Introducing Accelerate for Swift

Simon Gladman, Vector and Numerics
What is Accelerate?
Accelerate

Functionality
Accelerate

Functionality

Performance
Accelerate

Functionality

Performance

Energy Efficiency
Accelerate

High-performance, energy-efficient vectorized computation
Accelerate

High-performance, energy-efficient vectorized computation

macOS
Accelerate

High-performance, energy-efficient vectorized computation

macOS  iOS
Accelerate

High-performance, energy-efficient vectorized computation

macOS  iOS  watchOS
Accelerate

High-performance, energy-efficient vectorized computation

macOS  iOS  watchOS  tvOS
Accelerate

High-performance, energy-efficient vectorized computation

Classic C interfaces are awkward when working with Swift
Accelerate

High-performance, energy-efficient vectorized computation

Classic C interfaces are awkward when working with Swift

New Swift API is clearer and more concise
vDSP: Digital signal processing functions
Accelerate

**vDSP:** Digital signal processing functions

**vForce:** Arithmetic and transcendental functions
Accelerate

vDSP: Digital signal processing functions
vForce: Arithmetic and transcendental functions
Quadrature: Numerical integration functions
Accelerate

**vDSP**: Digital signal processing functions

**vForce**: Arithmetic and transcendental functions

**Quadrature**: Numerical integration functions

**vImage**: Image-processing functions
Accelerate

Scalar calculations execute serially

```swift
let a: [Float] = [10, 20, 30, 40]
let b: [Float] = [1, 2, 3, 4]
var c: [Float] = [0, 0, 0, 0]

for i in 0..<c.count {
    c[i] = a[i] * b[i]
}

// c = [10.0, 40.0, 90.0, 160.0]
```
Accelerate

Scalar calculations execute serially

```swift
let a: [Float] = [10, 20, 30, 40]
let b: [Float] = [1, 2, 3, 4]
var c: [Float] = [0, 0, 0, 0]

for i in 0..<c.count {
    c[i] = a[i] * b[i]
}

// c = [10.0, 40.0, 90.0, 160.0]
```
Scalar calculations execute serially

```swift
let a: [Float] = [10, 20, 30, 40]
let b: [Float] = [1, 2, 3, 4]
var c: [Float] = [0, 0, 0, 0]

for i in 0..<c.count {
    c[i] = a[i] * b[i]
}

// c = [10.0, 40.0, 90.0, 160.0]
```
Accelerate

Scalar calculations execute serially

```swift
let a: [Float] = [10, 20, 30, 40]
let b: [Float] = [1, 2, 3, 4]
var c: [Float] = [0, 0, 0, 0]

for i in 0..<c.count {
    c[i] = a[i] * b[i]
}

// c = [10.0, 40.0, 90.0, 160.0]
```
Accelerate

Vectorized instructions return multiple results

```swift
let a: [Float] = [10, 20, 30, 40]
let b: [Float] = [1, 2, 3, 4]
var c: [Float] = [0, 0, 0, 0]
vDSP.multiply(a, b,
    result: &c)

// c = [10.0, 40.0, 90.0, 160.0]
```
vDSP
vDSP

Fourier transforms
vDSP

Fourier transforms

Biquadratic filtering
vDSP

Fourier transforms

Biquadratic filtering

Convolution and correlation
vDSP

Fourier transforms
Biquadratic filtering
Convolution and correlation
Vector and matrix arithmetic
vDSP

Fourier transforms
Biquadratic filtering
Convolution and correlation
Vector and matrix arithmetic
Type conversion
vDSP for Vector Arithmetic

For all elements in arrays $a$, $b$, $c$, and $d$, compute the following:

\[ \text{result}[i] = (a[i] + b[i]) \times (c[i] - d[i]) \]
vDSP for Vector Arithmetic
Vector arithmetic using scalar code

```swift
var result = [Float](repeating: 0, count: n)

for i in 0..<n {
    result[i] = (a[i] + b[i]) * (c[i] - d[i])
}
```
vDSP for Vector Arithmetic

Vector arithmetic using existing vDSP API

```swift
var result = [Float](repeating: 0,
    count: n)

vDSP_vasbm(a, 1,
    b, 1,
    c, 1,
    d, 1,
    &result, 1,
    vDSP_Length(result.count))
```
vDSP for Vector Arithmetic
Vector arithmetic using new vDSP API

```swift
var result = [Float](repeating: 0,
    count: n)

vDSP.multiply(addition: (a, b),
    subtraction: (c, d),
    result: &result)
```
vDSP for Vector Arithmetic
Vector arithmetic using new self-allocating API

```swift
let result = vDSP.multiply(addition: (a, b),
                           subtraction: (c, d))
```
vDSP for Type Conversion

For all double-precision elements in an array, return UInt16
vDSP for Type Conversion
Using scalar code

```swift
let result = source.map {
    return UInt16($0.rounded(.towardZero))
}
```
vDSP for Type Conversion
Using existing API

```swift
let result = Array<UInt16>(_unsafeUninitializedCapacity: source.count) {
    buffer, initializedCount in

    vDSP_vfixu16(source, 1,
                 buffer.baseAddress!, 1,
                 vDSP_Length(source.count))

    initializedCount = source.count
}
```
vDSP for Type Conversion
Using new API

```swift
let result = vDSP.floatingPointToInteger(source,
    integerType: UInt16.self,
    rounding: .towardZero)
```
vDSP for Discrete Fourier Transform

Decompose a signal into its component frequencies

Recreate a signal from its component frequencies
vDSP for Discrete Fourier Transform
Using existing API

```swift
let setup = vDSP_DFT_zop_CreateSetup(
    nil,
    vDSP_Length(n),
    .FORWARD)!

var outputReal = [Float](repeating: 0, count: n)
var outputImag = [Float](repeating: 0, count: n)

vDSP_DFT_Execute(setup,
    inputReal, inputImag,
    &outputReal, &outputImag)

vDSP_DFT_DestroySetup(setup)
```
vDSP for Discrete Fourier Transform
Using new API

```swift
let fwdDFT = vDSP.DFT(
    count: n,
    direction: .forward,
    transformType: .complexComplex,
    ofType: Float.self)!
```
vDSP for Discrete Fourier Transform

Using new API

```swift
let fwdDFT = vDSP.DFT(
    count: n,
    direction: .forward,
    transformType: .complexComplex,
    ofType: Float.self)!

var outputReal = [Float](repeating: 0, count: n)
var outputImag = [Float](repeating: 0, count: n)

fwdDFT.transform(inputReal: inputReal,
                 inputImaginary: inputImag,
                 outputReal: &outputReal,
                 outputImaginary: &outputImag)
```
vDSP for Discrete Fourier Transform
Using new API

```swift
let fwdDFT = vDSP.DFT(
    count: n,
    direction: .forward,
    transformType: .complexComplex,
    ofType: Float.self)!

var outputReal = [Float](repeating: 0, count: n)
var outputImag = [Float](repeating: 0, count: n)

fwdDFT.transform(inputReal: inputReal,
                  inputImaginary: inputImag,
                  outputReal: &outputReal,
                  outputImaginary: &outputImag)
```
vDSP for Discrete Fourier Transform
Using self-allocating API

```swift
let fwdDFT = vDSP.DFT(
    count: n,
    direction: .forward,
    transformType: .complexComplex,
    ofType: Float.self)!

let returnedResult = fwdDFT.transform(
    inputReal: inputReal,
    inputImaginary: inputImag)
```
vDSP for Discrete Fourier Transform
Using self-allocating API

```swift
let fwdDFT = vDSP.DFT(
    count: n,
    direction: .forward,
    transformType: .complexComplex,
    ofType: Float.self)!

let returnedResult = fwdDFT.transform(
    inputReal: inputReal,
    inputImaginary: inputImag)
```
vDSP for Biquadratic Filtering

\[ H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \]
vDSP for Biquadratic Filtering

Given these values:

```swift
let sections = vDSP_Length(1)

let b0 = 0.0001
let b1 = 0.001
let b2 = 0.0005
let a1 = -1.9795
let a2 = 0.98

let channelCount = vDSP_Length(2)
```
vDSP for Biquadratic Filtering

Setting up biquad using existing API

```swift
var output = [Float](repeating: -1, count: n)

let setup = vDSP_biquadm_CreateSetup([b0, b1, b2, a1, a2, b0, b1, b2, a1, a2], vDSP_Length(sections), vDSP_Length(channelCount))!
```
vDSP for Biquadratic Filtering
Applying a biquad filter using existing API

```swift
signal.withUnsafeBufferPointer { inputBuffer in
    output.withUnsafeMutableBufferPointer { outputBuffer in
        let length = vDSP_Length(n) / channelCount
        var inputs: [UnsafePointer<Float>] = (0 ..< channelCount).map { i in
            return inputBuffer.baseAddress!.advanced(by: Int(i * length))
        }
        var outputs: [UnsafeMutablePointer<Float>] = (0 ..< channelCount).map { i in
            return outputBuffer.baseAddress!.advanced(by: Int(i * length))
        }
        vDSP_biquadm(setup, &inputs, 1, &outputs, 1,
            vDSP_Length(n) / channelCount)
    }
}
```
vDSP for Biquadratic Filtering
Setting up and applying a biquad with new API

```swift
var biquad = vDSP.Biquad(coefficients: [b0, b1, b2, a1, a2, b0, b1, b2, a1, a2],
channelCount: channelCount,
sectionCount: sections,
ofType: Float.self)!
```
vDSP for Biquadratic Filtering
Setting up and applying a biquad with new API

```swift
var biquad = vDSP.Biquad(coefficients: [b0, b1, b2, a1, a2, b0, b1, b2, a1, a2],
                          channelCount: channelCount,
                          sectionCount: sections,
                          ofType: Float.self)!

let output = biquad.apply(input: signal)
```
vDSP for Biquadratic Filtering
Setting up and applying a biquad with new API

```swift
var biquad = vDSP.Biquad(coefficients: [b0, b1, b2, a1, a2, b0, b1, b2, a1, a2],
    channelCount: channelCount,
    sectionCount: sections,
    ofType: Float.self)!

let output = biquad.apply(input: signal)
```
vForce

Arithmetic functions: floor, ceil, abs, remainder, ...

Exponential and logarithmic functions: exp, log, ...

Trigonometric functions: sin, cos, tan, ...

Hyperbolic functions: sinh, asinh, ...
vForce
Calculating square roots using scalar code

```swift
let a: [Float] = ...

let result = a.map {
    sqrt($0)
}
```
vForce
Calculating square roots using existing vDSP API

```swift
var result = [Float](repeating: 0,
    count: count)

var n = Int32(result.count)

vvsqrtf(&result,
    a,
    &n)
```
vForce
Calculating square roots using new vDSP API

```swift
var result = [Float](repeating: 0,
    count: count)

vForce.sqrt(a,
    result: &result)
```
vForce
Calculating square roots using new self-allocating API

let result = vForce.sqrt(a)
Quadrature
Integrate a function over an interval

\[ y = \sqrt{r^2 - x^2} \]
Quadrature
Defining the integrate function using existing API

```swift
var integrateFunction: quadrature_integrate_function = {
    return quadrature_integrate_function(
        fun: { (arg: UnsafeMutableRawPointer?, n: Int,
            x: UnsafePointer<Double>, y: UnsafeMutablePointer<Double>) in
            guard let radius = arg?.load(as: Double.self) else { return }

            (0 ..< n).forEach { i in
                y[i] = sqrt(radius * radius - x[i] * x[i])
            }
        },
        fun_arg: &radius)
}()
```
Quadrature
Defining the integrate options using existing API

```python
var options = quadrature_integrate_options(integrator: QUADRATURE_INTEGRATE_QNG,
                                           abs_tolerance: 1.0e-8,
                                           rel_tolerance: 1.0e-2,
                                           qag_points_per_interval: 0,
                                           max_intervals: 0)
```
Quadrature
Performing the integration using existing API

```
var status = QUADRATURE_SUCCESS
var estimatedAbsoluteError: Double = 0

let result = quadrature_integrate(&integrateFunction,
  -radius,
  radius,
  &options,
  &status,
  &estimatedAbsoluteError,
  0,
  nil)
```
Quadrature
Using new API

```swift
let quadrature = Quadrature(integrator: .nonAdaptive,
absOLUTEtOLerance: 1.0e-8,
relativeTOLerance: 1.0e-2)

let result = quadrature.integrate(over: -radius ... radius) { x in
return sqrt(radius * radius - x * x)
}
```
let quadrature = Quadrature(integrator:
    .adaptive(pointsPerInterval: .fifteen,
                maxIntervals: 7),
    absoluteTolerance: 1.0e-8,
    relativeTolerance: 1.0e-2)

let result = quadrature.integrate(over: -radius ... radius) { x in
    return sqrt(radius * radius - x * x)
}
vlimage
Core Graphics interoperability
Core Graphics interoperability

Core Video interoperability
vlImage

Core Graphics interoperability
Core Video interoperability
Alpha blending
vlImage

Core Graphics interoperability
Core Video interoperability
Alpha blending
Format conversions
Core Graphics interoperability
Core Video interoperability
Alpha blending
Format conversions
Histogram operations
vlImage

Core Graphics interoperability
Core Video interoperability
Alpha blending
Format conversions
Histogram operations
Convolution
Core Graphics interoperability
Core Video interoperability
Alpha blending
Format conversions
Histogram operations
Convolution
Geometry
vlImage

Core Graphics interoperability
Core Video interoperability
Alpha blending
Format conversions
Histogram operations
Convolution
Geometry
Morphology
Flags are now Swift `OptionSet`

Throws proper Swift errors

Enumerations for pixel formats and buffer types

Hides requirements for unmanaged types and mutable buffers

Moves free functions to properties on buffers and formats
vlImage Working with Buffers
Create a buffer from a Core Graphics image

Create format description

Instantiate the buffer

Initialize the buffer from the CGImage
Creating a buffer from an image using existing API

```swift
var format = vImage_CGImageFormat(
    bitsPerComponent: 8,
    bitsPerPixel: 32,
    colorSpace: nil,
    bitmapInfo: CGBitmapInfo(rawValue: CGImageAlphaInfo.first.rawValue),
    version: 0,
    decode: nil,
    renderingIntent: .defaultIntent)
```
vImage Working with Buffers
Creating a buffer from an image using existing API

```swift
var sourceBuffer = vImage_Buffer()

var error = kvImageNoError
error = vImageBuffer_InitWithCGImage(&sourceBuffer,
    &format,
    nil,
    image,
    vImage_Flags(kvImageNoFlags))

guard error == kvImageNoError else {
    fatalError("Error in vImageBuffer_InitWithCGImage: \(error)"
}
```
Creating a buffer from an image using new API

```swift
let sourceBuffer = try? vImage_Buffer(cgImage: image)
```
vImage Working with Buffers
Creating a buffer from an image using new API

```swift
let format = vImage_CGImageFormat(cgImage: image)!

let sourceBuffer = try? vImage_Buffer(cgImage: image,
                                    format: format)
```
vImage Working with Buffers
Creating an image from a buffer using existing API

```swift
let cgImage = vImageCreateCGImageFromBuffer(
    &sourceBuffer,
    &format,
    nil,
    nil,
    vImage_Flags(kvImageNoFlags),
    &error)
```
vcImage Working with Buffers
Creating an image from a buffer using new API

```swift
let cgImage = try? sourceBuffer.createCGImage(format: format)
```
vlImage Converting Any-to-Any

Core Graphics to Core Graphics
Core Graphics to Core Video
Core Video to Core Graphics
vImage Converting Any-to-Any
Creating a converter using existing API

```swift
let cmykToRgbUnmanagedConverter = vImageConverter_CreateWithCGImageFormat(
    &cmykSourceImageFormat,
    &rgbDestinationImageFormat,
    nil,
    vImage_Flags(kvImageNoFlags),
    nil)
```
Performing the conversion using existing API

```swift
guard let cmykToRgbConverter = cmykToRgbUnmanagedConverter?.takeRetainedValue() else {
    return
}

vImageConvert_AnyToAny(cmykToRgbConverter,
                         &cmykSourceBuffer,
                         &rgbDestinationBuffer,
                         nil,
                         vImage_Flags(kvImageNoFlags))
```
vImage Converting Any-to-Any
Using new API

```swift
let converter = try? vImageConverter.make(sourceFormat: cmykSourceImageFormat,
                                          destinationFormat: rgbDestinationImageFormat)
```
let converter = try? vImageConverter.make(sourceFormat: cmykSourceImageFormat,
    destinationFormat: rgbDestinationImageFormat)

try? converter?.convert(source: cmykSourceBuffer,
    destination: &rgbDestinationBuffer)
vlImage Working with CV Image Formats

Create format description from `CVPixelBuffer`

Calculate channel count
vImage Working with CV Image Formats
Using existing API

```swift
let cvImageFormat = 
vImageCVImageFormat_CreateWithCVPixelBuffer(pixelBuffer).takeRetainedValue()

let cvImageFormatPointer = UnsafeMutableRawPointer.allocate(
    byteCount: MemoryLayout<vImageCVImageFormat>.size,
    alignment: MemoryLayout<vImageCVImageFormat>.alignment)

cvImageFormatPointer.storeBytes(of: cvImageFormat,
    as: vImageCVImageFormat.self)

let cvConstImageFormat = cvImageFormatPointer.load(as: vImageConstCVImageFormat.self)

let channelCount = vImageCVImageFormat_GetChannelCount(cvConstImageFormat)
```
vImage Working with CV Image Formats
Using new API

```swift
let cvImageFormat = vImageCVImageFormat.make(buffer: pixelBuffer)
```
vImage Working with CV Image Formats

Using new API

```swift
let cvImageFormat = vImageCVImageFormat.make(buffer: pixelBuffer)

let channelCount = cvImageFormat?.channelCount
```
Linpack Benchmark
LINPACK Benchmark

How fast can you solve a system of equations?

Actually three separate benchmarks

• 100-by-100 system
• 1000-by-1000 system
• “No holds barred”
LINPACK Benchmark
Performance in GFLOPS (Bigger is better)

iPhone XS

Using Accelerate
LINPACK Benchmark
Performance in GFLOPS (Bigger is better)

iPhone XS
Using Accelerate

1.8
LINPACK Benchmark
Performance in GFLOPS (Bigger is better)

<table>
<thead>
<tr>
<th>Device</th>
<th>Performance (GFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPhone XS</td>
<td>1.8</td>
</tr>
<tr>
<td>iPhone XS (Using Accelerate)</td>
<td>10.0</td>
</tr>
</tbody>
</table>
LINPACK Benchmark
Performance in GFLOPS (Bigger is better)

iPhone XS

Using Accelerate

1.8

44.75
BLAS

SGEMM (Single Precision General Matrix Multiply)

General matrix multiply

• Is the workhorse for other matrix-matrix operations
• Matrix-matrix operations comprise most of the work done in matrix solvers
SGEMM: iPhone XS
Performance in GFLOPS (Bigger is better)

Eigen
Accelerate
SGEMM: iPhone XS
Performance in GFLOPS (Bigger is better)

- Eigen: 51.01
- Accelerate: 0.01
SGEMM: iPhone XS
Performance in GFLOPS (Bigger is better)

- Eigen: 51.01
- Accelerate: 122.82
Summary

Accelerate provides a huge range of fast and energy efficient functions

New Swift API simplifies implementing Accelerate’s libraries
More Information

developer.apple.com/wwdc19/718