GPU-Accelerated Symmetry Transform for Object Keypoints

Jason Owens

January 8, 2017

1 Introduction

Finding salient object candidates in arbitrary natural images is a problem in computer vision and robotic perception that has yet to be solved. There are many ways this challenge is currently approached:

- scan the entire image, at multiple scales, with some kind of object recognition algorithm, generating a heat map indicating the score or probability of a known, detected object (i.e. sliding window),
- propose a smaller set of likely object regions based on some engineered or learned features of the image (e.g. edge boxes, geodesic object proposals, objectness),
- compute a segmentation of the image to propose object-like regions,
- learn a saliency function to predict regions of the image that may contain objects.

In recent years and in combination with advanced deep learning systems for object recognition tasks, the use of object proposal algorithms has become almost a de-facto standard. As indicated, object proposal algorithms produce a significantly smaller of image regions to test than almost any other mechanism. The main idea is to either engineer or train a detector for what has been called the "objectness" of a region, i.e. how well that region reflects aspects of containing an object. Object aspects often include such properties as closed contours, convexity, and compactness. In the image domain, the contours can often be reflected in the edges derived from the image gradient.

However, processing image contours without any additional information (e.g. learning an object contour prior) often yields undesirable results, with many proposals that do not represent an object. When we discuss objectness, we usually care about whole physical objects we can pick up (box of cereal, phone, pencil, flashlight, coffee mug) and not the aspects of an object's appearance that may also provide strong gradient edge responses, e.g. logos or pictures on a cereal box.

In 1995, Reisfeld et al. proposed the use of a symmetry transform operator in the image domain for use as an attentional operator. Since symmetry is considered a strong indicator of an object with shape [5, 3], it's reasonable to conclude that regions exhibiting strong symmetry are likely to contain objects of interest.

Reisfeld showed that by extracting contours from the symmetry magnitude and selecting local maxima, it was possible to compute attention keypoints that indicated important features of the image. For example, figure 1 shows an example from [6] where selecting the maximum output of the radial symmetry transform produces keypoints for the face of Nixon, the forehead of Elvis, as well as several interesting points on the flags.

Need to discuss Kootstra use of the symmetry in an object saliency framework (plus the limitations).

Like [4]. Extends the 2D appearance-based symmetry into a 3D approach on depth maps, and yields better detection results (check this out).

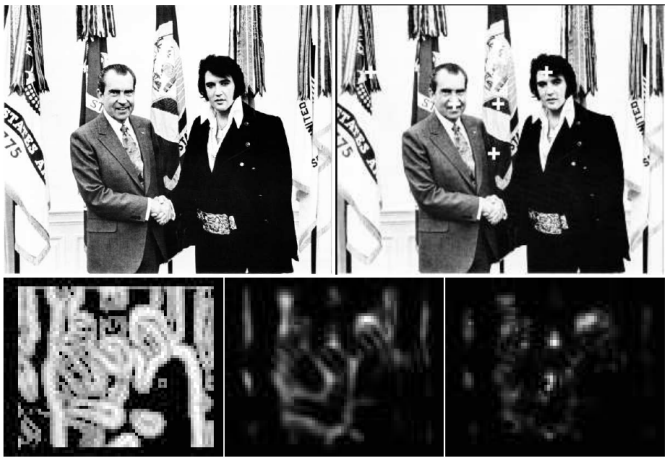

Figure 1: Salient keypoint extraction using symmetry magnitude, from Reisfeld 1995

2 Mathematical Approach

Reisfeld's symmetry transform uses the gradient image to compute symmetry magnitude and direction for every pixel in the image. Given the magnitude image, one can then use non-maximum suppression to select local maxima as salient points in order to direction attentional processing. For convenience, we restate Reisfeld's mathematical formulation to provide background for the computational approach discussed in the next section.

Let $\mathcal{I} : \Omega \to [0, 1]$ be a grayscale image with domain $\Omega \subset \mathbb{Z}^2$. Then $p_k \in \Omega$ represents some pixel coordinate in the image, and $\nabla(p_k) = \left(\frac{\partial}{\partial x}\mathcal{I}(p_k), \frac{\partial}{\partial y}\mathcal{I}(p_k)\right)$ is the gradient at that coordinate. Reisfeld then computes a 2D polar coordinate for each pixel, (r_k, θ_k), where $r_k = \log(1 + \|\nabla(p_k)\|)$, and $\theta_k = \text{atan2}(\frac{\partial}{\partial y}\mathcal{I}(p_k), \frac{\partial}{\partial x}\mathcal{I}(p_k))$. Let l_{ij} be the line passing through two points p_i and p_j, and let α_{ij} be the angle l_{ij} makes with the horizontal (x) axis. For any pixel p_k, define the set $\Gamma(p_k) = \{(i, j) \mid \frac{p_i+p_j}{2} = p_k\}$, i.e. the set of pixel index pairs such that p_k resides on the center of the separating line l. A distance, $D_\sigma(i, j)$, and phase, $P(i, j)$ function are used to determine the contribution $C(i, j)$ for each point pair in $\Gamma(p_k)$, defined as follows:

$$D_\sigma(i,j) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{\|p_i - p_j\|}{2\sigma}} \tag{1}$$

$$P(i,j) = (1 - \cos(\theta_i + \theta_j - 2\alpha_{ij}))\,(1 - \cos(\theta_i - \theta_j)) \tag{2}$$

$$C(i,j) = D_\sigma(i,j)P(i,j)r_i r_j \tag{3}$$

Finally, the *symmetry magnitude* for a point p is defined as:

$$M_\sigma(p) = \sum_{(i,j)\in\Gamma(p)} C(i,j) \tag{4}$$

which simply sums the weighted contributions over the entire "symmetric pixel" neighborhood of p (producing an averaged value). A direction contribution function for each pixel, $\psi(i, j)$, used to compute the *symmetry direction*, $\phi(p)$ for p, is defined as follows:

$$\psi(i,j) = \frac{\theta_i + \theta_j}{2} \tag{5}$$

$$\phi(p) = \psi(i^*, j^*) \quad \text{where} \quad (i^*, j^*) = \underset{(i,j)\in\Gamma(p)}{\text{argmax}}\, C(i,j) \tag{6}$$

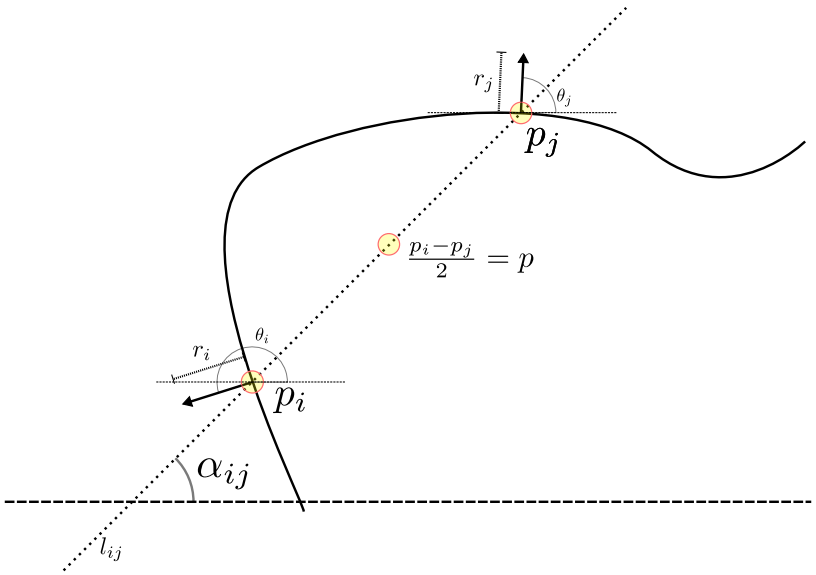

Figure 2: Illustration of the geometry and quantities involved in computing pixel pair contributions for the symmetry around point p.

We can then define the overall output of the symmetry transform as:

$$S_\sigma(p) = \{M_\sigma(p), \phi(p)\}. \tag{7}$$

Please see figure 2 for an illustration indicating the values involved in the transform.

Note that in the original paper, Reisfeld justifies using the logarithm of the gradient magnitude in order to reduce the contribution of stronger gradients and makes the correlation measure ($C(i,j)$) less sensitive to stronger edges. Also note that the 2D Gaussian function used in the distance function is circular; Reisfeld points out that this can be modified to emphasize elliptical features. In addition, he defines a modified symmetry magnitude he calls *radial symmetry* that emphasizes symmetries that are perpendicular to the primary symmetry direction (i.e. $\phi(p)$:

$$RS_\sigma(p) = \sum_{(i,j)\in\Gamma(p)} C(i,j)\sin^2\left(\psi(i,j) - \phi(i,j)\right). \tag{8}$$

An important aspect of this function is to note that the $M_\sigma(p)$ value must already be computed (and $C(i,j)$ computed twice, or otherwise cached), by virtue of the use of $\phi(i,j)$, since it is a function of the entire neighborhood $\Gamma(p)$.

3 Computational Approach

From the mathematical definition of the symmetry transform, we can see that there are no mutual data dependencies between pixels given the gradient image; in other words, the problem is embarrassingly parallel. Each pixel *does* depend on a neighborhood (defined both by equation 1 and $\Gamma(\cdot)$), but $M_\sigma(p)$ does not need values computed by any other neighboring pixel. We can therefore compute the symmetry transform for each pixel independently, which suggests that an adaptation of the algorithm for GPU computation should be relatively straightforward.

In this section, we present the basic sequential algorithm (with no optimizations), and then discuss how it was readily adapted for computation on a GPU.

3.1 Sequential Algorithm

Algorithm 1 represents the pseudocode for the symmetry transform, which accepts the minimum radius σ, the gradient magnitude image g_m, and the gradient direction image g_θ, where $\sigma \in \mathbb{Z}$, $g_m : \Omega \to \mathbb{R}$ and $g_\theta : \Omega \to \mathbb{R}$.

There are several specific aspects we highlight in this formulation. While Reisfeld leaves the definition of $\Gamma(p)$ open to *all* symmetric points surrounding p, the effects of points further away are clearly limited by the distance function $D_\sigma(i, j)$ (i.e. eq 1). In this code, we explicitly limit the bounds of the per-pixel neighbor iteration to a square region defined by the $p_{\min}$ and $p_{\max}$, with sides equal to $2\rho = 5\sigma$, see lines 10-11. This follows the implementation details given by Kootstra et al. in [2], where they use default min/max radius values of 7 and 17, respectively. In this implementation, σ is given as a parameter to the function to control the scale, and we compute the max radius as ρ directly from σ. Also, note that the argmax is implemented inline in lines 25-28 to avoid running another loop to compute the maximum. Finally, we only have to process *half* the neighborhood region, since by definition the other half of the points are symmetric to the first half; line 15 shows where this early termination occurs.

3.2 Parallel Adaptation

To adapt algorithm 1 for the GPU, we simply extract lines 6-33 and convert them to an appropriate GPU kernel function (see the code listing in Algorithm 2). In our implementation, we use the CUDA language for NVIDIA GPUs (the most common discrete GPU in our environment). We do not claim any particular ingenuity in converting this problem to a parallel implementation; we simply document the implementation and show the performance benefits.

Note that it is impossible to get full utilization of the thread warps near the edges of the image, since some threads will be sitting idle due to the `valid\_pt` checks ensuring we don't process pixels outside the image bounds. Also note in lines 30-31 that we apply the modification from [2] that ignores the gradients near the center of the point, to help emphasize the gradients at the given radius (σ); this produces a "no computation zone" that causes portions of thread warps to become idle, due to the SIMD nature of the CUDA computation model.

The kernel is then called with a default block size of 16×16. No optimizations have yet been implemented for device-dependent occupancies or shared memory usage. Both adjustments could improve overall performance, but require additional complexity in the kernel and the host calling function.

4 Results

4.1 Performance

The performance difference between single-CPU and GPU implementations is staggering. As Figure **??** shows using a logarithmic-scale y axis, there are two orders of magnitude improvement in execution time for the GPU version, primarily due to the massive parallelism on the GPU and the virtually non-existent dependencies between pixel values (i.e. no reductions necessary, and no waiting), even with idle threads due to the internal "no computation zone."

4.2 Transform Output

For our purposes, we are much less interested in the symmetry direction as we are in the symmetry magnitude (as discussed in the introduction). Figure 4 shows example output from the implemented symmetry transform as well as simple "feature" detection output derived from the symmetry magnitude images. A quick and dirty method for finding features is to use non-maximum suppression to find local maxima (and then suppress any other maxima within a given radius). To show the usefulness of the transform on natural images, we implemented a simple detector that computes an image pyramid, runs

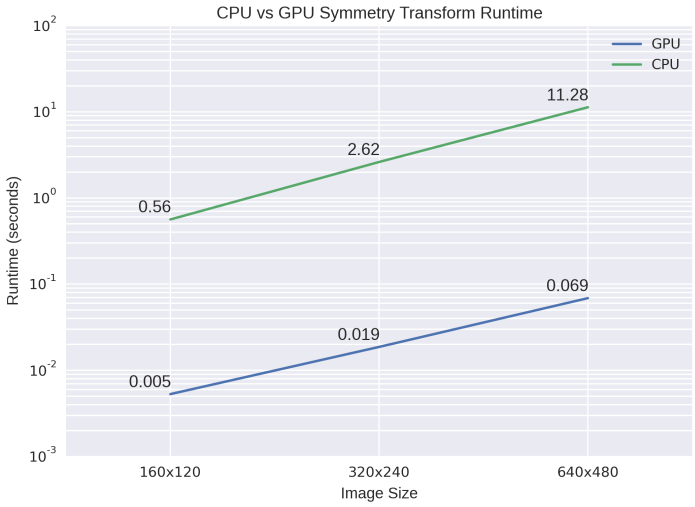

Figure 3: Single-CPU vs GPU performance comparison. Note that there is two orders of magnitude improvement between the GPU and the CPU runtime. Tests were performed on a 4th-Gen Core i7 2.9 GHz processor and NVidia Quadro M3000M GPU with 768 CUDA cores.

the symmetry transform on each layer, and then merges the result into a single full-scale magnitude image in order to find local maxima. In the results shown, our detector uses a suppression radius of 15 pixels.

Evident in the images...

5 Future work

6 Conclusion

References

[1] Peter Henry et al. "RGB-D Mapping: Using Kinect-Style Depth Cameras for Dense 3D Modeling of Indoor Environments". In: *The International Journal of Robotics Research* 31.5 (Apr. 1, 2012), pp. 647–663. ISSN: 0278-3649, 1741-3176. DOI: 10.1177/0278364911434148. URL: http://ijr.sagepub.com/cgi/content/abstract/31/5/647.

[2] Gert Kootstra, Niklas Bergstrom, and Danica Kragic. "Using Symmetry to Select Fixation Points for Segmentation". In: *Pattern Recognition (ICPR), 2010 20th International Conference on*. IEEE, 2010, pp. 3894–3897. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5597580 (visited on 10/26/2016).

[3] Yunfeng Li et al. "Symmetry Is the Sine qua Non of Shape". In: *Shape Perception in Human and Computer Vision*. Ed. by Sven J. Dickinson and Zygmunt Pizlo. London: Springer London, 2013, pp. 21–40. ISBN: 978-1-4471-5194-4 978-1-4471-5195-1. URL: http://link.springer.com/10.1007/978-1-4471-5195-1_2 (visited on 12/04/2016).

[4] Ekaterina Potapova, Michael Zillich, and Markus Vincze. "Local 3d Symmetry for Visual Saliency in 2.5 D Point Clouds". In: *Asian Conference on Computer Vision*. Springer, 2012, pp. 434–445. URL: http://link.springer.com/chapter/10.1007/978-3-642-37331-2_33 (visited on 12/04/2016).

[5] D. Reisfeld and Y. Yeshurun. "Robust Detection of Facial Features by Generalized Symmetry". In: IEEE Comput. Soc. Press, 1992, pp. 117–120. ISBN: 978-0-8186-2910-5. DOI: 10.1109/ICPR.1992.201521. URL: http://ieeexplore.ieee.org/document/201521/ (visited on 11/15/2016).

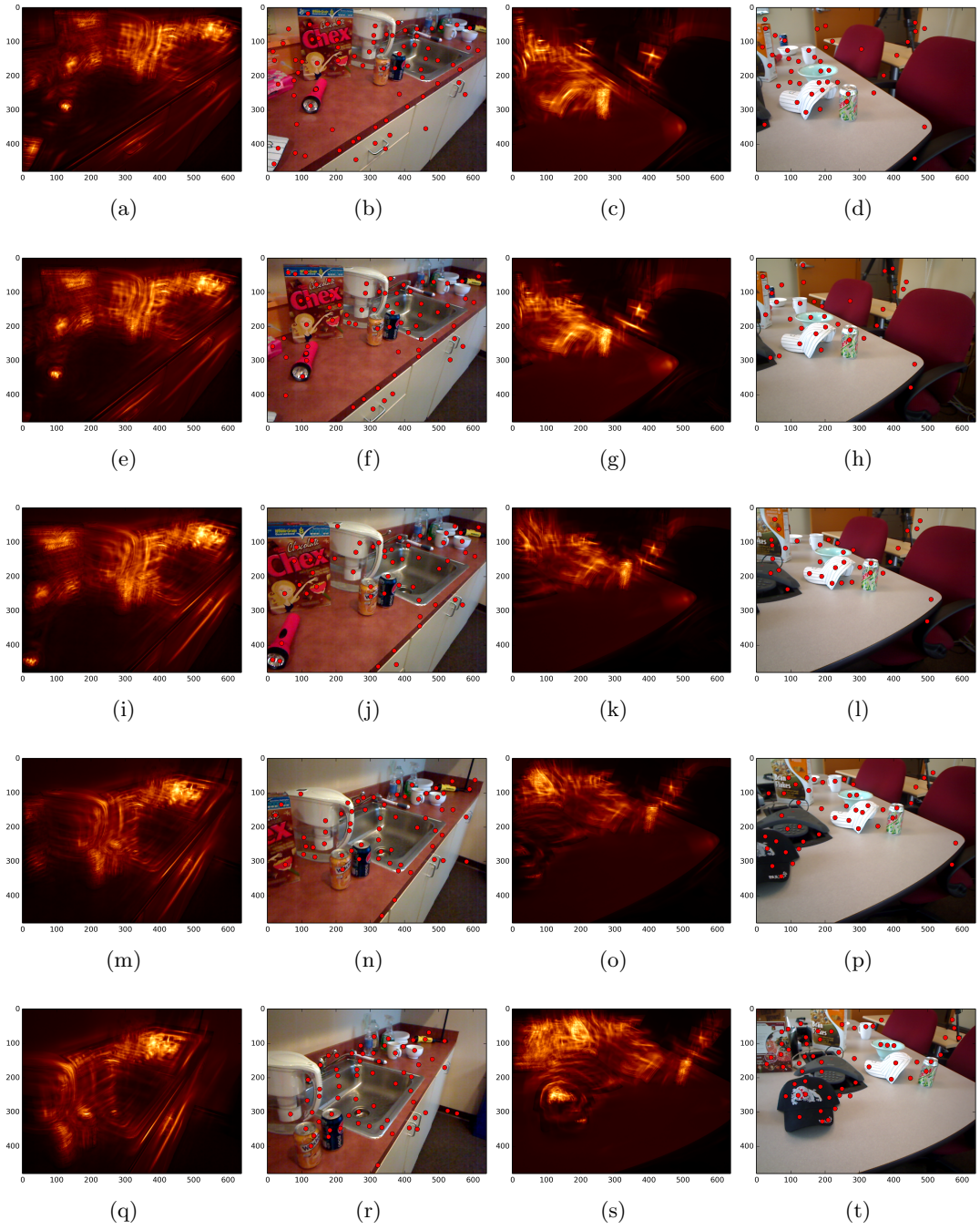

Figure 4: Five consecutive images each of the =kitchen_small= (left pair) and =meeting_small= (right pair) scenes from the rgbd_scenes dataset [1]. For each pair, the computed symmetry magnitude is shown on the left, and the local maxima found using non-maximum suppression is shown on the left, using radius 15.

[6] Daniel Reisfeld, Haim Wolfson, and Yehezkel Yeshurun. “Context Free Attentional Operators: The Generalized Symmetry Transform”. In: *Int. J. Comput. Vision* 14 (1995), pp. 119–130.

Algorithm 1 Symmetry Transform

function SYMMETRY(σ, g_m, g_θ) ▷ Symmetry transform with radius σ for $\mathcal{I}$ with gradient magnitude g_m and gradient direction g_θ.
  $\rho \leftarrow 2.5\sigma$
  $S_m, S_\theta \leftarrow$ new arrays of $\mathbb{R}$ compatible with $\mathcal{I}$
  for $y \leftarrow 0, \text{rows}(\mathcal{I})$ **do** ▷ Iterate over all pixels in Ω
    for $x \leftarrow 0, \text{cols}(\mathcal{I})$ **do**
      $M, C_{ij}, \psi_{ij}, C_{\max}, \phi_p, \alpha_{ij} \leftarrow 0$
      $p \leftarrow [x, y]^\mathsf{T}$
      $p_{\min} \leftarrow p - \rho$
      $p_{\max} \leftarrow p + \rho$
      for $j \leftarrow p_{\min}[y], p_{\max}[y]$ **do** ▷ Iterate over all pixel indices in the square $(p_{\min}, p_{\max})$
        for $i \leftarrow p_{\min}[x], p_{\max}[x]$ **do**
          $p_i \leftarrow [i, j]^\mathsf{T}$
          $p_j \leftarrow p - (p_i - p)$ ▷ Compute the mirror point
          if $p_i = p$ **then**
            terminate neighborhood loop ▷ All remaining p_i, p_j pixel pairs are symmetric
          end if
          if valid_pt(p_i) $\wedge$ valid_pt(p_j) **then**
            $r_i, \theta_i \leftarrow \text{pt\_gradient}(g_m, g_\theta, p_i)$
            $r_j, \theta_j \leftarrow \text{pt\_gradient}(g_m, g_\theta, p_j)$
            $\delta_{ij} \leftarrow p_j - p_i$
            $\alpha_{ij} \leftarrow \text{atan2}(\delta_{ij}[y], \delta_{ij}[x])$
            $C_{ij} \leftarrow r_i r_j \, \text{D}(i, j, \sigma) \, \text{P}(\alpha_{ij}, i, j)$
            $M \leftarrow M + C_{ij}$
            $\psi_{ij} \leftarrow \frac{(\theta_i + \theta_j)}{2}$
            if $C_{ij} > C_{\max}$ **then**
              $C_{\max} \leftarrow C_{ij}$
              $\phi_p \leftarrow \psi_{ij}$
            end if
          end if
        end for
      end for
      $S_m(p) \leftarrow M$
      $S_\theta(p) \leftarrow \phi_p$
    end for
  end for
  return S_m, S_θ
end function

Algorithm 2 Parallel Cuda Kernel

```
__global__ void cu_symmetry(cv::gpu::PtrStepSzf smag,
                            cv::gpu::PtrStepSzf sdir,
                            const cv::gpu::PtrStepSzf mag,
                            const cv::gpu::PtrStepSzf dir,
                            int sigma)
{
  // prepare region for given pixel x,y
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  int radius = (int)(2.5 * sigma);
  int ROWS = mag.rows;
  int COLS = mag.cols;

  // target point, we are computing the symmetry for THIS point
  int2 p = make_int2(x, y);
  if (p.x >= COLS || p.y >= ROWS) return;

  int2 min = make_int2(p.x - radius, p.y - radius);
  int2 max = make_int2(p.x + radius + 1, p.y + radius + 1);
  int2 pi, pj, d;
  float2 rtheta_i, rtheta_j;
  float M = 0;
  float C_ij = 0;
  float psi_ij = 0;
  float maxC = 0;
  float maxTheta = 0;
  float alpha_ij = 0;
  for (int j = min.y; j <= p.y; j++) {
    for (int i = min.x; i < max.x; i++) {
      if (abs(i-p.x) < sigma &&
          abs(j-p.y) < sigma) continue;
      pi = make_int2(i,j);
      if (pi == p) break; // we are done, since this computation is symmetric
      d = pi - p;
      pj = p - d;
      if (valid_pt(pi, ROWS, COLS) && valid_pt(pj, ROWS, COLS)) {
        rtheta_i = pt_gradient(mag, dir, pi);
        rtheta_j = pt_gradient(mag, dir, pj);
        alpha_ij = atan2f(pi.y - pj.y, pi.x - pj.x);
        C_ij = rtheta_i.x * rtheta_j.x * dist(pi, pj, (float)sigma) *
          phase(alpha_ij, rtheta_i.y, rtheta_j.y);
        M += C_ij;
        psi_ij = (rtheta_i.y + rtheta_j.y) * 0.5;
        if (C_ij > maxC) {
          maxC = C_ij;
          maxTheta = psi_ij;
        }
      }
    }
  }
  smag(p.y,p.x) = M;
  sdir(p.y,p.x) = maxTheta;
}
```