058165 - PARALLEL COMPUTING

Fabrizio Ferrandi

a.a. 2022-2023

References

- "Structured Parallel Programming: Patterns for Efficient Computation," Michael McCool, Arch Robinson, James Reinders, 1st edition, Morgan Kaufmann, ISBN: 978-0-12-415993-8, 2012

Outline

- Gather Pattern
  - Shifts, Zip, Unzip
- Scatter Pattern
  - Collision Rules: atomic, permutation, merge, priority
- Pack Pattern
  - Split, Unsplit, Bin
  - Fusing Map and Pack
  - Expand
- Partitioning Data
- AoS vs. SoA
- Example Implementation: AoS vs. SoA

Data Movement

- Performance is often more limited by data movement than by computation
  - Transferring data across memory layers is costly
    - locality is important to minimize data access times
    - data organization and layout can impact this
  - Transferring data across networks can take many cycles
    - attempting to minimize the # messages and overhead is important
  - Data movement also costs more in power
- For "data intensive" application, it is a good idea to design the data movement first
  - Design the computation around the data movements
  - Applications such as search and sorting are all about data movement and reorganization

Parallel Data Reorganization

- Remember we are looking to do things in parallel
- How to be faster than the sequential algorithm?
- Similar consistency issues arise as when dealing with computation parallelism
- Here we are concerned more with parallel data movement and management issues
- Might involve the creation of additional data structures (e.g., for holding intermediate data)

Gather Pattern

- Gather pattern creates a (source) collection of data by reading from another (input) data collection
  - Given a collection of (ordered) indices
  - Read data from the source collection at each index
  - Write data to the output collection in index order
- Transfers from source collection to output collection
  - Element type of output collection is the same as the source
  - Shape of the output collection is that of the index collection
    - same dimensionality
- Can be considered a combination of map and random serial read operations
  - Essentially does a number of random reads in parallel

Gather: Serial Implementation


```
1  template<typename Data, typename Idx>
2  void gather(
3      size_t n, // number of elements in data collection
4      size_t m, // number of elements in index collection
5      Data a[], // input data collection (n elements)
6      Data A[], // output data collection (m elements)
7      Idx idx[] // input index collection (m elements)
8  ) {
9      for (size_t i = 0; i < m; ++i) {
10         size_t j = idx[i]; // get ith index
11         assert(0 <= j && j < n); // check array bounds
12         A[i] = a[j]; // perform random read
13     }
14 }
```

Serial implementation of gather in pseudocode

Gather: Serial Implementation


```
1  template<typename Data, typename Idx>
2  void gather(
3      size_t n, // number of elements in data collection
4      size_t m, // number of elements in index collection
5      Data a[], // input data collection (n elements)
6      Data A[], // output data collection (m elements)
7      Idx idx[] // input index collection (m elements)
8  ) {
9      for (size_t i = 0; i < m; ++i) {
10         size_t j = idx[i]; // get ith index
11         assert(0 <= j && j < n); // check array bounds
12         A[i] = a[j]; // perform random read
13     }
14 }
```

Serial implementation of gather in pseudocode

Do you see opportunities for parallelism?

Gather: Serial Implementation


```
1  template<typename Data, typename Idx>
2  void gather(
3      size_t n, // number of elements in data collection
4      size_t m, // number of elements in index collection
5      Data a[], // input data collection (n elements)
6      Data A[], // output data collection (m elements)
7      Idx idx[] // input index collection (m elements)
8  ) {
9      for (size_t i = 0; i < m; ++i) {
10         size_t j = idx[i]; // get ith index
11         assert(0 <= j && j < n); // check arra
12         A[i] = a[j]; // perform random read
13     }
14 }
```

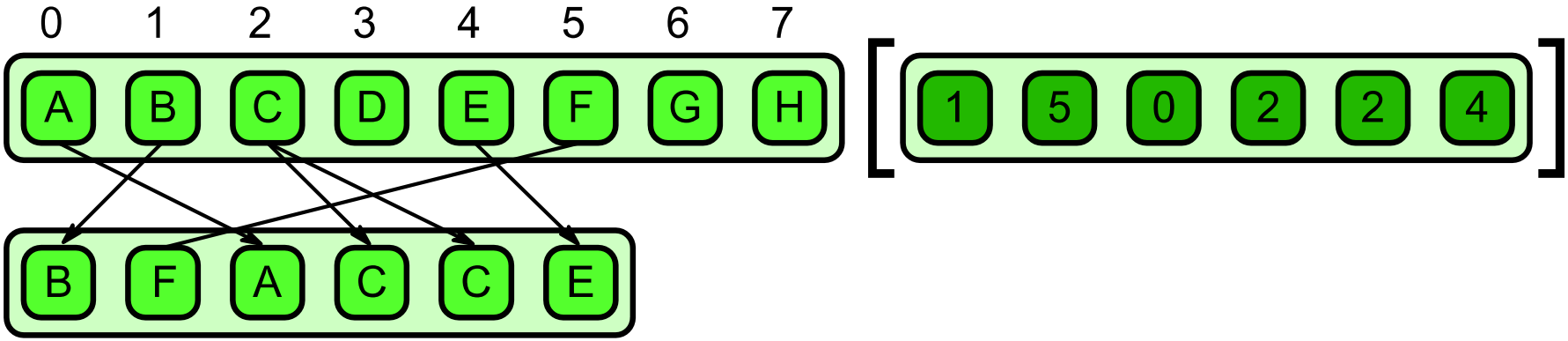
Parallelize over for loop to perform random read

Serial implementation of gather in pseudocode

Are there any conflicts that arise?

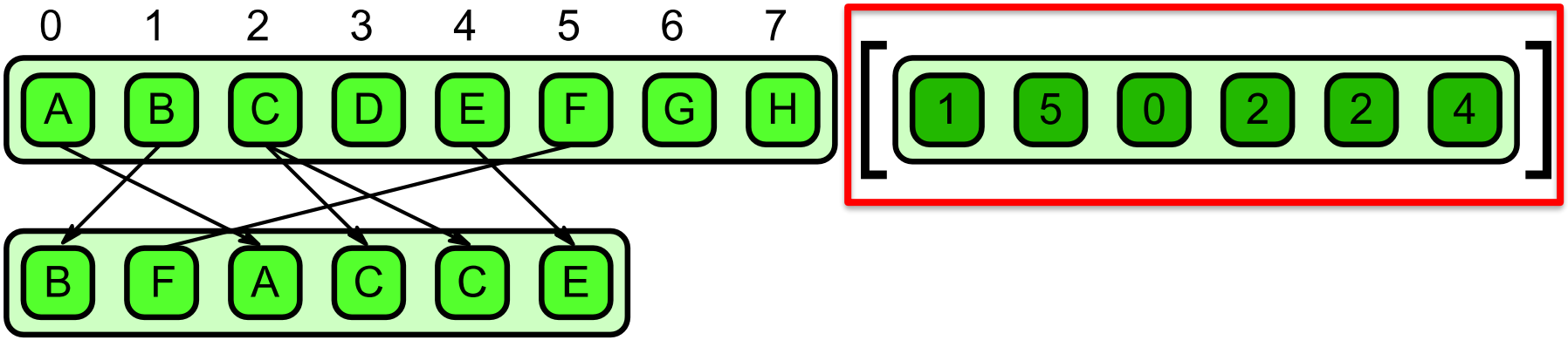
Gather: Defined (parallel perspective)

- Results from the combination of a map with a random read



- Simple pattern, but with many special cases that make the implementation more efficient

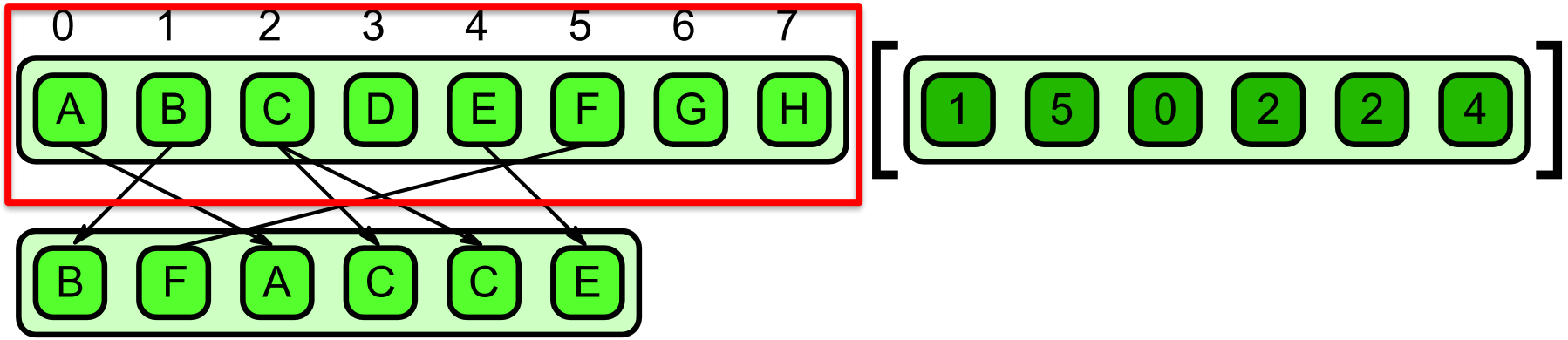
Gather: Defined



Given a collection of read locations

- address or array indices

Gather: Defined

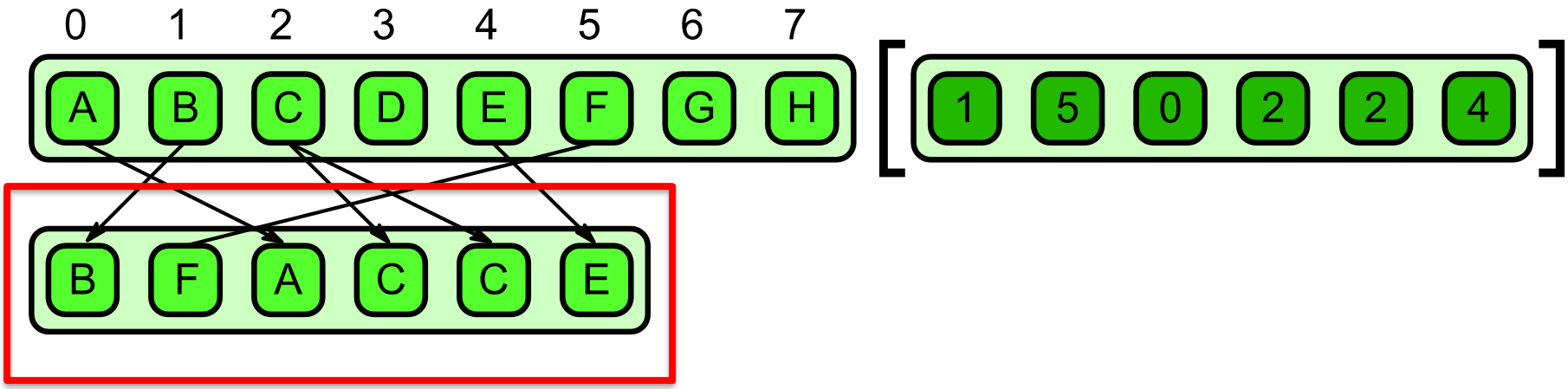


Given a collection of read locations

- address or array indices

and a source array

Gather: Defined



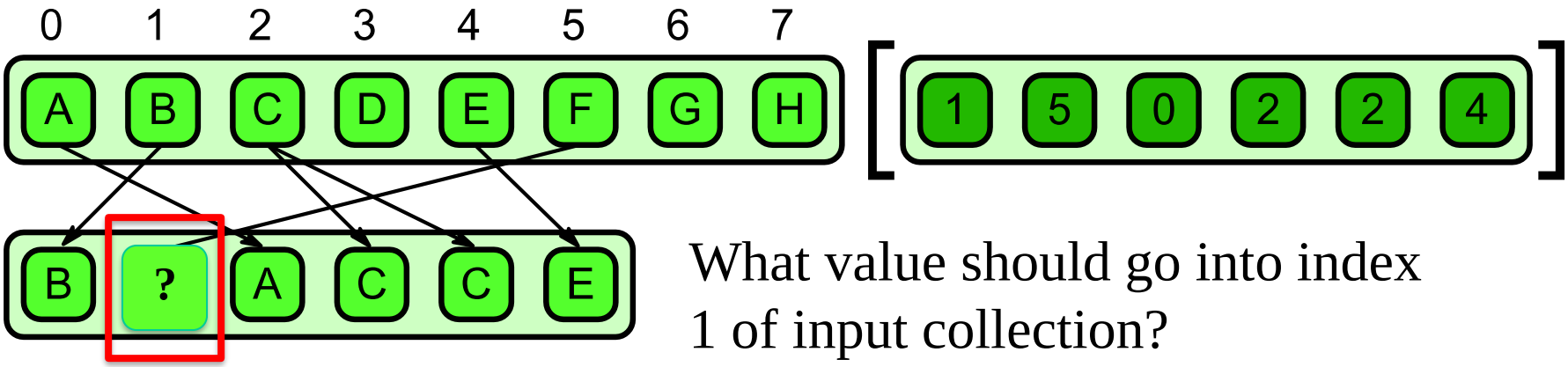
Given a collection of read locations

- address or array indices

and a source array

gather all the data from the source array at the given locations and places them into an **output collection**

Gather: Defined



What value should go into index 1 of input collection?

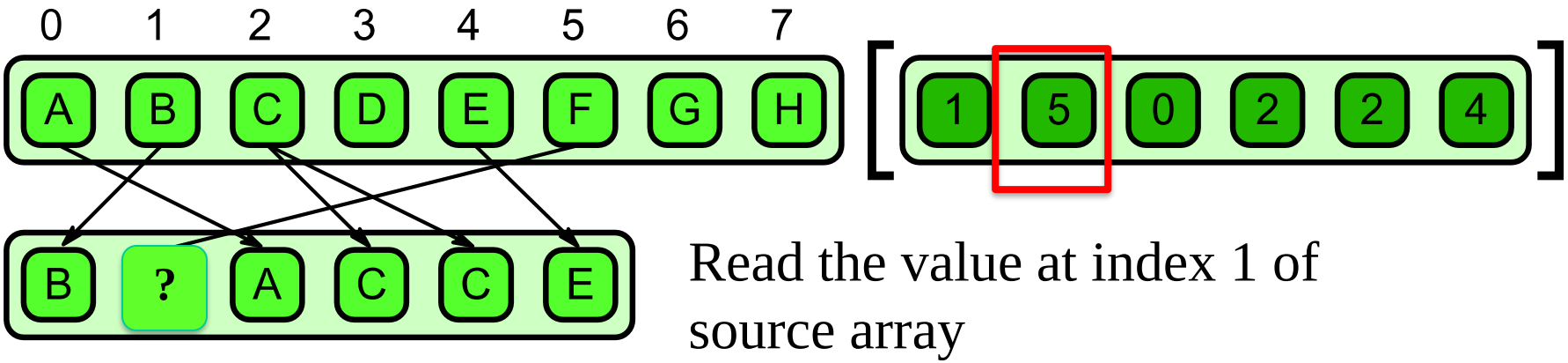
Given a collection of read locations

- address or array indices

and a source array

gather all the data from the source array at the given locations and places them into an **output collection**

Gather: Defined



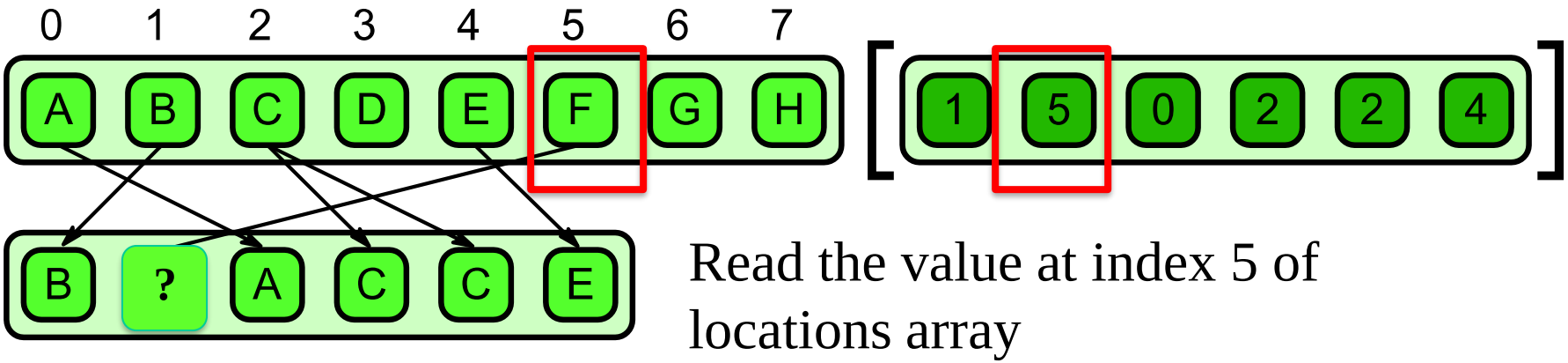
Given a collection of read locations

- address or array indices

and a source array

gather all the data from the source array at the given locations and places them into an **output collection**

Gather: Defined



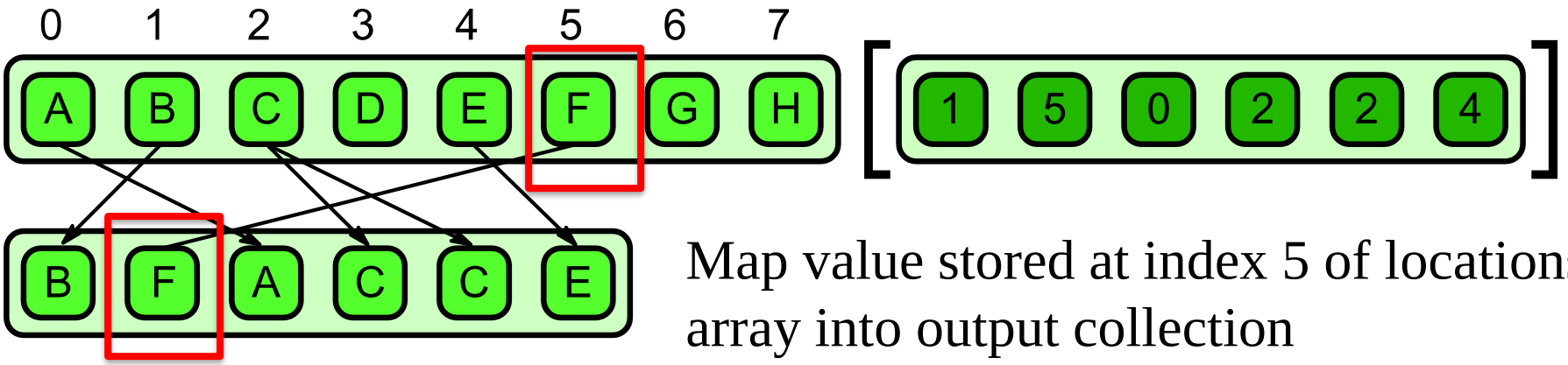
Given a collection of read locations

- address or array indices

and a source array

gather all the data from the source array at the given locations and places them into an **output collection**

Gather: Defined



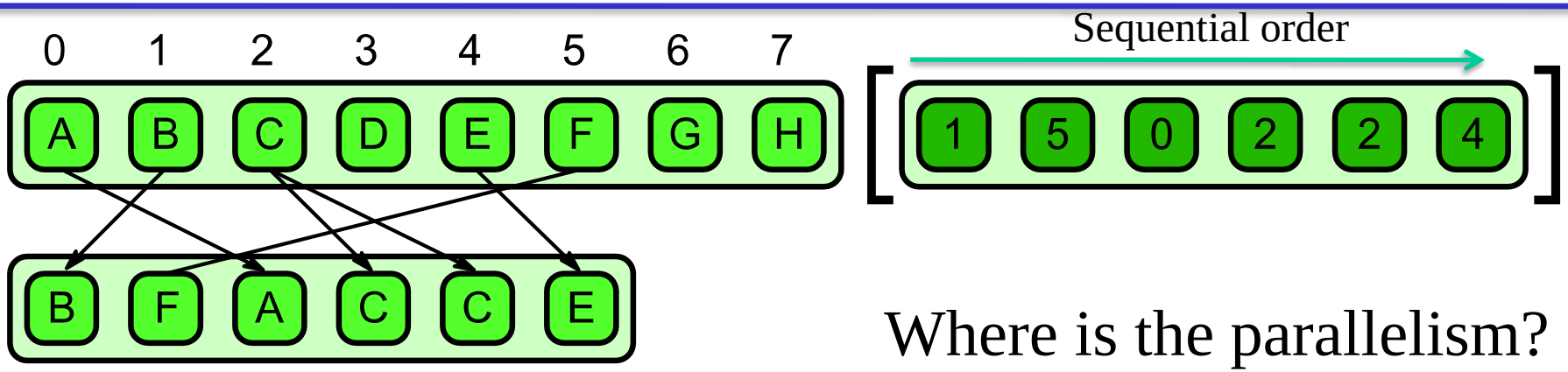
Given a collection of read locations

- address or array indices

and a source array

gather all the data from the source array at the given locations and places them into an output collection

Gather: Defined



Where is the parallelism?

Given a collection of read locations

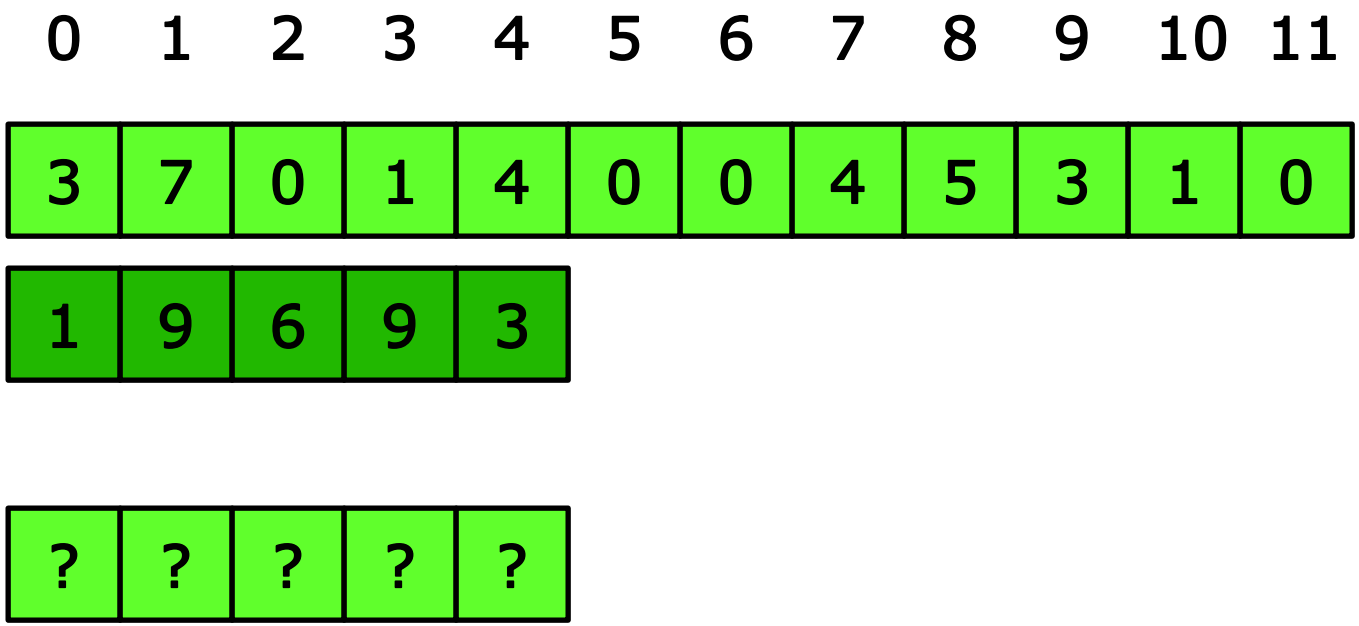
- address or array indices

and a source array

gather all the data from the source array at the given locations and places them into an **output collection**

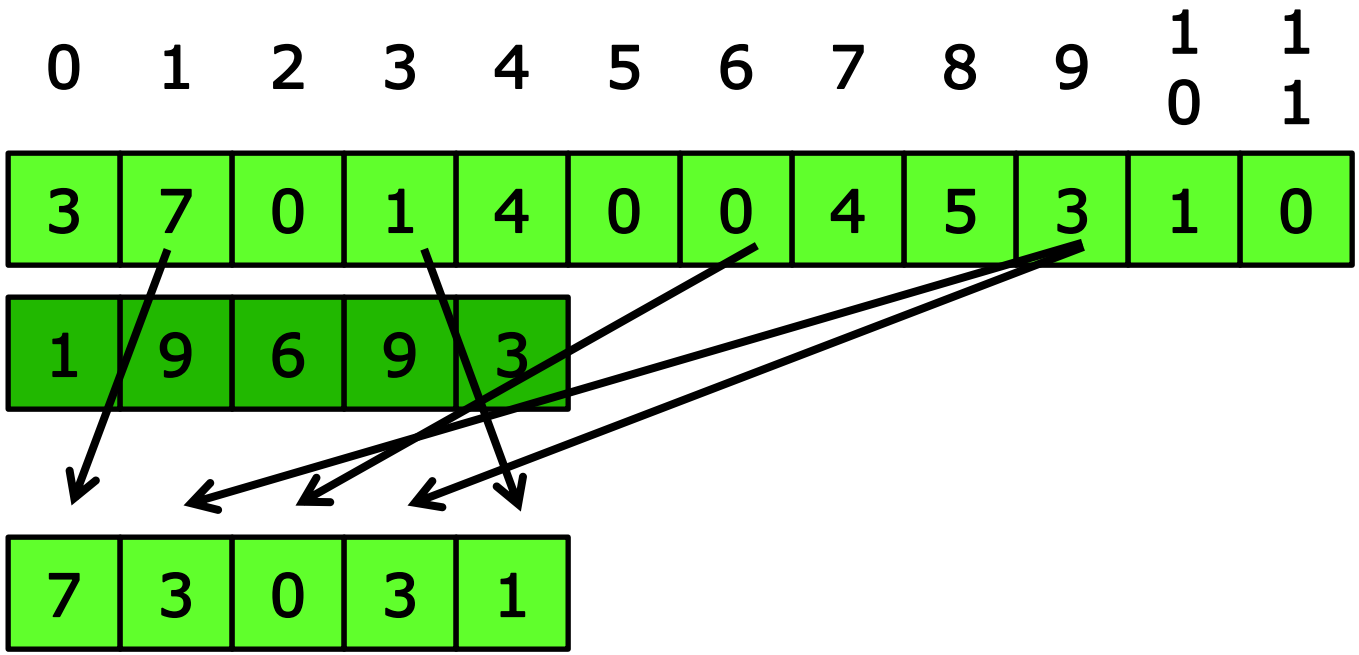
Quiz 1

Given the following locations and source array, use a gather to determine what values should go into the output collection:

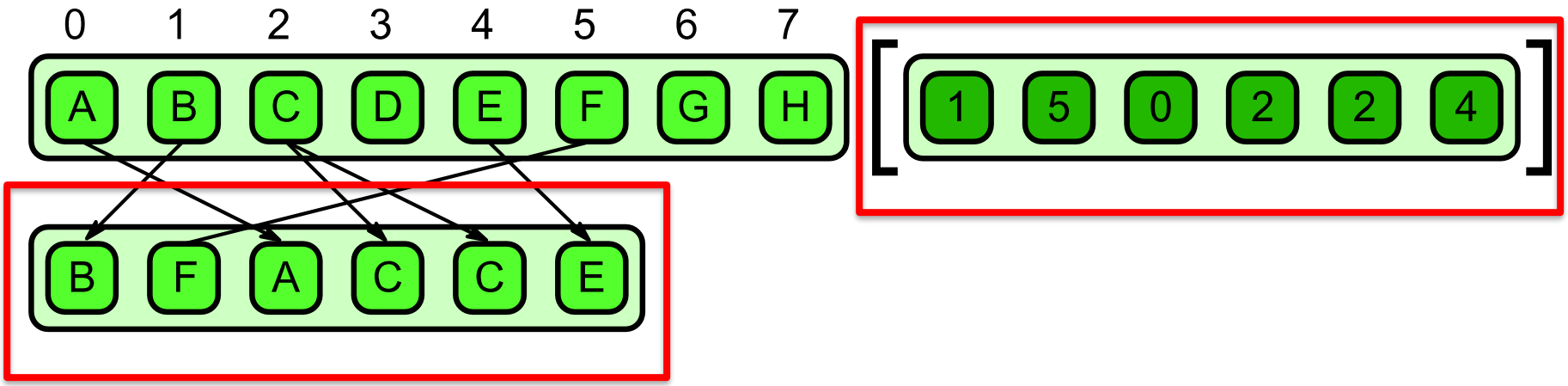


Quiz 1

Given the following locations and source array, use a gather to determine what values should go into the output collection:

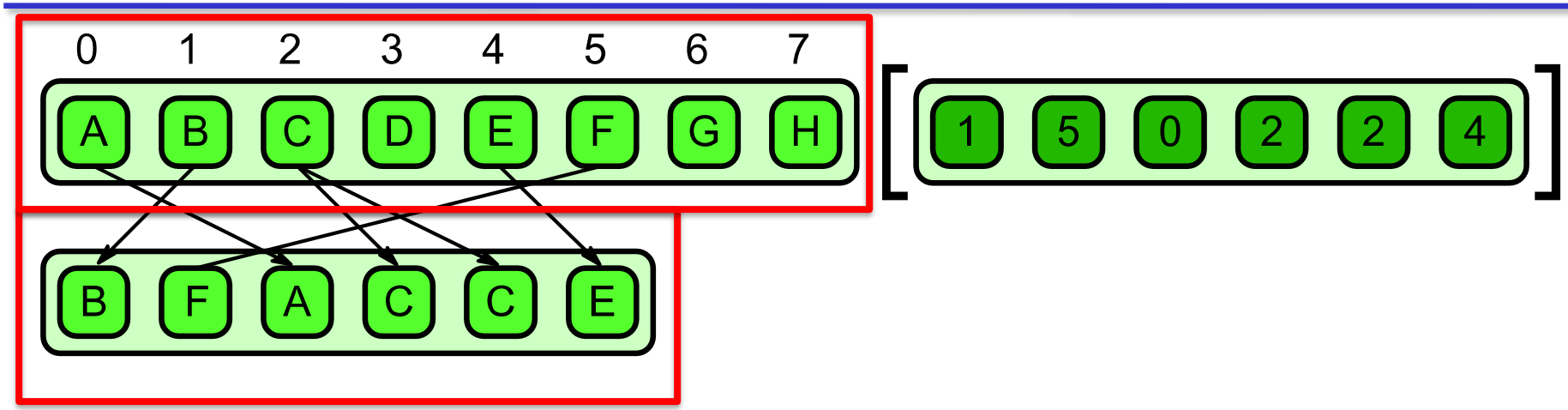


Gather: Array Size



- Output data collection has the same number of elements as the number of indices in the index collection
  - Same dimensionality

Gather: Array Size

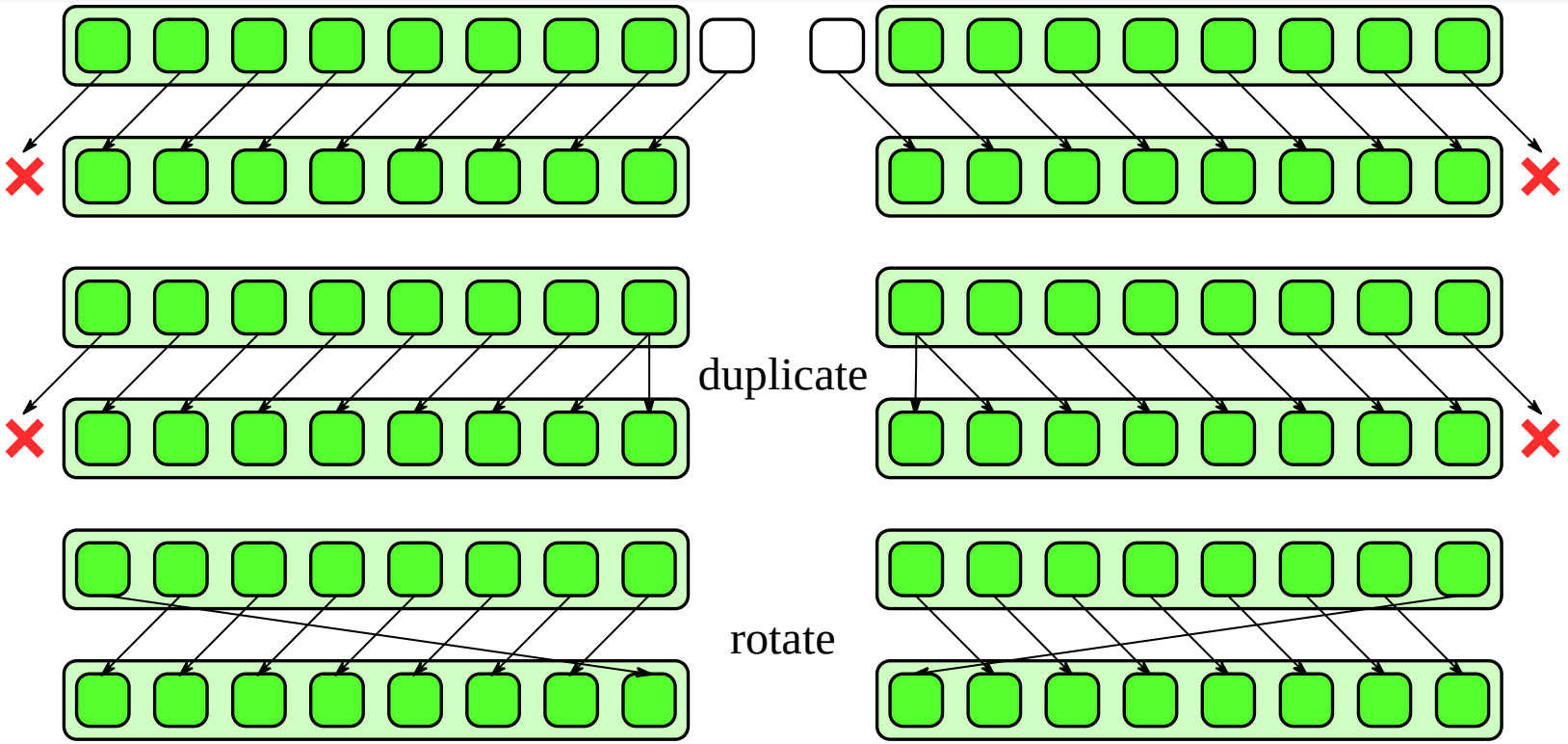


- Output data collection has the same number of elements as the number of indices in the index collection
- Elements of the output collection are the same type as the input data collection

Outline

- Gather Pattern
  - Shifts, Zip, Unzip
- Scatter Pattern
  - Collision Rules: atomic, permutation, merge, priority
- Pack Pattern
  - Split, Unsplit, Bin
  - Fusing Map and Pack
  - Expand
- Partitioning Data
- AoS vs. SoA
- Example Implementation: AoS vs. SoA

Special Case of Gather: Shifts



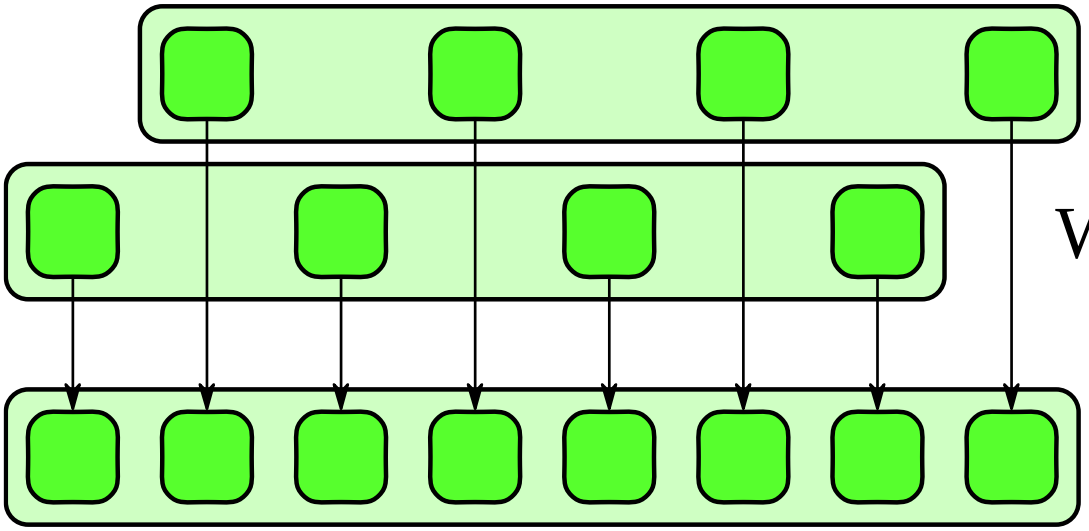
- Moves data to the left or right in memory
- Data accesses are offset by fixed distances

More about Shifts

- Regular data movement
- Variants from how boundary conditions handled
  - Requires "out of bounds" data at edge of the array
  - Options: default value, duplicate, rotate
- Shifts can be handled efficiently with vector instructions because of regularity
  - Shift multiple data elements at the same time
- Shifts can also take advantage of good data locality

Table of Contents

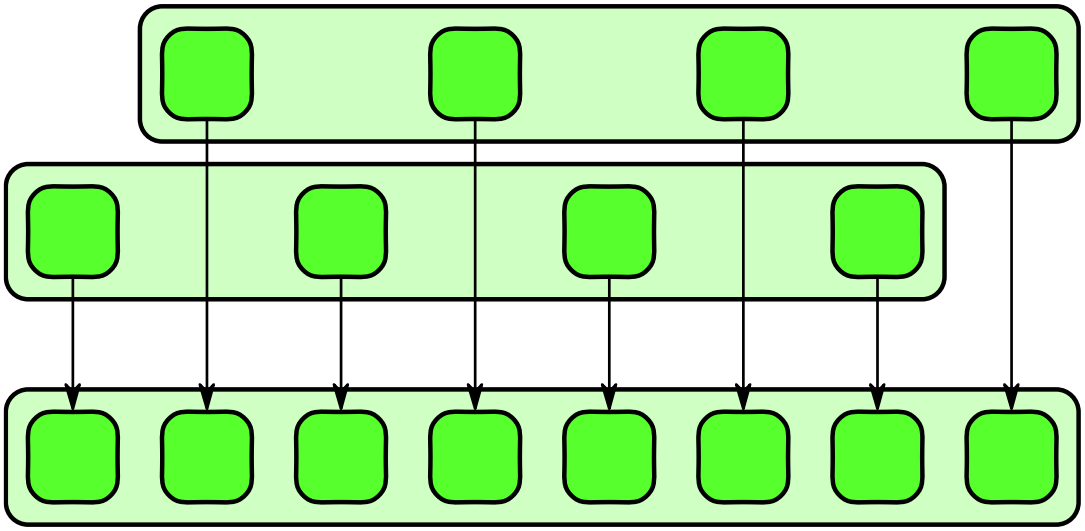
Special Case of Gather: Zip



Where is the parallelism?

- Function is to interleaves data (like a zipper)

Zip Example



Array of Real Parts

Array of Imaginary Parts

Combined Sequence of
Real and Imaginary Parts

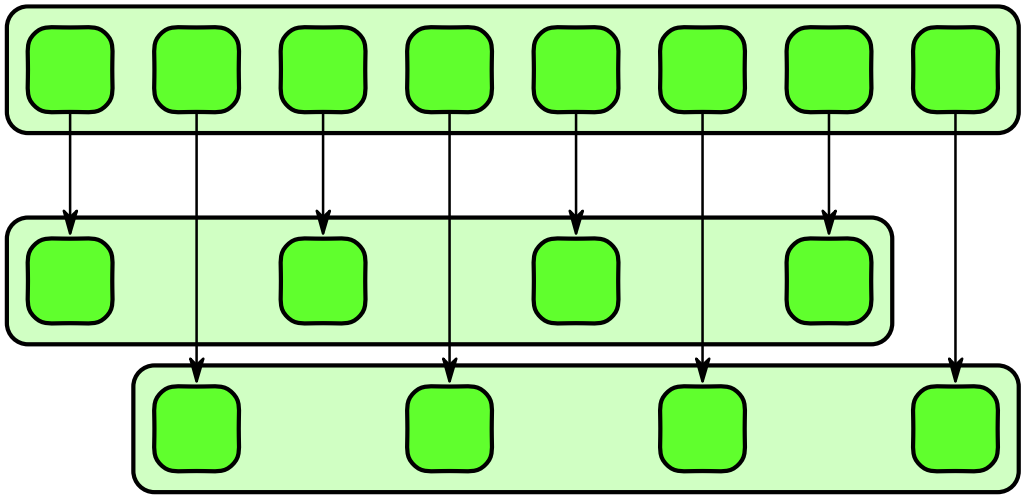
- Given two separate arrays of real parts and imaginary parts
- Use zip to combine them into a sequence of real and imaginary pairs

More about Zip

- Can be generalized to more elements
- Can zip data of unlike types

Table of Contents

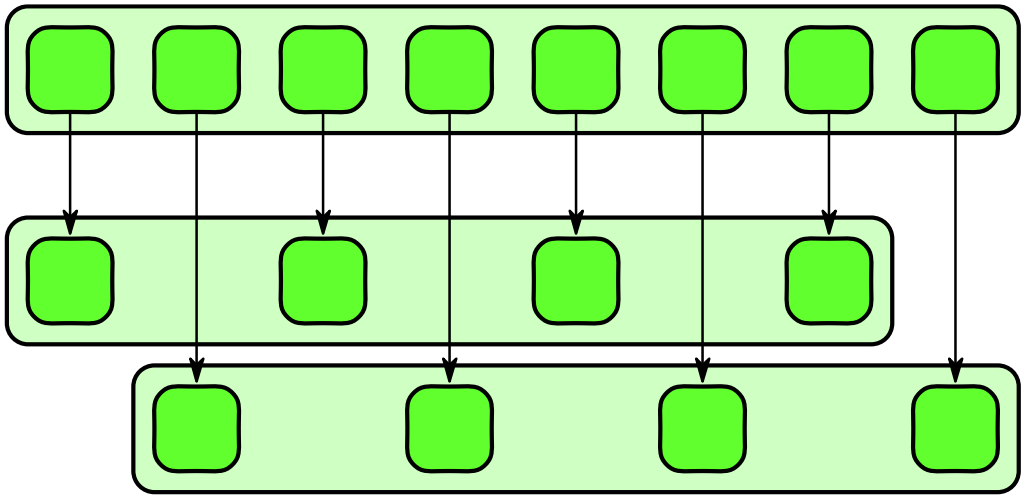
Special Case of Gather: Unzip



Where is the parallelism?

- Reverses a zip
- Extracts sub-arrays at certain offsets and strides from an input array

Unzip Example



Combined Sequence of
Real and Imaginary Parts

Array of Real Parts

Array of Imaginary Parts

- Given a sequence of complex numbers organized as pairs
- Use unzip to extract real and imaginary parts into separate arrays

Contents

- Gather Pattern
  - Shifts, Zip, Unzip
- Scatter Pattern
  - Collision Rules: atomic, permutation, merge, priority
- Pack Pattern
  - Split, Unsplit, Bin
  - Fusing Map and Pack
  - Expand
- Partitioning Data
- AoS vs. SoA
- Example Implementation: AoS vs. SoA

Gather vs. Scatter

Gather

- Combination of map with random **reads**
- Read locations provided as input

Scatter

- Combination of map with random **writes**
- Write locations provided as input
- Race conditions … Why?

Scatter: Serial Implementation


```
1  template<typename Data, typename Idx>
2  void scatter(
3      size_t n, // number of elements in output data collection
4      size_t m, // number of elements in input data and index collection
5      Data a[], // input data collection (m elements)
6      Data A[], // output data collection (n elements)
7      Idx idx[] // input index collection (m elements)
8  ) {
9      for (size_t i = 0; i < m; ++i) {
10         size_t j = idx[i]; // get ith index
11         assert(0 <= j && j < n); // check output array bounds
12         A[j] = a[i]; // perform random write
13     }
14 }
```

Serial implementation of scatter in pseudocode

Scatter: Serial Implementation


```
1  template<typename Data, typename Idx>
2  void scatter(
3      size_t n, // number of elements in output data collection
4      size_t m, // number of elements in input data and index collection
5      Data a[], // input data collection (m elements)
6      Data A[], // output data collection (n elements)
7      Idx idx[] // input index collection (m elements)
8  ) {
9      for (size_t i = 0; i < m; ++i) {
10         size_t j = idx[i]; // get ith index
11         assert(0 <= j && j < n); // check output array bo
12         A[j] = a[i]; // perform random write
13     }
14 }
```

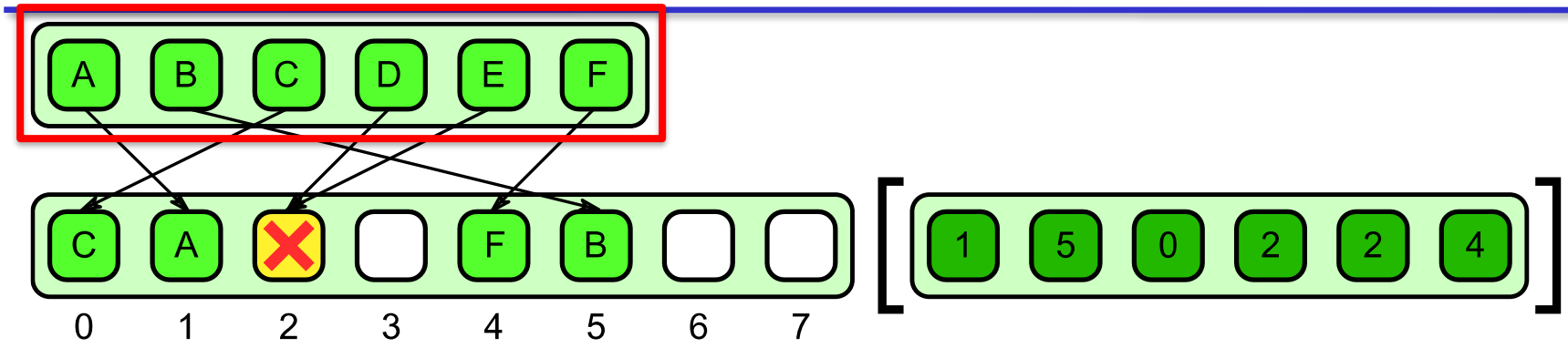
Parallelize over for loop to perform random write

Serial implementation of scatter in pseudocode

Scatter: Defined

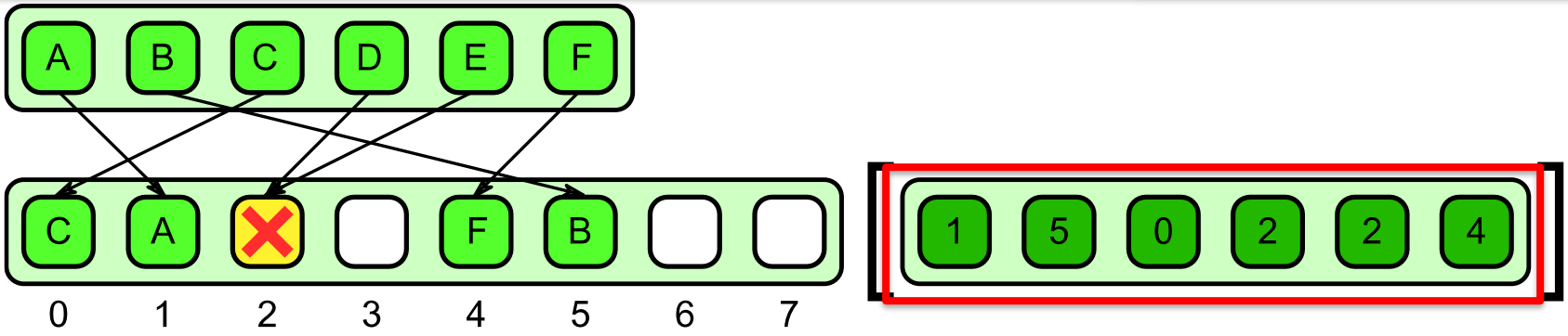
- Results from the combination of a map with a random write
- Writes to the same location are possible
- Parallel writes to the same location are **collisions**

Scatter: Defined



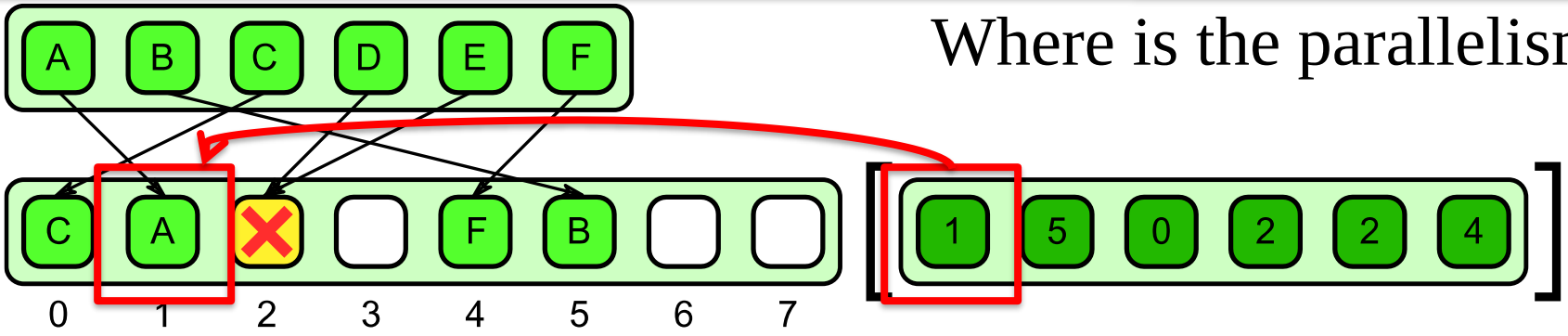
Given a collection of input data

Scatter: Defined



Given a collection of input data
and a collection of write locations

Scatter: Defined



Where is the parallelism?

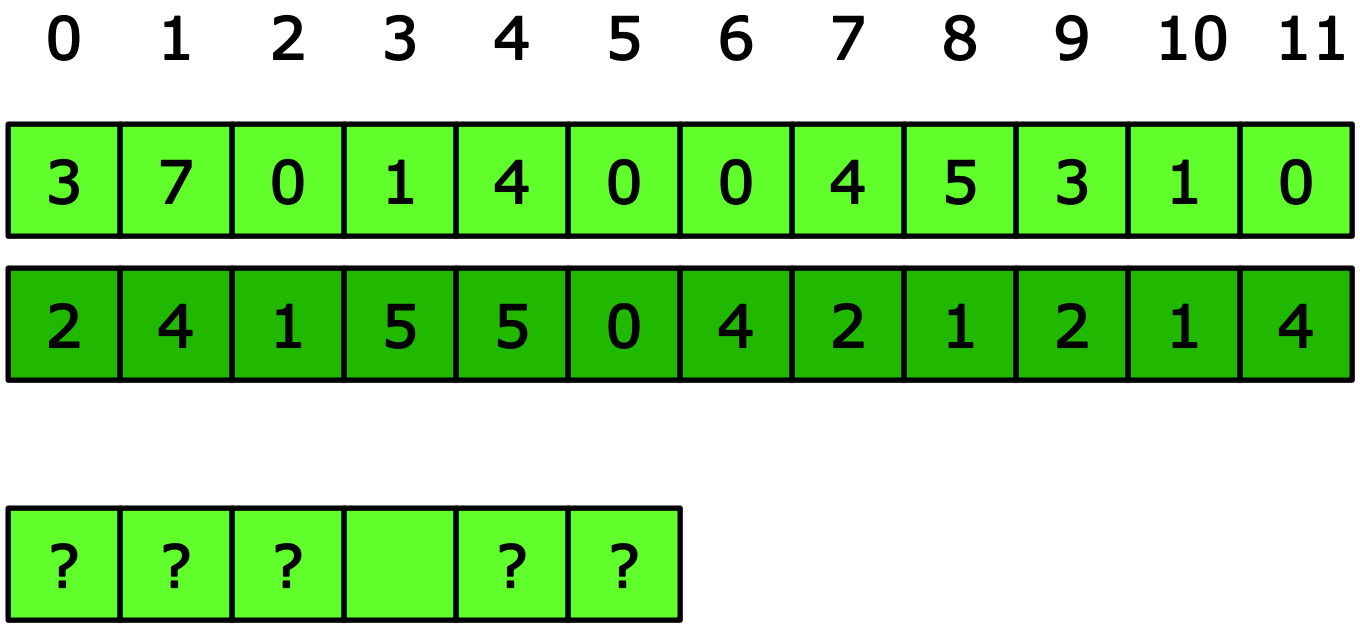
Given a collection of input data
and a collection of write locations
scatter data to the output collection

Problems?

Does the output collection have to be larger in size?

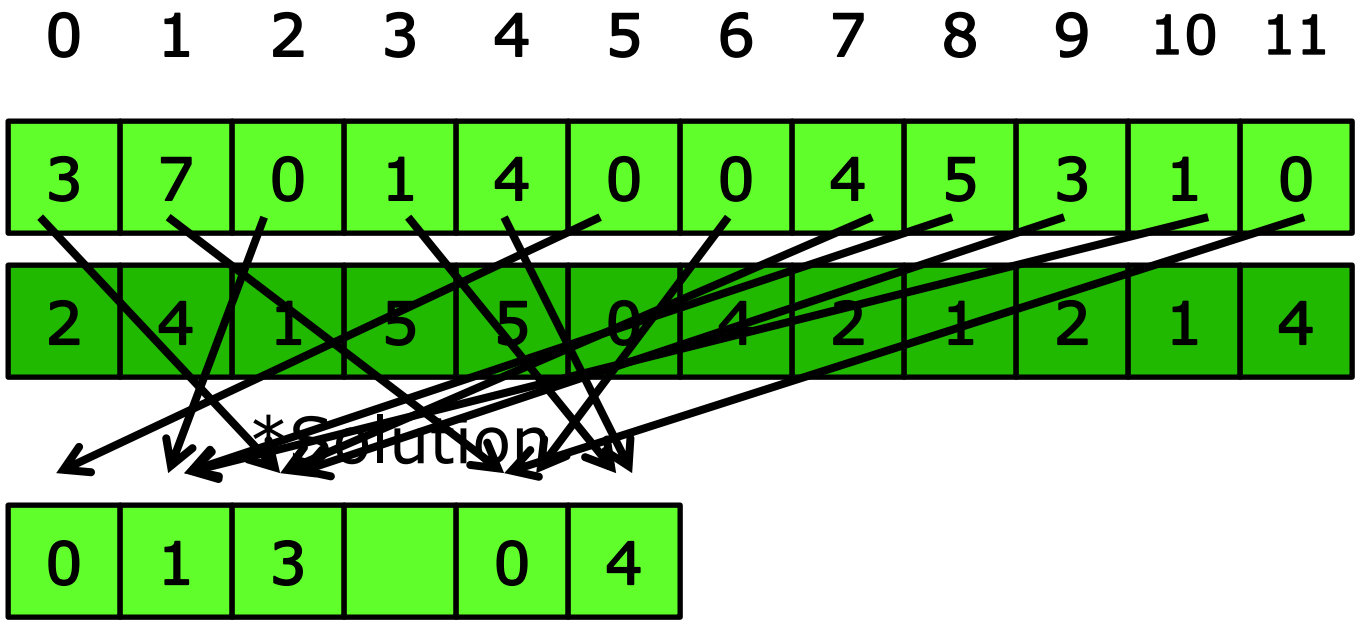
Quiz 2

Given the following locations and source array, what values should go into the input collection:

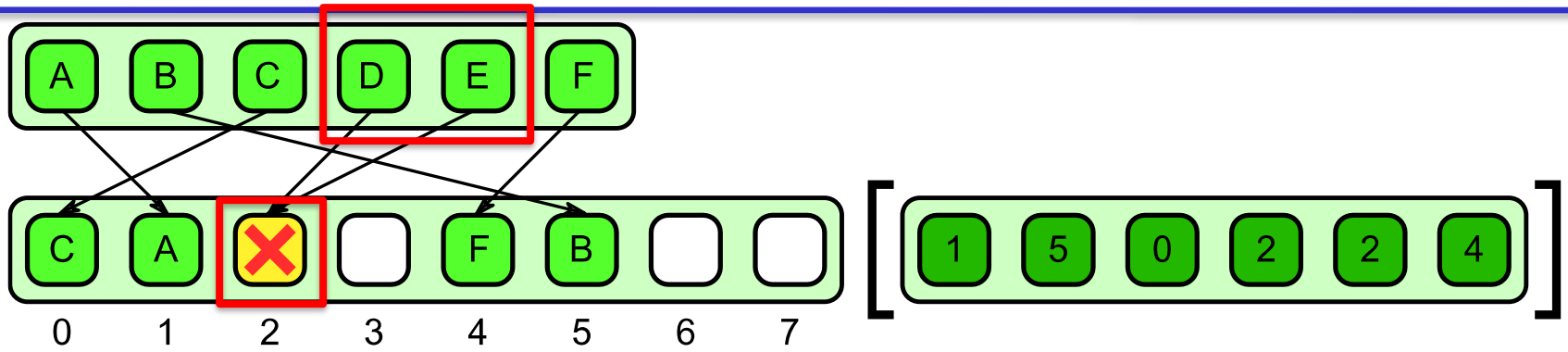


Quiz 2

Given the following locations and source array, what values should go into the input collection:



Scatter: Race Conditions

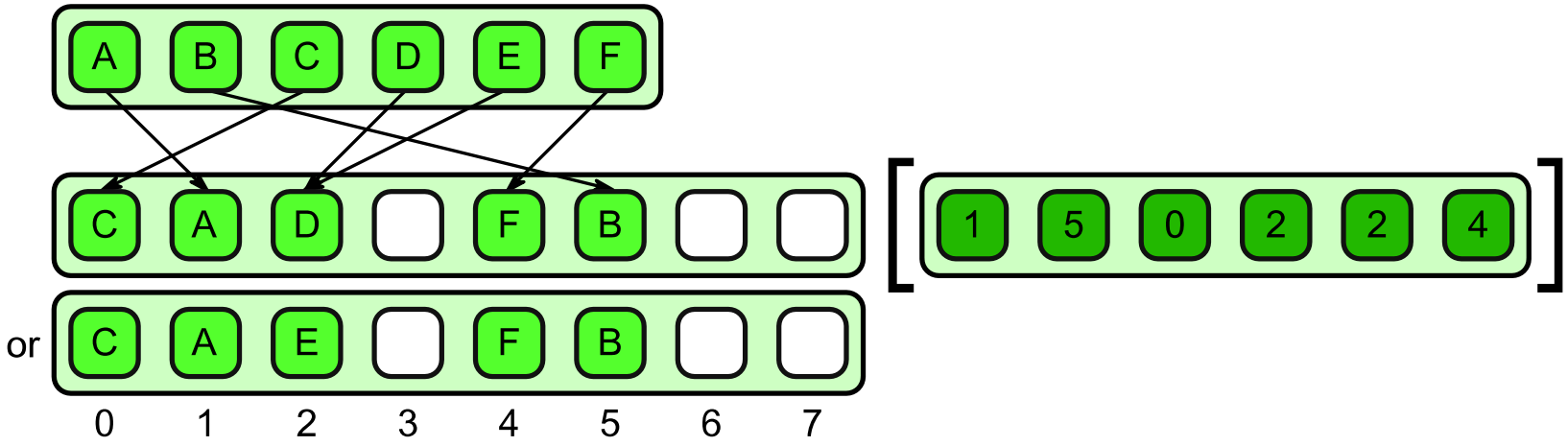


Given a collection of input data
and a collection of write locations
scatter data to the output collection

Race Condition: Two (or more) values being written to the same location in output collection. Result is undefined unless enforce rules. **Need rules to resolve collisions!**

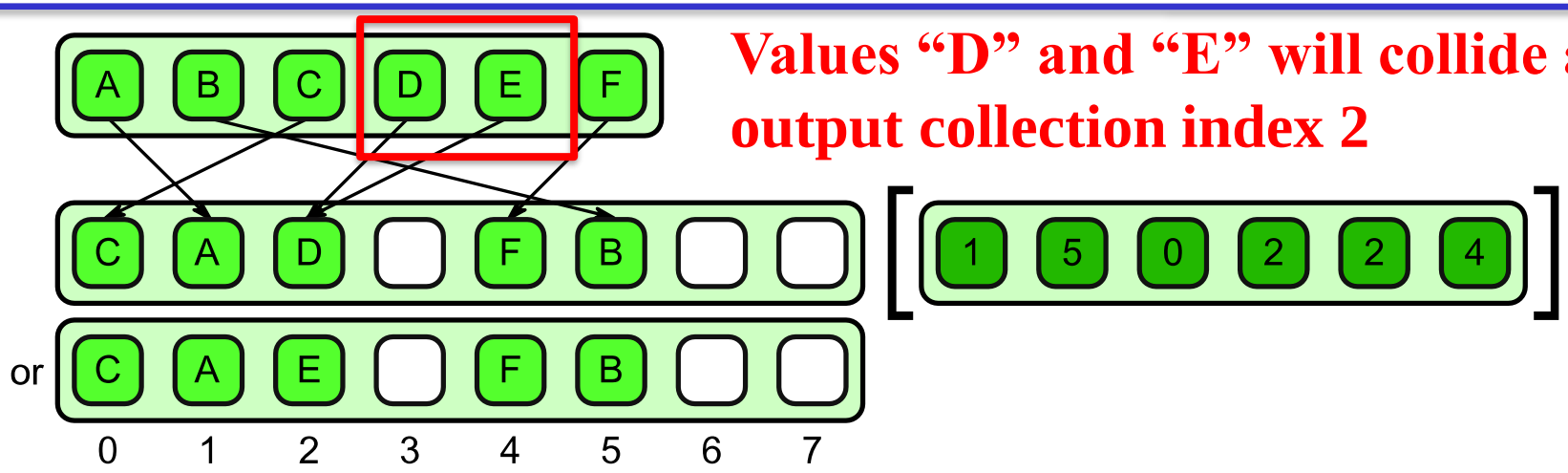
Table of Contents

Collision Resolution: Atomic Scatter



- **Non-deterministic** approach
- Upon collision, one and only one of the values written to a location will be written in its entirety

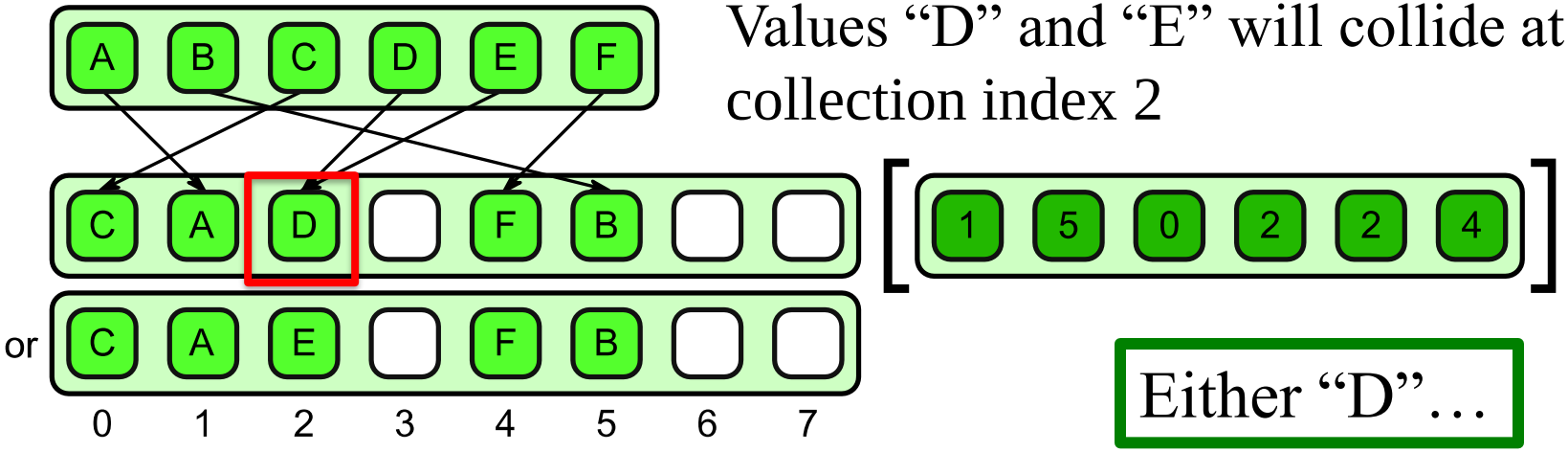
Collision Resolution: Atomic Scatter



Values "D" and "E" will collide at output collection index 2

- Non-deterministic approach
- Upon collision, one and only one of the values written to a location will be written in its entirety

Collision Resolution: Atomic Scatter

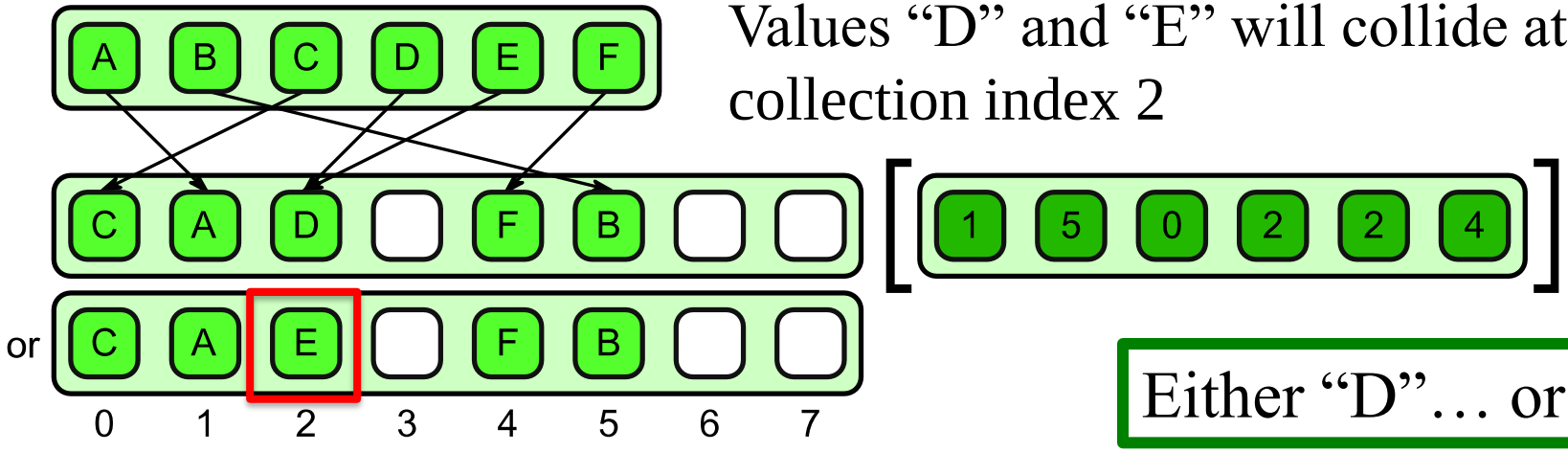


Values “D” and “E” will collide at output collection index 2

Either “D”…

- Non-deterministic approach
- Upon collision, one and only one of the values written to a location will be written in its entirety
- No rule determines which of the input items will be retained

Collision Resolution: Atomic Scatter

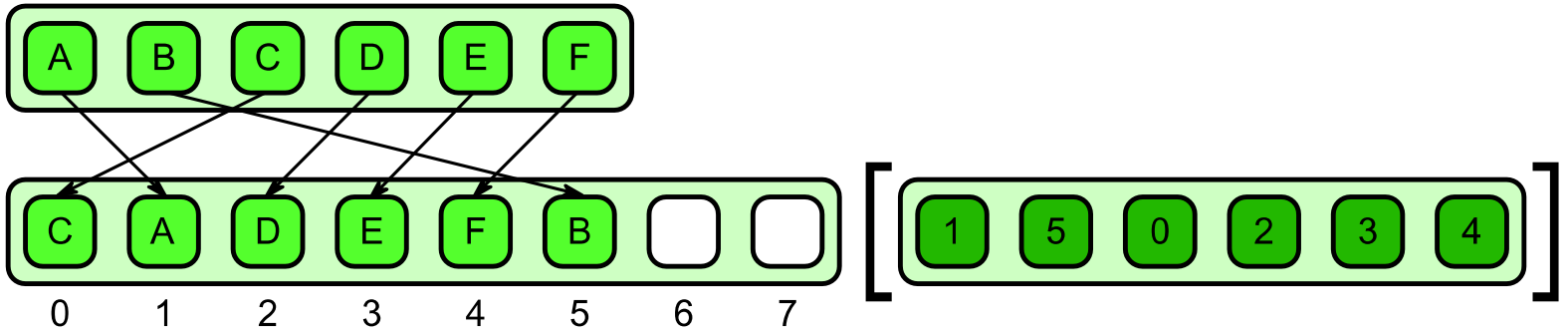


Values “D” and “E” will collide at output collection index 2

Either “D”… or “E”

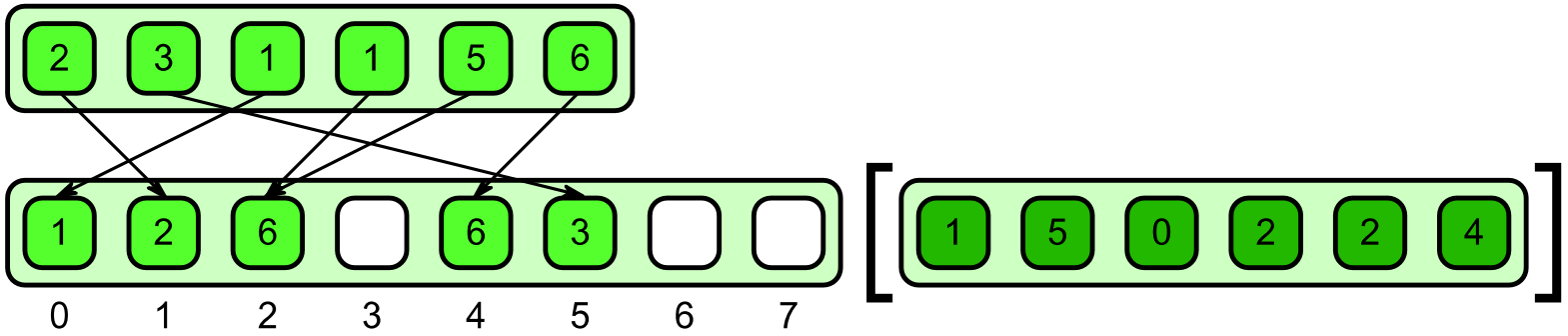
- Non-deterministic approach
- Upon collision, one and only one of the values written to a location will be written in its entirety
- No rule determines which of the input items will be retained

Collision Resolution: Permutation Scatter



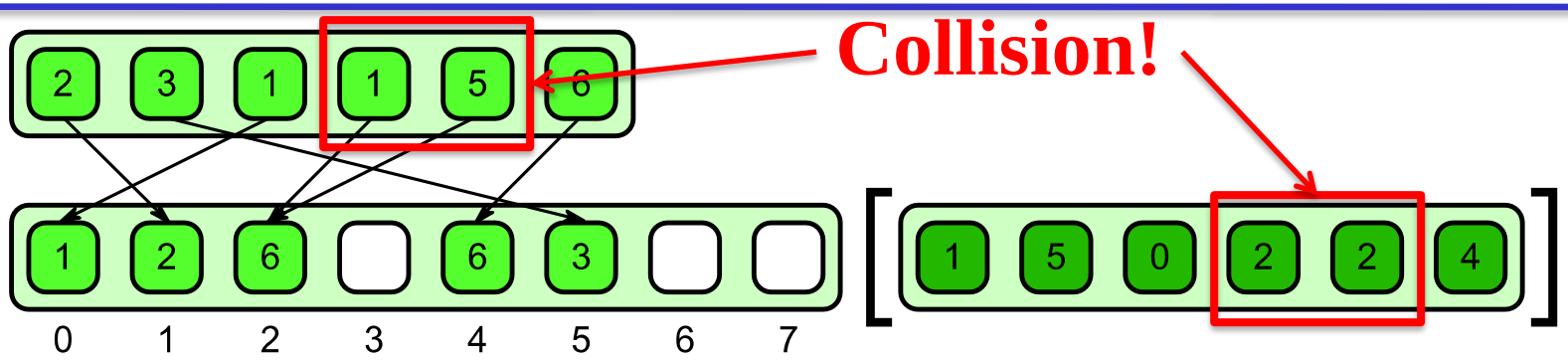
- Pattern simply states that collisions are **illegal**
  - Output is a permutation of the input
- Check for collisions in advance
  → turn scatter into gather
- Examples
  - FFT scrambling, matrix/image transpose, unpacking

Collision Resolution: Merge Scatter



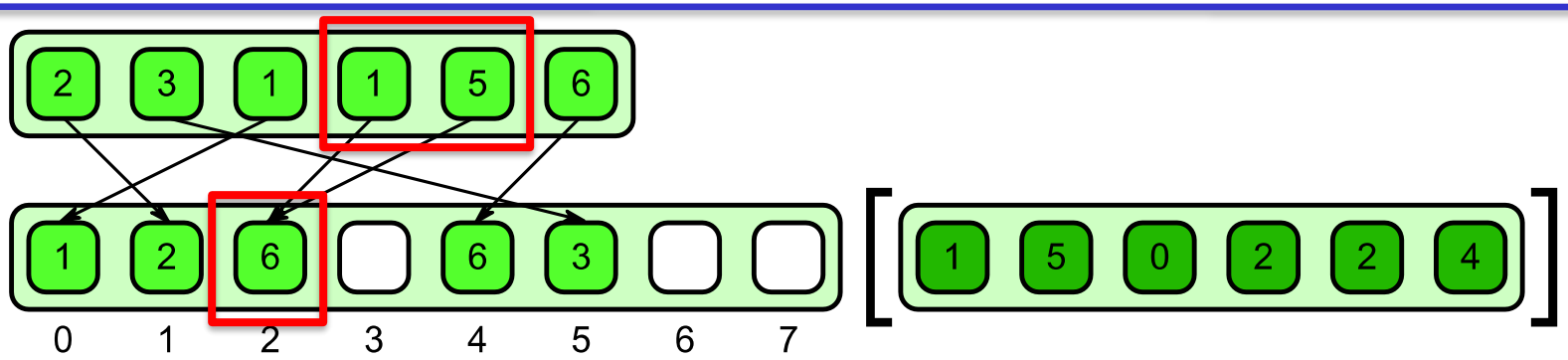
- Associative and commutative operators are provided to merge elements in case of a collision

Collision Resolution: Merge Scatter



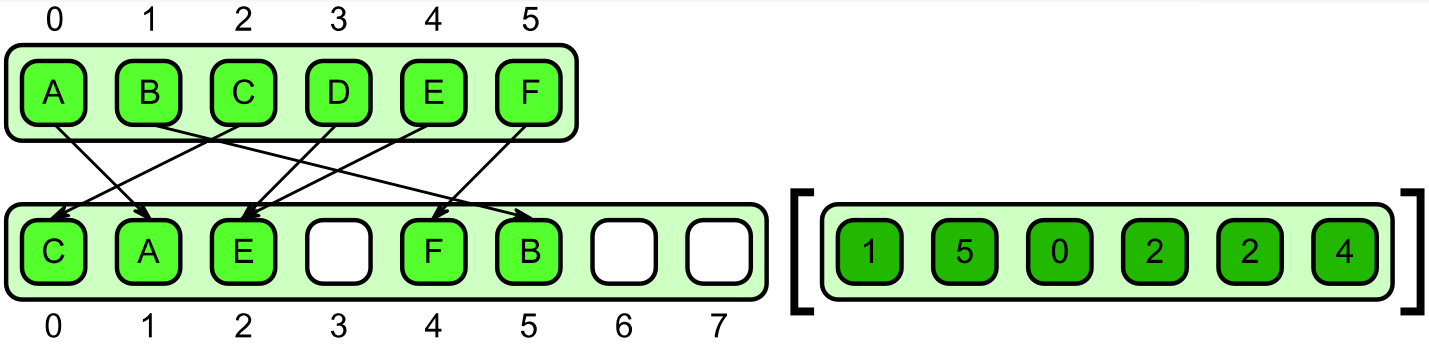
- Associative and commutative operators are provided to merge elements in case of a collision

Collision Resolution: Merge Scatter



- Associative and commutative operators are provided to merge elements in case of a collision
- Use addition as the merge operator
- Both associative and commutative properties are required since scatters to a particular location could occur in any order

Collision Resolution: Priority Scatter



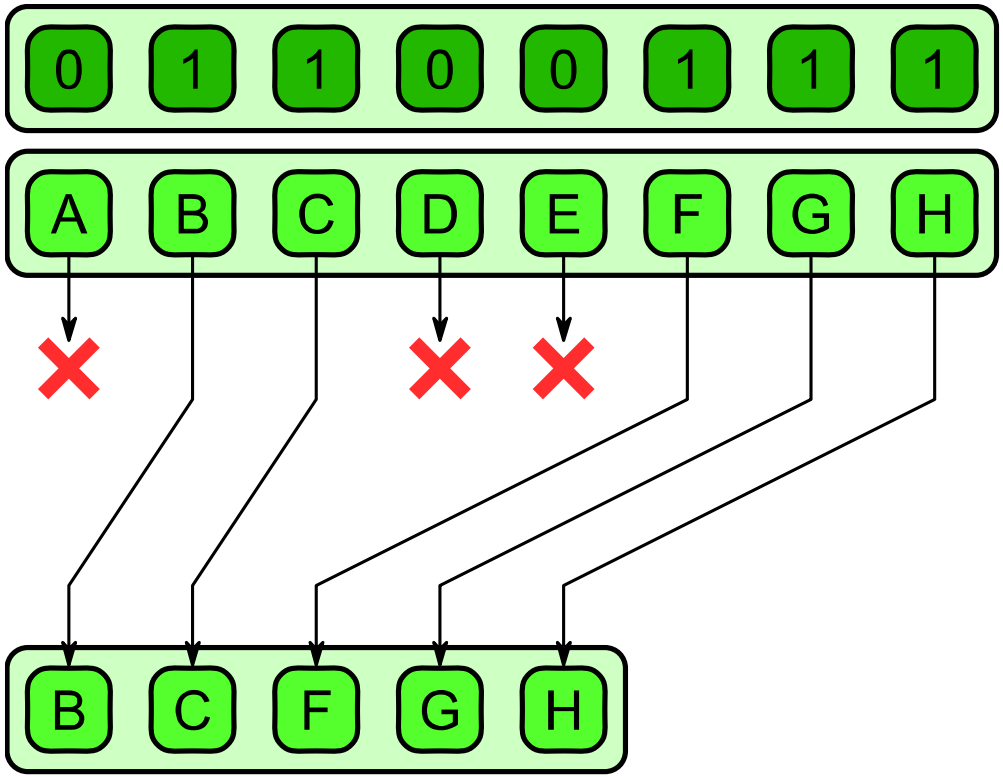
- Every element in the input array is assigned a priority based on its position
- Priority is used to decide which element is written in case of a collision
- Example
  - 3D graphics rendering

Converting Scatter to Gather

- Scatter is a more expensive than gather
  - Writing has cache line consequences
  - May cause additional reading due to cache conflicts
  - **False sharing** is a problem that arises
    - writes from different cores go to the same cache line
- Can avoid problems if addresses are know "in advance"
  - Allows optimizations to be applied
  - Convert addresses for a scatter into those for a gather
  - Useful if the same pattern of scatter address will be used repeatedly so the cost is amortized

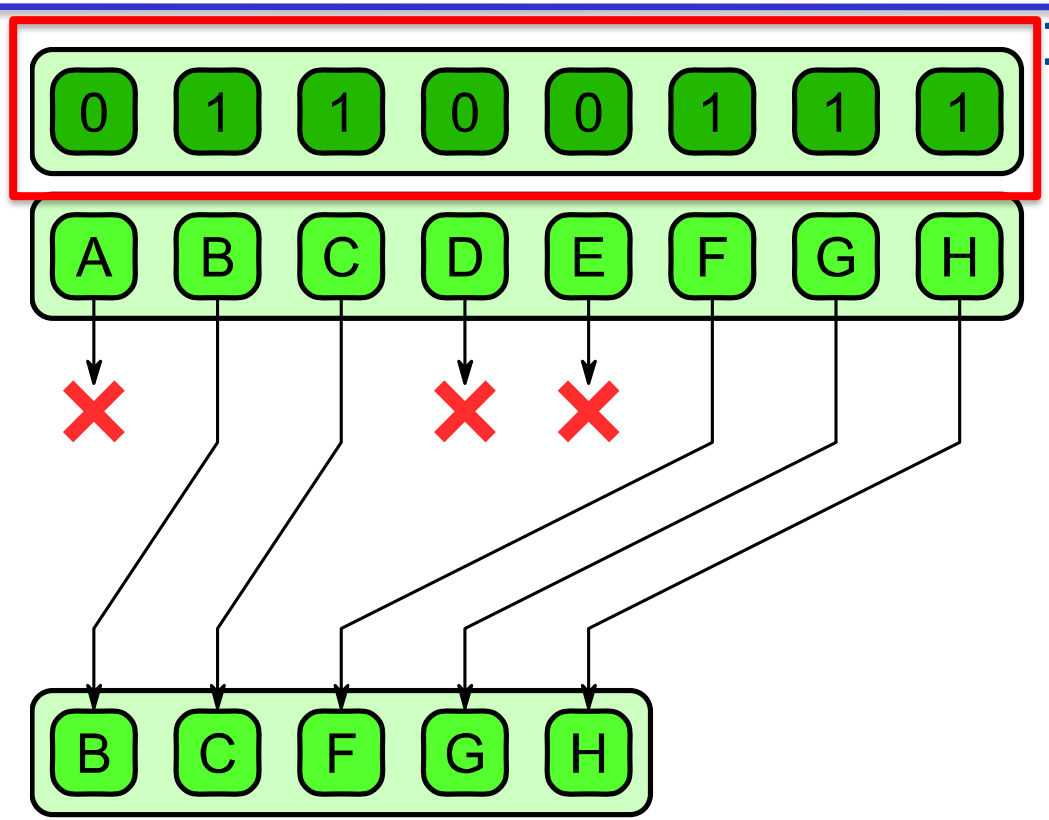
Table of Contents

Pack: Defined



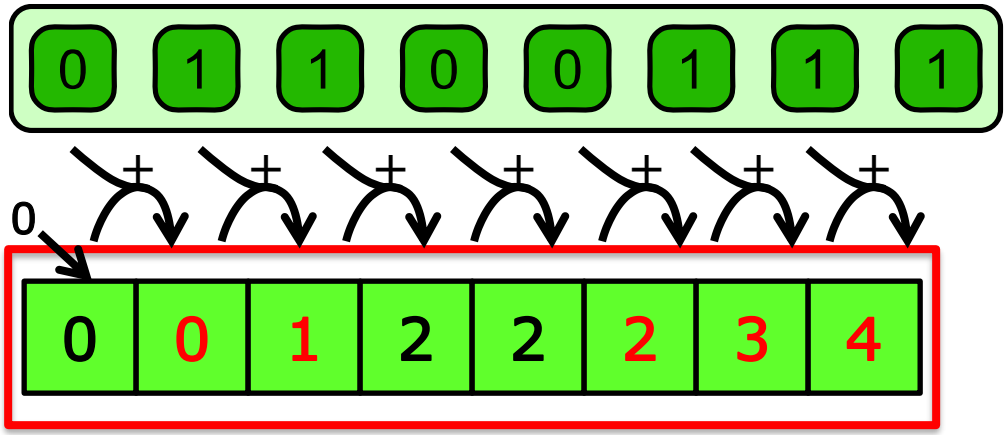
- Used to eliminate unused elements from a collection
- Retained elements are moved so they are contiguous in memory
- Goal is to improve the performance … How?

Pack Algorithm



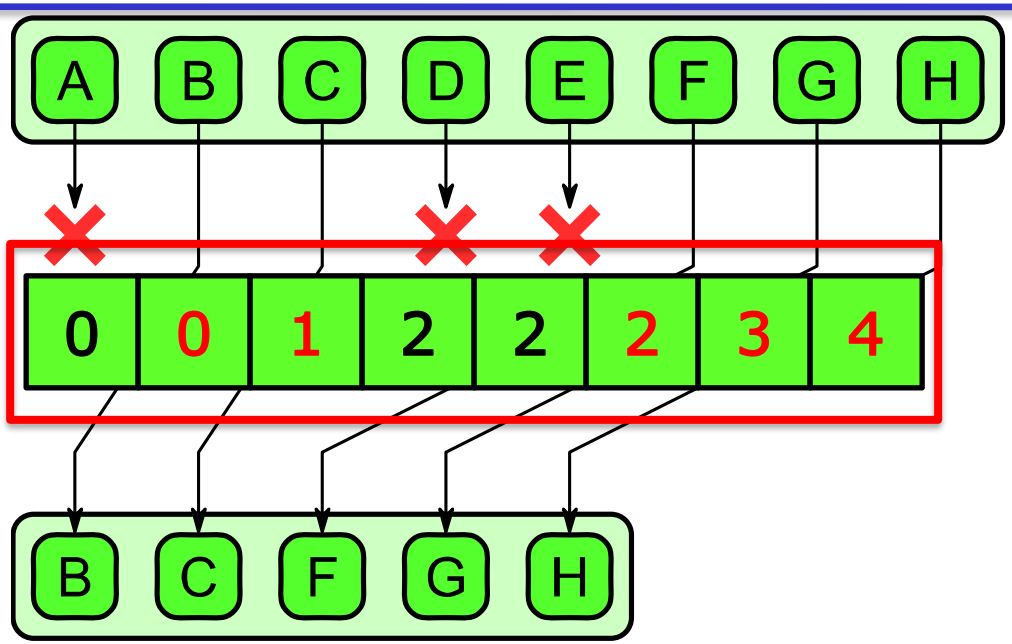
1. Convert input array of Booleans into integer 0's and 1's

Pack Algorithm



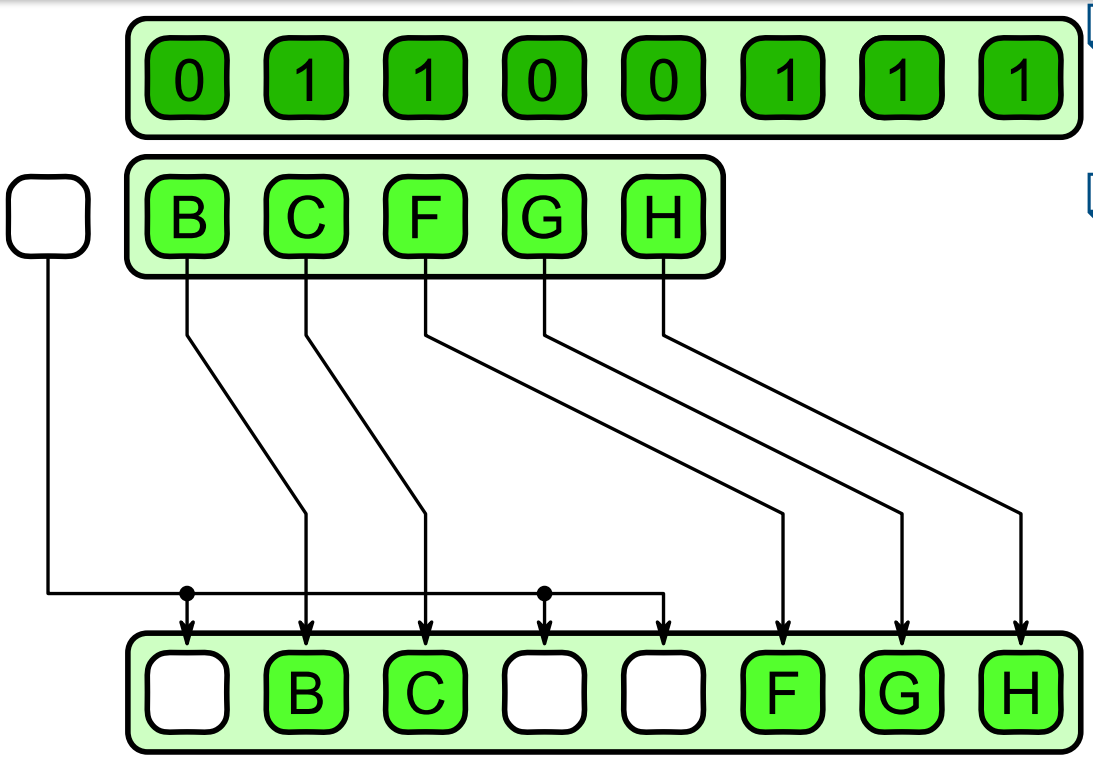
1. Convert input array of Booleans into integer 0's and 1's
2. **Exclusive scan of this array with the addition operation**

Pack Algorithm



1. Convert input array of Booleans into integer 0's and 1's
2. Exclusive scan of this array with the addition operation
3. **Write values to output array based on offsets**

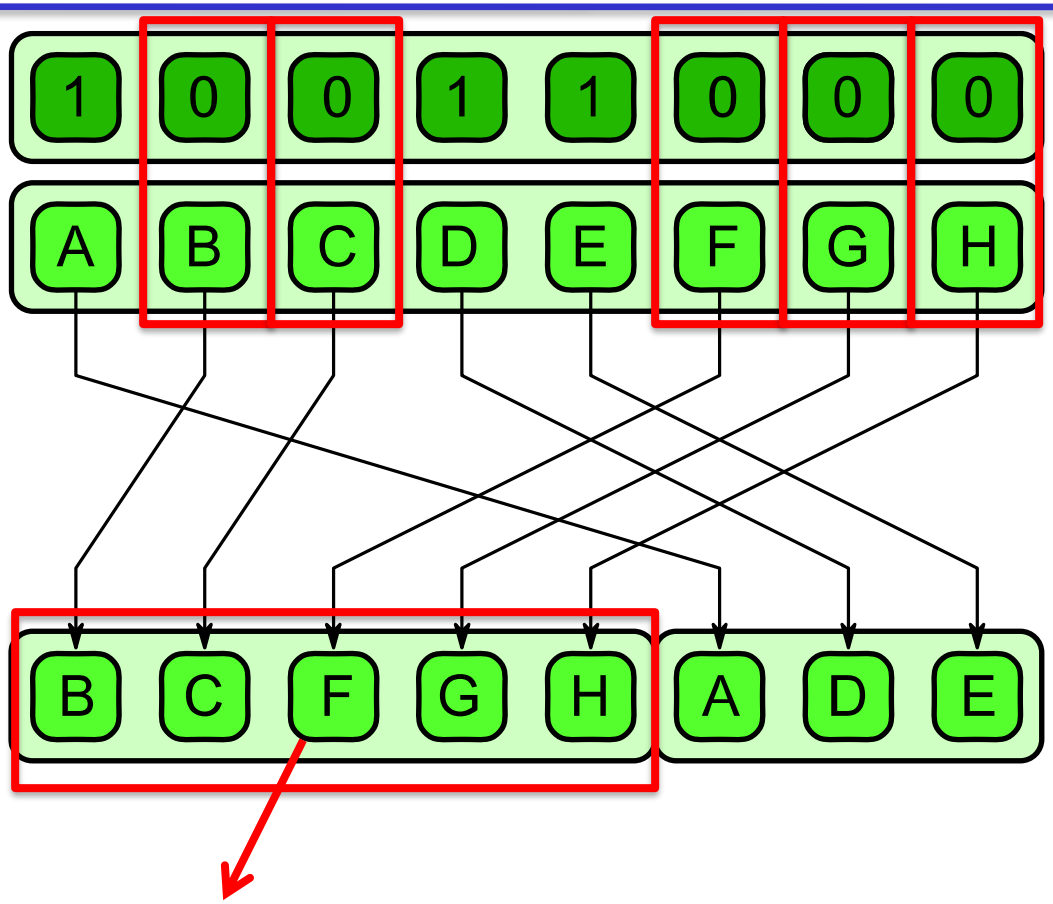
Unpack: Defined



- Inverse of pack operation
- Given the same data on which elements were kept and which were discarded, spread elements back in their original locations

Table of Contents

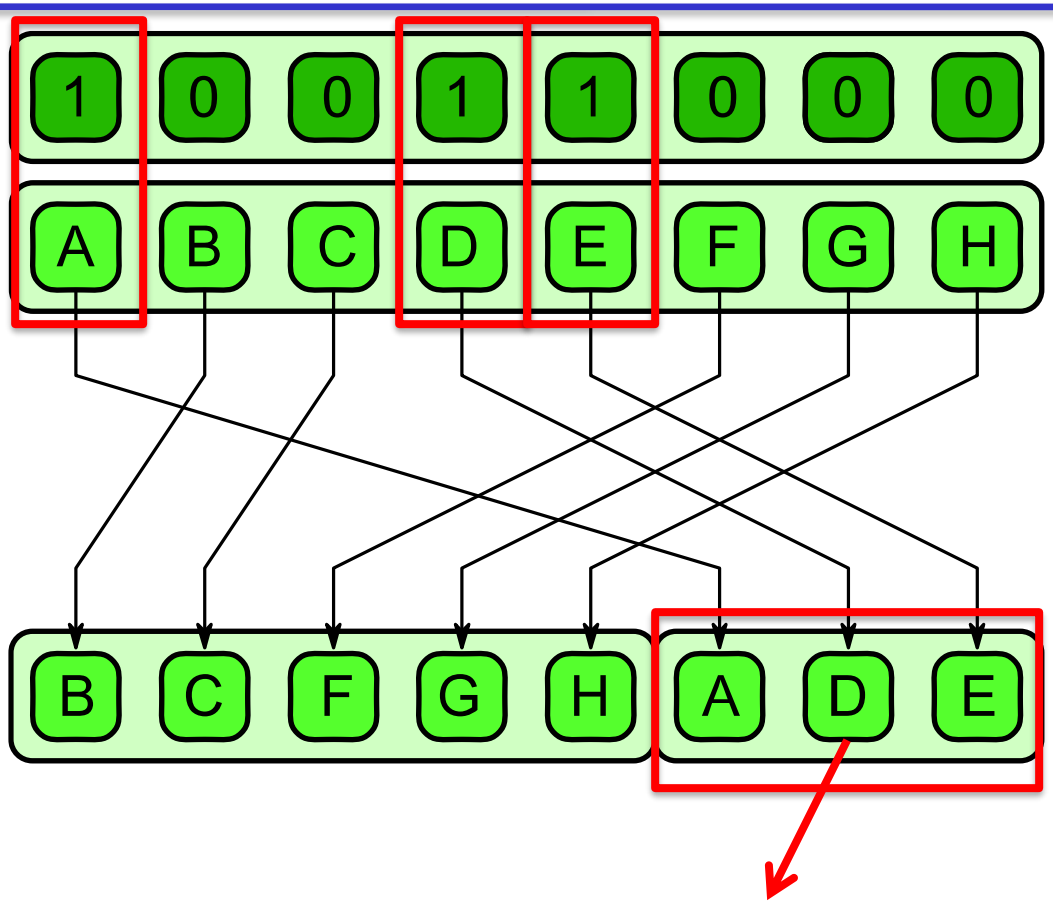
Generalization of Pack: Split



Upper half of output collection: values equal to 0

- Generalization of pack pattern
- Elements are moved to upper or lower half of output collection based on some state
- Does not lose information like pack

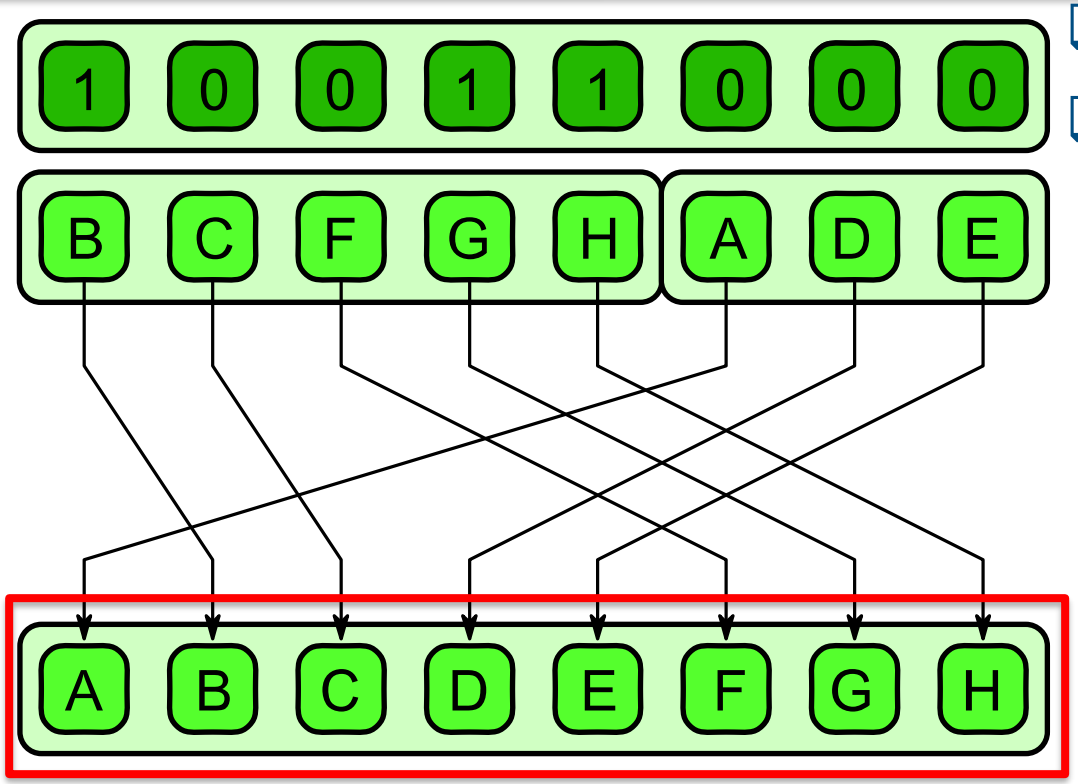
Generalization of Pack: Split



- Generalization of pack pattern
- Elements are moved to upper or lower half of output collection based on some state
- Does not lose information like pack

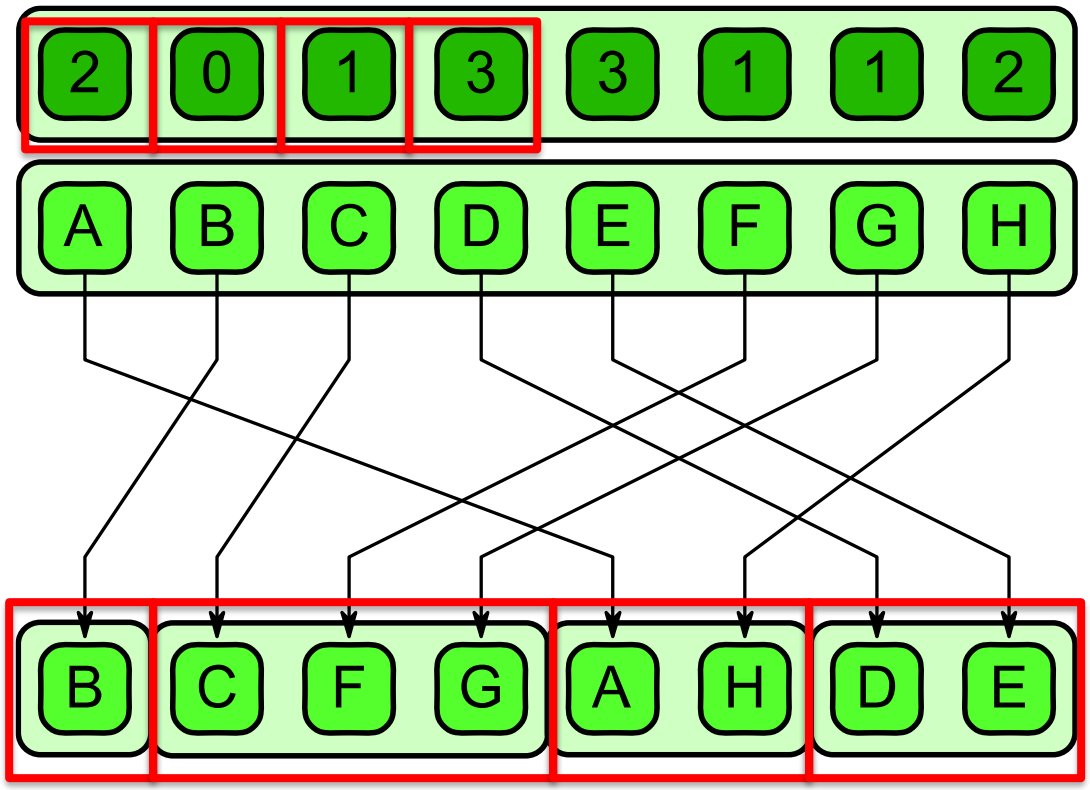
Lower half of output collection: values equal to 1

Generalization of Pack: Unsplit



- Inverse of split
- Creates **output collection** based on original input collection

Generalization of Pack: Bin

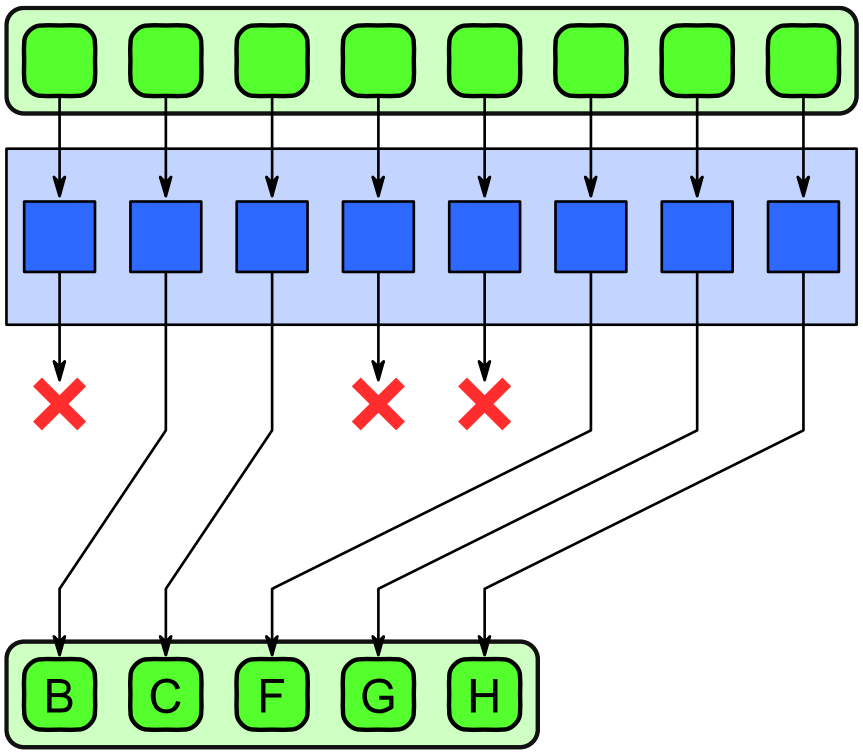


- Generalized split to support more categories (>2)
- Examples
  - Radix sort
  - Pattern classification

4 different categories = 4 bins

Table of Contents

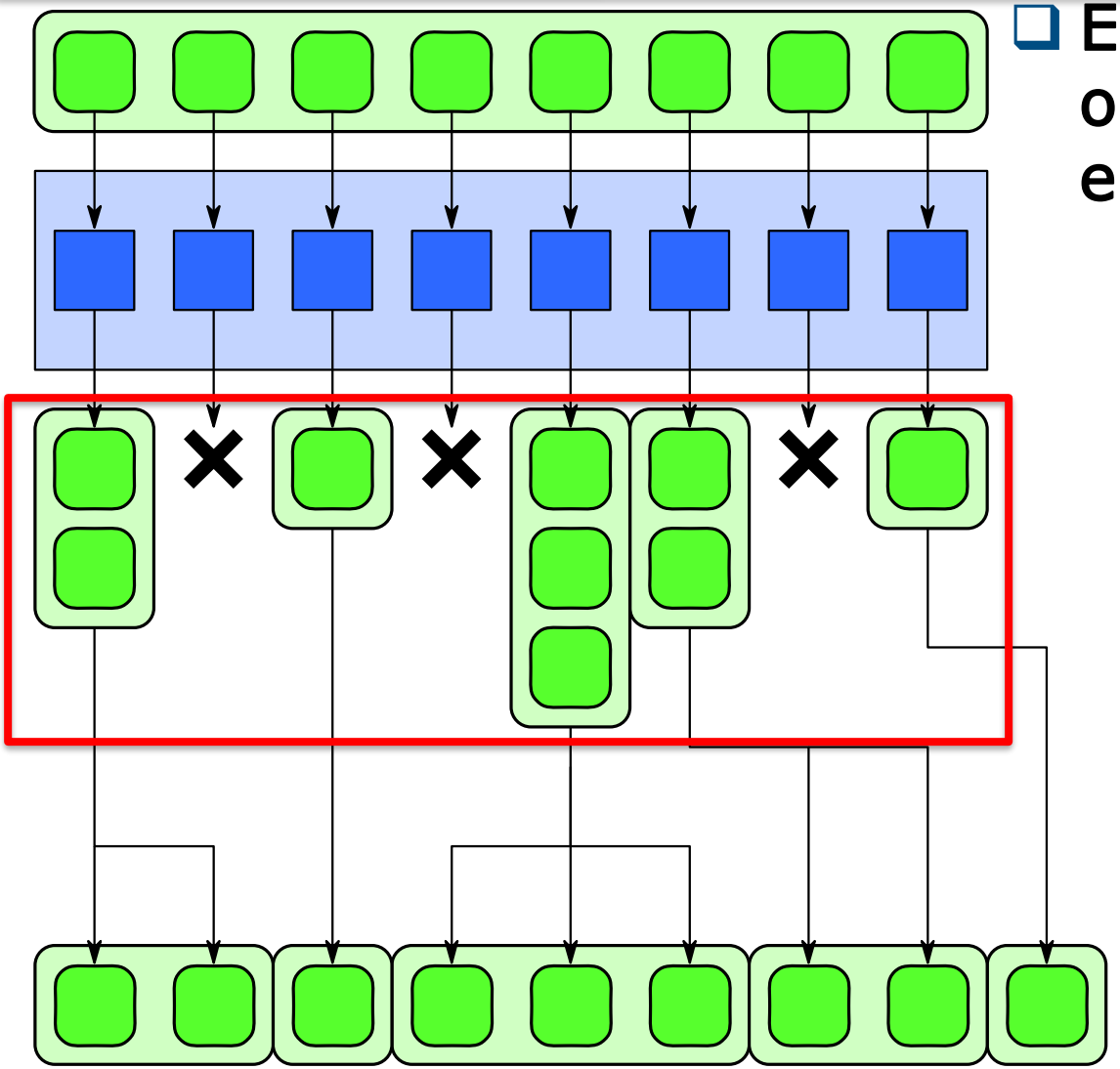
Fusion of Map and Pack



- Advantageous if most of the elements of a map are discarded
- **Map** checks pairs for collision
- **Pack** stores only actual collisions
- Output BW ~ results reported, not number of pairs tested
- Each element can output 0 or 1 element

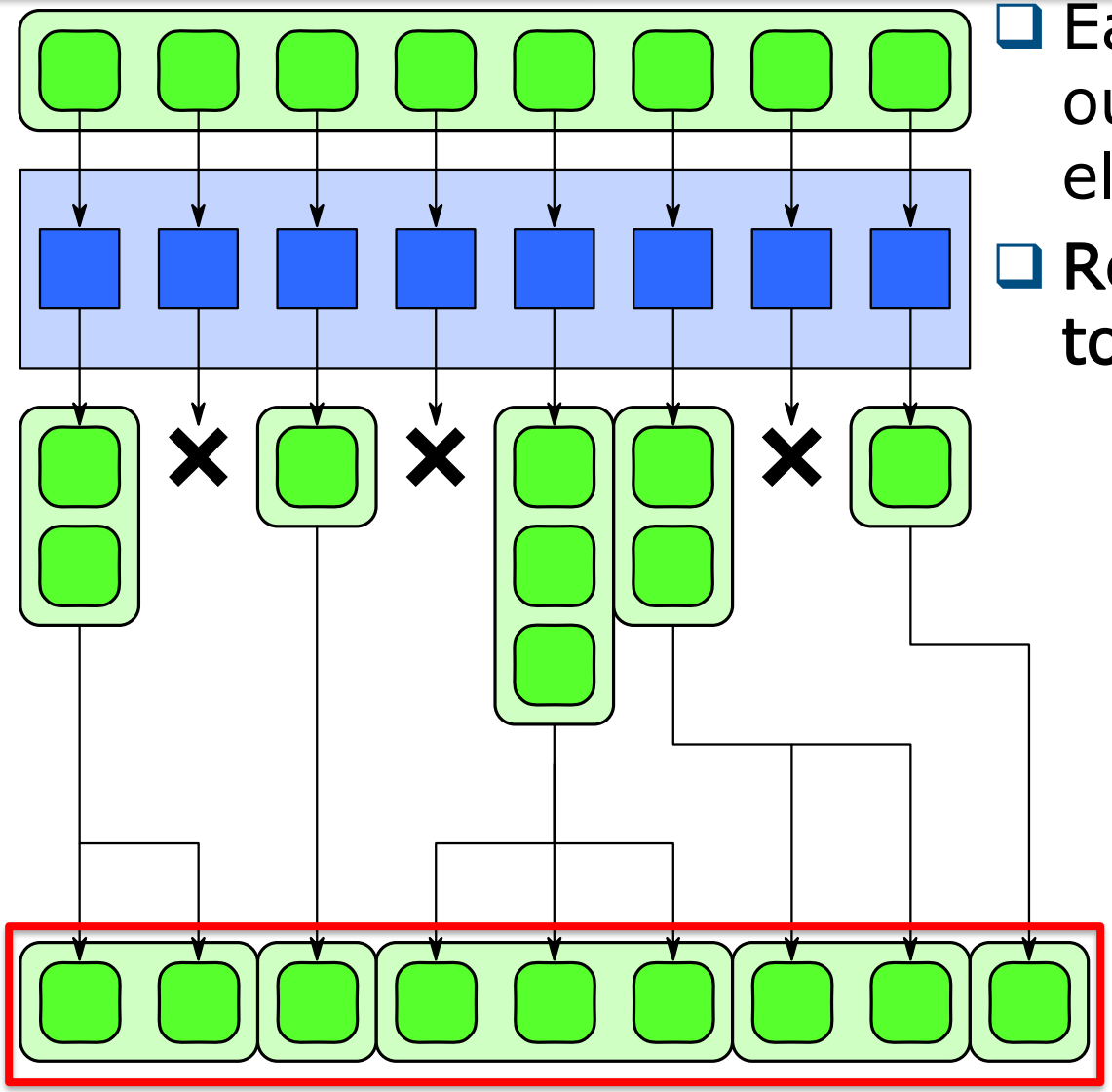
Table of Contents

Generalization of Pack: Expand



- Each element can output any number of elements

Generalization of Pack: Expand



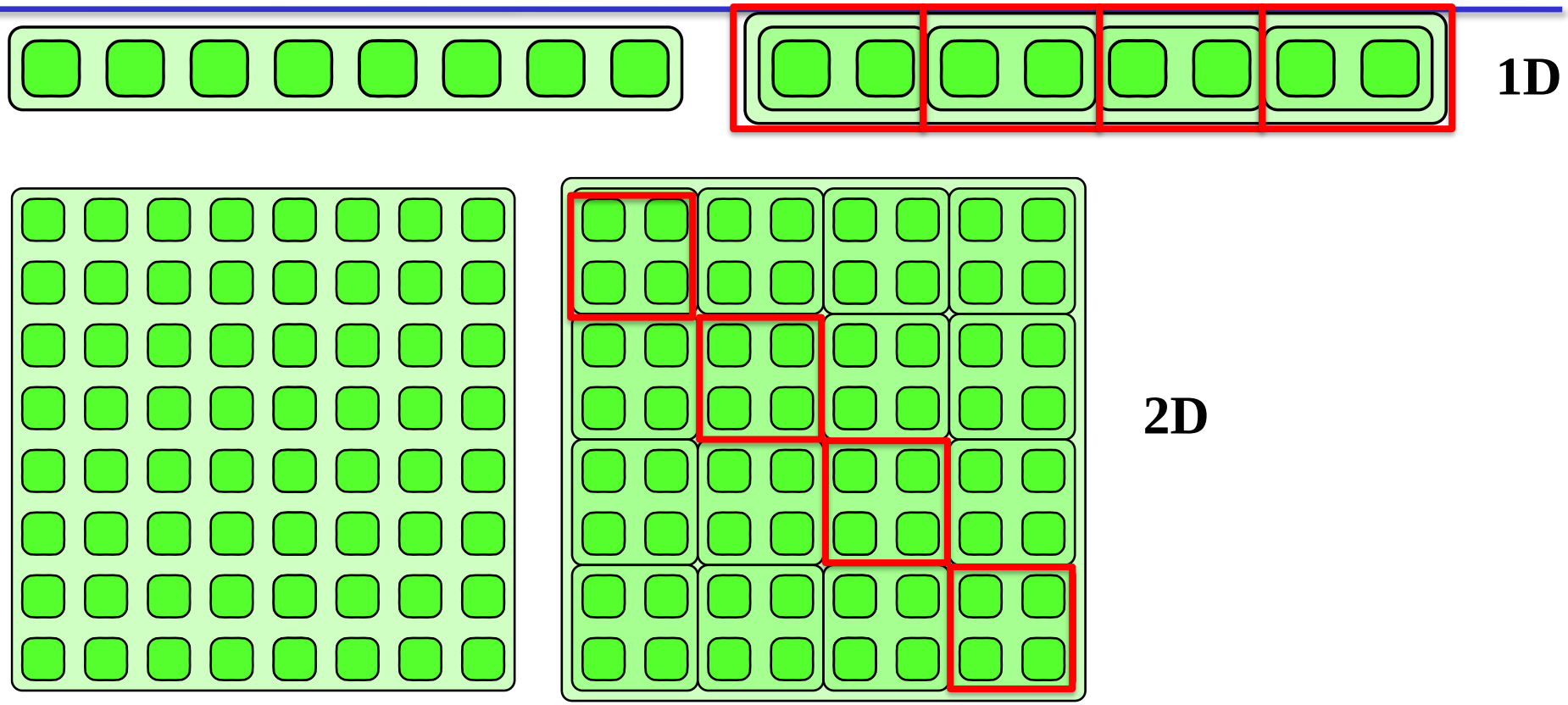
- Each element can output any number of elements
- **Results are fused together in order**

Table of Contents

Parallelizing Algorithms

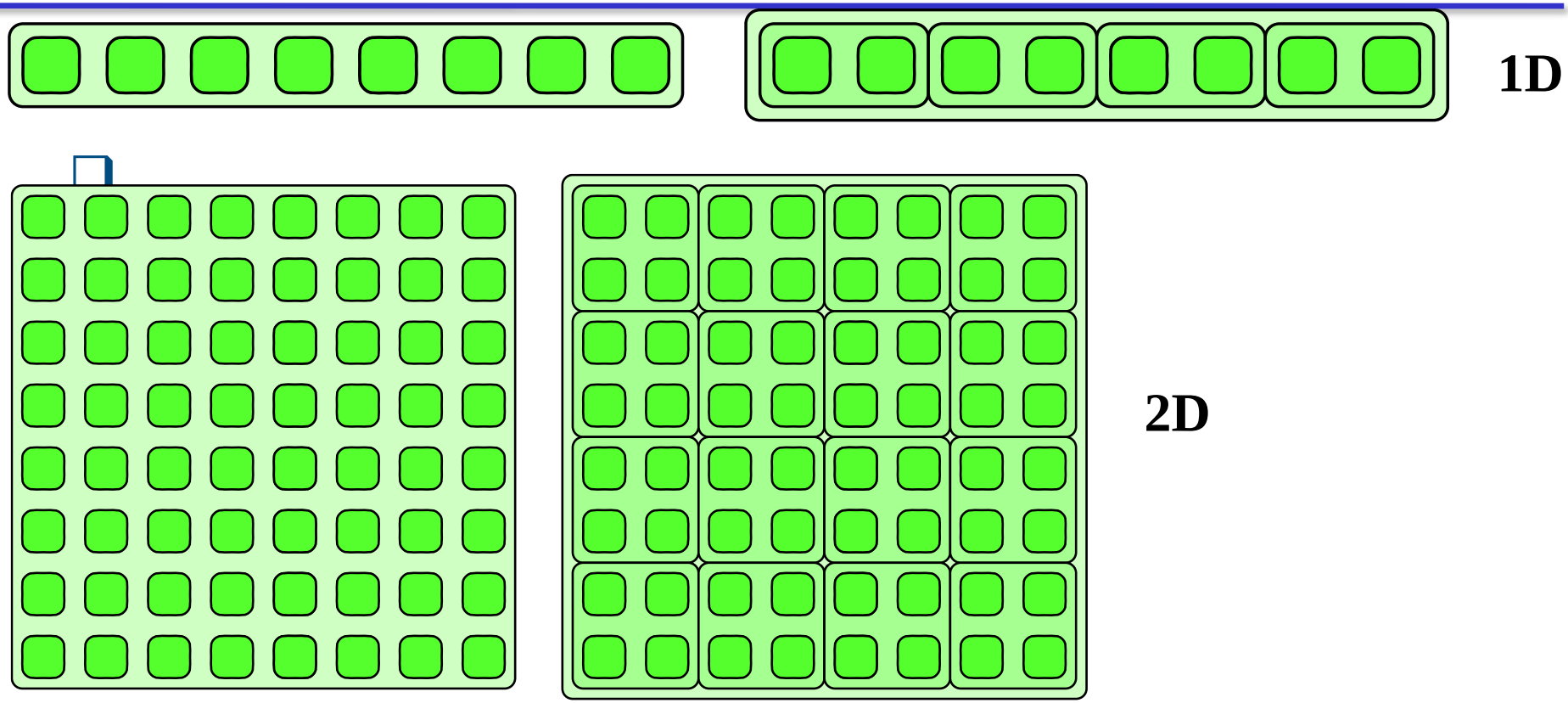
- Common strategy:
  1. Divide up the computational domain into sections
  2. Work on the sections individually
  3. Combine the results
- Methods
  - Divide-and-conquer
  - Fork-join (discussed in Chapter 8)
  - Geometric decomposition
  - Partitions
  - Segments

Partitioning



- Data is divided into
  - **non-overlapping**
  - **equal-sized** regions

Partitioning



- Data is divided into
  - non-overlapping
  - equal-sized regions

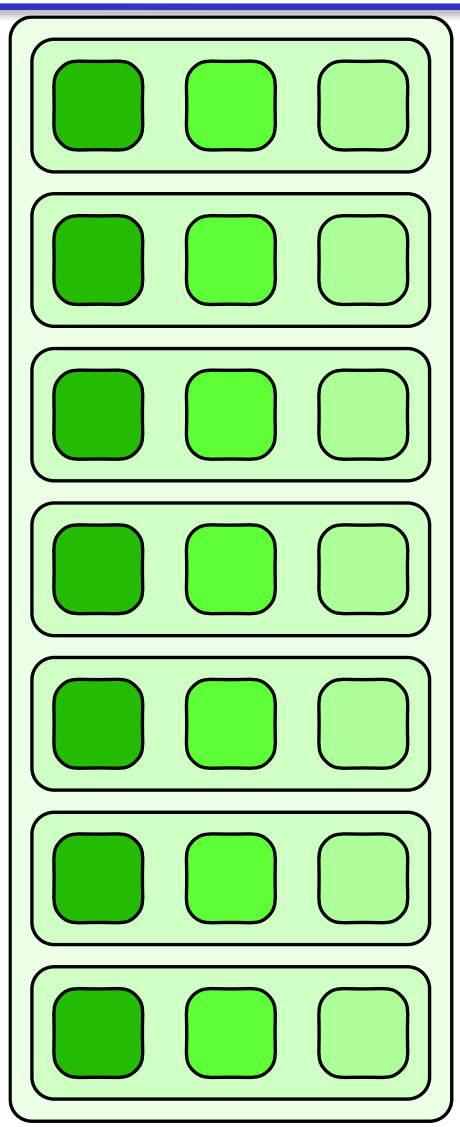
Avoid write conflicts and race conditions

Segmentation

- Data is divided into **non-uniform** non-overlapping regions
- Start of each segment can be marked using:
  - Array of integers
  - Array of Boolean flags

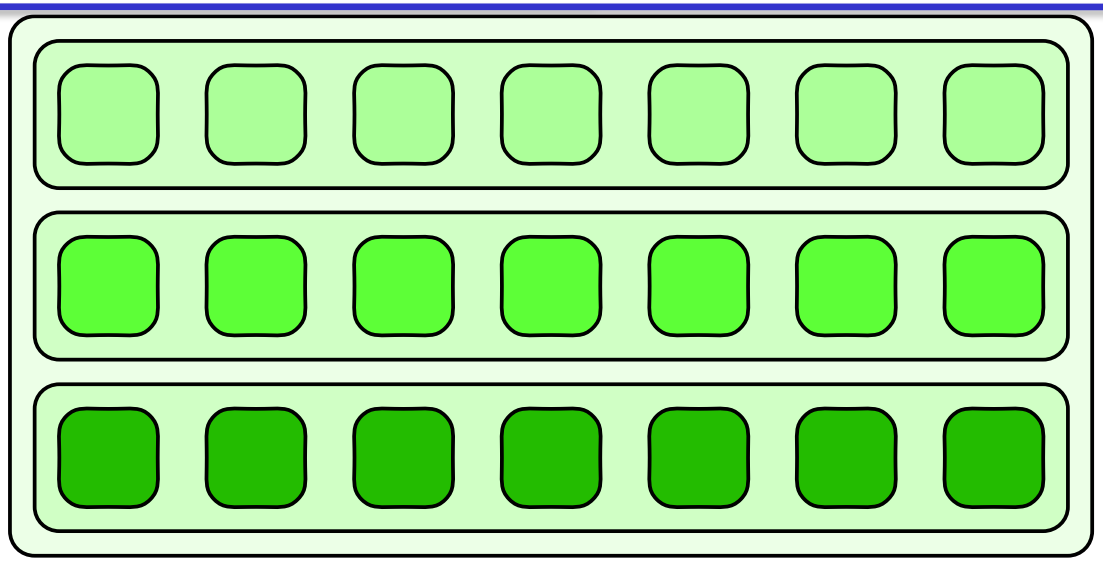
Table of Contents

Array of Structures (AoS)



- May lead to better cache utilization if data is accessed randomly

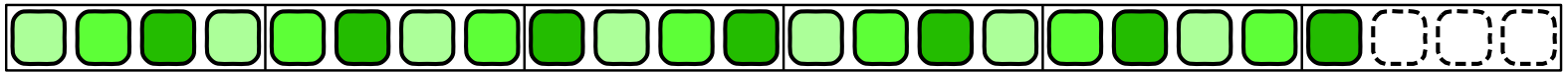
Structures of Arrays (SoA)



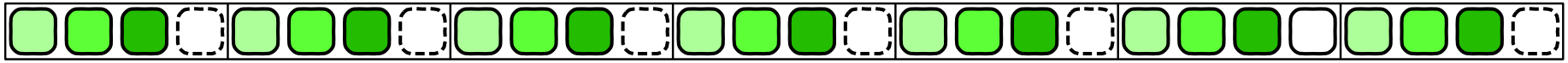
- Typically better for vectorization and avoidance of false sharing

Data Layout Options

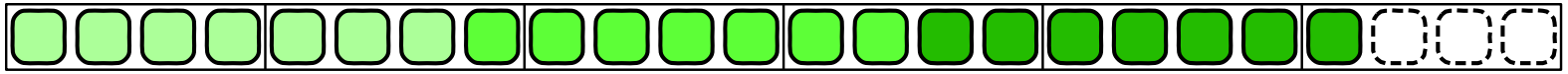
Array of Structures (AoS), padding at end



Array of Structures (AoS), padding after each structure



Structure of Arrays (SoA), padding at end



Structure of Arrays (SoA), padding after each component

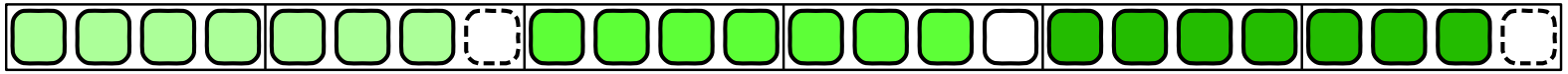


Table of Contents

Example Implementation

AoS Code

```
struct node {
  float x, y, z;
};
struct node NODES[1024];

float dist[1024];
for(i=0;i<1024;i+=16){
   float x[16],y[16],z[16],d[16];
   x[:] = NODES[i:16].x;
   y[:] = NODES[i:16].y;
   z[:] = NODES[i:16].z;
   d[:] = sqrtf(x[:]*x[:] + y[:]*y[:] + z[:]*z[:]);
   dist[i:16] = d[:];
}
```

SoA Code

```
struct node1 {
  float x[1024], y[1024], z[1024];
}
struct node1 NODES1;

float dist[1024];
for(i=0;i<1024;i+=16){
  float x[16],y[16],z[16],d[16];
  x[:] = NODES1.x[i:16];
  y[:] = NODES1.y[i:16];
  z[:] = NODES1.z[i:16];
  d[:] = sqrtf(x[:]*x[:] + y[:]*y[:] + z[:]*z[:]);
  dist[i:16] = d[:];
}
```

AoS Code


```
struct node {
  float x, y, z;
};
struct node NODES[1024];

float dist[1024];
for(i=0;i<1024;i+=16){
   float x[16],y[16],z[16],d[16];
   x[:] = NODES[i:16].x;
   y[:] = NODES[i:16].y;
   z[:] = NODES[i:16].z;
   d[:] = sqrtf(x[:]*x[:] + y[:]*y[:] + z[:]*z[:]);
   dist[i:16] = d[:];
}
```

- Most logical data organization layout
- Extremely difficult to access memory for reads (gathers) and writes (scatters)
- Prevents efficient vectorization

SoA Code


```
struct node1 {
  float x[1024], y[1024], z[1024];
}
struct node1 NODES1;

float dist[1024];
for(i=0;i<1024;i+=16){
  float x[16],y[16],z[16],d[16];
  x[:] = NODES1.x[i:16];
  y[:] = NODES1.y[i:16];
  z[:] = NODES1.z[i:16];
  d[:] = sqrtf(x[:]*x[:] + y[:]*y[:] + z[:]*z[:]);
  dist[i:16] = d[:];
}
```

- Separate arrays for each structure-field keeps memory accesses contiguous when vectorization is performed over structure instances