Logistics

- Homework #1 due today
- Placed data dependence articles by Michael Wolfe on schedule
- MPI in Friday’s lab
  - Begin looking at MPI tutorial
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)
- Think about shared memory data movement
- Distributed memory data movement is an extension
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
Finally, we present some memory layout optimizations—in particular, the conversion of arrays of structures into structures of arrays. This conversion is an important data layout optimization for vectorization. The zip and unzip patterns are special cases of gather that can be used for such data layout reorganization.

### 6.1 GATHER

The gather pattern, introduced in Section 3.5.4, results from the combination of a map with a random read. Essentially, gather does a number of independent random reads in parallel.

#### 6.1.1 General Gather

A defining serial implementation for a general gather is given in Listing 6.1. Given a collection of locations (addresses or array indices) and a source array, gather collects all the data from the source array at the given locations and places them into an output collection. The output data collection has the same number of elements as the number of indices in the index collection, but the elements of the output collection are the same type as the input data collection. If multidimensional index collections are supported, generally the output collection has the same dimensionality as the index collection, as well. A diagram showing an example of a specific gather on a 1D collection (using a 1D index) is given in Figure 6.1.

The general gather pattern is simple but there are many special cases which can be implemented more efficiently, especially on machines with vector instructions. Important special cases include shift and zip, which are diagrammed in Figures 6.2 and 6.3. The inverse of zip, unzip, is also useful.

```cpp
template<
typename Data, typename Idx>
void gather(
    size_t n, // number of elements in data collection
    size_t m, // number of elements in index collection
    Data a[], // input data collection (n elements)
    Data A[], // output data collection (m elements)
    Idx idx[] // input index collection (m elements)
) {
    for (size_t i = 0; i < m; ++i) {
        size_t j = idx[i]; // get ith index
        assert(0 <= j && j < n); // check array bounds
        A[i] = a[j]; // perform random read
    }
}
```

Serial implementation of gather in pseudocode
Gather: Serial Implementation

Finally, we present some memory layout optimizations—in particular, the conversion of arrays of structures into structures of arrays. This conversion is an important data layout optimization for vectorization. The zip and unzip patterns are special cases of gather that can be used for such data layout reorganization.

6.1 GATHER

The gather pattern, introduced in Section 3.5.4, results from the combination of a map with a random read. Essentially, gather does a number of independent random reads in parallel.

6.1.1 General Gather

A defining serial implementation for a general gather is given in Listing 6.1. Given a collection of locations (addresses or array indices) and a source array, gather collects all the data from the source array at the given locations and places them into an output collection. The output data collection has the same number of elements as the number of indices in the index collection, but the elements of the output collection are the same type as the input data collection. If multidimensional index collections are supported, generally the output collection has the same dimensionality as the index collection, as well. A diagram showing an example of a specific gather on a 1D collection (using a 1D index) is given in Figure 6.1.

The general gather pattern is simple but there are many special cases which can be implemented more efficiently, especially on machines with vector instructions. Important special cases include shift and zip, which are diagrammed in Figures 6.2 and 6.3. The inverse of zip, unzip, is also useful.

```
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

```cpp
template<typename Data, typename Idx>
void gather(
    size_t n, // number of elements in data collection
    size_t m, // number of elements in index collection
    Data a[], // input data collection (n elements)
    Data A[], // output data collection (m elements)
    Idx idx[] // input index collection (m elements)
) {
    for (size_t i = 0; i < m; ++i) {
        size_t j = idx[i]; // get ith index
        assert(0 <= j && j < n); // check array bounds
        A[i] = a[j]; // perform random read
    }
}
```

Serial implementation of gather in pseudocode

What can go wrong?
Are there any conflicts that arise?

Parallelize over for loop to perform random read
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
- Index collection
Gather: Defined

Given a collection of read locations
- Address or array indices
- Index collection

and a source array
**Gather: Defined**

Given a **collection of read locations**
- Address or array indices
- Index collection

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
- Index collection

and a source array

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

What value should go into index 1 of input collection?

FIGURE 6.1
Gather pattern. A collection of data is read from an input collection given a collection of indices. This is equivalent to a map combined with a random read in the map’s elemental function.

FIGURE 6.2
Shifts are special cases of gather. There are variants based on how boundary conditions are treated. Boundaries can be duplicated, rotated, reflected, a default value can be used, or most generally some arbitrary function can be used. Unlike a general gather, however, shifts can be efficiently implemented using vector instructions since in the interior, the data access pattern is regular.

FIGURE 6.3
Zip and unzip (special cases of gather). These operations can be used to convert between array of structures (AoS) and structure of arrays (SoA) data layouts.
Gather: Defined

Given a collection of read locations
- Address or array indices
- Index collection

and a source array

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

Read the value at index 1 of source array

FIGURE 6.1 Gather pattern. A collection of data is read from an input collection given a collection of indices. This is equivalent to a map combined with a random read in the map's elemental function.

FIGURE 6.2 Shifts are special cases of gather. There are variants based on how boundary conditions are treated. Boundaries can be duplicated, rotated, reflected, a default value can be used, or most generally some arbitrary function can be used. Unlike a general gather, however, shifts can be efficiently implemented using vector instructions since in the interior, the data access pattern is regular.

FIGURE 6.3 Zip and unzip (special cases of gather). These operations can be used to convert between array of structures (AoS) and structure of arrays (SoA) data layouts.
Gather: Defined

Given a **collection of read locations**
- Address or array indices
- Index collection

and a **source array**

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

Read the value at index 5 of locations array

**FIGURE 6.1**
Gather pattern. A collection of data is read from an input collection given a collection of indices. This is equivalent to a map combined with a random read in the map's elemental function.

**FIGURE 6.2**
Shifts are special cases of gather. There are variants based on how boundary conditions are treated. Boundaries can be duplicated, rotated, reflected, a default value can be used, or most generally some arbitrary function can be used. Unlike a general gather, however, shifts can be efficiently implemented using vector instructions since in the interior, the data access pattern is regular.

**FIGURE 6.3**
Zip and unzip (special cases of gather). These operations can be used to convert between array of structures (AoS) and structure of arrays (SoA) data layouts.
Gather: Defined

Given a collection of read locations
- Address or array indices
- Index collection

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
  - Index collection

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:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>6</td>
<td>9</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Quiz 1**

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

```
3 7 0 1 4 0 0 4 5 3 1 0
1 9 6 9 3
7 3 0 3 1
```
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

- 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: Zip

Function is to interleaves data (like a zipper)

Where is the parallelism?
Zip Example

- 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

- **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: Unzip

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

FIGURE 6.1
Gather pattern. A collection of data is read from an input collection given a collection of indices. This is equivalent to a map combined with a random read in the map's elemental function.

FIGURE 6.2
Shifts are special cases of gather. There are variants based on how boundary conditions are treated. Boundaries can be duplicated, rotated, reflected, a default value can be used, or most generally some arbitrary function can be used. Unlike a general gather, however, shifts can be efficiently implemented using vector instructions since in the interior, the data access pattern is regular.

FIGURE 6.3
Zip and unzip (special cases of gather). These operations can be used to convert between array of structures (AoS) and structure of arrays (SoA) data layouts.

Where is the parallelism?
**Unzip Example**

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

**FIGURE 6.1** Gather pattern. A collection of data is read from an input collection given a collection of indices. This is equivalent to a map combined with a random read in the map's elemental function.

**FIGURE 6.2** Shifts are special cases of gather. There are variants based on how boundary conditions are treated. Boundaries can be duplicated, rotated, reflected, a default value can be used, or most generally some arbitrary function can be used. Unlike a general gather, however, shifts can be efficiently implemented using vector instructions since in the interior, the data access pattern is regular.

**FIGURE 6.3** Zip and unzip (special cases of gather). These operations can be used to convert between array of structures (AoS) and structure of arrays (SoA) data layouts.
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

```cpp
template<typename Data, typename Idx>
void scatter(
    size_t n, // number of elements in output data collection
    size_t m, // number of elements in input data and index collection
    Data a[], // input data collection (m elements)
    Data A[], // output data collection (n elements)
    Idx idx[] // input index collection (m elements)
) {
    for (size_t i = 0; i < m; ++i) {
        size_t j = idx[i]; // get ith index
        assert(0 <= j && j < n); // check output array bounds
        A[j] = a[i]; // perform random write
    }
}
```

Serial implementation of scatter in pseudocode.
Scatter: Serial Implementation

template<typename Data, typename Idx>
void scatter(
    size_t n, // number of elements in output data collection
    size_t m, // number of elements in input data and index collection
    Data a[], // input data collection (m elements)
    Data A[], // output data collection (n elements)
    Idx idx[] // input index collection (m elements)
) {
    for (size_t i = 0; i < m; ++i) {
        size_t j = idx[i]; // get ith index
        assert(0 <= j && j < n); // check output array bounds
        A[j] = a[i]; // perform random write
    }
}

Serial implementation of scatter in pseudocode

Parallelize over for loop to perform random write
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
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:

0 1 2 3 4 5 6 7 8 9 10 11

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

Quiz 2

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

0 1 2 3 4 5 6 7 8 9 10 11

*Solution

3 7 0 1 4 0 0 4 5 3 1 0

2 4 1 5 5 0 4 2 1 2 1 4

0 1 3 0 4

*Solution
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

- 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
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

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

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

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

- 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

- Either “D”… or “E”
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

![Diagram of Priority Scatter Pattern](image)
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

- **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**
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?

**Diagram:**

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
B C F G H
```

**How?**

1. Convert the input array of Booleans into integer 0's and 1's.
2. Perform an exclusive scan of this array with an initial value of 1 and the addition operation.
3. This produces a set of offsets into an output array where each data input to be kept should be written.
4. Use a permutation scatter since locations will not be duplicated.
5. Note that the scatter should be conditional, since only the locations with a true flag should result in a write.

**Example:**

Pack operation is diagrammed in Figure 6.9.
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

- 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
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

Upper half of output collection: values equal to 0
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 does.

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

- 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
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

- 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
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

![Diagram showing the fusion of map and pack patterns with the expand pattern.](image)
Table of 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
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

Various representations of segmented data are possible. The start of each segment can be marked using an array of flags. Alternatively, the start point of each segment can be indicated using an array of integers. The second approach allows zero-length segments; the first does not. The use of Boolean flags to mark the start point of each segment is convenient and is useful for efficient implementation of segmented scans and reductions. The Boolean flags can be stored reasonably efficiently using a packed representation. However, this representation does not allow zero-length segments which are important for some algorithms. Also, it may be space inefficient if segments are very long on average. An alternative approach is to record the start position of every segment. This approach makes it possible to represent empty segments. Note that differences of adjacent values also give the length of each segment. The overall length of the collection can be included as an extra element in the length array to make this regular and avoid a special case.

Many of the patterns we have discussed could output segmented collections. For example, the output of the expand, split, and bin patterns, discussed in Section 6.4, could be represented as a segmented collection. Of course it is always possible to discard the segment start data and so “flatten” a segmented collection. It would also be possible to support nested segmentation, but this can always be represented, for any given maximum nesting depth, using a set of segment-start auxiliary arrays \[BH C+93, Ble90\]. It is possible to map recursive algorithms, such as quicksort, onto such data representations \[Ble96, Ble90\].

Various extensions to multidimensional segmentations are possible. For example, you could segment along each dimension. A kD-tree-like generalization is also possible, where a nested segmentation rotates among dimensions. However, the 1D case is probably the most useful.
Table of 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
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.

FIGURE 6.20
Array of structures (AoS) versus structure of arrays (SoA). SoA form is typically better for vectorization and avoidance of false sharing. However, if the data is accessed randomly, AoS may lead to better cache utilization.

FIGURE 6.21
Data layout options for arrays of structures and structures of arrays. Data can be laid out structure-by-structure, structure-by-structure with padding per structure, or, for structure of array, array-by-array or array-by-array with padding. The structure of array form, either with or without padding, makes vectorization much easier.
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

FIGURE 6.20
Array of structures (AoS) versus structure of arrays (SoA). SoA form is typically better for vectorization and avoidance of false sharing. However, if the data is accessed randomly, AoS may lead to better cache utilization.
Table of 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
Example Implementation

AoS Code

```c
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

```c
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[:];
}
```
Most logical data organization layout

Extremely difficult to access memory for reads (gathers) and writes (scatters)

Prevents efficient vectorization

```c
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

```c
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.