Accelerating Image Convolution for Edge Detection

Noah Vaden
Undergrad Student, Dept. of Computer Science
University of Central Florida
Orlando, FL, USA
no572033@ucf.edu

Cameron Dudeck
Undergrad Student, Dept. of Computer Science
University of Central Florida
Orlando, FL, USA
ca535398@ucf.edu

Euan Selkirk
Undergrad Student, Dept. of Computer Science
University of Central Florida
Orlando, FL, USA
eu128372@ucf.edu

Jason Laveus
Undergrad Student, Dept. of Computer Science
University of Central Florida
Orlando, FL, USA
ja005353@ucf.edu

***Abstract*—This paper will describe the process in which we (students listed above) programmed a Canny Edge Detection Algorithm to detect and display prominent and necessary edges that create an outline of the major objects in an image. We then have then improved the runtime of the algorithm by implementing parallelization concepts that will allow simultaneous calculations. This speed increase will allow for better edge detection use in things that need faster calculations such as video stream edge detection.**

I. Introduction

A. Process

The Edge detection algorithm that we have chosen to enhance with parallelization is the Canny Edge Detection algorithm. This algorithm can be broken into five major sections that process the image sequentially. In order, these sections are the loading of a black and white image, applying a gaussian filter, calculating the intensity gradients of each pixel, applying a magnitude threshold, and finally processing the remaining edges through hysteresis. We specifically chose the Canny algorithm due to its more complex and time consuming yet more accurate running. This will allow the magnification of the improvements we implement using parallelization concepts. In order to distribute the work among members effectively we have each taken a separate major portion of the functionality of the algorithm and as such, each function will be described in the first couple sections of this paper with both how the function works normally and how parallelization techniques were applied through multithreading.

B. Motivation

The motivation and reasoning behind wanting to speed up the process of this edge detection algorithm is to allow for quick succession calculations of edges in images. An example of a use case in which one might want this is for displaying the prominent edges in a video. This process would need quick edge detection calculations per frame to increase frame rate for a smooth video.

II. Edge Detection Functions

A. Get Image

This is the first step in the Canny Edge Detection process as this is where we grab our input image from a file and convert to a format which is easy to use and modify during the rest of the algorithm. For the initial processing of the image from a file we are using stb_image library which processes the image file into an unsigned char array that stores the color data for each pixel in the image. The next step in this function converts the image to grayscale by averaging the rgb color values. This single calculated value also represents the intensity of a pixel. Instead of storing the pixel values back into an unsigned char array, we have made a Vector2 struct that will store the intensity in the x and the gradient (which will be set to 0 at the moment) in the y. We then store this array of Vector2s along with the height and width of the image into our own custom IntensityGradientImage struct to ensure that we have a universal, easy to use image format that will allow ease of manipulation throughout the algorithm.

B. Gaussian Filter

A Gaussian filter is a smoothing filter used in image processing and computer vision to reduce noise and blur images. It works by convolving an image with a Gaussian function, which is a bell-shaped curve defined by:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where x and y are the spatial coordinates, and σ (standard deviation) controls the spread of the filter. The function first creates a kernel based on the Gaussian function. Larger σ results in stronger blurring. Then we Apply Convolution operation. For each pixel (x, y), the algorithm multiplies the surrounding pixels with corresponding kernel values and sums the results. This operation is performed only on valid pixels, (ignoring edges where a fill 5x5 region is not available).

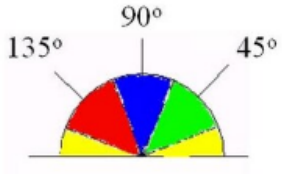
Gaussian Filtering is Useful because it performs Noise Reduction which removes random variations in intensity. Smooth the image and help create a gradual transition

between intensity variations. Finally It Preserves the Structure of the image. Unlike box filters, Gaussian filters give more importance to the center pixel, preserving edges better.

In summary, this function smooths an image while maintaining important structures, making it ideal for preprocessing before further analysis.

C. Intensity Gradient

The intensity gradient function is the next big stepping stone in setting up the image for having the strongest edges chosen. This function goes through every pixel and calculates the gradient for each one. The gradient of the pixel describes the direction that an edge is likely going in when it hits this pixel. However, the calculated gradient is not an exact angle but instead a rounded degree that will be to the nearest approximate line of horizontal, vertical, and the two diagonals (0°, 45°, 90°, 135°).



This will allow the future functions to choose and fit pixels together to figure out where the strongest lines are located and display those.

D. Non-Maximum Suppression

This function implements Non-Maximum Suppression (NMS), a crucial step in producing high-quality Canny edge detection images. NMS refines detected edges by retaining only the local maxima in the gradient magnitude along the direction of the gradient, suppressing all other pixels. It determines whether a pixel is a local maximum in its gradient direction and either retains or suppresses it. The function suppressAtPixel performs Non-Maximum Suppression for each pixel.

Gradient Retrieval: The function retrieves both the gradient magnitude and the gradient direction of the current pixel. The direction is normalized to the range [0,100) to simplify comparisons.

Direction Quantization: The gradient direction is quantized into one of four sectors:

- Horizontal (left–right): Angles close to 0° or 180°
- Diagonal (↙ / ↗): Angles close to 45°
- Vertical (top–bottom): Angles close to 90°
- Diagonal (↖ / ↘): Angles close to 135°

Neighbor Identification: Based on the quantized direction, the function identifies the two neighboring pixels along the gradient direction (referred to as neighbor1 and neighbor2).

Local Maximum Check: If the current pixel's gradient magnitude is greater than or equal to both neighboring pixels, it is retained in the output array. Otherwise, the pixel is suppressed by setting its magnitude to 0.0.

Fig. 1. Before non-Maximum Suppression

Fig. 2. After non-Maximum Suppression

E. Magnitude Threshold

The magnitude threshold function applies gradient magnitude thresholding to filter weak edges in an image, based on a given threshold value. It uses the IntensityGradientImage structure, which stores gradient vectors for each pixel. The function retrieves image dimensions and the pointer to the gradient array. Each pixel's gradient magnitude, stored in the x-component of the gradient vector, is compared to the threshold. If the magnitude is below the weakThresh value, the gradient vector is suppressed by setting both its x and y components to zero, removing weak edges from the image. This approach allows for only strong edges to contribute to the final edge map, increasing the accuracy of the edge detection.

Fig. 3. Before Magnitude Threshold

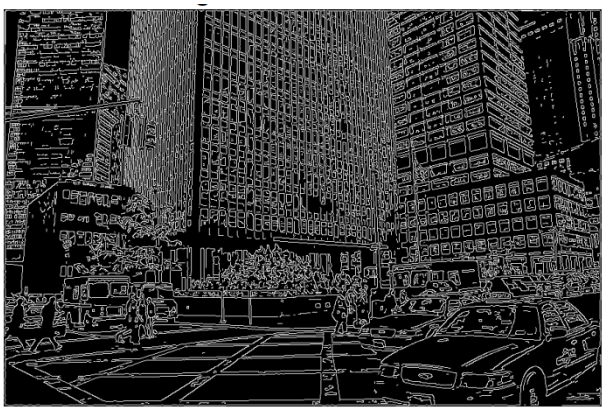


Fig. 4. After Magnitude Threshold

While this process results in a loss of finer details and depth, effectively reducing multiple edge strengths to a single binary distinction, it significantly improves the clarity of strong edges in the image.

F. Hysteresis

The hysteresis function builds upon the process of removing weak edges started in the magnitude threshold function by going through the remaining edges with a more "fine toothed comb" so to speak. It starts with having a classification for strong edges. These are edges that have been determined to be strong enough that we want to keep them no matter what. This function then proceeds to navigate along the edges formed through other pixels that follow this starting strong edge and if they haven't already been removed the will be kept as edges for the final product. This process continues until all strong edges have been looped through to find their connected important edges and all other edges are deemed noise and thus removed.

Fig. 5. After Hysteresis

G. Save Image

The final function we have included is made for the sole purpose of outputting the image after its been modified. This means that this function is not officially part of the Canny Edge Detection algorithm, but is instead something we have added to formulate an output from our custom image struct. The first step is to break down our custom image struct back into an unsigned char array representing the grayscale pixel values of the image. This is necessary because of the stb_image library that we use for image input and output functions on these unsigned char arrays. We then simply call the write function from the library to write the image to an output file where we can see the progress of the image. This function will be very useful for seeing the output and for inspecting the image at the end of each function to ensure that everything is running correctly and just to have the steps of the process in a visual format.

III. Problems with the functions

An overall theme of all the functions described above is the poor optimization of their bases. The amount of nested for loops greatly increases the runtime and cost of the program. Specifically, the magnitude threshold function holds a large sum of runtime on the program. Since it is checking each pixel, and there can be a lot of them, the function tends to slow the overall runtime down.

Another problem identified in the making of our program (before multithreading) is within the intensity gradient function. This function, similar to the magnitude threshold, goes through every pixel to calculate gradients. Since this function is performing calculations for each pixel, the cost and runtime are significantly increased over one thread. The specific issue, common to some other functions like magnitude threshold and gaussian filter, is that this is originally being accomplished through one thread. If we look at the gaussian filter method, it has an even more complex calculation to make on each pixel (disregarding irrelevant ones).

To reduce the runtime, we devised a plan to increase the amount of threads being used and divide the work among them using a few different methods described in the next section.

IV. Parallelizing the Functions

A. MultiThreading Implementation

The main technique that was used to implement multithreading to the program was the line of code "#pragma omp parallel for collapse(2)". This code flattens 2 nested loops into one loop for better parallelism, and also starts multiple threads and runs the block of code once per thread.". Because almost every function in the program uses a nested for loop we decided to implement #pragma into our project.

We made sure to split every function into its base function and its helper function. For example, the gaussianFilter() function will compute the gaussian kernel and normalize it. It will use a helper function named applyGaussianAtPixel() that will apply the gaussian filter on each pixel. GaussianFilter() function will first call forEachPixel2D() function, and then the forEachPixel2D() function will call the helper function applyGaussianAtPixel().

The main function used for multithreading functionality is the forEachPixel2D() function which iterates over the 2D range of pixels. The function supports single-threaded and multi-threaded execution, depending on the multithreaded parameter. If multithreaded is true, the function uses OpenMp to parallelize the nested loop. Single-threaded execution runs if multithreaded is false, and just processes the pixels sequentially using standard nested for loops. This is simpler and avoids the overhead of thread management. The PixelFunc parameter allows us to define any operation to be performed on each pixel. This makes the function flexible and reusable, for various tasks such as filters, computing gradients, or modifying pixel values. etc.

B. Get Image

Since getting the image is a relatively simple and not computation heavy function, a very simple multithreading solution was able to slightly increase runtime. This solution was applied to the heaviest part of the function that I could apply it to, which is the conversion of the original image into grayscale and transforming it into the IntensityGradientImage struct format. The original solution was to create a thread for every pixel since there is no particular order that the pixels have to be done in. However, the overhead caused by making a thread for each pixel was too much and caused a dramatic decrease in speed. To fix this, we gave each thread a batch of pixels to compute to reduce the amount of threads used, but still having the benefits of multithreading. Overall, these changes only increased the speed by a couple milliseconds per image since the base implementation without multithreading was very simple and not computation heavy.

C. Gaussian Filter

First the forEachPixel2D() function will call the applyGaussionAtPixel() function on multiple threads at the same time. Each thread will handle a different pixel.

The function initializes the sum to 0.0. It will accumulate the weighted sum of the pixel intensities in the 5x5 neighborhood. The function iterates over a 5x5 window centered on the pixel (y,x). The offsets ky and kx range from -2 to 2, representing the relative position of neighboring pixels. For each neighbor, the function calculates it 1D index in the sourcImg array using the specific formula (y + ky) * width + (x + kx). After the convolution is complete, the computed value (sum) represents the smoothed intensity for the pixel (y,x). This value is stored in the x component, of the corresponding Vector2 object in the destImg array.

The y component of the Vector 2 object is set to 0.0, as the Gausian filter does not compute gradient direction. These functionalities are created by utilizing multiple threads.

In summary, The applyGausianAtPixel function performs a convolution operation using a gaussian kernel to smooth the intensity of a single pixel. By averaging the intensities of the pixel and its neighbors with Gaussian weights, the duction reduced noises and preserve important image features. This operation is applied to every pixel in the image as part of the larger Gaussian filtering precess, which is implemented using either a nested loop, or a parallelized utility in forEachPixel2d.

D. Intensity Gradient

Utilizing forEachPixel2D() to apply multithreading first, the function computeSobelAtPixel() is executed on multiple threads. First the function computeSobelAtPixel() initializes the gradient. The variables gx and gy are initialized to 0.0 and will accumulate to horizontal and vertical gradient values.

The function then convolutes with a Sobel Kernel. The function iterates over a 3x3 window centered on pixel (x,y). The offside ky and kx range from -1 to 1, representing the relative position of neighboring pixels. For each neighbor the function will calculate its 1d index in the src array using the formula (y + ky) * width + (x + kx). The intensity of the neighbor **(src[pixelIndex].x)** is multiplied by the corresponding wight in the Sobel kernel **(Gx[ky + 1][kx + 1]** and **Gy[ky + 1][kx + 1])**, and the result are then added to gx and gy. We then compute the magnitude and the direction.

The gradient magnitude is calculated as **sqrt(gx * gx + gy * gy)**, which represents the strength of the edge at the pixel.
The gradient direction is calculated as **atan2(gy, gx) * (180.0 / M_PI)**, which gives the angle of the edge in degrees.

The computed magnitude and direction is stored in dst array at the appropriate pixel index **(y * width + x)** as a Vector2 object. The x component of the Vector2 store the

magnitude while the y component stores the direction.

In summary, the computedSbelAtPixel function applies a Sobel operation to a single pixel, calculating its gradient magnitude and direction based on the intensity value of its 3x3 neighborhood. The Sobel kernel Gx and Gy are then used to calculate the horizontal and vertical gradients, which are then combined to determine the overall gradient properties. This function is called for every pixel on the image, (excluding borders) as part of a larger edge detection process. The results are stored in a Vector2 array, where each pixel's gradient information is encapsulated for further processing.

E. Non-Maximum Suppression

Utilizing forEachPixel2D() to apply multithreading first, the function computeSobelAtPixel() is executed on multiple threads.

First off, the function retrieves the gradient magnitude and direction. The function calculates the 1d index of the pixel Idx in the gradient array using the formula y * width + x. It then retrieves the gradient magnitude (inArr[idx].x) and direction (inArr[idx].y of the specific pixel.

We then normalize the gradient direction to the range [0,180) to simplify comparisons. This is because gradient direction is periodic, and angles outside this range can be mapped back into it.

We then Quantize the gradient Direction. Into one of four sections:

- Horizontal (left–right): Angles close to 0° or 180°.
- Diagonal (↙ / ↗): Angles close to 45°.
- Vertical (top–bottom): Angles close to 90°.
- Diagonal (↖ / ↘): Angles close to 135°.

Based on the quantized direction, the function identifies the two neighboring pixel along the gradient direction. For example: If the direction is horizontal the neighbor pixels are to the left and right of the current pixel. If the direction is diagonal (bottom left, upper right), the neighbor pixels are in the top-right and bottom-left directions.

We then check the local Maximum. The function compares the magnitude of the gradient of the current pixel with the magnitude of its two neighbors along the gradient direction. If the current pixel magnitude is greater than or equal to both neighbors, it is retained in the output array (outArr[idx]), with its magnitude and direction preserved. Otherwise, the pixel is suppressed by setting its magnitude and direction to 0.0.

In summary, the suppressAtPixel function ensures that only the strongest edges are retained by suppressing non-maximum pixels. This is achieved by comparing the magnitude of the gradient of each pixel with its neighbors along the gradient direction. The result is a refined gradient array where edges are thin and well-defined, which is crucial for accurate edge detection. This function is typically called for every pixel in the image (excluding borders) as part of a larger NMS process.

F. Magnitude Threshold

The parallelization of the magnitude threshold function was done to improve performance and efficiency when dealing with large images. Since every pixel in the image is processed independently during thresholding, this makes the function a perfect candidate for parallel processing.

To achieve this, OpenMP was used to parallelize the two nested loops that go through each pixel in the image. These loops were combined using the collapse(2) directive, which flattens the nested structure into a single, longer loop. This allows the work to be more evenly divided among multiple threads, which speeds up processing.

Each thread checks a pixel's gradient magnitude and sets it as a strong edge, weak edge, or non-edge based on the given thresholds. If the magnitude is strong, it's kept at full intensity. If it's weak, it's marked differently. If it's below the weak threshold, the gradient is essentially erased. Since this process doesn't require any pixel to interact with another, it's a safe and effective use of parallelism.

This method was chosen because it keeps things simple while giving a major performance boost. It allows the function to run faster without affecting the accuracy of the results, making it especially useful when working with high-resolution images or large batches of data.

Magnitude Threshold without MultiThreading

Test	Runtime(ms)
1	203
2	228
3	201
AVG	211

Fig. 6. Magnitude Threshold without MultiThreading Table

Magnitude Threshold with MultiThreading

Test	Runtime(ms)
1	193
2	187
3	196
AVG	192

Fig. 7. Magnitude Threshold with MultiThreading Table

This parallelization resulted in a speedup of approximately **1.10x** compared to the sequential version of the function. While this may seem modest, it still represents a meaningful improvement in efficiency, especially for larger images where processing time becomes more significant. The relatively small gain can be attributed to factors such as limited image size, thread overhead, or memory access patterns, but nonetheless it demonstrates the benefit of leveraging parallel computing in image processing tasks.

G. Hysteresis

Utilizing forEachPixel2D() to apply multithreading first, the function computeSobelAtPixel() is executed on multiple threads. The function computeSobelAtPixel() first identifies weak edges by calculating the 1D index of the pixel (idx) in the gradient array using the formula y * width + x. It processes only weak-edge pixels identified by a gradient magnitude of 128.0 in the x component of the input array (arr[idx].x). We then check connectivity to strong edges by iterating over the *-connected neighbors of the pixel using the offsets defined in dx and dy. For each neighbor, it calculates the 1D index (nidx) and checks if the neighbor is a strong edge (gradient magnitude of 255.0). If any neighbor is a strong edge, the pixel is then marked as a "connected to strong" by setting the connectedToStrong form to true and breaking out of the loop. We finally promote or suppress the Pixel. If the pixel is connected to a strong edge, it is promoted to a strong edge by setting its gradient magnitude to the output array (newArr[idx].x) to 255.0. Otherwise, the pixel is suppressed as a non-edge by setting both its gradient magnitude and direction in the output array (newArr[idx].x and newArr[idx].y) to 0.0.

In summary, the applyHysteresisAtPixel function ensures that weak edges are retained only if they are connected to strong edges, reducing noise and improving the accuracy of edge detection. By examining the 8-connected neighbors of each weak edge pixel, the function determines whether the pixel should be promoted or suppressed. This pixel-level operation is applied to all pixels in the image as part of a larger hysteresis thresholding process, which refines the detected edges and prepares the image for further processing or visualization.

V. Conclusion

In our research and implementation of this edge detector we have come to find that our plan in increasing the efficiency of the Canny Edge Detection algorithm using multithreading was successful. This increase in efficiency may allow edge detection algorithms to be more efficient in video settings where you must handle a stream of images. Having a more efficient algorithm means you could increase the frame rate of the output for a smoother video that displays the prominent edges of the frames provided.

A. Runtimes

Without MultiThreading

Test	Runtime (ms)
1	5486
2	5517
3	5462
AVG	5488

With MultiThreading

Test	Runtime (ms)
1	2477
2	2541
3	2636
AVG	2551

Fig. 8. Summary of runtimes

The data above shows some sample runs of the algorithm both with and without the multithreading active. The overall average improvement of the multithreading addition is about 215% faster. The majority of the improvements are a result of the pragma function described in subsection A of section III above, which cuts the runtime by about 50% across most functions.

References

[1] "Canny Edge Detection", https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html
[2] "Pragma Basics", https://www.gnu.org/software/c-intro-and-ref/manual/html_node/Pragma-Basics.html
[3] "Edge Detection in Image Processing", https://blog.roboflow.com/edge-detection/
[4] "GitHub Repository", https://github.com/RedDogCity/Edge-Detection