The Accelerate framework for iPhone OS

We work hard so your phone doesn't, provided that you make good use of our APIs

Steve Canon
Senior Engineer
Vector Numerics Group

What We’ll Talk About

- ARM architecture highlights
- Overview of the Accelerate Framework
- Examples

What to Take Away

- What is in the Accelerate Framework
- Reference for documentation of the Accelerate APIs
- How to use Accelerate effectively

ARM Architecture Background

ARMv6

- iPhone
- iPhone 3G
- First and second generation iPod Touch

ARMv6

- Integer unit with 16 registers
- Hardware floating-point (VFP)
  - 32 single precision registers
  - 16 double precision registers

ARMv7

- iPhone 3GS
- Third generation iPod touch
- iPad

ARMv7

- Integer unit with 16 registers
  - Capable of executing two instructions at a time
- Legacy VFP support
- New SIMD unit, "NEON", with 16 128-bit vector registers
  - Single precision floating-point uses NEON

NEON Registers

0x00	0x00	0x00	0x00	0xdb	0x0f	0x49	0x40	0x54	0xf8	0x2d	0x40	0x00	0x00	0x80	0xbf

0x0000	0x0000	0x0fdb	0x4049	0xf854	0x402d	0x0000	0xbf80

0x00000000	0x40490fdb	0x402df854	0xbf800000

0.0f	3.1415927f	2.7182817f	-1.0f

0x40490fdb00000000	0xbf800000402df854

NEON Examples

vadd.i16 q0, q1, q2

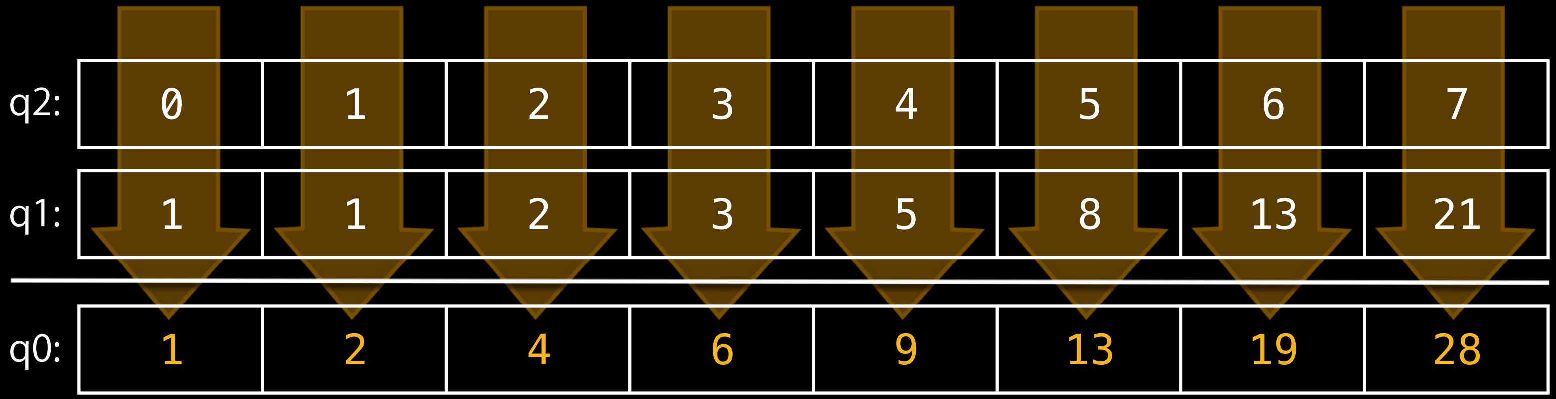
- Simultaneously add eight 16-bit integers in q2 to eight 16-bit integers in q1 and put the resulting eight 16-bit integers in q0.

q2:	0	1	2	3	4	5	6	7
q1:	1	1	2	3	5	8	13	21
q0:								

NEON Examples

vadd.i16 q0, q1, q2

- Simultaneously add eight 16-bit integers in q2 to eight 16-bit integers in q1 and put the resulting eight 16-bit integers in q0.



q2:	0	1	2	3	4	5	6	7
q1:	1	1	2	3	5	8	13	21
q0:	1	2	4	6	9	13	19	28

NEON Examples

vmul.f32 q0, q1, q2

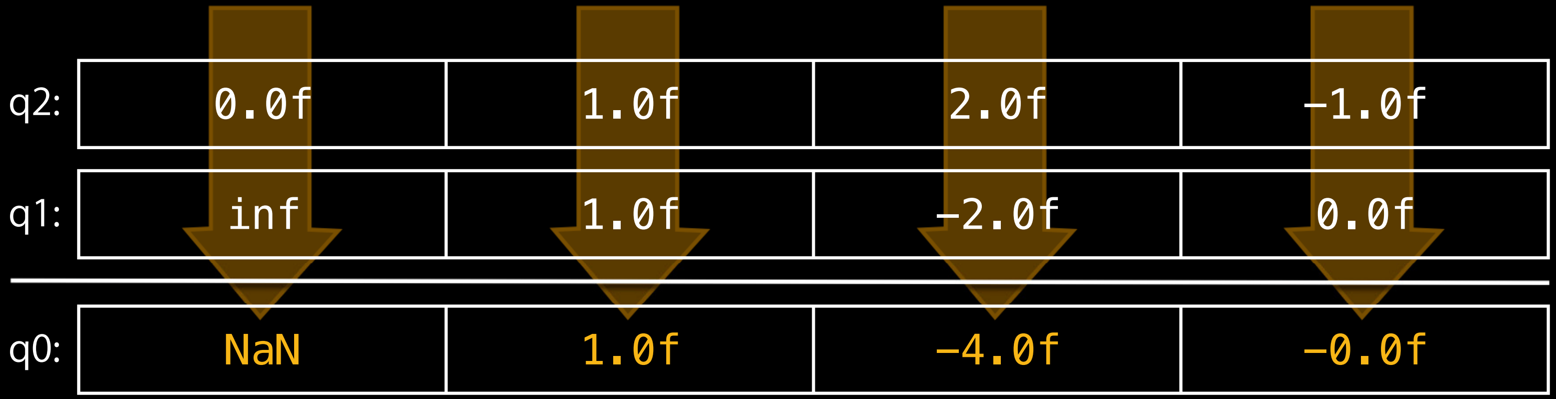
- Simultaneously multiply four 32-bit floats from q2 by four 32-bit floats from q1 and store the resulting four 32-bit floats in q0

q2:	0.0f	1.0f	2.0f	-1.0f
q1:	inf	1.0f	-2.0f	0.0f
q0:				

NEON Examples

vmul.f32 q0, q1, q2

- Simultaneously multiply four 32-bit floats from q2 by four 32-bit floats from q1 and store the resulting four 32-bit floats in q0



q2:	0.0f	1.0f	2.0f	-1.0f
q1:	inf	1.0f	-2.0f	0.0f
q0:	NaN	1.0f	-4.0f	-0.0f

What Is Available in NEON

- Several types of load and store instructions
- Single precision floating-point arithmetic
    - Basic operations, conversions, reciprocal and square root estimates
- Integer arithmetic
    - 8-, 16-, and 32-bit, signed and unsigned integer operations
    - Basic operations, multiply-accumulate, wide multiplies
    - Saturating operations
    - Fixed-point arithmetic

What Is Not Available in NEON

- Double precision floating-point uses the older VFP extension
    - Much slower than NEON

NEON

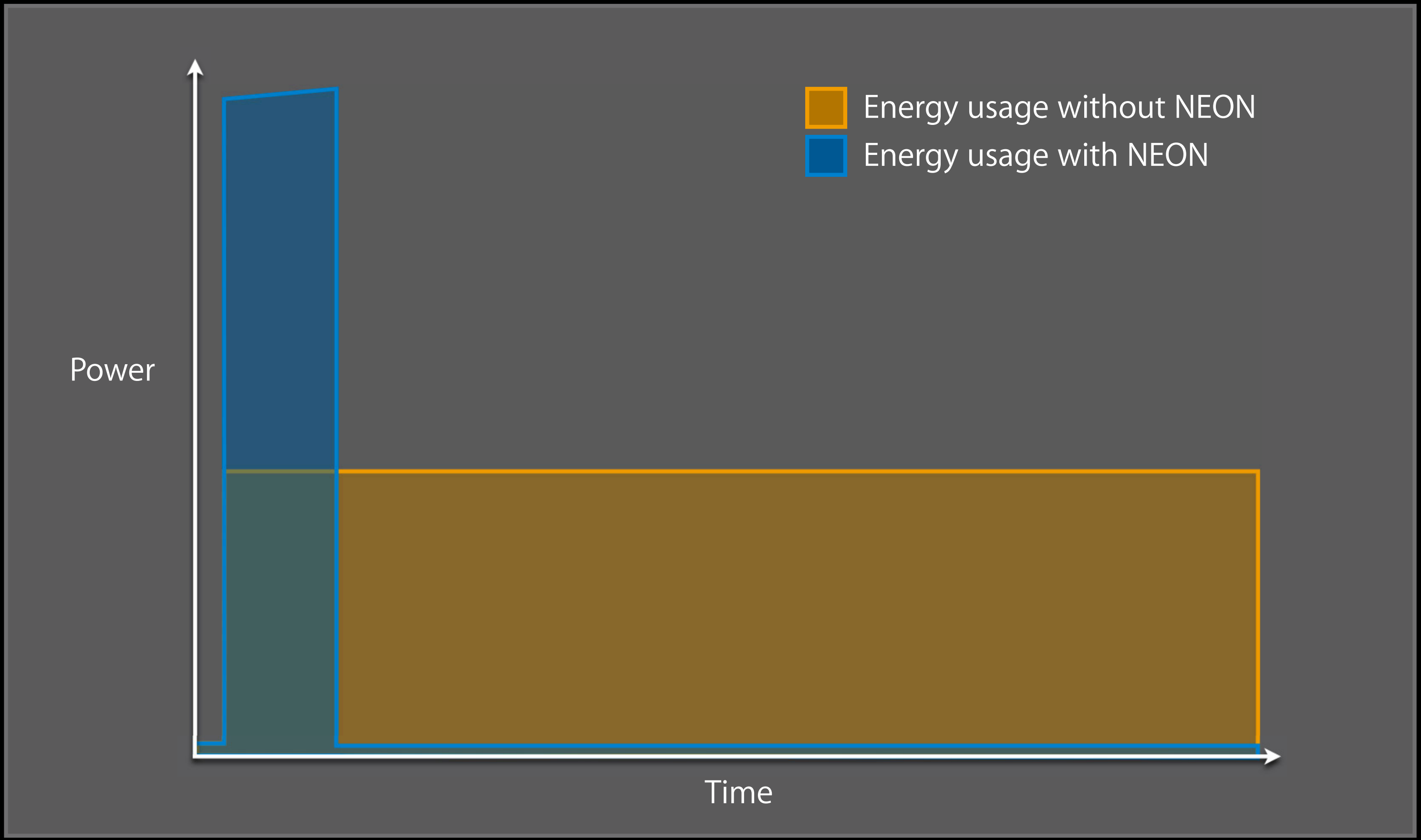
More power, less energy

Energy used is the total instantaneous power used over time

$$\Delta E = \int P dt$$

NEON

More power, less energy



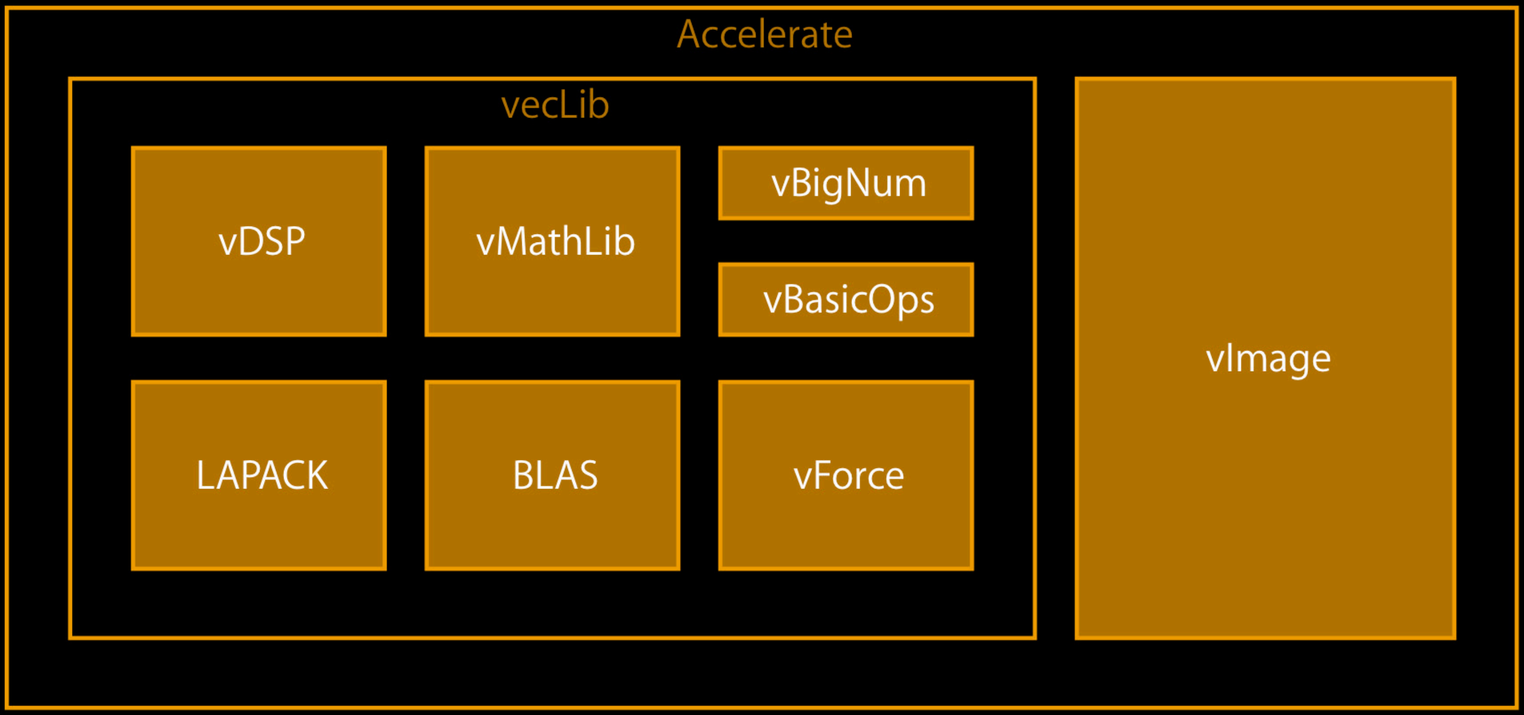
The Accelerate Framework

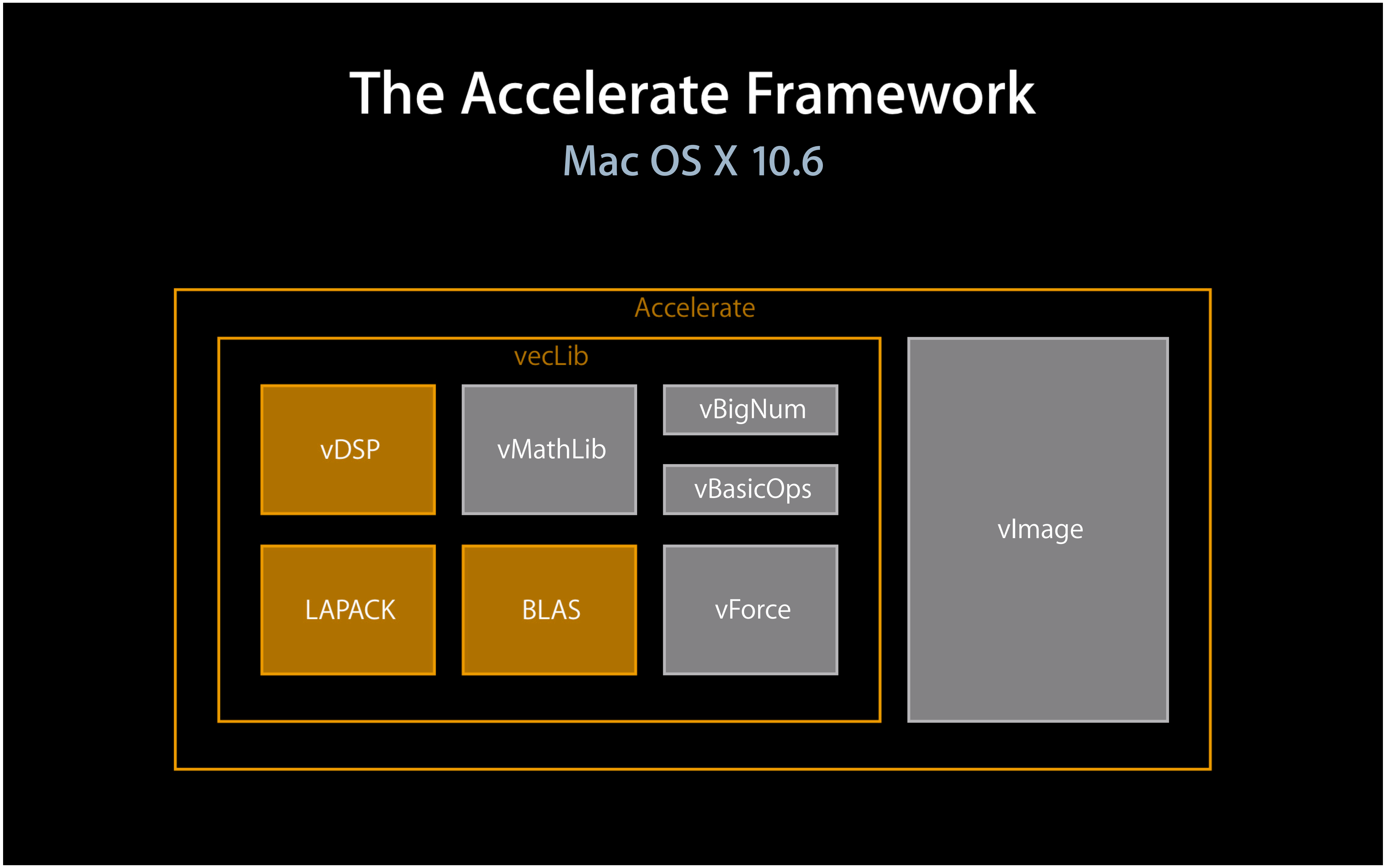
“Over 2000 APIs for hardware accelerated math functions”

iOS 4 Introduction

The Accelerate Framework

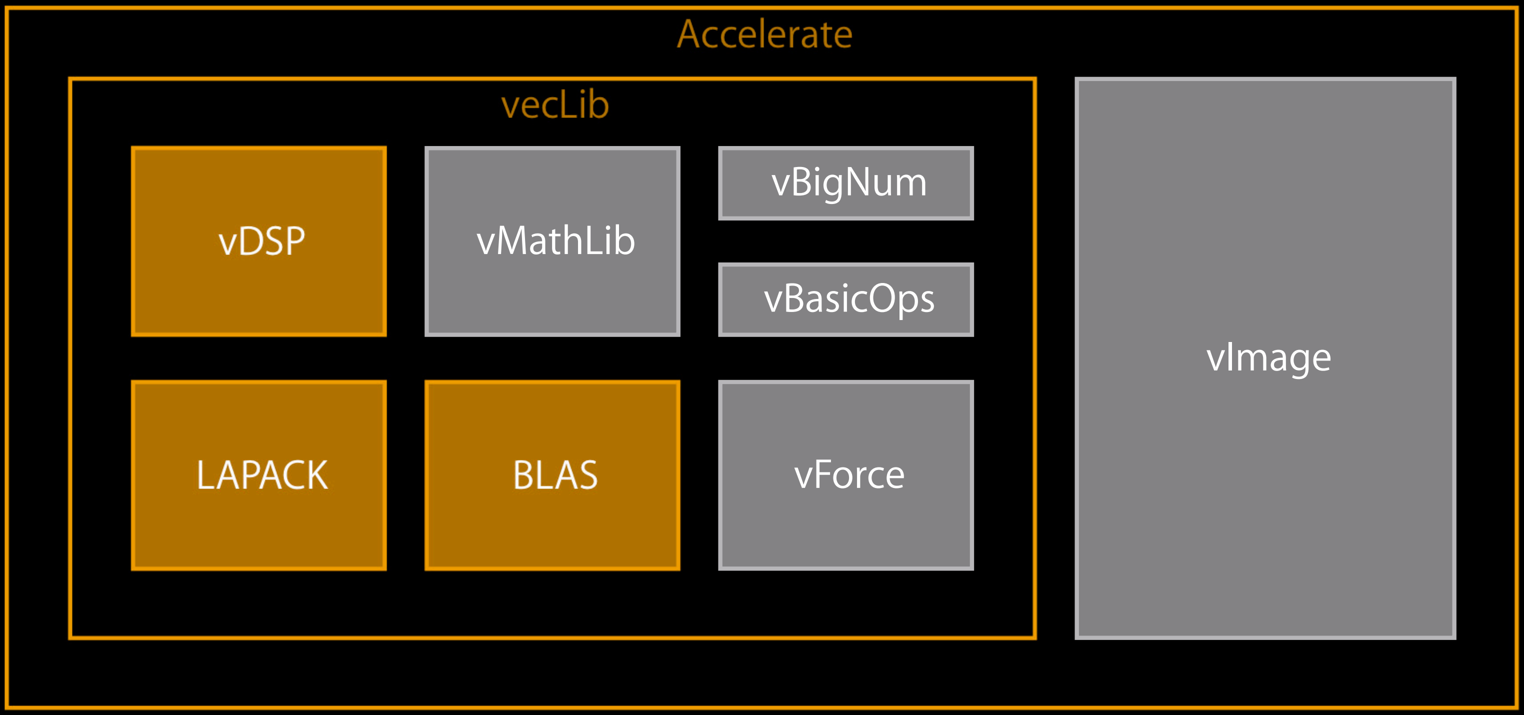
Mac OS X 10.6





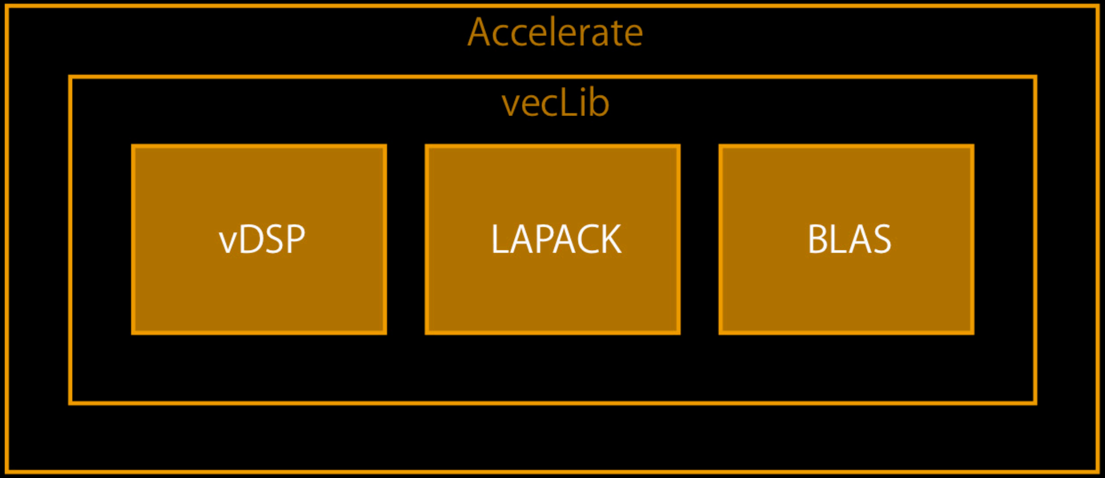
The Accelerate Framework

Mac OS X 10.6



The Accelerate Framework

iOS 4



vDSP

Digital Signal Processing

Dot Product

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} a_i b_i$$

Dot Product

$$\mathbf{a} \cdot \mathbf{b} = a_0 b_0 + a_1 b_1 + \ldots + a_{n-1} b_{n-1}$$

Using a “for loop”

```
float dotProduct = 0.f;
for (int i=0; i<n; ++i)
    dotProduct += a[i] * b[i];
```

Using Accelerate

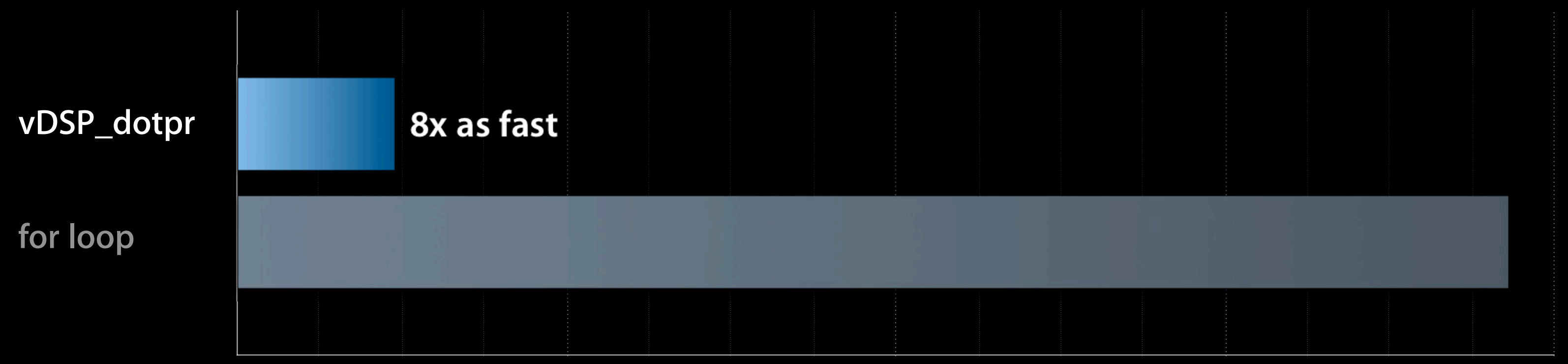
```
#include <Accelerate/Accelerate.h>

float dotProduct;
vDSP_dotpr(a, 1, b, 1, &dotProduct, n);
```

Using Accelerate

```
#include <Accelerate/Accelerate.h>

float dotProduct;
vDSP_dotpr(a, 1, b, 1, &dotProduct, n);
```

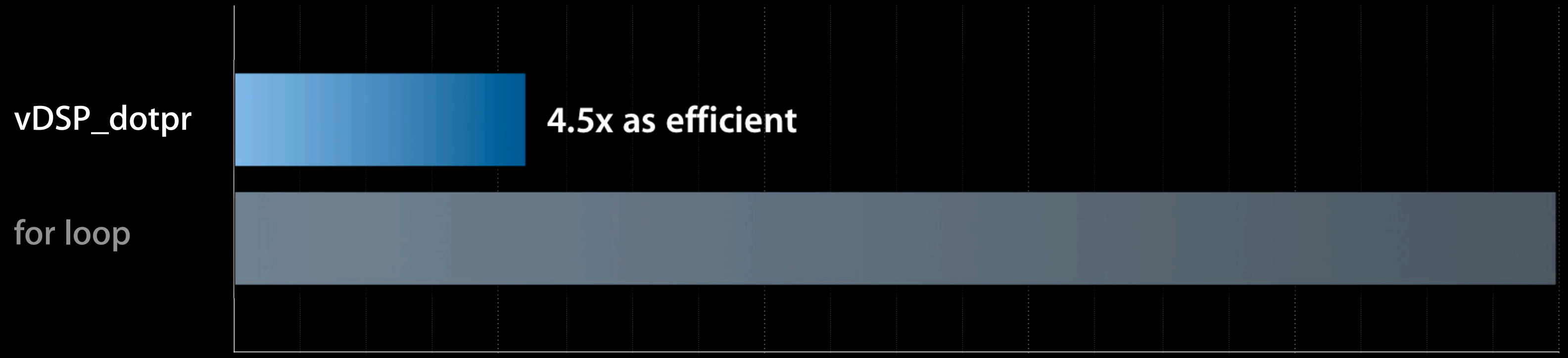
Execution Time

Length 1024 dot product on ARMv7 (smaller bars are better)



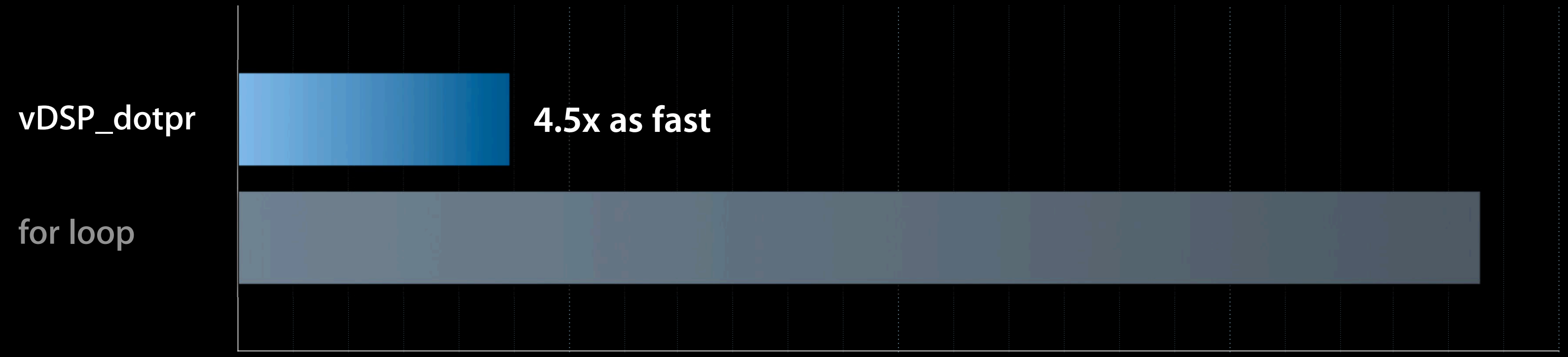
Energy Used

Length 1024 dot product on ARMv7 (smaller bars are better)



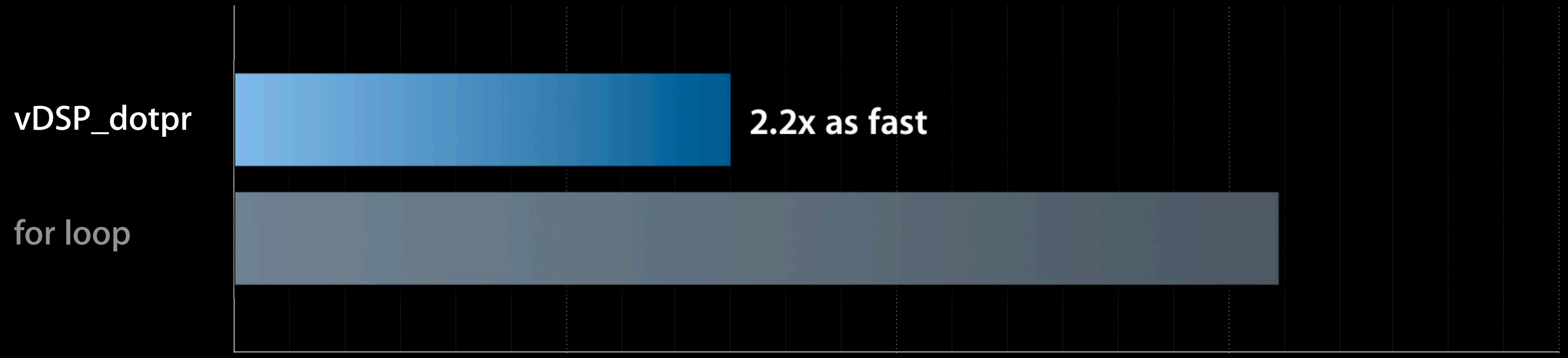
Execution Time

Length 1024 dot product on ARMv6 (smaller bars are better)



Execution Time

Length 3 dot product on ARMv7 (smaller bars are better)



Why Is vDSP_dotpr Faster?

for loop

```
for (int i=0; i<n; ++i)
    dotProduct += a[i] * b[i];
```

```
0:  mov.w     r3,  lr,  lsl #2
    add.w     ip,  r0,  r3
    add       r3,  r1
    flds      s12, [ip]
    flds      s14, [r3]
    vmul.f32  d7,  d6,  d7
    vadd.f32  d5,  d5,  d7
    add.w     lr,  lr,  #1
    cmp       lr,  r2
    blt.n     0b
```

Why Is vDSP_dotpr Faster?

for loop

```
for (int i=0; i<n; ++i)
    dotProduct += a[i] * b[i];
```

```
0:  mov.w     r3,  lr,  lsl #2
    add.w     ip,  r0,  r3
    add       r3,  r1
    flds      s12, [ip]
    flds      s14, [r3]
    vmul.f32  d7,  d6,  d7
    vadd.f32  d5,  d5,  d7
    add.w     lr,  lr,  #1
    cmp       lr,  r2
    blt.n     0b
```

Why Is vDSP_dotpr Faster?

for loop

```
for (int i=0; i<n; ++i)
    dotProduct += a[i] * b[i];
```

```
0:  mov.w     r3,  lr,  lsl #2
    add.w     ip,  r0,  r3
    add       r3,  r1
    flds      s12, [ip] |   <———  Load the ith float from each array
    flds      s14, [r3] |
    vmul.f32  d7,  d6,  d7
    vadd.f32  d5,  d5,  d7
    add.w     lr,  lr,  #1
    cmp       lr,  r2
    blt.n     0b
```

Why Is vDSP_dotpr Faster?

for loop

```
for (int i=0; i<n; ++i)
    dotProduct += a[i] * b[i];
```

```
0:  mov.w     r3,  lr,  lsl #2
    add.w     ip,  r0,  r3
    add       r3,  r1
    flds      s12, [ip]
    flds      s14, [r3]
    vmul.f32  d7,  d6,  d7  ←—— Multiply the floats we just loaded
    vadd.f32  d5,  d5,  d7
    add.w     lr,  lr,  #1
    cmp       lr,  r2
    blt.n     0b
```

Why Is vDSP_dotpr Faster?

for loop

```
for (int i=0; i<n; ++i)
    dotProduct += a[i] * b[i];
```

```
0:  mov.w     r3,  lr,  lsl #2
    add.w     ip,  r0,  r3
    add       r3,  r1
    flds      s12, [ip]
    flds      s14, [r3]
    vmul.f32  d7,  d6,  d7
    vadd.f32  d5,  d5,  d7  ←——  Add the product to the running sum
    add.w     lr,  lr,  #1
    cmp       lr,  r2
    blt.n     0b
```

Why Is vDSP_dotpr Faster?

for loop

```
for (int i=0; i<n; ++i)
    dotProduct += a[i] * b[i];
```

```
0:  mov.w      r3,  lr,  lsl #2
    add.w      ip,  r0,  r3
    add        r3,  r1
    flds       s12, [ip]
    flds       s14, [r3]
    vmul.f32   d7,  d6,  d7
    vadd.f32   d5,  d5,  d7
    add.w      lr,  lr,  #1
    cmp        lr,  r2
    blt.n      0b
```

Stall!
This multiply depends on the result of the previous instructions; it can't execute until the processor has finished loading

Why Is vDSP_dotpr Faster?

for loop

```
for (int i=0; i<n; ++i)
    dotProduct += a[i] * b[i];
```

```
0:  mov.w      r3,  lr,  lsl #2
    add.w      ip,  r0,  r3
    add        r3,  r1
    flds       s12, [ip]
    flds       s14, [r3]
    vmul.f32   d7,  d6,  d7
    vadd.f32   d5,  d5,  d7  <------ Stall!
    add.w      lr,  lr,  #1
    cmp        lr,  r2
    blt.n      0b
```

Stall!
This add depends on the result of the multiply, so it can't execute until the multiply is finished

Why Is vDSP_dotpr Faster?

vDSP_dotpr

```
0:  subs      r5,   #16
    vadd.f32  q0,   q0, q8
    vld1.f32 {q8}, [r0,:128]!
    vmul.f32  q10,  q10, q14
    vld1.f32 {q12},[r2,:128]!
    vadd.f32  q1,   q1, q9
    vld1.f32 {q9}, [r0,:128]!
    vmul.f32  q11,  q11, q15
    vld1.f32 {q13},[r2,:128]!
    vadd.f32  q2,   q2, q10
    vld1.f32 {q10},[r0,:128]!
    vmul.f32  q8,   q8, q12
    vld1.f32 {q14},[r2,:128]!
    vadd.f32  q3,   q3, q11
    vld1.f32 {q11},[r0,:128]!
    vmul.f32  q9,   q9, q13
    vld1.f32 {q15},[r2,:128]!
    bpl       0b
```

Why Is vDSP_dotpr Faster?

vDSP_dotpr

```
0:  subs      r5,   #16
    vadd.f32  q0,   q0, q8
    vld1.f32 {q8}, [r0,:128]!
    vmul.f32  q10,  q10, q14
    vld1.f32 {q12},[r2,:128]!
    vadd.f32  q1,   q1, q9
    vld1.f32 {q9}, [r0,:128]!
    vmul.f32  q11,  q11, q15
    vld1.f32 {q13},[r2,:128]!
    vadd.f32  q2,   q2, q10
    vld1.f32 {q10},[r0,:128]!
    vmul.f32  q8,   q8, q12
    vld1.f32 {q14},[r2,:128]!
    vadd.f32  q3,   q3, q11
    vld1.f32 {q11},[r0,:128]!
    vmul.f32  q9,   q9, q13
    vld1.f32 {q15},[r2,:128]!
    bpl       0b
```

Why Is vDSP_dotpr Faster?

vDSP_dotpr

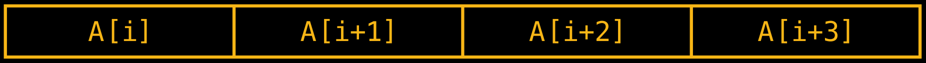
```
0:  subs      r5,   #16
    vadd.f32  q0,   q0, q8
    vld1.f32 {q8}, [r0,:128]!
    vmul.f32  q10,  q10, q14
    vld1.f32 {q12},[r2,:128]!
    vadd.f32  q1,   q1, q9
    vld1.f32 {q9}, [r0,:128]!
    vmul.f32  q11,  q11, q15
    vld1.f32 {q13},[r2,:128]!
    vadd.f32  q2,   q2, q10
    vld1.f32 {q10},[r0,:128]!
    vmul.f32  q8,   q8, q12
    vld1.f32 {q14},[r2,:128]!
    vadd.f32  q3,   q3, q11
    vld1.f32 {q11},[r0,:128]!
    vmul.f32  q9,   q9, q13
    vld1.f32 {q15},[r2,:128]!
    bpl       0b
```

Why Is vDSP_dotpr Faster?

vDSP_dotpr

```
0:  subs       r5,   #16
    vadd.f32  q0,   q0, q8
    vld1.f32 {q8}, [r0,:128]!
    vmul.f32  q10,  q10, q14
    vld1.f32 {q12},[r2,:128]!
    vadd.f32  q1,   q1, q9
    vld1.f32 {q9}, [r0,:128]!
    vmul.f32  q11,  q11, q15
    vld1.f32 {q13},[r2,:128]!
    vadd.f32  q2,   q2, q10
    vld1.f32 {q10},[r0,:128]!
    vmul.f32  q8,   q8, q12
    vld1.f32 {q14},[r2,:128]!
    vadd.f32  q3,   q3, q11
    vld1.f32 {q11},[r0,:128]!
    vmul.f32  q9,   q9, q13
    vld1.f32 {q15},[r2,:128]!
    bpl        0b
```

← Load four floats from A

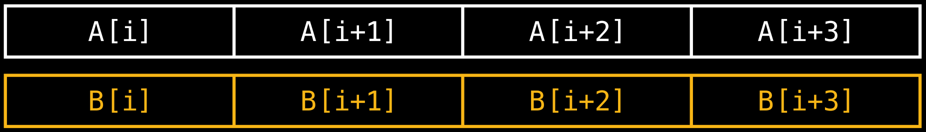


Why Is vDSP_dotpr Faster?

vDSP_dotpr

```
0:  subs      r5,    #16
    vadd.f32  q0,    q0, q8
    vld1.f32 {q8}, [r0,:128]!
    vmul.f32  q10,   q10, q14
    vld1.f32 {q12},[r2,:128]!
    vadd.f32  q1,    q1, q9
    vld1.f32 {q9}, [r0,:128]!
    vmul.f32  q11,   q11, q15
    vld1.f32 {q13},[r2,:128]!
    vadd.f32  q2,    q2, q10
    vld1.f32 {q10},[r0,:128]!
    vmul.f32  q8,    q8, q12
    vld1.f32 {q14},[r2,:128]!
    vadd.f32  q3,    q3, q11
    vld1.f32 {q11},[r0,:128]!
    vmul.f32  q9,    q9, q13
    vld1.f32 {q15},[r2,:128]!
    bpl       0b
```

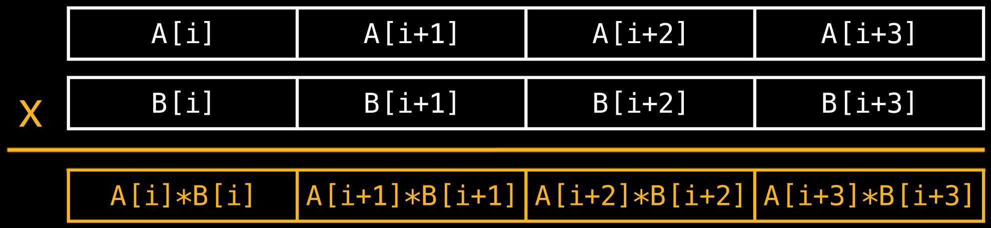
← Load four floats from B



Why Is vDSP_dotpr Faster?

vDSP_dotpr

```
0:  subs      r5,   #16
    vadd.f32  q0,   q0, q8
    vld1.f32 {q8}, [r0,:128]!
    vmul.f32  q10,  q10, q14
    vld1.f32 {q12},[r2,:128]!
    vadd.f32  q1,   q1, q9
    vld1.f32 {q9}, [r0,:128]!
    vmul.f32  q11,  q11, q15
    vld1.f32 {q13},[r2,:128]!
    vadd.f32  q2,   q2, q10
    vld1.f32 {q10},[r0,:128]!
    vmul.f32  q8,   q8, q12
    vld1.f32 {q14},[r2,:128]!
    vadd.f32  q3,   q3, q11
    vld1.f32 {q11},[r0,:128]!
    vmul.f32  q9,   q9, q13
    vld1.f32 {q15},[r2,:128]!
    bpl       0b
```



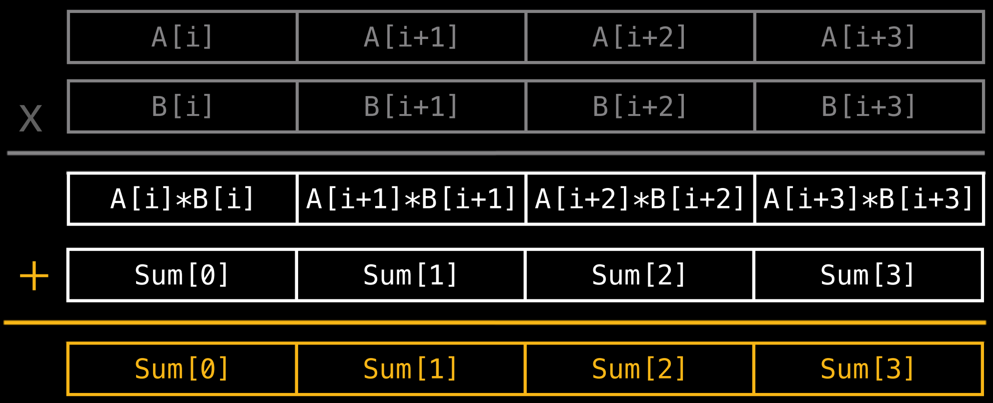
Multiply the four pairs of floats we loaded to get four products

Why Is vDSP_dotpr Faster?

vDSP_dotpr

```
0:  subs      r5,    #16
    vadd.f32  q0,    q0, q8
    vld1.f32 {q8}, [r0,:128]!
    vmul.f32  q10,   q10, q14
    vld1.f32 {q12},[r2,:128]!
    vadd.f32  q1,   q1, q9
    vld1.f32 {q9}, [r0,:128]!
    vmul.f32  q11,   q11, q15
    vld1.f32 {q13},[r2,:128]!
    vadd.f32  q2,    q2, q10
    vld1.f32 {q10},[r0,:128]!
    vmul.f32  q8,    q8, q12
    vld1.f32 {q14},[r2,:128]!
    vadd.f32  q3,    q3, q11
    vld1.f32 {q11},[r0,:128]!
    vmul.f32  q9,    q9, q13
    vld1.f32 {q15},[r2,:128]!
    bpl       0b
```

Add the four products to four running sums



Easy to Use

What's Available in vDSP

- Basic operations on arrays
  - Add, subtract, multiply, conversion, accumulation, and lots more
- Convolution and correlation
- fast Fourier transform
  - Complex to complex, real to complex
  - Support for (some) non-power-of-two sizes

Data Types in vDSP

- Single and double precision
- Real and complex
- Support for strided data access

fast Fourier transform Setup

```
#include <Accelerate/Accelerate.h>
#include <stdlib.h>

const int log2n = 10;
const int n = 1 << log2n;

DSPSplitComplex input, output;
input.realp = malloc(n * sizeof *input.realp);
input.imagp = malloc(n * sizeof *input.imagp);
output.realp = malloc(n * sizeof *output.realp);
output.imagp = malloc(n * sizeof *output.imagp);

FFTSetup setup = vDSP_create_fftsetup(log2n, FFT_RADIX2);
```

fast Fourier transform Setup

```
#include <Accelerate/Accelerate.h>
#include <stdlib.h>

const int log2n = 10;
const int n = 1 << log2n;

DSPSplitComplex input, output;
input.realp = malloc(n * sizeof *input.realp);
input.imagp = malloc(n * sizeof *input.imagp);
output.realp = malloc(n * sizeof *output.realp);
output.imagp = malloc(n * sizeof *output.imagp);

FFTSetup setup = vDSP_create_fftsetup(log2n, FFT_RADIX2);
```

fast Fourier transform Setup

```
#include <Accelerate/Accelerate.h>
#include <stdlib.h>

const int log2n = 10;
const int n = 1 << log2n;

DSPSplitComplex input, output;
input.realp = malloc(n * sizeof *input.realp);
input.imagp = malloc(n * sizeof *input.imagp);
output.realp = malloc(n * sizeof *output.realp);
output.imagp = malloc(n * sizeof *output.imagp);

FFTSetup setup = vDSP_create_fftsetup(log2n, FFT_RADIX2);
```

fast Fourier transform Setup

```
#include <Accelerate/Accelerate.h>
#include <stdlib.h>

const int log2n = 10;
const int n = 1 << log2n;

DSPSplitComplex input, output;
input.realp = malloc(n * sizeof *input.realp);
input.imagp = malloc(n * sizeof *input.imagp);
output.realp = malloc(n * sizeof *output.realp);
output.imagp = malloc(n * sizeof *output.imagp);

FFTSetup setup = vDSP_create_fftsetup(log2n, FFT_RADIX2);
```

vDSP Example

Forward fast Fourier transform

```
#include <Accelerate/Accelerate.h>
#include <stdlib.h>

FFTSetup setup = vDSP_create_fftsetup(log2n, FFT_RADIX2);

vDSP_fft_zop(setup, &input, 1, &output, 1, log2n, FFT_FORWARD);
```

vDSP Example

Inverse fast Fourier transform

```
#include <Accelerate/Accelerate.h>
#include <stdlib.h>

FFTSetup setup = vDSP_create_fftsetup(log2n, FFT_RADIX2);
vDSP_fft_zop(setup, &input, 1, &output, 1, log2n, FFT_FORWARD);

vDSP_fft_zip(setup, &output, 1, log2n, FFT_INVERSE);

const float scale = 1.f / n;
vDSP_vsmul(output.realp, 1, &scale, output.realp, 1, n);
vDSP_vsmul(output.imagp, 1, &scale, output.imagp, 1, n);

vDSP_destroy_fftsetup(setup);
```

vDSP Example

Inverse fast Fourier transform

```
#include <Accelerate/Accelerate.h>
#include <stdlib.h>

FFTSetup setup = vDSP_create_fftsetup(log2n, FFT_RADIX2);
vDSP_fft_zop(setup, &input, 1, &output, 1, log2n, FFT_FORWARD);

vDSP_fft_zip(setup, &output, 1, log2n, FFT_INVERSE);

const float scale = 1.f / n;
vDSP_vsmul(output.realp, 1, &scale, output.realp, 1, n);
vDSP_vsmul(output.imagp, 1, &scale, output.imagp, 1, n);

vDSP_destroy_fftsetup(setup);
```

vDSP Example

Inverse fast Fourier transform

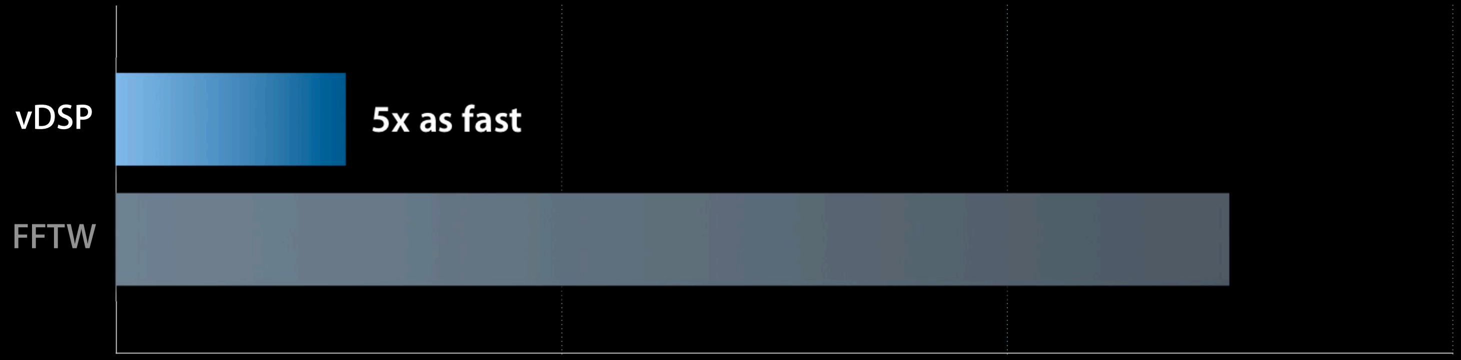
```
#include <Accelerate/Accelerate.h>
#include <stdlib.h>

FFTSetup setup = vDSP_create_fftsetup(log2n, FFT_RADIX2);
vDSP_fft_zop(setup, &input, 1, &output, 1, log2n, FFT_FORWARD);

vDSP_fft_zip(setup, &output, 1, log2n, FFT_INVERSE);

const float scale = 1.f / n;
vDSP_vsmul(output.realp, 1, &scale, output.realp, 1, n);
vDSP_vsmul(output.imagp, 1, &scale, output.imagp, 1, n);

vDSP_destroy_fftsetup(setup);
```

Performance

- FFTW
  - “Fastest Fourier Transform in the West“
  - The best commercial portable FFT library
  - http://www.fftw.org/

Execution Time

Length 1024 single precision FFT on ARMv7
(smaller bars are better)



LAPACK

Linear Algebra PACKage

What’s Available

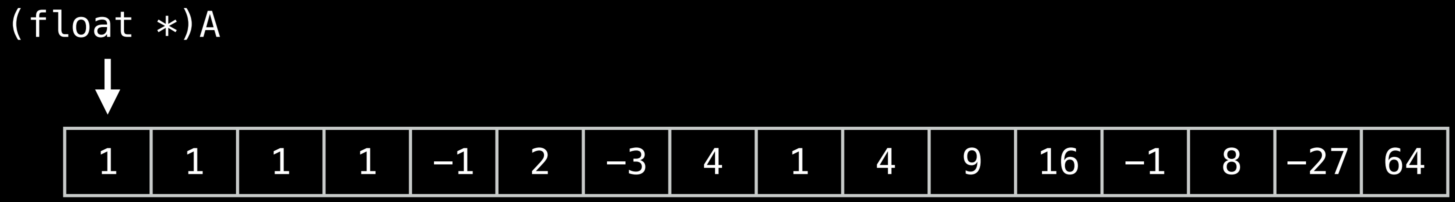
- High level linear algebra
- Solve linear systems
- Matrix factorizations
- Eigenvalues and Eigenvectors

Data Types

- Single and double precision
- Real and complex
- Lots of special matrix types
    - Symmetric, Triangular, Banded, Hermitian, …

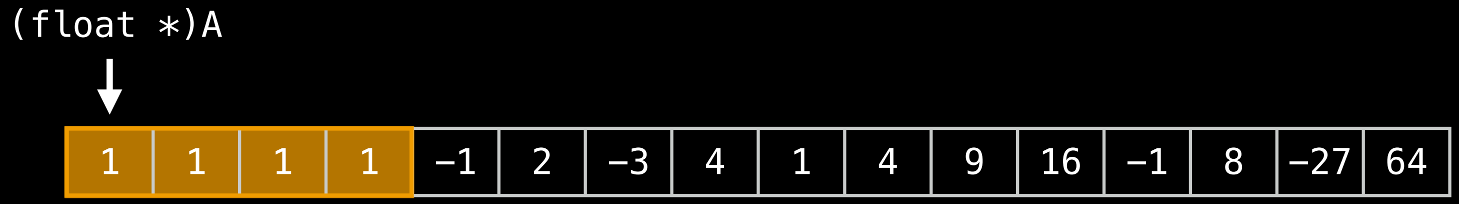
Column Major Order

$$A = \begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 2 & 4 & 8 \\ 1 & -3 & 9 & -27 \\ 1 & 4 & 16 & 64 \end{bmatrix}$$



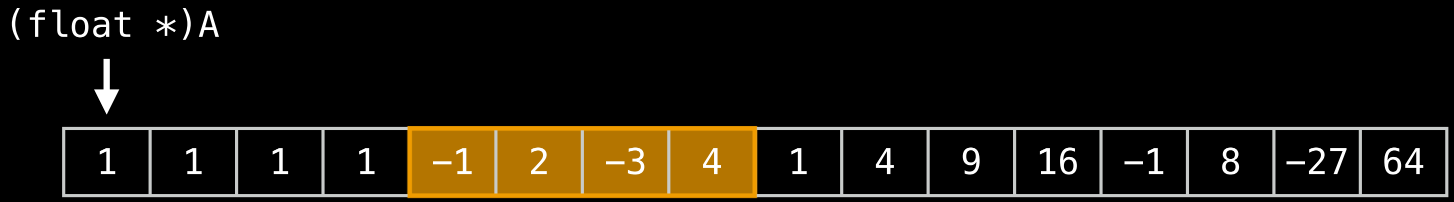
Column Major Order

$$A = \begin{bmatrix} \boxed{1} & -1 & 1 & -1 \\ \boxed{1} & 2 & 4 & 8 \\ \boxed{1} & -3 & 9 & -27 \\ \boxed{1} & 4 & 16 & 64 \end{bmatrix}$$



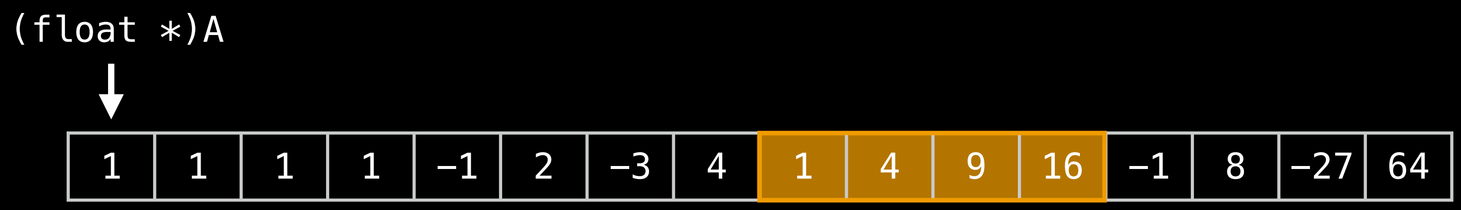
Column Major Order

$$A = \begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 2 & 4 & 8 \\ 1 & -3 & 9 & -27 \\ 1 & 4 & 16 & 64 \end{bmatrix}$$



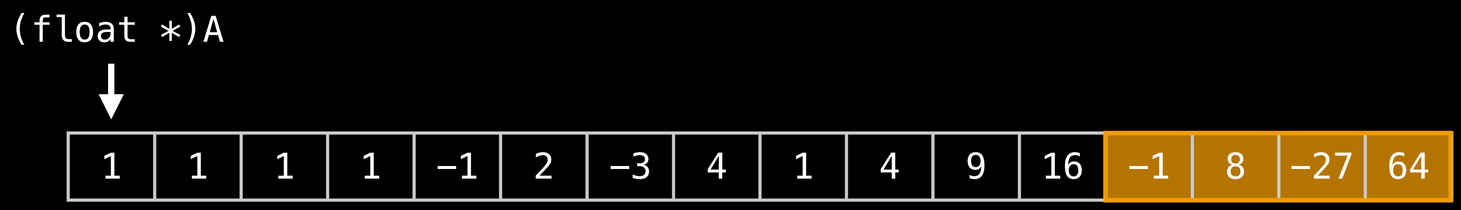
Column Major Order

$$A = \begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 2 & 4 & 8 \\ 1 & -3 & 9 & -27 \\ 1 & 4 & 16 & 64 \end{bmatrix}$$



Column Major Order

$$A = \begin{bmatrix} 1 & -1 & 1 & \boxed{-1} \\ 1 & 2 & 4 & \boxed{8} \\ 1 & -3 & 9 & \boxed{-27} \\ 1 & 4 & 16 & \boxed{64} \end{bmatrix}$$



LAPACK Example

Solving a system of linear equations

$$A = \begin{bmatrix} 1 & 0 & 0 & 1 \\ -1 & 1 & 0 & 1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & 1 \end{bmatrix} \qquad b = \begin{bmatrix} 5 \\ 5 \\ 4 \\ -2 \end{bmatrix}$$

$$Ax = b$$

LAPACK Example

Solving a system of linear equations

```
__CLPK_integer n = 4;
float A[n][n];

for (int i=0; i<n; ++i) {
    for (int j=0; j<n; ++j) {
        if (j < i)
            A[j][i] = -1.f;
        else if (j == i)
            A[j][i] = 1.f;
        else if (j == n-1)
            A[j][i] = 1.f;
        else
            A[j][i] = 0.f;
    }
}
```

$$A = \begin{bmatrix} 1 & 0 & 0 & 1 \\ -1 & 1 & 0 & 1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & 1 \end{bmatrix}$$

LAPACK Example

Solving a system of linear equations

```
__CLPK_integer n = 4;
float A[n][n];

for (int i=0; i<n; ++i) {
    for (int j=0; j<n; ++j) {
        if (j < i)
            A[j][i] = -1.f;
        else if (j == i)
            A[j][i] = 1.f;
        else if (j == n-1)
            A[j][i] = 1.f;
        else
            A[j][i] = 0.f;
    }
}
```

$$A = \begin{bmatrix} 1 & 0 & 0 & 1 \\ -1 & 1 & 0 & 1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & 1 \end{bmatrix}$$

LAPACK Example

Solving a system of linear equations

```
__CLPK_integer n = 4;
float A[n][n];

for (int i=0; i<n; ++i) {
    for (int j=0; j<n; ++j) {
        if (j < i)
            A[j][i] = -1.f;
        else if (j == i)
            A[j][i] = 1.f;
        else if (j == n-1)
            A[j][i] = 1.f;
        else
            A[j][i] = 0.f;
    }
}
```

$$A = \begin{bmatrix} 1 & 0 & 0 & 1 \\ -1 & 1 & 0 & 1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & 1 \end{bmatrix}$$

LAPACK Example

Solving a system of linear equations

```
__CLPK_integer n = 4;
float A[n][n];

for (int i=0; i<n; ++i) {
    for (int j=0; j<n; ++j) {
        if (j < i)
            A[j][i] = -1.f;
        else if (j == i)
            A[j][i] = 1.f;
        else if (j == n-1)
            A[j][i] = 1.f;
        else
            A[j][i] = 0.f;
    }
}
```

$$A = \begin{bmatrix} 1 & 0 & 0 & 1 \\ -1 & 1 & 0 & 1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & 1 \end{bmatrix}$$

LAPACK Example

Solving a system of linear equations

```
float b[n];
b[0] = 5.f;
b[1] = 5.f;
b[2] = 4.f;
b[3] = -2.f;

__CLPK_integer nrhs = 1;
__CLPK_integer ipiv[n];
__CLPK_integer info;

sgesv_(&n, &nrhs, A, &n, ipiv, b, &n, &info);

printf("The solution is: (%f, %f, %f, %f)\n", b[0],b[1],b[2],b[3]);
```

LAPACK Example

Solving a system of linear equations

```
float b[n];
b[0] = 5.f;
b[1] = 5.f;
b[2] = 4.f;
b[3] = -2.f;

__CLPK_integer nrhs = 1;
__CLPK_integer ipiv[n];
__CLPK_integer info;

sgesv_(&n, &nrhs, A, &n, ipiv, b, &n, &info);

printf("The solution is: (%f, %f, %f, %f)\n", b[0],b[1],b[2],b[3]);
```

LAPACK Example

Solving a system of linear equations

```
float b[n];
b[0] = 5.f;
b[1] = 5.f;
b[2] = 4.f;
b[3] = -2.f;

__CLPK_integer nrhs = 1;
__CLPK_integer ipiv[n];
__CLPK_integer info;

sgesv_(&n, &nrhs, A, &n, ipiv, b, &n, &info);

printf("The solution is: (%f, %f, %f, %f)\n", b[0],b[1],b[2],b[3]);
```

LAPACK Example

Solving a system of linear equations

```
canon$ gcc lapack.c -framework Accelerate -std=c99 -olapackExample
canon$
```

LAPACK Example

Solving a system of linear equations

```
canon$ gcc lapack.c -framework Accelerate -std=c99 -olapackExample
canon$ ./lapackExample
The solution is: (1.000000, 2.000000, 3.000000, 4.000000)
canon$
```

BLAS

Basic Linear Algebra Subroutines

What’s Available

- Low level linear algebra
- Vector
  - Dot product, scalar product, vector sum
- Matrix-vector
  - Matrix-vector product, outer product
- Matrix-matrix
  - Matrix multiply

Data Types

- Single and double precision
- Real and complex
- Multiple data layouts
    - Row and column major
    - Dense, banded, triangular, …
    - Transpose, conjugate transpose

BLAS Example

Matrix multiply

```
#include <Accelerate/Accelerate.h>

float A[2][2] =
    {{1.f, 1.f},
     {0.f, 1.f}};
float B[2][2] =
    {{1.f, 2.f},
     {3.f, 4.f}};
float C[2][2];

// Put the result of A*B in the matrix C.
cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            2, 2, 2, 1.f, A, 2, B, 2, 0.f, C, 2);
```

Best Practices for Accelerate

Reasons to Use Accelerate

- Speed
- Functionality
- Ease of use
- Architecture independence

Design Tradeoffs in Accelerate

- Make the common use cases fast
- Support many use cases

Cache Awareness

Buffer size

- Excessively small buffers have more call overhead
- Excessively large buffers don’t benefit from cache
- A working set of about 32 KiB is generally just right

Use Contiguous Arrays

- Many functions support strided access
- Don’t use non-unit strides unless you need to
- Try to have the first function you call convert to unit strides

Architectural Awareness

- Make sure to build your binary fat, for both ARMv6 and ARMv7
- Prefer floats to doubles, when possible
  - Float is a little bit faster than double on ARMv6
  - Float is enormously faster than double on ARMv7

More Information

Paul Danbold
Technology Evangelist
danbold@apple.com

Apple Developer Forums
http://devforums.apple.com

More Information

Documentation
vDSP Reference Collection
http://developer.apple.com/mac/library/documentation/Performance/Reference/vDSP_Reference_Collection/

LAPACK Users' Guide
http://www.netlib.org/lapack/lug/

BLAS Quick Reference
http://www.netlib.org/lapack/lug/node145.html

Headers
/System/Library/Frameworks/Accelerate.framework/Headers/Accelerate.h

Q&A

WWDC10

[redacted]