SIMD & Co

A tour through Accelerate.framework

CocoaHeads Aachen | 2023-03-30 | cocoaheads.de

Torsten Kammer @zcochrane https://github.com/cochrane/SIMDDemo

Accelerate.framework

Vast collection of sub-frameworks to get the most out of Apple's most powerful product:

The Power Mac G4 (...and newer)

Part one: SIMD

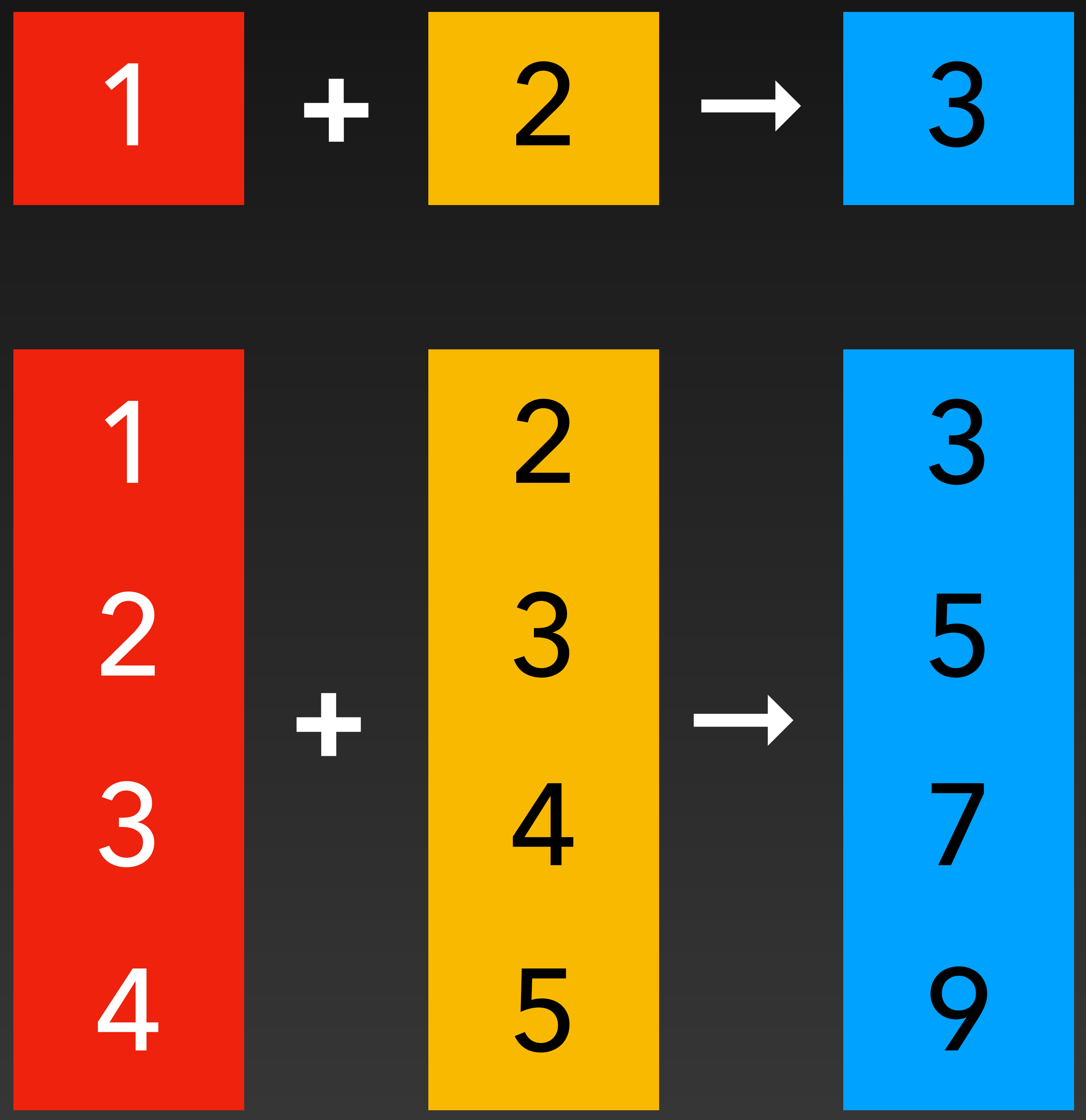
Single Instruction, Multiple Data

Normal:

a = b + c

SIMD:

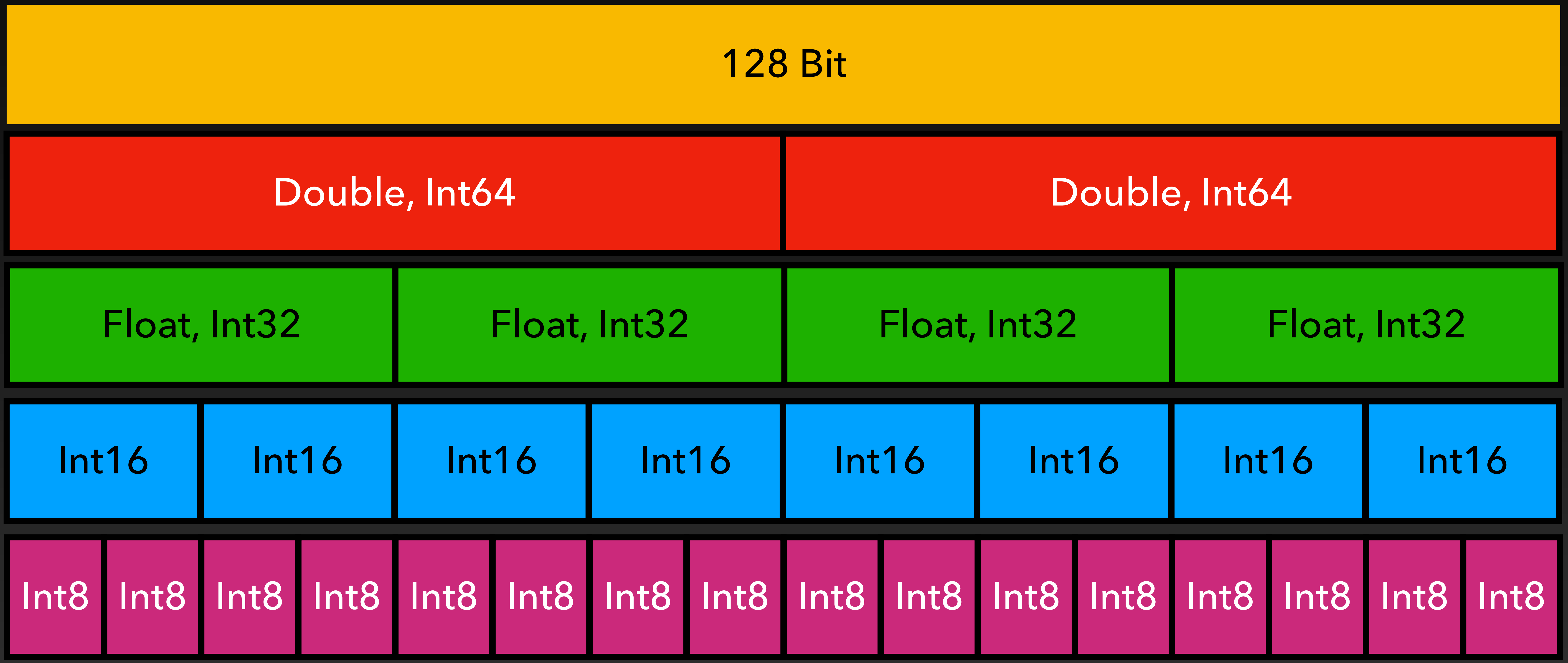
a = b + c



SIMD for Apple

- At all since 1999, everywhere since 2003
- ~~PowerPC: Altivec~~
- Intel: SSE2, SSE3, SSE4, AVX, AVX2
- ARM: Neon

SIMD Hardware



(Intel: Also 256 Bit, 512 Bit available.
And 64 Bit if you want.)

SIMD in Swift

- Built-in types
- SIMD2< SIMDScalar >, SIMD4< SIMDScalar >, … SIMD64< SIMDScalar >
- SIMDScalar: Float, Double, Int, UInt, (U)Int8…(U)Int64
- Some combinations exceed 128 Bit (at times very clearly)
  - Legal but potentially slower; may become faster in future
- Special case: SIMD3 - actually a SIMD4 with fourth lane hidden

SIMD in Swift

Generic Operations

- Normal operators: Act per lane
- Most maths functions are defined for SIMD
- Operations with scalar: Automatically extended

```swift
let a: SIMD3<Float> = generateVector()
let b: SIMD3<Float> = generateVector()
let c: SIMD3<Float> = a + b * 2
```

Booleans with Swift

```
let tooSmall = vector .< deadzone
let adjusted = vector - deadzone
return adjusted.replacing(with: 0.0, where: tooSmall)
```

- Comparison operators start with dot
- Compare per lane, return SIMDMask
- Functions any(), all() give normal boolean
- SIMD*N*.replacing to choose per lane

Other interesting operations

- scalarCount, subscript - can iterate over elements
- Init from sequence
- Horizontal min, max, add

```
let indices: SIMD2<Int> = …
let vector: SIMD4<Float> = …
let chosenElements: SIMD2<Float> = vector[indices]
```

Philosophy of using SIMD

Approach 1: Exactly two, three, four floats

```
let direction: SIMD3<Float> = positions[1] - positions[0]
```

- X, Y, Z, W, or R, G, B, A
- Easy to reason about
- Use as general 2D/3D/3D affine point/direction structure
- Wastes bits (rarely care about w)
- Wastes bits on hypothetical future hardware

Philosophy of SIMD

Approach 2: N numbers, N depends on CPU

```
let dirX: SIMD64<Float> = positionsX[1] - positionsX[0]
let dirY: SIMD64<Float> = positionsY[1] - positionsY[0]
let dirZ: SIMD64<Float> = positionsZ[1] - positionsZ[0]
```

- Every element has same meaning, just for different element
- Higher performance
- At times more difficult, more code
- Requires special load, store logic
- Not that easy in Swift

Part 2

Accelerate

(...and others)

simd

The Library

- C-based, in /usr
- Geometry functions for float and double
- Confusing name: SIMD*N*<Type> does not require simd
- Provides own type aliases e.g. simd_float4
- Functions for vectors, matrices, quaternions

simd

Vectors

- Generic functions:
  - min, max, abs, clamp, sign, min element, …
- Geometric functions:
  - Normalize, Distance, reflect, refract
  - Dot and cross products
  - Very specific intersection tests

simd

Matrices

- Float, Double matrices from 2x2 to 4x4
  - Swift operators for addition, multiplication
- Inverse (full)
- No methods to generate standard matrices (rotation, projection…), need to write these yourself

simd

Quaternions

- Generalisation of complex numbers to four dimensions used to represent rotations in 3D space that I don’t really understand
- Supports all standard operations
  - Multiplication with each other
  - Transforming vectors
  - Turning from and into matrices

vForce

Finally Accelerate.framework

cos, sqrt, floor, …
for large arrays

vImage

- Image Processing
  - Slower than Core Image
  - Can do things CI can’t
  - Processing before/after CoreImage, OpenGL, Metal…
- Scaling, Shearing, Flipping
- Format Conversion
- Histograms

vDSP

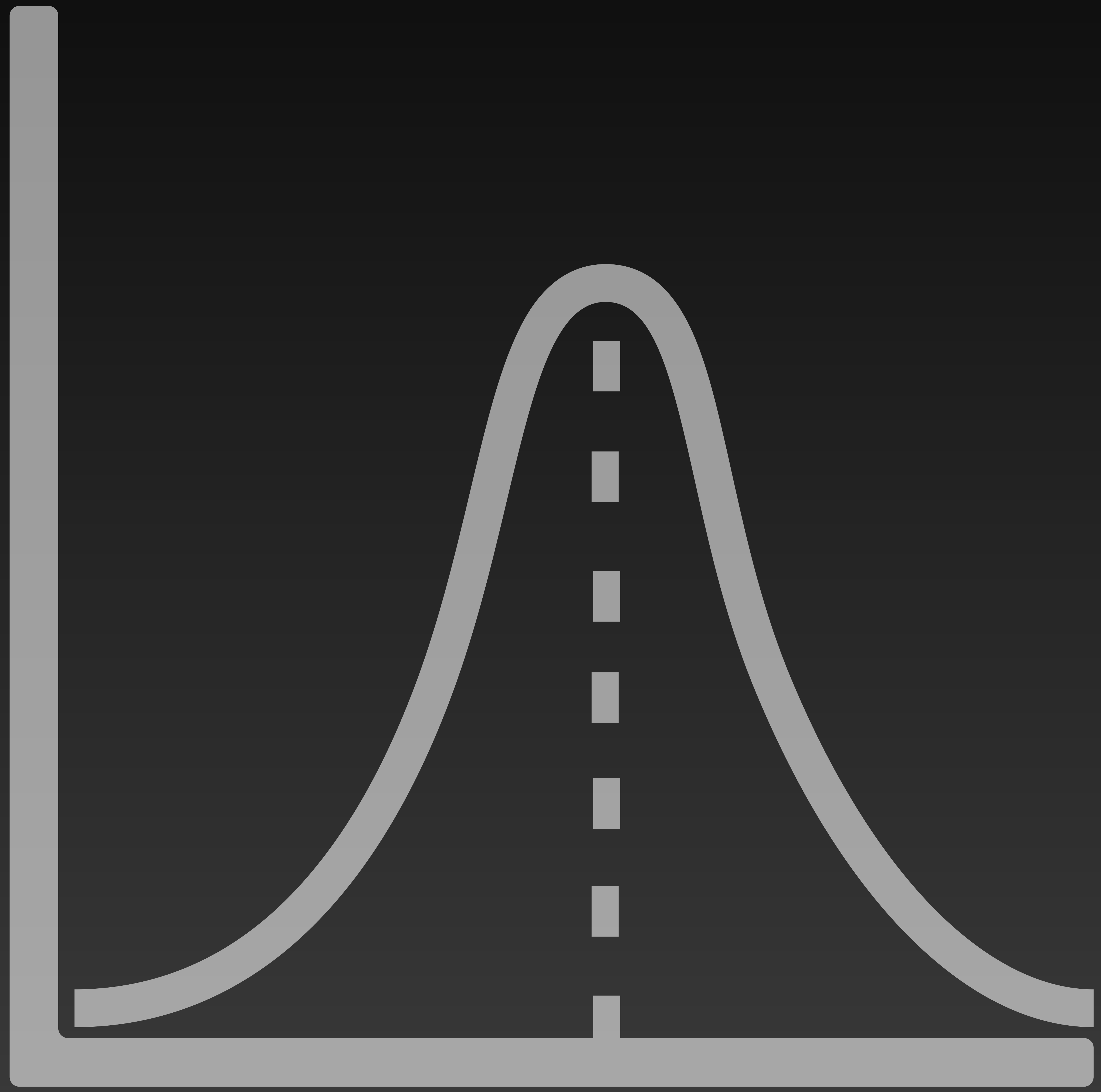
- DSP operations
- Fourier Transforms, Cosine Transforms

vBigNum

- Basic operations on 256-1024 bit numbers
- + - * /
- Signed and unsigned
- Interface not very Swift-like

Quadrature

- Numeric integration of functions
- Special Swift API

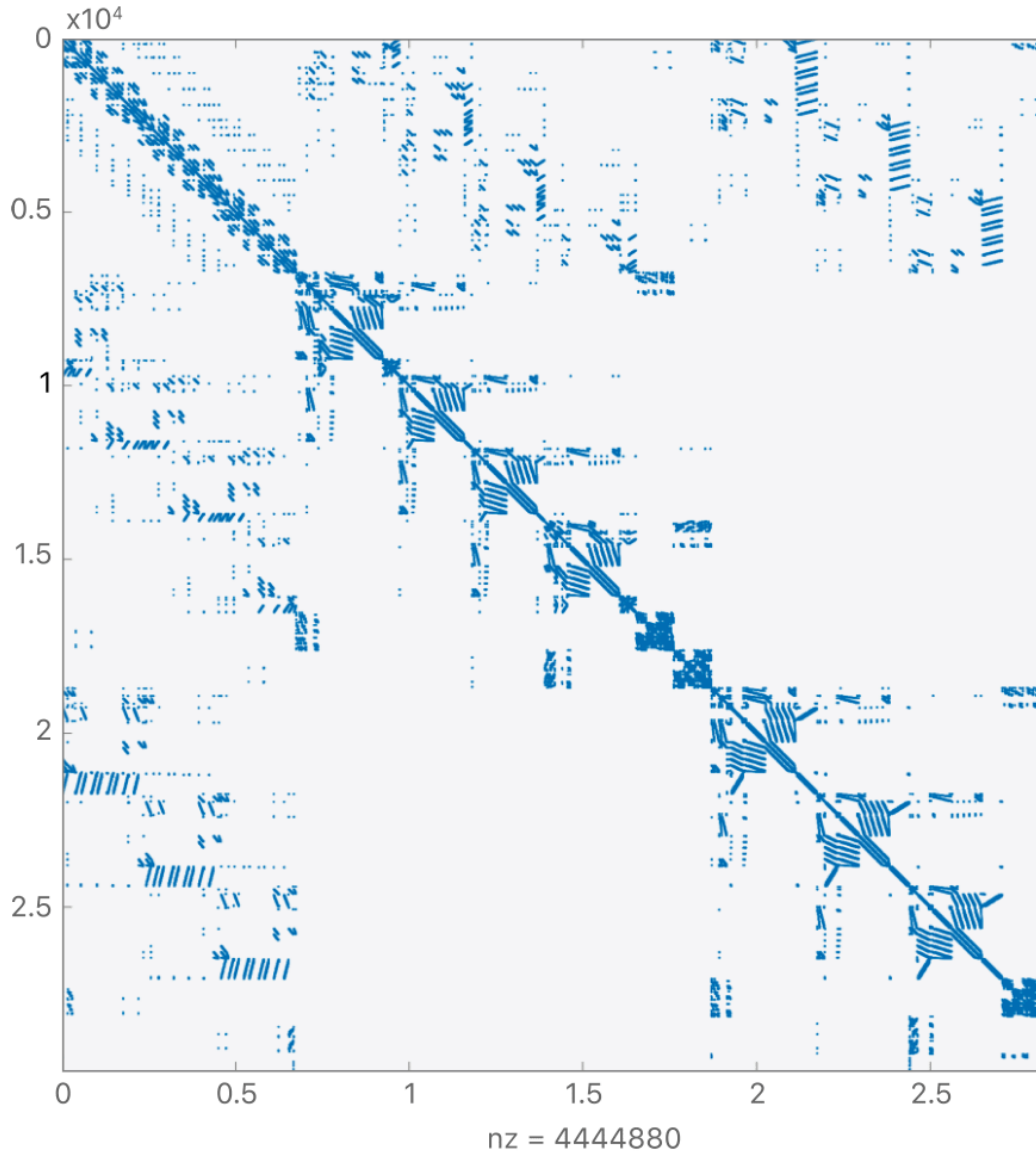


BLAS, LAPACK

- Industry standard for large linear maths operations
- Apple’s version specifically optimised

Sparse Solvers

- Equivalent to LAPACK for sparse matrices
- Special data structures to describe which items are non-zero



BNSS

- Fast operations for neural networks
- Probably useful if you don’t like CoreML

Spatial

- 3D geometry library
- Types for transformations, primitives
- Uses own type system - not designed to easily generate matrices for use anywhere

Compression, Apple Archive

- Compress raw data
- Write and read Zip files

Questions?