Data Reorganization Pattern

Parallel Computing
CIS 410/510
Department of Computer and Information Science
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

```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 <= i && i < n); // check array bounds
        A[i] = a[j]; // perform random read
    }
}
```

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 pattern](image)
**Gather: Defined**

![Diagram of gather pattern](image)

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**

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**

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
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

Read the value at index 5 of locations 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**

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**

Where is the parallelism?

**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.
**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>

```
1 9 6 9 3
```

```
```
**Quiz 1**

Given the following locations and source array, use a gather to determine what values should go into the output 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
1  9  6  9  3
7  3  0  3  1```

- Values at locations 0, 3, 4, 5, 6, 8 go into outputs 0, 1, 2, 3, 4, 5.
- Values at locations 1, 2, 7, 9 go into outputs 6, 7, 8, 9.
- Values at locations 10, 11 go into output 10.
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

- 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?

- Function is to interleaves data (like a zipper)
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

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

---

- Combined Sequence of Real and Imaginary Parts
- Array of Real Parts
- Array of Imaginary Parts
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

```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
    }
}
```

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.

- **Permutation scatter** makes collisions illegal (see Figure 6.6).
- **Atomic scatter** resolves collisions non-deterministically but atomically (see Figure 6.5).
- **Priority scatter** resolves collisions deterministically using priorities (see Figure 6.8).
- **Merge scatter** resolves collisions by combining values (see Figure 6.7).
**Scatter: Defined**

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

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

![Diagram of Atomic Scatter pattern](image)
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
- No rule determines which of the input items will be retained

![Diagram of Atomic Scatter Pattern]

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

Either “D”…
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
- No rule determines which of the input items will be retained

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

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**

1. **Collision!**

   - **Figure 6.7**
   - Merge scatter pattern. Associative and commutative operators are used to combine values upon collision.

2. **Collision!**

   - **Figure 6.8**
   - Priority scatter pattern. Every element is assigned a priority, which is used to resolve collisions.

---

**6.2.1 Atomic Scatter**

- The atomic scatter pattern is non-deterministic. Upon collision, in an atomic scatter one and only one of the values written to a location will be written in its entirety. All other values written to the same location will be discarded. See **Figure 6.5** for an example. Note that we do not provide a rule saying which of the input items will be retained. Typically, it is the last one written but in parallel implementations of atomic scatter the timing of writes is non-deterministic.

   - This pattern resolves collisions atomically but non-deterministically. Use of this pattern may result in non-deterministic programs. However, it is still useful and deterministic in the special case that all input data elements written to the same location have the same value. A common example of this is the writing of **true** Boolean flags into an output array that has initially been cleared to **false**. In this case, there is an implicit OR merge between the written values, since only one of the writes needs to update the output location to turn it into a **true**, and the result is the same whichever write succeeds.

   - Examples of the use of atomic scatter include marking pairs in collision detection, and computing set intersection or union as are used in text databases. Note that these are both examples where Boolean values may be used.

---

**6.2.2 Permutation Scatter**

- The permutation scatter pattern simply states that collisions are illegal; in other words, legal inputs should not have duplicates. See **Figure 6.6** for an example of a legal permutation scatter. Permutation scatters can always be turned into gathers, so if the addresses are known in advance, this optimization...
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 known “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 of Pack Pattern]

---

**Figure 6.9** Pack pattern.
Pack Algorithm

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

```
0 1 1 0 0 1 1 1
A B C D E F G H
```

```
B C F G H
```
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 does

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

Introduction to Parallel Computing, University of Oregon, IPCC
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
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

---

For example, suppose you wanted to create a parallel implementation of L-system substitution. In L-system substitution, the input and output are strings of characters. Every element of the input string...
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 regions. Avoid write conflicts and race conditions.

- non-overlapping
- equal-sized regions
**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

---

**FIGURE 6.18**

Segmentation. If data is divided into non-uniform non-overlapping regions, it can be referred to as **segmentation** (a generalization of partitioning).

**FIGURE 6.19**

Segmentation representations. 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.

- 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 [BHC+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

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.
Structures of Arrays (SoA)

- Typically better for vectorization and avoidance of false sharing.
**Data Layout Options**

Array of Structures (AoS), padding at end

![Diagram of AoS with padding at end]

Array of Structures (AoS), padding after each structure

![Diagram of AoS with padding after each structure]

Structure of Arrays (SoA), padding at end

![Diagram of SoA with padding at end]

Structure of Arrays (SoA), padding after each component

![Diagram of SoA with padding after each component]
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**

```
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

```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[:];
}
```

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