Accelerate Your Code

with the Accelerate framework

Jeff Biggus — HyperJeff, Inc — @hyperjeff

Who am I?

- Programmer from a science background, not from CS
- Creator of various OS X resources over the years
- Someone who thought the biggest announcement about iOS 4 was the inclusion of Accelerate.framework
- CocoaLit.com
- http://blog.hyperjeff.net/blasLookup.pdf

Accelerate

- On OS X since Jaguar (10.2) — Originally: vecLib
- Currently on OS X: Made up of 8 libraries
  - some have evolved over time since the 1960s
  - some from the early 2000's
- Slow adoption among general programmers

Accelerate

- Pure C set of APIs specially tuned in assembly to take advantage of hardware (CPU*)
- Operates on Arrays and Matrices of Floats, Complex, Doubles, Double Complex
- Very large collection of functions
- Works best on large amounts of data at once
- Can actually provide clean readable code

Accelerate

- Common use cases:
    - Pure math algorithms / equation solving / modeling
        - Science, Finance, Database, Etc
    - Image processing
    - Audio processing

Accelerate

- Code re-use advantages:
  - Apple is keeping up the libraries, not you
  - Your code stays the same
  - They are optimizing it for the CPUs

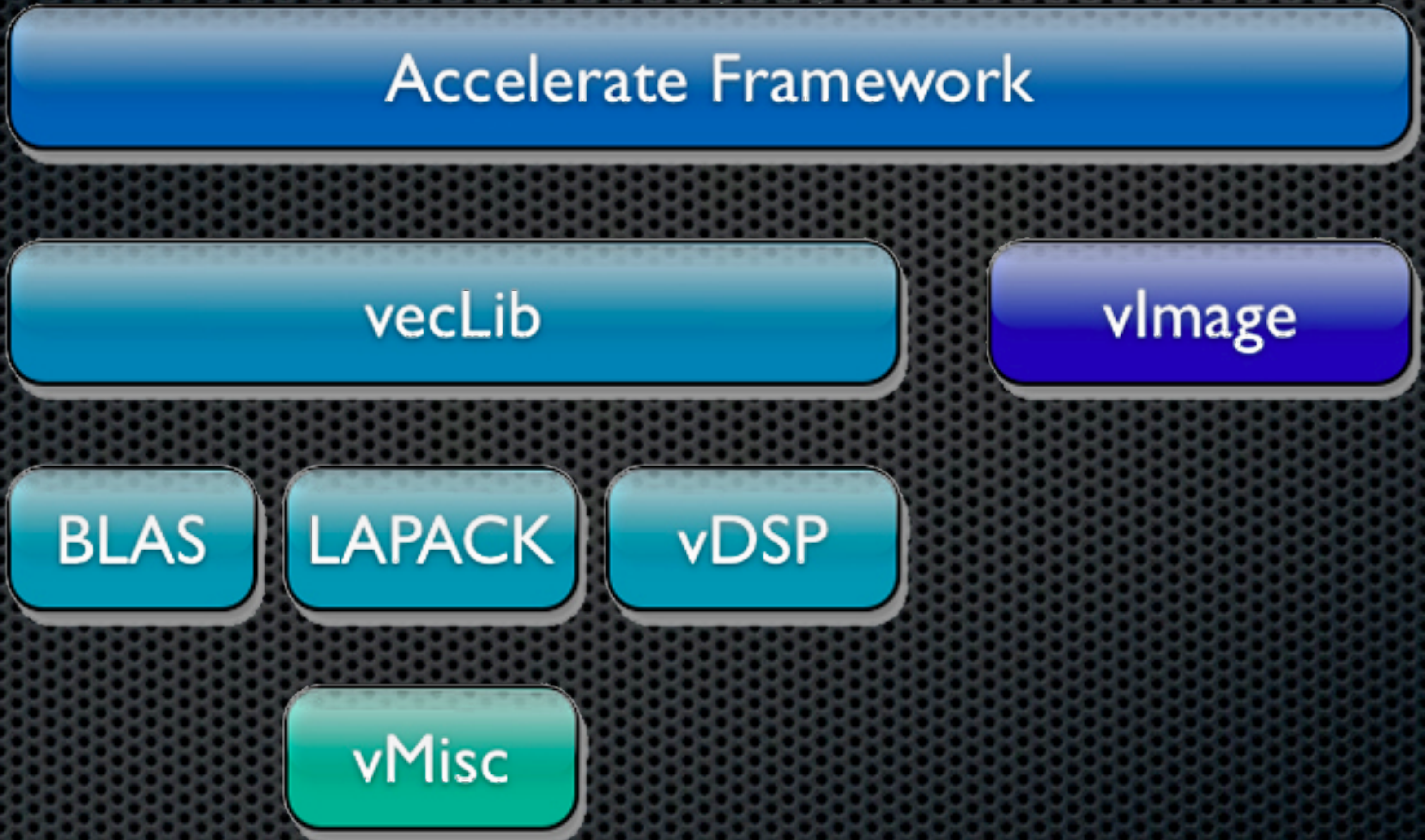
Accelerate

- Fast, if the problem is matched to routines available
- Very fast, if the problem is very well matched
- Energy-efficient
- Used in (only) 27* GitHub projects

Accelerate

- iOS 4: The 3 core libraries added (á là Jaguar)
  - BLAS, LAPACK, vDSP
- iOS 5: 2 more libraries (á là Tiger)
  - vImage, vForce

Accelerate



(vForce)

Accelerate

vImage	218 functions
vDSP	401 functions
vForce	80 functions
LAPACK	1471 functions
BLAS	196 functions

2366 functions

ARM v6, v7

- v6: 1st/2nd gen iPhones / iPod Touches
  - 16 integers registers
  - hardware floating point registers: 32 float, 16 double
- v7: everything since
  - faster, L2 cache, can exec 2 instructures per tic
  - “NEON” SIMD unit

ARM v7 : NEON

- SIMD unit w/ 16 128-bit vector registers
  - processes multiple ints or floats simultaneously
- doubles use standard VFP á là v6…
  - A5 chips hardware-accelerate double calculations
- Less power hungry

A note on math-avoidance

- Don’t fear the math (most of the time)
- n-element vectors
  - Not doing n-dimensional algebra
  - ex: **a** * **b**, really just meaning a[i] * b[i]
  - ex: **a** + **b**, really just a[i] + b[i]
- That said… lots of crazy math possibilities if you need it

vImage

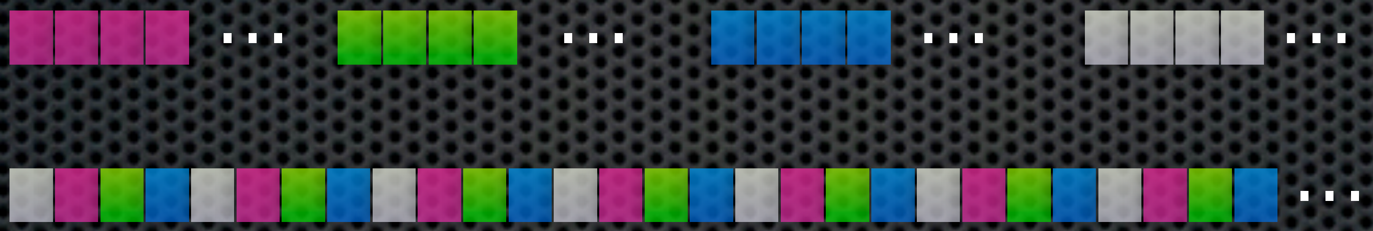
- Aimed at following scenarios:
  - Processing large or high-res images
  - Repeating several operations on an image
  - Real-time image processing
- Otherwise use Core Image for regular images

vImage

- Shopping for image effects
- vImage function_ format
  - ex: vImageVerticalReflect_ARGBFFFF(…)
- Essentially ~58 functions (vs 218)
- Lots of conversion functions between vImage formats
- vImage_Error

vImage

- format
  - monochromatic “planar” vs interleaved (ARGB)



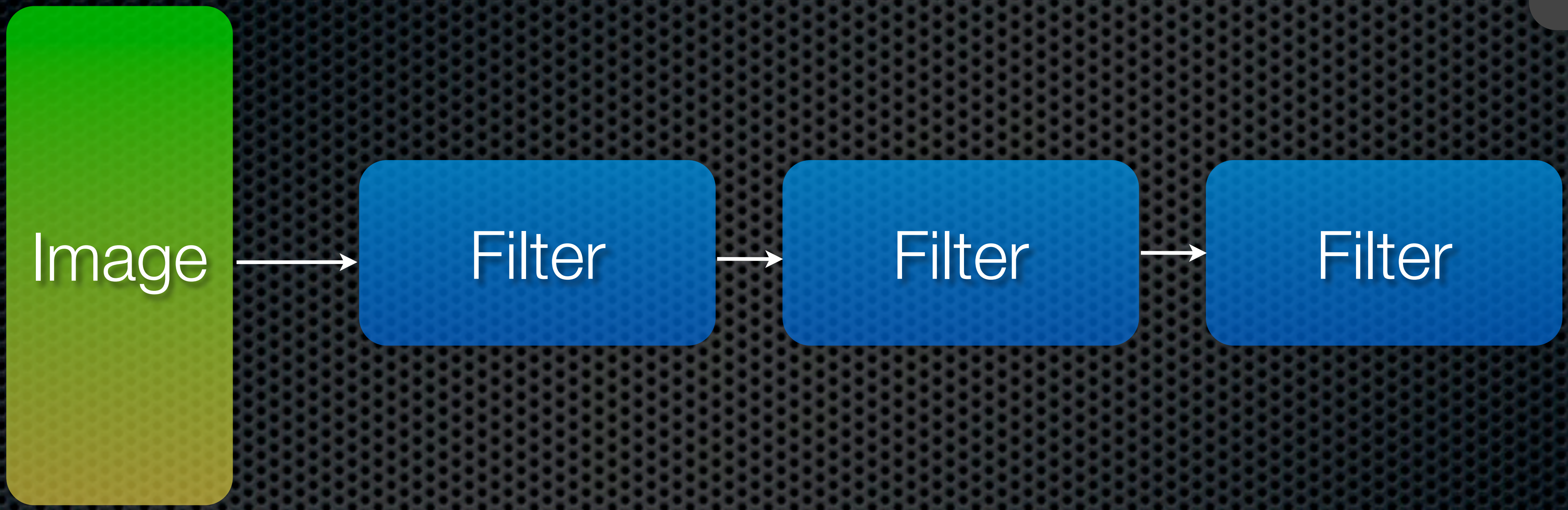
  - 8-bit vs 32 (unsigned chars vs floats)
  - …_Planar8, …_PlanarF
  - …_ARGB8888, …_ARGBFFFF
  - planar can be significantly faster

vImage

```
typedef struct vImage_Buffer
{
    void *data;
    vImagePixelCount  height;
    vImagePixelCount  width;
    size_t rowBytes;
}
vImage_Buffer;
```

- “Caution: except where otherwise documented, most vImage functions do not work correctly in place”

vImage



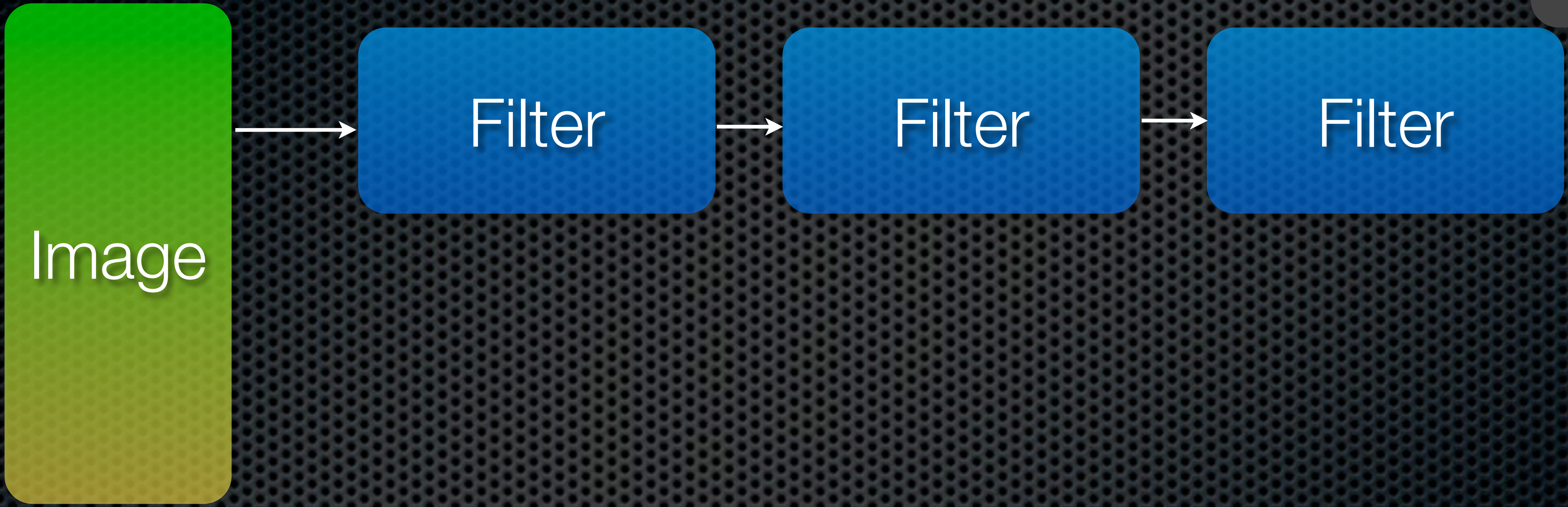
Image

Filter

Filter

Filter

vImage



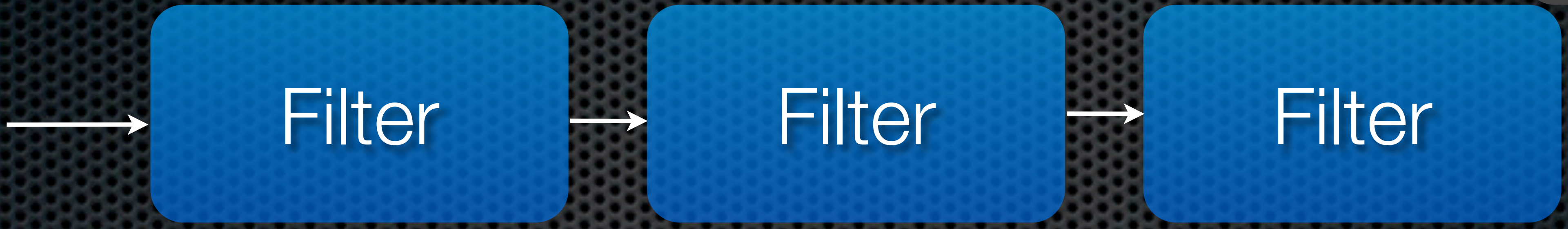
Image

Filter

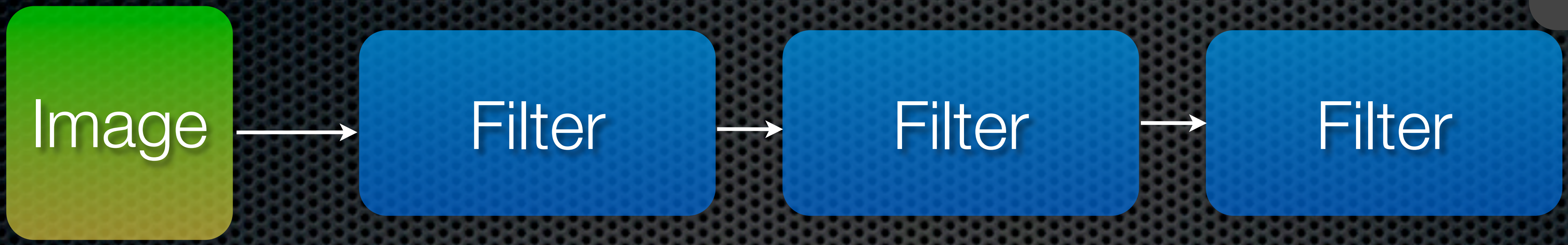
Filter

Filter

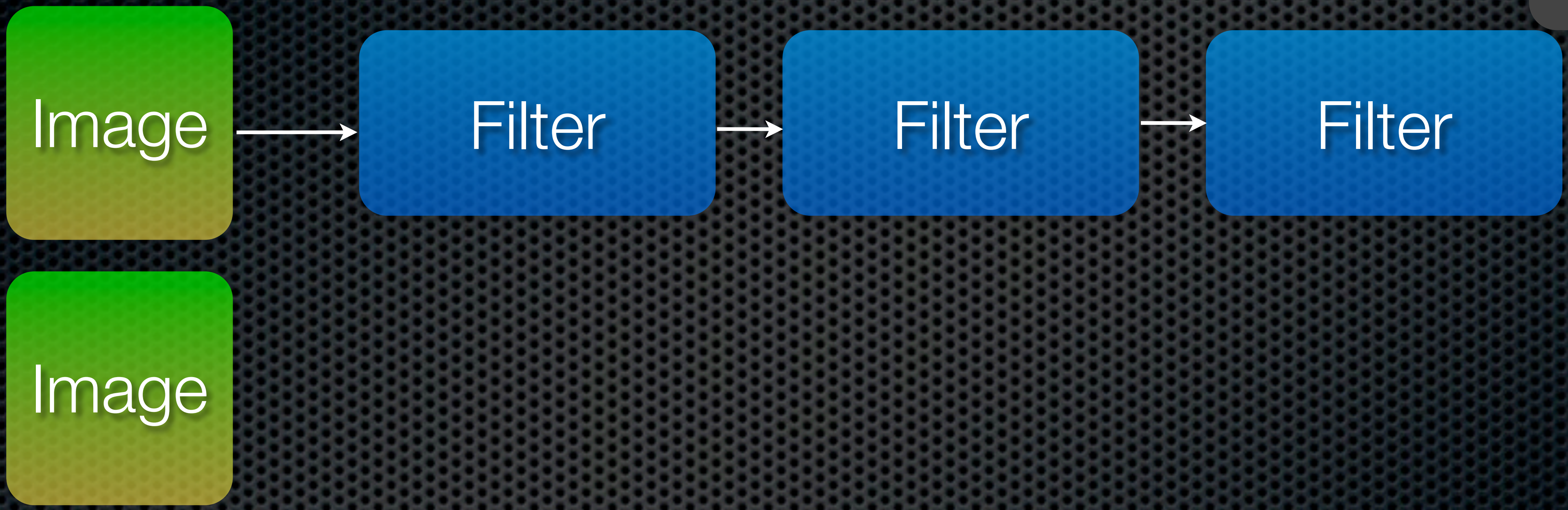
vImage



vImage



vImage



vImage

vDSP

vForce

LAPACK

BLAS

vImage

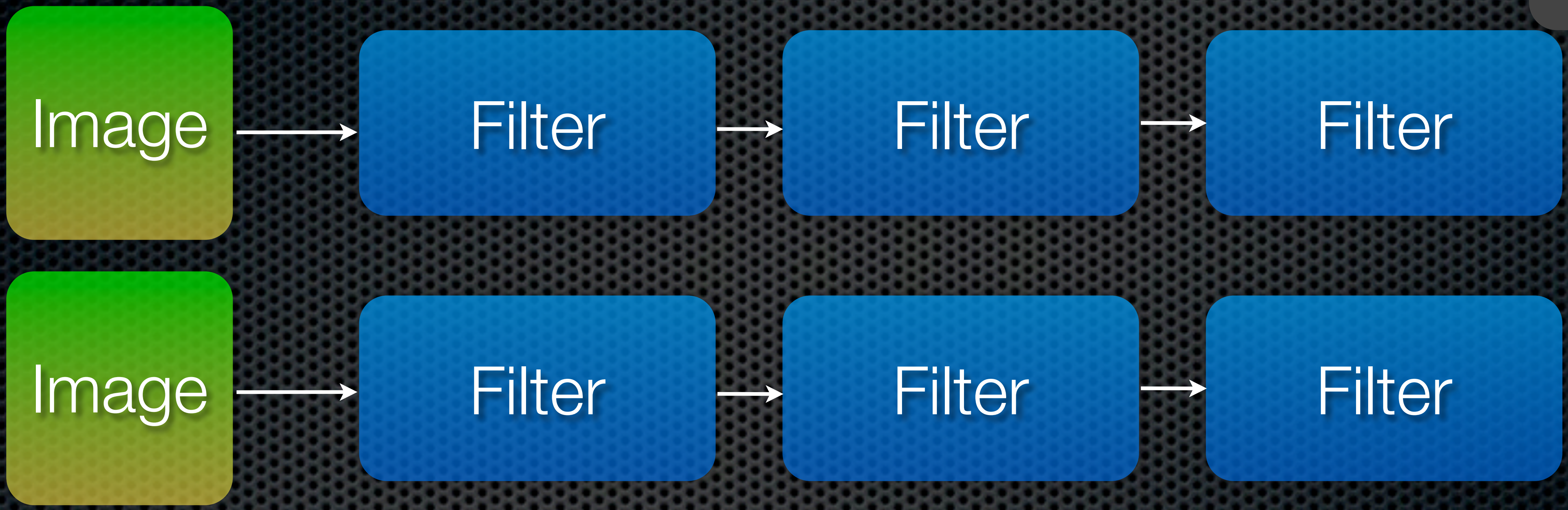


Image → Filter → Filter → Filter

Image → Filter → Filter → Filter

vImage

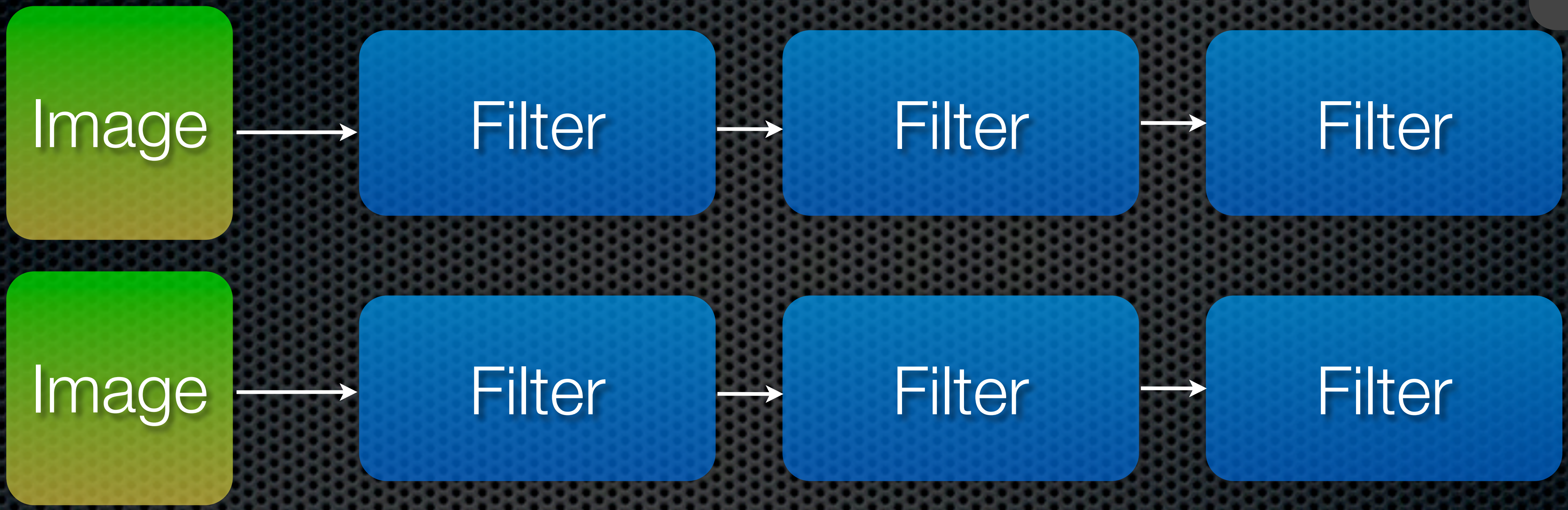


Image → Filter → Filter → Filter

Image → Filter → Filter → Filter

- “All vImage functions are thread safe and may be called reentrantly.”
- All threading done via GCD & can be turned off

vImage

- Pixel transformations
- Expansion, Contraction
- Rotation, Scaling, Warping, Reflection, Shearing
- Convolution: Smoothing, Sharpening
- Alpha compositing
- Colorspace manipulations

vDSP

- Digital Signal Processing
- Audio, image, but really whatever
- Fast Fourier Transforms (1-d, 2-d)
- Vector → Scalar / Vector
- Has its own struct for complex numbers (2 kinds, even)

vImage

vDSP

vForce

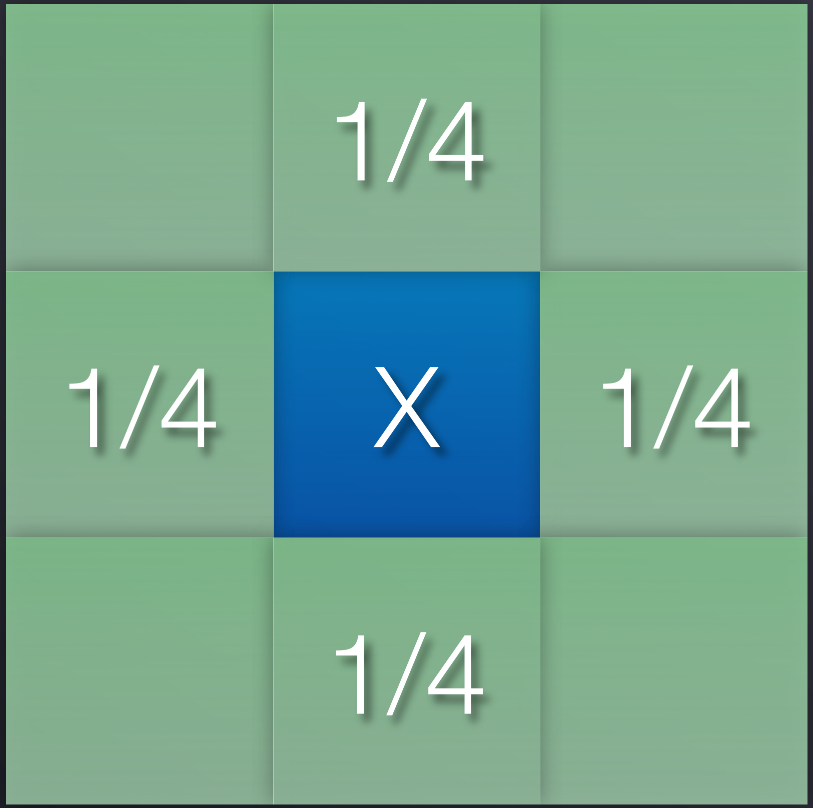
LAPACK

BLAS

	1/4	
1/4	X	1/4
	1/4	

	1/4	
1/4	X	1/4
	1/4	

X = ave(sides)



X = ave(sides)

```
for (int i=0; i<GRID_SIZE; i++)
    for (int j=0; j<GRID_SIZE; j++)
        result[i*GRID_SIZE+j] = 0.25 * (
            grid[(i+1)*GRID_SIZE + j] +
            grid[(i-1)*GRID_SIZE + j] +
            grid[i*GRID_SIZE   + j-1] +
            grid[i*GRID_SIZE   + j+1]
        );
```

vImage

vDSP

vForce

LAPACK

BLAS

	1/4	
1/4	X	1/4
	1/4	

```
for (int i=0; i<GRID_SIZE; i++)
    for (int j=0; j<GRID_SIZE; j++)
        result[i*GRID_SIZE+j] = 0.25 * (
            grid[(i+1)*GRID_SIZE + j] +
            grid[(i-1)*GRID_SIZE + j] +
            grid[i*GRID_SIZE   + j-1] +
            grid[i*GRID_SIZE   + j+1]
        );
```

	1/4	
1/4	X	1/4
	1/4	

vImage

vDSP

vForce

LAPACK

BLAS

```
for (int i=0; i<GRID_SIZE; i++)
    for (int j=0; j<GRID_SIZE; j++)
        result[i*GRID_SIZE+j] = 0.25 * (
            grid[(i+1)*GRID_SIZE + j] +
            grid[(i-1)*GRID_SIZE + j] +
            grid[i*GRID_SIZE   + j-1] +
            grid[i*GRID_SIZE   + j+1]
        );
```

vImage

vDSP

vForce

LAPACK

BLAS

	1/4	
1/4	X	1/4
	1/4	

```
float filter[] = {
    0.0,  0.25, 0.0,
    0.25, 0.0,  0.25,
    0.0,  0.25, 0.0
};

...

if (USE_vDSP) {

    vDSP_f3x3( grid, GRID_SIZE, GRID_SIZE, filter, result );

}
else {

    for (int i=0; i<GRID_SIZE; i++)
        for (int j=0; j<GRID_SIZE; j++)
            result[i*GRID_SIZE+j] = 0.25 * (
                grid[(i+1)*GRID_SIZE + j] +
                grid[(i-1)*GRID_SIZE + j] +
                grid[i*GRID_SIZE   + j-1] +
                grid[i*GRID_SIZE   + j+1]
            );
}
```

vImage

vDSP

vForce

LAPACK

BLAS

	α	
α	2-4α	α
	α	

	α	
α	2-4α	α
	α	

vImage

vDSP

vForce

LAPACK

BLAS

```
for (int i=0; i<GRID_SIZE; i++)
    for (int j=0; j<GRID_SIZE; j++)
        result[i*GRID_SIZE+j] = grid[i*GRID_SIZE + j] +
            grid[(i+1)*GRID_SIZE + j] + grid[(i-1)*GRID_SIZE + j] +
            grid[i*GRID_SIZE   + j-1] + grid[i*GRID_SIZE   + j+1];

for (int i=0; i<GRID_SIZE; i++)
    for (int j=0; j<GRID_SIZE; j++)
        result[i*GRID_SIZE + j] -= oldGrid[i*GRID_SIZE + j];

memcpy( oldGrid, grid, GRID_AREA * sizeof( float ) );
memcpy( grid, result, GRID_AREA * sizeof( float ) );

for (int i=0; i<GRID_SIZE; i++)
    for (int j=0; j<GRID_SIZE; j++)
        points[3 * (i * GRID_SIZE + j) + 1] = grid[i*GRID_SIZE + j];
```

```
if (USE_vDSP) {

    float filter[] = {
        0.0,         alpha,          0.0,
        alpha,   2.0 - 4.0*alpha, alpha,
        0.0,         alpha,          0.0
    };

    vDSP_f3x3( grid, GRID_SIZE, GRID_SIZE, filter, result );
    cblas_saxpy( GRID_AREA, -1.0, oldGrid, 1, result, 1 );

    memcpy( oldGrid, grid, GRID_AREA * sizeof( float ) );
    memcpy( grid,  result, GRID_AREA * sizeof( float ) );

    cblas_scopy( GRID_AREA, grid, 1, &points[1], 3 );
}
else {

    for (int i=0; i<GRID_SIZE; i++)
        for (int j=0; j<GRID_SIZE; j++)
            result[i*GRID_SIZE+j] = grid[i*GRID_SIZE + j] +
                grid[(i+1)*GRID_SIZE + j] + grid[(i-1)*GRID_SIZE + j] +
                grid[i*GRID_SIZE   + j-1] + grid[i*GRID_SIZE   + j+1];

    for (int i=0; i<GRID_SIZE; i++)
        for (int j=0; j<GRID_SIZE; j++)
            result[i*GRID_SIZE + j] -= oldGrid[i*GRID_SIZE + j];

    memcpy( oldGrid, grid, GRID_AREA * sizeof( float ) );
    memcpy( grid, result, GRID_AREA * sizeof( float ) );

    for (int i=0; i<GRID_SIZE; i++)
        for (int j=0; j<GRID_SIZE; j++)
            points[3 * (i * GRID_SIZE + j) + 1] = grid[i*GRID_SIZE + j];
}
```

vImage

vDSP

vForce

LAPACK

BLAS

WAVE DEMO

AUDIO UNIT DEMO

vImage

vDSP

vForce

LAPACK

BLAS

```
for (int i=0; i<5; i++) {     // what I want: z = z*z + c

   vDSP_zvmul( &z, 1, &z, 1, &z, 1, size, 0 ); // z = z * z
   vDSP_zvadd( &z, 1, &c, 1, &z, 1, size );    // z = z + c
}

vDSP_zvmags( &z, 1, m, 1, size );

vDSP_vclip( m, 1, &low, &high, m, 1, size );

vDSP_vsdiv( m, 1, &high, m, 1, size );
```

vForce

- Just pure mathy goodness

vForce

- v v (function name) f*
- ~30 functions at the core (vs 80)

```
                                           ceil
acos      cos                              copysign
          cosisin                          div
          cospi                            exp(2,m1)
asin      sin                              fabs
          sincos                           floor
          sinpi                            fmod
atan(2)   tan                              int
          tanpi                            log(10,1p,2,b)
                                           extafter
acosh     cosh                             pow
asinh     sinh                             rec
atanh     tanh                             remainedr
                                           (r)sqrt
```

vForce

- v v (function name) f*
- ~30 functions at the core (vs 80)

```
                                           ceil
acos      cos                              copysign
          cosisin                          div
          cospi                            exp(2,m1)
asin      sin                              fabs
          sincos                           floor
          sinpi                            fmod
atan(2)   tan                              int
          tanpi                            log(10,1p,2,b)
                                           extafter
acosh     cosh                             pow
asinh     sinh                             rec
atanh     tanh                             remainedr
                                           (r)sqrt
```

vForce

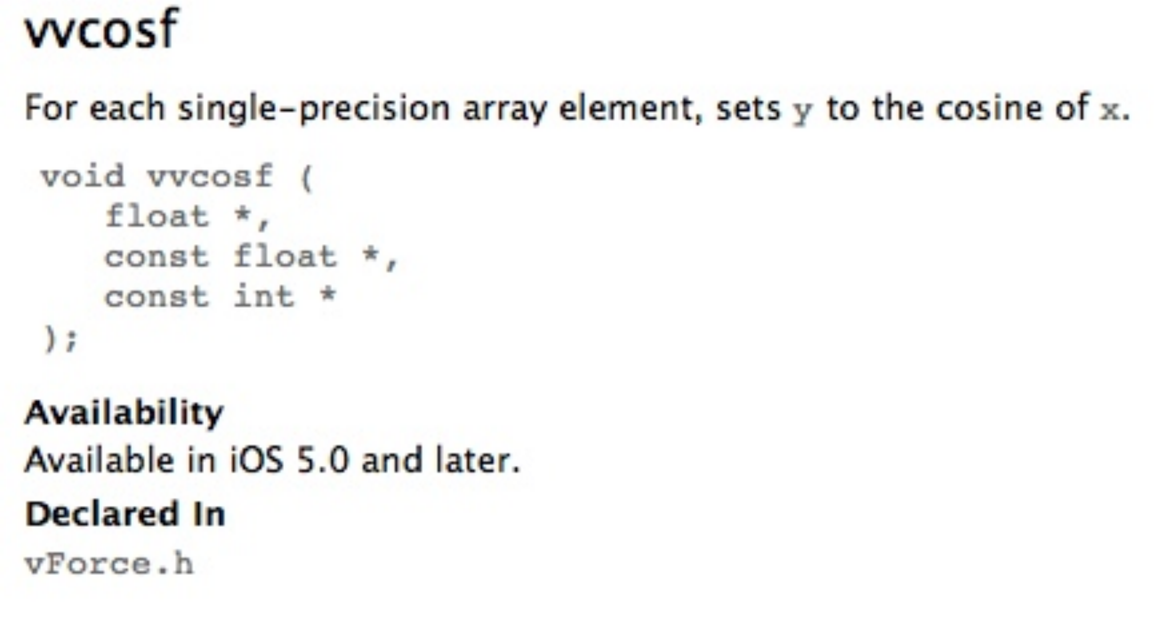
vImage

vDSP

vForce

LAPACK

BLAS



vvcosf

For each single-precision array element, sets y to the cosine of x.

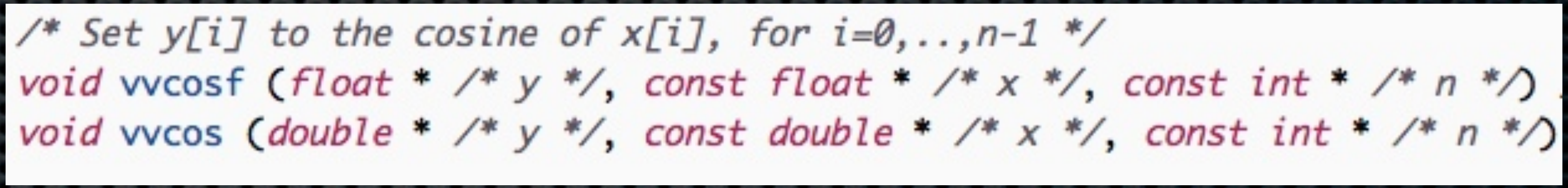
```
void vvcosf (
    float *,
    const float *,
    const int *
);
```

Availability

Available in iOS 5.0 and later.

Declared In

vForce.h



```
/* Set y[i] to the cosine of x[i], for i=0,..,n-1 */
void vvcosf (float * /* y */, const float * /* x */, const int * /* n */)
void vvcos (double * /* y */, const double * /* x */, const int * /* n */)
```

vImage

vDSP

vForce

LAPACK

BLAS

vForce

y, x, n ☚ ?

$y_i \leftarrow \square(x_i) \mid 0 \leq i < n$

z, y, x, n

$z_i \leftarrow \square(y_i, x_i) \mid 0 \leq i < n$

sometimes C, x, n

LAPACK

- Linear Algebra PACKage
- Primarily for *solving* systems of equations
- Compiled from Fortran (CLAPACK),
  so slightly odder function call issues

LAPACK

- Implications of Fortran underbelly
  - Matrices are all column-major-ordered
  - All functions are postfixed with _
  - *All* arguments sent to functions are references only

vImage

vDSP

vForce

LAPACK

BLAS

```
float A[] = {
   3., 1., 3.,
   1., 5., 9.,
   2., 6., 5.
};

float b[] = { -1., 3., -3. };

int output, pivot[3], numberOfEquations = 3,
    bSolutionCount = 1,
    leadingDimA = 3, leadingDimB = 3;

sgesv_( &numberOfEquations, &bSolutionCount,
        A, &leadingDimA, pivot, b, &leadingDimB, &output );
```

BLAS

- Basic Linear Algebra Subprograms
- Vector → Vector operations BLAS 1, $O(n)$
- Matrix / Vector → Vector BLAS 2, $O(n^2)$
- Matrix / Vector → Matrix BLAS 3, $O(n^3)$
- Dense matrix routines (along with LAPACK)

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

int main(int argc, const char *argv[]) {

    float x[] = { 1., 2., 3. };
    float y[] = { 3., 4., 5. };

    //      y =     10   x  +  y

    cblas_saxpy( 3, 10., x, 1, y, 1 );

    printf( "\n y = { %2.f, %2.f, %2.f}\n", y[0], y[1], y[2] );

    return 0;
}
```

vImage

vDSP

vForce

LAPACK

BLAS

vImage
vDSP
vForce
LAPACK
BLAS

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

int main(int argc, const char *argv[]) {

   float x[] = { 1., 2., 3. };
   float y[] = { 3., 4., 5. };

   //      y =     10   x  +  y

   cblas_saxpy( 3, 10., x, 1, y, 1 );

   printf( "\n y = { %2.f, %2.f, %2.f}\n", y[0], y[1], y[2] );

   return 0;
}
```

vImage

vDSP

vForce

LAPACK

BLAS

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

int main(int argc, const char *argv[]) {

    float x[] = { 1., 2., 3. };
    float y[] = { 3., 4., 5. };

    //      y =      10   x  +  y

    cblas_saxpy( 3, 10., x, 1, y, 1 );

    printf( "\n y = { %2.f, %2.f, %2.f}\n", y[0], y[1], y[2] );

    return 0;
}
```

```
% clang -o cblas_saxpy cblas_saxpy.c -framework Accelerate
% ./cblas_saxpy

 y = { 13, 24, 35 }
```

```
float x[] = { 1., 2., 3. };
float y[] = { 3., 4., 5. };

cblas_saxpy( 3, 10., x, 1, y, 1 );
```

vImage

vDSP

vForce

LAPACK

BLAS

```
float x[] = { 1., 2., 3. };
float y[] = { 3., 4., 5. };

cblas_saxpy( 3, 10., x, 1, y, 1 );
```

vImage

vDSP

vForce

LAPACK

BLAS

```
float x[] = { 1., 2., 3. };
float y[] = { 3., 4., 5. };

cblas_saxpy( 3, 10., x, 1, y, 1 );
```

vImage

vDSP

vForce

LAPACK

BLAS

```
float x[] = { 1., 2., 3. };
float y[] = { 3., 4., 5. };

cblas_saxpy( 3, 10., x, 1, y, 1 );
```

```
float x[] = { 1., 9., 2., 9., 3., 9. };
float y[] = { 3., 9., 4., 9., 5., 9. };

cblas_saxpy( 3, 10., x, 2, y, 2 );
```

vImage

vDSP

vForce

LAPACK

BLAS

```
float x[] = { 1., 2., 3. };
float y[] = { 3., 4., 5. };

cblas_saxpy( 3, 10., x, 1, y, 1 );
```

```
float x[] = { 1., 9., 2., 9., 3., 9. };
float y[] = { 3., 9., 4., 9., 5., 9. };

cblas_saxpy( 3, 10., x, 2, y, 2 );
```

```
float x[] = { 1., 2., 3. };
float y[] = { 3., 4., 5. };

cblas_saxpy( 3, 10., x, 1, y, 1 );
```

```
float x[] = { 1., 9., 2., 9., 3., 9. };
float y[] = { 3., 9., 4., 9., 5., 9. };

cblas_saxpy( 3, 10., x, 2, y, 2 );
```

vImage

vDSP

vForce

LAPACK

BLAS

```
float x[] = { 1., 2., 3. };
float y[] = { 3., 4., 5. };

cblas_saxpy( 3, 10., x, 1, y, 1 );
```

```
float x[] = { 1., 9., 2., 9., 3., 9. };
float y[] = { 3., 9., 4., 9., 5., 9. };

cblas_saxpy( 3, 10., x, 2, y, 2 );
```

```
y:         { 13,  9, 24,  9, 35,  9 }
```

cblas_saxpy

vImage

vDSP

vForce

LAPACK

BLAS

cblas_ saxpy

vImage

vDSP

vForce

LAPACK

BLAS

cblas_ s axpy

cblas_ s axpy

BLAS

cblas_	s	axpy
BLAS	type	

cblas_	s	axpy
BLAS	type	function

vImage

vDSP

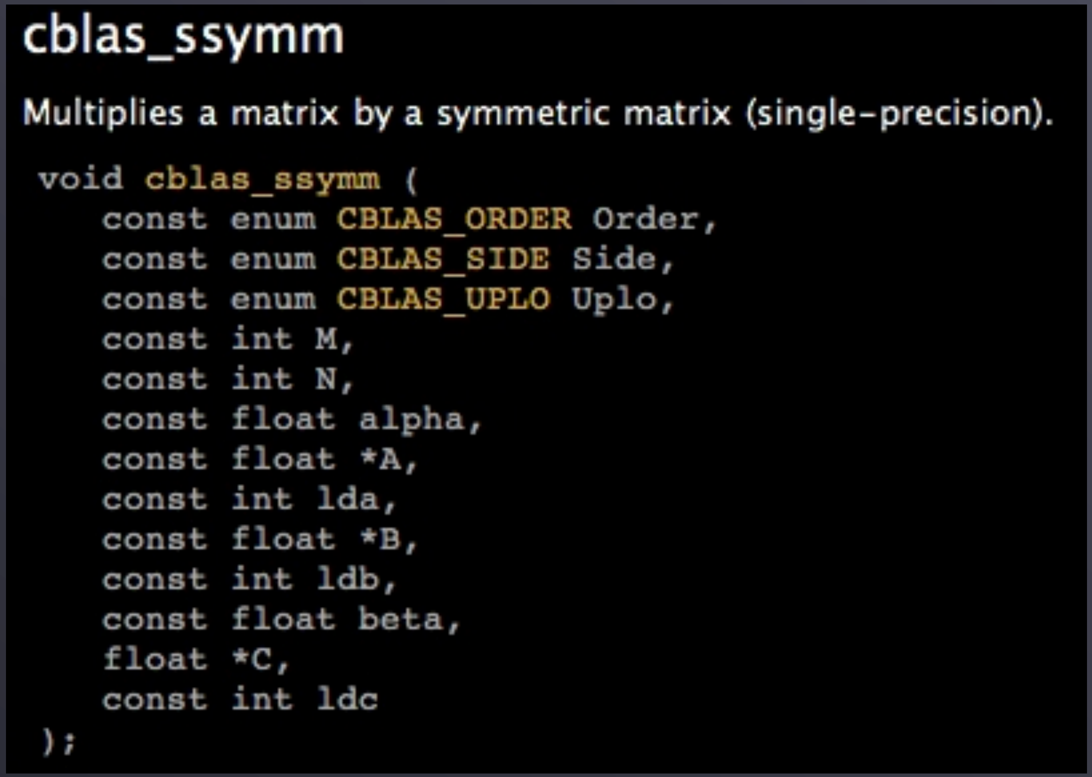
vForce

LAPACK

BLAS

```
double complex u[] = { -3. + 4.*I,  5. + 7.*I };
double complex w[] = {  1. + 2.*I, -1. + 5.*I };

double complex alpha[] = { 10. + 100.*I };

//          w = alpha  u   + w

cblas_zaxpy( 2, alpha, u, 1, w, 1 );
```

```
float a[] = {
   10., 5., 3.,
    5., 4., 2.,
    3., 2., 1.
};

float b[] = {
   1., 2.,
   3., 4.,
   5., 6.
};

float c[9];

cblas_ssymm(
   CblasRowMajor,
   CblasLeft,
   CblasUpper,
   3, 2,
   1., a, 3, b, 2,
   0., c, 2
);
```



cblas_ssymm

Multiplies a matrix by a symmetric matrix (single-precision).

```
void cblas_ssymm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_SIDE Side,
   const enum CBLAS_UPLO Uplo,
   const int M,
   const int N,
   const float alpha,
   const float *A,
   const int lda,
   const float *B,
   const int ldb,
   const float beta,
   float *C,
   const int ldc
);
```

```
C = A B

C:
   |  40  58 |
   |  27  38 |
   |  14  20 |
float x[] = { 1., 2., 3. };

float A[] = {              // upper-triangular _packed_ matrix
   2., 5., 10.,
       1.,  5.,
            3.
};

// x = A x

cblas_stpmv(
   CblasRowMajor, CblasUpper, CblasNoTrans, CblasNonUnit,
   3, A, x, 1
);
```

```
% ./cblas_stpmv

x = { 42, 17, 9 }
```

```
actual matrix: 3 2 1 . . .
               2 3 2 1 . .
               1 2 3 2 1 .
               . 1 2 3 2 1
               . . 1 2 3 2
               . . . 1 2 3
```

```
float bandedMatrix2[] = {
   0., 0., 3., 2., 1.,
   0., 2., 3., 2., 1.,
   1., 2., 3., 2., 1.,
   1., 2., 3., 2., 1.,
   1., 2., 3., 2., 0.,
   1., 2., 3., 0., 0.,
};

float symmetric[] = {
   3., 0., 0., 0., 0., 0.,
   2., 3., 0., 0., 0., 0.,
   1., 2., 3., 0., 0., 0.,
   0., 1., 2., 3., 0., 0.,
   0., 0., 1., 2., 3., 0.,
   0., 0., 0., 1., 2., 3.
};

// CblasLower
```

Accelerate Your Code

Accelerate Performance

Accelerate Performance

- Not always the fastest solution

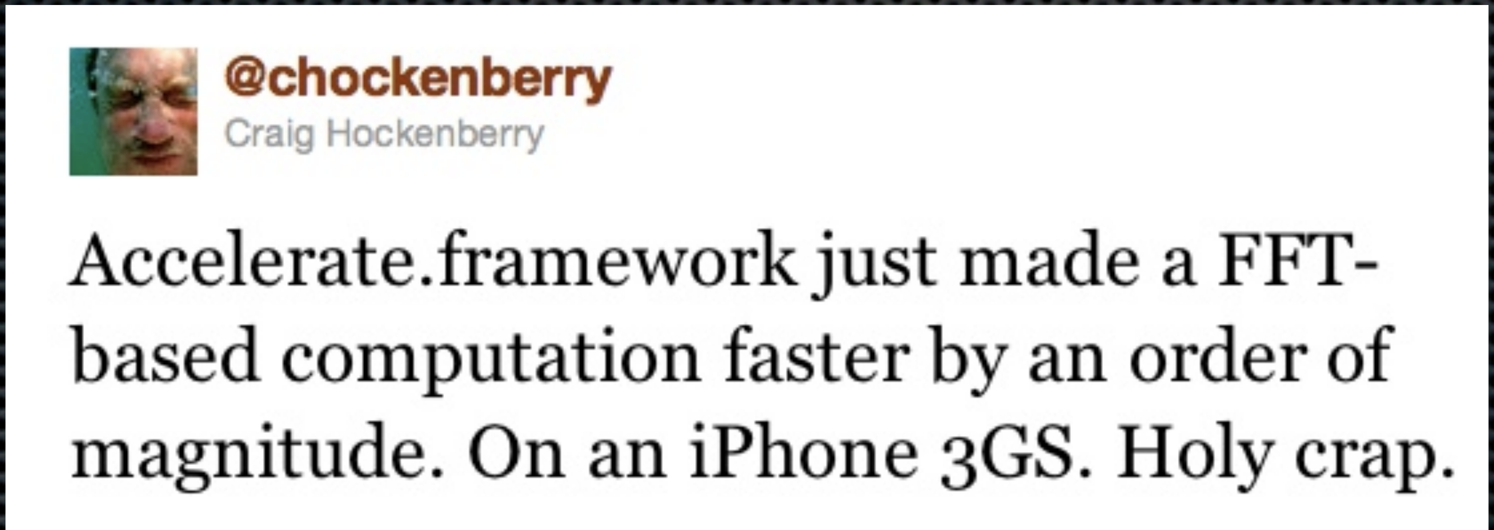
Accelerate Performance

- Not always the fastest solution
  - Small vectors / matrices sometimes are worse
  - Processing large data sets at once is the ideal
  - At some point, too much is also bad (faulting)
- Test out different sized data sets if it’s an option
  - Apple suggests ~32KiB (size of L2)
- Avoid striding where possible

Accelerate Your Code

- Look for places where you are dealing with...
  - image effects (real-time, large)
  - audio processing
  - arrays of data of any kind needing processing
  - loops that could be turned into arrays
  - functions that are hard to do on your own
  - fns that may benefit from hardware acceleration

Testify!



@chockenberry
Craig Hockenberry

Accelerate.framework just made a FFT-based computation faster by an order of magnitude. On an iPhone 3GS. Holy crap.

Accelerate Your Code

Step back and glance at your code every once in a while and see if it couldn’t make use of some bit of love from Accelerate

Check the libraries to see if there aren’t some gems in there that could help make your program stand out from the crowd

Accelerate Your Code

fin