Efficient, High-Quality Image Contour Detection

Bryan Catanzaro, Bor-Yiing Su, Narayanan Sundaram, Yunsup Lee, Mark Murphy, Kurt Keutzer
EECS Department, University of California at Berkeley
573 Soda Hall, Berkeley, CA 94720
{catanzar, subrian, narayans, yunsup, mjmurphy, keutzer}@cs.berkeley.edu

Abstract

Image contour detection is fundamental to many image analysis applications, including image segmentation, object recognition and classification. However, highly accurate image contour detection algorithms are also very computationally intensive, which limits their applicability, even for offline batch processing. In this work, we examine efficient parallel algorithms for performing image contour detection, with particular attention paid to local image analysis as well as the generalized eigensolver used in Normalized Cuts. Combining these algorithms into a contour detector, along with careful implementation on highly parallel, commodity processors from Nvidia, our contour detector provides uncompromised contour accuracy, with an F-metric of 0.70 on the Berkeley Segmentation Dataset. Runtime is reduced from 4 minutes to 1.8 seconds. The efficiency gains we realize enable high-quality image contour detection on much larger images than previously practical, and the algorithms we propose are applicable to several image segmentation approaches. Efficient, scalable, yet highly accurate image contour detection will facilitate increased performance in many computer vision applications.

1. Introduction

We present a set of parallelized image processing algorithms useful for highly accurate image contour detection and segmentation. Image contour detection is closely related to image segmentation, and is an active area of research, with significant gains in accuracy in recent years. The approach outlined in [11], called gPb, achieves the highest published contour accuracy to date, but does so at high computational cost. On small images of approximately 0.15 megapixels, gPb requires 4 minutes of computation time on a high-end processor. Many applications, such as object recognition and image retrieval, could make use of such high quality contours for more accurate image analysis, but are still using simpler, less accurate image segmentation approaches due to their computational advantages.

At the same time, the computing industry is experiencing a massive shift towards parallel computing, driven by the capabilities and limitations of modern semiconductor manufacturing [2]. The emergence of highly parallel processors offers new possibilities to algorithms which can be parallelized to exploit them. Conversely, new algorithms must show parallel scalability in order to guarantee increased performance in the future. In the past, if a particular algorithm was too slow for wide application, there was reason to hope that future processors would execute the same code fast enough to make it practical. Unfortunately, those days are now behind us, and new algorithms must now express large amounts of parallelism, if they hope to run faster in the future.

In this paper, we examine efficient parallel algorithms for image contour detection, as well as scalable implementation on commodity, manycore parallel processors, such as those from Nvidia. Our image contour detector, built from these building blocks, demonstrates that high quality image contour detection can be performed in a matter of seconds rather than minutes, opening the door to new applications. Additionally, we show that our algorithms and implementation scale with increasing numbers of processing cores, pointing the way to continued performance improvements on future processors.

2. The gPb Detector

As mentioned previously, the highest quality image contour detector currently known, as measured by the Berkeley Segmentation Dataset, is the gPb detector. The gPb detector consists of many modules, which can be grouped into two main components: mPb, a detector based on local image analysis at multiple scales, and sPb, a detector based on the Normalized Cuts criterion. An overview of the gPb detector is shown in figure 1.

The mPb detector is constructed from brightness, color and texture cues at multiple scales. For each cue, the detector from [13] is employed, which estimates the probability of boundary $Pb_{C,\sigma}(x, y, \theta)$ for a given image channel, scale, pixel, and orientation by measuring the difference in

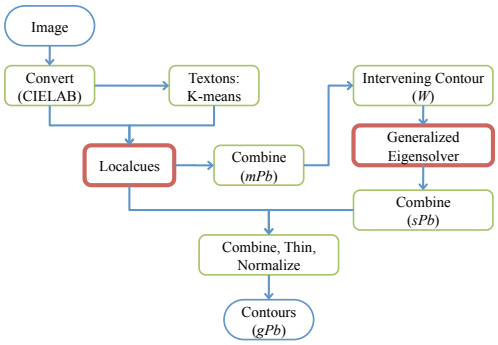

Figure 1. The gPb detector

image channel C between two halves of a disc of radius σ centered at (x, y) and oriented at angle θ. The cues are computed over four channels: the CIELAB 1976 L channel, which measures brightness, and A, B channels, which measure color, as well as a texture channel derived from texton labels [12]. The cues are also computed over three different scales $[\frac{\sigma}{2}, \sigma, 2\sigma]$ and eight orientations, in the interval $[0, \pi)$. The mPb detector is then constructed as a linear combination of the local cues, where the weights α_{ij} are learned by training on an image database:

$$mPb(x, y, \theta) = \sum_{i=1}^{4} \sum_{j=1}^{3} \alpha_{ij} Pb_{C_i,\sigma_j}(x, y, \theta) \quad (1)$$

The mPb detector is then reduced to a pixel affinity matrix W, whose elements W_{ij} estimate the similarity between pixel i and pixel j by measuring the intervening contour [9] between pixels i and j. Due to computational concerns, W_{ij} is not computed between all pixels i and j, but only for some pixels which are near to each other. In this case, we use Euclidean distance as the constraint, meaning that we only compute W_{ij} $\forall i, j$ $s.t.$ $||(x_i, y_i) - (x_j, y_j)|| \leq r$, otherwise we set $W_{ij} = 0$. In this case, we set $r = 5$.

This constraint, along with the symmetry of the intervening contour computation, ensures that W is a symmetric, sparse matrix (see figure 5), which guarantees that its eigenvalues are real, significantly influencing the algorithms used to compute sPb. Once W has been constructed, sPb follows the Normalized Cuts approach [16], which approximates the NP-hard normalized cuts graph partitioning problem by solving a generalized eigensystem. To be more specific, we must solve the generalized eigenproblem:

$$(D - W)v = \lambda Dv, \quad (2)$$

where D is a diagonal matrix constructed from W: $D_{ii} = \sum_j W_{ij}$. Only the $k+1$ eigenvectors v_j with smallest eigenvalues are useful in image segmentation and need to be extracted. In this case, we use $k = 8$. The smallest eigenvalue of this system is known to be 0, and its eigenvector is not used in image segmentation, which is why we extract $k + 1$ eigenvectors. After computing the eigenvectors, we extract their contours using Gaussian directional derivatives at multiple orientations θ, to create an oriented contour signal $sPb_{v_j}(x, y, \theta)$. We combine the oriented contour signals together based on their corresponding eigenvalues:

$$sPb(x, y, \theta) = \sum_{j=2}^{k+1} \frac{1}{\sqrt{\lambda_j}} sPb_{v_j}(x, y, \theta) \quad (3)$$

The final gPb detector is then constructed by linear combination of the local cue information and the sPb cue:

$$gPb(x, y, \theta) = \gamma \cdot sPb(x, y, \theta) + \sum_{i=1}^{4} \sum_{j=1}^{3} \beta_{ij} Pb_{C_i,\sigma_j}(x, y, \theta) \quad (4)$$

where the weights γ and β_{ij} are also learned via training. To derive the final $gPb(x, y)$ signal, we maximize over θ, threshold to remove pixels with very low probability of being a contour pixel, skeletonize, and then renormalize.

3. Algorithmic Exploration

3.1. Local cues

Computing the local cues for all channels, scales, and orientations is computationally expensive. There are two major steps: computing the local cues, and then smoothing them to remove spurious edges. We found significant efficiency gains in modifying the local cue computation to utilize integral images, so we will detail how this was accomplished.

3.1.1 Explicit local cues

Given an input channel, orientation, and scale, the local cue computation involves building two histograms per pixel, which describe the input channel's intensity in the opposite halves of a circle centered at that pixel, with the orientation describing the angle of the diameter of the half-discs, and the scale determining the radius of the half-discs. The two histograms are optionally blurred with a Gaussian, normalized, and then compared using the χ^2 distance metric:

$$\chi^2(x, y) = \frac{1}{2} \sum_i \frac{(x_i - y_i)^2}{x_i + y_i} \quad (5)$$

If the two histograms are significantly different, there is likely to be an edge at that pixel, orientation and scale.

When computing these histograms by explicitly summing over half-discs, computation can be saved by noticing that the computation for each orientation overlapped significantly with other orientations, so the histograms were computed for wedges of the circle, and then assembled into

the various half-disc histograms necessary for each orientation. However, this approach does not consider that the circle overlapped with circles centered at neighboring pixels. Additionally, this approach recomputes the histograms completely for each of the different scales, and the computation necessary is a function of the scale radius itself, meaning that larger scales incur significantly more computational cost than smaller scales. Furthermore, parallel implementations of this approach are complicated by the data-dependent nature of constructing histograms, which incurs higher synchronization costs than algorithms with static data dependency patterns.

3.1.2 Integral Images

To alleviate these problems, we turned to the well-known technique of integral images [10]. Integral images allow us to perform sums over rectangles in $O(1)$ time instead of $O(N)$ time, where N is the number of pixels in the rectangle. To construct an integral image, one computes I from an image F as

$$I(x,y) = \sum_{x'=1}^{x} \sum_{y'=1}^{y} F(x',y') \tag{6}$$

Computing the sum of a shape then involves summing as many entries from the integral image as there are corners in the shape. For example, a rectangle with extent ranging from (x_1, y_1) to (x_2, y_2) is summed as follows:

$$\begin{aligned} \sum_{x=x_1}^{x_2} \sum_{y=y_1}^{y_2} F(x,y) &= I(x_1-1, y_1-1) \\ -I(x_1-1, y_2) &- I(x_2, y_1-1) + I(x_2, y_2) \end{aligned} \tag{7}$$

We use integral images to compute histograms of the half-discs discussed previously. To do so, we approximate each half-disc as a rectangle of equal area. Although integral images can be used efficiently for summing other shapes than rectangles, we found that this approximation worked well.

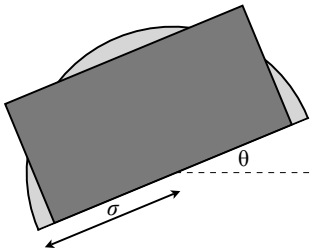

Figure 2. Approximating a half-disc with a rectangle

We then compute an integral image for each bin of the histogram, similarly to [17]. Complicating the use of integral images in this context is the fact that integral images can only compute sums of rectangles, whereas we need to compute sums of rotated rectangles. Computing integral images for rotated images has been tried previously, but was restricted to special angles, such as [7].

Our approach to rotated integral images reduces rotation artifacts and can handle arbitrary angles, based on the use of Bresenham lines [5]. The problem associated with computing integral images on rotated images is that standard approaches to rotating an image interpolate between pixels. This is not meaningful for texton labels: since the labels are arbitrary integers without a partial ordering, bin n bears no relation to bin $n+1$, and therefore bin $n+0.5$ has no meaning. Nearest neighbor interpolation does not require interpolating the pixel values, but under rotation it omits some pixels, while counting others multiple times, introducing artifacts. To overcome this, we rotate the image using Bresenham lines. This method ensures a one-to-one correspondence between pixels in the original image and pixels in the rotated image, at the expense of introducing some blank pixels. The effect can be seen in Figure 3. Bresenham rotation does introduce some discretization of the rotation angle, but this discretization tends to zero as the image size increases[1].

The Bresenham rotation produces images that are larger than the original image, but are bounded at $(w+h)^2$ pixels, which bound is encountered at $\theta = \frac{\pi}{4}$.

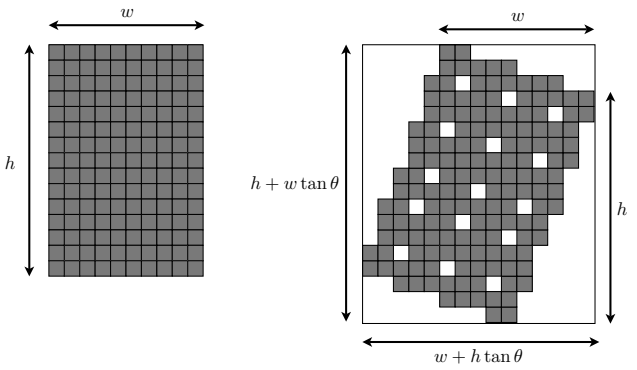

Figure 3. Bresenham rotation: Rotated image with $\theta = 18^\circ$ clockwise, showing "empty pixels" in the rotated image.

Although Bresenham rotation introduces some computational inefficiencies due to empty pixels, it is more accurate than nearest neighbor interpolation, since pixels are not missed or multiply counted during the image integration, as occurs using nearest neighbor interpolation. Therefore, we use it in our local cue detector.

Integral images for computing histograms over rectangles remove some of the computational complexity of the local cues extraction. The explicit method for image histogram creation has complexity $O(Nr^2st)$, where N is the

[1] More analysis found in supplementary material

number of pixels, r is the radius of the half-disc being extracted, s is the number of scales, and t is the number of orientations. It should be noted that some detectors might wish to scale r^2 with N, making the complexity $O(N^2st)$. Using integral images reduces the complexity of histogram construction to $O(Nst)$.

3.2. Eigensolver

The generalized eigenproblem needed for Normalized Cuts is the most computationally intensive part of the gPb algorithm. Therefore, an efficient eigensolver is necessary for achieving high performance. We have found that a Lanczos-based eigensolver using the Cullum-Willoughby test without reorthogonalization provides the best performance on the eigenproblems generated by Normalized Cuts approaches. We also exploit the special structure and properties of the graph Laplacian matrices generated for the Normalized Cuts algorithm in our eigensolver. Before explaining our improvements, we present the basic algorithm used for solving these eigenproblems.

3.2.1 Lanczos Algorithm

The generalized eigenproblem from Normalized Cuts can be transformed into a standard eigenproblem [16]: $A\bar{v} = \lambda\bar{v}$, with $A = D^{-\frac{1}{2}}(D - W)D^{-\frac{1}{2}}$.

The matrix A is Hermitian, positive semi-definite, and its eigenvalues are well distributed. Additionally, we only need a few of the eigenvectors, corresponding to the smallest $k + 1$ eigenvalues. Considering all the issues above, the Lanczos algorithm is a good fit for this problem [3], and is summarized in Figure 4. The complete eigenproblem has complexity $O(n^3)$ where n is the number of pixels in the image, but the Lanczos algorithm is $O(mn) + O(mM(n))$, where m is the maximum number of matrix vector products, and $M(n)$ is the complexity of each matrix vector product, which is $O(n)$ in our case. Empirically, m is $O(n^{\frac{1}{2}})$ or better for normalized cuts problems [16], meaning that this algorithm scales at approximately $O(n^{\frac{3}{2}})$ for our problems.

For a given symmetric matrix A, the Lanczos algorithm proceeds by iteratively building up a basis V, which is used to project the matrix A into a tridiagonal matrix T. The eigenvalues of T are computationally much simpler to extract than those of A, and they converge to the eigenvalues of A as the algorithm proceeds. The eigenvectors of A are then constructed by projecting the eigenvectors of T against the basis V. More specifically, v_j denotes the Lanczos vector generated by each iteration, V_j is the orthogonal basis formed by collecting all the Lanczos vectors $v_1, v_2, \ldots, v_j$ in column-wise order, and T_j is the symmetric $j \times j$ tridiagonal matrix with diagonal equal to $\alpha_1, \alpha_2, \ldots, \alpha_j$, and upper diagonal equal to $\beta_1, \beta_2, \ldots, \beta_{j-1}$. S and Θ form the eigendecomposition of matrix T_j. Θ contains the approximation to the eigenvalues of A, while S in conjunction with V approximates the eigenvectors of A: $x_j = V_j s_j$.

Algorithm: Lanczos
Input: A (Symmetric Matrix)
v (Initial Vector)
Output: Θ (Ritz Values)
X (Ritz Vectors)

Start with $r \leftarrow v$;
$\beta_0 \leftarrow \|r\|_2$;
for $j \leftarrow 1, 2, \ldots,$ **until** convergence
  $v_j \leftarrow r/\beta_{j-1}$;
  $r \leftarrow Av_j$;
  $r \leftarrow r - v_{j-1}\beta_{j-1}$;
  $\alpha_j \leftarrow v_j^* r$;
  $r \leftarrow r - v_j\alpha_j$;
  Reorthogonalize if necessary ;
  $\beta_j \leftarrow \|r\|_2$;
  Compute Ritz values $T_j = S\Theta S$;
  Test bounds for convergence ;
end for
Compute Ritz vectors $X \leftarrow V_j S$;

Figure 4. The Lanczos algorithm.

There are three computational bottlenecks of the Lanczos algorithm: Matrix-vector multiplication, Reorthogonalization, and the eigendecomposition of the tridiagonal matrix T_j. We discuss reorthogonalization for Normalized Cuts problems in section 3.2.2, and the matrix-vector multiplication problem in section 3.2.3. We solve the third bottleneck by diagonalizing T_j infrequently, since it is only necessary to do so when checking for convergence, which does not need to be done at every iteration.

3.2.2 Reorthogonalization and the Cullum-Willoughby test

In perfect arithmetic, the basis V_j constructed by the Lanczos algorithm is orthogonal. In practice, however, finite floating-point precision destroys orthogonality in V_j as the iterations proceed. Many Lanczos algorithms preserve orthogonality by selectively reorthogonalizing new Lanczos vectors v_j against the existing set of Lanczos vectors V_{j-1}. However, this is very computationally intensive. An alternative is to proceed without reorthogonalization, as proposed by Cullum and Willoughby [6]. We have found that this alternative offers significant advantages for Normalized Cuts problems in image segmentation and image contour detection.

When V_j is not orthogonal, spurious and duplicate Ritz values will appear in Θ, which need to be identified and re-

moved. This can be done by constructing $\hat{T}$ as the tridiagonal matrix constructed by deleting the first row and first column of T_j. The spurious eigenvalues of T_j can then be identified by investigating the eigenvalues of $\hat{T}$. An eigenvalue is spurious if it exists in T_j only once and exists in $\hat{T}$ as well. For more details, see [6]. Because the lower eigenvalues of affinity matrices encountered from the Normalized Cuts approach to image segmentation are well distributed, we can adopt the Cullum-Willoughby test to screen out spurious eigenvalues. This approach improved eigensolver performance by a factor of 20× over full reorthogonalization, and 5× over selective reorthogonalization, despite requiring significantly more Lanczos iterations.

This approach to reorthogonalization can be generally applied to all eigenvalue problems solved as part of the normalized cuts method for image segmentation. In general, the eigenvalues corresponding to the different cuts (segmentations) are well spaced out at the low end of the eigenspectrum. For the normalized Laplacian matrices with dimension N, the eigen values lie between 0 and N (loose upper bound) as $tr[A] = \sum_i \lambda_i = N$ and $\lambda_i \geq 0$. Since the number of eigenvalues is equal to the number of pixels in the image, one might think that as the number of pixels increases, the eigenvalues will be more tightly clustered, complicating convergence analysis using the Cullum-Willoughby test. However, we have observed that this clustering is not too severe for the smallest eigenvalues of matrices derived from natural images, which are the ones needed by normalized cuts. As justification for this phenomenon, we observe that very closely spaced eigenvalues at the smaller end of the eigenspectrum would imply that several different segmentations with different numbers of segments are equally important, which is unlikely in natural images where the segmentation, for a small number of segments, is usually distinct from other segmentations. In practice, we have observed that this approach works very well for Normalized Cuts image segmentation computations.

3.2.3 Sparse Matrix Vector Multiplication (SpMV)

The Lanczos algorithm requires repeatedly multiplying the matrix by dense vectors; given a randomly initialized vector v_0, this process generates the sequence of vectors $A \cdot v_0, A^2 \cdot v_0, \ldots$. As the matrix is very large ($N \times N$, where N is the number of pixels in the image), and the multiplication occurs in each iteration of the Lanczos algorithm, this operation accounts for approximately 2/3 of the runtime of the serial eigensolver.

SpMV is a well-studied kernel in the domain of scientific computing, due to its importance in a number of sparse linear algebra algorithms. A naïvely written implementation runs far below the peak throughput of most processors. The poor performance is typically due to low-efficiency of memory access to the matrix as well as the source and destination vectors.

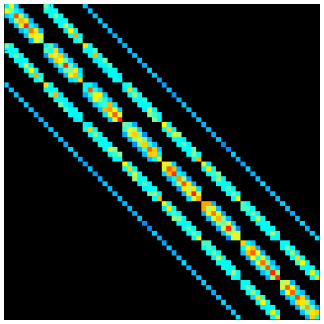

Figure 5. Example W matrix

The performance of SpMV depends heavily on the structure of the matrix, as the arrangement of non-zeroes within each row determine the pattern of memory accesses. The matrices arising from Normalized Cuts are all multiply banded matrices, since they are derived from a stencil pattern where every pixel is related to a fixed set of neighboring pixels. Figure 5 shows the regular, banded structure of these matrices. It is important to note that the structure arises from the pixel-pixel affinities encoded in the W matrix, but the A matrix arising from the generalized eigenproblem retains the same structure. Our implementation exploits this structure in a way that will apply to any stencil matrix.

In a stencil matrix, we can statically determine the locations of non-zeroes. Thus, we need not explicitly store the row and column indices, as is traditionally done for general sparse matrices. This optimization alone nearly halves the size of the matrix data structure, and doubles performance on nearly any platform. We store the diagonals of the matrix in consecutive arrays, enabling high-bandwidth unit-stride accesses, and reduce the indexing overhead to a single integer per row. Utilizing similar optimizations as described in [4], our SpMV routine achieves 40 GFlops/s on matrices derived from the intervening contour approach, with $r = 5$, leading to 81 nonzero diagonals.

4. Implementation and Results

Our code was written in CUDA [15], and comprises parallel k-means, convolution, and skeletonization routines in addition to the local cues and eigensolver routines. Space constraints prohibit us from detailing these routines, but it is important to note that our routines require CUDA architecture 1.1, with increased performance on the k-means routines on processors supporting CUDA architecture 1.2. The code from our implementation is freely available at `http://parlab.eecs.berkeley.edu/research/damascene`.

4.1. Accuracy

4.1.1 Berkeley Segmentation Dataset

Firstly, we need to show that our algorithms have not degraded the contour quality. We evaluate the quality of our contour detector by the BSDS benchmark [14]. As shown in Figure 6, we achieve the same F-metric (0.70) as the gPb algorithm [11], and the quality of our P-R curve is also very competitive to the curve generated by the gPb algorithm. Figure 7 illustrates contours for several images generated by our contour detector.

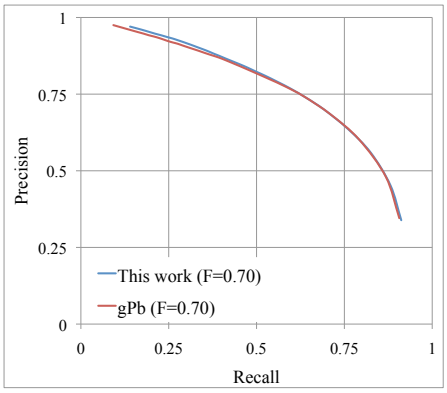

Figure 6. Precision Recall Curve for our Contour Detector

4.1.2 Larger Images

To investigate the accuracy of our contour detector on larger images, we repeated this precision-recall test on 4 images . To generate this data, we hand labeled 4 images multiple times to create a human ground truth test for larger images [2]. We used our existing contour detector, without retraining or changing the scales, and found that the contour detector worked on larger images as well, with an indicated F-metric of 0.75. Obviously, our test set was very small, so we are not claiming that this is the realistic F-metric on larger images, rather we are simply showing that the detector provides reasonable results on larger images.

4.2. Runtime

To compare runtimes, we use the published gPb code, running on an Intel Core i7 920 (2.66 GHz) with 4 cores and 8 threads. The original gPb code is written mostly in C++, coordinated by MATLAB scripts, as well as MATLAB's `eigs` eigensolver, which is based on ARPACK and is reasonably optimized. We found MATLAB's eigensolver performed similarly to TRLan [18] on Normalized Cuts problems.

Although most of the computation in gPb was done in C++, there was one routine which was implemented in MATLAB and performed unacceptably: the convolutions required for local cue smoothing. In order to make our runtime comparisons fair, we wrote our own parallel convolution routine, taking full advantage of SIMD & thread parallelism on the Intel processor, and report the runtime using our convolution routine instead of the one which accompanies the gPb code.

To be conservative in our comparisons of our fully parallelized implementation with the serial gPb detector, we also took advantage of thread-level parallelism in our Intel convolution routine, and allowed MATLAB to parallelize the eigensolver over our 8 threaded Core i7 processor. This means that a completely serial version of gPb would be somewhat slower than the version we compare against.

Comparisons between gPb and this work are found in table 1.

Component	gPb (Core i7)	This work (GTX 280)	Speedup
Preprocess	0.090	0.001	90×
Textons	8.58	0.159	54×
Local Cues	53.18	0.569	93×
Smoothing	0.59	0.270	2.2×
Int. Contour	6.32	0.031	204×
Eigensolver	151.2	0.777	195×
Post Process	2.7	0.006	450×
Total	236.7	1.822	130×

Table 1. Runtimes in seconds (0.15 MP image)

4.3. Algorithmic Improvements

To isolate the algorithmic efficiency gains from the implementation efficiency gains, we examine the performance of the local cues extraction and the eigensolver.

Local cues	Explicit Method	Integral Images
Runtime (s)	4.0	0.569

Table 3. Local Cues Runtimes on GTX 280

The explicit local cues method utilizes a parallelized version of the same histogram building approach found in gPb: it explicitly counts all pixels in each half-disc, for each orientation and scale. As shown in table 3, the integral image approach is about about 7× more efficient than the explicit method.

Eigensolver Reorthogonalization	Full	Selective	None (C-W)
Runtime (s)	15.83	3.60	0.78

Table 4. Eigensolver Runtimes on GTX 280

[2] Labeled images and results in supplementary material

Figure 7. Selected image contours

Processor	Preprocess	Textons	Local Cues	Smoothing	Int. Contour	Eigensolver	Postprocess	Total
8600M GT	0.010	10.337	7.761	2.983	0.300	7.505	0.041	28.962
9800 GX2	0.003	2.311	1.226	0.530	0.056	1.329	0.009	5.497
GTX 280	0.001	0.159	0.569	0.270	0.031	0.777	0.006	1.822
Tesla C1060	0.002	0.178	0.584	0.267	0.03	1.166	0.006	2.243

Table 2. GPU Scaling. Runtimes in seconds

Table 4 shows the effect of various reorthogonalization strategies. Full reorthogonalization ensures that every new Lanczos vector v_j is orthogonal to all previous vectors. Selective reorthogonalization monitors the loss of orthogonality in the basis and performs a full reorthogonalization only when the loss of orthogonality is numerically significant to within machine floating-point tolerance. The strategy we use, as outlined earlier, is to forgo reorthogonalization, and use the Cullum-Willoughby test to remove spurious eigenvalues due to loss of orthogonality. As shown in the table, this approach provides a $20\times$ gain in efficiency.

4.4. Scalability

We ran our detector on a variety of commodity, single-socket graphics processors from Nvidia, with widely varying degrees of parallelism. These experiments were performed to demonstrate that our approach scales to a wide variety of processors. The exact specifications of the processors we used can be found in Table 5.

Processor model	Cores (Multi processors)	Memory Bandwidth GB/s	Clock Frequency GHz	Available Memory MB
8600M GT	4	12.8	0.92	256
9800 GX2	16	64	1.51	512
GTX 280	30	141.7	1.30	1024
C1060	30	102	1.30	4096

Table 5. Processor Specifications

Figure 8 shows how the runtime of our detector scales with increasingly parallel processors, with more details in table 2. Each of the 4 processors we evaluated is represented on the plot of performance versus the number of cores. We have two processors with the same number of cores, but different amounts of memory bandwidth, which explain the different results at 30 cores. Clearly, our work efficiently capitalizes on parallel processors, which gives us confidence that performance will continue to increase on future generations of manycore processors.

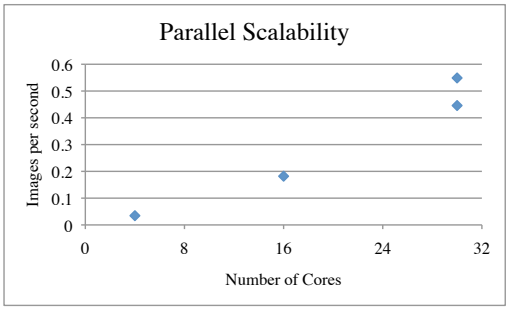

Figure 8. Performance scaling with parallelism (0.15 MP images)

Figure 9 demonstrates the runtime dependence on input image size. These experiments were all run on the Tesla C1060 processor, since we require its large memory capacity to compute contours on the larger images. Runtime dependence is mostly linear in the number of pixels over this range of image sizes.

5. Conclusion

In this work, we have demonstrated how the careful choice of parallel algorithms along with implementation on

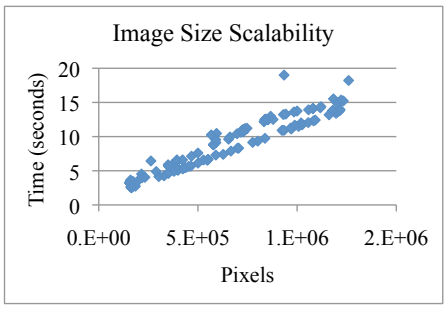

Figure 9. Runtime scaling with increased image size

manycore processors can enable high quality, highly efficient image contour detection. We have detailed how one can use integral images to improve efficiency by replacing histogram construction with parallel prefix operations even under arbitrary rotations. We have also shown how eigenproblems encountered in Normalized Cuts approaches to image segmentation can be efficiently solved by the Lanczos algorithm with Cullum-Willoughby test.

Combining these contributions to create a contour detector, we show that runtime can be reduced over $100\times$, while still providing equivalent contour accuracy. We have also shown how our routines allow us to find image contours for larger images, and detailed how our detector scales across processors with widely varying amounts of parallelism. This makes us confident that future, even more parallel manycore processors will continue providing increased performance on image contour detection.

Future work includes using the components we have developed in other computer vision problems. It is possible that doing more image analysis with our optimized components will allow for yet higher image contour detection quality. Our contours could be integrated into a method which produces image segments, such as [1], which can be more natural in some applications, such as object recognition [8]. Other possibilities are also open, such as video segmentation. We believe that the efficiency gains we realize will allow for high quality image segmentation approaches to be more widely utilized in many contexts.

6. Acknowledgements

Thanks to Michael Maire for suggesting we investigate integral images, and Pablo Arbeláez for assisting us with the gPb algorithm. Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding, and by matching funding from U.C. Discovery (Award #DIG07-10227).

References

[1] P. Arbeláez, M. Maire, and J. Malik. From contours to regions: An empirical evaluation. In *CVPR*, 2009.

[2] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst. *Templates for the solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, 2000.

[4] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput oriented processors. In *Supercomputing '09*, Nov. 2009.

[5] J. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[6] J. K. Cullum and R. A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*. Vol. I: Theory. SIAM, 2002.

[7] S. Du, N. Zheng, Q. You, Y. Wu, M. Yuan, and J. We. Rotated haar-like features for face detection with in-plane rotation. *LNCS*, 4270/2006:128–137, 2006.

[8] C. Gu, J. Lim, P. Arbeláez, and J. Malik. Recognition using regions. In *CVPR*, 2009.

[9] T. Leung and J. Malik. Contour continuity in region based image segmentation. In *In Proc. ECCV, LNCS 1406*, pages 544–559. Springer-Verlag, 1998.

[10] R. Lienhart and J. Maydt. An extended set of haar-like features for rapid object detection. In *Proc. IEEE Conf. on Image Processing*, pages 155–162, New York, USA, 2002.

[11] M. Maire, P. Arbeláez, C. Fowlkes, and J. Malik. Using contours to detect and localize junctions in natural images. *CVPR*, pages 1–8, June 2008.

[12] J. Malik, S. Belongie, J. Shi, and T. Leung. Textons, contours and regions: Cue integration in image segmentation. In *ICCV '99*, page 918, Washington, DC, USA, 1999. IEEE Computer Society.

[13] D. Martin, C. Fowlkes, and J. Malik. Learning to detect natural image boundaries using brightness and texture, 2002.

[14] D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *ICCV 2001*, volume 2, pages 416–423, July 2001.

[15] Nvidia. Nvidia CUDA, 2007. `http://nvidia.com/cuda`.

[16] J. Shi and J. Malik. Normalized cuts and image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(8):888–905, Aug 2000.

[17] M. Villamizar, A. Sanfeliu, and J. Andrade-Cetto. Computation of rotation local invariant features using the integral image for real time object detection. In *Int'l. Conf. on Pattern Recognition*, 2006.

[18] K. Wu and H. Simon. Thick-restart lanczos method for large symmetric eigenvalue problems. *SIAM Journal on Matrix Analysis and Applications*, 22(2):602–616, 2001.