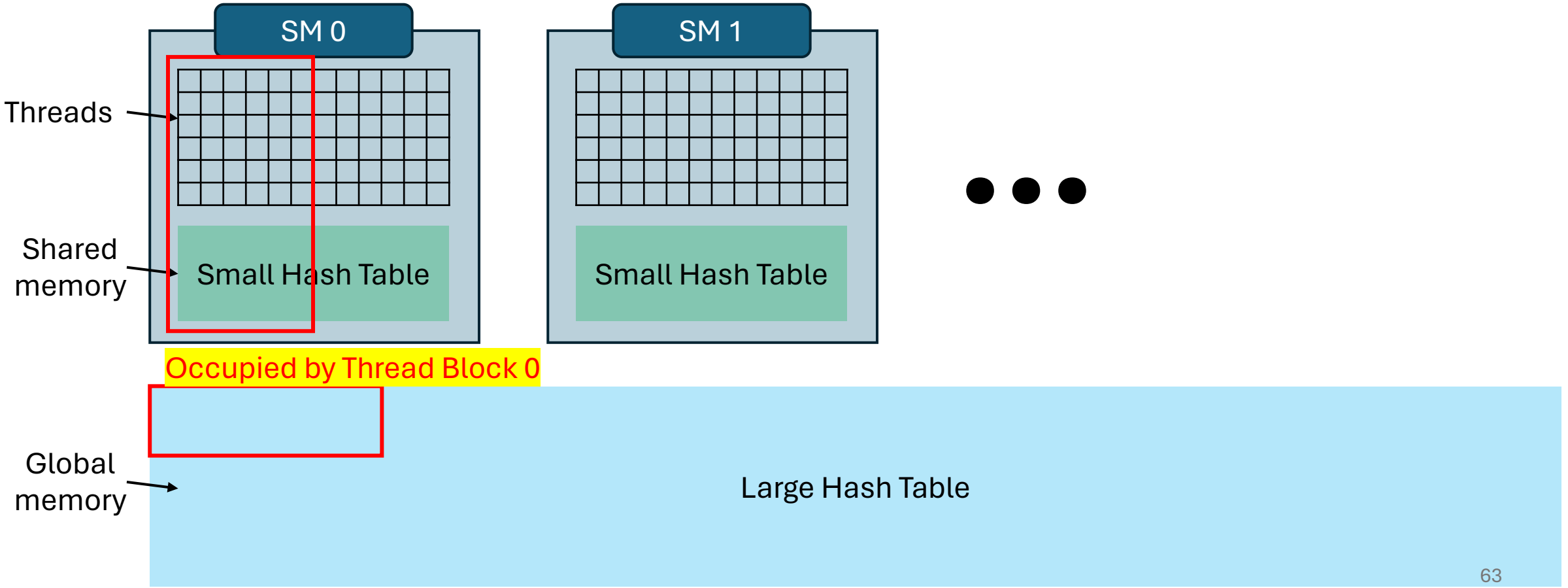
- Parallel assembly for the output.
  - The sparse output matrix needs to be constructed in parallel.
  - The output size is unknown a priori.

- Accumulating partial products.
  - Accumulating partial products in global memory significantly hurts performance.
    - Causes uncoalesced atomic memory accesses.

- Load balancing.
  - All matrices are irregular.
  - Achieving load-balanced execution in SpGEMM is significantly more challenging compared to SpMV.

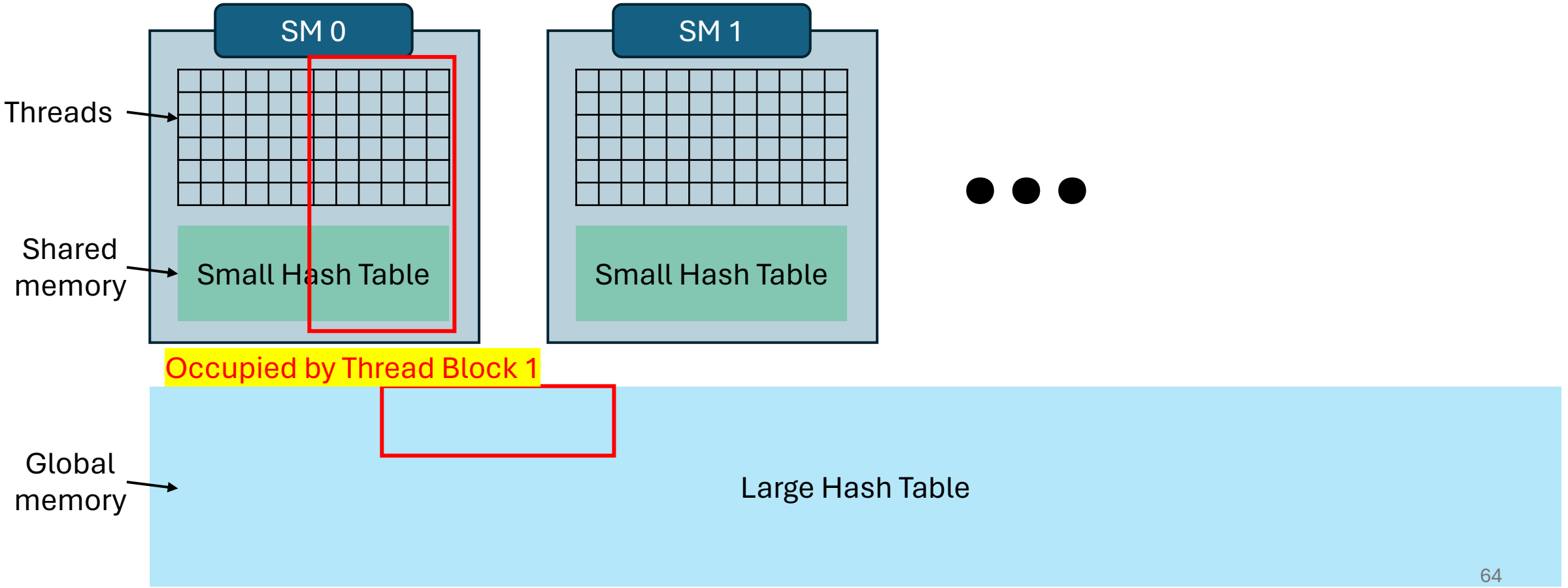
Accumulation using Hashmaps

- For brevity, each thread block processes each row of the output matrix.
- Each thread block has two (hierarchical) hash tables in shared memory and global memory.
- Utilize a small hash table initially. If the small hash table is unable to contain all entries, transfer all data from the small hash table and rely solely on the large hash table.



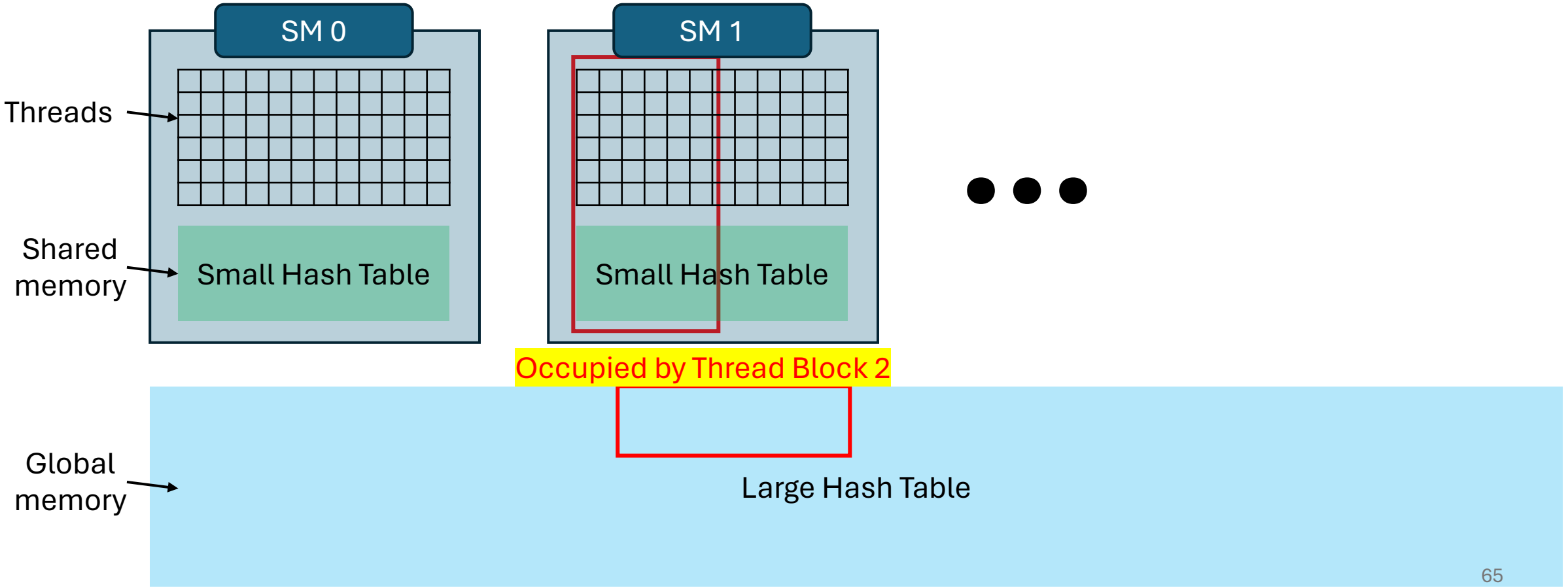
Accumulation using Hashmaps

- For brevity, each thread block processes each row of the output matrix.
- Each thread block has two (hierarchical) hash tables in shared memory and global memory.
- Utilize a small hash table initially. If the small hash table is unable to contain all entries, transfer all data from the small hash table and rely solely on the large hash table.



Accumulation using Hashmaps

- For brevity, each thread block processes each row of the output matrix.
- Each thread block has two (hierarchical) hash tables in shared memory and global memory.
- Utilize a small hash table initially. If the small hash table is unable to contain all entries, transfer all data from the small hash table and rely solely on the large hash table.



Accumulation using Hashmaps

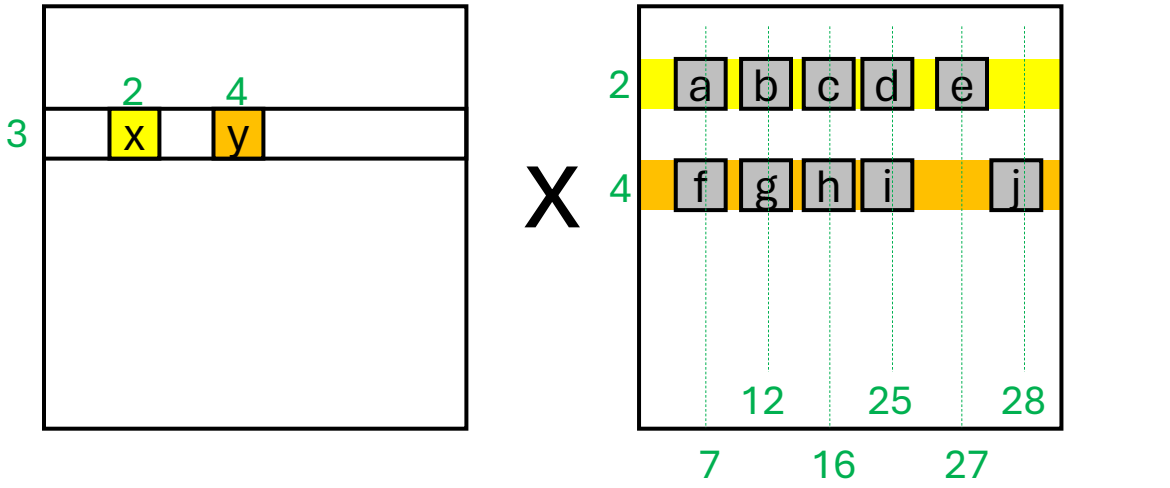
- During the Symbolic phase, the number of nonzero entries in each row of the output remains unknown.
  - Hash table size = the number of partial products (representing the upper bound).
  - Store only the nonzero column indices in hash tables.

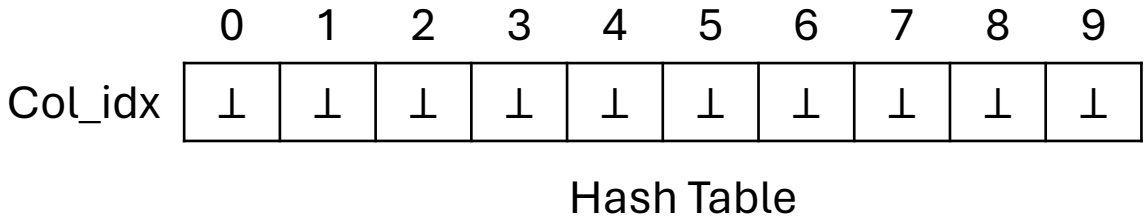
- In the Numeric phase, we gain knowledge about the number of nonzero entries in each output row.
  - Hash table size = 1.5 × the number of nonzero entries.
  - Gathering nonzero entries of the hash table is achieved through parallel prefix-sum operations.
  - Following the gathering process, sorting becomes necessary to arrange nonzero entries of an output row in ascending order of column indices.
  - Store both the nonzero column indices and their corresponding values in hash tables.
    - Need more space for hash tables

- Linear probing is used

Accumulation using Hashmaps

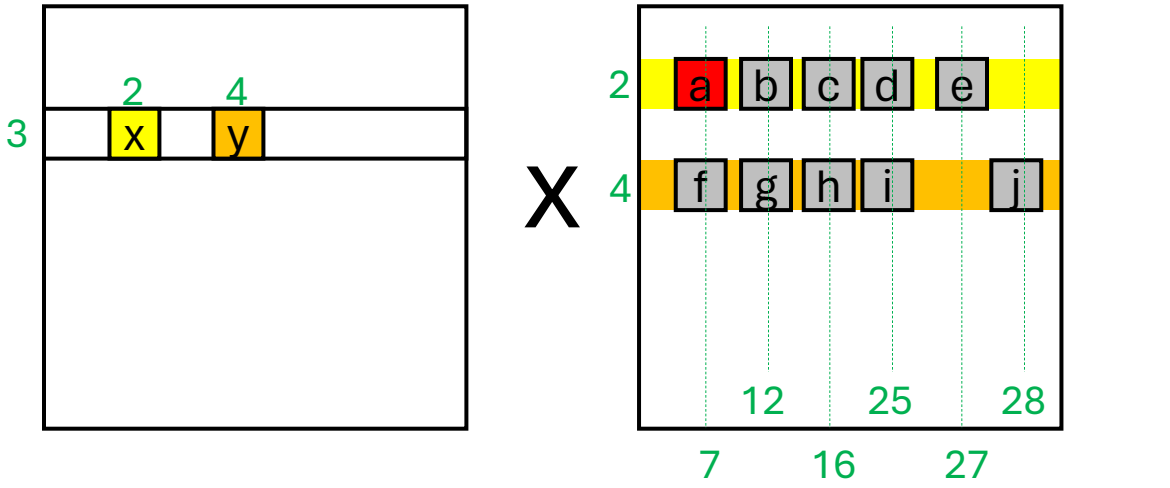
- In Symbolic phase,
  - Hash table size = the number of partial products = 5+5 = 10
  - Only maintain column indices
  - Assume hash function h = (h+1)%10 in case of collisions.

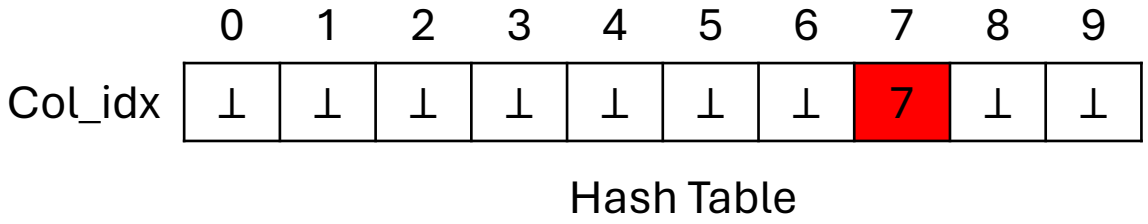




Accumulation using Hashmaps

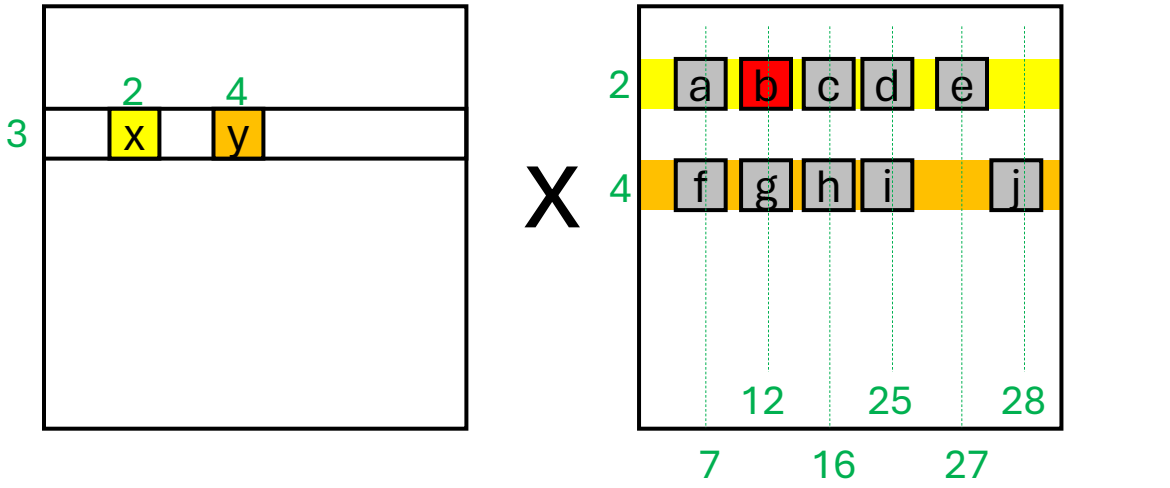
- In Symbolic phase,
  - Hash table size = the number of partial products = 5+5 = 10
  - Only maintain column indices
  - Assume hash function h = (h+1)%10 in case of collisions.

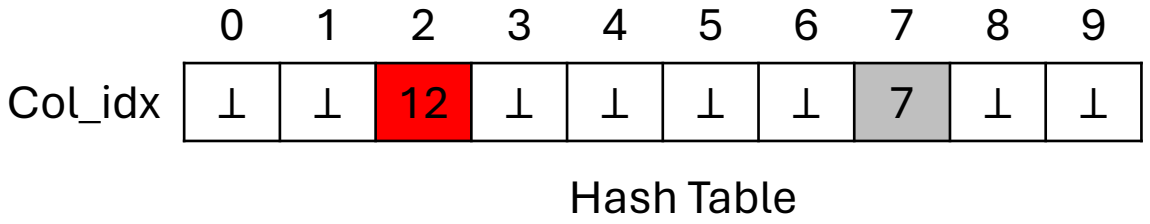




Accumulation using Hashmaps

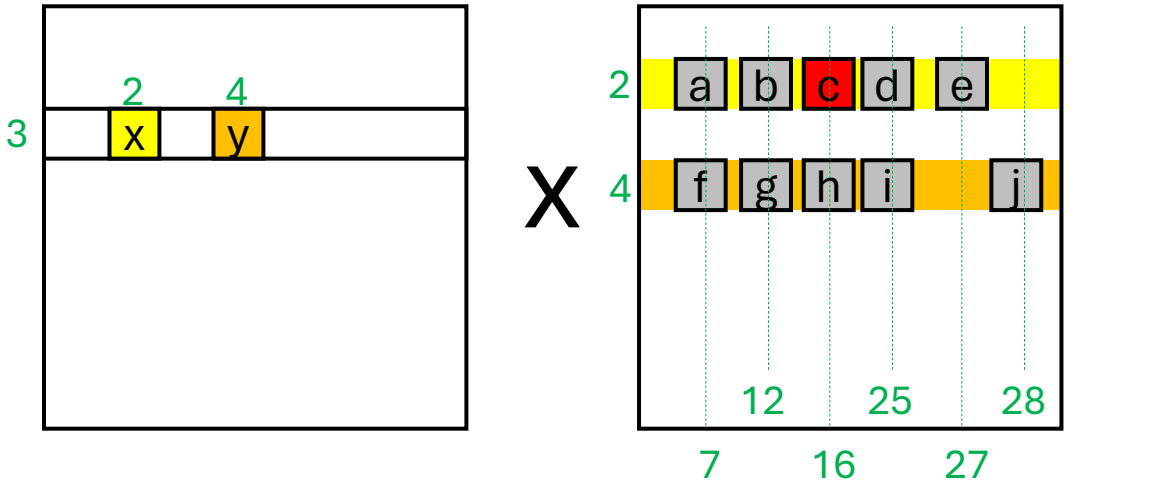
- In Symbolic phase,
  - Hash table size = the number of partial products = 5+5 = 10
  - Only maintain column indices
  - Assume hash function h = (h+1)%10 in case of collisions.

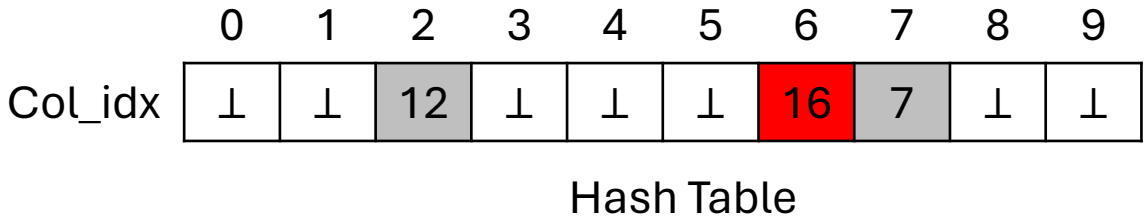




Accumulation using Hashmaps

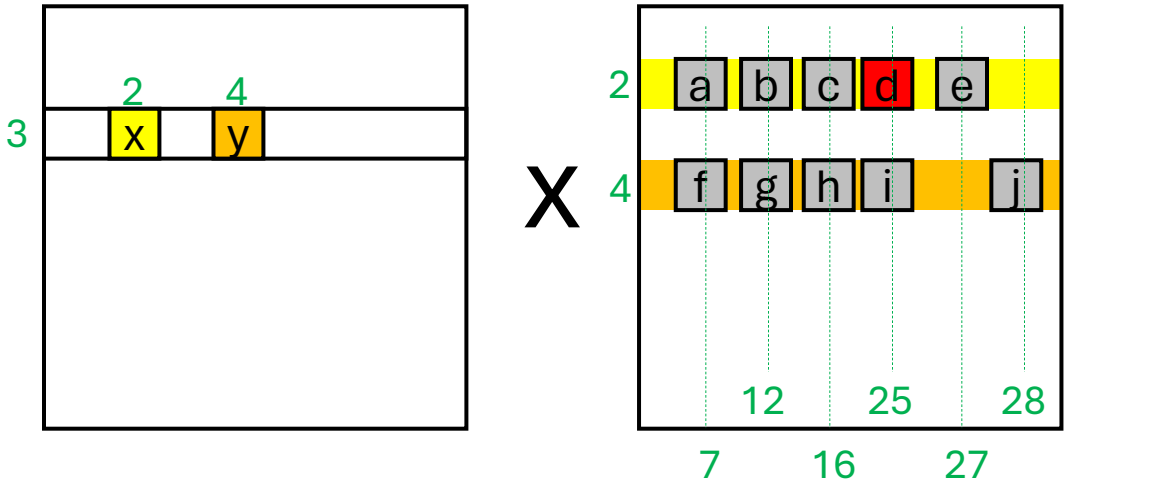
- In Symbolic phase,
  - Hash table size = the number of partial products = 5+5 = 10
  - Only maintain column indices
  - Assume hash function h = (h+1)%10 in case of collisions.

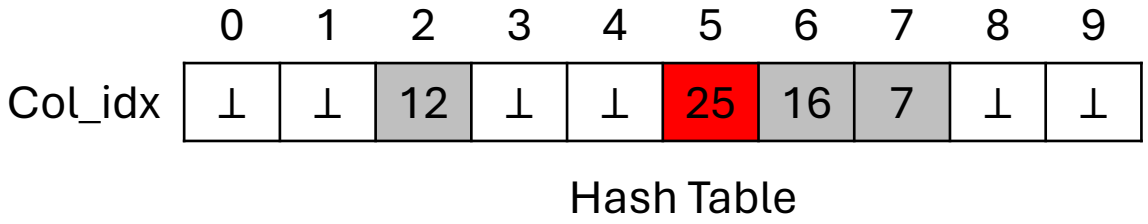




Accumulation using Hashmaps

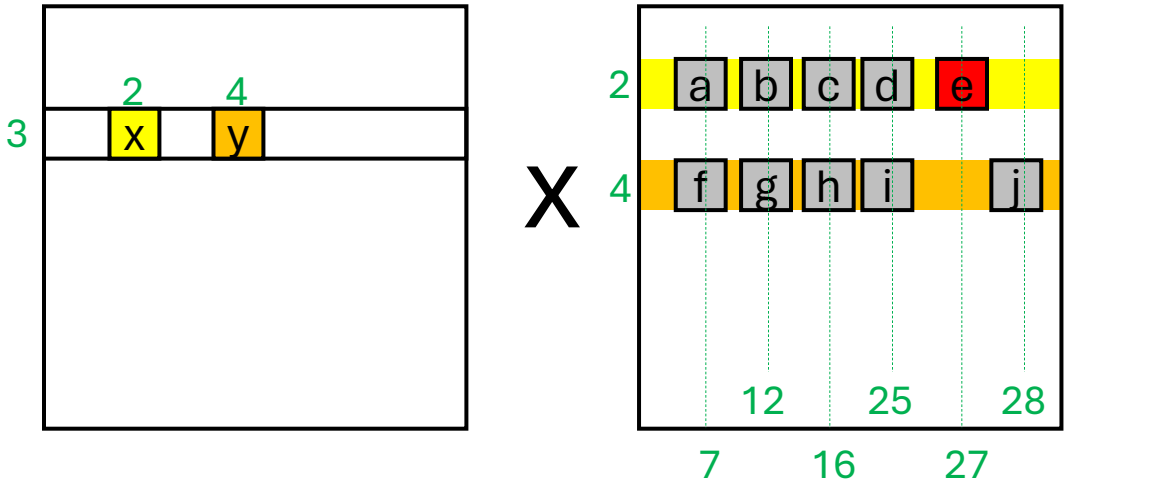
- In Symbolic phase,
  - Hash table size = the number of partial products = 5+5 = 10
  - Only maintain column indices
  - Assume hash function h = (h+1)%10 in case of collisions.

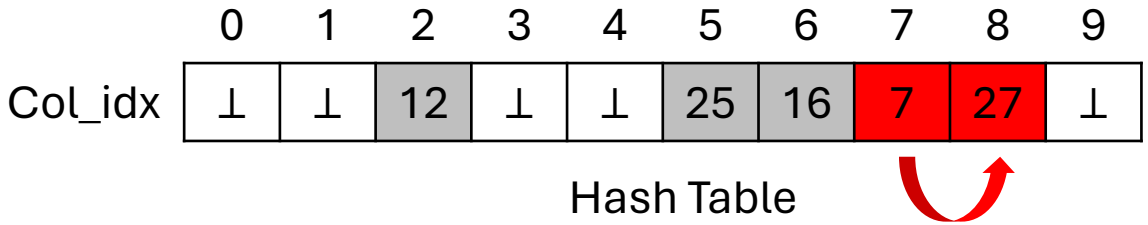




Accumulation using Hashmaps

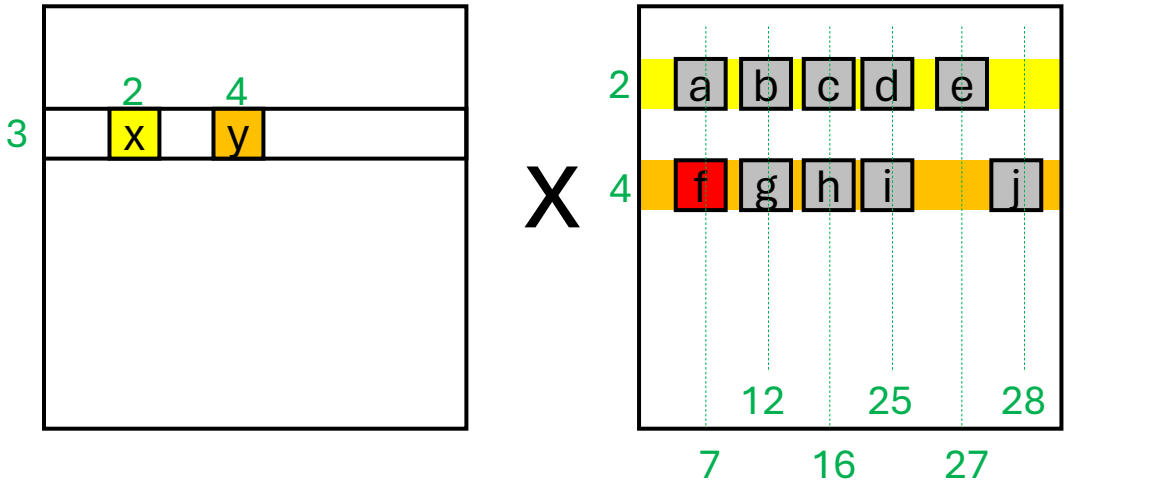
- In Symbolic phase,
  - Hash table size = the number of partial products = 5+5 = 10
  - Only maintain column indices
  - Assume hash function h = (h+1)%10 in case of collisions.

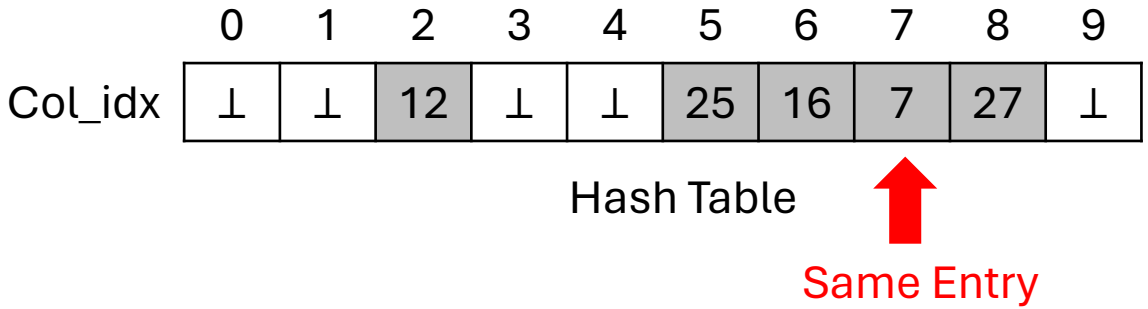




Accumulation using Hashmaps

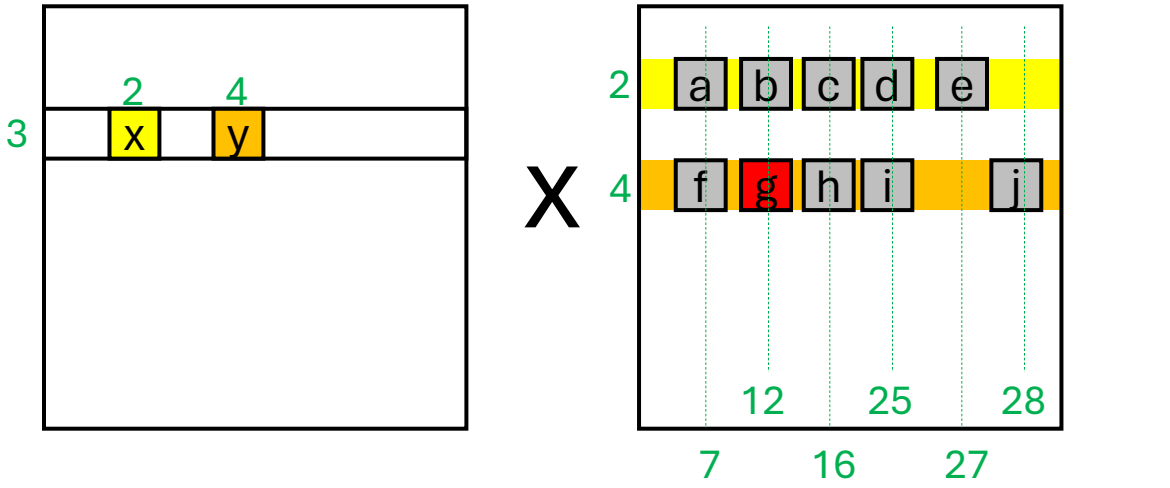
- In Symbolic phase,
  - Hash table size = the number of partial products = 5+5 = 10
  - Only maintain column indices
  - Assume hash function h = (h+1)%10 in case of collisions.

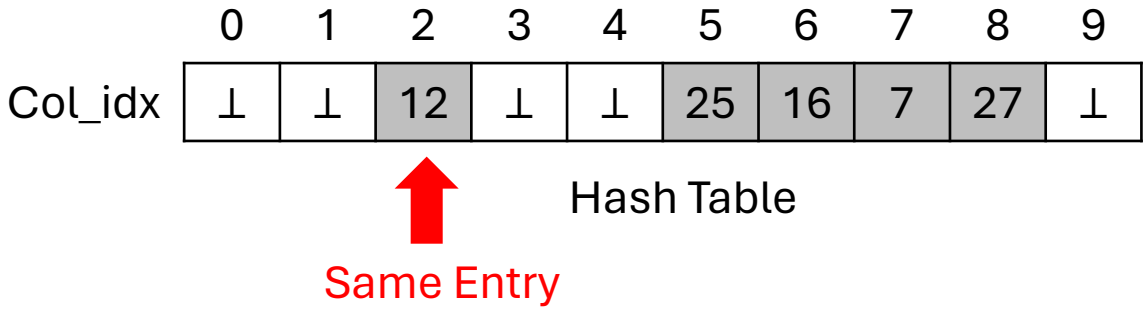




Accumulation using Hashmaps

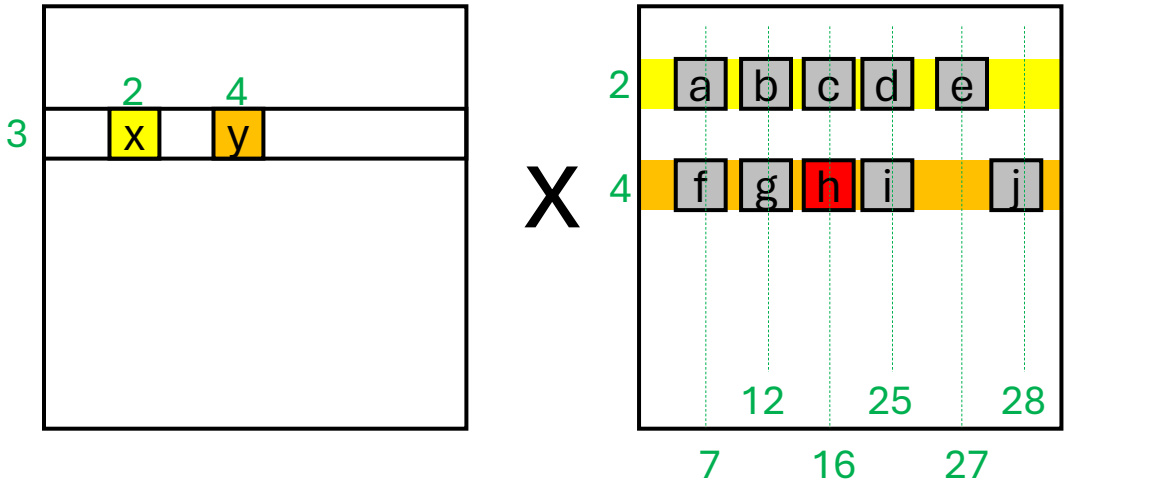
- In Symbolic phase,
  - Hash table size = the number of partial products = 5+5 = 10
  - Only maintain column indices
  - Assume hash function h = (h+1)%10 in case of collisions.

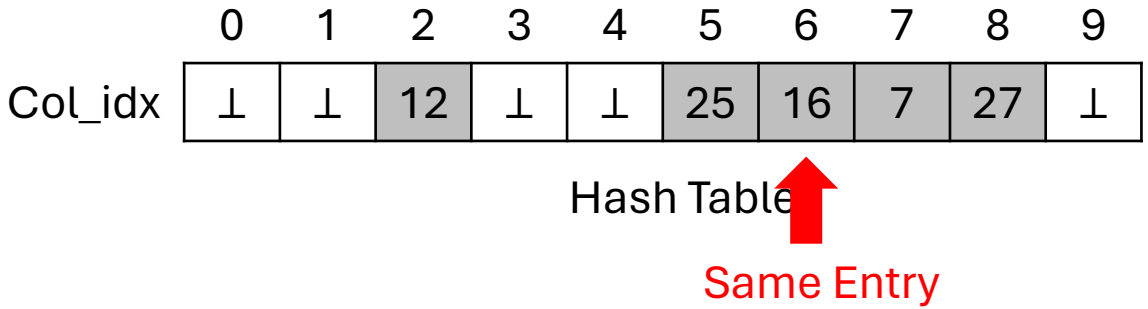




Accumulation using Hashmaps

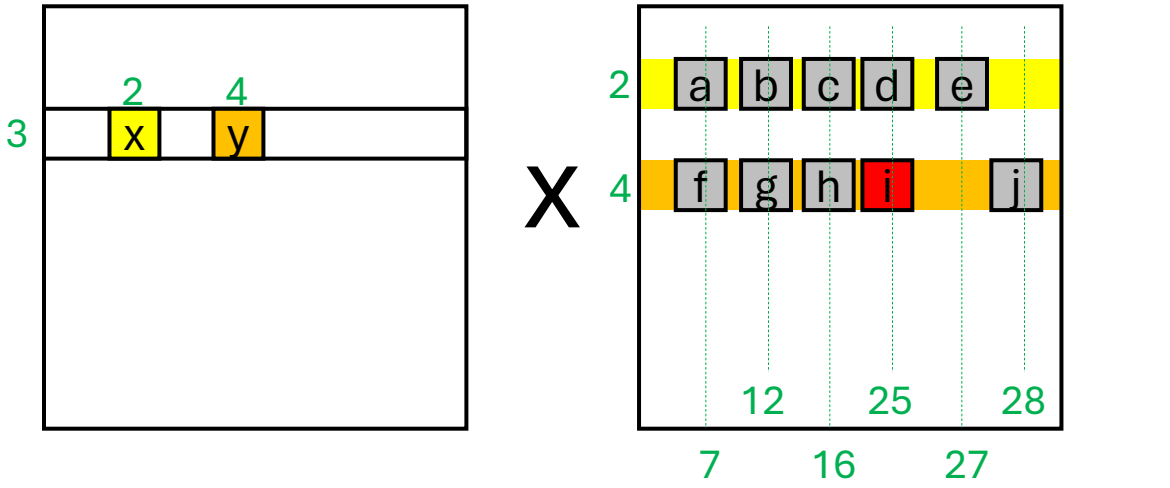
- In Symbolic phase,
  - Hash table size = the number of partial products = 5+5 = 10
  - Only maintain column indices
  - Assume hash function h = (h+1)%10 in case of collisions.

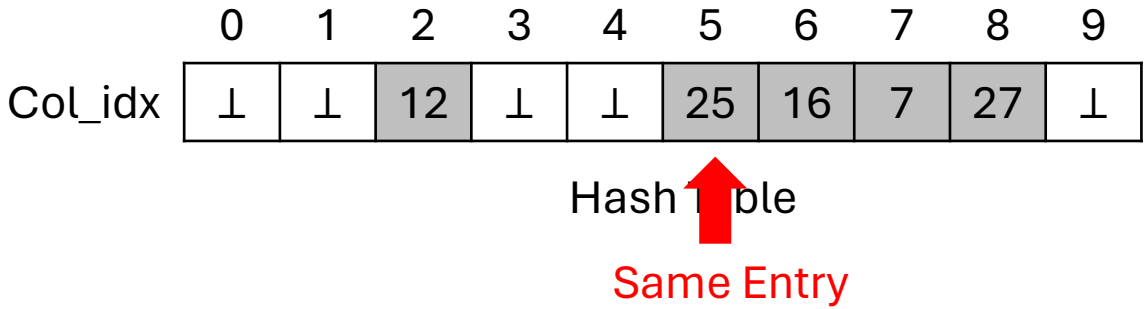




Accumulation using Hashmaps

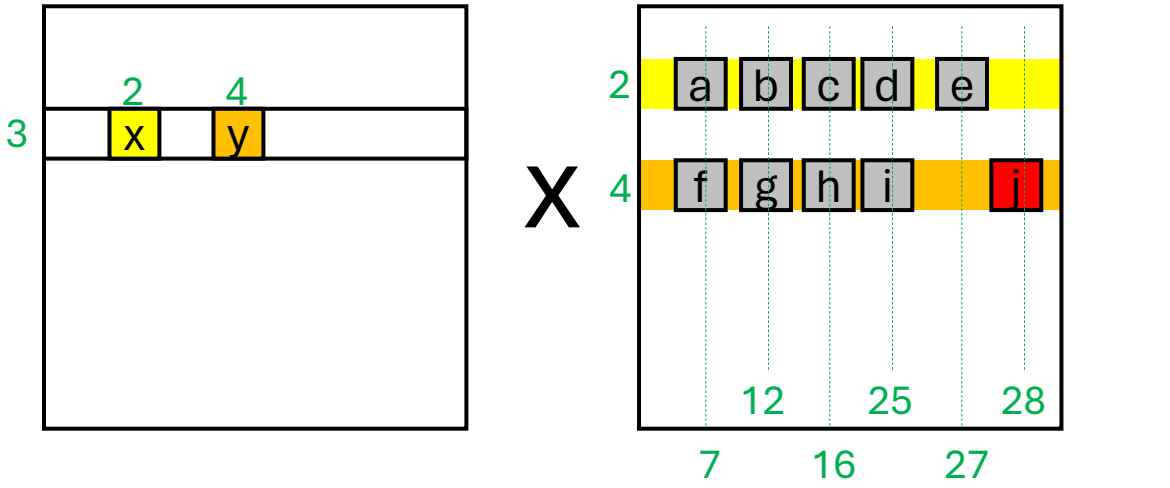
- In Symbolic phase,
  - Hash table size = the number of partial products = 5+5 = 10
  - Only maintain column indices
  - Assume hash function h = (h+1)%10 in case of collisions.

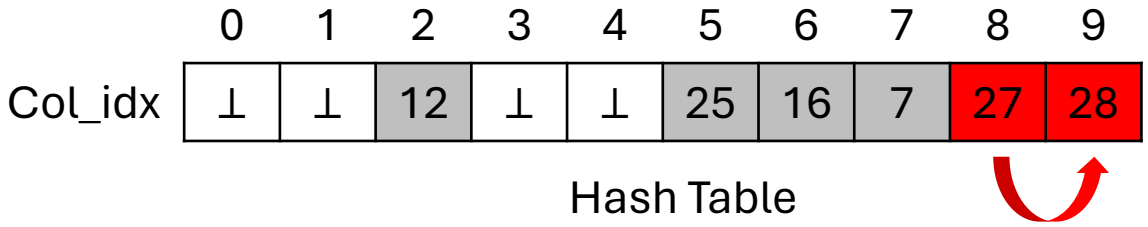




Accumulation using Hashmaps

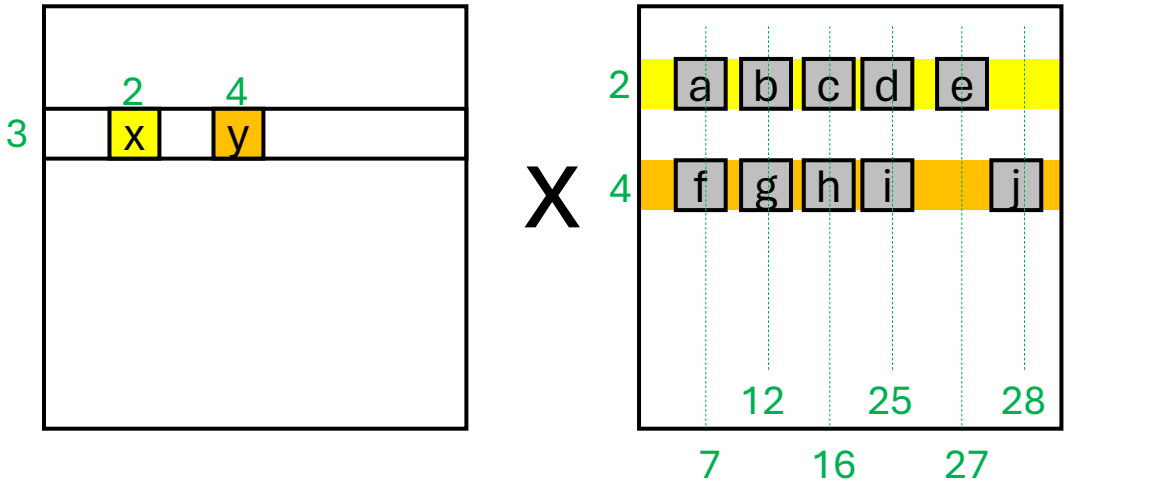
- In Symbolic phase,
  - Hash table size = the number of partial products = 5+5 = 10
  - Only maintain column indices
  - Assume hash function h = (h+1)%10 in case of collisions.





Accumulation using Hashmaps

- In Numeric phase,
  - Hash table size = 1.5 ⨉ the number of nonzero entries = 1.5 ⨉ 6 = 9.
  - Maintain both column indices and corresponding values.
  - Assume hash function h = (h+1)%9 in case of collisions.

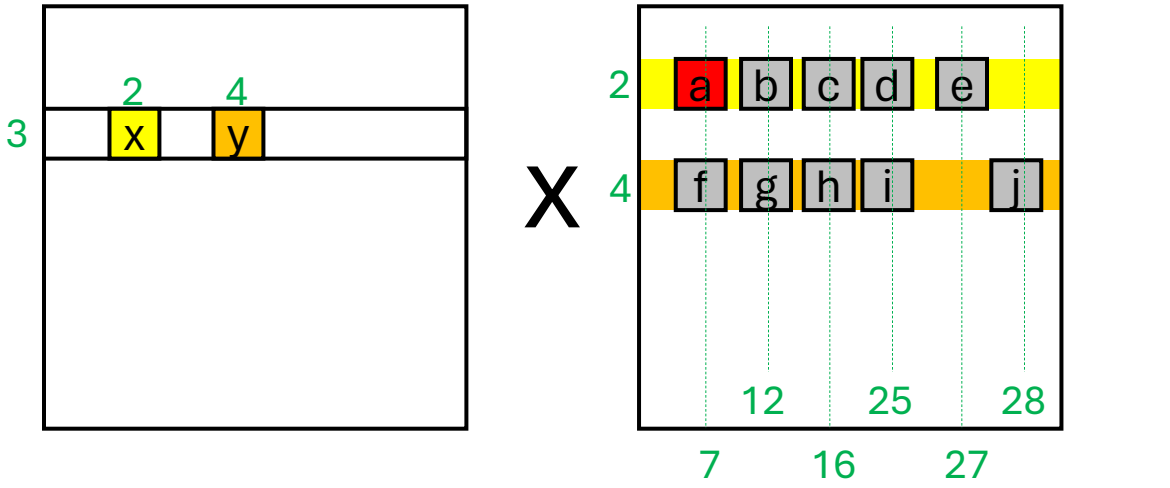


	0	1	2	3	4	5	6	7	8
Col_idx	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
Value	0	0	0	0	0	0	0	0	0

Hash Table

Accumulation using Hashmaps

- In Numeric phase,
  - Hash table size = 1.5 ⨉ the number of nonzero entries = 1.5 ⨉ 6 = 9.
  - Maintain both column indices and corresponding values.
  - Assume hash function h = (h+1)%9 in case of collisions.

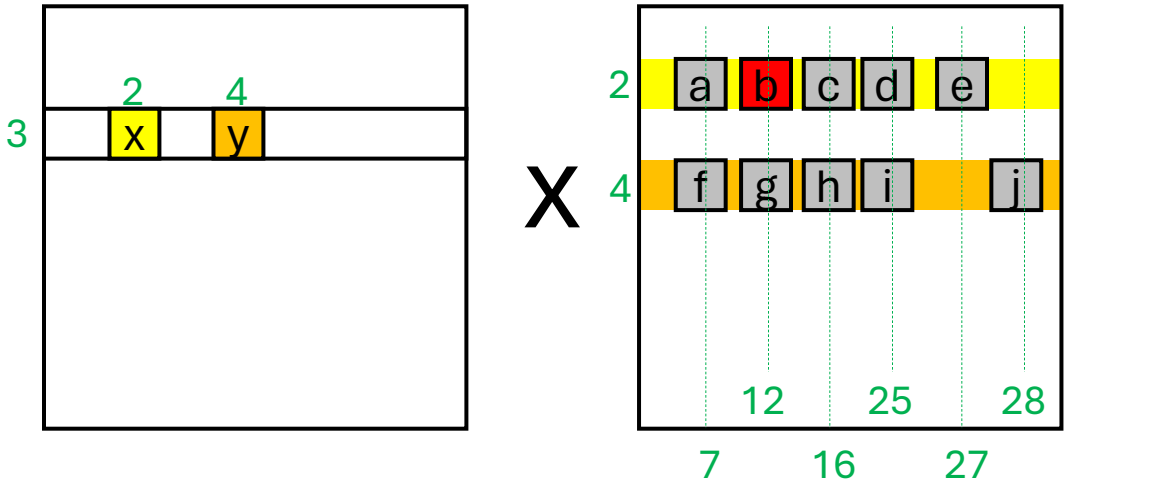


	0	1	2	3	4	5	6	7	8
Col_idx	⊥	⊥	⊥	⊥	⊥	⊥	⊥	7	⊥
Value	0	0	0	0	0	0	0	ax	0

Hash Table

Accumulation using Hashmaps

- In Numeric phase,
  - Hash table size = 1.5 ⨉ the number of nonzero entries = 1.5 ⨉ 6 = 9.
  - Maintain both column indices and corresponding values.
  - Assume hash function h = (h+1)%9 in case of collisions.

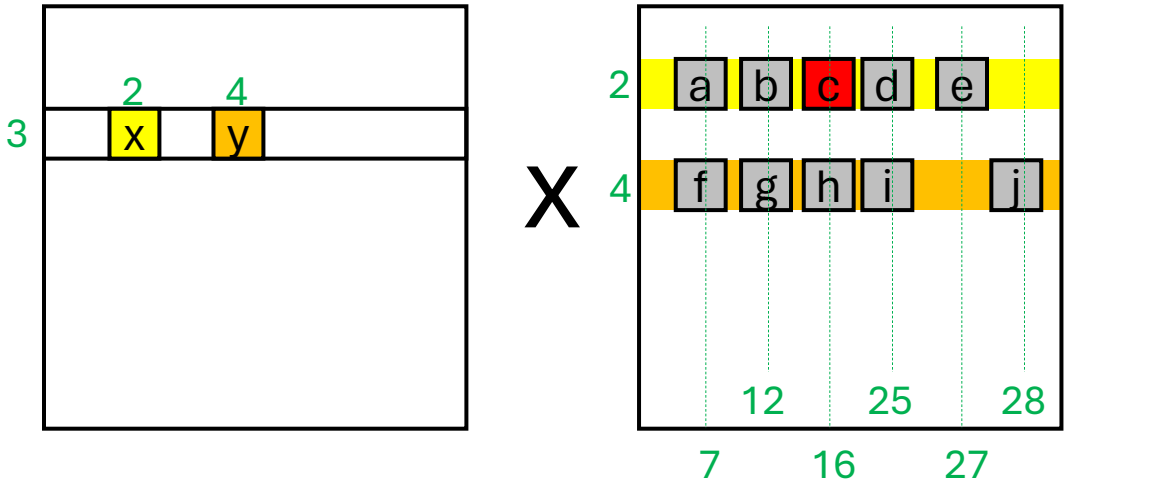


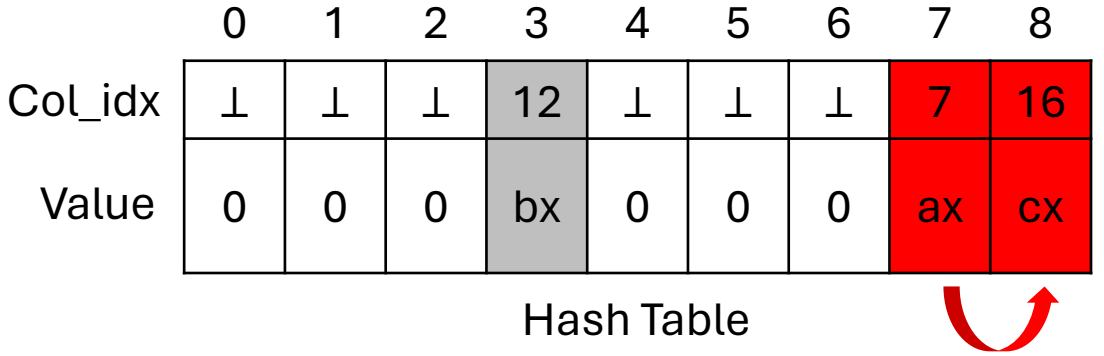
	0	1	2	3	4	5	6	7	8
Col_idx	⊥	⊥	⊥	12	⊥	⊥	⊥	7	⊥
Value	0	0	0	bx	0	0	0	ax	0

Hash Table

Accumulation using Hashmaps

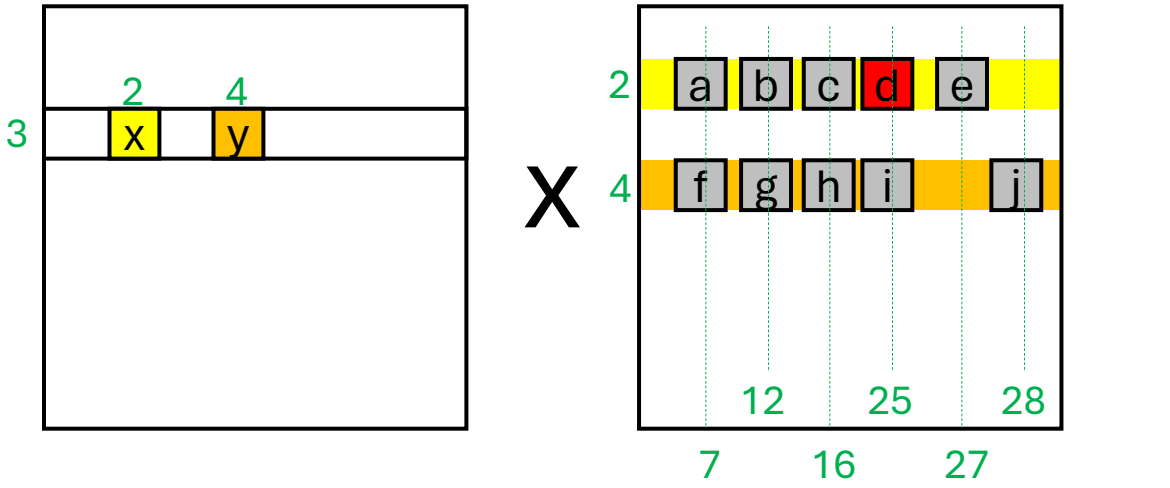
- In Numeric phase,
  - Hash table size = 1.5 ⨉ the number of nonzero entries = 1.5 ⨉ 6 = 9.
  - Maintain both column indices and corresponding values.
  - Assume hash function h = (h+1)%9 in case of collisions.

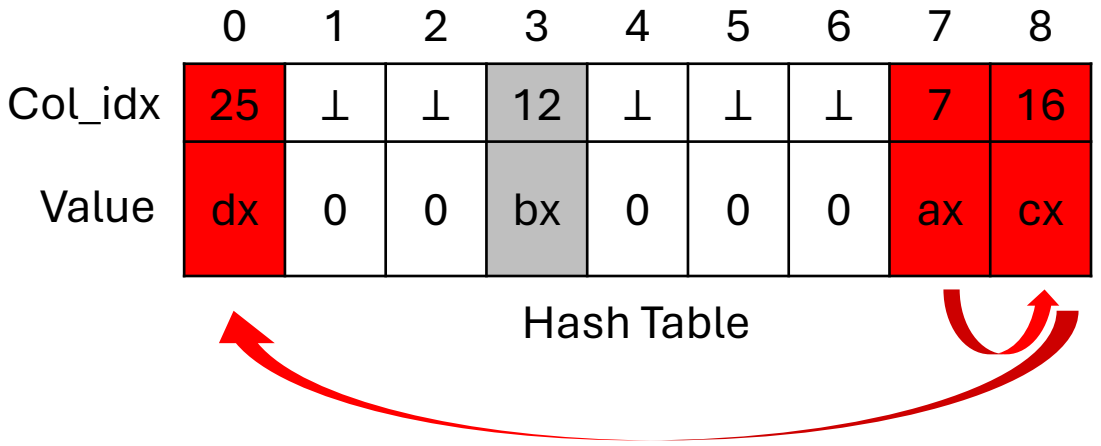




Accumulation using Hashmaps

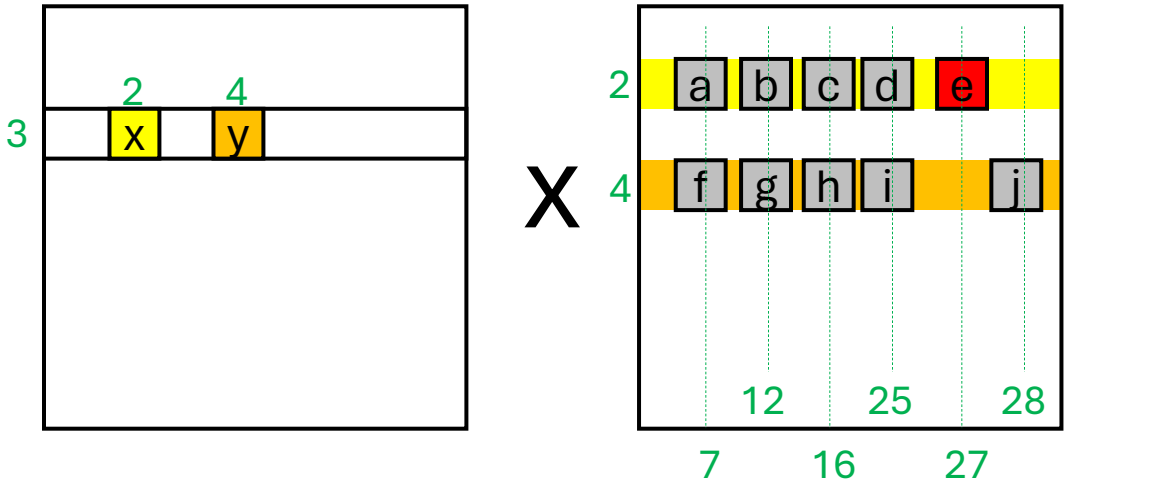
- In Numeric phase,
  - Hash table size = 1.5 ⨉ the number of nonzero entries = 1.5 ⨉ 6 = 9.
  - Maintain both column indices and corresponding values.
  - Assume hash function h = (h+1)%9 in case of collisions.

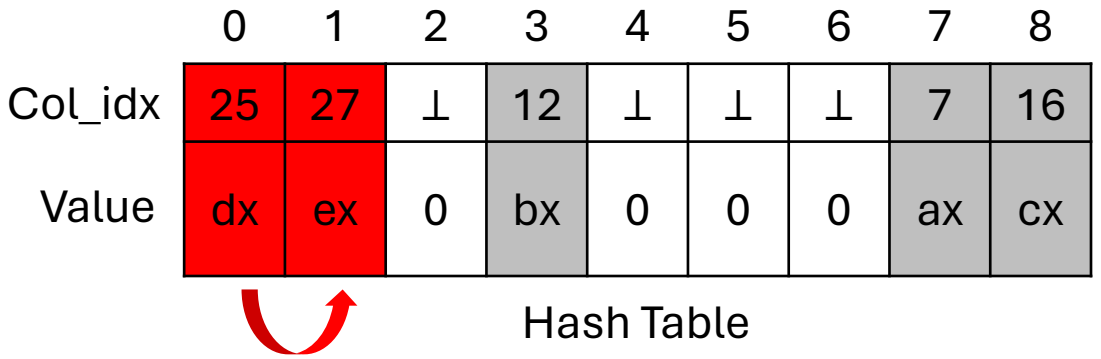




Accumulation using Hashmaps

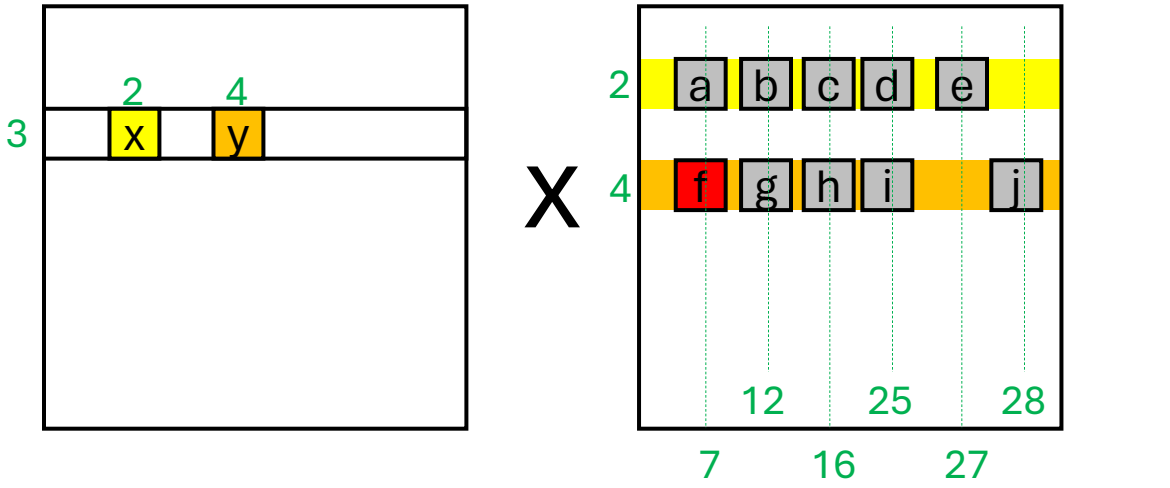
- In Numeric phase,
  - Hash table size = 1.5 ⨉ the number of nonzero entries = 1.5 ⨉ 6 = 9.
  - Maintain both column indices and corresponding values.
  - Assume hash function h = (h+1)%9 in case of collisions.

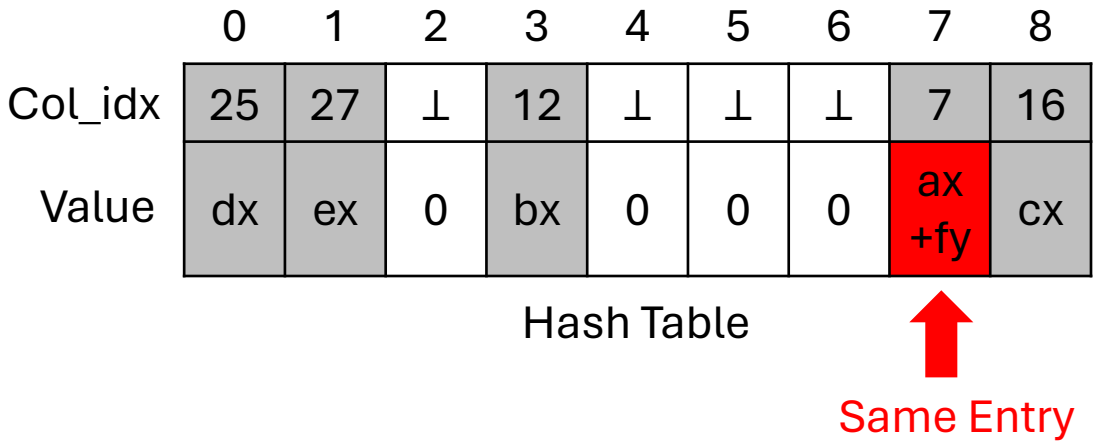




Accumulation using Hashmaps

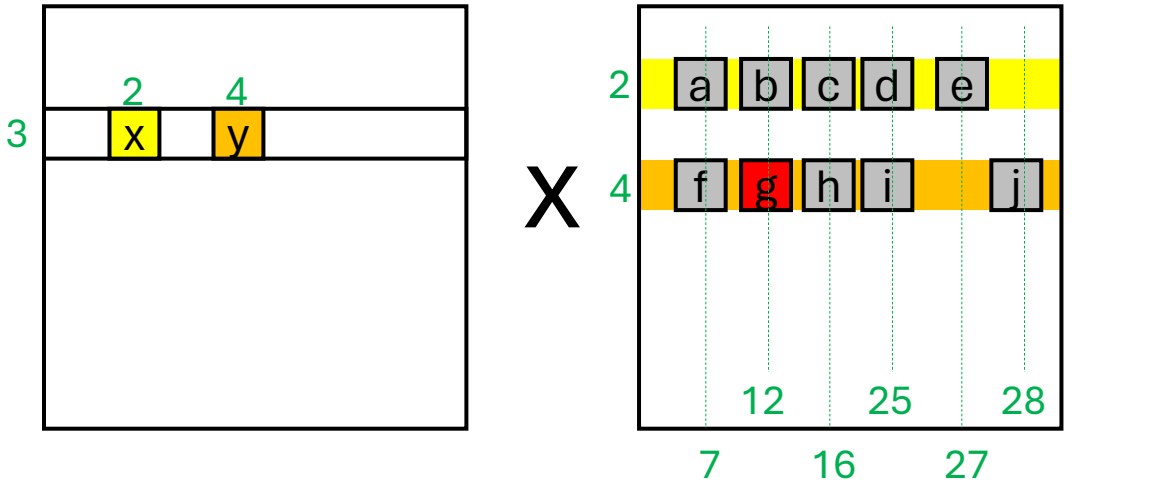
- In Numeric phase,
  - Hash table size = 1.5 ⨉ the number of nonzero entries = 1.5 ⨉ 6 = 9.
  - Maintain both column indices and corresponding values.
  - Assume hash function h = (h+1)%9 in case of collisions.

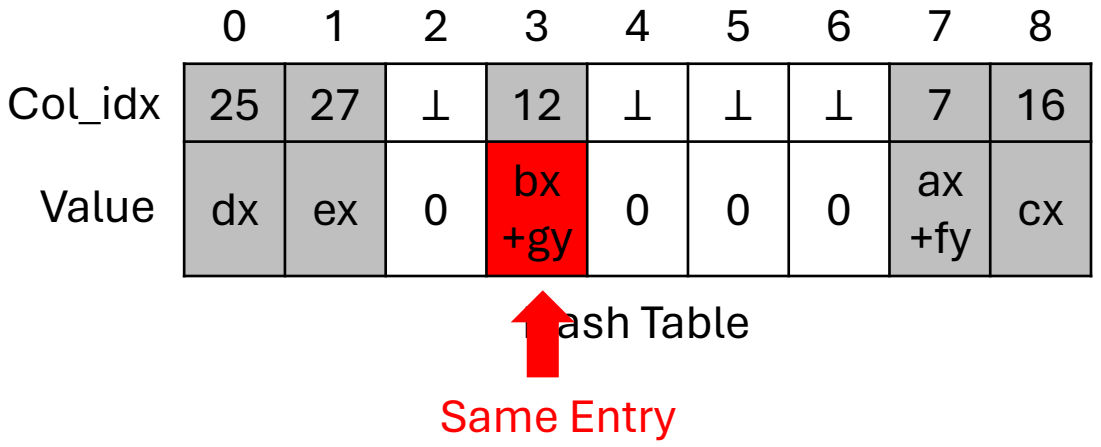




Accumulation using Hashmaps

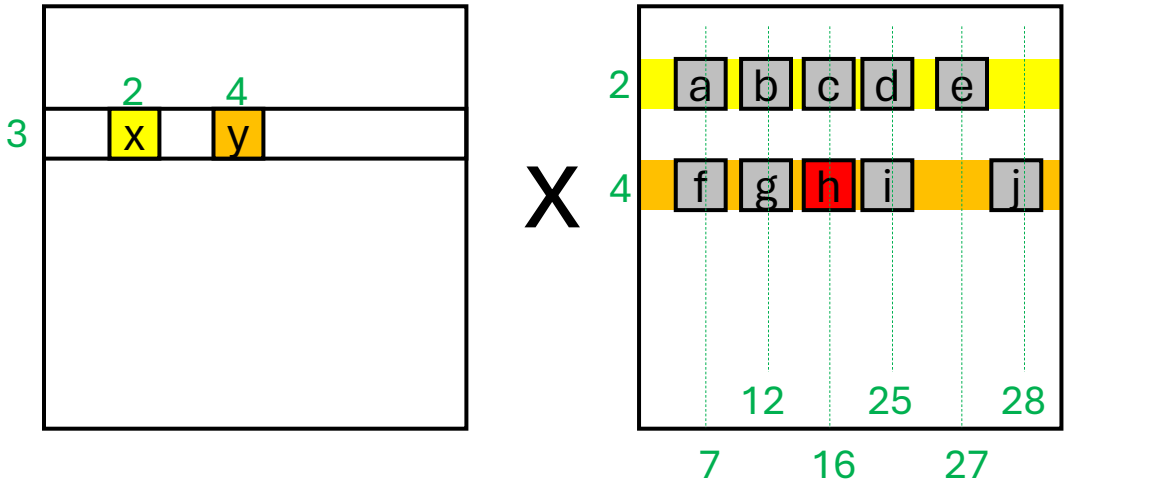
- In Numeric phase,
  - Hash table size = 1.5 ⨉ the number of nonzero entries = 1.5 ⨉ 6 = 9.
  - Maintain both column indices and corresponding values.
  - Assume hash function h = (h+1)%9 in case of collisions.





Accumulation using Hashmaps

- In Numeric phase,
  - Hash table size = 1.5 ✕ the number of nonzero entries = 1.5 ✕ 6 = 9.
  - Maintain both column indices and corresponding values.
  - Assume hash function h = (h+1)%9 in case of collisions.



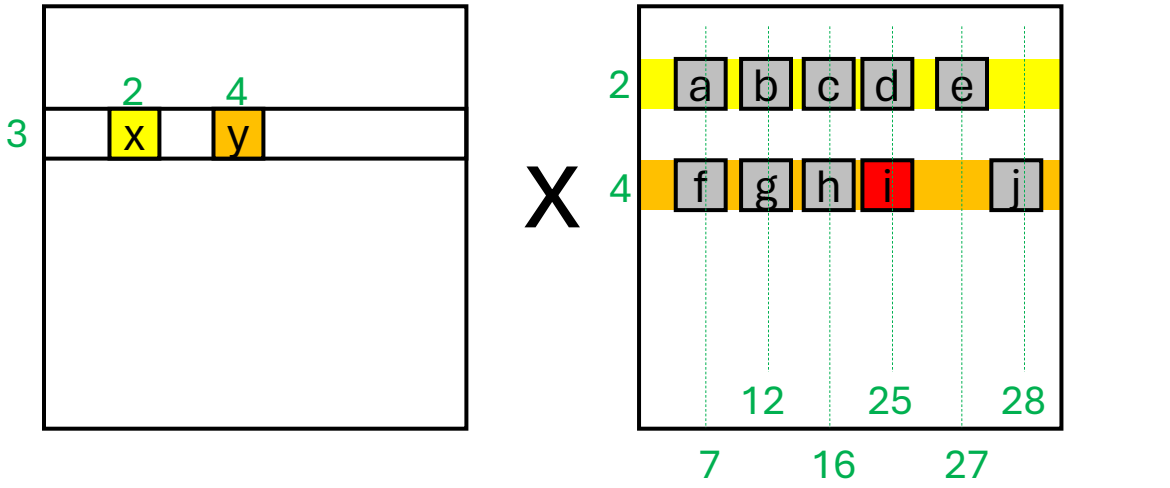
	0	1	2	3	4	5	6	7	8
Col_idx	25	27	⊥	12	⊥	⊥	⊥	7	16
Value	dx	ex	0	bx +gy	0	0	0	ax +fy	cx +hy

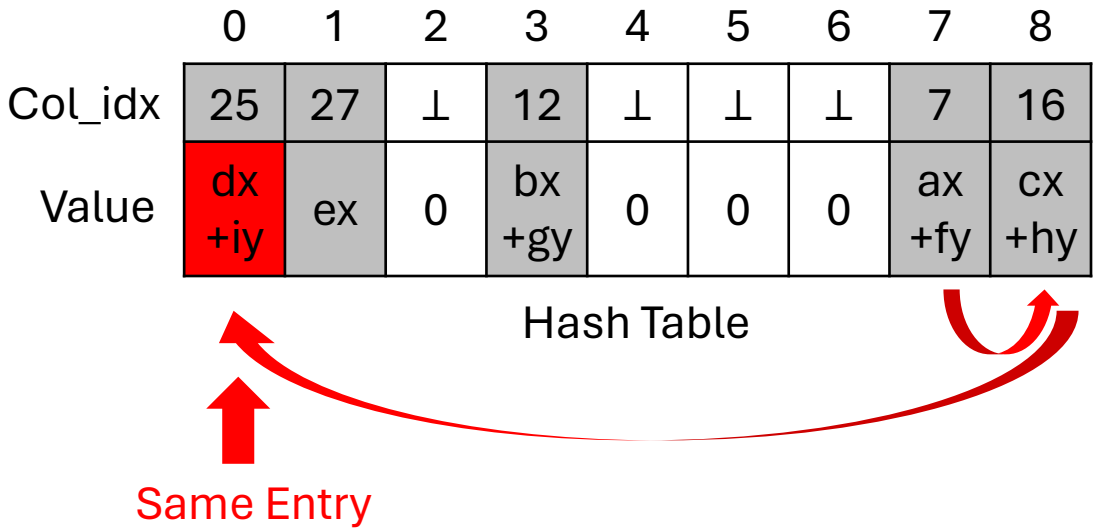
Hash Table

Same Entry

Accumulation using Hashmaps

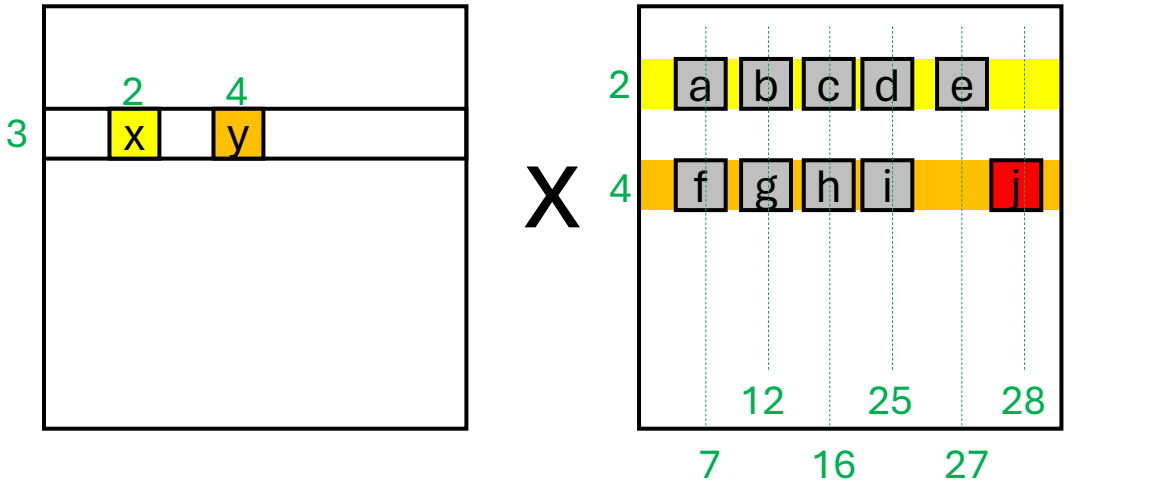
- In Numeric phase,
  - Hash table size = 1.5 ✕ the number of nonzero entries = 1.5 ✕ 6 = 9.
  - Maintain both column indices and corresponding values.
  - Assume hash function h = (h+1)%9 in case of collisions.

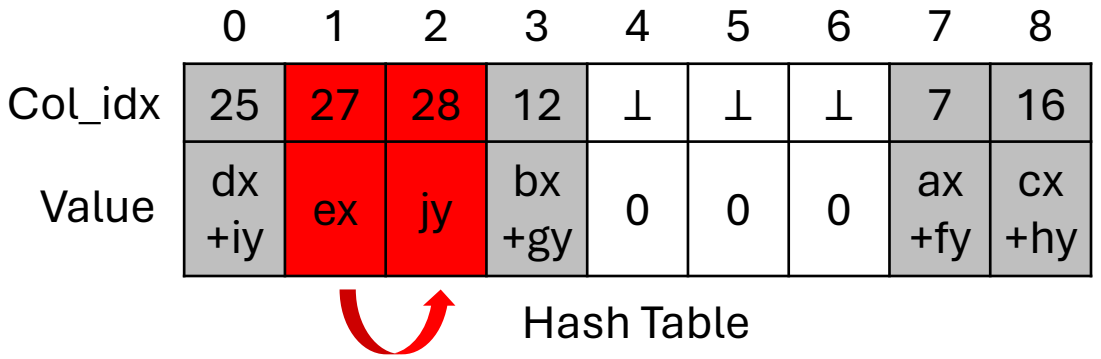




Accumulation using Hashmaps

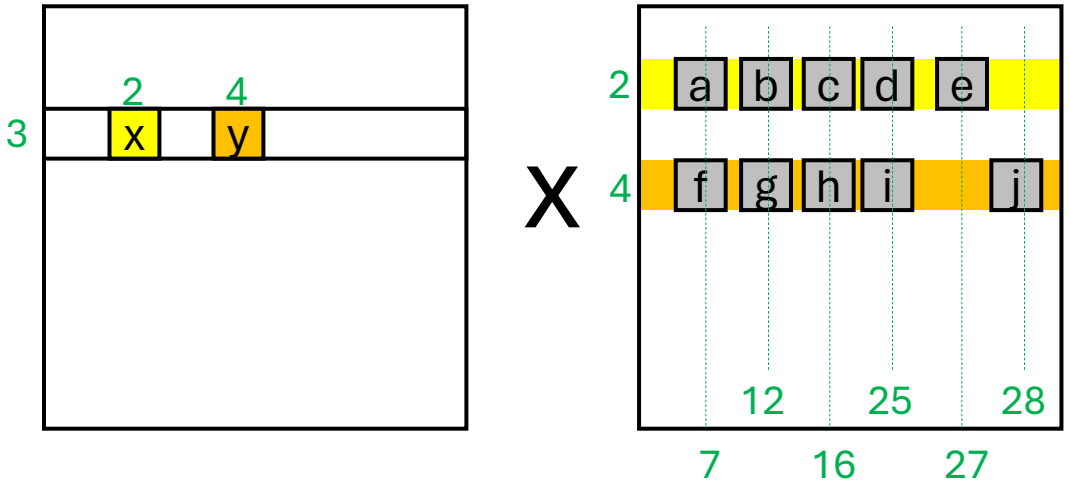
- In Numeric phase,
  - Hash table size = 1.5 ⨉ the number of nonzero entries = 1.5 ⨉ 6 = 9.
  - Maintain both column indices and corresponding values.
  - Assume hash function h = (h+1)%9 in case of collisions.





Accumulation using Hashmaps

- In Numeric phase,
  - Hash table size = 1.5 ⨉ the number of nonzero entries = 1.5 ⨉ 6 = 9.
  - Maintain both column indices and corresponding values.
  - Assume hash function h = (h+1)%9 in case of collisions.

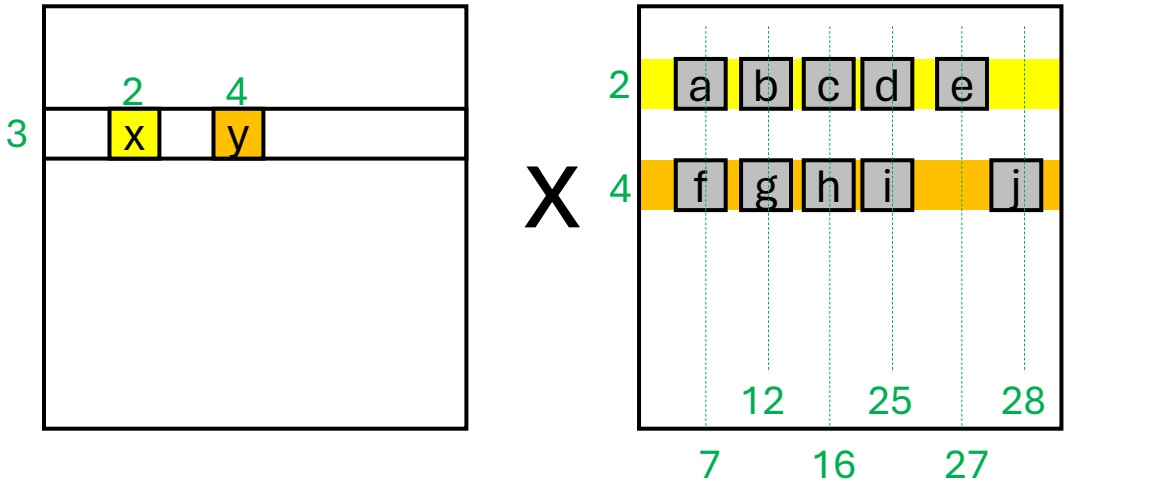


	0	1	2	3	4	5	6	7	8
Col_idx	25	27	28	12	⊥	⊥	⊥	7	16
Value	dx +iy	ex	jy	bx +gy	0	0	0	ax +fy	cx +hy

Hash Table

Accumulation using Hashmaps

- In Numeric phase,
  - Hash table size = 1.5 ⨉ the number of nonzero entries = 1.5 ⨉ 6 = 9.
  - Maintain both column indices and corresponding values.
  - Assume hash function h = (h+1)%9 in case of collisions.



	0	1	2	3	4	5	6	7	8
Col_idx	25	27	28	12	⊥	⊥	⊥	7	16
Value	dx +iy	ex	jy	bx +gy	0	0	0	ax +fy	cx +hy

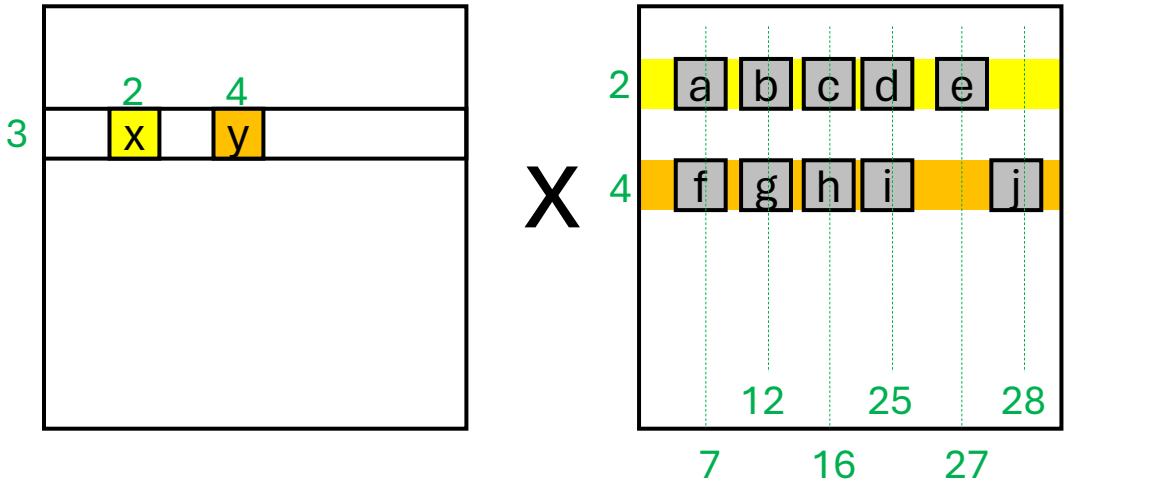
Hash Table

Gather using prefix-sum

Col_idx	25	27	28	12	7	16
Value	dx +iy	ex	jy	bx +gy	ax +fy	cx +hy

Accumulation using Hashmaps

- In Numeric phase,
  - Hash table size = 1.5 × the number of nonzero entries = 1.5 × 6 = 9.
  - Maintain both column indices and corresponding values.
  - Assume hash function h = (h+1)%9 in case of collisions.



	0	1	2	3	4	5	6	7	8
Col_idx	25	27	28	12	⊥	⊥	⊥	7	16
Value	dx +iy	ex	jy	bx +gy	0	0	0	ax +fy	cx +hy

Hash Table

Gather using prefix-sum

Col_idx	25	27	28	12	7	16
Value	dx +iy	ex	jy	bx +gy	ax +fy	cx +hy

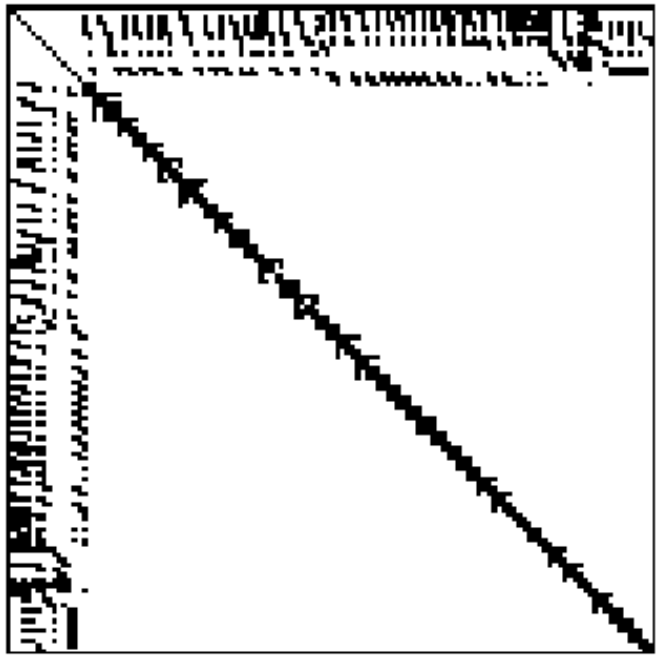
Sort

Col_idx	7	12	16	25	27	28
Value	ax +fy	bx +gy	cx +hy	dx +iy	ex	jy

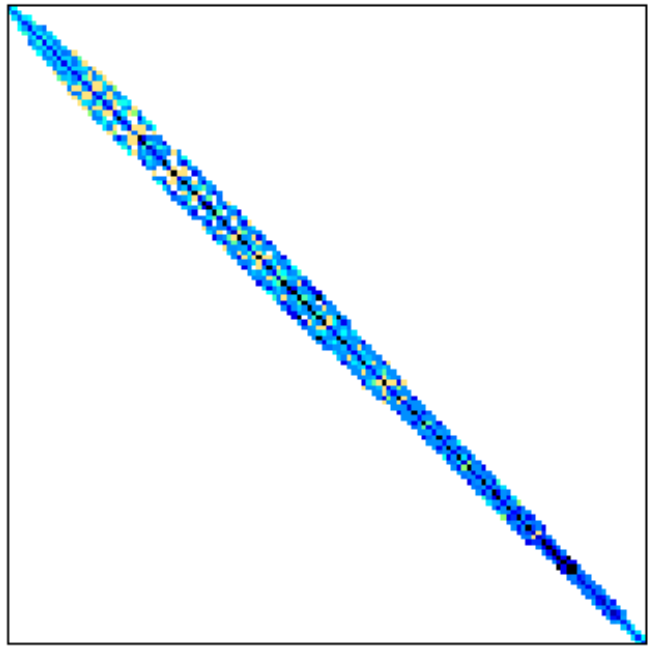
Partial CSR output

Many real-world sparse matrices are structured

- In many real-world sparse matrices, the nonzero entries are typically densely populated.
- Leveraging clustered entries is crucial for achieving high performance.



mip1



shipsec1

* http://sparse.tamu.edu

Accumulation using Densemaps

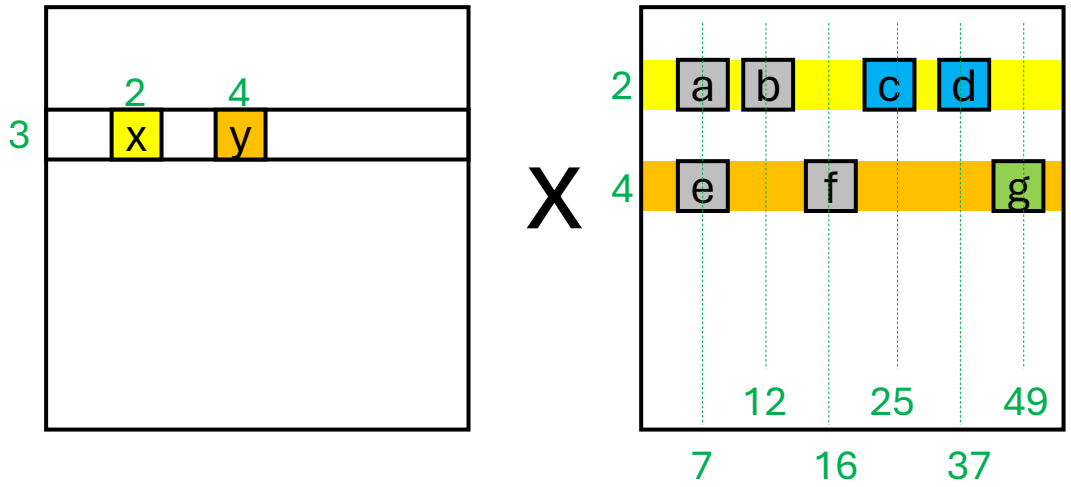
- Hashmaps incur overheads.
  - Keeping track of nonzero column indices.
  - Additional costs arise in the event of collision.
  - Random access of the hash table, particularly problematic for hash tables in global memory.

- When the nonzero entries of a row in the output are densely populated, a viable alternative is to use a dense array in shared memory.
  - This approach eliminates the need for storing column indices and handling hash function collisions.
  - It results in redundant memory space consumption.
  - The use of a dense array is advantageous only when the entries within a row are densely populated.

- If the range from minimum to maximum column index in the resulting row does not fit in scratchpad memory, the dense accumulator needs multiple iterations on different column ranges, successively progressing through the output row.
  - Need to store the positions of the last element that could be processed in the current iteration for each row.

Accumulation using Densemaps

- Assume the shared memory size is 16



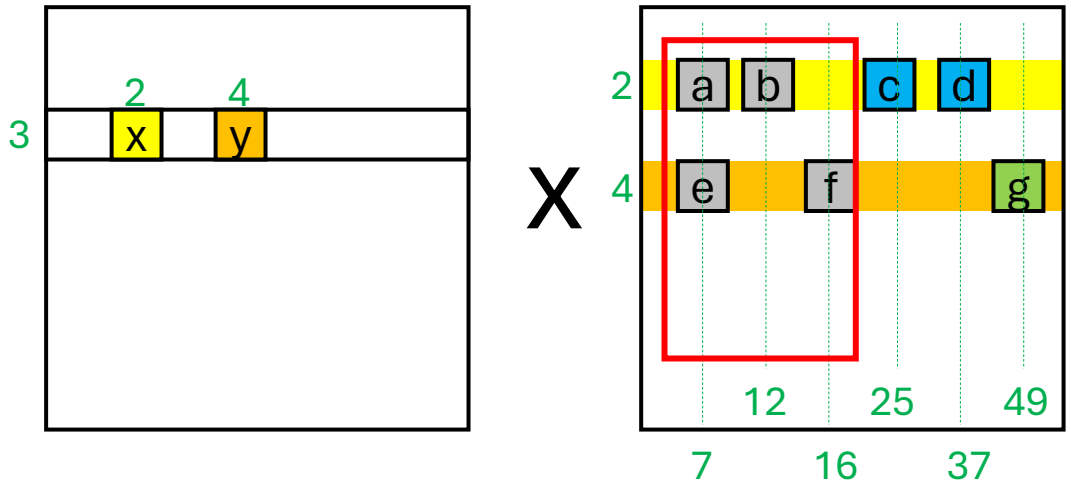
Array idx (implicit)

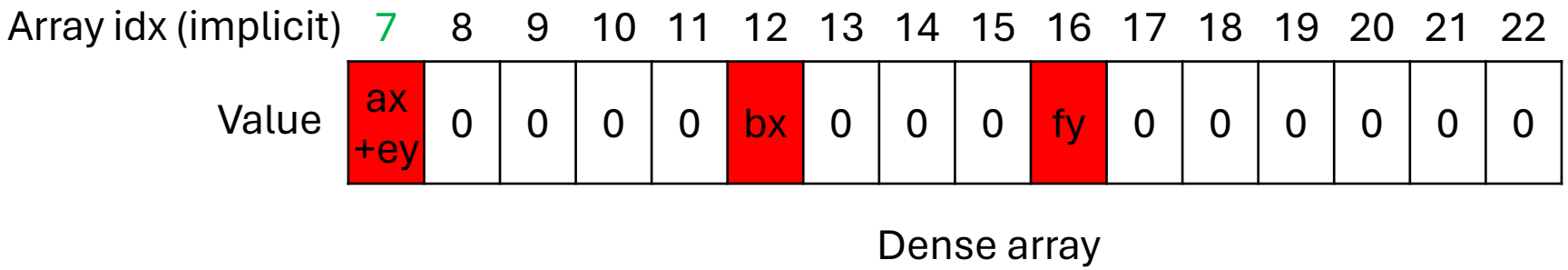
Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Dense array

Accumulation using Densemaps

- Assume the shared memory size is 16

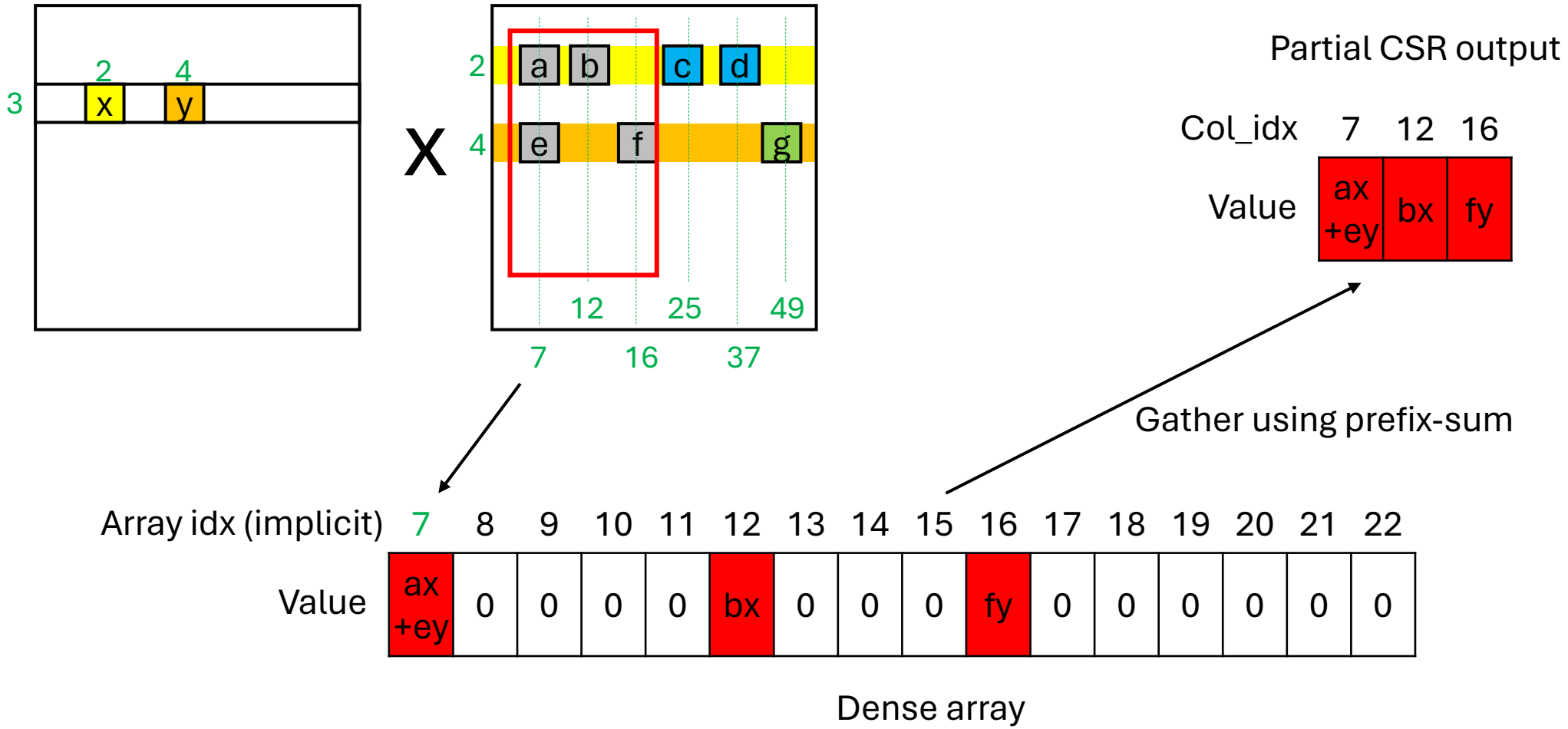




Dense array

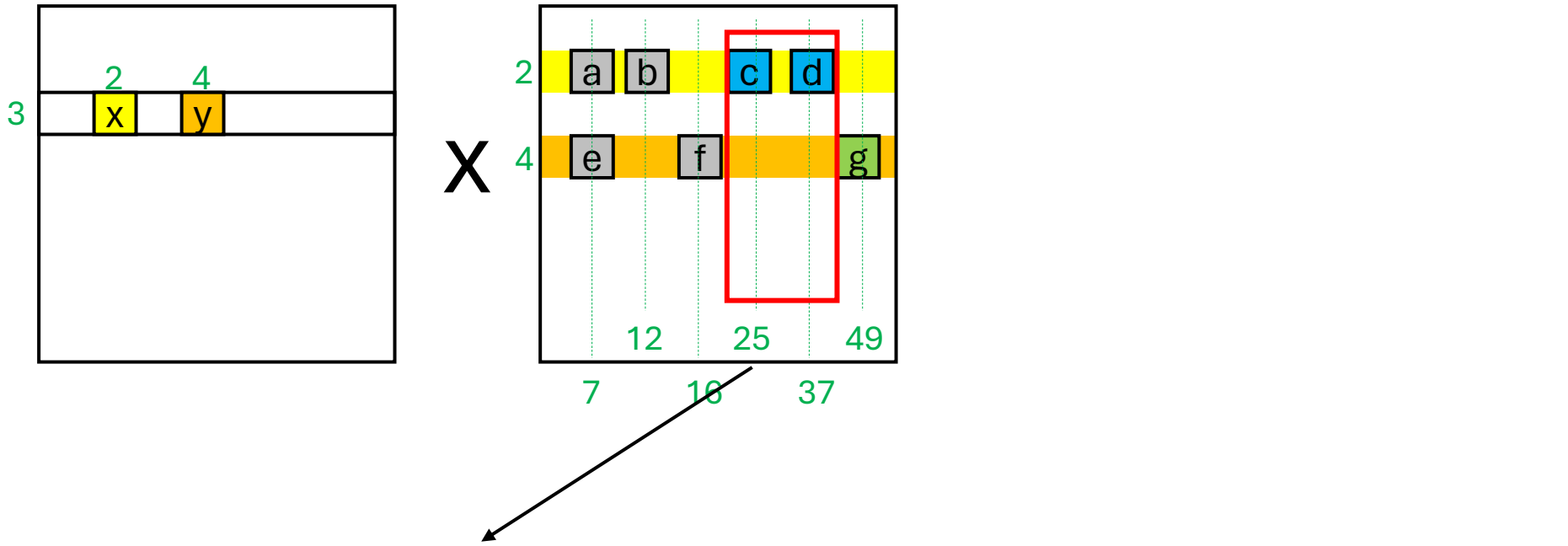
Accumulation using Densemaps

- Assume the shared memory size is 16



Accumulation using Densemaps

- Assume the shared memory size is 16

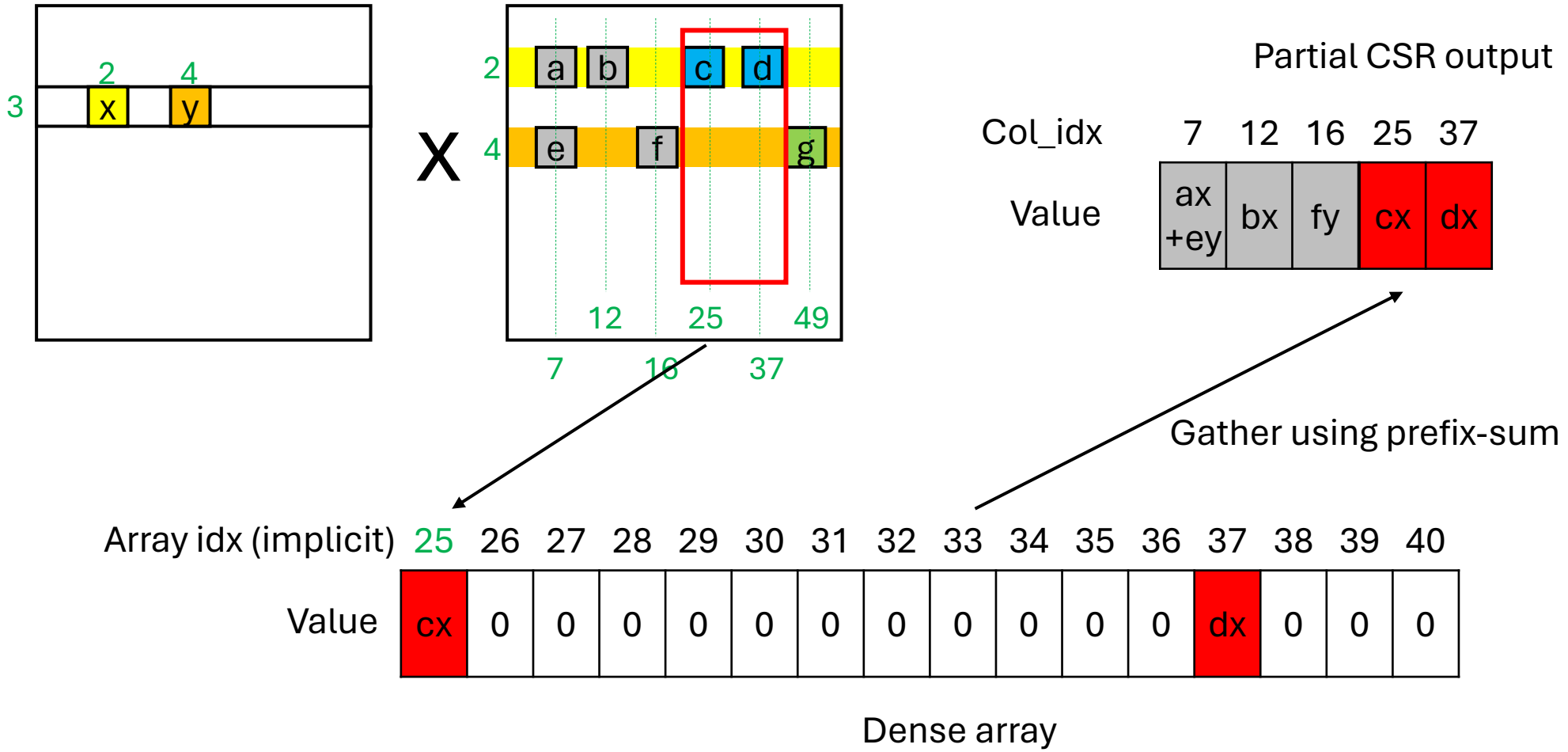


Array idx (implicit)	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
Value	cx	0	0	0	0	0	0	0	0	0	0	0	dx	0	0	0

Dense array

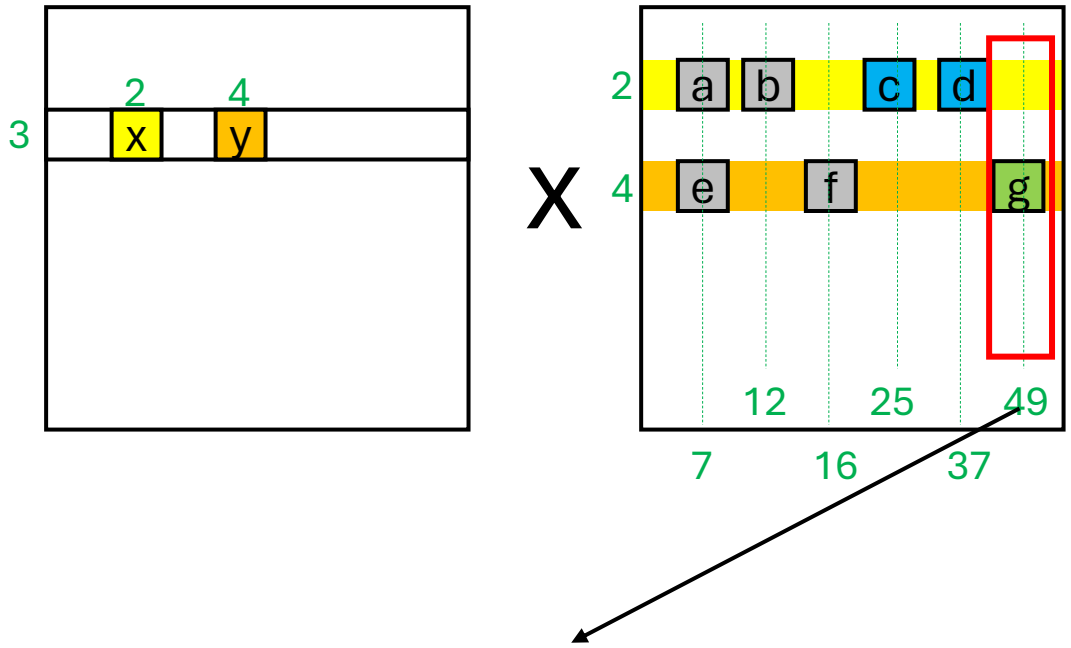
Accumulation using Densemaps

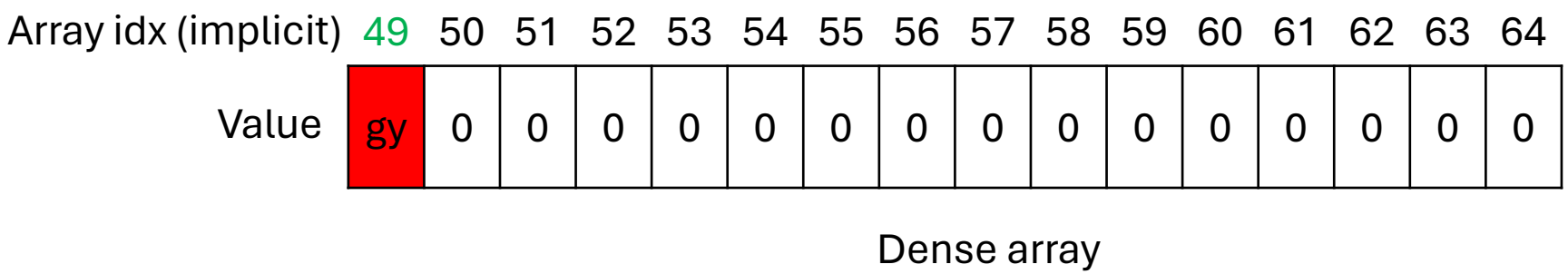
- Assume the shared memory size is 16



Accumulation using Densemaps

- Assume the shared memory size is 16

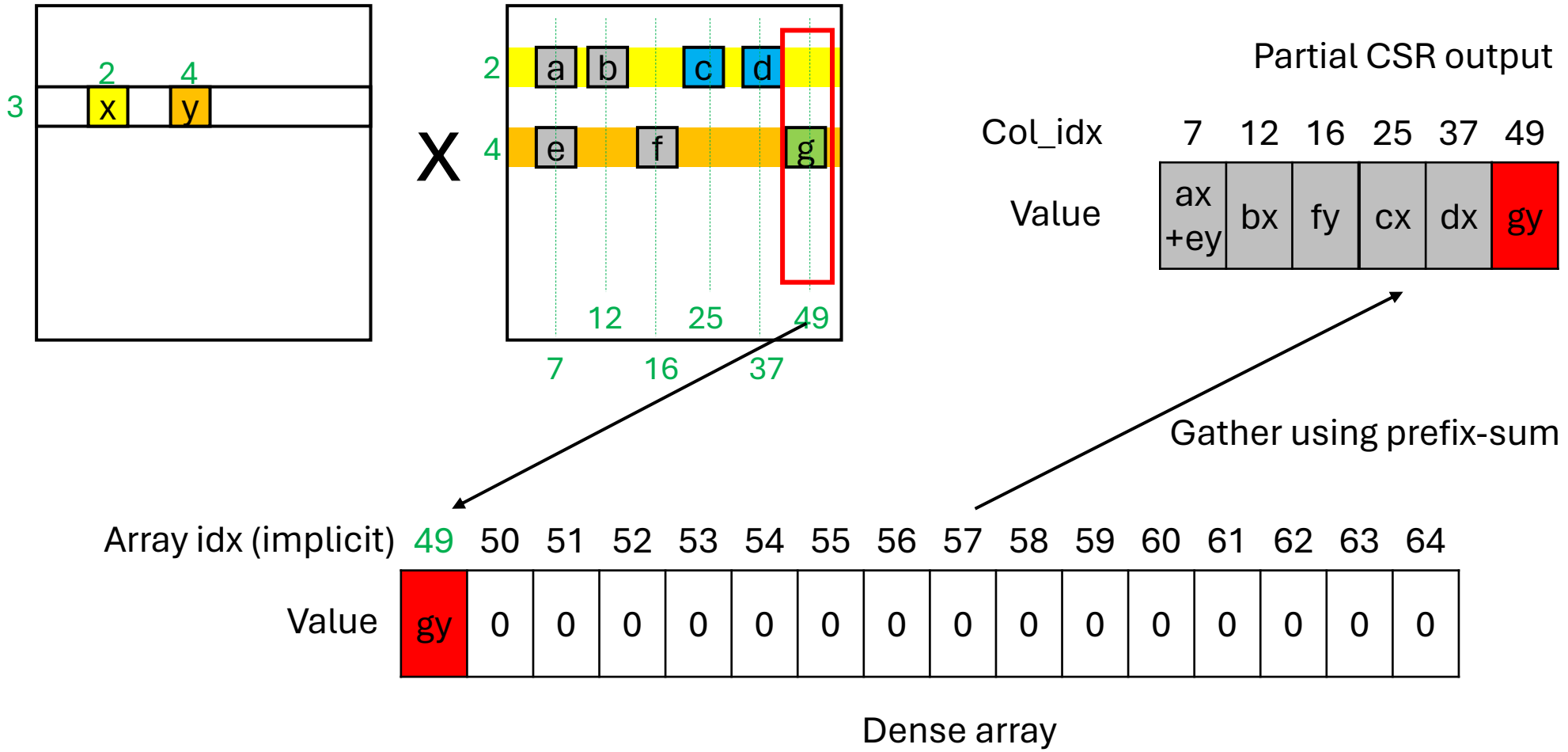




Dense array

Accumulation using Densemaps

- Assume the shared memory size is 16



Optimizing SpGEMM on GPUs is challenging

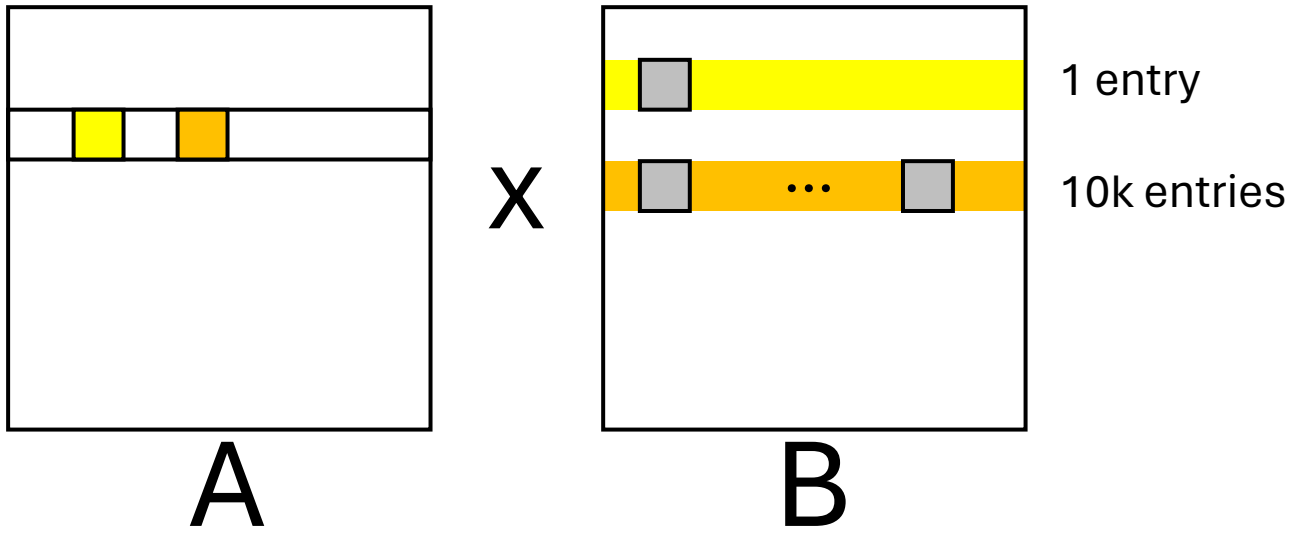
- Parallel assembly for the output.
  - The sparse output matrix needs to be constructed in parallel.
  - The output size is unknown a priori.

- Accumulating partial products.
  - Accumulating partial products in global memory significantly hurts performance.
    - Causes uncoalesced atomic memory accesses.

- Load balancing.
  - All matrices are irregular.
  - Achieving load-balanced execution in SpGEMM is significantly more challenging compared to SpMV.

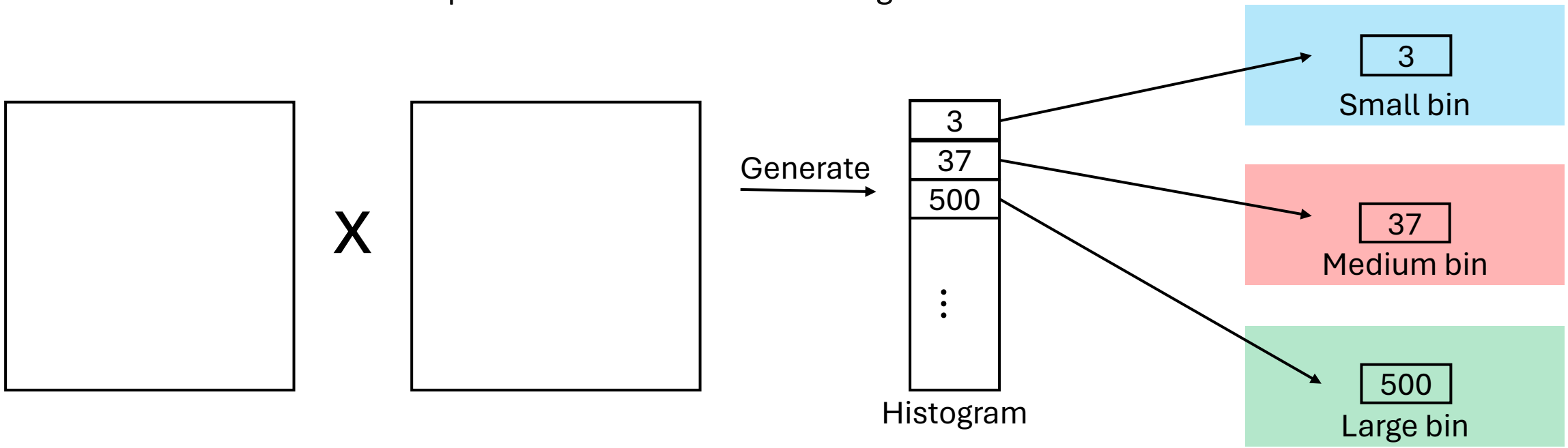
Load balancing

- Considering load-balanced execution across A and B (C = A X B) seems to be crucial.
  - Assigning a predefined number of thread to each nonzero entry of A is insufficient.



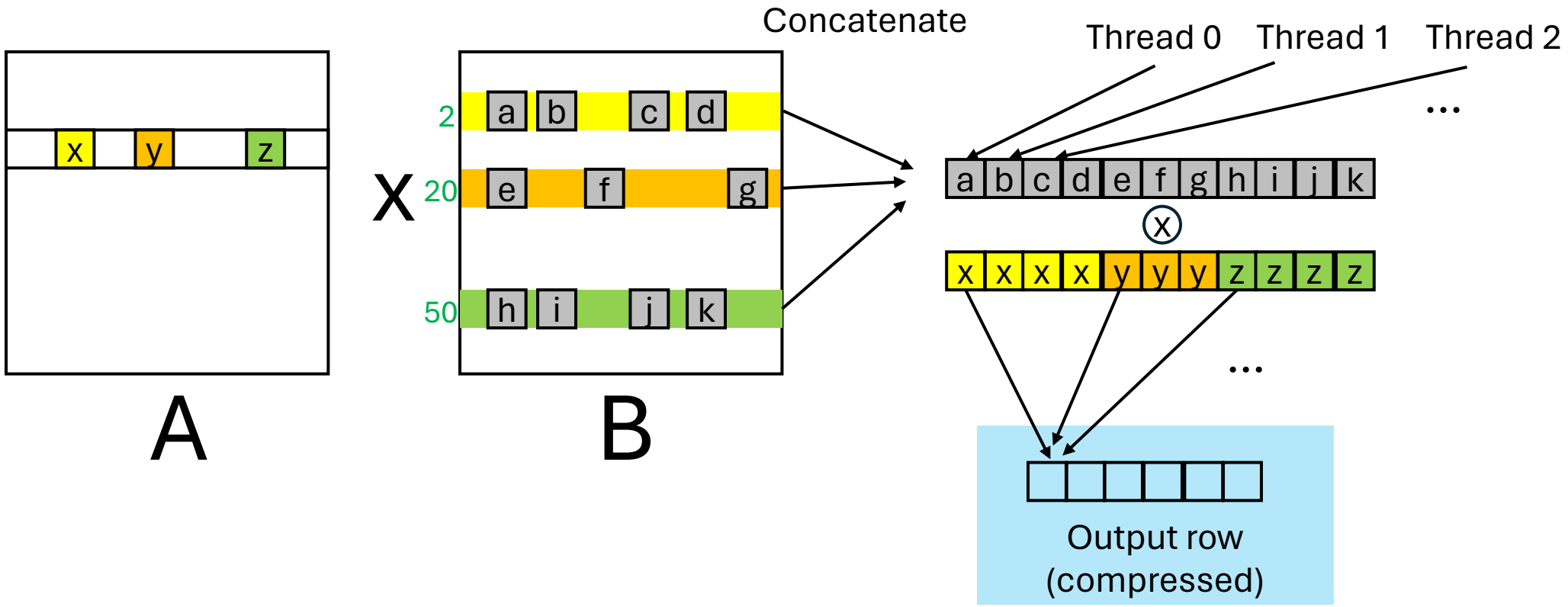
Load balancing

- Obtain the number of partial products for each row of the output
- Apply binning similar to SpMV's binning
  - Small bin contains rows of the output with < 4 partial products
    - Each thread processes each row in the small bin
  - Medium bin contains rows of the output with < 128 partial products
    - Each warp processes each row in the medium bin
  - Large bin contains rows of the output with ≥ 128 partial products
    - Each thread block processes each row in the large bin



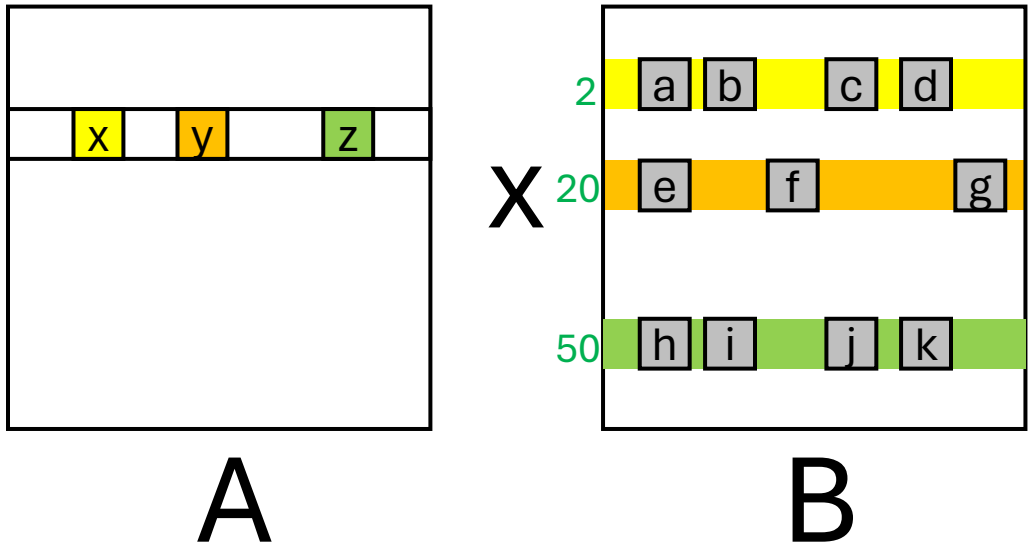
Load balancing

- Apply ‘strict nonzero splitting’ for each row of the input.
- Virtually concatenate rows of B corresponding to nonzero column indices of a row of A.
  - Threads process each entry in the concatenated structure in a cyclic fashion.



Load balancing

- Using binary search on the auxiliary arrays 'scatter_row_ptr' and 'scatter_col_idx' facilitates 'strict nonzero splitting' without actual concatenation.
  - Actually, 'scatter_col_idx' is a partial 'col_idx' of the original CSR for A.



Binary search is utilized to access the CSR structure of B

Scatter_row_ptr	0	4	7	11
Scatter_col_idx	2	20	50	

A portion of the CSR structure of A

Questions?