Contents

10.5.3 Zip

The **Zip operation** is a special case of the gather pattern where **two (or more) arrays are combined by interleaving their elements**. It functions like a zipper, **taking one element from each array in sequence to form a new combined array**. It is important to note that it **works with different types**, so we can zip elements of different types, like integers and floats, or even complex objects.

How does it work?

The operation **takes an element from the first array**, then **one from the second** array, another from the third, and so on, and **repeats the process**. The **output is the combined sequence**.

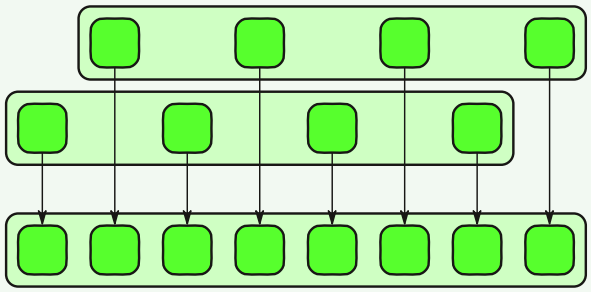
Example 46: zip operation

Consider two arrays:

1. Real Parts: contains real numbers.
2. Imaginary Parts: contains imaginary numbers.

The output is a combined sequence like:

```
[Real0, Imag0, Real1, Imag1, Real2, Imag2, ...]
```



Parallelism

Each pair of elements (one from each array) can be **combined independently**. This independence allows **parallel execution since there's no dependency between the operations for different pairs**.

10.5.4 Unzip

The **Unzip operation** is essentially the **reverse of the zip operation**. While zip combines multiple arrays into one by interleaving their elements, **unzip separates a single interleaved array back into its original components**.

How does it work?

The operation **extracts sub-arrays at regular offsets** (strides) **to separate the elements into their original groups**. The input is a combined sequence (e.g., alternating real and imaginary parts).

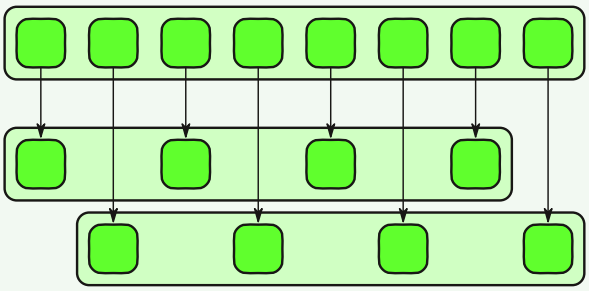
Example 47: unzip operation

Consider the input array:

```
[Real0, Imag0, Real1, Imag1, Real2, Imag2, ...]
```

The output are:

- Array of Real Parts: `[Real0, Real1, Real2]`
- Array of Imaginary Parts: `[Imag0, Imag1, Imag2]`



Parallelism

Each element extraction is independent. This allows for **parallel data access** because there's no dependency between different extractions.

However, the unzip operation is an **efficient data extraction because it takes advantage of stride-based memory access**, which can be optimized for performance.