~~Image Learning and~~ Computer Vision in CUDA

Peter Andreas Entschev - peter@arrayfire.com

HPC Engineer

Finding visually similar images: Perceptual image hashing

Explanation of the problem

Want to find images that are similar in appearance

- Can't do pixel-per-pixel subtraction
  - Shifts / rotation
  - Color table changes
  - Modifications

Need an alternative method!

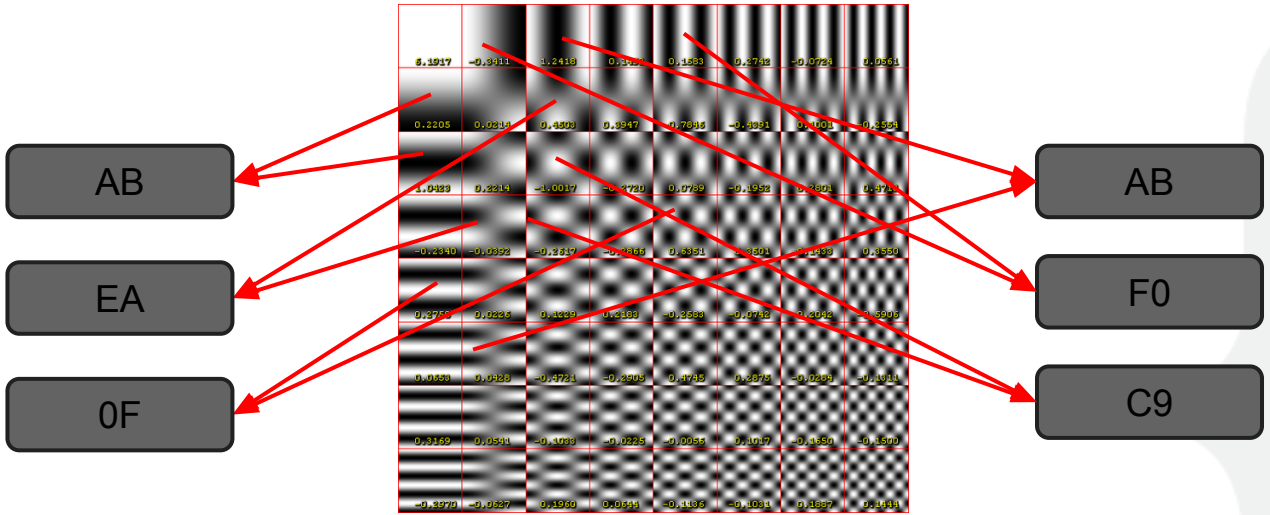
One solution: Perceptual image hashing

Create a hash from an image… but how?

1. Filter and resize an image to a standard resolution
2. Compute the DCT coefficients of the image
3. Create a hash from the DCT coefficients

See Zauner (2010) Ph.D. thesis for further details

Why DCT coefficients?



Invariant to

- Color changes
- Image compression artifacts
- Translation and minor rotation

DFT coefficient graphic from Wikimedia commons

Hamming Distance Matching

- A measurement of the distance between two strings

String 1	String 2	Distance
00101	**1**0101	1
1234**5**	1334**4**	2
well	**wa**ll	1

Calculating the Hamming distance (CPU)

```
hamming_distance(const uint64_t hash1,
                 const uint64_t hash2)
{
    const uint64_t x = hash1^hash2;
    const uint64_t m1  = 0x5555555555555555LL;
    const uint64_t m2  = 0x3333333333333333LL;
    const uint64_t h01 = 0x0101010101010101LL;
    const uint64_t m4  = 0x0f0f0f0f0f0f0f0fLL;
    x -= (x >> 1) & m1;
    x = (x & m2) + ((x >> 2) & m2);
    x = (x + (x >> 4)) & m4;
    return (x * h01) >> 56;
}
```

Calculating the Hamming distance (GPU)

```
// Load image
af::array hash1 = constant(0xe93c2649cb96f449, 1, 1);
af::array hash2 = constant(0xe93c2649cb96f449, 1, 1);

af::array idx, dist;

af::hamming_matcher(idx, dist, hash1, hash2);
```

Perceptual hashing benefits

Hash: 0xe93c2649cb96f449

Hash: 0xe9382669db92f449
Distance: 4

Algorithm is invariant to

- **Minor color changes**
- Image compression artifacts
- Translation and minor rotations
- Image resizing

Image from the Blender Foundation's Big Buck Bunny video

Perceptual hashing benefits

Hash: 0xe93c2649cb96f449

Hash: 0xf9342649dbb2f049
Distance: 6

Algorithm is invariant to

- Minor color changes
- **Image compression artifacts**
- Translation and minor rotations
- Image resizing

Image from the Blender Foundation's Big Buck Bunny video

Perceptual hashing benefits

Hash: 0xe93c2649cb96f449

Hash: 0xe8610e4d9b96f469
Distance: 12

Algorithm is invariant to

- Minor color changes
- Image compression artifacts
- **Translation and minor rotations**
- Image resizing

Image from the Blender Foundation's Big Buck Bunny video

Perceptual hashing benefits

1080p original
Hash: 0x316977325ec8f8a8

Resized to 480p
Hash: 0x716937325ec8f8a8
Distance: 2

Algorithm is invariant to

- Minor color changes
- Image compression artifacts
- Translation and minor rotations
- **Image resizing**

Image from the Blender Foundation's Big Buck Bunny video

pHash phases

1. Create luminance image from RGB values
2. Apply 7x7 mean filter to image
3. Resize image to 32x32 pixels
4. Compute the DCT of the image
5. Extract 64 coefficients ignoring the lowest order
6. Find the median coefficient
7. Create the hash using the median as a threshold

Implementing pHash using ArrayFire

```
// Load image
af::array image = af::loadImage(filename.c_str(), true);

// Create luminance image from RGB values
af::array luminance = af::rgb2gray(image, 0.2126, 0.7152, 0.0722);

// Apply 7x7 mean filter to image
af::array mean_filter = af::constant(1.0 / (7 * 7), 7, 7);
af::array filtered = af::convolve(luminance, mean_filter);

// Resize image to 32x32 pixels
af::array resized = af::resize(filtered, 32, 32, AF_INTERP_NEAREST);
```

Implementing pHash using ArrayFire

```
// Create the DCT matrix and its transpose
af::array C_k = af::dctmatrix(32);
af::array C_k_T = C_k.T();
C_k_T.eval()

// Compute the DCT by matrix multiplication
af::array imageDCT = matmul(matmul(C_k, resized), C_k_T);
```

Implementing pHash using ArrayFire

```
// Extract 64 coefficients ignoring the lowest order
af::array subset = imageDCT(seq(1,8), seq(1,8));

// Find the median coefficient
af::array median = af_median<float>(subset);
float * h_subset = subset.host<float>();
```

Implementing pHash using ArrayFire

```
// Compute the 64-bit hash
uint64_t one =  0x0000000000000001;
uint64_t hash = 0x0000000000000000;
for (int i = 0; i < 64; i++){
    float current = h_subset[i];
    if (current > median)
        hash |= one;
     one = one << 1;
}

return hash;
```

Performance - ArrayFire vs. pHash

- Dataset:
  - Proprietary
  - ~50 million images
  - Size distribution:
    - 32 x 32 - 2048 x 2048 pixels
    - Most images are not square
  - Selected 50k images at random
- Speed up using ArrayFire vs. pHash
  - 5.6x using CUDA backend including disk I/O

Feature detection and tracking

Definition: Feature Tracking

The act of finding highly distinctive image properties (features) in a given scene

Definition: Object Recognition

The act of identifying an object based on its geometry

Image Source: Visual Geometry Group (2004). University of Oxford, http://www.robots.ox.ac.uk/~vgg/data/data-aff.html

Feature Tracking Phases

1. Feature detection:
   - ➔ Finding highly distinctive properties of objects (e.g., corners)
2. Descriptor extraction:
   - ➔ Encoding of a texture patch around each feature
3. Descriptor matching:
   - ➔ Finding similar texture patches in distinct images

Feature Tracking History - 17 Year Review

- SIFT - Scale Invariant Feature Transform (1999, 2004)
- SURF - Speeded Up Robust Features (2006)
- FAST - High-speed Corner Detection (2006, 2010)
- BRIEF - Binary Robust Independent Elementary Features (2010)
- ORB - Oriented FAST and Rotated BRIEF (2011)
- KAZE/Accelerated KAZE Features (2012, 2013)

Computer Vision Applications

- 3D scene reconstruction
- Image registration
- Object recognition
- Content retrieval

Computational Challenges

- Computationally expensive
- Real-time requirement
- Memory access patterns
- Memory footprint

Harris Feature Detector

1. Compute image gradients
2. Second-order derivatives
3. Filter second-order derivatives with a filter (Gaussian or weighted-sum)
4. Compute determinant and trace of derivatives matrix
5. Calculate response as a function of determinant and trace

Harris Feature Detector – Implementation

```
// Load image
af::array img = af::loadImage("image.png", true);
```

Harris Feature Detector – Implementation

`img`

Harris Feature Detector – Implementation

```
// Load image
af::array img = af::loadImage("image.png", false);

// Calculate image gradients
af::array ix, iy;
af::grad(iy, ix, img);
```

Harris Feature Detector – Implementation

ix iy

Harris Feature Detector – Implementation

```
// Load image
af::array img = af::loadImage("image.png", true);

// Calculate image gradients
af::array ix, iy;
af::grad(iy, ix, img);

// Second-order derivatives
af::array ixx = ix * ix;
af::array ixy = ix * iy;
af::array iyy = iy * iy;
```

Harris Feature Detector – Implementation

ixx

ixy

iyy

Harris Feature Detector – Implementation

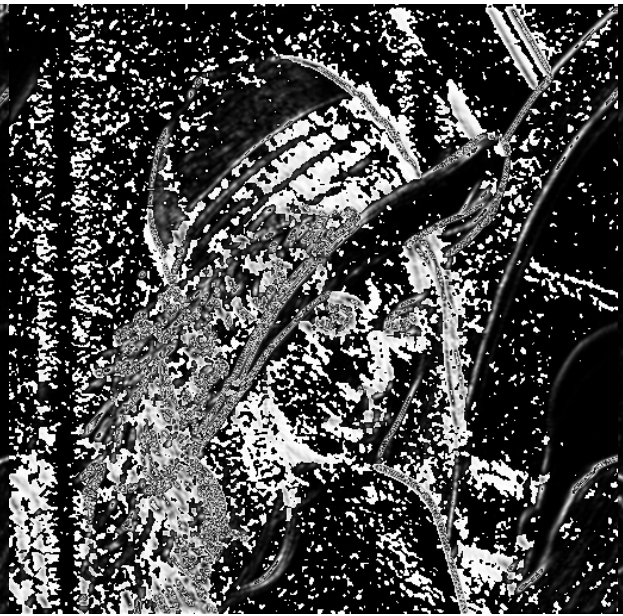
```
// Load image
af::array img = af::loadImage("image.png", true);

// Calculate image gradients
af::array ix, iy;
af::grad(iy, ix, img);

// Second-order derivatives
af::array ixx = ix * ix;
af::array ixy = ix * iy;
af::array iyy = iy * iy;

// Gaussian kernel with standard deviation of 1.0 and length of 5 pixels
af::array gauss_filt = af::gaussianKernel(5, 5, 1.0, 1.0);
```

Harris Feature Detector – Implementation

```
// Filter second-order derivatives with Gaussian kernel
ixx = af::convolve(ixx, gauss_filt);
ixy = af::convolve(ixy, gauss_filt);
iyy = af::convolve(iyy, gauss_filt);
```

Harris Feature Detector – Implementation

ixx



ixy

iyy

Harris Feature Detector – Implementation

```
// Filter second-order derivatives with Gaussian kernel
ixx = af::convolve(ixx, gauss_filt);
ixy = af::convolve(ixy, gauss_filt);
iyy = af::convolve(iyy, gauss_filt);

// Calculate trace
af::array tr = ixx + iyy;
// Calculate determinant
af::array det = ixx * iyy - ixy * ixy;
```

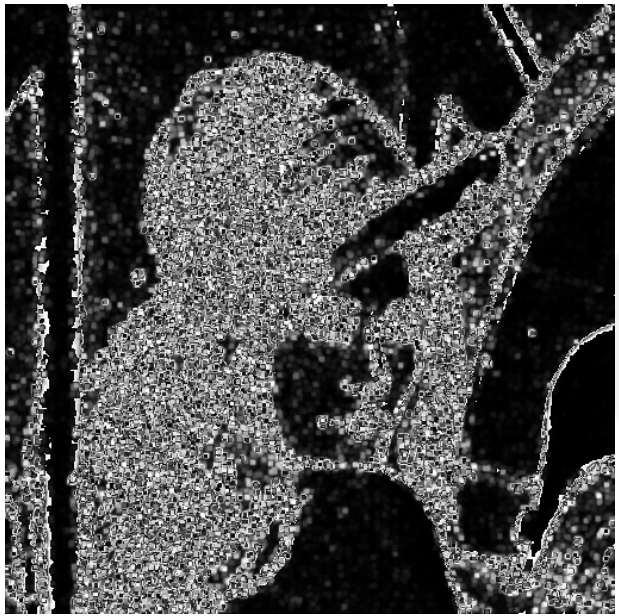
Harris Feature Detector – Implementation

```
// Filter second-order derivatives with Gaussian kernel
ixx = af::convolve(ixx, gauss_filt);
ixy = af::convolve(ixy, gauss_filt);
iyy = af::convolve(iyy, gauss_filt);

// Calculate trace
af::array tr = ixx + iyy;
// Calculate determinant
af::array det = ixx * iyy - ixy * ixy;

// Calculate Harris response
af::array response = det - 0.04f * (tr * tr);
```

Harris Feature Detector – Implementation

```
// Gets maximum response for each 3x3 neighborhood
af::array mask = constant(1,3,3);
af::array max_resp = dilate(response, mask);
```

Harris Feature Detector – Implementation

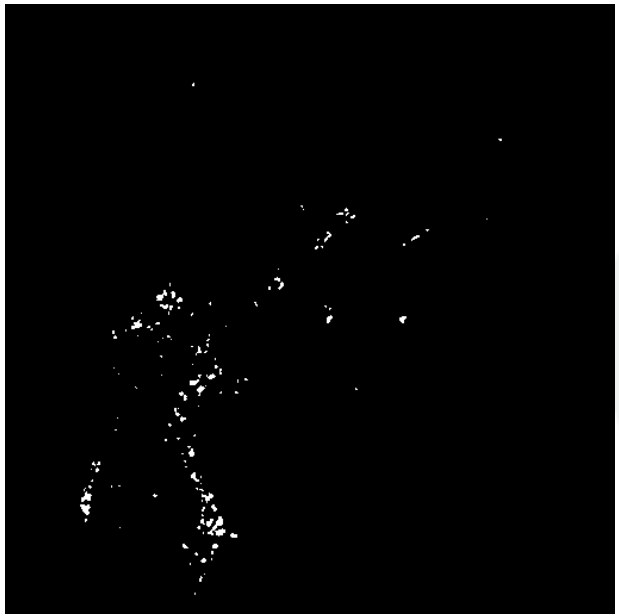


max_resp

Harris Feature Detector – Implementation

```
// Gets maximum response for each 3x3 neighborhood
af::array mask = constant(1,3,3);
af::array max_resp = dilate(response, mask);

// Discard responses that are not greater than threshold
af::array corners = response > 1e5f;
corners = corners * response;
```

Harris Feature Detector – Implementation

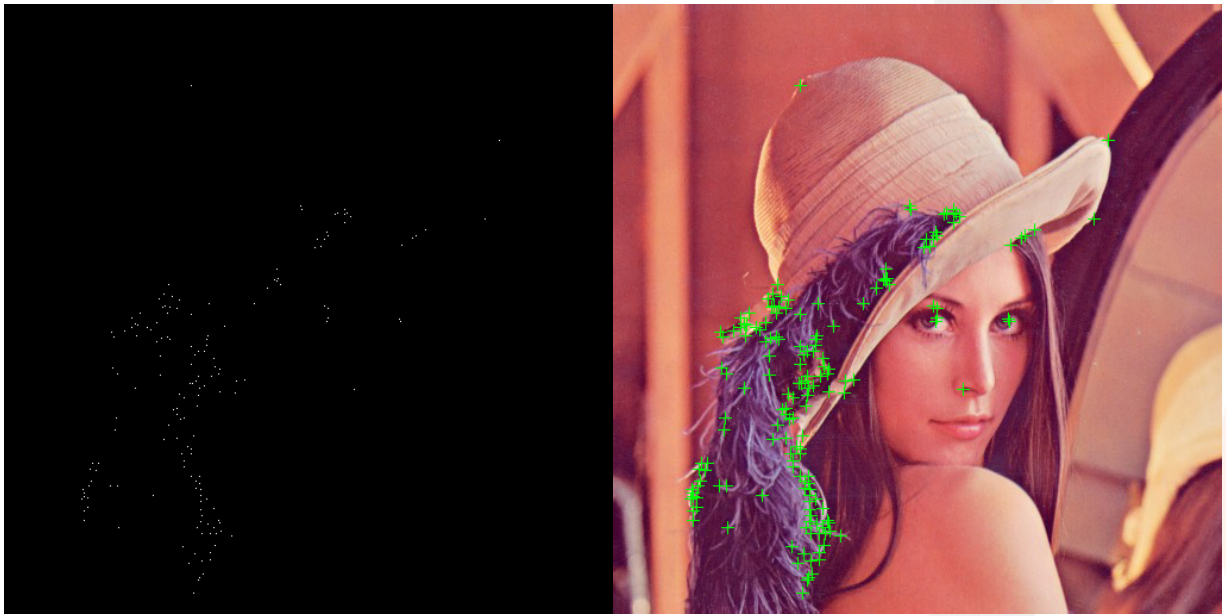


corners

Harris Feature Detector – Implementation

```
// Gets maximum response for each 3x3 neighborhood
af::array mask = constant(1,3,3);
af::array max_resp = dilate(response, mask);

// Discard responses that are not greater than threshold
af::array corners = response > 1e5f;
corners = corners * response;

// Discard responses that are not equal to maximum neighborhood
response,
// scale them to original response value
corners = (corners == max_resp) * corners;
```

Harris Feature Detector – Implementation



corners_nonmax

Harris Feature Detector – Coding Yourself

- Should I? Well...

Harris Feature Detector – Coding Yourself

```cpp
template<typename ty>
__global__ void second_order_deriv(
    ty* ixx_out,
    ty* ixy_out,
    ty* iyy_out,
    const unsigned in_len,
    const ty* ix_in,
    const ty* iy_in)
{
    const unsigned x = blockDim.x * blockIdx.x + threadIdx.x;

    if (x < in_len) {
        ixx_out[x] = ix_in[x] * ix_in[x];
        ixy_out[x] = ix_in[x] * iy_in[x];
        iyy_out[x] = iy_in[x] * iy_in[x];
    }
}

template<typename ty>
__global__ void keep_corners(
    float* x_out,
    float* y_out,
    float* resp_out,
    const float* x_in,
    const float* y_in,
    const float* resp_in,
    const float* resp_idx,
    const unsigned n_corners)
{
    const unsigned f = blockDim.x * blockIdx.x + threadIdx.x;

    // Keep only the first n_feat features
    if (f < n_corners) {
        x_out[f] = x_in[(unsigned)resp_idx[f]];
        y_out[f] = y_in[(unsigned)resp_idx[f]];
        resp_out[f] = resp_in[f];
    }
}

template<typename ty>
__global__ void harris_responses(
    ty* resp_out,
    const unsigned idim0,
    const unsigned idim1,
    const ty* ixx_in,
    const ty* ixy_in,
    const ty* iyy_in,
    const float k_thr,
    const unsigned border_len)
{
    const unsigned r = border_len;

    const unsigned x = blockDim.x * blockIdx.x + threadIdx.x + r;
    const unsigned y = blockDim.y * blockIdx.y + threadIdx.y + r;

    if (x < idim1 - r && y < idim0 - r) {
        const unsigned idx = x * idim0 + y;

        // Calculates matrix trace and determinant
        ty tr = ixx_in[idx] + iyy_in[idx];
        ty det = ixx_in[idx] * iyy_in[idx] - ixy_in[idx] * ixy_in[idx];

        // Calculates local Harris response
        resp_out[idx] = det - k_thr * (tr*tr);
    }
}
```

```cpp
template<typename ty>
__global__ void non_maximal(
    float* x_out,
    float* y_out,
    float* resp_out,
    unsigned* count,
    const unsigned idim0,
    const unsigned idim1,
    const ty* resp_in,
    const float min_resp,
    const unsigned border_len,
    const unsigned max_corners)
{
    // Responses on the border don't have 8-neighbors to compare, discard them
    const unsigned r = border_len + 1;

    const unsigned x = blockDim.x * blockIdx.x + threadIdx.x + r;
    const unsigned y = blockDim.y * blockIdx.y + threadIdx.y + r;

    if (x < idim1 - r && y < idim0 - r) {
        const ty v = resp_in[x * idim0 + y];

        // Find maximum neighborhood response
        ty max_v;
        max_v = max_val(resp_in[(x-1) * idim0 + y-1], resp_in[x * idim0 + y-1]);
        max_v = max_val(max_v, resp_in[(x+1) * idim0 + y-1]);
        max_v = max_val(max_v, resp_in[(x-1) * idim0 + y  ]);
        max_v = max_val(max_v, resp_in[(x+1) * idim0 + y  ]);
        max_v = max_val(max_v, resp_in[(x-1) * idim0 + y+1]);
        max_v = max_val(max_v, resp_in[(x)   * idim0 + y+1]);
        max_v = max_val(max_v, resp_in[(x+1) * idim0 + y+1]);

        // Stores corner to {x,y,resp}_out if it's response is maximum compared
        // to its 8-neighborhood and greater or equal minimum response
        if (v > max_v && v >= (ty)min_resp) {
            unsigned idx = atomicAdd(count, 1u);
            if (idx < max_corners) {
                x_out[idx]    = (float)x;
                y_out[idx]    = (float)y;
                resp_out[idx] = (float)v;
            }
        }
    }
}

template<typename ty>
af_error af_harris_corner_detector(unsigned* corners_out,
                                   float** x_out,
                                   float** y_out,
                                   float** resp_out,
                                   const unsigned idim0,
                                   const unsigned idim1,
                                   const ty* d_in,
                                   const unsigned max_corners=500,
                                   const float min_response=1e5f,
                                   const float sigma=1.f,
                                   const unsigned block_size=0,
                                   const float k_thr=0.04f)
{
    const unsigned img_len = idim0 * idim1;
    unsigned in_dims[] = {idim0, idim1, 1};

    // Window filter
    ty* h_filter = NULL;
    unsigned filter_len;
    // Decide between rectangular or circular filter
    if (block_size > 0) {
```

```cpp
        filter_len = block_size;
        h_filter = (ty*)malloc(filter_len * sizeof(ty));
        for (unsigned i = 0; i < filter_len; i++)
            h_filter[i] = (ty)1.f / (filter_len);
    }
    else {
        // 6 sigma rule to determine kernel length
        filter_len = floor(6.f * sigma);

        // if filter length is even, round it down to nearest odd integer
        if (filter_len % 2 == 0)
            filter_len--;

        h_filter = (ty*)malloc(filter_len * sizeof(ty));

        gaussian1D<ty>(h_filter, filter_len, 2.f);
    }

    unsigned filter_dims_v[] = {filter_len, 1, 1};
    unsigned filter_dims_h[] = {1, filter_len, 1};

    // +1 each side for gradient (first-order derivative)
    const unsigned border_len = filter_len / 2 + 1;

    // +1 each side for non-maximal
    const unsigned min_side = border_len + 2;
    if (idim0 < min_side || idim1 < min_side)
        THROW("harris_corner_detector: image dimensions must be >= %d for the current
sigma/block_size",
              min_side);

    ty* d_ix = NULL;
    ty* d_iy = NULL;
    AF(af_malloc((void **)&d_ix, img_len * sizeof(ty)));
    AF(af_malloc((void **)&d_iy, img_len * sizeof(ty)));

    // Compute first-order derivatives as gradients
    AF(af_grad(d_iy, d_ix, idim0, idim1, d_in));

    ty* d_ixx = NULL;
    ty* d_ixy = NULL;
    ty* d_iyy = NULL;
    AF(af_malloc((void **)&d_ixx, img_len * sizeof(ty)));
    AF(af_malloc((void **)&d_ixy, img_len * sizeof(ty)));
    AF(af_malloc((void **)&d_iyy, img_len * sizeof(ty)));

    // Compute second-order derivatives
    dim3 threads(LINE_SIZE, 1);
    dim3 blocks(divup(img_len, threads.x), 1);
    second_order_deriv<ty><<<blocks, threads>>>(d_ixx, d_ixy, d_iyy,
                                                img_len, d_ix, d_iy);

    AF(af_free(d_ix));
    AF(af_free(d_iy));

    ty* d_ixx_tmp = NULL;
    ty* d_ixy_tmp = NULL;
    ty* d_iyy_tmp = NULL;
    AF(af_malloc((void **)&d_ixx_tmp, img_len * sizeof(ty)));
    AF(af_malloc((void **)&d_ixy_tmp, img_len * sizeof(ty)));
    AF(af_malloc((void **)&d_iyy_tmp, img_len * sizeof(ty)));

    // Separable filtering
    AF(af_convh(d_ixx_tmp, 2, in_dims, d_ixx, 1, 2, filter_dims_v, h_filter, 1, false
));
    AF(af_convh(d_ixx, 2, in_dims, d_ixx_tmp, 1, 2, filter_dims_h, h_filter, 1, false
));
```

Harris Feature Detector – Coding Yourself

```
199     AF(af_convh(d_ixy_tmp, 2, in_dims, d_ixy, 1, 2, filter_dims_v, h_filter, 1, false));
200     AF(af_convh(d_ixy, 2, in_dims, d_ixy_tmp, 1, 2, filter_dims_h, h_filter, 1, false));
201     AF(af_convh(d_iyy_tmp, 2, in_dims, d_iyy, 1, 2, filter_dims_v, h_filter, 1, false));
202     AF(af_convh(d_iyy, 2, in_dims, d_iyy_tmp, 1, 2, filter_dims_h, h_filter, 1, false));
203
204     AF(af_free(d_ixx_tmp));
205     AF(af_free(d_ixy_tmp));
206     AF(af_free(d_iyy_tmp));
207
208     unsigned corner_lim = img_len * 0.2f;
209
210     unsigned* d_corners_found = NULL;
211     AF(af_malloc((void **)&d_corners_found, sizeof(unsigned)));
212     CUDA(cudaMemset(d_corners_found, 0, sizeof(unsigned)));
213
214     float* d_x_corners = NULL;
215     float* d_y_corners = NULL;
216     float* d_resp_corners = NULL;
217     AF(af_malloc((void **)&d_x_corners, corner_lim * sizeof(float)));
218     AF(af_malloc((void **)&d_y_corners, corner_lim * sizeof(float)));
219     AF(af_malloc((void **)&d_resp_corners, corner_lim * sizeof(float)));
220
221     ty* d_responses = NULL;
222     AF(af_malloc((void **)&d_responses, img_len * sizeof(ty)));
223
224     // Calculate Harris responses for all pixels
225     threads = dim3(BLOCK_SIZE, BLOCK_SIZE);
226     blocks = dim3(divup(idim1 - border_len*2, threads.x),
227                   divup(idim0 - border_len*2, threads.y));
228     harris_responses<ty><<<blocks, threads>>>(d_responses,
229                                               idim0, idim1,
230                                               d_ixx, d_ixy, d_iyy,
231                                               k_thr, border_len);
232
233     AF(af_free(d_ixx));
234     AF(af_free(d_ixy));
235     AF(af_free(d_iyy));
236
237     const float min_r = (max_corners > 0) ? 0.f : min_response;
238
239     // Perform non-maximal suppression
240     threads = dim3(BLOCK_SIZE, BLOCK_SIZE);
241     blocks = dim3(divup(idim1 - border_len*2, threads.x),
242                   divup(idim0 - border_len*2, threads.y));
243     non_maximal<ty><<<blocks, threads>>>(d_x_corners, d_y_corners,
244                                          d_resp_corners, d_corners_found,
245                                          idim0, idim1, d_responses,
246                                          min_r, border_len, corner_lim);
247
248     unsigned corners_found;
249     CUDA(cudaMemcpyD2H(&corners_found, d_corners_found, sizeof(unsigned)));
250
251     AF(af_free(d_responses));
252     AF(af_free(d_corners_found));
253
254     if (max_corners > 0) {
255         float* d_resp_sorted;
256         float* d_resp_idx;
257         AF(af_malloc((void **)&d_resp_sorted, corners_found * sizeof(float)));
258         AF(af_malloc((void **)&d_resp_idx, corners_found * sizeof(float)));
259
260         // Sort Harris responses
261         AF(af_sort_S(d_resp_sorted, d_resp_idx, 1, &corners_found, d_resp_corners,
262                      true, 0, 1, false));
263
264         AF(af_free(d_resp_corners));
265
```

```
266         const unsigned corners_to_keep = min(corners_found, max_corners);
267
268         AF(af_malloc((void **)x_out, corners_to_keep * sizeof(float)));
269         AF(af_malloc((void **)y_out, corners_to_keep * sizeof(float)));
270         AF(af_malloc((void **)resp_out, corners_to_keep * sizeof(float)));
271
272         // Keep only the first corners_to_keep corners with higher Harris
273         // responses
274         threads = dim3(LINE_SIZE, 1);
275         blocks = dim3(divup(corners_to_keep, threads.x), 1);
276         keep_corners<ty><<<blocks, threads>>>(*x_out, *y_out, *resp_out,
277                                               d_x_corners, d_y_corners,
278                                               d_resp_sorted, d_resp_idx,
279                                               corners_to_keep);
280
281         *corners_out = corners_to_keep;
282
283         AF(af_free(d_x_corners));
284         AF(af_free(d_y_corners));
285         AF(af_free(d_resp_sorted));
286         AF(af_free(d_resp_idx));
287     }
288     else {
289         *x_out = d_x_corners;
290         *y_out = d_y_corners;
291         *resp_out = d_resp_corners;
292
293         *corners_out = min(corners_found, corner_lim);
294     }
295
296     return AF_SUCCESS;
297 }
```

Harris Feature Detector – Coding Yourself

```
static void harris_demo(string imgFile)
{
    array img = loadImage(imgFil, false);

    array ix, iy;
    grad(ix, iy, img);

    // Compute second-order derivatives
    array ixx = ix * ix;
    array ixy = ix * iy;
    array iyy = iy * iy;

    // Compute a Gaussian kernel with standard deviation of 1.0 and length of 5 pixels
    // These values can be changed to use a smaller or larger window
    array gauss_filt = gaussiankernel(5, 5, 1.0, 1.0);

    // Filter second-order derivatives with Gaussian kernel computed previously
    ixx = convolve(ixx, gauss_filt);
    ixy = convolve(ixy, gauss_filt);
    iyy = convolve(iyy, gauss_filt);

    // Calculate trace
    array tr = ixx + iyy;
    // Calculate determinant
    array det = ixx * iyy - ixy * ixy;

    // Calculate Harris response
    array response = det - 0.04f * (tr * tr);

    // Gets maximum response for each 3x3 neighborhood
    //array max_resp = maxfilt(response, 3, 3);
    array mask = constant(1,3,3);
    array max_resp = dilate(response, mask);

    // Discard responses that are not greater than threshold
    array corners = response > 1e5f;
    corners = corners * response;

    // Discard responses that are not equal to maximum neighborhood response,
    // scale them to original response value
    corners = (corners == max_resp) * corners;
}
```

Harris Feature Detector – ArrayFire

Even easier:

```
af::array img = loadimage("image.png");
af::features f;
f = af::harris(img);
```

FAST – High-Speed Corner Detection

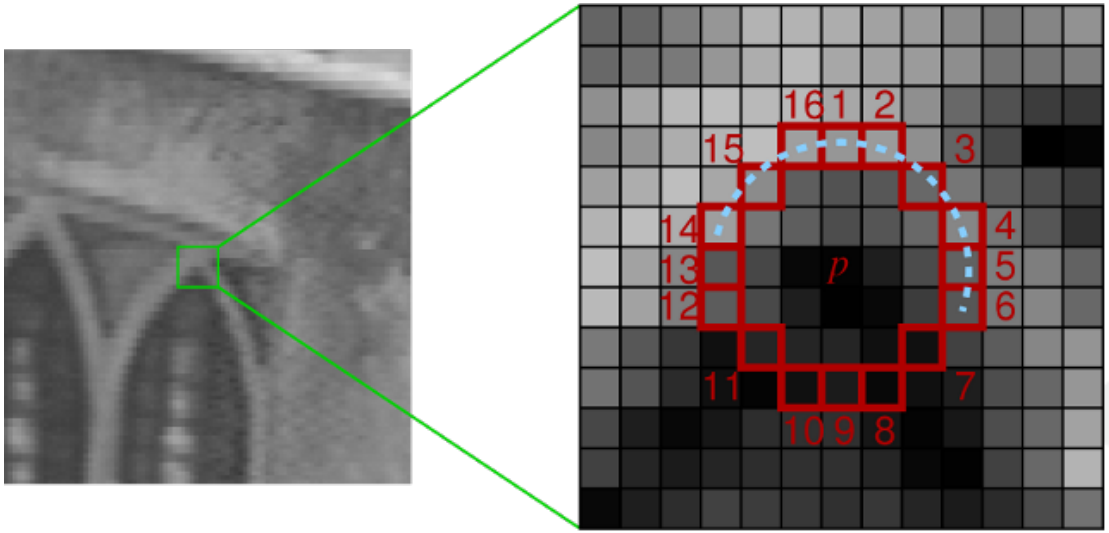


Image source: Rosten, Edward, and Tom Drummond. "Machine learning for high-speed corner detection." *Computer Vision–ECCV 2006*. Springer Berlin Heidelberg, 2006. 430-443.

This is "FAST" because the number of comparisons is pruned (explained in the next slides)

FAST – High-Speed Corner Detection

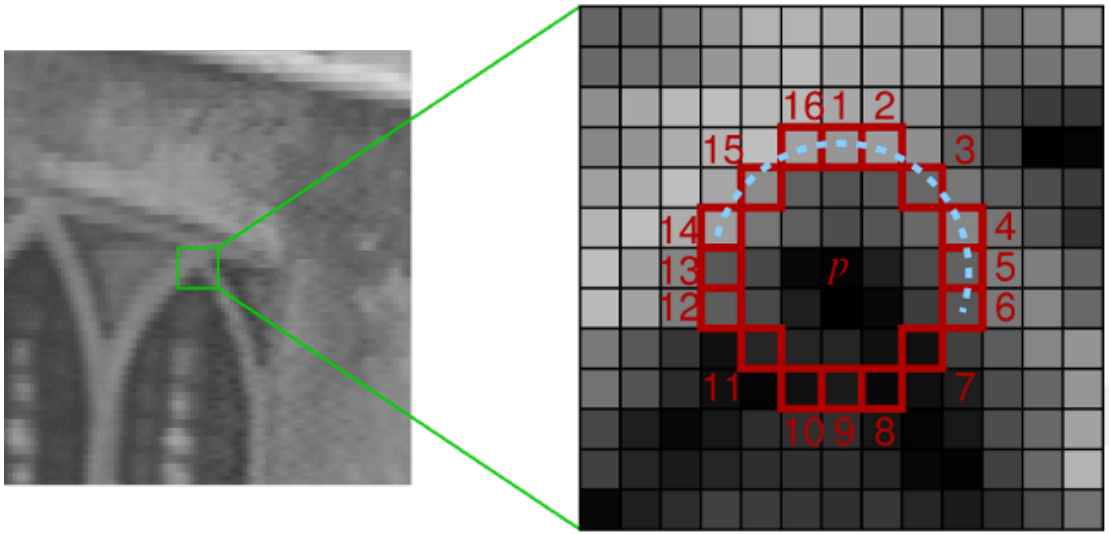
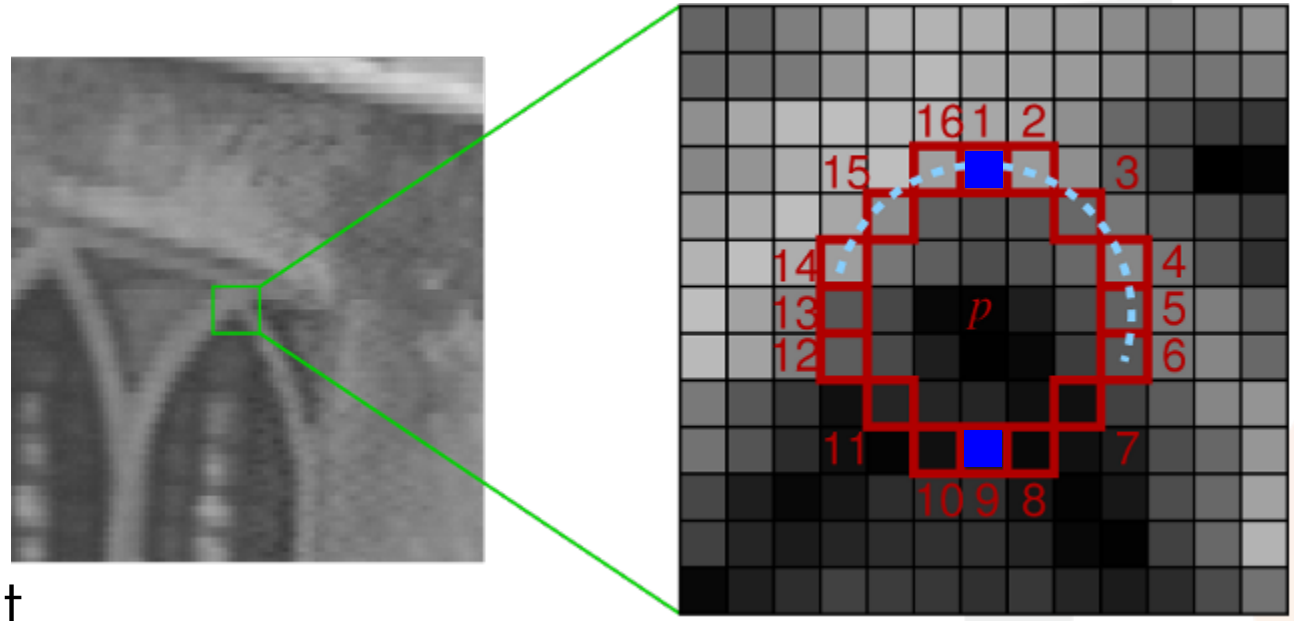


Image source: Rosten, Edward, and Tom Drummond. "Machine learning for high-speed corner detection." *Computer Vision–ECCV 2006*. Springer Berlin Heidelberg, 2006. 430-443.

$p > I_p - t$

$p < I_p + t$ - Arc pixels must match one condition

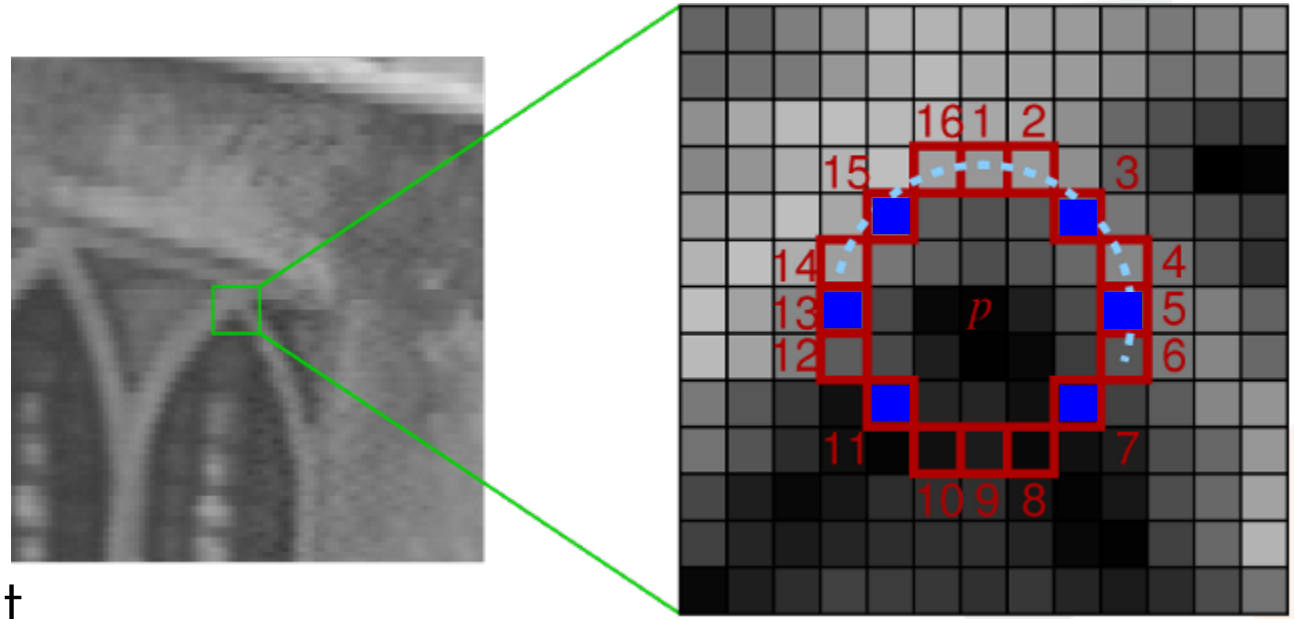
FAST – High-Speed Test 1



$p > I_p - t$

$p < I_p + t$ - Discard if pixels don't match condition

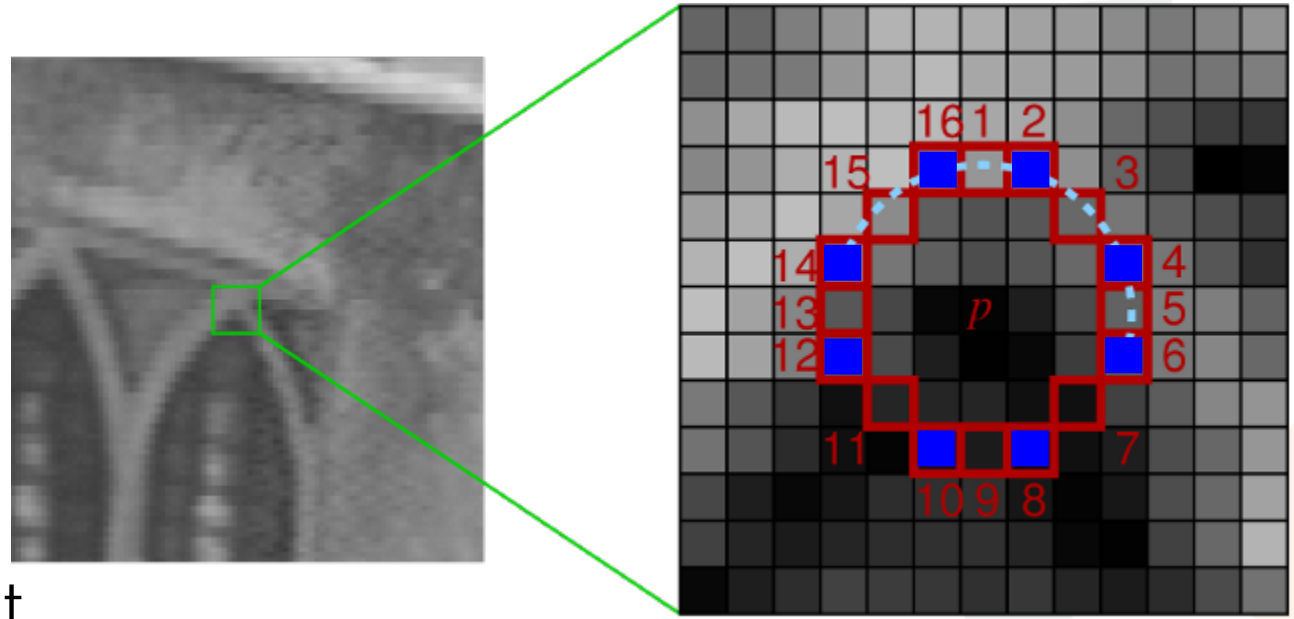
FAST – High-Speed Test 2



$p > I_p - t$

$p < I_p + t$ - Discard if pixels don't match condition

FAST – High-Speed Test 3



$p > I_p - t$

$p < I_p + t$ - Discard if pixels don't match condition

Parallel FAST

- Each block contains HxV threads
  - H - Number of "horizontal" threads
  - V - Number of "vertical" threads
- Block will read from shared memory, (H+r+r)x(V+r+r) pixels, where r is the radius (3 for 16 pixel ring)

Parallel FAST (Cont.)

- Avoid using “if” statements - due to branch divergence
- Entire blocks are discarded after high-speed test (good “if” condition usage!)

Parallel FAST (Cont.)

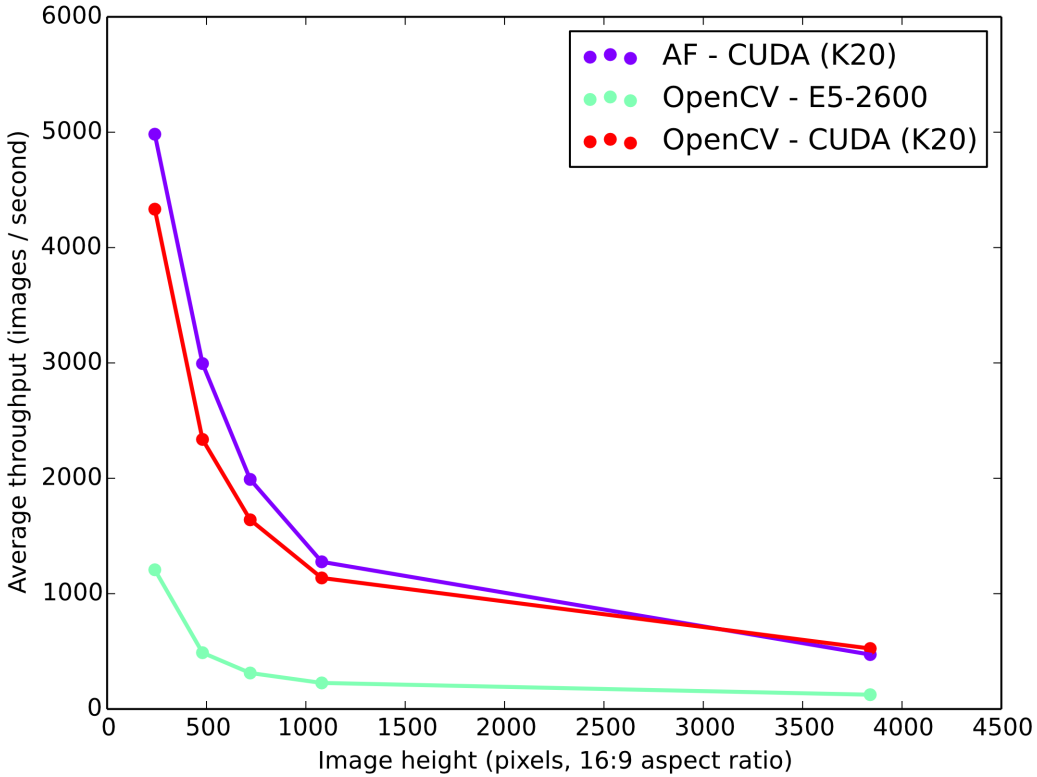
- Calculate a binary string (16 pixel ring = 16 bits) for each of $p > I_p - t$ and $p < I_p + t$ conditions
- Generate a Look-Up Table containing the maximum length of a segment (2^{16} = 65,536 conditions)
- Check the LUT for the existence of a segment of desired length

Parallel FAST - ArrayFire

Even easier:

```
af::array img = loadimage("image.png");
af::features f;
f = af::fast(img);
```

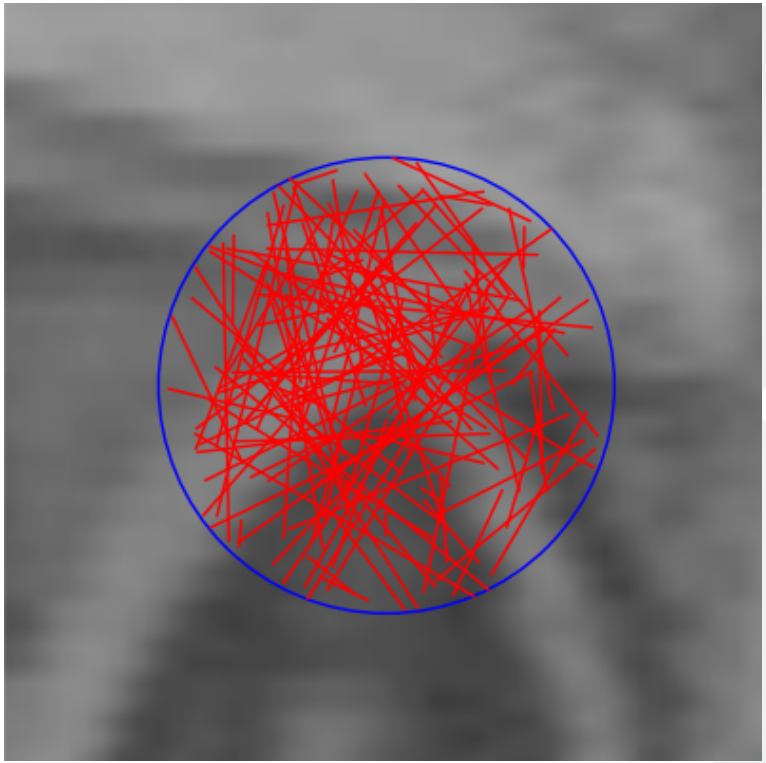
FAST performance: ArrayFire vs. OpenCV



BRIEF – Binary Robust Independent Elementary Features

- Pair-wise intensity comparisons
- Pairs sampled from Gaussian isotropic distribution
- Descriptor is a binary vector
- Fast comparison (Hamming distance)

BRIEF – Binary Robust Independent Elementary Features



FAST + BRIEF - Issues

- Rotation
- Scale

ORB – Oriented FAST and Rotated BRIEF

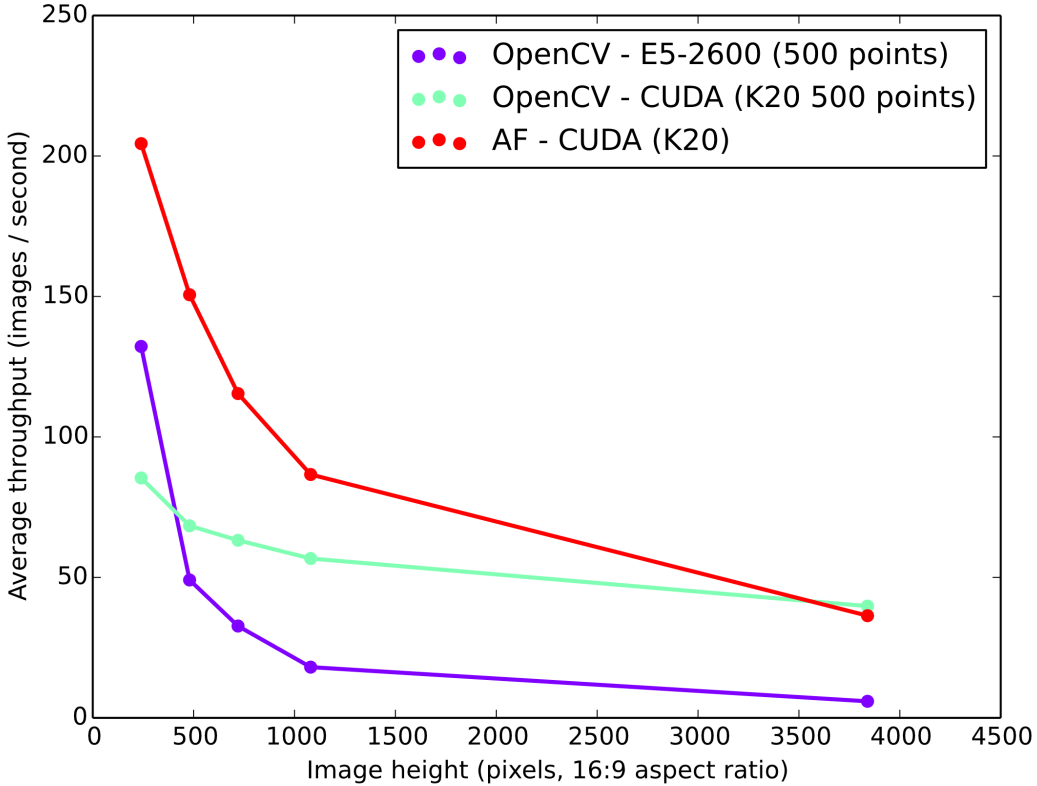
- Detects FAST features in multiple scales
- Calculates feature orientation using intensity centroid
- Extract oriented BRIEF descriptor

Parallel ORB – ArrayFire

Even easier:

```
af::array img = loadimage("image.png");
af::features f;
af::array desc;
af::orb(f, desc, img);
```

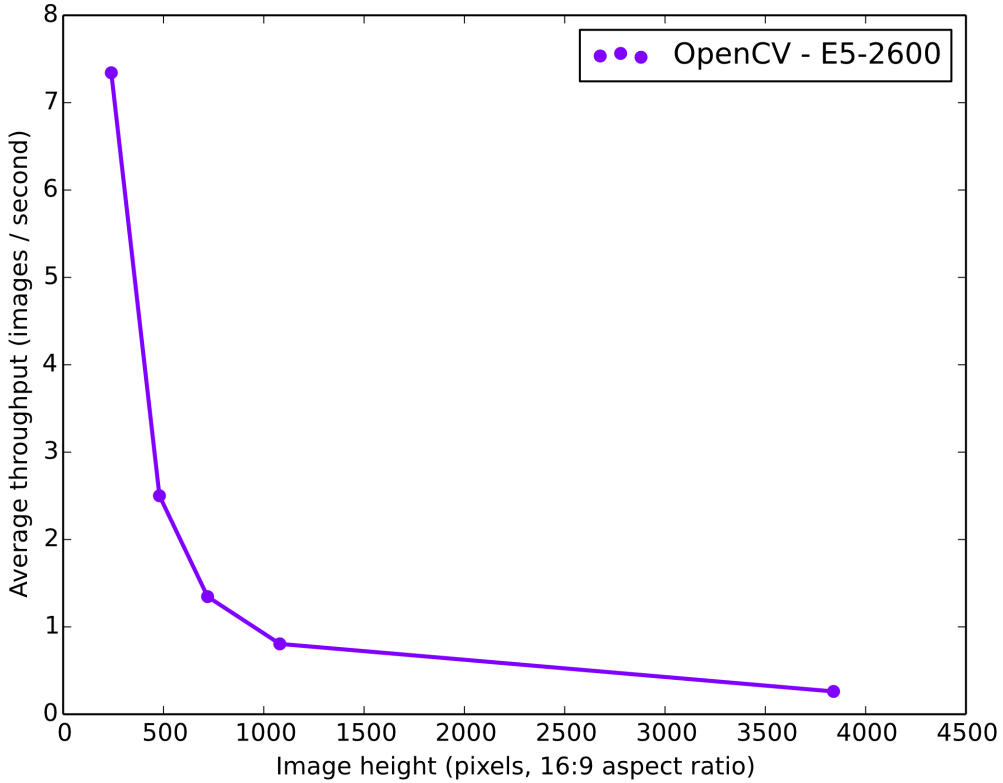
ORB performance: ArrayFire vs. OpenCV



Other Feature Detectors/Extractors

```
af::array img = loadimage("image.png");
af::features f;
af::array desc;
af::sift(f, desc, img); // Coming Soon!
af::surf(f, desc, img); // Coming Soon!
```

SIFT performance: OpenCV



SURF performance: OpenCV