Article

Controlling vDSP operations with stride

Operate selectively on the elements of a vector at regular intervals.

Overview

Many functions in vDSP provide support for specifying a *stride* (that is, the distance between the elements that functions read from or write to) for a particular vector. For example, if you want to access consecutive elements, use a stride of 1 (referred to as a *unit stride*). If you want to access every third element, for example to work with the red channel in interleaved RGB data, use a stride of 3. If you want to access every second element, for example to work with a single audio channel in interleaved stereo audio data, use a stride of 2.

Typically, you use a unit stride. Use other strides to, for example, operate along a column of a matrix, where the stride is the number of elements per row.

Use a unit stride for the best performance and energy efficiency

For most of the functions in vDSP, you obtain the best performance when the stride is 1. Any other stride value generally prevents the use of vectorized code, and reduces both performance and energy efficiency.

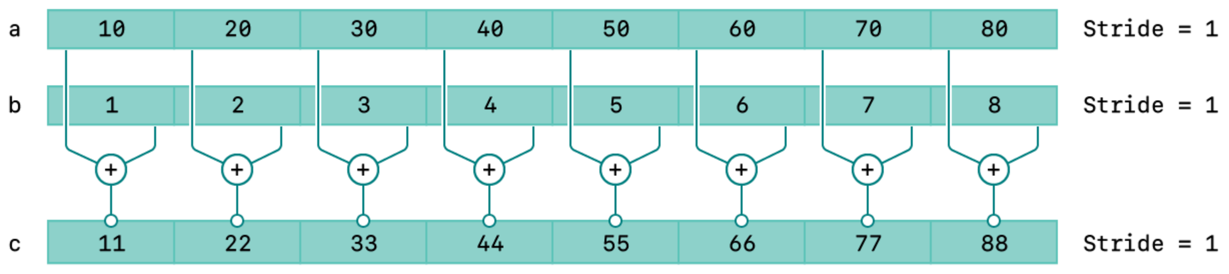
The major exception to this limitation is in functions that support the use of interleaved complex data, such as `vDSP_ctoz` and `vDSP_ztoc`. In these cases, use a stride of 2.

Set the stride for each vector independently

The code below calls the `vDSP_vadd` function to add each element in array a to the corresponding element in array b, and write the result to array c. Note that the stride used for each array is 1.

```
let strideA = vDSP_Stride(1)
let strideB = vDSP_Stride(1)
let strideC = vDSP_Stride(1)

let a: [Float] = [10, 20, 30, 40, 50, 60, 70, 80]
let b: [Float] = [ 1,  2,  3,  4,  5,  6,  7,  8]

let n = vDSP_Length(a.count)

var c = [Float](repeating: .nan,
                count: a.count)

vDSP_vadd(a, strideA,
          b, strideB,
          &c, strideC,
          n)
```

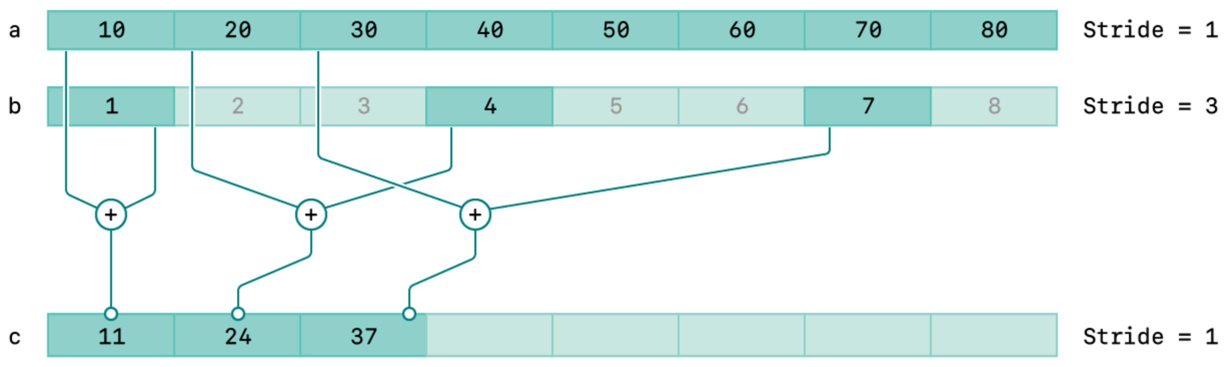
In this example, the result is `[11.0, 22.0, 33.0, 44.0, 55.0, 66.0, 77.0, 88.0]`.



Use a nonunit stride on inputs

A nonunit stride allows you to, for example, access a particular color channel in interleaved RGB data. If you change `strideB` to 3, the operation adds the first, fourth, and seventh items in array b to the first, second, and third items in array a.

```
let strideA = vDSP_Stride(1)
let strideB = vDSP_Stride(3)
let strideC = vDSP_Stride(1)
...
let n = vDSP_Length(3)
```

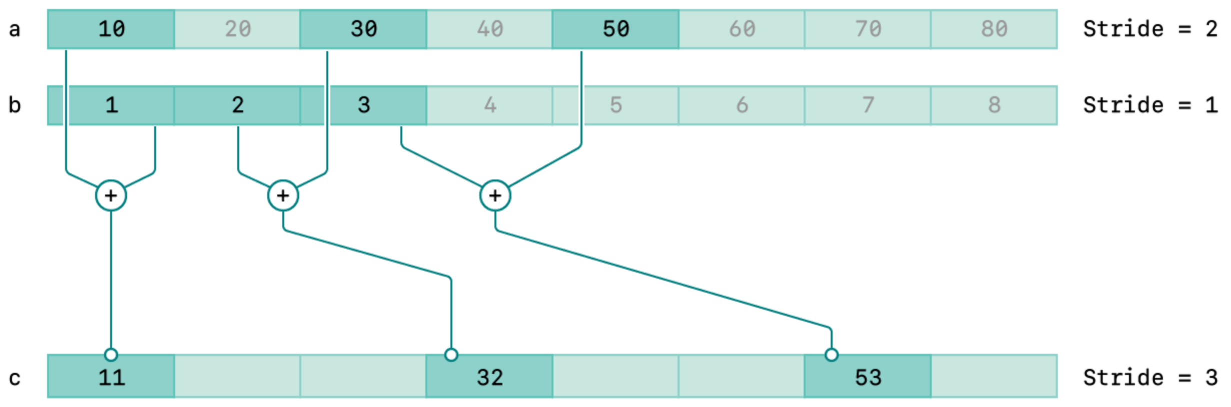


Note that vDSP operations always read n elements. Therefore, your collections require at least `((n - 1) * stride) + 1` elements.

Use a nonunit stride on output

If you change array c's stride to 3, the calculation writes the result to its first, fourth, and seventh items. Using the example of interleaved RGB data discussed in Controlling vDSP operations with stride, this approach would write the result of an operation to the red channel. The example below defines the stride for the input array, a, as 2, so the operation uses the first, third, and fifth elements:

```
let strideA = vDSP_Stride(2)
let strideB = vDSP_Stride(1)
let strideC = vDSP_Stride(3)
...
let n = vDSP_Length(3)
```



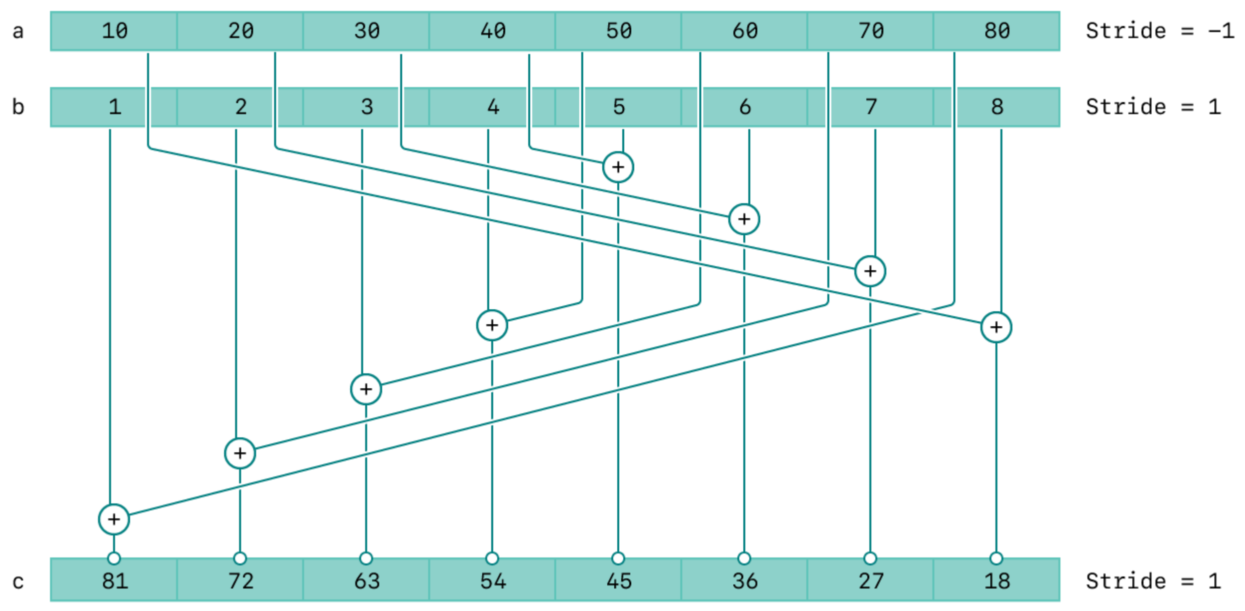
Use a negative stride

Use a negative stride to access a vector in reverse order, for example, define the stride as −1 when convolving with a filter using `vDSP_conv`).

To use a negative stride, pass the vDSP_vadd function a pointer to the address of the last element in the array. The example below shows the Swift code required to reverse the elements in array a:

```
let strideA = vDSP_Stride(-1)
let strideB = vDSP_Stride(1)
let strideC = vDSP_Stride(1)
...
a.withUnsafeBufferPointer { buffer in
    vDSP_vadd(buffer.baseAddress!.advanced(by: buffer.count - 1), strideA,
              b, strideB,
              &c, strideC,
              n)
}
```

The result of adding a and b with a stride of −1 for a is `[81.0, 72.0, 63.0, 54.0, 45.0, 36.0, 27.0, 18.0]`:



Controlling vDSP operations with strides

With interleaved complex data, vDSP stores alternating real and imaginary components consecutively. Use a stride of 2 for interleaved complex data, counting the individual component elements rather than counting complex numbers.

For example, use the code below to copy the contents of a `DSPSplitComplex` structure to an array of `DSPComplex` values:

```
var real: [Float] = [10, 20, 30, 40, 50, 60, 70, 80]
var imag: [Float] = [ 1,  2,  3,  4,  5,  6,  7,  8]

let n = real.count

var complex = [DSPComplex](repeating: DSPComplex(),
                           count: n)

real.withUnsafeMutableBufferPointer { realPtr in
    imag.withUnsafeMutableBufferPointer { imagPtr in

        var splitComplex = DSPSplitComplex(realp: realPtr.baseAddress!,
                                           imagp: imagPtr.baseAddress!)

        let strideSplitComplex = vDSP_Stride(1)
        let strideComplex = vDSP_Stride(2)

        vDSP_ztoc(&splitComplex, strideSplitComplex,
                  &complex, strideComplex,
                  vDSP_Length(n))
    }
}
```

On return, `complex` contains the pairs `[10.0 1.0], [20.0 2.0], [30.0 3.0] ... [80.0 8.0]`.

Conversely, use the example below to copy the values of an array of `DSPComplex` values to a `DSPSplitComplex` structure:

```
vDSP_ctoz(&complex, strideComplex,
          &splitComplex, strideSplitComplex,
          vDSP_Length(n))
```

See Also

Signal Processing Essentials

Using linear interpolation to construct new data points

Fill the gaps in arrays of numerical data using linear interpolation.

Using vDSP for vector-based arithmetic

Increase the performance of common mathematical tasks with vDSP vector-vector and vector-scalar operations.

Resampling a signal with decimation

Reduce the sample rate of a signal by specifying a decimation factor and applying a custom antialiasing filter.

vDSP

Perform basic arithmetic operations and common digital signal processing (DSP) routines on large vectors.