#WWDC19

Introducing Accelerate for Swift

Simon Gladman, Vector and Numerics

What is Accelerate?

Accelerate

Functionality

Accelerate

Functionality

Performance

Accelerate

Functionality

Performance

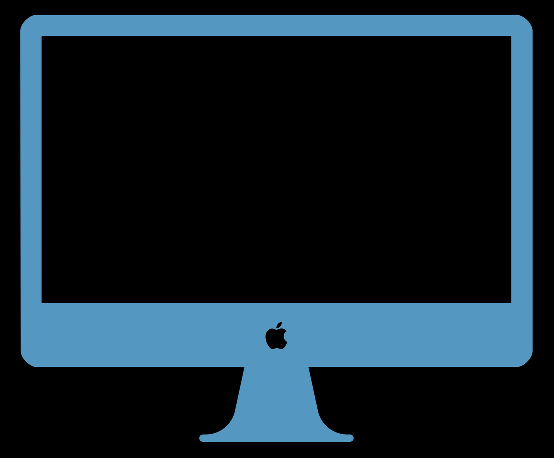
Energy Efficiency

Accelerate

High-performance, energy-efficient vectorized computation

Accelerate

High-performance, energy-efficient vectorized computation



macOS

Accelerate

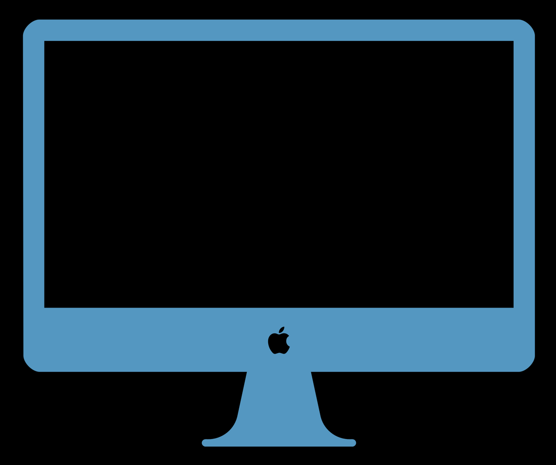
High-performance, energy-efficient vectorized computation

macOS

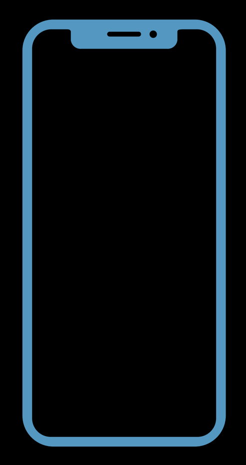
iOS

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

Accelerate

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

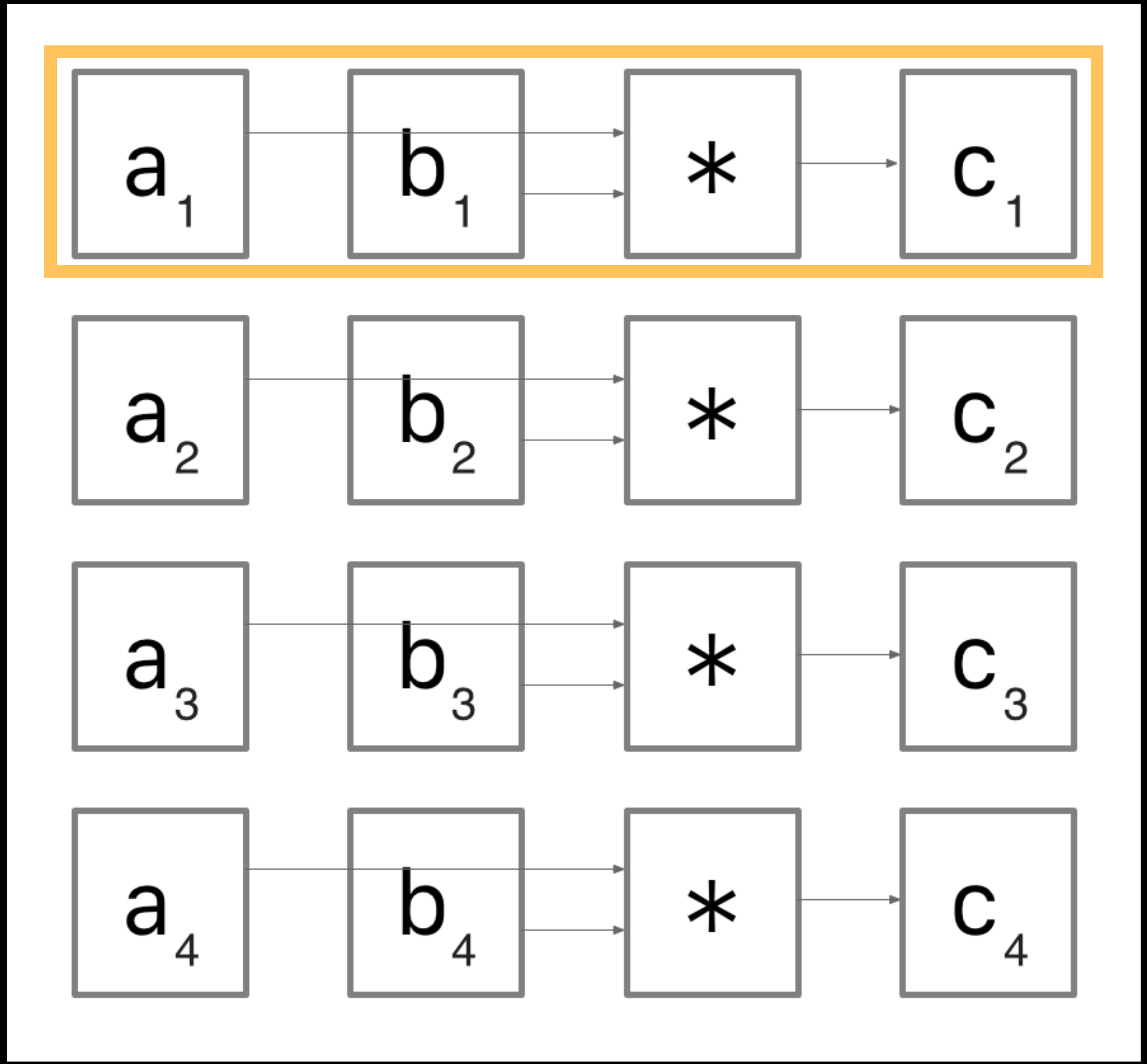
Accelerate

Scalar calculations execute serially

```
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]
```



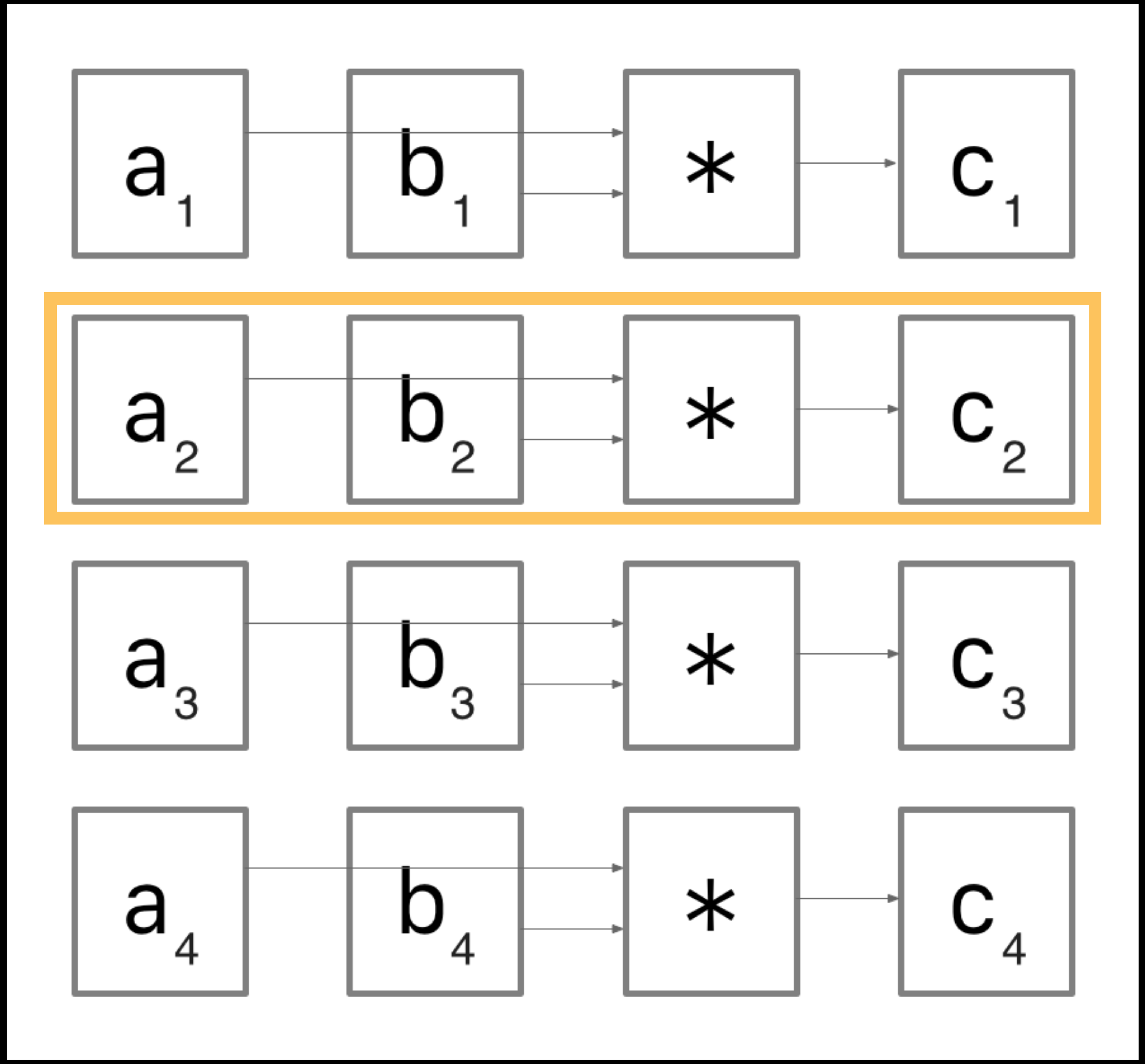
Accelerate

Scalar calculations execute serially

```
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]
```



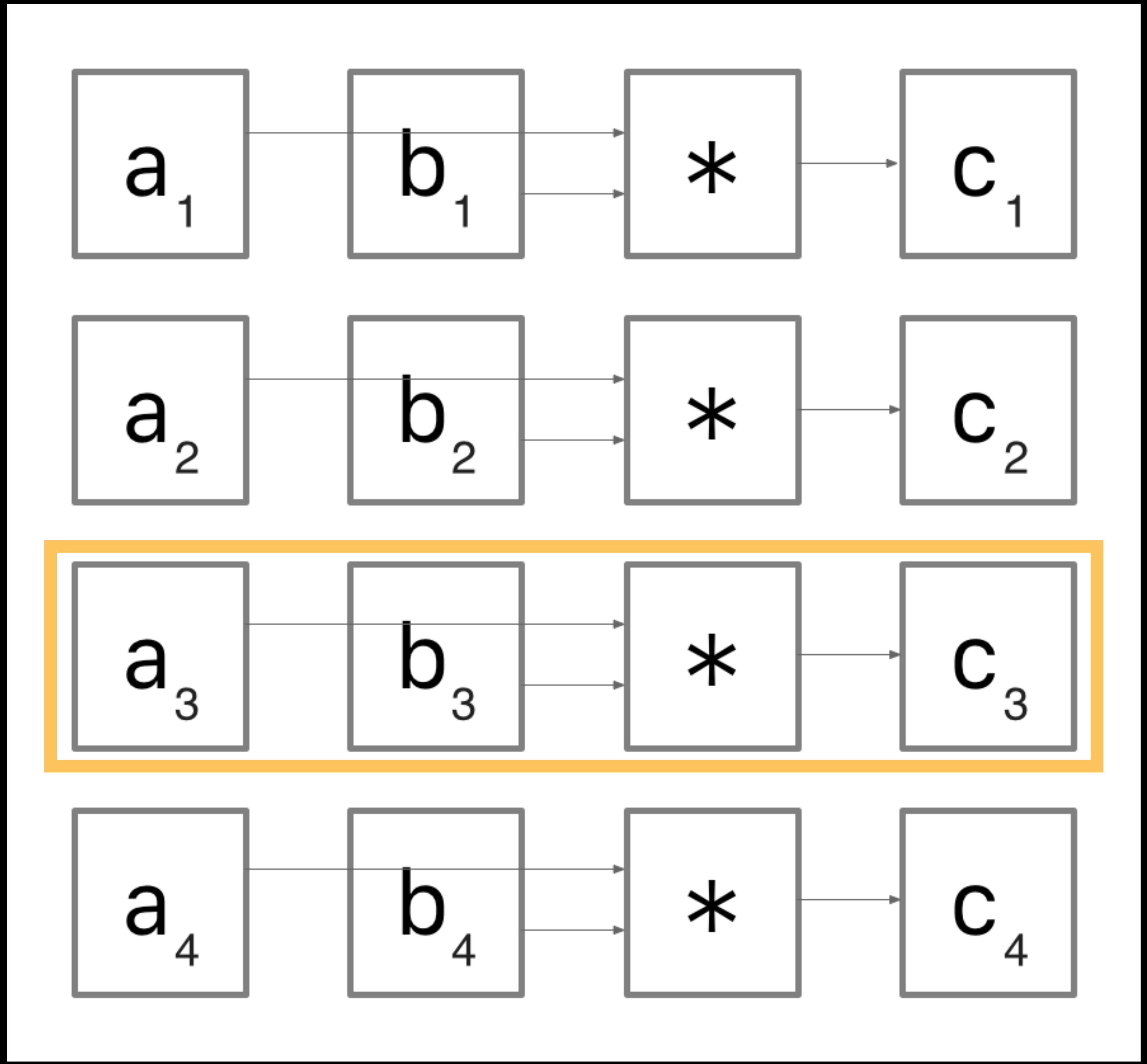
Accelerate

Scalar calculations execute serially

```
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]
```



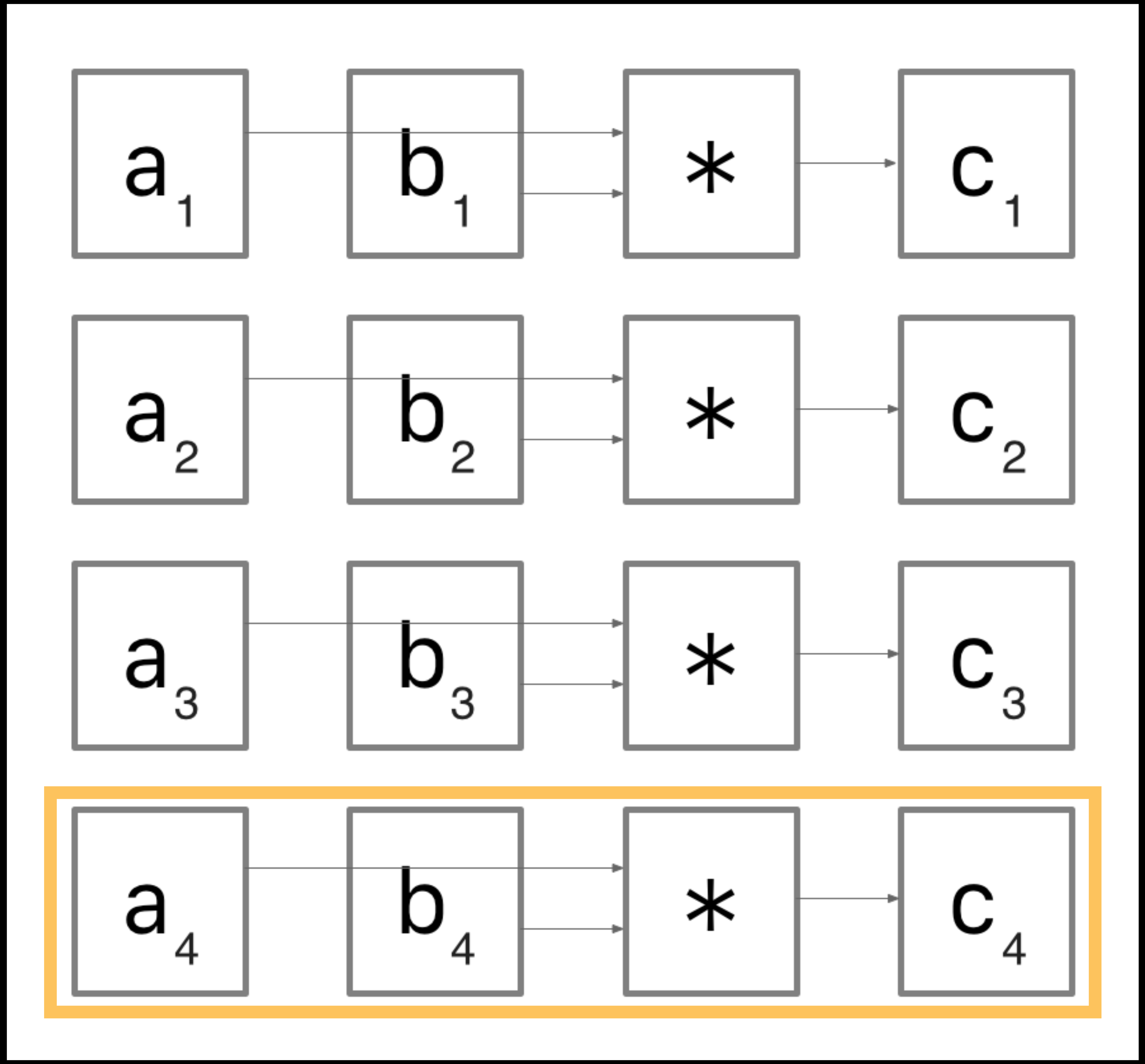
Accelerate

Scalar calculations execute serially

```
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]
```



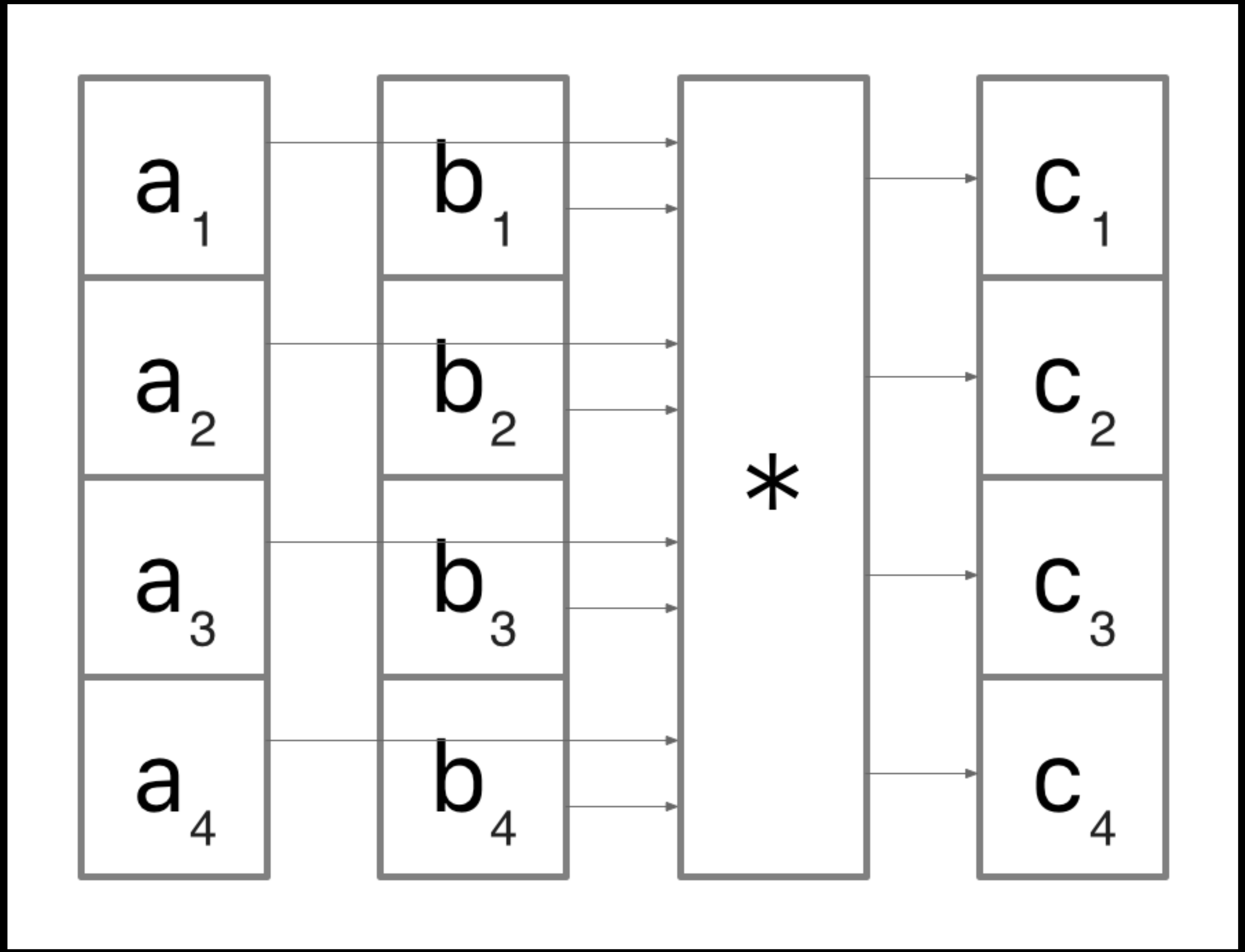
Accelerate

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

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:

```
result[i] = (a[i] + b[i]) * (c[i] - d[i])
```

vDSP for Vector Arithmetic

Vector arithmetic using scalar code

```
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

```
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

```
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

```
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

```
let result = source.map {
    return UInt16($0.rounded(.towardZero))
}
```

vDSP for Type Conversion

Using existing API

```
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

```
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

```
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

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

vDSP for Discrete Fourier Transform

Using new API

```
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

```
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

```
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

```
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:

```
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

```
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

```
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

```
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

```
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

```
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

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

```
let a: [Float] = …

let result = a.map {
    sqrt($0)
}
```

vForce

Calculating square roots using existing vDSP API

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

var n = Int32(result.count)

vvsqrtf(&result,
        a,
        &n)
```

vForce

Calculating square roots using new vDSP API

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

vForce.sqrt(a,
            result: &result)
```

vForce

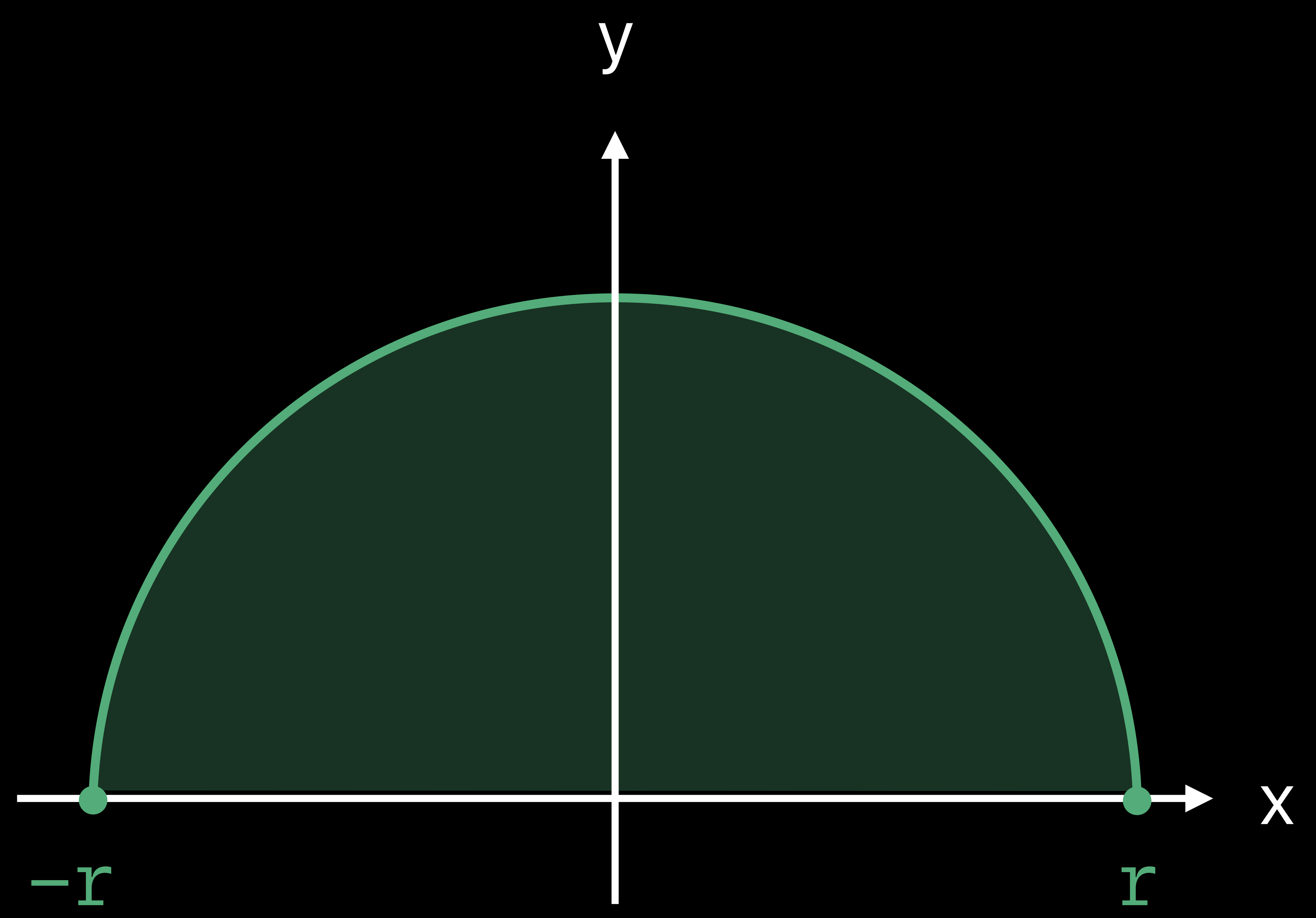
Calculating square roots using new self-allocating API

```
let result = vForce.sqrt(a)
```

Quadrature

Quadrature

Integrate a function over an interval




```
y = sqrt(r * r - x * x)
```

Quadrature

Defining the integrate function using existing API

```
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

```
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

```
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)
}
```

Quadrature

Using alternative integrator with existing API

```
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)
}
```

vImage

vImage

vImage

Core Graphics interoperability

vImage

Core Graphics interoperability

Core Video interoperability

vImage

Core Graphics interoperability

Core Video interoperability

Alpha blending

vImage

Core Graphics interoperability

Core Video interoperability

Alpha blending

Format conversions

vImage

Core Graphics interoperability

Core Video interoperability

Alpha blending

Format conversions

Histogram operations

vImage

Core Graphics interoperability

Core Video interoperability

Alpha blending

Format conversions

Histogram operations

Convolution

vImage

Core Graphics interoperability

Core Video interoperability

Alpha blending

Format conversions

Histogram operations

Convolution

Geometry

vImage

Core Graphics interoperability

Core Video interoperability

Alpha blending

Format conversions

Histogram operations

Convolution

Geometry

Morphology

vImage

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

vImage Working with Buffers

Create a buffer from a Core Graphics image

Create format description

Instantiate the buffer

Initialize the buffer from the `CGImage`

vImage Working with Buffers

Creating a buffer from an image using existing API

```
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

```
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)")
}
```

vImage Working with Buffers

Creating a buffer from an image using new API

```
let sourceBuffer = try? vImage_Buffer(cgImage: image)
```

vImage Working with Buffers

Creating a buffer from an image using new API

```
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

```
let cgImage = vImageCreateCGImageFromBuffer(
    &sourceBuffer,
    &format,
    nil,
    nil,
    vImage_Flags(kvImageNoFlags),
    &error)
```

vImage Working with Buffers

Creating an image from a buffer using new API

```swift
let cgImage = try? sourceBuffer.createCGImage(format: format)
```

vImage 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

```
let cmykToRgbUnmanagedConverter = vImageConverter_CreateWithCGImageFormat(
    &cmykSourceImageFormat,
    &rgbDestinationImageFormat,
    nil,
    vImage_Flags(kvImageNoFlags),
    nil)
```

vImage Converting Any-to-Any

Performing the conversion using existing API

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

vImageConvert_AnyToAny(cmykToRgbConverter,
                       &cmykSourceBuffer,
                       &rgbDestinationBuffer,
                       nil,
                       vImage_Flags(kvImageNoFlags))
```

vImage Converting Any-to-Any

Using new API

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

vImage Converting Any-to-Any

Using new API

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

try? converter?.convert(source: cmykSourceBuffer,
                        destination: &rgbDestinationBuffer)
```

vImage Working with CV Image Formats

Create format description from `CVPixelBuffer`

Calculate channel count

vImage Working with CV Image Formats

Using existing API

```
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

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

vImage Working with CV Image Formats

Using new API

```
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)

LINPACK Benchmark

Performance in GFLOPS (Bigger is better)

LINPACK Benchmark

Performance in GFLOPS (Bigger is better)

LINPACK Benchmark

Performance in GFLOPS (Bigger is better)

BLAS

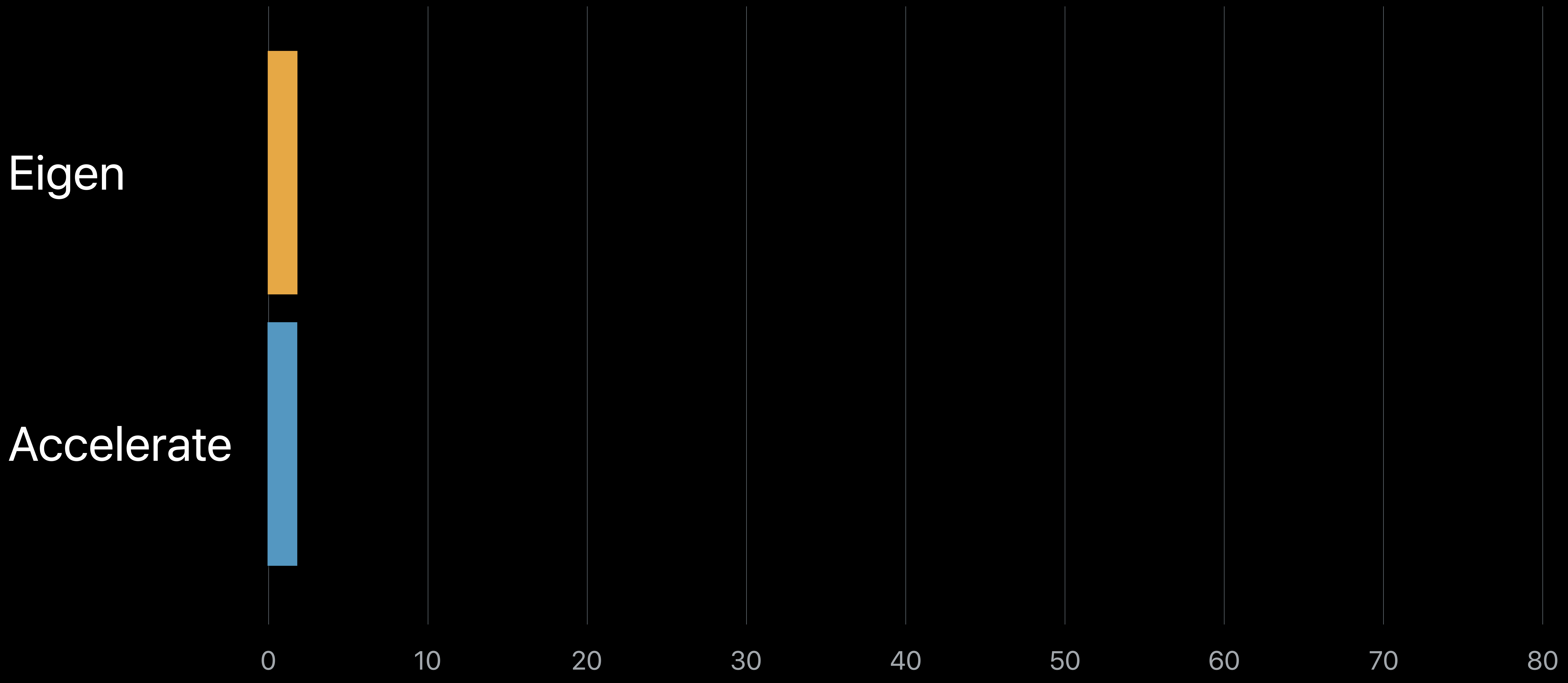
SGEMM (Single Precision General Matrix Multiply)

General matrix multiply

- Is the workhorse for other matrix-matrix operations
- Matrix-matrix operations comprise most of the work done in matrix solvers

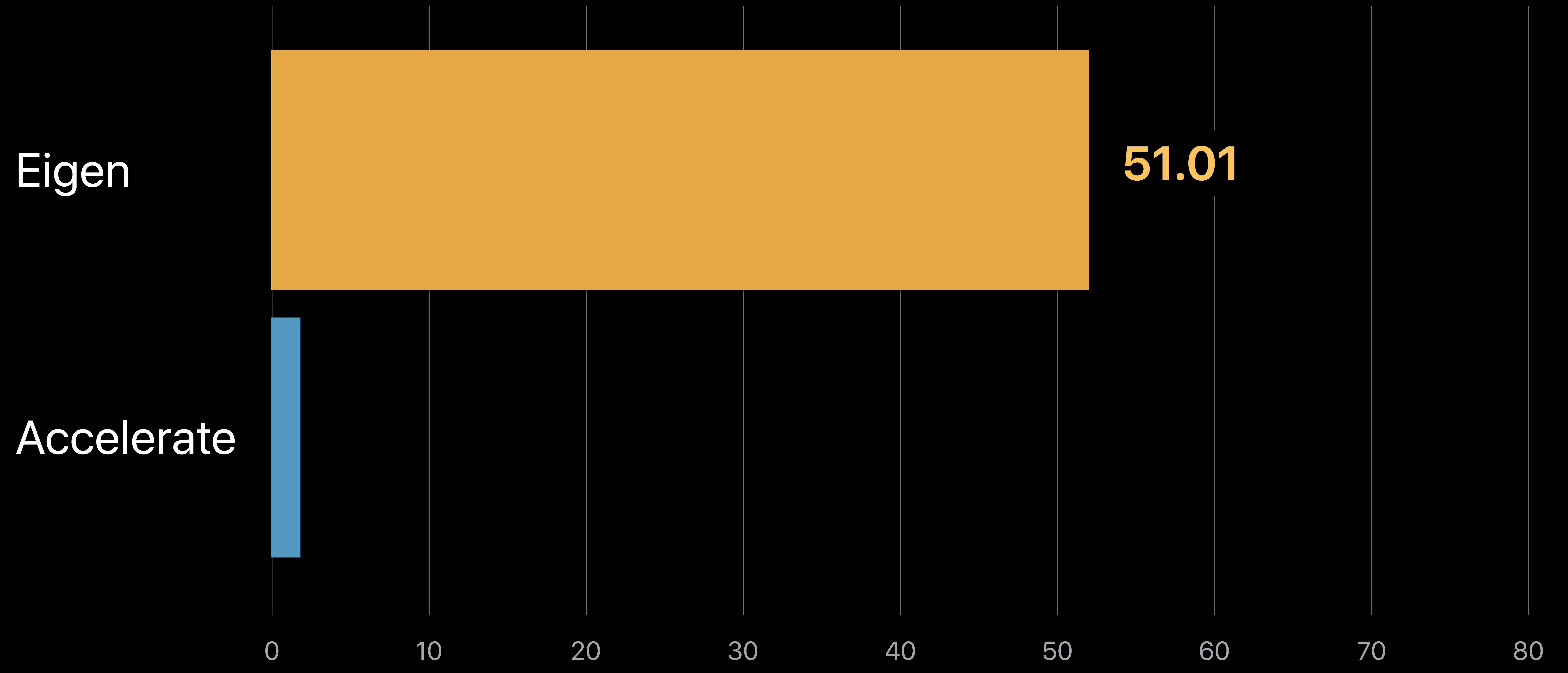
SGEMM: iPhone XS

Performance in GFLOPS (Bigger is better)



SGEMM: iPhone XS

Performance in GFLOPS (Bigger is better)



SGEMM: iPhone XS

Performance in GFLOPS (Bigger is better)

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

WWDC19