

# Parallelized Canny Edge Detection

A Comparison of OpenMP and CUDA Paradigms for Parallelized Image Processing

Sarrankan Pathmanathan  
Institute for Aerospace Studies  
University of Toronto

**Abstract--** The automated extraction of geometric features, including edges, is a common problem in the field of Image Processing and Computer Vision. However, edge detection is computationally expensive as image size and complexity scales, and many of the algorithms that perform such detection may not leverage parallelization to optimize its semantics. This project investigates applying parallelization to Canny Edge Detection, one of several image processing techniques that exist within the scope of Computer Vision. There are several sequential and parallel implementations of it in varying languages. Therefore, there are several ways parallelism can be achieved. However, this report provides insights into the use of OpenMP and NVIDIA CUDA by a first-time programmer in both paradigms. This is done by comparing the sequential implementation of each of the different stages of Canny Edge Detection, in the C programming language, to the parallel implementation of these stages in OpenMP and CUDA to determine overall ease of use, speedup, and complexity. GPU and CPU Google Colab Cloud VMs are used to successfully create a functional model of Canny Edge Detection in OpenMP and CUDA. It was found that CUDA and OpenMP outperformed the sequential implementation, with OpenMP completing 2X faster and CUDA bordering at 2.5X, both able to produce an edge map with over 90% accuracy. However, CUDA was harder to grasp due to its vast array of explicit functions. Nonetheless, for a bounded problem like pixel manipulation in images, GPU programming prevails, as is the reason for today's dedicated graphic cards.

**Index Terms**—CUDA, OpenMP, Canny Edge Detection

## I. INTRODUCTION

Computer vision is a vast field whose principles can be applied in many applications ranging from determining cell boundaries in microbiology to scene reconstruction in autonomous space exploration. It encompasses several well-known techniques and algorithms that are leveraged and improved upon constantly, with several publications that detail research in the attempt to increase their efficiency. Scalability is a very common problem in computer vision as dealing with large inputs is a computationally expensive task. Preprocessing is a common technique applied to the input data to reduce the complexity and noise that are prevalent. One such technique is that is very well known in computer vision is Canny Edge Detection. Canny Edge Detection is a process used to segment pixels that form edges in an image from non-edge pixels. It is often used as a preprocessing technique for detecting features either from an image through 2D hand

engineered features, or through convolutional neural network that provide feature proposals.

If an image is fed to these models with, no form of preprocessing like Canny Edge Detection, the result often leads to several false positive detections due to unfiltered noise. In cases like these it is important to not only leverage preprocessing but ensure that such techniques also do not add overhead to the overall complexity of the system, as they are often relied upon in real-time applications.

There are several implementations in many programming languages and paradigms of Canny Edge Detection that are both sequential and parallel [1]. I have also used this technique in Python using the OpenCV library as a preprocessing step to detect the edge features in a scene. I was motivated to explore a low-level parallelized implementation of Canny Edge Detection using the C programming language to understand its semantics and determine if I could achieve considerable speedup from the sequential C implementation.

As part of this project, I also wanted to determine how well an average programmer could leverage some of the parallel programming paradigms discussed in ECE1747 for the first time. This work provides insights into such a challenge by comparing the sequential C implementation of Canny Edge Detection to a multithreaded CPU implementation using OpenMP and a parallelized GPU implementation using NVIDIA CUDA. By reading this report, an adept programmer should be able to reason as to the usability of OpenMP and CUDA in the field of Computer Vision and the practicality of using such techniques to optimize image preprocessing.

The rest of the paper is organized as follows. Section II introduces background on Canny Edge Detection including each stage of its pipelined approach. Section III details the practicality of OpenMP and CUDA for this algorithm as a first-time developer in each paradigm. Section IV details the implementation of OpenMP and CUDA on each of the stages that form Canny Edge Detection and compares the complexity of developing these solutions. Section V details the experimental methodology, Section VI the experimental results, Section VII compares this to related works, and Section VIII concludes the paper with final thoughts and considerations.

## II. BACKGROUND ON CANNY EDGE DETECTION

Canny Edge Detection (CED) is pipelined image preprocessor where a series of 5 stages are performed sequentially to not only obtain the most prominent edges in an image but obtain an overall edge map that minimizes noise and non-edge pixels. In order, these stages are gray scaling, to monochrome an input color image, gaussian filtering, to blur the image and remove significant noise, gradient calculation to determine edge intensities and direction, non-maximum suppression, to identify and suppress pixels whose intensities do not exceed the suppression threshold, and double threshold and hysteresis, to convert weaker pixels into stronger pixels and trace the final edge map. To implement each, especially in parallel, we must first understand their semantics.

### A. Grayscale

Each pixel (x,y) in a 2D image have a value associated to each channel in RGB (red, green, blue). To grayscale an image, we must adjust the contribution of each channel through the luminosity method [2].

$$newRGB(x, y) = 0.3R_{xy} + 0.58G_{xy} + 0.11B_{xy} \quad (1)$$

This method effectively monochromes the input image by weighting green and blue on opposite spectrums and keeping red in between. The luminosity method applies the weights in (1) however other weights can also be used. A sequential implementation requires a nested for loop containing two loops for the row-column based traversal, as well as a uniform operation to retrieve the RGB channels and compute the new set. The runtime complexity of such a loop is  $O(MN)$  with a worse case run-time of  $O(N^2)$ . It is clear this operation becomes expensive as the size of the image increases.

### B. Gaussian Filtering

Gaussian filtering is a mathematical technique that employs the use of kernel matrices to blur an image, effectively reducing the amount of noise that may affect edge detection. A kernel matrix is an  $N \times N$  matrix, whose size as well as the value of sigma used when computing the matrix, determines the amount of blurring that is applied [3]. To compute the Gaussian Kernel Matrix for an image with pixel coordinates (x, y) the following equation is used:

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

In order to blur the image through the kernel matrix K, a convolution of the matrix K on the image I is performed, resulting in a new image  $I * K$ . Convolutions are also a mathematical operation of the form:

$$g(x, y) = \omega * f(x, y) = \sum_{dx=-a}^a \sum_{dy=-b}^b \omega(dx, dy) f(x + dx, y + dy) \quad (3)$$

Similar to the analysis done in gray scaling, we can translate the convolution of matrix K on image I as a nested for loop of four levels. That is, the inner for loop performs the convolution operation of the kernel against the image pixels while the outer for loop performs traversal across the entire image.

Convolutions are a fundamental operation in pixel manipulation, when a filter denoted by a kernel matrix is to be applied to an image. Therefore, the larger the kernel is, the more expensive the operation becomes. With traversal in an  $N \times N$  image as well as an  $N \times N$  kernel matrix for each pixel, the worst-case run-time can be up to  $O(N^4)$ , making this stage the most expensive of all.

### C. Gradient Calculation

Gradient calculation detects the edge intensities and directions within the image by calculating the magnitude of the gradient of each pixel using kernel operators. Edges correspond to a change of pixels' intensity. To detect this change, these operators are applied in the x and y directions of the image to highlight pixel intensities. There are a variety of operators available for this task such the one discussed in [4]. However, since they are all  $3 \times 3$  matrices, any one of them will not impose a difference in our analysis. For our purposes, we will leverage the Sobel Kernel [3]  $G_x$  and  $G_y$  depicted in (4).

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad (4)$$

Similar to the gaussian filter operation, we must apply the convolution operation using both Sobel Kernels, on all pixels in the image. The magnitude and direction are then obtained using (5).

$$|G| = \sqrt{x^2 + y^2} \quad \theta = \arctan\left(\frac{y}{x}\right) \quad (5)$$

Since convolution is once again needed to be performed, it is evident that the major bottleneck to scalability is related to the convolution operation.

### D. Non-Maximum Suppression

Non-maximum suppression alongside double threshold and hysteresis do not require a convolution operation to take place. However, they are reliant on comparison operations between neighboring pixels. Non-maximum suppression compares each candidate pixel (x,y) to its 8 connected neighbors in the direction of the edge intensity defined by  $\theta$ . If there exists a pixel in the direction  $\theta$  that is more intense than the candidate (the value of  $|G|$  is greater), that pixel is kept while the candidate pixel's RGB channels are rendered black [5]. Otherwise, the candidate is kept using the same intensity  $|G|$ . Each pixel is traversed to determine candidate edges from non-candidates. Finally, a new image is produced with the non-maximum suppression candidates to obtain the output that is fed into the final stage of Canny Edge Detection. Figure 1 demonstrates traversing the gradient magnitudes and comparing the 8 connected neighbors in the direction provided by  $\theta$ . Since comparisons often lead to short-circuits (not all

pixels are needed to be compared), we are not as concerned with the implications this stage may have compared to convolutions.

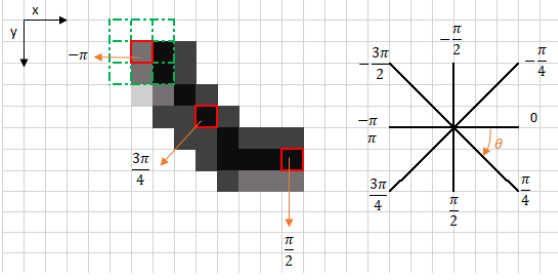


Figure 1: Comparing intensities of the 8-connected neighbours in the direction  $\theta$  [4]

### E. Double Threshold and Hysteresis

The double threshold step is used to classify pixels by their intensity values into either strong, weak, or non-relevant. A strong pixel is one whose intensity will be part of the final subset of edge pixels. A weak pixel has an intensity that is not strong, but also relevant. A non-relevant pixel is not kept as part of the final subset of edge pixels. As the name suggests, double threshold uses a lower and upper threshold to determine these classifications [6].

Once double threshold classifies all pixels in the non-maximum suppression map, hysteresis is used to convert the weak pixels into strong ones [7], as losing them in the final edge map would lead to inaccuracies. Hysteresis performs a similar 8-connected neighbor comparison. Given a candidate weak pixel  $(x, y)$ , and its 8-connected neighbors, a weak pixel is converted to a strong pixel if and only if one neighboring pixel is a strong pixel, as shown in Figure 2. Similar to non-max suppression, this stage is also subject to short-circuiting as not all comparison may be needed.

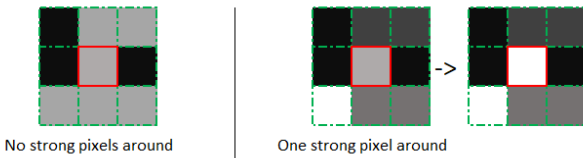


Figure 2: Converting weak pixels to strong ones for final edge map [4].

## III. OPENMP AND CUDA

One of the goals of this report and the course project is to not only achieve speed up, but also determine the practicality of paradigms that have been explored through papers that have reviewed in class. Two such papers explore the use of OpenMP [8] and NVIDIA CUDA GPU programming [9]. By leveraging each in creating a parallelized model of Canny Edge Detection, not only can we determine if they provide speedup, but also the complexity in designing a parallelized solution as a first-time programmer in each. Understanding

the architectural semantics of their provided programming directives produces a large learning gap. We must be able to interpret each directive translate them to expected multithreading behavior. This is imperative for image processing as threads are often performing work on a single shared image. For this project, we start by understanding OpenMP and CUDA from a programmer's perspective to determine a baseline implementation in each that functions. We then optimize each model to remove fine-grain parallelism errors in the implementation phase.

### A. OpenMP

OpenMP is a multithreading paradigm that provides program directives and libraries for run-time support of parallelism in C. Programs can be annotated with such directives using `#pragma` statements to enable parallel execution by classifying areas of the program as parallel regions in a fork-join model [8]. A master thread executes sequential code while parallel sections are executed by N threads, including the master. By determining the proper use of data environment directives for sharing of program variables, reduction directives against an identified reduction variable, and standard synchronization directives such as barrier and critical, the aim is to develop an OpenMP Model that can parallelize Canny Edge Detection without significant reduction to edge detection accuracy.

The `#pragma omp parallel for` is a directive that is relied upon heavily in the OpenMP implementation as most operations require traversal of the image. Other directives can be leveraged include `#pragma omp critical` to lock critical sections where data races may occur when manipulating pixels, and `#pragma omp reduction(+:pixel)` when a summation such as (3) is performed on the shared pixel variable. These directives including others are detailed further in Section IV as they are utilized for implementation.

### B. Compute Unified Device Architecture (CUDA)

NVIDIA CUDA is a GPU programming paradigm that aims at utilizing the large number of threads provided by the GPU device to attain high speed up. CUDA works by separating operations between the CPU and the GPU, commonly referred to as the host and device, respectively [9]. Computationally expensive operations that benefit from parallelism are delegated to the GPU and wrapped as kernel functions which can provide high level of multithreading. These functions are then called from the CPU.

Kernel operations utilize many threads collectively called a grid. Each grid is made from thread blocks where each thread block can have a maximum of 512 or 1024 threads [9]. Thread block size is dependent on the NVIDIA GPU architecture we leverage. Kernel operations must then utilize these threads to parallelize operations by assigning a thread ID from a respective block to a determined portion of the overall operation. During this time, the programmer will determine the point where the CPU should be halted as subsequent program instructions rely results provided by the GPU. Since the memory space between the CPU and GPU are not shared, kernel functions cannot return variables in the traditional

sense. The results must be transferred from the GPU to the CPU using CUDA directives, which also allocate memory on the GPU memory space.

A CUDA programmer must determine how operations should be delegated to the GPU and which operations within an algorithm benefit from being run on the device. They must also determine the number of blocks of threads to use, as well as be aware of their behavior from the moment the kernel function is launched.

For the GPU to access the pixel data of the image loaded first on the CPU memory space, the array must be copied to GPU memory. To do so, memory of the same size of what is to be copied to the GPU space must be allocated on the device. This is done using **CudaMalloc**, like **malloc**. CUDA does not have an explicit **Calloc** implementation. As **Calloc** provides zero-initialized buffers, **CudaMalloc** and **CudaMemSet** are used to allocate memory and zero the buffer, respectively. **CudaMemcpy** is then used to copy the image pixel data to the device using the **HostToDevice** Flag and back using **DeviceToHost**. The CPU calls the kernel function and passes the GPU allocated copy of the pixel data, functional arguments not dynamically allocated and the number of blocks and threads that will perform the required operations. The CPU is halted using **cudaDeviceSynchronize** as it waits for the GPU to return with results. It is good practice as a CUDA programmer to wrap CUDA commands that are executed from the CPU with error checking. Like other parallel programming paradigms there is no guarantee that the memory we have dynamically allocated for use with our program is allocated correctly on each execution. This notion is even more prevalent when we have two memory spaces where allocation occurs. CUDA provides error checking using **cudaPeekLastError** as well as **cudaGetErrorString** informing the programmer of errors.

On return from the kernel function, we copy the results of the memory in the GPU space to a variable in the CPU space using the **CUDAMemCopy** directive which now allows the CPU to act on the results within its memory space. Finally, we must free the dynamically allocated memory from both the CPU and GPU memory spaces using **Free** and **CudaFree** respectively.

Figure 3 demonstrates this common instructional sequence of CUDA functions which are often wrapped around the kernel call.

```
//Allocate Memory on Host
unsigned char *image_data = (unsigned char*)malloc(size * sizeof(unsigned char));
unsigned char *image_data_2 = (unsigned char*)malloc(size * sizeof(unsigned char));
//Read all image data
fread(image_data, size, 1, fIn);
unsigned char *image_data_CUDA;
//Allocate memory on Device and copy image data to it
cudaMalloc((unsigned char**)&image_data_CUDA, size * sizeof(unsigned char));
cudaMemcpy(image_data_CUDA, image_data, size * sizeof(unsigned char), cudaMemcpyHostToDevice);
// Define Grid and Block
dim3 DimGrid((width)/32, (height)/32);
dim3 DimBlock(32, 32);
//Call grayscale kernel function
grayscaleCUDA<<<DimGrid, DimBlock>>>(image_data_CUDA, size);
//Check for any errors on kernel launch
gpuErrchk(cudaPeekAtLastError());
//Halt CPU until GPU kernel function completes
gpuErrchk(cudaDeviceSynchronize());
//Copy computed grayscale back to host
cudaMemcpy(image_data_2, image_data_CUDA, size * sizeof(unsigned char), cudaMemcpyDeviceToHost);
//Write all data to output file with offset
fwrite(image_data_2, size * 1078, 1, fOut);
```

Figure 3: A commonly used structure for CUDA kernel launches

### C. Parallelism in Image Processing

Image processing requires two major operations:

- 1) Traversal of the image using (x,y) pixel coordinates for NxM images.
- 2) Computations that either alter the RGB channels of a pixel or compares the pixel to another pixel.

Canny Edge Detection is a bounded problem. Since we are aware of the image data size, width and height, and this techniques does not alter the bounds of the image, as long as memory is allocated correctly to house the image data, and data races are mitigated on this shared variable, we can guarantee some form of speed up will be achieved. Often, computations that need to be done in image processing are uniform and mutually exclusive. For example, gray scaling the pixel (0,0) and (0,1) is a uniform operation on each pixel. There is no data dependency between them, therefore we can preserve atomicity of the operation.

An optimal use of the GPU and device functions as well as OpenMP threads are operations which can utilize a correct number of threads to minimize resource waste while achieving the best speedup possible and function correctly. Manipulating the pixels of an NxM image aligns very well with these criteria. Because the search space is constant, most operations guarantee that all pixels must be traversed. If a uniform operation is to be applied to every pixel, then a function that assigns a thread to perform such an operation on every pixel of the image such that all operations are performed in parallel, are atomic, consistent, isolated and durable, would be classified as an optimal use of multithreading. During the implementation phase, it is imperative that we can determine which stages of Canny Edge Detection perform uniform operations, and which do not. Since traversal and computation in each stage requires the use of for loops, and may also require nested loops, the goal is to reduce these common bottlenecks to speed through the OpenMP and CUDA models.

## IV. IMPLEMENTATION

The sequential C implementation of Canny Edge Detection was taken from Rosetta Code [1], as I noticed other projects, which are explored in Section VII, had uses this implementation as a starting point. It also uses BMP as a workload (discussed in section V) and provides a good initial separation of all the stages of Canny Edge Detection. In general, for loops are a requirement for traversal of the image data as well as operations which will require nested loops. These instructions incur the most overhead in a sequential program. Any operations within Canny Edge Detection that require the use of a for loop is automatically determined to be a candidate for parallelism. However augmenting a for loop through OpenMP directives is much easier than how CUDA handles traversal operations. The following sections dive into the details of the implementation of each stage of Canny Edge Detection in the sequential, OpenMP and CUDA models. For brevity, we will

analyze the complete implementation, including code, of gray scaling in each paradigm to understand commonly used syntax and considerations. For the remaining techniques we will compare what was implemented in gray scaling to the operations required of the technique to determine what must be varied or considered for parallelism.

## A. Gray scaling

### 1) Sequential

The sequential implementation from Rosetta Code assumes that the input image is already gray scaled. Therefore, the load and save bmp methods does not support 24-bit color depth input images. To experiment with the entirety of algorithm, I included an implementation of gray scaling as described in Section 2.1. The code is illustrated in Figure 4.

```
//Grayscale each Pixel
unsigned char pixel[3];

for (int y = 0; y < height; ++y)
{
    for (int x = 0; x < width; ++x)
    {
        fread(pixel, 3, 1, fin);
        unsigned char gray = pixel[0] * 0.3 + pixel[1] * 0.58 + pixel[2] * 0.11;
        memset(pixel, gray, sizeof(pixel));
        fwrite(&pixel, 3, 1, fout);
    }
    fread(pixel, padding, 1, fin);
    fwrite(pixel, padding, 1, fout);
}
```

Figure 4: Sequential Gray Scaling

A nested for loop is used to travel the (x,y) pixel coordinates of the image. For each pixel coordinate we read its RGB channels into the variable pixel and apply the luminosity equation (1) to the channels which is then written to an output BMP.

### 2) Grayscale OpenMP

Applying OpenMP to the sequential model directly is a non-optimal solution as the File I/O operations are being done on a single BMP. Therefore, as the number of threads increase, so does the contention for the file, which in turn results in fine-grained errors such as the misalignment of pixels from the original image. It is often not recommended to parallelize I/O when the underlying operating system does not support it. To mitigate this we would need to serialize the reading of the file to a single instruction such that all the image data is read, parallelize the gray scaling process for each thread to coherently read a pixel to perform gray scaling on, and then serialize the writing of the gray scaled image data to file. In essence, we are restricting parallelism to only the gray scaling computation as shown in Figure 5.

```
#pragma omp parallel for
for(int i = 0; i < size; i= i+3)
{
    unsigned char gray = image_data[i] * 0.3 + image_data[i + 1] * 0.58 + image_data[i + 2] * 0.11;
    #pragma omp critical
    memset(pixel, gray, sizeof(pixel));
    image_data[i] = pixel[0];
    image_data[i+1] = pixel[1];
    image_data[i+2] = pixel[2];
}
```

Figure 5: OpenMP Gray Scaling

As an OpenMP programmer we need to determine how threads will behave in such a loop and what precautions require which OpenMP directives. Usually there are more iterations in a loop than there are threads. Since gray scaling is a uniform operation and we are aware of the time it takes to complete, we do not need to load balance threads through a **schedule(dynamic)** clause. This notion is shared for other uniform image filtering operations.

### 3) Grayscale CUDA

Implementing gray scaling, as well as every other technique in the Canny Detection pipeline, in CUDA requires a much different mindset. Although both paradigms utilize threads, the method with which parallelism is achieved is different. As mentioned in Section III B, CUDA requires the programmer to consider porting computations normally run on the CPU to the GPU instead. As illustrated in Figure 3 and compared to Figure 5, most of the underlying code is with regards to transferring all relevant computational object to the GPU. Once the kernel is launched, for loops must be replaced with a serialized approach whereby the time the kernel function ends, the picture is gray scaled. Threads in Figure 6 are assigned, by thread ID, to a pixel in the image and each of the RGB channels are manipulated using the luminosity method, resulting in a gray scaled image Figure 7 (a).

```
global__ void grayscaleCUDA(unsigned char *image_data, int size){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    image_data[idx] = ((0.3 * image_data[3*idx]) + (0.58 * image_data[3*idx+1]) + (0.11 * image_data[3*idx+2]));
}
```

Figure 6: Grayscale with CUDA

## B. Gaussian Blur

### 1) Sequential Gaussian Blur

This stage requires calculating the kernel matrix and convolving it against the image. I used a 7x7 matrix as it and the 5x5 are commonly used. However, using such a large matrix in the convolution operation makes this operation the most expensive of the pipeline when convolution is applied. A nested for loop is used to compute each index of the 7x7 matrix. A doubly nested for loop is then required to implement equation (3) which performs the convolution. This operation is detailed in the Gradient Calculation analysis. The result of Gaussian Blur is depicted in Figure 7 (b).

## 2) OpenMP Gaussian Blur

For the OpenMP implementation, the nested for loop used to build the kernel matrix is first panelized. This is done by using the **#pragma omp parallel for** directive. Since the equation can be translated into a perfectly nested loop, the **collapse(2)** directive is used to increase the total number of iterations that will be partitioned across the available number of threads by reducing the granularity of the work done by each thread [10]. Finally, the index which is incremented to compute the next kernel operator in the matrix is appended with the **#pragma omp atomic** directive as it is a shared variable. Although the **#pragma omp critical** would provide the same results, atomic allows the compiler more opportunity for optimization by using hardware level instruction while also allowing different elements of the array to be updated concurrently.

## 3) CUDA Gaussian Blur

The CUDA implementation is similar to that of the gray scale implementation. We allocate a 2-dimension block of threads using the **dim3** directive such that a thread is assigned to an index in the matrix. We then serialize the access to the matrix by computing the access index using the thread IDs. This significantly improve the time taken to retrieve the 7x7 Kernel Matrix as all threads work towards computing an index of the matrix. The **\_\_syncthreads** directive is used to block threads until all indices have been computed before the convolution method is called.

## C. Gradient Calculation

### 1) Sequential Convolution

Since gradient calculation leverages preset Sobel kernels, this section is used to analyze the convolution operation used it and Gaussian Blur. The calculation of the magnitude intensity and direction is discussed in D. Non-Maximum Suppression. Translating equation (3) to code requires the use of a doubly nested for loop. As the size of the convolving matrix increases, so that does the run-time complexity of this operation. The outer loop requires traversal of the image array while the inner loop performs the summation operation illustrated in (3) using the image data and the kernel matrix. The result of gradient calculation is depicted in Figure 7 (c).

### 2) OpenMP Convolution

As expected, the double nested for loop is the primary candidate for parallelism. Using equation (3), we can infer that several variables are in play to perform this computation. Therefore, those that are shared amongst threads must be handled appropriately. To do so, we append the familiar **#pragma omp parallel for** with a **private** clause such that the variable is replicated, and a local copy provides to each thread. A reduction operation is then used on the summation portion of the operation which improves the performance by performing the summation as a sum of partial sums. However, the index to retrieve the kernel cannot since an explicit critical clause must be used. Since the equation (3) also forms a

perfectly nested loop, we can use the collapse directive similar to computing the gaussian kernel to serialize the for loop for the same benefits.

### 3) CUDA Convolution

Converting a doubly nested for loop to follow CUDA semantics is not as intuitive as using directive in OpenMP. Not only must we serialize access to the image data for convolution, but we must also serialize access to the kernel as well. Planning is required to sync access between the kernel and a pixel in the image, such that convolution can be performed properly, as out-of-bounds access due to properly managing the high number of threads is often a recurring problem in CUDA. However, when done correctly, this provides great benefits in terms of speed. Following Figure 6 we include an addition index for the kernel matrix to perform the convolution as in (3).

## D. Non-Maximum Suppression

Non-Maximum Suppression alongside Double Threshold and Hysteresis incur the least amount of run-time amongst all the stages of Canny Edge Detection. This is because both perform comparisons and do not require that all pixels be compared which short-circuits the operations. Since all pixels are being traversed, and the resultant magnitude intensity and directions are being analyzed here, we can balance the pipeline by performing at least the direction computation within this step. This mitigates having to add an additional nested for loop just to compute direction. Each neighboring candidate pixel is computed uniformly followed by direction as well as the comparison operations to suppress non-maximums. The result of Non-Maximum Suppression is depicted in Figure 7 (d).

To apply openMP to this stage, we can simply add collapse to the perfectly nested for loop which is used to traverse the image. To apply CUDA to this stage, we follow the example in Figure 6 to compute a serialized index. Both reap the same benefits as discussed in other stages.

## E. Double Threshold and Hysteresis

Tracing of the final edge map is performed using the hysteresis step, while double threshold aims at determining weak pixels to which the hysteresis converts to strong. Image traversal is required which translates to a nested for loop. Threshold comparison can be accomplished in a single instruction however edge tracing requires computing the neighboring pixels and edges. Therefore, in the OpenMP implementation the **#pragma omp atomic** operation is used frequently to increment edge calculations and indices when the final edge map is built. As this is also a perfectly nested loop, a collapse can be leveraged like previous steps. The CUDA implementation follows the same principles applied in Non-maximum suppression. Figure 7 (e) provides the final edge map computed through Canny Edge Detection.



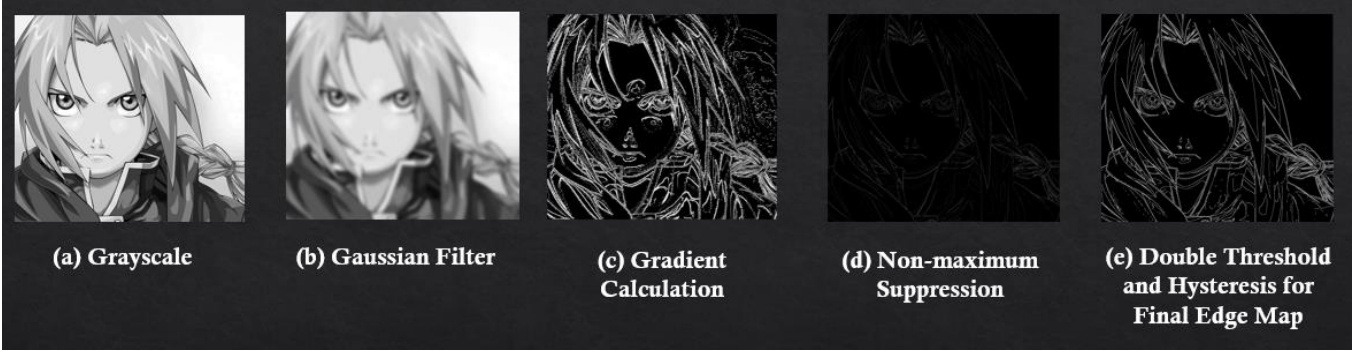


Figure 7: The Complete Canny Edge Detection Pipeline with Resultant Edge Map

## V. EXPERIMENTAL METHODOLOGY

### A. Bitmap Workload

The workload that will be used in this project are images in the bitmap format (BMP) as they are the most intuitive for pixel level manipulation in C and can be easily translated to a C based data structure. Bitmaps are arranged in a sequence of bytes where a WORD of 2 bytes (16 bits) and a DOUBLE WORD of 4 bytes (32 bits) store information regarding the bitmap image including image size, width, and height. For a bitmap image that uses the red (R), green (G), and blue (B) colour channel, this sequence is classified into four blocks:

- **BITMAPFILEHEADER (14 bytes):** File specifications of the BMP including FileType and FileSize.
- **BITMAPINFOHEADER (40 bytes):** Information regarding the image data stored in the bitmap including ImageSize, ImageWidth, and ImageHeight.
- **COLORTABLE (4 Bytes):** Stores the RGB channels used by each pixel within the image
- **PIXELDATA (variable):** The image data described by the BITMAPINFOHEADER.

If one needed to retrieve the width of the image, they would need to read into the BITMAPINFOHEADER at the byte location where this information is stored. C structs are the best way to translate the BMP structure and retrieve the necessary information for computations.

Since Canny Edge Detection converts inputs into grayscale, all input images will initially have a 24-bit color depth, where 8 bits are allocated for each channel in RGB, but will then have 8-bit depth as we only require a single channel ranging from 0-255 for gray. The final edge map will also be of 8-bit color depth as defined edges will be colored in 2 white and non-edges in Black.

Since it is assumed that BMPs are the sole input type for the project, to load an image we only require determining the image size N, to malloc and array of size N of PIXEL DATA, as well as the height and width of the image to traverse across each pixel.

Once Canny Edge Detection is complete and it is time to

reconstruct a new bitmap file with the edge results, we can use the same BMP template and our C structs to redefine each element in the sequence of bytes and form a new BMP containing the edge map.

### B. Jupyter Notebooks and Google Colab Environment

NVIDIA CUDA is reliant on GPU hardware, where device functions will perform its computations. Therefore, an environment with a GPU is required for this project. Since my personal computer did not have a dedicated GPU, I utilized Google Colab, which offers a GPU and CPU runtime, to analyze the sequential, OpenMP and CUDA models of Canny Edge Detection.

Google Colab hosts cloud-based virtual machines and provides Jupyter notebooks. These notebooks are comparable to a shared Google Docs where code can be written collaboratively with peers and run using the Google Colab run-times. On creation of a notebook, a virtual machine is allocated to the user for the duration of experimentation. If Google Colab detects no activity for a duration of time, it will require reconnection to the instance, at which time the user is allocated a new instance with prior work removed. Figure 8 provides details into the hardware specification for the CPU and GPU run-times.

Parameter	Google Colab (CPU VM)	Parameter	Google Colab (GPU Enabled VM)
CPU Model Name	Intel Xeon	GPU	Nvidia k80/T4
CPU Frequency	2.30 Ghz	GPU Memory	12GB/16GB
CPU Cores	2 core	GPU Memory Clock	0.82GHz/1.59GHz
CPU Family	Haswell	GPU Cores	2496 CUDA cores
Available Ram	12GB	Performance	4.1TFLOPS/8.1TFLOPS
Disk Space	25GB	Available Ram	12GB
		Disk Space	358 GB



Figure 8: Google Colab GPU and CPU specs

To simulate running my experiments on an on-premises machine, with similar specifications a few constraints were applied:

- 1) A dedicated file share was created on Google Drive to preserve code artifacts in cases where reconnection was required to a new instance. The Jupyter notebook was provided access to this share.
- 2) Experimental results were collected between the timeframes of Canny Edge Detection START and Canny Edge Detection END. These timeframes are equivalent to the start of the pipeline where gray scaling is performed, to the end where Hysteresis is complete, and an edge map is produced. This is to remove the network latency faced when loading the input bitmap into the program and saving the edge map onto the disk. Although parallelism could be added to these steps, they are not part of the Canny Edge Detection algorithm.
- 3) Since network latency can also affect operations like computing execution time, this is also considered.

### C. Experiments

The bottlenecks in Canny Edge detection are scalability in the size of the image and the number of defined edges that should be part of the final edge map. Therefore, it is crucial to determine how well each of the sequential, OpenMP and CUDA models perform when these two factors are varied. Table 1 provides the BMP workloads that will be used for experimentation.

Table 1: Experiments and Evaluated Metrics		Characteristics	
BMP Images	Image Size	BMP Image	
512 x 512 Anime	230 KB		
2048 x 2048 Anime	4MB		
4096 x 4096 Anime	16MB		
512 x 512 lanes	257 KB		
2048 x 2048 lanes	4 MB		
4096 x 4096 lanes	16MB		

Anime is chosen as cartoons often have well defined edges which can be easily detected by Canny Edge Detection. It is used as a validator to ensure that we receive as many known edges as possible. Ground truth edges are taken using the sequential implementation of Canny Edge Detection.

Lanes depicts the application of Canny Edge Detection to a real-life scene. As such, computing the edge map for this image is more complex. Fine-grained parallelism errors can be detected very quickly by viewing the result of this image. The 512 image is used as a baseline to determine if the implementations work. Results are reported on the 2048 and 4096 versions of Anime and Lanes as the sequential

implementation requires several seconds to compute the edge map for each.

Each of the sequential, OpenMP and CUDA models of Canny Edge Detection will be run on the entirety of the images in Table 1. There also exists a parallel implementation where we could have split the image into parts and performed Canny Edge Detection from each parallel model on each part before performing a restitch operation. However, such an experimentation does not test traditional Canny Edge Detection which is normally performed on the entire image. This also reduces the accuracy of the steps within the pipeline that depend on the 8-connected neighbor analysis. Therefore, we deviate from the project proposal which aimed at applying parallelism in this manner, and to ensure that edge correctness is not compromised.

To compare the OpenMP, CUDA and sequential implementation, we want to granularize our analysis to each stage in the Canny Edge Detection pipeline instead of the entire algorithm. We can first determine, in the sequential model, which of the stages imposes the biggest bottlenecks and determine if the parallel models at the same stage provide any performance improvements. As such, all collected metrics will be taken from each stage in each paradigm for comparison and analysis.

## VI. EXPERIMENTAL RESULTS

### A. Comparison of Pipeline Stage Execution

To determine program execution time, we run each model on the Google Colab platform for each of the images in Table 1. The average execution time of each stage as a percentage of total execution of the program for each image size is reported in Figure 9 for the sequential model.

From this figure we see that Gaussian Blurring incurs the highest cost in terms of execution time followed by gray scaling averaging over 50% of total computation combined. This trend is prevalent as well in the OpenMP and CUDA models as we cannot change the fundamental computations that are needed to be performed. However, we can speed them up. Figure 10 compares the average execution time of each paradigm on the BMP workloads. Compared to the sequential implementation the 32-Thread OpenMP model performs significantly better completing in 2X faster than the



sequential. The CUDA model performs slightly better than

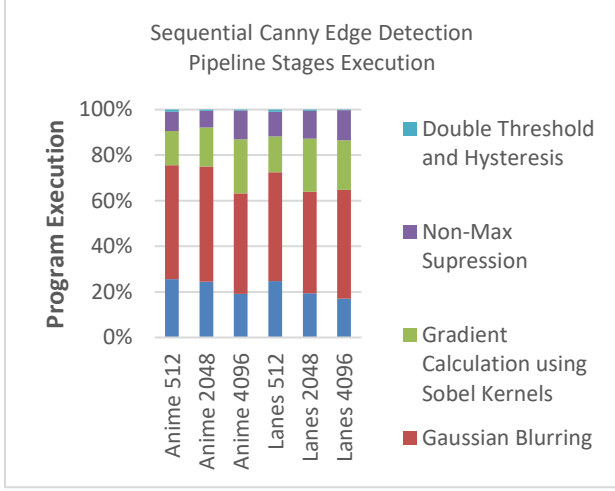


Figure 9: Execution time of each Canny Detection Stage as a percent of total execution

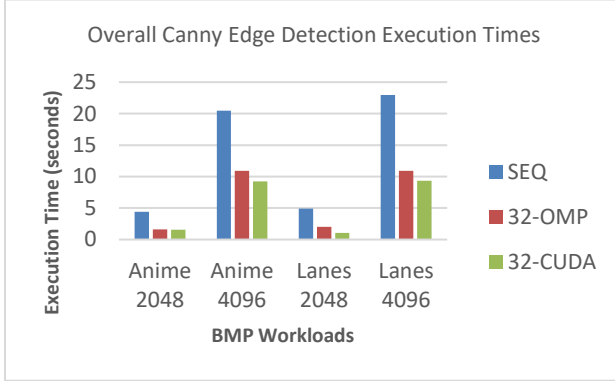


Figure 10: Overall Execution Time of Canny Edge Detection in each Model

OpenMP, bordering on over 2.5X the speed of sequential, however since a cloud-based GPU runtime is used due to not having access to physical hardware, I believe CUDA can achieve much higher speedup.

Figure 11 and 12 illustrates varying the number of threads for the parallelized implementations on each of Anime and Lanes. We can see that at 32 threads, there is not a significant jump from the 8-thread model. I believe the existing implementations could be optimized to allow better handling of threads in this case and that experimentation on physical hardware would also contribute to higher speedups.

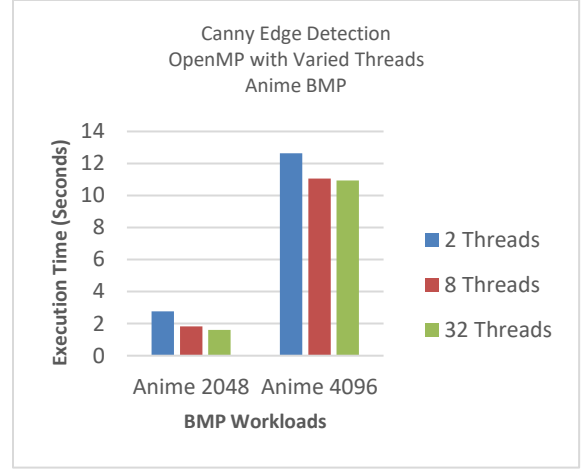


Figure 11: Parallel Canny Edge on Anime with Varied Threads

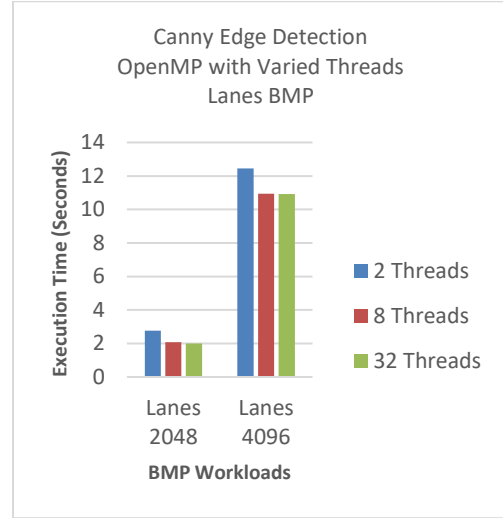


Figure 12: Parallel Canny Edge on Lanes with Varied Threads

### B. Detection Accuracy

In terms of execution time, Canny Edge Detection does benefit well from parallelization. However, we must also ensure that introducing parallelism does not alter the accuracy of the edge maps that are produced, especially when the number of threads are altered. Figure 13 provides insight into the accuracy of the edge map against the ground truth computed from the sequential implementation. We compare the RGB channel of each pixel in the ground truth edge map to the RGB channel of the edge maps produced using OpenMP and CUDA. Although we did not retrieve 100% accuracy, meaning there were fine-grain parallelism errors as a result of data races, we were able to attain over 90% accuracy in all cases and perform better in the complex Lanes image.

Since the parallel models requires all threads to share the same BMP, data races are inevitable. Handling these races requires extensive research into the semantics of both models. An ablation study would have been beneficial to determine how every directive in the CUDA and OpenMP stack affects the overall Canny Edge Detection Algorithm. However due to the multitude of different directives and the constraint in

bandwidth to familiarize myself with them all, I had studied only those I knew would provide large speedup while minimizing as much of impact on accuracy as possible.

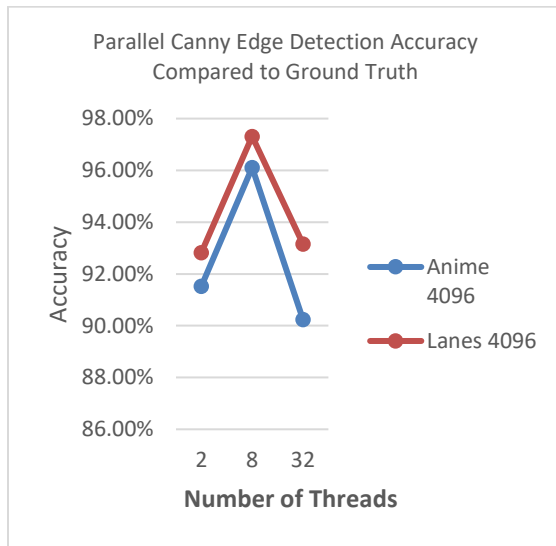


Figure 13: Accuracy of Parallel Canny Edge for Varied Threads

### C. Ease of Utilization

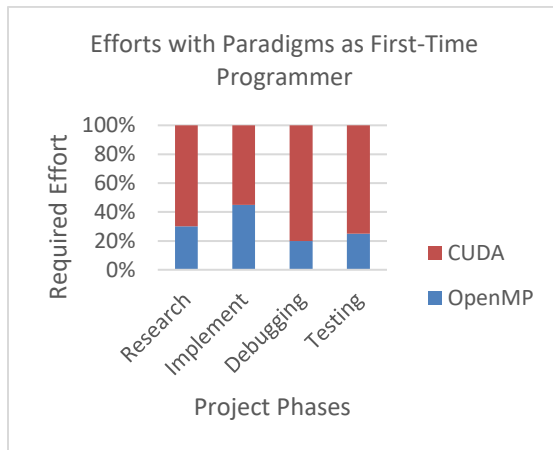


Figure 14: Required project effort in each phase of OpenMP and CUDA

As part of the evaluation in applying OpenMP and CUDA to sequential Canny Edge detection, we also want to determine the ease with which a first-time programmer in each paradigm could grasp and apply their respective semantics. OpenMP follows similar notions explored in other lock-based paradigms. Therefore, familiarizing myself with it did not require as much as it did with CUDA. Once grayscale was implemented in each, it became simpler to implement the remaining stages. However, because CUDA deals with a CPU and GPU memory space and requires different semantics for each, it requires much more effort to grasp than OpenMP as around 80% (Figure 14) of my efforts for debugging and testing went to CUDA. CUDA's library of functions is also much larger than OpenMP. OpenMP is meant to augment existing CPU calls using `#pragma` directives, whereas CUDA

requires explicit functions to be used for memory allocation, memory transfer and several other operations. However, CUDA can be a very powerful tool, if used well, as it provides much higher thread counts through its GPU.

## VII. RELATED WORKS

Canny Edge Detection is a popular edge map generator as an implementation of it exists in several languages and libraries. As it is often leveraged in many real-world applications as a preprocessor of images, there are several publications available that explore parallelizing this algorithm.

Mogale [11] provides insight in the use of multi core processors for an 8-core high-performance implementation of Canny Edge Detection. Their implementation utilizes Intel Cilk Plus for load balancing and resource communication. If it were not for time spent learning CUDA, I would have liked to add profiling of threads in OpenMP to determine thread utilization similar to Mogale's efforts. This would have helped to increase the number of the threads significantly. Canny Edge Detection can often be clustered with other preprocessing techniques to not only improve accuracy but enable better opportunities for speedup. Chen's [12] implementation aligns with this notion as they utilize the OTSU operator, to improve edge detection while leveraging a MapReduce parallel programming model attaining a 67.2% on a large dataset of images.

Chawla et al. [13] is one of many other implementations of Canny Edge Detection using CUDA as GPU programming and image/video manipulation often go together. They achieved true GPU speedup on a dedicated NVIDIA GeForce GPU with speeds of up to 262 times faster than the sequential. Since my project did not look into optimizing the existing sequential implementation but rather augment it through parallelism, there are several windows opportunity to re-explore this project and continuously improve upon the results.

## VIII. CONCLUSION

Overall, CUDA and OpenMP are suitable paradigms to apply to Canny Edge Detection, with CUDA performing around 2.5X better than the sequential implementation. It is clear for these reasons why dedicated graphics cards are a necessity in image and video manipulation for real-time applications. Multithreading on a single input image makes data races inevitable however by applying constructs that were learned in each of CUDA and OpenMP, we were able to retrieve an edge map that was over 90% accurate on average. If you are a first-time programmer who is constrained for time and wants to leverage parallelism that can be applied intuitively through directives, then OpenMP is a suitable candidate. However, if your application requires a large amount of threads, and a dedicate GPU is available, then CUDA can provide the speedup required. However, significant research is required to understand all the explicit function calls spread across host and device memory spaces. Future work would go into

understanding more of these constructs and performing an ablation study to determine how exactly each directive in OpenMP and CUDA affect execution time, accuracy, and thread scheduling.

## IX. REFERENCES

- [1] "Canny edge detector," Canny edge detector - Rosetta Code. [Online]. Available: [https://rosettacode.org/wiki/Canny\\_edge\\_detector](https://rosettacode.org/wiki/Canny_edge_detector).
- [2] Kanan C, Cottrell GW (2012) Color-to-Grayscale: Does the Method Matter in Image Recognition?. PLOS ONE 7(1): e29740
- [3] Gedraite, Estevao & Hadad, M.. (2011). Investigation on the effect of a Gaussian Blur in image filtering and segmentation. 393-396.
- [4] N. Tsankashvili, "Comparing Edge Detection Methods," Medium, 22-Sep-2020.
- [5] Rothe, Rasmus & Guillaumin, Matthieu & Van Gool, Luc. (2015). Non-Maximum Suppression for Object Detection by Passing Messages between Windows. LNCS. 9003. 10.1007/978-3-319-16865-4\_19.
- [6] S. Sahir, "Canny Edge Detection Step by Step in Python-Computer Vision," Medium, 27-Jan-2019. [Online]. Available: <https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123>.
- [7] Muhler, M.. (2020). hysteresis. 10.1002/9783527809080.catanz08699.
- [8] OpenMP for Networks of SMPs , Y.C. Hu, H. Lu, A.L. Cox, and W. Zwaenepoel, Journal of Parallel and Distributed Computing, vol. 60 (12), pp. 1512-1530, December 2000
- [9] From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming, Du Peng, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra, Parallel Computing 38, no. 8 (2012): 391-407.
- [10] "OpenMP Loop Collapse Directive," Intel. [Online]. Available: <https://software.intel.com/content/www/us/en/develop/articles/openmp-loop-collapse-directive.html>
- [11] High Performance Canny Edge Detector using Parallel Patterns for Scalability on Modern Multicore Processors. Hope Mogale, North-West University, NW, RSA
- [12] Cao, Y. (2018). Implementing a Parallel Image Edge Detection Algorithm Based on the Otsu-Canny Operator on the Hadoop Platform Computational Intelligence and Neuroscience, 2018, 3598284.
- [13] Jain, Adhir & Namdev, Anand & Chawla, Meenu. (2016). Parallel edge detection by SOBEL algorithm using CUDA C. 1-6. 10.1109/SCEECS.2016.7509360.