# Engraver: A Parallel Image Segmentation Tool

Team members:

Dustin Liu (kaigel)

Karen He (rhe1)

**URL:** https://dukaige.github.io/kaigel_rhe1_418project/

## Summary

We are going to implement a parallel graph segmentation tool based on Canny edge detection. We will be working with CUDA on GPU to parallelize the edge detection and edge linking processes

## Background

Our sequential algorithm for edge detection is based on Canny's Edge Detection algorithm developed by Professor John Canny and optimized by many researchers.

The algorithm is divided into a few steps:

Step 1: We take in an image, and we will convert the image to grayscale. The grayscale image shouldn't change the position of edges.

Step 2: The second step Gaussian Blur to remove noises before further processing the image. Gaussian blur is a convolution operation on the original image with a typical Gaussian kernel.

Step 3: The third step involves calculating the gradient in both horizontal and vertical directions on the image for each pixel. This step can also be performed by calculating the convolution using a gradient kernel on each pixel on the image. We will also calculate all magnitude of gradient and angle of gradient at each pixel.

Step 4: The fourth step is non maximum suppression. By the image obtained from the magnitude of gradient at each pixel, we will have a resulting image with a thick edge.

Non-maximum suppression is a set of operation performed on each potential edge candidate pixel. The pixel is only considered to be an edge pixel if it is a local maximum compared to its neighbors.
Step 5: Double thresholding sets a low and high threshold on the determination of edges. A pixel with grayscale value below the low threshold will not be considered an edge pixel, and above the high threshold will be nailed as a “strong edge” pixel. Everything in between are called weak edges, and they will be considered candidates for Step 6.
Step 6: Given a set of weak edges, we will determine for each weak edge by whether its connected to a strong edge in step 5. Now we have a complete image of all edges.
Step 7: After we generate all the edges, given a specific point in the image, we find an edge surrounding the point, and sequentially follow the edges in a cycle to form a contour. If edges become disconnected, we will find the nearest other edge in the range.

We believe this task can significantly benefit from parallelism, since for the most part the dependencies between pixels are small. The job we do on each pixel only depends on its immediate neighbors, so we can utilize CUDA’ s block shared memory feature to accelerate data access.

**The Challenge**
One important bound on the performance of this parallel implementation is the amount of memory access, since a worker thread will be assigned to each pixel in the image, and therefore repeatedly accessing the pixel data in the image will cause a large amount of memory access. In terms of the communication cost, each step will involve working on each single pixel, and examining the pixels around the pixel. Therefore, constant communication with the nearby pixels is unavoidable. Besides, another challenge is after we decide on a set of pixels to be potential candidates of the final edges, since the potential edges are very unlikely to be consecutive in the image array,

and we only assign a worker to a pixel if the pixel is a potential candidate, very low locality can be used at this step.

Another potential challenge is in step 6 and 7, there is an internal order in determining the connected components and determining the contour from the set of edges. The first problem is solvable using a Bouruka like MST generating algorithm, and the second problem will require a new parallel algorithm design to allow parallelism.

The edge finding problem itself turns out to be a highly parallelizable task in general, since the each pixel can be mostly handled separately during the process. However, the contour forming process will require connecting different component of edges, and therefore requires internal ordering between different edges. And therefore, we will be exploring different parallelization strategies on the contour forming part.

**Resources**

We will be starting from scratch, without using any data processing platforms such as OpenCV. Our major reference resource would be John F. Canny's paper, *A Computational Approach to Edge Detection*. We will also reference the book *Machine Vision* by Ramesh Jain, Rangachar Kasturi and Brian G. Schunck, especially Chapter 4 (Image Filtering), Chapter 5 (Edge Detection), and Chapter 6 (Contours).

**Goals And Deliverables**

We plan to achieve a parallel implementation of the Canny edge detection algorithm with at least 10x speedup. After we extract edges of an image, we will achieve image segmentation by linking edges to constitute closed-boundary contours. If our image segmentation implementation turn out performant, we also hope to explore various applications of image segmentation, such as object cutout, and even object detection. In case the work goes more slowly, we will confine our work to visualization and performance analysis of our image segmentation algorithm. It is hard to state precise performance goals at this time, but we believe a 10x speedup is a reasonable benchmark since this speedup is common in many image processing technique optimizations, and we have not discovered many inherent dependencies in the task. Still, the actual speedup performance depend on many factors, such as the size, the complexity, and degree of sharpness of the processed image.

During poster session, we plan to show the image segmentation outputs of our program for a number of pictures, as well as the speedup graph. We also plan to show several interesting application of our image segmentation program, such as object cutouts and object detection.

**Platform Choice**

We will be running our solution on Dustin's laptop equipped with GTX 1070. We will implement the program with C++ using CUDA. We will be using CUDA because the number of independent tasks in this application is quite large, specifically each pixel can be considered as an independent unit of computation. Additionally, since each pixel is only dependent on pixels around itself, we can utilize the CUDA block feature to utilize shared memory and improve performance of memory access overall.

**Schedule**

| | |
|---|---|
| Oct 29 | 1. Find and read papers<br>2. Get familiar with algorithms and theoretical background |
| Nov 5 | Implement sequential version of edge detection |
| Nov 12 | Parallelize Gaussian filter, non-maximum suppression |
| Nov 19 | Parallelize double thresholding |
| Nov 26 | Parallelize edge tracking by hysteresis and edge linking |

| | |
|---|---|
| Dec 3 | Code clean up and performance analysis |
| Dec 10 | Prepare poster and demo materials |