Really quick shift: Image segmentation on a GPU

Brian Fulkerson and Stefano Soatto

Department of Computer Science,
University of California, Los Angeles
{bfulkers,soatto}@cs.ucla.edu
http://vision.ucla.edu/

Abstract. The paper presents an exact GPU implementation of the quick shift image segmentation algorithm. Variants of the implementation which use global memory and texture caching are presented, and the paper shows that a method backed by texture caching can produce a 10-50X speedup for practical images, making computation of super-pixels possible at 5-10Hz on modest sized (256x256) images.

Key words: super-pixels, segmentation, CUDA, GPU programming

1 Introduction

Segmentation algorithms have played an important role in computer vision research, both as an end goal [1–3] and more recently as a preprocessing step for other domains, including stereo [4] and category-level scene parsing [5, 6]. Breaking the image into smaller components, often called super-pixels, allows algorithms to consider the image in meaningful chunks, rather than at the lowest common denominator (pixels).

Unfortunately, algorithms developed for segmentation are often quite costly in both memory usage and computation. This bottleneck limits the scale of the applications and data that they can be applied to.

In this work, we show that a GPU implementation of quick shift [3] can improve the performance of an already (relatively) fast segmentation algorithm by 10X-50X, opening up a host of potential new applications such as scene understanding in videos, and improved real time video abstraction [7].

2 Related Work

Most related work involving GPUs for segmentation is in the medical imaging domain, where the extra dimension of data (a volume instead of an image) has made speed a requirement rather than an option [8–11]. One notable exception found outside of medical imaging is that of Catanzaro *et al.* [12] who adapt a boundary detection technique (gPb [13]) to the GPU. While gPb can be used

for segmentation [14], our exact implementation of quick shift is over ten times faster on similar hardware.

In recognition, GPU based feature detectors and trackers [15, 16] have been proposed, as have learning components such as support vector machines [17] and k-nearest neighbors [18]. Recently, Wojek *et al.* [19] even proposed a GPU accelerated sliding window categorization scheme.

Other recent successes in using GPUs for vision include general purpose libraries such as OpenVIDIA [20], and specific applications which are often centered around video such as motion detection [21] or particle filtering [22].

Carreira *et al.* [23] have done work on approximating Gaussian Mean Shift (GMS) by decreasing the number of iterations required by the algorithm and the cost per iteration (by approximating the density). We effectively circumvent the need to optimize the number of iterations because quick shift only requires one iteration. Instead of approximating the density, we simply exploit the parallelism of the density computation to achieve a speedup by using hardware suited for the task (a GPU). We note that we could also approximate the density as in [23], and that would result in further speedups.

3 Quick shift algorithm

Quick shift is a kernelized version of a mode seeking algorithm similar in concept to mean shift [2, 24] or medoid shift [25]. Given N data points $x_1, \dots, x_N$, it computes a Parzen density estimate around each point using, for example, an isotropic Gaussian window:

$$P(x) = \frac{1}{2\pi\sigma^2 N} \sum_{i=1}^{N} e^{\frac{-\|x - x_i\|^2}{2\sigma^2}}$$

Once the density estimate $P(x)$ has been computed, quick shift connects each point to the nearest point in the feature space which has a higher density estimate. Each connection has a distance d_x associated with it, and the set of connections for all pixels forms a tree, where the root of the tree is the point with the highest density estimate.

Quick shift may be used for any feature space, but for the purpose of this paper we restrict it to one we can use for image segmentation: the raw RGB values augmented with the (x, y) position in the image. So, the feature space is five dimensional: (r, g, b, x, y). To adjust the trade-off between the importance of the color and spatial components of the feature space, we simply pre-scale the (r, g, b) values by a parameter λ, which for these experiments we fix at $\lambda = 0.5$.

To obtain a segmentation from a tree of links formed by quick shift, we choose a threshold τ and break all links in the tree with $d_x > \tau$. The pixels which are a member of each resulting disconnected tree form each segment.

3.1 Segmentation specific optimizations

In the case where our feature space is restricted to contain components which are defined on the image plane, and our set of data points are the set of pixels, we can immediately put some useful bounds on both the density computation and the neighbor linking process.

First, when computing the energy we can restrict the domain of pixels we consider to a window which is less than 3σ pixels away, because beyond this the contribution to the density is guaranteed to be small. Second, when linking the neighbors, there is also a natural bound for the search window, because pixels which are further than τ away in the image plane must be at least that far away in the feature space. Conceptually we will talk about the density computation and linking process as separate components of the algorithm, because one (the density computation) must precede the other, and they operate on different domains of data. A pseudo-code implementation is shown in Figure 2, and some segmentations with various parameters are shown in Figure 1.

4 Quick shift on a GPU

Because quick shift operates on each pixel of an image, and the computation which takes place at each pixel is independent of its distant surroundings, it is a good candidate for implementation on a parallel architecture.

We use CUDA 3.0 to develop a first implementation which simply copies the image to the device and breaks the computation of the density and the neighbors into blocks for the GPU to process.

Although this is faster than the CPU version, the bottleneck is clearly memory latency. Global memory on GPUs is slow, requiring hundreds of cycles to access, and for each pixel quick shift needs to access $\mathrm{ceil}((6 * \sigma)^2)$ neighbors.

To address this, one option is to load an *apron* of pixels surrounding the block being computed into shared memory, so that when an element of the block computes its similarity with a pixel outside of the block, the memory access is cached. However, because this operation is not easily separable, the shared memory requirement scales quadratically with sigma. Even modest values of sigma will quickly exhaust the 16000 bytes of shared memory available on modern GPUs.

So, we instead map the image and the estimate of the density to a 3D and 2D texture, respectively. We have good locality of access because each thread accesses a block of pixels around it. The results based of this texture cached approach are labeled with a “Tex” suffix in the next section.

5 Evaluation

There are two aspects of the algorithm to evaluate: the correctness and the time required. To confirm the correctness of the GPU implementation, we compare

Fig. 1. Sample quick shift results. Increasing σ smoothes the underlying estimate of the density, providing fewer modes. Increasing τ increases the average size of a region as well as the error in the distance estimate. The top row of images have $\sigma = 2$, the bottom row $\sigma = 10$. The left column has $\tau = 10$ and the right $\tau = 20$.

```
function computeDensity()
for x in all pixels
  P[x] = 0
  for n in all pixels less than 3*sigma away
    P[x] += exp(-(f[x]-f[n])^2 / (2*sigma*sigma))

function linkNeighbors()
for x in all pixels
  for n in all pixels less than tau away
    if P[n] > P[x] and distance(x,n) is smallest among all n
      d[x] = distance(x,n)
      parent[x] = n
```

Fig. 2. Quick shift image segmentation in pseudo-code. The algorithm proceeds in two steps. First it iterates over the image creating a Parzen estimate of the density at each pixel. Then, it links each pixel to the nearest pixel (in the feature space) which increases the estimate of the density.

Fig. 3. Evaluation images. Four images from PASCAL-2007 used to evaluate the speed of the proposed algorithm.

the energy and segmentation to the one returned by the publicly available implementation of quick shift in VLFeat [26].

To measure the speed of the algorithm, we pick a few random images from the PASCAL-2007 dataset (shown in Figure 3). The images are cropped and up-sampled to 1024x1024. All reported performance numbers are obtained by averaging the results from all of the images.

We explore the effect of each parameter which changes the runtime of the algorithm. First, in Figure 4 we show the performance of the algorithms as the resolution of the image is increased while keeping σ and τ fixed. Next, in Figure 5 we keep the resolution fixed at 512x512, fix τ, and adjust σ, showing how it affects the runtime of just the density computation part of the algorithm. Finally, Figure 6 keeps both the resolution and σ fixed and instead adjusts τ, showing the time required to link the neighbors.

Hardware. The CPU ground truth version is evaluated on a 2.4Ghz Core 2 Duo. We show results for two GPUs: a laptop board (GeForce 8600M GT), and a mid-range desktop card (GeForce 9800 GT). The 8600M GT has 4 multiprocessors, 32 cores, and a core clock speed of 475MHz. The 9800 GT has 14 multiprocessors, 112 cores, a 550MHz core clock speed. Due to limits on the runtime of CUDA kernels on the 8600M, in Figures 5 and 6 results are not reported for the slowest running case because the kernel was stopped before completion. We note that while newer hardware (such as cards based on the recently released FERMI architecture) would undoubtedly be faster, we want to show what is possible with only limited hardware investment.

For both GPUs evaluated we use a block size of 16x16, even though it has been shown that tuning the block size for a particular GPU can provide a boost in performance.

Our complete source code is available on our website at `http://vision.ucla.edu/~brian/gpuquickshift.html`.

6 Conclusion

We have shown a GPU implementation of quick shift which provides a 10 to 50 times speedup over the CPU implementation, resulting in a super-pixelization algorithm which can run at 10Hz on 256x256 images. The implementation is an exact copy of quick shift, and could be further speeded up by approximating the density, via subsampling or other methods. It is likely that the implementation would also present similar speedups for exact mean shift.

Acknowledgements

This research was supported by ONR 67F-1080868/N00014-08-1-0414, ARO56765-CI and AFOSR FA9550-09-1-0427.

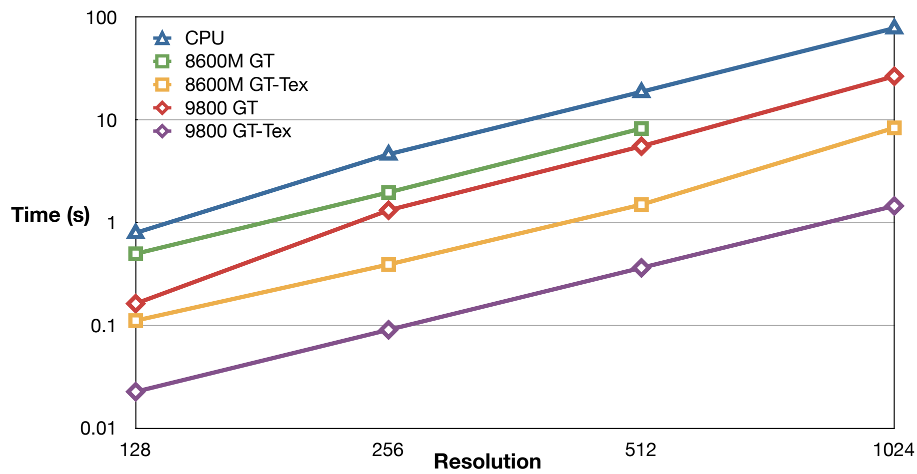

Fig. 4. Quick shift CPU vs GPU. The graph shows the amount of time required on two different GPUs as the resolution of the image is increased. Results are averaged over the four images from PASCAL-2007 shown in Figure 3. For this data, $\sigma = 6$ and $\tau = 10$. At 1024x1024, the speedup compared to the CPU version is 54X.

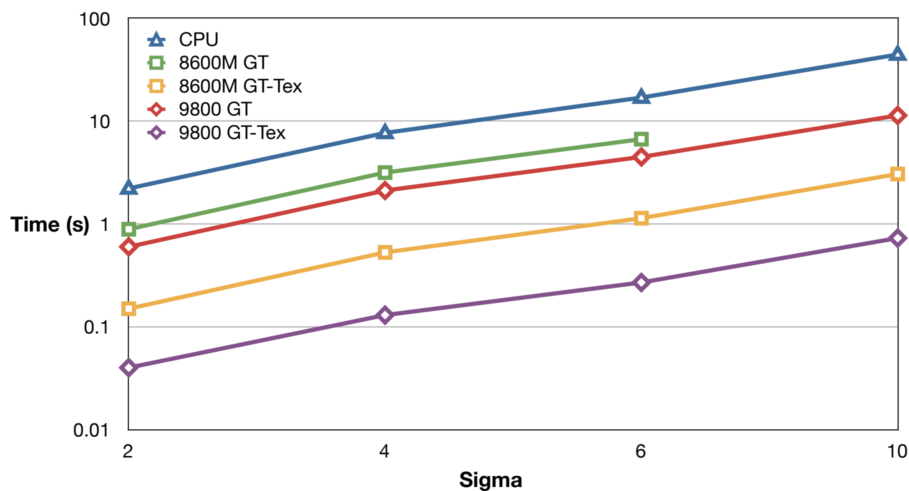

Fig. 5. Effect of σ on density computation time. As in Figure 4, we show that as σ is increased, processing time is increased and the texture memory-backed GPU version remains the most efficient option. Here we fix $\tau = 10$ and the image resolution to 512x512. Results are averaged over the same four images as before.

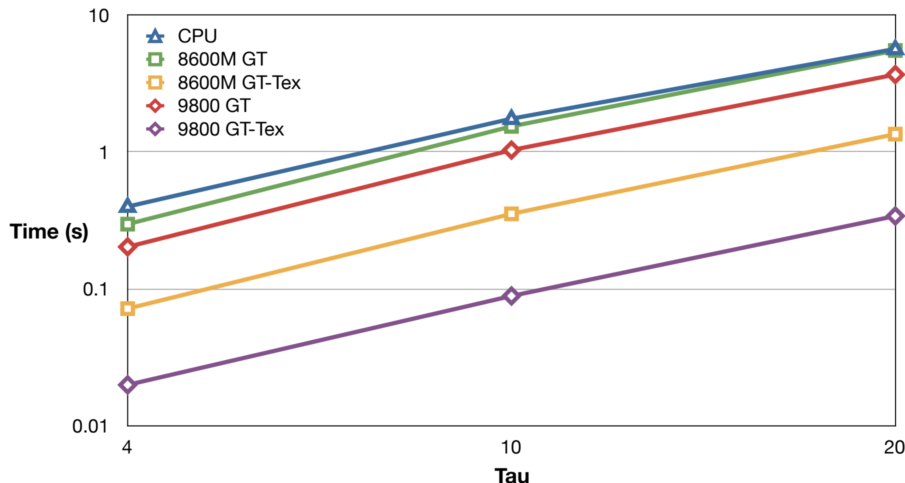

Fig. 6. Effect of τ on neighbor linking time. We show that as τ is increased, the amount of time required for finding the nearest neighbor which increases the density estimate is naturally increased. Here we fix $\sigma = 6$ and the image resolution to 512x512. Results are averaged over the same four images as before.

References

1. Shi, J., Malik, J.: Normalized cuts and image segmentation. PAMI **22** (2000) 888
2. Comaniciu, D., Meer, P.: Mean shift: A robust approach toward feature space analysis. PAMI **24** (2002)
3. Vedaldi, A., Soatto, S.: Quick shift and kernel methods for mode seeking. In: Proc. ECCV. (2008)
4. Lei, C., Selzer, J., Yang, Y.: Region-tree based stereo using dynamic programming optimization. In: Proc. CVPR. (2006)
5. Fulkerson, B., Vedaldi, A., Soatto, S.: Class segmentation and object localization with superpixel neighborhoods. In: Proc. ICCV. (2009)
6. Gould, S., Rodgers, J., Cohen, D., Elidan, G., Koller, D.: Multi-class segmentation with relative location prior. In: IJCV. (2008)
7. Winnemoller, H., Olsen, S., Gooch, B.: Real-time video abstraction. ACM Transactions on Graphics (TOG) **25** (2006) 1226
8. Sherbondy, A., Houston, M., Napel, S.: Fast volume segmentation with simultaneous visualization using programmable graphics hardware. In: Proceedings of the 14th IEEE Visualization 2003 (VIS'03), IEEE Computer Society (2003) 23
9. Cates, J., Lefohn, A., Whitaker, R.: GIST: an interactive, GPU-based level set segmentation tool for 3D medical images. Medical Image Analysis **8** (2004) 217–231
10. Lefohn, A., Cates, J., Whitaker, R.: Interactive, gpu-based level sets for 3d segmentation. Medical Image Computing and Computer-Assisted Intervention-MICCAI 2003 (2003) 564–572
11. Lin, Y., Medioni, G.: Mutual information computation and maximization using gpu. In: Workshop on Computer Vision using GPUs. (2008)

12. Catanzaro, B., Su, B., Sundaram, N., Lee, Y., Murphy, M., Keutzer, K.: Efficient, high-quality image contour detection. In: Proc. ICCV. (2009)
13. Maire, M., Arbelaez, P., Fowlkes, C., Malik, J.: Using contours to detect and localize junctions in natural images. In: Proc. CVPR. (2008)
14. Arbeláez, P., Maire, M., Fowlkes, C., Malik, J.: From contours to regions: An empirical evaluation. In: Proc. CVPR. (2009)
15. Sinha, S., Frahm, J., Pollefeys, M., Genc, Y.: GPU-based video feature tracking and matching. In: EDGE, Workshop on Edge Computing Using New Commodity Architectures. Volume 278., Citeseer (2006)
16. Heymann, S., Maller, K., Smolic, A., Froehlich, B., Wiegand, T.: SIFT implementation and optimization for general-purpose GPU. In: Proc. WSCG. (2007)
17. Catanzaro, B., Sundaram, N., Keutzer, K.: Fast support vector machine training and classification on graphics processors. In: Proceedings of the 25th international conference on Machine learning, ACM (2008) 104–111
18. Garcia, V., Debreuve, E., Barlaud, M.: Fast k nearest neighbor search using gpu. In: Workshop on Computer Vision using GPUs. (2008)
19. Wojek, C., Dorkó, G., Schulz, A., Schiele, B.: Sliding-windows for rapid object class localization: A parallel technique. Pattern Recognition (2008) 71–81
20. Fung, J., Mann, S.: OpenVIDIA: parallel GPU computer vision. In: Proceedings of the 13th annual ACM international conference on Multimedia. (2005) 852
21. Yu, Q., Medioni, G.: A gpu-based implementation of motion detection from a moving platform. In: Workshop on Computer Vision using GPUs. (2008)
22. Murphy-Chutorian, E., Trivedi, M.M.: Particle filtering with rendered models: A two pass approach to multi-object 3d tracking with the gpu. In: Workshop on Computer Vision using GPUs. (2008)
23. Carreira-Perpinán, M.: Acceleration strategies for Gaussian mean-shift image segmentation. In: Proc. CVPR. (2006)
24. Fukunaga, K., Hostler, L.D.: The estimation of the gradient of a density function, with applications in pattern recognition. IEEE Trans. on Information Theory **21** (1975)
25. Sheikh, Y.A., Khan, E.A., Kanade, T.: Mode-seeking by medoidshifts. In: Proc. CVPR. (2007)
26. Vedaldi, A., Fulkerson, B.: VLFeat - an open and portable library of computer vision algorithms. In: Proceedings of the 18th annual ACM international conference on Multimedia. (2010)