Parallelized Edge Detection

Rishi Gadhia
Undergrad Student, Dept.
Computer Science
University of Central Florida
Orlando, Fl, USA
ri384851@ucf.edu

Steven Camacho
Undergrad Student, Dept.
Computer Science
University of Central Florida
Orlando, Fl, USA
st201789@ucf.edu

Osa Naghise
Undergrad Student, Dept.
Computer Science
University of Central Florida
Orlando, Fl, USA
os279883@ucf.edu

Connor Price
Undergrad Student, Dept.
Computer Science
University of Central Florida
Orlando, Fl, USA
co882249@ucf.edu

***Abstract*—The following paper investigates the possibility of improving the speed of the Sobel Edge Detection algorithm through parallelization. The Sobel Edge detection algorithm is a technique commonly used in Edge Detection which lacks in performance compared to other more advanced algorithms. Specifically, this paper looks for improvements in runtime to make it a more competitive option for algorithm selection. Three methods of attempting this improvement were developed. A base program in which Sobel was implemented as a single-threaded program, following the traditional Sobel algorithm. They were each implemented in C++ and each implemented multi-threading as a way to parallelize the Sobel Edge Detection algorithm. Overall, each method offered a different result, with two of the said methods improving on the runtime of the base Sobel code, while one of the methods showed a worse runtime.**

***Index Terms*—Sobel, Multi-threading, Canny, Prewitt, Edge Detection**

I. Introduction

Edge detection software plays a significant role in fields such as computer vision, image processing, and machine learning. Various real-world applications such as medical imaging and autonomous vehicles would not be where it's at today if not for edge detection software. There are many approaches and techniques used in edge detection but the primary focus for us is to parallelize the Sobel edge detection algorithm for an improvement on runtime.

Sobel is one of the most common techniques used in edge detection and works by doing convolution to the image data matrix with two 3x3 Sobel kernels. We can then take the output of the convolution by the horizontal kernel as Gx and the output of the convolution by the vertical kernel as Gy. The next step in the algorithm is to compute the gradient magnitude G using Gx and Gy. G is compared to a previously defined threshold value T and if it is greater than T it makes up an edge otherwise it's not part of an edge. Finally, we can output the matrix of edges.

Sobel is one of the many options people have when selecting an edge detection algorithm. Two key factors of any algorithm are speed and effectiveness. When compared to other available edge detection algorithms, Sobel may lack in effectiveness as other options like Canny or Prewitt as they provide better edge detection results. To make it a more competitive option, Sobel can be looked to be improved upon in terms of speed. That possibility is what we will be investigating in this paper.

This technique can be done in parallel and there are many reasons why this algorithm can benefit in doing this. The most significant advantage to doing this algorithm in parallel is the speedup in runtime. Speeding up such a widely used algorithm will benefit many applications that use this edge detection technique.

There are also many ways to parallelize a traditional sequential Sobel algorithm and one of the goals of the research is to find the most efficient way to do this. We have developed and tested some methods to find out which one has the best outcome in the real world.

II. Methodology

In the goal of attempting to parallelize the Sobel edge detection, the Sobel edge detection algorithm was decomposed into what parts of said algorithm can be parallelized. The first step of this process was to first implement a "base" sobel edge detection program implementation in which the algorithm runs on one thread and with no type of attempt into parallelization. This program implementation would serve as the control group for experimentation, while also serving as the base for decomposition of the algorithm in search of what can be parallelized. With the base program completed, the next step in development is to develop the methods in which we parallelize the Sobel algorithm, which parts will be modified, and which parts will be kept the same. To satisfy the second step in this process, three different methods were developed. The three methods can be found below

*A. **Method A***

For this method, the first task is to divide the input image into patches that match in size to the Sobel kernels. Each patch will be accessible via a single coordinate point. There will be an N number of threads and each thread will be able to grab/access the patches via a given coordinate. Once given the coordinates, the thread will perform the Sobel algorithm on the selected patch and output the new patch onto a new image. The last two statements must have each thread be able to perform them in an unrestricted, independent manner from the neighboring threads. Finally, the threads will keep grabbing patches until all patches have been processed.

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A}$$

$$\mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

Fig. 1. Sobel Kernels, Gx and Gy[4]

B. *Method B*

For this method, the work is divided up by the number of columns that an image has, starting from the rightmost column. Each slice will have the sobel operation run on it from top to bottom (e.g. from row 0 of the slice to row n). On each pixel of each column an algorithm is run that calculates a patch of Sobel, so for one pixel the final result of Sobel (after thresholding) is calculated and returned to a final matrix. Threads will start and finish new columns until no columns remain in the image, each thread will process its column until completion (e.g. reaching row n of the column).

C. *Method C*

The final method will split the work of calculating the horizontal, vertical, and magnitude gradients as separate tasks that each have N threads working to accomplish. The image is split into N slices, with each slice covering the full width, but 1/N the height. The next step is to create N threads to calculate the vertical gradient for each slice, and N more threads to calculate the horizontal gradient for each slice. Waiting is done until the threads calculating the vertical and horizontal gradients finish. N threads are created to calculate the magnitude gradient for each slice.

III. Background

The Sobel edge detection algorithm was first devised in 1968 by Sobel and Feldman. It was originally aimed to be an edge detection technique that was computationally inexpensive in comparison to other techniques in that time. The algorithm can be divided into 4 simple steps.

A. *Convolution with Sobel kernels*

Figure 1 above shows the two kernels that are convolved with the input image. The algorithm will iterate pixel by pixel in the input image matrix and apply convolution with the kernel resulting in images G_x and G_y. Both images at each point contain the horizontal and vertical derivative approximations.

B. *Gradient Magnitude*

$$\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2}$$

The next step in the algorithm is the compute the gradient magnitude G given by the formula above.

C. *Comparing with defined threshold*

We can compare G to our defined threshold value T. If it is greater than T it becomes a part of the edge image, otherwise it is not part of the edge image. The threshold value is decided to be any value within the range of pixel values, and by trial and error we can determine the best value.

D. *Output image*

The last step of the algorithm is to output our image of edges. The value that results from the threshold comparison is added to the output image.

IV. Implementation

With the development of three possible methods for parallelization, each will have a different implementation/construction. Each method was implemented in C++. (Add other details concerning the common elements implementations of the methods) The three methods can be found below:

A. *Method A: Patch-wise Sobel Calculation*

The thread make-up is described by the following:

- Sobel Kernels will be used by each thread
  - each thread must have access to the SobelX (horizontal) and SobelY (vertical) kernels as each thread will need to use these masks to scan image for edges
- A counter will be shared among each thread
  - each thread must have access to the shared counter
- An output image will be shared among all threads
  - each thread must have access to the shared output image where they can return their processed patch from the input image to that output image
- Function that performs Sobel algorithm
  - Access to a function that performs the Sobel edge detection algorithm on the selected patch
- How do the threads know when to stop?
  - A function used to check whether all the patches have been processed or not
- Each thread will do the following when called:
  - When threads call the run function, they will run a loop, till all patches are processed, in which they use a "get coordinates" function to grab the next available patch and call the "Sobel function" on said patch

The shared counter between all the threads will function as follows:

- Patches will be located through coordinates
  - X and Y coordinate, used to locate the patch in the image that will be used by the thread
  - The counter will grab a new coordinate when called by an incoming thread in the order specified below
- Each patch will be grabbed through two singular coordinate values
  - essentially, a coordinate will look like (0,0) and the thread will take that and grab that pixel and the other

  8 pixels around that one pixel to make up the 3x3 patch

- Each patch is processed with the Sobel function
  - On this patch, the thread will apply the Sobel algorithm through a separate Sobel function
- Patches are not accessed through a random order
  - The order of going through the patches per how the counter works, considering that the image will be divided into patches of 3x3, then we go from the top left corner of the image, going from left to right at that level or row, and once we reach the right, we drop down to the following row and start again from the left side
- Counter needs to notify threads of availability
  - The counter will have a variable to signify the completion of going through all the possible patches
  - It will be checked as completed (the variable) once the last possible coordinates are selected and processed
- Coordinates will be accessed through a function
  - Function used by incoming threads to grab a new coordinate for selecting a new patch
- Threads will always check for availability
  - Function used by threads to check status of available patches

How the counter gives a coordinate to a thread:

```
getCoordinates(){
   synchronized (lock) {
       if (x and y = last patch)
          isSobelfinished=yes;
          break;
      else
          if (x=right side of the image)
              y+=1
          else go down a row
              x+=1
            return x and y
    }
}
```

What is being parallelized in this implementation compared to the original Sobel code is the method in which the pixels of the image are being processed/traversed through. This is done by changing the method of performing the Sobel edge detection algorithm on each pixel by using the proposed coordinate-grabbing method in which the patches/pixels to be processed are accessed independently by each thread and without any waiting time for the threads by which they can grab patches/pixels to perform Sobel edge detection continuously until all patches are processed, and this is done instead of processing/traversing the image pixels sequentially through a for-loop.

B. Method B: Image Slicing + Column-Wise Sobel Calculations

- The process here is started by loading in the image as grayscale (we let OpenCv do the work here)
- We then pass our arbitrary number of threads and threshold to our MultithreadedSobel constructor.
- Here we then pass our grayscale image in to be processed
  - We gather the number of rows and columns of the input image
  - We create our return image in the form of an OpenCv Mat
- Starting from the rightmost column the following steps occur
  - We create a thread via lambda that will process the column it was given.
  - For each pixel in said column, we will calculate the output of the entire Sobel algorithm
    - *This means finding the convolutions for the x and y gradient, approximating their magnitude, and checking the threshold value, the final value is then returned to be placed in its corresponding location in the return image.*
  - After a thread completes its column, it then moves on to the next (given that there are any remaining columns to process)
  - After all threads have completed their columns the final image is returned and the process is complete.

The code segment does the following:

The program makes a return matrix with the same type and size of the input image We then create a thread for each column of the image and for each thread we calculate the Sobel operation on each pixel of the column. Finally, we join all the remaining threads and return the result matrix.

```
Function performSobelEdgeDetection(image):
    imageRows = image.rows
    imageCols = image.cols
   retval = CreateMat(imageRows, imageCols,
    CV_8UC1)
   threads = CreateEmptyVector()
   ReserveSpace(threads, numberOfThreads)

   For colNum from imageCols - 1 down to 0:
        threads.emplace_back([this, &imageRows,
          &image, &retval, colNum]()
        For i from 0 to imageRows:
            retval.at<uchar>(i, colNum) =
            performSobelOnPatch(image, i,
            colNum,
            threshold)
             );
        If threads.size() >= numberOfThreads:
           For each thread in threads:
               JoinThread(thread)
           ClearVector(threads)

    For each remaining thread in threads:
        JoinThread(thread)

    Return retval
```

C. *Method C: Image Slicing + Parallel Vertical/Horizontal Gradient Calculation*

- Threads are stored in an array vector and created using a lambda function. This function holds all of the code that the thread will run.
- Before threads are created:
  - Divide the image's height by the number of threads. This will be the number of rows each thread calculates.
  - Create empty matrices for the vertical, horizontal, and magnitude gradients. Threads will modify the matrices directly.
- Create threads to calculate the gradients:
  - N threads for vertical gradient, N threads for horizontal gradient
  - Each thread takes the image, empty vertical gradient, the thread's number, and the number of rows each thread takes.
  - Threads calculate what their slice of the image is based on their thread number and the number of rows each thread has.
  - Threads modify the matrix for the vertical/horizontal gradient directly as they apply the gradient kernel to their slice of the image.
- Wait for all vertical and horizontal gradient threads to finish
- Using the same process as before, create N threads to calculate the magnitude gradient.
- Wait for all magnitude gradient threads to finish and return the final magnitude gradient.

The code segment does the following:

The program calculates how many rows each thread should get, then uses a for loop, creates each thread. Because threads are created with lambda functions, they can access all local variables (the "=" in the parameter list does this), so it uses the loop index and the calculated rows per thread to determine what slice of the image it is working on.

```
// Calculate number of rows each thread should get
int threadRows = image.rows // numThreads
for (int i = 0; i < numThreads; i++){
    //Create new thread
    threads.push( thread([=, &input, &output] ()
    {
        //Thread calculates what slice it is working on
        int startRow = i * threadRows;
        int endRow = startRow + threadRows;

        //For each pixel of the image in the thread's slice
        for (int j = startRow; j < endRow; ++j)
        {
            for (int k = 1; k < numCols - 1; ++k)
            {
                // Calculate the gradient
            }
        }
    }));
}
```

V. Results

Each method was tested with three different and unique sizes for the image at hand. These images were RGB in coloration and were used in testing as the following three sizes: 1000x668, 3840x2160 (4k resolution), and 7680x4320 (8k resolution). Averages for each method found after 25 test per unique thread count.

A. *Average Runtimes for Unchanged Sobel*

- For 1000x668 image:
  - 1 Thread: 0.0463749s
- For 3840x2160 (4k) image:
  - 1 Thread: 0.564935s
- For 7680x4320 (8k) image:
  - 1 Thread: 2.28854s

B. *Average Runtimes for Method A*

- For 1000x668 image:
  - 1 Thread: 0.0961856s
  - 2 Threads: 0.19299s
  - 4 Threads: 0.289222s
  - 8 Threads: 0.384102s
- For 3840x2160 (4k) image:
  - 1 Thread: 1.34301s
  - 2 Threads: 2.66808s
  - 4 Threads: 3.97771s
  - 8 Threads: 5.28786s
- For 7680x4320 (8k) image:
  - 1 Thread: 5.58662s
  - 2 Threads: 11.1712s
  - 4 Threads: 16.7217s
  - 8 Threads: 22.2828s

Across varying image resolutions, the processing times exhibit a notable increase with an increase in the number of threads. These findings underscore the diminishing efficiency of parallelization with larger image sizes.

C. *Average Runtimes for Method B*

- For 1000x668 image:
  - 1 Thread: 0.153251s
  - 2 Threads: 0.0817591s
  - 4 Threads: 0.0569541s
  - 8 Threads: 0.0563766s
- For 3840x2160 (4k) image:
  - 1 Thread: 1.38859s
  - 2 Threads: 0.74067s
  - 4 Threads: 0.408929s
  - 8 Threads: 0.309931s
- For 7680x4320 (8k) image:
  - 1 Thread: 5.15427s
  - 2 Threads: 2.57397s
  - 4 Threads: 1.37988s
  - 8 Threads: 0.959669s

The results show faster processing times with more threads across different image resolutions. These findings underscore the efficiency of parallelization, especially for high-resolution image processing.

D. *Average Runtimes for Method C*

- For 1000x668 image:
  - 1 Thread: 0.0516547s
  - 2 Threads: 0.0311133s
  - 4 Threads: 0.0263651s
  - 8 Threads: 0.026317s
- For 3840x2160 (4k) image:
  - 1 Thread: 0.562379s
  - 2 Threads: 0.455996s
  - 4 Threads: 0.381443s
  - 8 Threads: 0.373536s
- For 7680x4320 (8k) image:
  - 1 Thread: 2.45475s
  - 2 Threads: 1.77213s
  - 4 Threads: 1.47348s
  - 8 Threads: 1.42112s

While a significant improvement over the single-threaded implementation, the runtime scales about the same as the single-threaded implementation. On average, the runtime for 4 or more threads is about half of the single-threaded implementation.

E. *Comparisons*

Comparing the results of each method on varying thread counts and over 25 tests per each unique thread count, it is shown that the following are the winning methods per different category (decimals show percentage of either positive or negative effect on runtime compared to the unchanged Sobel):

Best Performance per Thread Count and Image Type			
	1000x668 image	3840x2160 (4k) image	7680x4320 (8k) image
1 Thread	Method C (-.11)	Method C (.005)	Method C(-.7)
2 Thread	Method C (.33)	Method C (.19)	Method C (.23)
4 Thread	Method C (.43)	Method C (.32)	Method B (.40)
8 Thread	Method C (.43)	Method B (.45)	Method B (.58)

TABLE I
COMPARISON TABLE FOR RESULTS

The methods B and C show that for each size of image that was tested, 1000x668, 3840x2160 (4k) and 7680x4320 (8k) respectively, an improvement on the base Sobel program is achieved. For the 1000x668, the fastest runtime was achieved by Method C using specifically eight threads. Method C achieved a runtime of 0.026317s compared to 0.0463749s on the unchanged Sobel. Testing with the image of size 3840x2160 (4k), Method B achieved the fastest runtime using eight threads. Said method performed at a runtime of 0.309931s compared to the unchanged Sobel implementation with a 0.564935s. For the last image size that was tested, 7680x4320 (8k), Method B provided the most effective runtime using eight threads, achieving a runtime of 0.959669s compared to the runtime of 2.28854s from the unchanged Sobel. Overall, it is shown throughout each image size that the fastest runtime is achieved with the use of eight threads, and the fastest runtimes are shared between methods B and C. Regardless of image size, it is shown that the fastest runtime for Sobel completion is achieved by Method B using 8 threads on an image of size 7680x4320 (8k).

VI. CONCLUSION

This paper shows that an improvement in runtime for the Sobel algorithm through parallelization is possible. Through the creation of three methods, different ways of multi-threading the Sobel Edge Detection algorithm were observed and tested with three different sizes of input images. The methods B and C showed improvement in runtime compared to the unchanged Sobel implementation, while method A showed no improvement and worsened the runtime when compared to the unchanged Sobel implementation. Out of the three different sizes of image input, Method B provides the best runtime for two sizes (3840x2160 (4k) and 7680x4320 (8k) respectively) while C has the best runtime for one size (1000x668). Whether one method is better than the other depends on the size of the selected input. With all this said, the best runtime was always achieved with eight threads, meaning that using at least eight threads will guarantee a noticeable improvement compared to the unchanged Sobel implementation, thus making it a more competitive option for edge detection than it was before. More investigation and research can be done using these exact same methods in search of an even larger runtime improvement but with a focus on an increase in the number of threads given better resources for memory and CPU.

REFERENCES

[1] I. Irwin, “History and Definition of the so-called ”Sobel Operator”, more appropriately named the Sobel-Feldman Operator,” *ResearchGate*.

[2] Hueckel, M.H., ”An Operator which Locates Edges in Digitized Pictures” in Journal of the Association of Computing Machinery, Vol.18,No. 1, January 1971, pp. 113-125.

[3] A. Hast., ”Simple filter design for first and second order derivatives by a double filtering approach”, Pattern Recognition Letters, Vol. 42, no.1 June, pp. 65–71. 2014

[4] “Sobel operator” *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Sobel$_operator.[Accessed : March\ 24, 2024]$.