Dense Image Over-Segmentation on a GPU

ECE1724S Project Report
Alex Rodionov

Introduction

Image segmentation is a widely used technique in the field of computer vision that involves breaking up an image into arbitrarily shaped segments, typically in an attempt to recover the shapes of scene objects and features.

Dense image over-segmentation refers to when the image is intentionally broken up into hundreds or thousands of small segments, more than necessary to perform a segmentation into the objects in the scene. Typically they are uniform in size, and commonly referred to as superpixels.

Computer vision and machine learning algorithms can benefit from using a superpixellized image as input instead of the original image because of the reduced size of the problem as well as the shape of the superpixels already capturing some shape information from the scene.

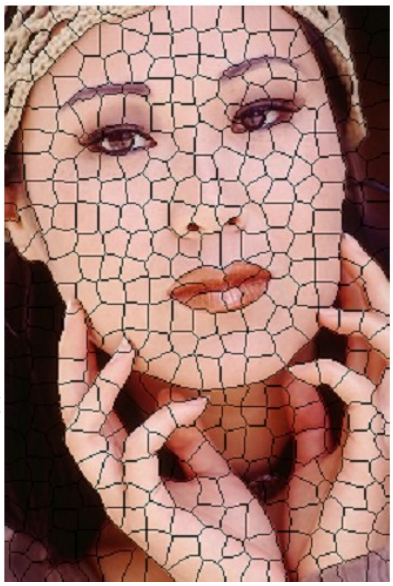

Figure 1: Example TurboPixels segmentation. Source: [1]

The goal of this project is to take such an algorithm and implement it on a GPU using CUDA in the hope of accelerating it such that it theory it could operate at interactive frame rates on moderately sized images. The TurboPixels[1] algorithm was chosen as the baseline, as it is faster than other algorithms such as Normalized Cuts[2] while providing as high or higher quality results. Figure 1 illustrates an example segmentation produced by TurboPixels.

At a high level, TurboPixels segments the image by planting seeds throughout the image which grow into superpixels over time, with the local speed of boundary expansion influenced by the edge strength of the underlying image.

Despite being faster than other algorithms, a software implementation of TurboPixels still takes at least several seconds to process an image. A modest performance target of 200ms (5 frames per second) for a standard sized image of 640x480 will be chosen, and the performance of the implementation compared to this target.

Related Work

There exists previous work on GPU-accelerated segmentation, but it has mostly been focused on volume rather than image segmentation and the extraction of a single feature rather than over-segmentation into superpixels.

For example Klar[3] used the same principle as TurboPixels of seeded region growth and level set evolution applied to 3D volumetric medical data. Lefohn and Cates[4] present a similar application. Both of these previous works were implemented on older graphics hardware that required the use of pixel and vertex shader interfaces.

Algorithm and Implementation

Overview

TurboPixels works by evolving a scalar function $\Psi(x, y, t)$ over time. This function is initialized as the signed distance to the nearest superpixel seed locations in the image. At any timestep, the superpixel boundaries are located on the level set $\Psi = 0$, and at every time step, the evolution of Ψ has the effect of expanding the boundaries and enlarging the superpixels.

As the boundaries evolve, boundary points slow down as they encounter edges in the image, and stop completely when they collide with another boundary. Once little to no forward expansion occurs in the image, the evolution is terminated and any regions of the image not yet covered by a superpixel are claimed by the nearest superpixel.

In the CUDA implementation, the image, and the Ψ and speed functions are discretized as pixels and are realized as linear buffers in device memory, and the different stages of the algorithm are implemented as kernels that operate on these image-sized buffers, usually with one thread allocated per pixel in 16x16 blocks. The buffers are copied to CUDA Arrays and used as textures when a kernel needs to read values in a nontrivial access pattern.

The following sections explain the algorithm and its stages in more detail.

Pre-processing

The input image, a 24-bit BMP file, is first converted to normalized floating-point grayscale to capture its intensity, which will be later used to place the seeds and calculate the local evolution speed.

The grayscale image is smoothed by repeated iterative application of a smoothing kernel. This removes noise and suppresses excessively sharp features. This iterative smoothing is calculated at each image pixel as $I(x, y, t+1) = I(x, y, t) + \Delta t \cdot \frac{(I_{xx} I_y^2 - 2 I_x I_y I_{xy} + I_{yy} I_x^2)}{(I_x^2 + I_y^2 + \epsilon)}$ where I is the image and $I_{??}$ are its partial derivatives taken using differences between surrounding pixels, and epsilon is a small floating-point number. The smoothing kernel is executed 10 times with a timestep of 0.1.

Initialization

Two things are done at initialization: the calculation of the affinity map, and the initialization of Ψ and the Assignment Map A.

The affinity map is an image-sized buffer that at every pixel holds a function of the edge strength that will be used to affect the speed of a superpixel boundary as it crosses that point. It is calculated once at the beginning of the algorithm: $\Phi(x, y) = e^{-E(x,y)/\upsilon}$ where $E(x, y) = \frac{\|\nabla I\|}{G_\sigma * \|\nabla I\| + \epsilon}$ is the magnitude of the gradient of the smoothed grayscale image (calculated by one kernel pass) divided by a gaussian-blurred version of itself to perform contrast normalization. The sigma used for the gaussian blur is proportional to the desired number of superpixels and thus the average distance between them.

Essentially, E is a constrast-normalized edge strength, so the affinity function will be high in areas of the image with no edges and small in areas with edges. This will be used to slow down superpixels as they approach edges.

In addition to calculating Φ, its gradient $\nabla\Phi$ is also pre-computed since it will be used during evolution to attract superpixel boundaries to the edges. Φ, and the x and y components of a $\nabla\Phi$ are stored in three separate floating-point buffers.

After Φ is calculated, it is used to help place the seeds for the superpixels. The seeds are initially placed as a uniform regular grid of points and then each seed is moved to the local maximum of Φ in its small surrounding area. This is done to prevent placing a seed on a strong edge in the image, because then it would not evolve fast or at all.

A single CUDA kernel takes the affinity map as input and outputs an array with x/y coordinates of superpixels. Since there are only hundreds to a few thousand seeds, this kernel parallelizes the problem at a coarse grain by assigning one thread per seed, and that thread scans the nearby pixels surrounding that seed for the local maximum affinity.

After the seed locations are known, the distance function Ψ is initialized along with the Assignment Map. For each pixel in the image-sized buffer Ψ, another CUDA kernel iterates through all the superpixel seeds, finds the closest one, and sets the value of $\Psi(x, y)$ to be that distance. The superpixel locations from the previous kernel are passed to this one through GPU Constant Memory to increase performance.

When initializing the distance function, it is offset by a constant such that at the location of each seed, the distance is -1.0 instead of 0.0. This will ensure that negative values of Ψ exist and will continue to exist during evolution and can be used to determine which pixels are inside superpixels.

A, the Assignment Map, is an image-sized buffer of integer values, with each pixel holding the ID of the superpixel it belongs to (with 0 representing no assignment). Initially, it is all zero with only the seed locations containing distinct nonzero values.

Evolution

The main loop of the algorithm evolves Ψ and A forwards by one timestep. As Ψ evolves, each superpixel will grow and claim more area thus updating A, changing the IDs of each newly claimed pixel to that of its nearest superpixel. Figure 2 shows how the values of Ψ and A look like when inside and outside a superpixel.

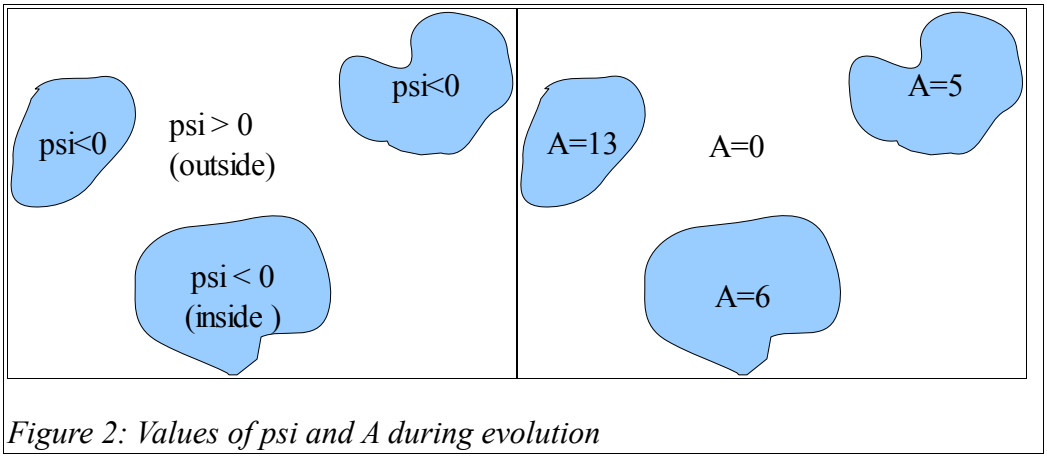

Figure 2: Values of psi and A during evolution

The evolution of Ψ follows this equation:
$\Psi^{t+1}(x,y)=\Psi^{t}(x,y)-S(x,y)\cdot\|\nabla\Psi^{t}(x,y)\|\cdot\Delta t$ where S is the speed:
$S(x,y)=[1-\alpha\kappa(x,y)]\phi(B(x,y))-\beta[\nabla\Psi^{t}(x,y)\cdot\nabla\phi(B(x,y))]$ and
$\kappa=\frac{\Psi_{xx}\Psi_{y}^{2}-2\Psi_{x}\Psi_{y}\Psi_{xy}+\Psi_{yy}\Psi_{x}^{2}}{(\Psi_{x}^{2}+\Psi_{y}^{2})^{\frac{3}{2}}}$

In addition, the Assignment Map A(x,y) is also updated:
$A^{t+1}(x,y)=\begin{cases} A^{t}(B(x,y)), \Psi^{t+1}(x,y)\le 0 \\ 0 \quad otherwise \end{cases}$

B(x,y) returns the nearest superpixel boundary point to (x,y).

The first term of S depends on the affinity function's value and makes the speed decrease when on an edge (recall that the affinity function is a function of the edge strength of the original grayscaled image). The kappa curvature term has the effect of smoothing out the shape of the boundary as it grows, and is calculated using partial derivatives of the distance function Ψ. Alpha is a constant that is set to 0.3 for this implementation.

The second term of S attracts the superpixel boundary towards areas of low affinity (where edges exist).In contrast to the first term, it can actually be negative and pull back the boundary towards an edge after it has already passed it. It is equal to the dot product of the *gradient* of the affinity function (precalculated during initialization) and the gradient of psi, which represents the outward normal of the superpixel boundary curve.

While Ψ is a two-dimensional function that is defined over the whole image, we only really care about the superpixel boundaries which are defined at $\Psi=0$. For this reason, pixels that are NOT on the boundary inherit the affinity function value of the closest point on the boundary. This point is returned by the Indirection Map B(x,y), which must be recalculated every timestep as superpixels change shape and claim more area. B is a two-dimensional array of 'short2' vectors.

If an image pixel becomes part of a superpixel after this timestep (it is on the boundary or inside, which happens when psi(x,y) <= 0), it is assigned the superpixel ID of the closest superpixel, which is again done through indexing via the Indirection Map B. This is how the Assignment Map A is updated.

Finally, to handle superpixel boundary collisions, each thread, which is assigned to a single pixel (x,y) will halt evolution and not update A or Ψ if A(x,y)>= 0 (this pixel is part of a superpixel) and an adjacent pixel is also part of a superpixel.

In order to calculate B, a GPU-friendly and highly parallel algorithm called Jump Flooding[5] is used. It works similarly to a shortest-path algorithm, with pixels propagating their closest-known-point value to their 8-neighbors in decreasing power of 2 strides and each receiving pixel selecting the closest of the 8 points it receives. The algorithm requires log(max(imgwidth,imgheight)) kernel invocations plus an initialization step. Please see the paper for more details. The rest of the timestep evolution outlined above happens in a single kernel invocation, including the calculation of the gradient of Ψ .

Termination

Evolution continues until the percentage of new covered pixels every timestep drops below a threshold. In order to count the number of covered pixels, a parallel reduction on the assignment map A is used. Once the relative increase in new pixels drops below 0.1%, and at least half of the image pixels are covered (this is to prevent termination at the beginning), the evolution loop exits.

Post-processing

In general, not all pixels will belong to a superpixel at the end. To fix this, any unassigned pixels inherit the Superpixel ID of the closest superpixel. This is done by calculating the indirection map B again and setting A(x,y) to be A(B(x,y)) with a single kernel invocation.

Finally, to extract the superpixel boundaries, an edge detection kernel is performed on the assignment map A and the resulting binary image is overlaid over the original colour image and written to a BMP file.

Results and Evaluation

The algorithm was implemented in C++ and CUDA as an executable that receives a BMP file as an input and outputs a segmentation using 1500 superpixels as another BMP file. Three images of 640x480 resolution were used to give the results below. The experiments were run on on an Intel Core 2 Quad Q9550 @ 2.83GHz with an NVIDIA GTX280 GPU. A fixed timestep of 0.5 was used for evolution.

Figure 3: First image. Time: 443ms

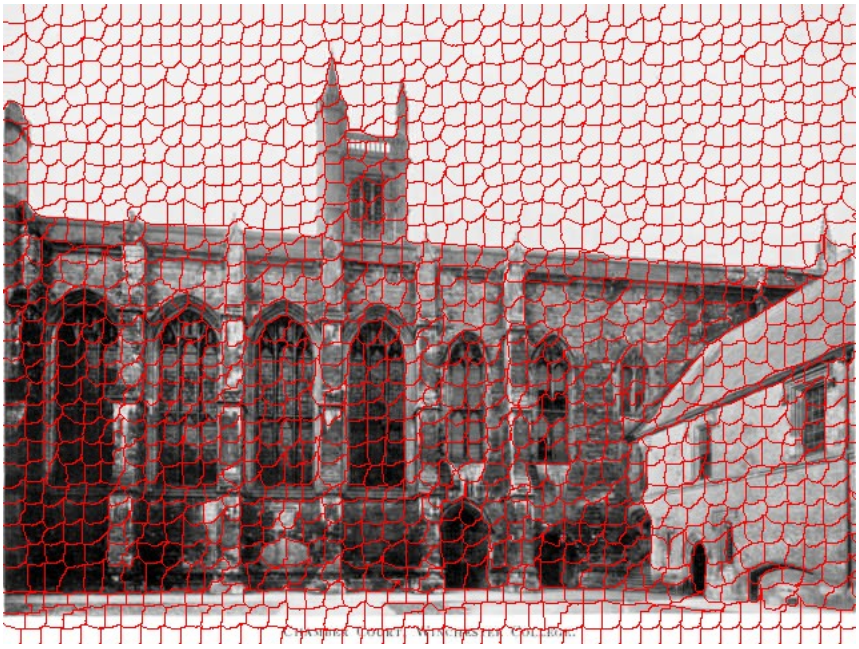

Figure 4: Second image. Time: 430ms

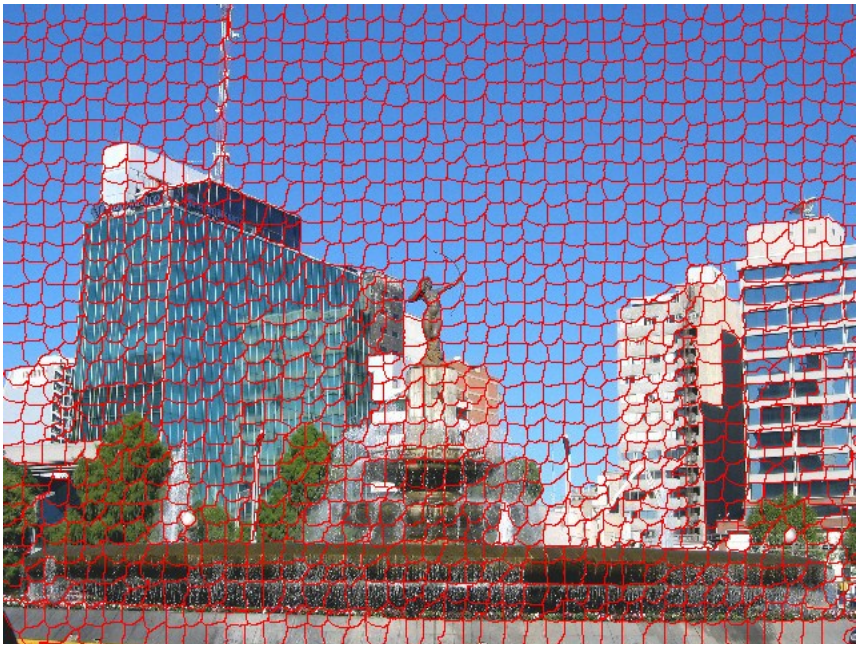

Figure 5: Third image. Time: 492ms

Segmentation Accuracy

The segmentation results can only be qualitatively judged – whether or not the superpixel boundaries represent a superset of the edges of the objects in the scene. For the most part, this appears to be the case, and the superpixels are roughly uniform in size which is a good thing.

Performance

The average time to process an image was 455ms, or 2.2 frames per second, which is below the target performance of 5 frames per second. However, it is within an order of magnitude of the goal.

Conclusions and Future Work

A TurboPixels-like algorithm was implemented on a GPU and achieved an image processing rate of 2.2FPS on 640x480 images. While it is somewhat far off from being used in realtime processing (in 30FPS video for example), there is still room for improvement in terms of performance optimization.

The biggest such optimization is algorithmic, and was in fact described and used in the original TurboPixels implementation in [1]. Instead of evolving the entire distance function every timestep, the idea was to only look at the area comprising a narrow band around the current superpixel boundaries, and then recalculate the position of this narrow band once the boundaries escape outside of it. A GPU-friendly version of this could be implemented to only update square blocks of the distance function that are known to contain part of the boundary.

Another possibility for improving performance is to converge on a solution using less iterations by using variable-length timesteps, based on how much new area was covered since last frame.

The overarching goal is to be able to run this algorithm on video, but performance is only one component. Even if this algorithm could be made to run at 30 frames per second, it may still not be 'correct' to simply treat each frame as an independent image, since the segmentations from frame to frame may not necessarily track the same objects or remain consistent over time. Future work would involve extending the algorithm to make it aware of such things.

References

[1] A. Levenshtein, A. Stere, K. Kutalakos, D. Fleet, S. Dickinson, and K. Siddiqi. Turbopixels: fast superpixels using geometric flows. *Unpublished.*

[2] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Trans. PAMI,* 22(8):888-905, 2000.

[3] O. Klar. Interactive GPU based segmentation of large medical volume data with level sets. *CESCG 2007.*

[4] A. Lefohn, J. Cates, R. Whitaker. Interactive, GPU-based level sets for 3D segmentation. *Medical Image Computing and Computer-Assisted Intervention,* 2003:564-572.

[5] G. Rong and T. Tan. Jump flooding in GPU with applications to Voronoi diagram and distance transform. *ACM i3d,* 2006:109-116.