OpenCV

AND OTHER TOOLS OF THE TRADE

Saumitro Dasgupta

Stanford University / **CS231M** 2015

Roadmap

- Introduction to OpenCV
- Matrices in OpenCV
- Common pitfalls
- Best practices
- Live Demo
- Introduction to the Eigen library
- Optimizing your algorithms using ARM NEON

OpenCV

- An open source BSD licensed computer vision library
    - Patent-encumbered code isolated into "non-free" module. (This includes SIFT!)
- Available on all major platforms
    - Android, iOS, Linux, Mac OS X, Windows
- Written primarily in C++
    - Bindings available for Python, Java, …
- Well documented at http://docs.opencv.org/
- Source available at https://github.com/Itseez/opencv
- **Caution**: Not necessarily the best tool for the job!

What can it do?

Image Processing	Filters, Histograms, Morphology, Color Ops…
Feature Detection	Edges, Corners, Lines, Circles, SIFT, SURF, ORB…
Object Detection	Haar, Latent SVM, Template Matching…
Machine Learning	SVM, Bayes, Decision Trees, Neural Networks, Clustering, Boosting…
Motion Tracking	Optical Flow, Kalman Filters, MeanShift…
3D Geometry	Camera Calibration, Homography, Fundamental Matrix…

OpenCV for Mobile Platforms

Android

- Include the Tegra optimized OpenCV makefile: `OpenCV-[version]-Tegra-sdk/sdk/native/jni/OpenCV-tegra3.mk` to Android.mk
- Tutorials

iOS

- Include OpenCV.framework
- Option 1: Pre-built binaries for iOS
- Option 2: Build from source
- Tutorials

The **Mat** datatype

```
// Matrix of doubles (64-bit floats) with 480 rows and 640 columns.
// Single channel (equivalent to CV_64FC1)
Mat grayscaleImage(480, 640, CV_64F);

// Matrix of 8-bit unsigned integers with 480 rows and 640 columns.
// Three channels.
Mat rgbImage(480, 640, CV_8UC3);
```

- The **Mat** class represents a fixed type dense n-dimensional array
- Used for representing a wide range of things: images, transformations, optical flow maps, trifocal tensor…
- A Mat can have multiple channels
    - Example: A 640x480 RGB image will be a Mat with 480 rows, 640 columns, and 3 channels.
    - Number of channels is part of the type signature (and not the matrix dimension)

Deep / Shallow Copies

- Assignment creates shallow copies.
- Be aware of accidental mutation.
- Memory allocation is expensive.
  Avoid unnecessary deep copies.
- Auto-memory management.
  Internally reference counted.

```cpp
// matrix_1 allocates a new block of memory.
Mat matrix_1(1024, 1024, CV_64F);

// matrix_2 and matrix_2 share the same data.
Mat matrix_2 = matrix_1;

// matrix_1 and matrix_2 are both affected.
matrix_2.at<double>(5, 7) = 42.0;

// row_10 also shares the same data.
Mat row_10 = matrix_1.row(10);

// matrix_3 creates a copy of the data.
Mat matrix_3 = matrix_1.clone();
```

Image Data: **Endianness**

- Each pixel can be represented using **four 8-bit values** (uint8_t).
  3 for **RGB** + **1** for **alpha** channel (transparency).

- Therefore, a single pixel can be packed into a **single 32-bit value** (uint32_t).

- Consider this code snippet, run on a modern x86 processor:

```
uint32_t pixel = 0xFEEDBEEF;
uint8_t* p = (uint8_t*)&pixel;
printf("(%x, %x, %x, %x)", p[0], p[1], p[2], p[3]);
```

```
Output: (ef, be, ed, fe)
```

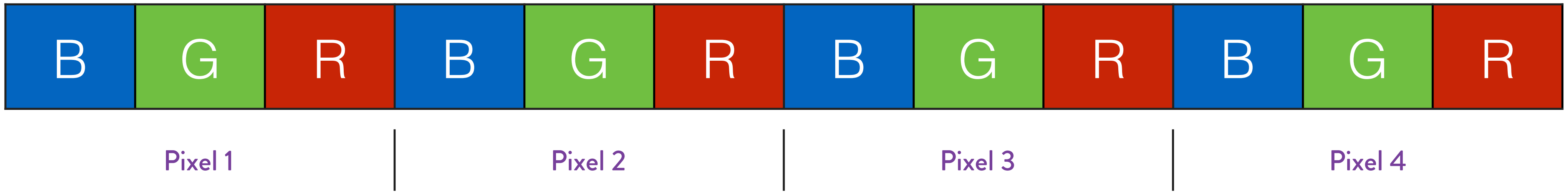
- Intel x86 processors are **little endian**: the LSB (least significant bit) is stored first.

- OpenCV uses the BGR(A) byte ordering.

Image Data: **Layout**

`Address(M[i,j]) = BaseAddress(M) + i·Stride(M) + j·PixelSize(M)`

```
Mat m(480, 640, CV_8UC3);
// Pixel size: 3, Stride (or step): 1920 (640*3)
size_t stride = m.step1(), pixelSize = m.elemSize();
// Set red channel of pixel at (x, y) to 128
int x = 500, y = 200, c = 2;
*(m.data + y*stride + x*pixelSize + c) = 128;
```



B	G	R	B	G	R	B	G	R	B	G	R

- OpenCV matrices are stored in row major order.
- Usually stored as a contiguous array (verify using the `isContinuous` method).

InputArray / OutputArray

```
// Converts point coordinates from normal pixel coordinates
// to homogeneous coordinates ((x,y)->(x,y,1))
void convertPointsToHomogeneous(InputArray src, OutputArray dst);
```

- InputArray and OutputArray are proxy classes.
  Based on the function, they can be concretely substituted by cv::Mat and/or std::vector.

- An InputArray is immutable (const-enforced).

Stack-based matrices

- Mat allocates memory on the heap.

- We want to avoid allocating memory on the heap.
  Dynamic memory allocation is expensive, can interfere with cache locality.

- The **Matx** type is suitable for small matrices.
  Allocated on the **stack**.

- Matx can usually be used wherever you'd use a Mat (a few exceptions exist).

```
// Use OpenCV's implementation of the Rodrigues transform
// to convert a rotation matrix to the angle-axis form.
Matx33f rotMat = getRotationMatrix();
Matx<float, 3, 1> rotVec;
Rodrigues(rotMat, rotVec);
```

Live Demo

- Let's build an **optical flow tracker.**
  (We'll have a whole lecture on optical flow later)

- We'll code on Mac OS X, deploy to iOS.

- The core vision code remains the same across platforms.

- Trivial to port to Android.

The **Eigen** library

- Open source **header-only** C++ linear algebra library.
- Cross-platform, elegant API, well documented.
- Optimized for multiple platforms.
  SSE 2/3/4, ARM NEON
- More flexible than OpenCV's Mat class.
- Interoperable with OpenCV's Mat class without copying.

A taste of Eigen

Linear least squares, three different flavors.

```
// Least squares solution to A*x = b.
MatrixXf A = MatrixXf::Random(3, 2);
VectorXf b = VectorXf::Random(3);
VectorXf x;

// Using SVD:
x = A.jacobiSvd(ComputeThinU | ComputeThinV).solve(b);
cout << "x = " << x << "\n";

// Using QR decomposition:
x = A.colPivHouseholderQr().solve(b);
cout << "x = " << x << "\n";

// Using the normal equations:
x = (A.transpose()*A).ldlt().solve(A.transpose()*b);
cout << "x = " << x << "\n";
```

Eigen Quick Reference

- Elegant API for matrix operations.
- Includes both fixed and dynamic matrices.
- Static consistency checks
- A more complete reference is available.

```
// Construction.
MatrixXd T = MatrixXd::Identity();
Matrix3f M;
M << 1, 2, 3,
     4, 5, 6,
     7, 8, 9;

// Accessing elements.
float matrixElem = M(2, 0);

// Reductions.
Vector3f rowSum = M.rowwise().sum();
Vector3f colAvg = M.colwise().mean();

// Arithmetic.
MatrixXf X = MatrixXf::Random(3, 4);
Matrix<float, 3, 4> matrixProduct = M*X;
// Compile-time error:
MatrixXf invalidProduct = matrixProduct*M;
// Run-time error:
MatrixXf alsoInvalid = X*M;
```

Aligned allocations

- Structs containing Eigen's fixed size vectorizable objects need to ensure that they're aligned.
- Required for SIMD operations (SSE).
- Not necessary for dynamically allocated objects.
- More details available over here and here.

```
class Landmark
{
private:
    Vector3f position;
    Quaternionf orientation;

public:
    // Add this macro:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW
    Landmark();
    ~Landmark();
};
```

```
// Use aligned_allocator for STL containers.
vector<Vector3f,aligned_allocator<Vector3f>> points;
```

We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil.**

Donald Knuth

ACM Computing Surveys, Vol 6, No. 4,December 1974

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. **Yet we should not pass up our opportunities in that critical 3%.**

Donald Knuth

ACM Computing Surveys, Vol 6, No. 4,December 1974

Optimizing using ARM NEON

- NEON is ARM’s packed SIMD coprocessor.
- Designed for vectorized operations (well-suited for image processing tasks).
- Significant speed-ups can be obtained for many common vision algorithms.
- Many libraries include NEON optimizations (OpenCV, Eigen, Skia…).
- 32 64-bit registers (or 16 128-bit registers).
- Straight-up assembly or C friendly intrinsics (`#include <arm_neon.h>`).
- Example: let’s optimize an RGB to grayscale color conversion function.

Baseline color conversion

```
void rgb_to_gray(const uint8_t* rgb, uint8_t* gray, int num_pixels)
{
    for(int i=0; i<num_pixels; ++i, rgb+=3)
    {
        int v = (77*rgb[0] + 150*rgb[1] + 29*rgb[2]);
        gray[i] = v>>8;
    }
}
```

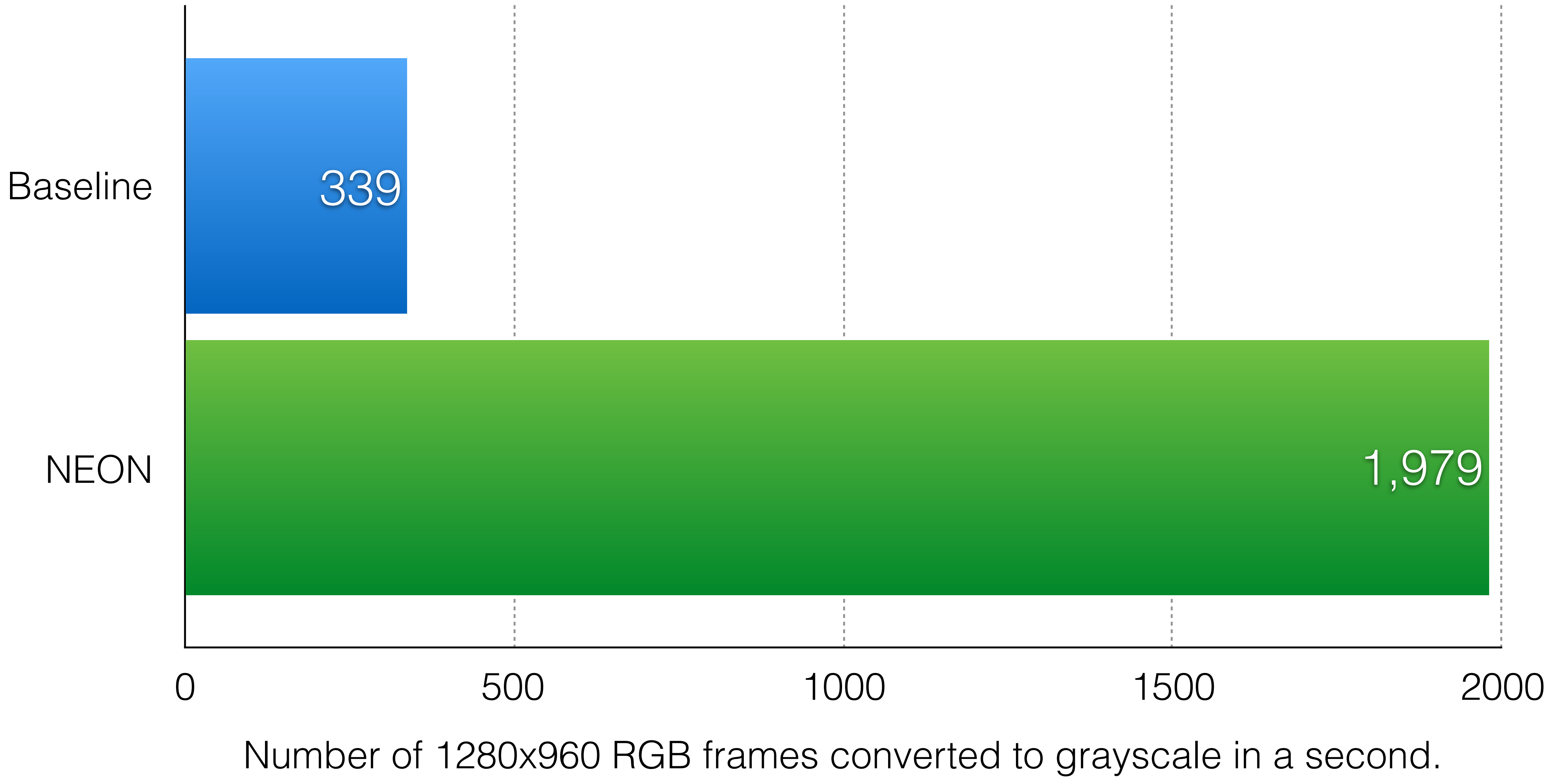
- Simple luminance-preserving RGB to grayscale conversion, based on weighted sums.
- Weights based on human perception of color.
- We process one pixel at a time.

Using NEON intrinsics

```c
void rgb_to_gray_neon(const uint8_t* rgb, uint8_t* gray, int num_pixels)
{
    // We'll use 64-bit NEON registers to process 8 pixels in parallel.
    num_pixels /= 8;
    // Duplicate the weight 8 times.
    uint8x8_t w_r = vdup_n_u8(77);
    uint8x8_t w_g = vdup_n_u8(150);
    uint8x8_t w_b = vdup_n_u8(29);
    // For intermediate results. 16-bit/pixel to avoid overflow.
    uint16x8_t temp;
    // For the converted grayscale values.
    uint8x8_t result;
    for(int i=0; i<num_pixels; ++i, rgb+=8*3, gray+=8)
    {
        // Load 8 pixels into 3 64-bit registers, split by channel.
        uint8x8x3_t src = vld3_u8(rgb);
        // Multiply all eight red pixels by the corresponding weights.
        temp = vmull_u8(src.val[0], w_r);
        // Combined multiply and addition.
        temp = vmlal_u8(temp, src.val[1], w_g);
        temp = vmlal_u8(temp, src.val[2], w_b);
        // Shift right by 8, "narrow" to 8-bits (recall temp is 16-bit).
        result = vshrn_n_u16(temp, 8);
        // Store converted pixels in the output grayscale image.
        vst1_u8(gray, result);
    }
}
```

Based on a version by Nils Pipenbrinck, hilbert-space.de

Optimization Results



Tested on an iPhone 5S.
Built using clang v6.0 (600.0.57) with -O3.

Other Libraries

- The Point Cloud Library (PCL)
  Framework for working with 3D point clouds.
- CCV
  Implements many modern vision algorithms (Predator, DPMs,...)
- VLFeat
  Mature and well-documented vision library, includes MATLAB bindings.
- LibCVD
  Includes an optimized FAST corner detector implementation.
- Sophus
  Lie groups library for Eigen. Useful for SLAM / Visual Odometry.
- Caffe
  Conv nets framework from Berkeley.
- DeepBeliefSDK
  Conv nets for mobile platforms.