Accelerate Framework & the Armadillo Library

Instructor - Simon Lucey

16-423 - Designing Computer Vision Apps
Today

• Motivation
• Accelerate Framework
• BLAS & LAPACK
• Armadillo Library
Algorithm
Software
Architecture
SOC Hardware
Abstract

Correlation filters take advantage of specific properties in the Fourier domain allowing them to be estimated efficiently: $O(ND\log D)$ in the frequency domain, versus $O(D^3 + ND^2)$ spatially where $D$ is signal length, and $N$ is the number of signals. Recent extensions to correlation filters, such as MOSSE, have reignited interest of their use in the vision community due to their robustness and attractive computational properties. In this paper we demonstrate, however, that this computational efficiency comes at a cost. Specifically, we demonstrate that only a proportion of shifted examples are unaffected by boundary effects which has a dramatic impact on detection/tracking performance. In this paper we propose a novel approach to correlation filter estimation that: (i) takes advantage of inherent computational redundancies in the frequency domain, (ii) dramatically reduces boundary effects, and (iii) is able to implicitly exploit all possible patches densely extracted from training examples during learning process. Impressive object tracking and detection results are presented in terms of both accuracy and computational efficiency.

1. Introduction

Correlation between two signals is a standard approach to feature detection/matching. Correlation touches nearly every facet of computer vision from pattern detection to object tracking. Correlation is rarely performed naively in the spatial domain. Instead, the fast Fourier transform (FFT) affords the efficient application of correlating a desired template/filter with a signal.

Correlation filters, developed initially in the seminal work of Hester and Casasent [15], are a method for learning a template/filter in the frequency domain that rise to some prominence in the 80s and 90s. Although many variants have been proposed [15, 18, 20, 19], the approach’s central tenet is to learn a filter, that when correlated with a set of training signals, gives a desired response, e.g. Figure 1 (b). Like correlation, one of the central advantages of the approach is that it attempts to learn the filter in the frequency domain due to the efficiency of correlation in that domain. Interest in correlation filters has been reignited in the vision world through the recent work of Bolme et al. [5] on Minimum Output Sum of Squared Error (MOSSE) correlation filters for object detection and tracking. Bolme et al.’s work was able to circumvent some of the classical problems.
// 5. Now apply some OpenCV operations
cv::Mat gray; cv::cvtColor(cvImage, gray, CV_RGBA2GRAY); // Convert to grayscale
cv::GaussianBlur(gray, gray, cv::Size(5,5), 1.2, 1.2); // Apply Gaussian blur
cv::Mat edges; cv::Canny(gray, edges, 0, 50); // Estimate edge map using Canny edge detector
SIMD Vector Extensions

What is it?
- Extension of the ISA
- Data types and instructions for the parallel computation on short (length 2, 4, 8, …) vectors of integers or floats
- Names: MMX, SSE, SSE2, ...

Why do they exist?
- Useful:
  - Many applications have the necessary fine-grain parallelism
  - Speedup by a factor close to vector length
- Doable:
  - Relative easy to design; chip designers have enough transistors to play with

<table>
<thead>
<tr>
<th>Register Width</th>
<th>MMX</th>
<th>SSE</th>
<th>AVX</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256 bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64 bit (only int)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Intel x86 Processors
- 8086
- 286
- 386
- 486
- Pentium
- Pentium MMX
- Pentium III
- Pentium 4
- Pentium 4E
- Pentium 4F
- Core 2 Duo
- Penryn
- Core i7 (Nehalem)
- Sandy Bridge
- Haswell
Reminder: Ideal Von Neumann Processor

- each cycle, CPU takes data from registers, does an operation, and puts the result back
- load/store operations (memory ←→ registers) also take one cycle
- CPU can do different operations each cycle output of one operation can be input to next

- CPU’s haven’t been this simple for a long time!

Taken from http://people.maths.ox.ac.uk/gilesm/cuda/lecs/lec0.pdf
Reminder: CPU clock is stuck!!!!

• CPU clock stuck at about 3GHz since 2006 due to high power consumption (up to 130W per chip)
• chip circuitry still doubling every 18-24 months
• ⇒ more on-chip memory and MMU (memory management units)
• ⇒ specialised hardware (e.g. multimedia, encryption) ⇒ multi-core (multiple CPU’s on one chip)
• peak performance of chip still doubling every 18-24 months

Taken from http://people.maths.ox.ac.uk/gilesm/cuda/ecs/lec0.pdf
Architecture Considerations

- Memory hierarchy.

- Vector instructions.

- Multiple threads.
Writing fast vision code…..

• In general you should **NOT** be trying to do these optimizations yourself.
• BUT, you should be using tools to find where the biggest losses in performance are coming from.
• Xcode comes with an excellent tool for doing this which is called “instruments”.
• Ray Wenderlich has a useful tutorial (see [link](#)) on using instruments in Xcode.
• More on this in later lectures.
Today

- Motivation
- Accelerate Framework
- BLAS & LAPACK
- Armadillo Library
Accelerate Framework

```cpp
// ViewController.m
// OpenCV_versus_Armadillo
// Created by Simon Lucey on 9
// Copyright (c) 2015 CMU_1643
//
#import "ViewController.h"

#ifndef __cplusplus
#include "armadillo" // Include
#include <opencv2/opencv.hpp>
#include <stdlib.h> // Include
#endif

@interface ViewController ()
```
Accelerate Framework

Accelerate (vForce)

Jaguar iOS 4
BLAS LAPACK vDSP
vMisc (vForce)

Tiger iOS 5
vImage

Taken from: http://www.mactech.com/sites/default/files/Biggus-Accelerate_IV.pdf
Accelerate Framework

1980 1990 2000 2010

BLAS LAPACK vDSP vForce vMathLib vBasicOps vBigNum vImage

Taken from: http://www.mactech.com/sites/default/files/Biggus-Accelerate_IV.pdf
Accelerate Framework

- **vImage**
  - “image operations”

- **vDSP**
  - “signal processing”

- **BLAS**
  - “matrix operations”

- **LAPACK**
  - “misc math”

- **vMisc**
  - “misc math”
Today

• Motivation
• Accelerate Framework
• BLAS & LAPACK
• Armadillo Library
Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz

Performance [Gflop/s]

- Multiple threads: 4x
- Vector instructions: 4x
- Memory hierarchy: 20x

\[ \text{MMM kernel function} \]

\[ \text{Multiple threads: 4x} \]

\[ \text{Vector instructions: 4x} \]

\[ \text{Memory hierarchy: 20x} \]

\[ \text{Matrix size} \]

\[ \text{Performance [Gflop/s]} \]

\[ \text{0} \quad \text{1,000} \quad \text{2,000} \quad \text{3,000} \quad \text{4,000} \quad \text{5,000} \quad \text{6,000} \quad \text{7,000} \quad \text{8,000} \quad \text{9,000} \]

\[ \text{0} \quad \text{5} \quad \text{10} \quad \text{15} \quad \text{20} \quad \text{25} \quad \text{30} \quad \text{35} \quad \text{40} \quad \text{45} \quad \text{50} \]

\[ \text{Matrix-Matrix Multiplication (MMM)} \]

\[ \text{in MATLAB} \]

\[ \text{A*B} \]

Taken from Markus Püschel - “How to Write Fast Numerical Code”.
BLAS

• Basic Linear Algebra Subprograms
  • Level 1 (70s)
    \[ \mathbf{y} \leftarrow \alpha \mathbf{x} + \mathbf{y} \]
  • Level 2 (mid 80s)
    \[ \mathbf{y} \leftarrow \alpha \mathbf{A} \mathbf{x} + \beta \mathbf{y} \]
  • Level 3 (late 80s)
    \[ \mathbf{C} \leftarrow \alpha \mathbf{A} \mathbf{B} + \beta \mathbf{C} \]

• BLAS was originally used to implement the linear algebra subroutine library (LINPACK).
The Path to LAPACK

- EISPACK and LINPACK (early 70s)
  - Libraries for linear algebra algorithms
  - Jack Dongarra, Jim Bunch, Cleve Moler, Gilbert Stewart
  - LINPACK still the name of the benchmark for the TOP500 (Wiki) list of most powerful supercomputers

- Problem
  - Implementation vector-based = low operational intensity (e.g., MMM as double loop over scalar products of vectors)
  - Low performance on computers with deep memory hierarchy (in the 80s)

- Solution: LAPACK
  - Reimplement the algorithms “block-based,” i.e., with locality
  - Developed late 1980s, early 1990s
  - Jim Demmel, Jack Dongarra et al.

Taken from Markus Püschel - “How to Write Fast Numerical Code”.
Availability of LAPACK

- LAPACK available on nearly all platforms.
- Numerous implementations,
  - Intel MKL (Windows, Linux, OS X)
  - AMD ACML
  - OpenBLAS (Windows, Linux, Android, OS X)
  - Apple Accelerate (OS X, iOS)
Which is Easier to Follow?

```c
#include <stdio.h>        /* I/O lib         ISOC */
#include <stdlib.h>       /* Standard Lib     ISOC */
#include "blaio.h"        /* Basic Linear Algebra I/O */

int main(int argc, char **argv) {
  double a[4*5] = { 1, 6, 11, 16,
                    2, 7, 12, 17,
                    3, 8, 13, 18,
                    4, 9, 14, 19,
                    5, 10, 15, 20
                   };

double x[5] = {2,3,4,5,6};
double y[4];

  printMatrix(CblasColMajor, 4, 5, a, 8, 3, NULL, NULL, NULL, NULL, NULL, NULL, "
        printVector(5, x, 8, 3, NULL, NULL, NULL, NULL, "
            x = " );

        /* row_order transform lenY lenX alpha a lda X incX
        cblas_dgemv(CblasColMajor, CblasNoTrans, 4, 5, 1, a, 4, x, 1,
            printVector(4, y, 8, 3, NULL, NULL, NULL, NULL, "
                y<-1.0*a*xT+0.0*y = " );

        /* row_order lenY lenX alpha X incX Y, incY A LDA */
        cblas_dger(CblasColMajor, 4, 5, 1, y, 1, x, 1, a, 4);
  printMatrix(CblasColMajor, 4, 5, a, 8, 3, NULL, NULL, NULL, NULL, NULL, NULL, "a

  return 0;
} /* end func main*/
```
Which is Easier to Follow?

```plaintext
>> y = A*x
```
MATLAB

- Invented in the late 70s by Cleve Moler
- Commercialized (MathWorks) in 84
- Motivation: Make LINPACK, EISPACK easy to use
- Matlab uses LAPACK and other libraries but can only call it if you operate with matrices and vectors and do not write your own loops
  - \( A*B \) (calls MMM routine)
  - \( A\backslash b \) (calls linear system solver)
“Computer Vision Algorithms”
Problems with MATLAB

• Proprietary command line interpreted package.
• Extremely large (current desktop version is 6.83 Gb - compressed!!!).
• Designed more for prototyping, on high-end desktops.
• Not very useful for mobile development.
“Computer Vision Algorithms”
Problems with OpenCV

- OpenCV improves greatly upon this issue.
  - Completely free and written in C++.
  - Has an OK matrix library, relatively easy to interpret.
  - Much light(er) weight (in size) than MATLAB.
- However, has problems.
  - Still relatively big - `opencv2.framework` is 23Mb compressed!!!
  - Not as fast as it should/could be.
- Alternate light-weight math libraries can help here,
  - **Eigen** (support for ARM NEON intrinsics)
  - **Armadillo** (uses LAPACK, MATLAB syntax)
Side Note: How Big Should an App Be?

• Customers and clients care about app size…
  • Average size of App is around 23 Mb, and for games is now 60Mb (see link).
  • Apple has a maximum cellular download limit of 100MB (see link).
  • Size of current opencv2.framework is 78.7 Mb - uncompressed!

• Important consideration in the design of a computer vision app is its size.
Accelerate Framework comes “built in” to all iOS devices.
NOTHING TO DOWNLOAD!!
Today

• Motivation
• Accelerate Framework
• BLAS & LAPACK
• Armadillo Library
“Computer Vision Algorithms”

BLAS  LAPACK

+  ?
Armadillo - C++ Algebra Library

• **Armadillo** is a clean C++ math/algebra library.

• Like MATLAB sits on top of BLAS + LAPACK.

• Unlike MATLAB it is,
  • it is extremely light-weight and small.
  • portable across any platform (iOS, Android, Linux, Windows, MAC OS X).
  • C++ [templated library](#) so it can be used easily within Objective C in iOS and other mobile platforms.
## Armadillo to MATLAB

<table>
<thead>
<tr>
<th>Matlab/Octave</th>
<th>Armadillo</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(1, 1)</td>
<td>A(0, 0)</td>
<td>indexing in Armadillo starts at 0</td>
</tr>
<tr>
<td>A(k, k)</td>
<td>A(k-1, k-1)</td>
<td></td>
</tr>
<tr>
<td>size(A, 1)</td>
<td>A.n_rows</td>
<td>read only</td>
</tr>
<tr>
<td>size(A, 2)</td>
<td>A.n_cols</td>
<td></td>
</tr>
<tr>
<td>size(Q, 3)</td>
<td>Q.n_slices</td>
<td>Q is a <strong>cube</strong> (3D array)</td>
</tr>
<tr>
<td>numel(A)</td>
<td>A.n_elem</td>
<td></td>
</tr>
<tr>
<td>A(:, k)</td>
<td>A.col(k)</td>
<td></td>
</tr>
<tr>
<td>A(k, :)</td>
<td>A.row(k)</td>
<td>this is a conceptual example only; exact conversion fr will require taking into account that indexing starts at 0</td>
</tr>
<tr>
<td>A(:, p:q)</td>
<td>A.cols(p, q)</td>
<td></td>
</tr>
<tr>
<td>A(p:q, :)</td>
<td>A.rows(p, q)</td>
<td></td>
</tr>
<tr>
<td>A(p:q, r:s)</td>
<td>A( span(p, q), span(r, s) )</td>
<td>A( span(first_row, last_row), span(first_col, last_col) )</td>
</tr>
</tbody>
</table>

- Please follow [link](#) for the full API documentation on the Armadillo library.
Armadillo in Xcode

```
// ViewController.m
// OpenCV_versus_Armadillo
// Created by Simon Lucey on 9
// Copyright (c) 2015 CMU_1643
//
#import "ViewController.h"

#ifdef _cplusplus
#include "armadillo"
#endif

#include <opencv2/opencv.hpp> // Include
#include <stdlib.h> // Include
@endinterface ViewController()
```
Armadillo versus OpenCV

• We are now going to have a play with Armadillo, in comparison to OpenCV.
• On your browser please go to the address,
  https://github.com/slucey-cs-cmu-edu/OpenCV_vs_Armadillo

• Or better yet, if you have git installed you can type from the command line.

$ git clone https://github.com/slucey-cs-cmu-edu/OpenCV_vs_Armadillo.git
Armadillo versus OpenCV

```cpp
#import "ViewController.h"

#ifdef __cplusplus
#include "armadillo" // Includes the armadillo library
#include <opencv2/opencv.hpp> // Includes the opencv library
#include <stdlib.h> // Include the standard library
#endif

@interface ViewController ()
@end

@implementation ViewController

-(void)viewDidLoad {
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.

    // Simple comparison between Armadillo and OpenCV
    using namespace std;

    int D = 3000; // Number of columns in A
    int M = 400; // Number of rows in A
    int trials = 3000; // Number of trials

    // Step 1. initialize random data
    // In MATLAB: x = randn(D,1);
    arma::fmat x; x.randn(D,1);
    // In MATLAB: A = randn(D,D);
    arma::fmat A; A.randn(M,D);
```
// Step 2. initialize the clock
arma::wall_clock timer;

// Step 3. apply matrix multiplication operation in OpenCV
// Remember: in OpenCV everything is stored in row order
// so cvA is a DxM matrix not a MxD matrix!!!!

cv::Mat cvA = Arma2Cv(A); // Convert to an OpenCV matrix
cv::Mat cvx = Arma2Cv(x); // Convert to an OpenCV vector
cv::Mat cvy(1,M,CV_32F); // Allocate space for y
timer.tic();
for(int i=0; i<trials; i++) {
    cvy = cvx*cvA; // Apply multiplication in OpenCV
}
double cv_n_secs = timer.toc();
cout << "OpenCV took " << cv_n_secs << " seconds." << endl;

// Step 4. apply matrix multiplication in Armadillo
arma::fmat y(M,1); // Allocate space first
timer.tic();
for(int i=0; i<trials; i++) {
    y = A*x; // Apply multiplication in Armadillo
}
double arma_n_secs = timer.toc();
cout << "Armadillo took " << arma_n_secs << " seconds." << endl;
cout << "Armadillo is " << cv_n_secs/arma_n_secs << " times faster than OpenCV!!!" << endl;
On the iPhone 6 Simulator

```cpp
// Simple comparison between Armadillo and OpenCV
using namespace std;

int D = 3000; // Number of columns in A
int M = 400; // Number of rows in A
int trials = 3000; // Number of trials

// Step 1. initialize random data
// In MATLAB: x = randn(D,1);
arma::fmat x; x.randn(D,1);
// In MATLAB: A = randn(D,D);
arma::fmat A; A.randn(M,D);

OpenCV took 3.99296 seconds.
Armadillo took 0.375662 seconds.
Armadillo is 10.6291 times faster than OpenCV!!!
On the Device - iPhone 6

```cpp
// Simple comparison between Armadillo and OpenCV
using namespace std;

int D = 3000; // Number of columns in A
int M = 400; // Number of rows in A
int trials = 3000; // Number of trials

// Step 1. initialize random data
// In MATLAB: x = randn(D,1);
arma::fmat x; x.randn(D,1);
// In MATLAB: A = randn(D,D);
arma::fmat A; A.randn(M,D);
```

OpenCV took 10.9482 seconds.
Armadillo took 2.73892 seconds.
Armadillo is 3.99727 times faster than OpenCV!!!
int M = 400; // Number of rows in A
int trials = 3000; // Number of trials

// Step 1. initialize random data
// In MATLAB: x = randn(D,1);
arma::fmat x; x.randn(D,1);
// In MATLAB: A = randn(D,D);
arma::fmat A; A.randn(M,D);

// Step 2. initialize the clock
arma::wall_clock timer;

// Step 3. apply matrix multiplication operation in OpenCV
// Remember: in OpenCV everything is stored in row order
// so cvA is a DxM matrix not a MxD matrix!!!!

cv::Mat cvA = arma2Cv(A); // Convert to an OpenCV matrix
cv::Mat cvx = arma2Cv(x); // Convert to an OpenCV vector

cv::Mat cvy(1,M,CV_32F); // Allocate space for y

timer.tic();
for(int i=0; i< trials; i++) {
    cvy = cvx*cvA; // Apply multiplication in OpenCV
}
double cv_n_secs = timer.toc();
cout << "OpenCV took " << cv_n_secs << " seconds." << endl;

// Step 4. apply matrix multiplication in Armadillo
arma::fmat y(M,1); // Allocate space first
timer.tic();
for(int i=0; i< trials; i++) {
    y = A*x; // Apply multiplication in Armadillo
}
double arma_n_secs = timer.toc();
cout << "Armadillo took " << arma_n_secs << " seconds." << endl;
cout << "Armadillo is " << cv_n_secs/arma_n_secs << " times faster than OpenCV!!" << endl;

-(void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

// Quick function to convert to Armadillo matrix header
arma::fmat Cv2Arma(cv::Mat &cvX) {
arma::fmat X(cvX.ptr<float>(0), cvX.cols, cvX.rows, false); // This is the transpose of the OpenCV X
return X; // Return the new matrix (no new memory allocated)
On the Device - iPad 2

```cpp
// Simple comparison between Armadillo and OpenCV
using namespace std;

int D = 3000; // Number of columns in A
int M = 400; // Number of rows in A
int trials = 3000; // Number of trials

// Step 1. initialize random data
// In MATLAB: x = randn(D,1);
arma::fmat x; x.randn(D,1);
// In MATLAB: A = randn(D,D);
arma::fmat A; A.randn(M,D);
```

OpenCV took 53.1328 seconds.
Armadillo took 18.7615 seconds.
Armadillo is 2.83202 times faster than OpenCV!!!
Armadillo Examples

- Feel free to try out this Armadillo example, that uses matrix multiplication, SVD, Backslash, and FFT.
- On your browser please go to the address,

https://github.com/slucey-cs-cmu-edu/Intro_iOS_Armadillo

- Or better yet, if you have git installed you can type from the command line.

$ git clone https://github.com/slucey-cs-cmu-edu/Intro_iOS_Armadillo.git