Parallel Image Convolution for Edge Detection

Anfal Bamardouf[1], Raneem Alqahtani [2], Razan Alkharasani[3], Haya Aldossary[4], Norah Aldossary [5], Sajedah Alqudaihi[6], Fatimah Alawami[7], and Rabab Alkhalifa[8]

[1]College of Computer Science and Information Technology, Imam Abdulrahman Bin Faisal University, Computer Engineering, Dammam, Saudi Arabia

December 26, 2025

Abstract

Purpose - This study aims to optimize the image convolution algorithm used in the Canny edge detection method by leveraging parallel computing techniques. The research provides new insights into the field of image processing, where computational speed and efficiency are critical.

Design/Methodology/Approach - The study employs C++ with OpenMP on an Ubuntu environment to optimize the Canny edge detection algorithm. The optimization targets key stages of the algorithm, including Gaussian blur, Sobel filtering, non-maximum suppression, double thresholding, and hysteresis. Both sequential and parallel implementations are developed to process the image convolution tasks and enable performance comparison. The dataset used in this study was obtained from a publicly available GitHub repository.

Findings - After resolving race conditions and applying synchronization techniques such as atomic operations, critical sections, and reduction, the results indicate that parallel computing improves processing speedup and efficiency compared to the sequential implementation.

Originality/Value - This project contributes to the image processing field by exploring efficiency improvements through a C++-based parallel implementation. It provides practical insights into how parallel computing can enhance scalability and performance in image processing pipelines.

Keywords: Image Processing 1; Canny Edge Detection 2; Parallel Computing 3; OpenMp 4; Benchmarking 5.

1 Introduction

Image processing is a fundamental area in computer science, and edge detection is one of the most essential techniques in image processing that focuses on identifying and outlining the boundaries of objects within an image. The goal of edge detection algorithms is to identify the most significant edges within an image or scene. Then these detected edges should have connected them to form meaningful boundaries. The segmented results can then be utilized in machine vision applications such as object counting, measurement, feature extraction, and classification [1].

Edge detection is typically performed using convolution operations with filters such as Canny operators. The Canny edge detection method, introduced by John F. Canny in 1986, is a multistage algorithm designed to identify a broad range of edges in images and is known for its optimal edge detection capabilities. The algorithm starts operation through a sequence of steps aimed at suppressing noise, detecting edges, and enhancing overall detection accuracy [1]. However, the computational cost of these convolution operations can become significant.

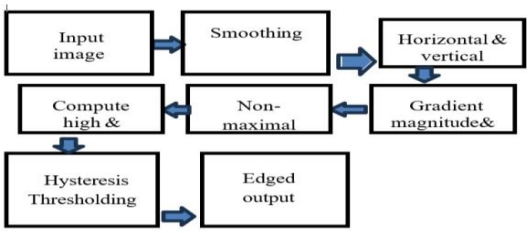

Figure 1: Canny edge detection process flow [2]

2 Related Work

Introduction

Edge detection is a fundamental image-processing technique used to highlight boundaries and important structures within images, typically performed using convolution filters. However, executing these operations sequentially can become slow and computationally expensive, especially with large images. Since this project focuses on applying a parallel convolution approach to process different image parts simultaneously, it enables faster execution and better scalability for real-world applications. Therefore, it is essential to explore existing parallelization methods in edge detection. Below is the literature review organized based on relevant themes.

First Theme - CPU/GPU:

Image processing and computer vision applications are widely used across many domains; however, a major challenge in their development is achieving both high accuracy and low execution time simultaneously. The authors in the study [3] presented a fast and accurate parallel implementation of an extended Canny Edge Detector (extCED) designed for medical imaging on an embedded many-core platform. Also, they proposed a hybrid parallelization strategy called Globally Coarse-grain and Locally Fine-grain (GCLF) to resolve the irregular computation and parallelization challenges introduced by adding a thinning (refinement) stage to improve edge accuracy and smoothness in the proposed extCED. The implementation is developed using OpenCL on a platform with an ARM host processor and 64 processing elements. As a result of experimental on 512×512 medical images the parallel extCED achieves a 25× speedup, runs at 10 frames per second, and produces highly accurate and smooth edges, meeting real-time and accuracy requirements of medical applications.

Also, the researchers [4] provided a high-performance and scalable implementation of the Canny Edge Detector (CED) for modern multicore processors using structured parallel patterns. They introduced a universal computational framework called the Golden Circle of Parallelism (GCP) Instead of relying on hardware-specific acceleration. The GCP organizes parallelization into three structured layers such as shell, kernel, and core. To expose and exploit parallelism. In addition, the main stages of CED Gaussian filtering, Sobel's algorithm, and non-maximum suppression are parallelized using Intel Cilk Plus. While the hysteresis stage remains largely sequential due to inherent data dependencies. The results on 4 core and 8 core Intel processors show improved CPU utilization, balanced workload distribution, and strong scalability compared to serial implementation.

Moreover, the study [5] presents a CPU based multithreaded approach to improve the performance of the Canny edge detection algorithm. The authors first explain the standard Canny process including Gaussian filtering, gradient calculation, non-maximum suppression, and double thresholding. After that they propose dividing the input image into smaller segments processed concurrently using multiple threads. Implementations using one (sequential), two, and four threads are evaluated. As a result, this study shows significant runtime improvements. For example, images of size 550×600 pixels saw execution time reduced from 2.19 seconds sequential to 0.43 seconds dual-threaded, and with further gains using four threads 0.370 seconds. Although minor artifacts appear with higher thread counts, the output quality remains comparable to the sequential version.

In addition, the paper [6] investigates how different parallelization techniques affect the performance and scalability of the Canny Edge Detector on two different parallel architectures. The authors evaluate two main strategies loop-level parallelism and domain decomposition by implementing the algorithm on a multi-core CPU using OpenMP and on a many-core GPU using CUDA. The Canny algorithm is divided into three sections based on potential parallelism. Experimental results show that domain decomposition provides better performance and scalability on multi-core CPUs, especially for large images, due to improved data locality while loop-level parallelism performs better for pure data parallel program running on GPUs. The study also demonstrates that a combined strategy yields the highest speedup on GPUs, achieving up to 33× compared to sequential execution. The paper concludes that no single parallelization strategy is optimal across all platforms, and that the choice must depend on data size, algorithm structure, and underlying hardware architecture.

The study [7] proposes a distributed edge detection approach to demonstrate better scalability and improved time performance of the traditional Canny algorithm when applied to largescale image datasets. The authors improve edge detection accuracy by integrating the Otsu method to find better thresholds for the Canny operator. Also, the optimized Otsu-Canny algorithm is parallelized to handle massive image collections efficiently using the MapReduce model on the Hadoop platform, where images are processed independently across cluster nodes. In addition, the authors conducted an experiment on large datasets derived from Pascal VOC2012 demonstrate that the proposed method shows significantly reducing execution time. By achieves around 67.2% reduction in runtime and a 3.4× speedup on a five-node Hadoop cluster.

Karlik and Ladanyi [8] aimed to optimize the speed of the Canny edge algorithm for object detection by implementing 2D spatial separable convolution to reduce the computational complexity and execution time. The algorithm utilized three steps that include: finding the intensity gradients, Non- maximum suppression to preserve the strong edges, and finally applying a double threshold to identify possible edges. The authors emphasized that Gaussian filtering is the most time-consuming part of the Canny edge detection algorithm, which made them focus on optimizing it. For the implementation, the authors applied a 2D spatial separable convolution, which lowered the number of arithmetic operations compared to standard 2D convolution. The experiment conducted in MATLAB showed that when □ = 3, the optimization saved 268,323,840 operations and reduced computation time by 390.6ms, showing that separable convolution kernels provide faster real-time processing and maintain a good accuracy.

In Jonsson et al.'s research [9], they introduced a new method for applying convolution to images, using Adaptive Molecular Representation (APR) instead of the traditional pixel grid representation of images. The goal of this research was to reduce the amount of memory required to perform convolution on large and sparse images, such as fluorescent microscopy images, as well as to speed up the processing time. The approach used a combination of parallel algorithms run on the CPU (using OpenMP) and GPU (using CUDA) with a projected throughput of 1 terabyte per second (Tbps) from a normal graphics card, while consuming 14 times less memory than conventional methods. The findings of this paper provide evidence that changing the data representation in conjunction with employing parallel processing can enable performance enhancements in the analysis of large images.

The research [10] shows that using parallel processing with a multi-core CPU and the Python multiprocessing library can increase the performance of OpenCV algorithms significantly. The main libraries the author tested were: Gaussian Blurrer, Canny Edge Detector, Resize Image, Convert Color Space. All of the tests show that by executing the above-mentioned algorithms in a parallel manner will result in much lower processing times than if they are executed in a sequenced manner. This effect is exaggerated when handling images with high resolution and the author achieves speeds that were over seven times faster than when executed in a sequential manner. The main contribution of this work is to show that even when a GPU cannot be used to accelerate the performance of OpenCV algorithms, by using simple software techniques on mid-range hardware will provide a substantial increase in speed.

Pacco Zuvieta and Torres Cruz [11] conducted a comparison of the sequential and parallel processes and the use of the Dask library to distribute the workload of processing multiple images. The images were processed with a series of different techniques, including smoothing, detecting edges, and rotating images. Based upon their findings, the Dask library significantly reduced the processing time when applied to a number of images - for example, for the edge detection process, the processing time was significantly less than the previously established average of 0.1275 seconds when processed in the sequential manner, with the Dask library processing the same image in 0.00012 seconds when run in parallel. Based upon their findings, the authors concluded that the implementation of high-level libraries to handle large data sets provides a large benefit in terms of efficiencies and that high-level libraries such as Dask can be implemented without sacrificing quality of output.

Another study [12] presents speeding up the edge detection using parallel processing and pipelining techniques. The authors implemented the algorithm using Visual Studio along with OpenCV, OpenMP, and MPI. Also, they introduce the types of parallelism, such as Bit-level, instruction-level, Data and Task parallelism. Hybrid parallelism combines OpenMP and MPI, which reduces the execution time dramatically regardless of image size. The results show that this method improves performance for any image size and makes the edge detection faster.

Another study in this category, paper [13], demonstrates how the way in which through the use of OpenCL, it is much faster to perform mean filtering since the work is run in parallel rather than sequentially. The authors note that mean filtering examines pixel neighborhoods that are as small as those with the Gaussian blur, making it simple to subdivide the image into sections and process them simultaneously. Due to this, the processing of the image becomes very quick. They break the picture into work-groups and allow the GPU or the CPU to calculate several pixels simultaneously. Their findings indicate high speedups on image sizes. Although the experiment of the paper is mainly concerning mean filtering, it does not fail to demonstrate that any operation relying on surrounding pixels may vastly benefit because of parallel programming models such as OpenCL, OpenMP, and CUDA. Large image workloads are much easier to handle using these models since a large number of pixels can be processed simultaneously.

Following the earlier GPU/CPU approaches, paper [14] extends this discussion by applying parallelism to a real-time robotic system. In the paper, the author integrates the use of the Canny edge detector with BFS and Harris corner detector to create a real-time based system to assist in controlling a robotic arm. The authors are concerned with achieving a higher speed at which the edge detection step is to be implemented in order to enable the system being used to operate successfully with large images and maintain real-time execution. They demonstrate that with the help of parallelization, provided by using GPUs, their system can achieve up to 110 frames per second, and they can maintain stable performance even in situations where the image size is increased. These findings indicate that Canny edge can be implemented in realtime when it is parallelized: when combined with other algorithms, such as Gaussian blur, gradient, suppression, and thresholding, steps in the Canny edge detector can be scaled on the parallel hardware. The research also reveals that even minor improvements can cause realtime systems to react much more effectively.

Additionally, paper [15] states that the Canny edge detection algorithm consists of several steps, including Gaussian blur, gradient calculation, non-maximum suppression, and double thresholding. With all these steps, the algorithm can take a long time to execute. To solve this problem, the author proposes a parallel Canny edge using an OpenCL accelerator, optimized through three techniques: task partitioning, vectorized memory access, and NDRange optimization. OpenCL is a framework that is used to write programs that can run in parallel on different types of hardware. To compare the proposed method, the author implemented the Canny algorithm on a CPU, a multi-threaded CPU, and CUDA. The results of the experiment show that the proposed OpenCL outperforms all other experiments, which shows the effectiveness of this approach.

Moreover, paper [16] investigates accelerating the Canny edge detection algorithm using a heterogeneous architecture, a CPU and a GPU integrated on the same chip. The traditional CPU cannot run the algorithm efficiently because it contains heavy computational steps, such as the Gaussian blur and the gradient calculation. To solve this issue, the author used a platform that contains both CPU and GPU, namely the Jetson TK1, which removes the usual delay that happens when transferring the data between CPU and GPU. The author tested the optimized algorithm on the Jetson TK1 and showed that the algorithm detected the edges faster than the standard Canny embedded on a CPU. The results of this study show that combining both CPU and GPU can achieve real-time performance for edge detection.

Furthermore, A. Bosakova-Ardenska et al. proposed accelerating Gaussian filtering by applying partial sums to reduce unnecessary repeated computations in the Gaussian2D algorithm. The authors developed three implementations: Gaussian1D, Gaussian2D, and Gaussian2DParSums, all parallelized using the MPI library and tested on a five-image RGB dataset using an Intel Core i5-11320H processor. Image quality was compared against MATLAB using the PSNR metric, and execution time was evaluated under 1, 2, 4, and 8 parallel processes. The results showed that Gaussian1D consistently achieved the highest PSNR values, followed by Gaussian2DParSums, while Gaussian2D caused more blurring, especially with larger kernels. Increasing kernel size decreased PSNR due to stronger smoothing. Performance-wise, processing time increased significantly for Gaussian2D as kernel size grew, but the proposed partial-sum method greatly reduced this growth and produced more stable timing. Speed-up improved when using more processes and when filtering larger images. For example, with kernel size 3 and eight processes, speed-up reached ~1.89 for Gaussian1D and ~2.49 for the largest image, while Gaussian2DParSums reached ~2.79. These results confirm that partial sums effectively accelerate Gaussian filtering while maintaining competitive filtering quality, especially compared to the standard 2D implementation [17].

In addition, L. Cadena et al. applied fast implementations of mean, Gaussian 2D, and median filtering algorithms using OpenMP to accelerate MRI brain image denoising. The experiments were performed on 80 MRI images with different kernel sizes (3×3, 5×5, 7×7, 9×9, and 11×11) using an Intel Core i5 CPU. The authors added a noise model (80% additive and 20% impulse noise) and evaluated quality using both PSNR and SSIM. The results showed that the fast median filter achieved the best noise suppression with the highest PSNR and SSIM values, although it slightly suppressed some details. In terms of performance, fast filtering achieved almost 2× speedup with 2 threads, and around 3.2× speedup with 3–4 threads. Acceleration was more effective for kernel sizes larger than 5×5, while the use of four threads showed reduced efficiency due to CPU resource allocation for background processes. Overall, the parallel fast filters provided significant processing time reduction while maintaining acceptable image quality, with median filtering showing the greatest improvement in both speed and noise reduction performance [18].

Lastly, A. Kamalakannan and G. Rajamanickam developed a multithreaded color image processing approach that includes fuzzy-based contrast enhancement and edge detection implemented in Visual C++ with Microsoft Foundation Class support, and their main goal was to improve performance and reduce execution time on multicore processors. Instead of executing the algorithm sequentially, they partitioned each input image into equal blocks and processed these blocks in parallel to take better advantage of the available CPU cores. To evaluate the effectiveness of this lock free multithreading technique, the authors tested both algorithms on an Intel Core i5 quad core processor using ten randomly selected color images with different resolutions ranging from 940×474 to 2880×1800 pixels. Each image was processed five times in both sequential and parallel approaches using two, four, and eight threads in order to calculate an accurate average execution time, and they also computed speedup and performance improvement to compare the results. The findings showed that as the number of threads increased, the execution time decreased and both speedup and performance improvement became more significant especially with larger images. In the contrast enhancement experiments, the speedup ranged from 2.53 to 3.43 times with performance improvement reaching up to about 70.89 percent, while the edge detection experiments achieved a speedup between 2.82 and 3.44 times with performance improvement up to 70.94 percent [19].

Second Theme - CUDA:

Vigil [20] aimed to improve the execution speed and maintain a good accuracy by implementing the Canny edge detection on a GPU using CUDA. The Canny edge algorithm consists of four main steps: Gaussian smoothing, Sobel edge detection, Non-maximum suppression, and Hysteresis Thresholding. In addition, the authors added two methods for improvement: they used the Otsu algorithm to automatically compute the high and low thresholds, and they used interpolation to enhance the quality of edge detection in the nonmaximum suppression stage. All of these steps were adapted to work in parallel on the GPU, with separable convolution used for the Gaussian and Sobel filters and masks stored in GPU constant memory. For implementation, Vigil worked with grayscale images and equal rows and columns, and they used a hybrid approach of GPU and CPU to in which the CPU controlled iterations through a flag variable to see whether the pixels that were classified before to be either a broader one or not. The experimental results demonstrate the comparison between the implementations of GPU against CPU versions in OpenCV and MATLAB, it is showing that the speed up of GPU increased with image size. Moreover, vigil observes that the hysteresis threshold spends a large amount of time running.

Luo and Duraiswami [21] aimed to implement the complete canny edge detection algorithm on the GPU using CUDA and to compare its performance. The algorithm consists of 4 steps: Gaussian smoothing, Sobel edge detection, Non-maximum suppression Hysteresis Thresholding. To adapt it to the parallel execution on Nvidia CUDA, they applied the separable convolution function for the Gaussian smoothing and Sobel detection to reduce the computational cost. After computing the gradients, Non-Maximum suppression compared each pixel with its neighbor to identify ridge pixels, then hysteresis, they employed the breadth-first search (BFS) in multi-threads, where each one will manage a local queue. However, because the threads cannot communicate with each other, they applied a multiplepass approach. Experimental results showed that the GPU speeds up compared to MATLAB around (30x-80x), and a three to five fold improvement over the OpenCV implementation, with the hysteresis step that was time-consuming, which was considered a main bottleneck.

Yadav and Gupta [22] aimed to accelerate the canny edge detection by applying it on NVIDIA CUDA to achieve a high performance on GPUs. The algorithm is the standard canny edge detection. First, they started with converting the RGB images to a grayscale, then applying the Gaussian blur to smooth the images, then using the Sobel filtering to compute the gradients, and performing the non-maximum suppression to eliminate the pixels that are not local maxima by comparing them with their neighbor. Finally, the hysteresis threshold was applied as double thresholding if the threshold is high retains the maximum edge pixel, but if the threshold is low, then it will only keep the weak edges if they're connected with strong edges. For the implementation to optimize the performance, they utilized four different approaches: Naïve implementation, shared memory, memory coalescing, and warp-level optimization. The algorithm was tested on a subset of the ILSVRC 2017 dataset with different resolutions ranging from (128 X 128 to 1024 X 1024). The experimental results showed that the improved kernels outperformed OpenCV by 56X for the 1024*1024 images, and achieved 83.3% compute and memory throughput. However, what remained the primary bottleneck is the hostdevice data transfers.

Mochurad [23] proposed a parallel computing approach for the canny edge detection algorithm using CUDA, and to analyze the complexity of object recognition based on the type and density of the image noise. The implementation of the algorithm consists of five steps: Gaussian blur, intensity gradient calculation, finding local maxima, applying a double threshold, and edge tracking with hysteresis. The author implemented the algorithm using C++ for both sequential and parallel implementations. For the sequential implementation, it used Magick++, the sequential implementation utilized nested for loops to iterate through the pixel buffer. On the other hand, the parallel implementation used CUDA extensions, which provide a multiple threads technique to support parallel execution. Each step has a separate function, so in the implementation, each will have its own kernel, except for the hysteresis thresholding, which was separated into two kernels because of the data dependency. The first kernel compares pixels that have the maximum value to define them as strong pixels. The second kernel to applies a low threshold to pixels was classified as strong edges to compare them with their neighbor to see if they are connected. Finally, to ensure a good parallel performance, all the convolutional kernels were stored in shared memory to have faster access to the memory. Mochurad's experimental results showed that parallel implementation achieved a maximum speedup of 68x when processing large images (10240 X 10240) and 26x when increasing the kernel size to 31 X 31, which showed that the parallel execution time remains constant while the sequential implementation execution time increased linearly. Also, the study showed that the Gaussian noise caused a high complexity object recognition compared to the salt and pepper noise.

Baltayeb and colleagues [24] proposed a fast method to detect the edges of air bubbles within an industrial image through the use of local variance and integral image calculations on mobile GPUs. The authors employed three distinct CUDA cores: one for membership processing, one for membership processing, and another for thinning. When tested on an NVIDIA GTX 780 video card, the algorithm produced a 17x speed increase (for 1024 x 1024 high resolution imagers) over the sequential implementation of this algorithm using a CPU. Compared with traditional techniques such as Sobel and Canny edge detection, the algorithm also displayed significantly higher performance levels within difficult environmental conditions. therefore, this algorithm offers balanced efficiency of CPU and GPU processing power for applications requiring high speed and accuracy during real-time scientific research.

In a study carried out by Liu [25], the author examined how working with a graphics processing unit (GPU) combined with C++ can be beneficial to the speed of performing basic imagers with Gaussian blur and film processing techniques on a C++ image-processing system. For example, in this study, the author developed a method using the spatial and data parallelism of the GPU as a replacement for the constraints of the traditional method of using the CPU. The results from this study showed that by utilizing a multi-threaded GPU architecture when performing these types of imagers, it was possible to achieve significantly less processing time and increased accuracy for the completed image processing results when multiple concurrent imagers were performed using a high level of computation. In particular, this was evident with convolution of Gaussian blur for computation-intensive images, leading to more effective methods in modern computer vision systems.

The authors [26] present an efficient CUDA implementation of the Canny edge detection called CudaCanny using a GPU and compare it to the standard ITK Canny running on a CPU. They used four datasets (B_1, B_2, B_3, B_4) with 100 images each; the datasets B_i were created by manipulating each image from B_{i-1} to create images that have 4 times as many pixels as the original image. The results indicate that CudaCanny outperformed the standard CPU implementation on all image size datasets, with increasing speedup of 3.6X to 50X. That concludes that using a parallel GPU can reduce the processing time of heavy convolution, like Canny, and make it practical for real-time applications.

Shifting to CUDA-based acceleration, paper [27] examines how Gaussian blur performance changes under CUDA and CPU parallelization. It closely examines the reason behind the slowing down of the Gaussian blur on large images and provides an explanation of the problem: the algorithm requires a 2D convolution operation on each individual pixel, and performing the operation in a series loop is expensive as the image widens. The authors compare two parallel versions of a traditional CPU version with two parallel versions, one on a multi-core CPU and one on CUDA GPU. They demonstrate that, despite the fact that multicore CPUs are able to accelerate things a notch higher, the GPU attains a significantly larger change since it handles many pixels simultaneously with thousands of lightweight threads. Their experiments include different image sizes, and the gap between CPU and GPU performance becomes more noticeable as the resolution increases. They also emphasize the way in which the GPUs treat data-parallel processes, particularly filters that share the same pattern of calculation between two pixels. This fact is brought out by the paper to show that parallelism does not merely represent a minor advancement in this case, but it entirely transforms the speed of these filtering processes as the image size increases.

Continuing within the CUDA category, in the paper [28], a complete CUDA implementation of the Canny algorithm is provided, and it is compared to a standard sequential version of the algorithm and a CPU parallel version of the algorithm based on TBB. They split every stage of Canny (grayscale conversion, Gaussian blur, Sobel, suppression, hysteresis) into individual CUDA kernels, and shared memory is used to reduce access time. Their results show huge improvements: the GPU was 100 times faster than the serial CPU, particularly on 4K pictures. They also demonstrate that even more speedup is provided by using shared memory and tiling. It is revealed in the paper that every stage of the Canny edge detector can be easily parallelized since every pixel may be processed individually, and it is also shown that with the help of a GPU, the overall algorithm can be significantly faster.

Moreover, paper [29] compares the performance of sequential and parallel Canny edge detection algorithm using different image sizes and different kernel sizes. In the first experiment, the Canny edge detection was applied to different image sizes that range from 70x70 to 10240x10240. After running the algorithm both in sequential and in parallel, using CUDA, the results show that as the image becomes larger, there is a need to parallelize the algorithm, as the computation time increases drastically. In the second experiment, where there are different kernel sizes ranging from 3x3 to 31x31, the results also show that large kernel sizes benefit from parallelization as they require more computations. The study findings show that parallelization improves the computation time for the Canny edge detection algorithm, and it also shows that it's particularly useful when dealing with large images and kernel sizes.

Finaly, P. Sriramakrishnan et al. presented GPU accelerated parallel edge detection algorithms for MRI volumes using CUDA technology. They applied seven operators including Roberts, Prewitt, Sobel, Prewitt Separable, Sobel Separable, Marr Hildreth, and Canny to process multimodal brain tumor images from the BraTS 2015 dataset which contains 120 skullstripped MRI slices. Their goal was to reduce the heavy computation required when edge detection is performed on the CPU, especially since each voxel operation is independent and highly suitable for parallelism. To evaluate performance, the authors tested the seven methods on different MRI volume sizes (1, 10, 50, 100, and 120 images) using both CPU and GPU implementations. The results demonstrated significant acceleration on the GPU, achieving up to 11× speedup for Roberts, 77× for Prewitt, 46× for Prewitt Separable, 98× for Sobel, 48× for Sobel Separable, and 87× for Marr Hildreth. Although part of the Canny process still runs on CPU, its parallel version reached up to 68× speedup compared to the serial implementation [30].

Third Theme - FPGA:

In addition, the authors [31] proposed an improved Canny edge detection implemented on a field programmable gate array (FPGA). They used Otsu's algorithm to adaptively calculate the threshold that separates the images into background and object, but due to its difficulty in implementing directly on an FPGA, the logarithmic operation is introduced to reduce its complexity. The proposed architecture is tested on a grayscale image size of 512×512 from the Standard Test Image Dataset. The results show that the proposed method can detect edges in only 1.231 ms per image when clocked at 50 MHz.

The paper [32] used a heterogeneous CPU-FPGA architecture using OpenCL to speed up the Canny edge detector. The authors used two implementations; in the first, they partitioned the computation between the CPU and the FPGA, and in the second, they used the Weighted Round Robin (WRR) algorithm to partition and distribute the images between the CPU and the FPGA. The results show that CPU-FPGA achieved a 4.8X speed up over CPU-only and 2.1X over FPGA-only implementation. That concludes that using hybrid execution between CPU and FPGA can impact the execution time.

Additionally, the authors [33] present a novel method for speeding up FPGA prototyping of Canny edge detection using High-level Synthesis (HLS) based on the HDL coder. Instead of writing hardware code manually, they describe the algorithm in high-level environments like MATLAB/Simulink to accelerate the FPGA design. Hardware acceleration was implemented on the Xilinx Zynq-7000 SoC platform using an HDL coder. The results show that this approach achieves real-time performance on Canny edge detection using the Xilinx FPGA platform.

Continuing with FPGA-based acceleration methods, the paper [34] describes how SYCL/DPC++ can be used to accelerate the Canny edge detector algorithm on other hardware platforms. The authors rearrange the Canny steps to be efficient in the SYCL model and implement a number of optimization techniques. These include merging some kernels to reduce overhead, unrolling loops to speed up repeated operations, and using pipelining so different parts of the algorithm can run in sequence without waiting. They also reduce how often data moves between CPU and device memory, which is typically a significant cause of slowness. Their performance has been positively reflected with the FPGA versions being the fastest with almost 10 times faster than the CPU implementation. The GPU is also faster than the CPU, but not as fast as the FPGA. The paper highlights the fact that Canny adapts well to parallel hardware and that performance and energy efficiency vary depending on the device. The approach gives developers a single codebase that can be optimized and run on multiple architectures, thus SYCL can be more easily optimized across platforms.

Also, in paper [35] aims to make the Canny edge detection algorithm faster by using a new hardware implementation that runs on FPGAs. Modern images are usually high resolution, which makes the normal Canny edge detection algorithm too slow. To solve this problem, the author proposes an approach that can process 4 pixels at the same time. This approach increased the speed while keeping the memory usage low. The proposed design was synthesized on different FPGA types, including Spartan 3E, Spartan 6, and Virtex 5, but the actual timing measurements were on the Spartan 6. The results show that the design achieved real-time performance even on large images.

Moreover, paper [36] states that the Canny edge detection is the most widely used algorithm in the image preprocessing field. The algorithm consists of heavy computation steps, which make it more complex than other edge detection algorithms. As a result, the paper proposes a distributed Canny edge algorithm that consists of several steps. First, the image is divided into overlapping blocks, and each block is processed independently, rather than the whole image. After that, for each block, the classical Canny steps are performed, including smoothing the image, calculating the gradient, finding edge direction, and applying non-maximum suppression. Also, the author added two steps: block classification and adaptive threshold. The proposed distributed Canny algorithm results in significant speedup and lower computational cost than the original algorithm when implemented on an FPGA.

The last study within this theme, J. Zhou and his team presented their final project, which focused on accelerating the Canny edge detection algorithm on a Zybo Z7-20 FPGA board. They applied the full Canny pipeline on grayscale images, including Gaussian filtering, Sobel gradient computation, non-maximum suppression, zero-padding, and edge tracing using connectivity analysis. To optimize performance, they used several HLS optimization strategies such as loop unrolling, pipelining, array reshaping, and partitioning, in addition to distributed memory access through a combination of line buffers and window buffers. They also explained how the HLS line/window buffer streaming architecture works. And achieved a 7× performance improvement over the baseline version and reduced BRAM usage from 82% down to only 5%. At the end, the team demonstrated their working implementation on the FPGA [37].

3 Crucial Code Snippets and their Parallelizing Potential

1. Sobel Filter

In the Sobel Filter, there are two 3x3 convolutions applied to each pixel in the image, along with several heavy operations, such as sqrt and atan2, as shown in Figure 2 and 3, which is why the execution time takes longer as the image size increases. It's a perfect part to parallelize, as each pixel in the image can be processed independently.

In the code, this function contains two nested loops that iterate over the image dimensions and two additional inner loops for the 3×3 Sobel kernels. Each pixel's gradient magnitude and direction are computed independently, which means the entire image can be divided into regions or rows that could be processed simultaneously. Because there are no dependencies between pixels, this stage has a very high potential for parallelization, and it is expected to yield the largest speedup when threads are applied.

```
void sobel_filter(float **img, float **sobel_img, float **theta, int h, int w){
    int padd = 1;
    float sobel_x[3][3] =
    {
        {-1, 0, 1},
        {-2, 0, 2},
        {-1, 0, 1}
    };
    float sobel_y[3][3] =
    {
        {1, 2, 1},
        {0, 0, 0},
        {-1, -2, -1}
    };
```

Figure 2: Sobel filter 1

```
    for(int i=padd; i<h-padd; i++){
        for(int j=padd; j<w-padd; j++){
            float grad_x = 0;
            float grad_y = 0;
            for(int k_i=-padd; k_i<=padd; k_i++){
                for(int k_j=-padd; k_j<=padd; k_j++){
                    grad_x += img[i+k_i][j+k_j]*sobel_x[k_i+padd][k_j+padd];
                    grad_y += img[i+k_i][j+k_j]*sobel_y[k_i+padd][k_j+padd];
                }
            }
            sobel_img[i][j] = sqrt(pow(grad_x,2) + pow(grad_y,2));///758*255;
            theta[i][j] = atan2(grad_y, grad_x);
        }
    }
    //#sobel_img = sobel_img / sobel_img.max() * 255
    //return sobel_img, theta
    return;
}
```

Figure 3: Sobel filter 2

2. Gaussian Blur

The Gaussian Blur, as shown in Figure 4 and 5, removes the noise from the image by looking at the neighboring rows and columns for each pixel, then multiplying them by a weight value from the kernel, and adding the results. This way, each pixel has an average value of its neighbors, and this operation can be processed independently.
The Gaussian blur performs a convolution between the image and the kernel, looping over each pixel and its neighbors. Since the result of each pixel depends only on its local neighborhood and not on other output pixels, these operations can be distributed across multiple threads. This makes Gaussian blur an embarrassingly parallel task, with predictable workload balance across threads. Parallelizing this stage would significantly reduce execution time for large images while maintaining accuracy.

```
void gaussian_kernel(float** gauss, int padd, float sigma){
    //x, y = np.mgrid[-size:size+1, -size:size+1]
    float coeff = 2 * pow(sigma, 2);
    int k_size = 1 + 2*padd;
    float sum = 0;
    // gauss = np.exp(-(x**2 + y**2) / coeff
    // technically not a true gaussian kernel, but sum(ker) == 1 is more important than closer approx. of gaus. dist.
    for(int i = 0; i < k_size; ++i){
        for(int j = 0; j < k_size; ++j){
            gauss[i][j] = exp(-((i-padd)*(i-padd)+(j-padd)*(j-padd)) / coeff);
            //printf("%f\n", gauss[i][j]);
            sum += gauss[i][j];
        }
    }
    //g /= np.abs(g).sum()
    for(int i = 0; i < k_size; ++i){
        for(int j = 0; j < k_size; ++j){
            gauss[i][j] = gauss[i][j] / sum;
        }
    }
    return;
}
```

Figure 4: Gaussian kernel

```
void gaussian_blur(float **img, float **blur_img, float **gauss, int h, int w, int k_size){
    int padd = k_size/2;
    for (int i=padd; i<h-padd; i++){
        for (int j=padd; j<w-padd; j++){
            for(int k_i=-padd; k_i<=padd; k_i++){
                for(int k_j=-padd; k_j<=padd; k_j++){
                    blur_img[i][j] += img[i+k_i][j+k_j]*gauss[k_i+padd][k_j+padd];
                }
            }
        }
    }
    return;
}
```

Figure 5: Gaussian blur

3. Non-Max Suppression

From the Sobel Filter, each pixel contains the gradient magnitude that shows how strong the edge is in this pixel, and the pixel also contains the gradient direction of the edge. The Non-Max-Suppression, shown in Figure 6 and 7, looks at the edge direction in each pixel and compares the pixel's strength with the neighboring pixels in the same direction. If it's the local maximum, then it's kept; otherwise, it's suppressed to 0. This step makes the edges thin and clearer. The operations in this step can be split across threads and processed independently.
In the implementation, the non-max suppression iterates through the gradient matrix and compares each pixel with its two neighbors in the edge direction. As each pixel's comparison is independent, different regions of the image can be processed concurrently. Although the per-pixel workload is lighter than Gaussian and Sobel, it still benefits from parallelization due to the large number of pixels processed in high-resolution images. Parallel execution would help maintain smooth performance when image sizes scale up.

```
void non_max_suppression(float **img, float **Z, float**theta, int h, int w){
    int padd = 1;
    for(int i=padd; i<h-padd; i++){
        for(int j=padd; j<w-padd; j++){
            theta[i][j] = theta[i][j] * 180 / M_PI;
            if(theta[i][j] < 0){
                theta[i][j] += 180;
            }

            float q = 255;
            float r = 255;
            if (0 <= theta[i][j] < 22.5 || 157.5 <= theta[i][j] <= 180){
                q = img[i][j+1];
                r = img[i][j-1];
            }
            else if(22.5 <= theta[i][j] < 67.5){
                q = img[i+1][j-1];
                r = img[i-1][j+1];
            }
```

Figure 6: Non-max suppression

```
            else if(67.5 <= theta[i][j] < 112.5){
                q = img[i+1][j];
                r = img[i-1][j];
            }
            else if(112.5 <= theta[i][j] < 157.5){
                q = img[i-1][j-1];
                r = img[i+1][j+1];
            }

            if (img[i][j] >= q && img[i][j] >= r){
                Z[i][j] = img[i][j];
            }
            else{
                Z[i][j] = 0;
            }
        }
    }
    return;
}
```

Figure 7: Non-max suppression 2

4. Double Threshold

After Non-Max Suppression, Thresholding classifies each pixel into three categories based on its gradient magnitude: strong edges (≥ high threshold), weak edges (between low and high), and nonedges (< low). Each pixel is handled independently, which means this step can be processed in parallel in later stages, but here we show the sequential core, as illustrated in Figure 8.

Thresholding involves only conditional comparisons for each pixel's value. Each pixel is classified independently, so there are no data dependencies between neighboring pixels. Because of this, the workload can be easily split among threads, each handling a portion of the image. While its computation time is relatively small compared to Sobel and Gaussian, it remains a good candidate for parallel execution due to its simplicity and independence.

```
void double_threshold(float **img, float **out_img, float**theta, int h, int w, float low_thres = 50, float high_thres = 100){
    int padd = 1;
    for(int i=padd; i<h-padd; i++){
        for(int j=padd; j<w-padd; j++){
            if(img[i][j] >= high_thres){
                out_img[i][j] = high_thres;
            }
            else if(low_thres <= img[i][j] && img[i][j] <= high_thres){
                out_img[i][j] = low_thres;
            }
            else{
                out_img[i][j] = 0;
            }
        }
    }
    return;
}
```

Figure 8: Double threshold

5. Hysteresis

Hysteresis promotes weak edges to strong edges if they are connected to any strong neighbor (8connectivity) and repeats until no more changes occur, as shown in Figure 9. It is iterative (multiple passes), but each pass still scans pixels locally. Below is the sequential core loop.
The hysteresis stage repeatedly scans the image until no further changes occur. Each iteration consists of independent neighbor checks, which can be processed by different threads in parallel. However, because the algorithm depends on the results of the previous iteration to determine convergence, synchronization is required between passes. Therefore, this stage has moderate parallel potential — parallelism can be applied within each iteration, but the outer loop must remain sequential to preserve correctness.

```
void hysteresis(float **out_img, int h, int w, float low_thres = 50, float high_thres = 100){
    bool changed = true;
    while(changed){
        changed = false;
        int padd = 1;
        for(int i=padd; i<h-padd; i++){
            for(int j=padd; j<w-padd; j++){
                if (out_img[i][j] == low_thres){
                    changed = true;
                    if ((out_img[i+1][j-1] == high_thres) || (out_img[i+1][j] == high_thres) || (out_img[i+1][j+1] == high_thres)
                        || (out_img[i][j-1] == high_thres) || (out_img[i][j+1] == high_thres)
                        || (out_img[i-1][j-1] == high_thres) || (out_img[i-1][j] == high_thres) || (out_img[i-1][j+1] == high_thres)){
                        out_img[i][j] = 255;
                    }
                    else{
                        out_img[i][j] = 0;
                    }
                }
            }
        }
    }
    return;
}
```

Figure 9: Hysteresis

4 Methodology

4.1 Data Preparation

We used the image dataset from a GitHub repository [38] that contains a variety of image resolutions, including 64x64, 128x128, 256x256, 512x512, and 1024x1024 pixels. These various sizes are suitable for testing the Canny edge detection algorithm performance across small to large images.

4.2 Sequential Algorithm Implementation

A sequential version of the Canny edge detection algorithm was implemented in C++. The code structure and logic were inspired by a publicly available implementation on GitHub [38]. The main stages included are Gaussian blur, Sobel filtering, non-maximum suppression, double thresholding, and hysteresis. The average execution time for each stage was measured to identify computational bottlenecks.

4.3 Parallel Algorithm Implementation

The sequential Canny edge algorithm was parallelized using OpenMP to improve performance. The parallelization was applied to different stages, including Gaussian blur, Sobel filtering, non-max suppression, and double thresholding using #pragma omp parallel for collapse(2) schedule(static). However, the hysteresis stage depends on previous results and therefore, it remains sequential to ensure correct edge connection.

4.4 Synchronization Techniques

Shared variables in Sobel filtering, such as strongCount, sumGrad, and maxGrad need proper synchronization to avoid race conditions. Different synchronization methods were applied, including critical, atomic, and reduction.

4.5 Machine Specifications

All experiments were conducted on a Linux-based virtual machine to ensure a consistent and controlled execution environment. The system runs Ubuntu 24.04.3 LTS (Noble) and is hosted on a KVM-based full virtualization platform. The virtual machine is equipped with an 11th Gen Intel® Core™ i7-1165G7 processor operating at 2.80 GHz, providing 4 physical CPU cores with 1 hardware thread per core (4 logical CPUs total). The system has approximately 9.7 GB of RAM, with 4 GB of swap space.
All programs were compiled using GCC version 13.3.0 with the -O3 optimization flag and OpenMP support enabled via -fopenmp. Parallelization was implemented using the OpenMP API, and all serial and parallel experiments were executed on the same machine under identical conditions to ensure a fair and reliable performance comparison.

5 Results

5.1 Experiment with Race Condition

```
            float mag = sqrt(grad_x*grad_x + grad_y*grad_y);
            sobel_img[i][j] = mag;
            theta[i][j] = atan2(grad_y, grad_x);

            // RACE CONDITION: multiple threads update sumGrad at the same time (lost updates)
            sumGrad += mag;

            // RACE CONDITION: multiple threads increment strongCount at the same time (lost increments)
            if (mag >= 100.0f) strongCount++;

            // RACE CONDITION: multiple threads update maxGrad at the same time (incorrect maximum)
            if (mag > maxGrad) maxGrad = mag;
        }
    }
```

Figure 10: Race conditions variables

There are 3 variables in the sobel_filter function that cause the race condition and are updated by multiple threads without synchronization:

Variable 1: sumGrad

The variable sumGrad produces a race condition because it is a shared variable that is updated by multiple threads simultaneously inside a parallel region. Each thread adds its local gradient magnitude to sumGrad, but since this update is not synchronized, some additions can be lost when threads write at the same time. As a result, the final value of sumGrad becomes incorrect and varies between executions.

Variable 2: strongCount

The variable strongCount causes a race condition because multiple threads increment it simultaneously whenever the gradient magnitude exceeds the threshold. The increment operation is not atomic, so multiple updates may overlap and lead to an incorrect count of strong edges.

Variable 3: maxGrad

The variable maxGrad produces a race condition because several threads may update it at the same time. While one thread is updating maxGrad, another thread may overwrite it with a smaller value. Without proper synchronization, the final maximum gradient may not be the true maximum value across the image.

5.2 Solving Race Conditions with Synchronization Techniques

Figures (11,12,13) show the results after applying the synchronization methods (Atomic, critical, and reduction). We've found that the shared variables sumGrad, strongCount, and maxGrad were equal across the threads. Also, results show that Reduction was the fastest method for most image sizes and has the highest FPS. In contrast, the critical method was the slowest method with the lowest FPS. While Atomic sits in the middle between critical and reduction with a balanced performance, at image size 1024x1024, it achieved the lowest execution time and highest FPS compared to other methods.

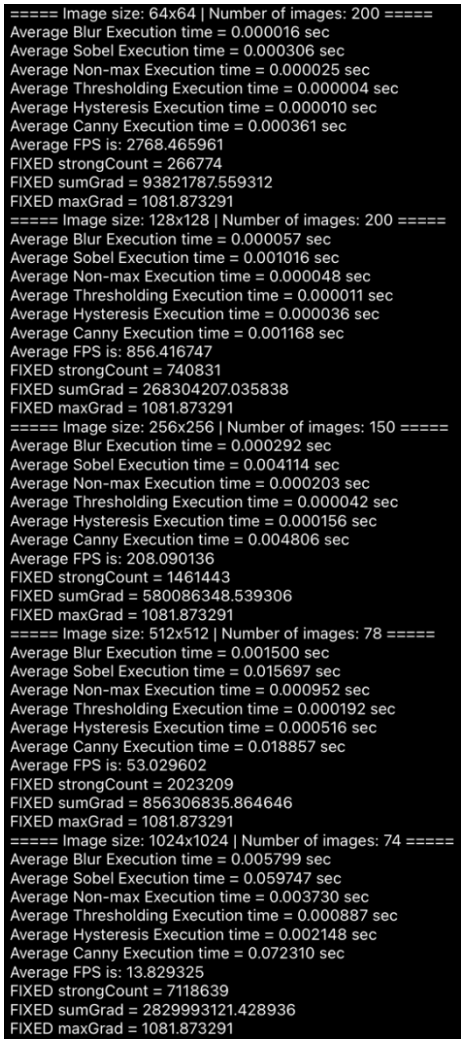

Figure 11: Atomic Results

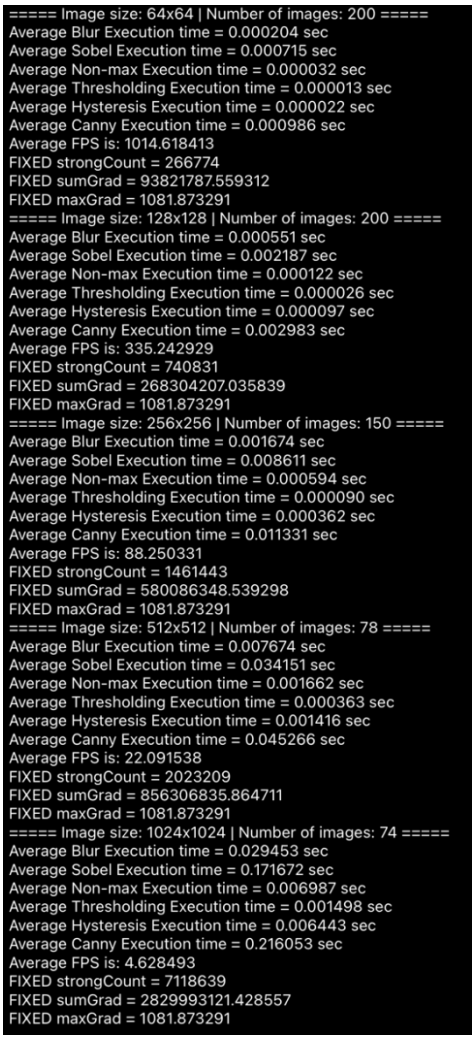

Figure 12: Critical Results

Figure 13: Reduction Results

5.3 Benchmarking Using Performance Metrics

After implementing and evaluating the three synchronization methods, which are atomic, critical, and reduction**,** and discussing their behavior previously, the reduction method demonstrates the best overall performance when specifying the number of threads, as shown in Table I.

The results indicate that performance improves as the number of threads increases from 1 to 4. At 4 threads, the method achieves the highest speedup values, ranging approximately from 1.808 to 2.699 across different image sizes. This improvement reflects effective utilization of parallelism while keeping synchronization overhead relatively low. In terms of efficiency, the values range between 0.452 and 0.675, which confirms that the available parallel resources are used efficiently. Overall, this behavior indicates good scalability up to 4 threads.

However, when increasing the number of threads to 8, performance generally degrades, especially for smaller image sizes such as 64×64 and 128×128. Although the execution time for larger image sizes (256×256**,** 512×512, and 1024×1024**)** remains lower compared to the single-threaded execution, the speedup does not scale proportionally with the number of threads, and efficiency drops significantly. This behavior reflects limited scalability at higher thread counts, mainly due to increased parallel overhead relative to the workload size.

Table I: Reduction Benchmark (Speedup and Efficiency)

Image Size	Threads (p)	Avg Canny Time Tp (sec)	Speedup S(p)=T1/Tp	Efficiency E(p)=S(p)/p
64×64	1	0.000637	1.000	1.000
	2	0.000358	1.779	0.890
	4	0.000236	2.699	0.675
	8	0.003191	0.200	0.025
128×128	1	0.002448	1.000	1.000
	2	0.001628	1.504	0.752
	4	0.001093	2.240	0.560
	8	0.004425	0.553	0.069
256×256	1	0.010198	1.000	1.000
	2	0.006813	1.497	0.749
	4	0.005640	1.808	0.452
	8	0.009042	1.128	0.141
512×512	1	0.042613	1.000	1.000
	2	0.029352	1.452	0.726
	4	0.020900	2.039	0.510
	8	0.027605	1.544	0.193
1024×1024	1	0.169402	1.000	1.000
	2	0.112934	1.500	0.750
	4	0.080018	2.117	0.529
	8	0.088978	1.904	0.238

5.4 Comparing Sequential and Parallel Execution Time

Table II: Comparison between Parallel (Reduction) and sequential runtimes

Image Size	Avg Canny Time (sec) with parallel	Avg Canny Time (sec) with sequential
64×64	0.000249	0.000458
128×128	0.000776	0.001693
256×256	0.003643	0.005474
512×512	0.016270	0.025133
1024×1024	0.100704	0.139772

The performance runtime shown in Table II demonstrates that parallelizing the canny edge algorithm reduces the runtime compared to the sequential version. For image sizes of 64x64 and 128x128, parallel completes the task in significantly less time compared to the sequential version. However, at image size 1024x1024, the execution time of the parallel code started to get closer to the execution time of the sequential code. A primary bottleneck we might face is the sequential nature of hysteresis thresholding, as the edge tracking depends on the previous pixels, which makes it hard to parallelize it.

6 Discussion

6.1 Interpreting the Results

The scalability results show that the impact of increasing the number of threads depends on the synchronization method used. In the reduction implementation, the execution time decreases and the FPS increases as the thread number goes from 1 to 4 across all image sizes. For example, at image size 64 x 64 the execution time drops from 0.000637s at thread 1 to 0.000236s at thread 4 and the FPS increases from 1569.93 to 4234.44 as shown in Figure 14. The atomic method however outperforms the reduction and the critical in most cases, specifically at 1,2, and 8 threads. However, at thread 4 reduction outperformed the other synchronization methods, demonstrating the best overall performance and scalability. The critical method exhibits the poorest performance in increasing the number of threads. This is because this method forces serialized access to shared resources and as more threads are added, synchronization overhead increases leading to degraded performance. At 8 threads, the performance of all methods degrades due to excessive threading overhead. In addition to the analysis, a sample Canny edge detection algorithm is shown in Figures 15 and 16.

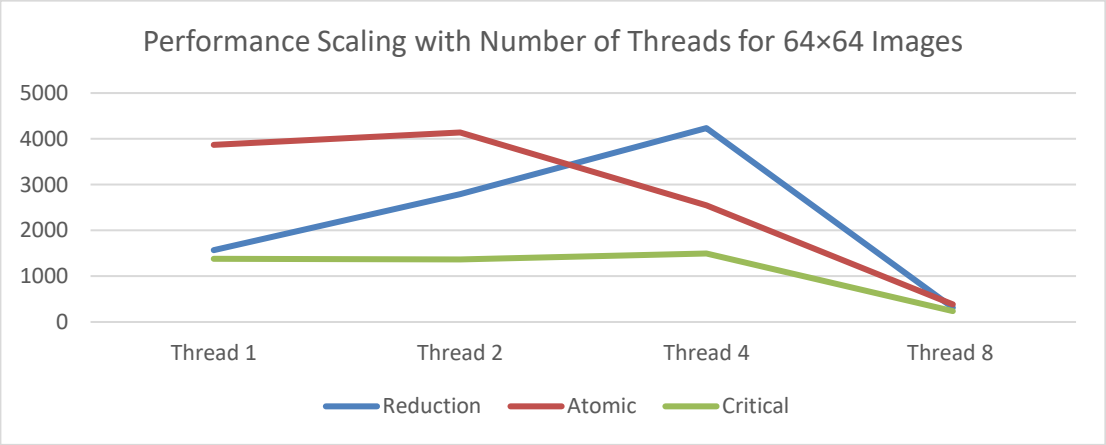

Figure 14: Performance Analysis

Figure 15: Original Image

Figure 16: Result After Applying the Canny Detection

6.2 Challenges Encountered

During the parallelization of the Canny edge detection, we faced several challenges in the Sobel filter stage related to some shared variables. The variables sumGrad, strongCount, and maxGrad caused race conditions because they were updated by multiple threads at the same time without any synchronization. This led to incorrect and inconsistent results, such as lost updates, wrong edge counts, and inaccurate maximum gradient values. Even though synchronization techniques such as atomic, critical, and reduction were used to ensure correctness, they introduced additional overhead that affected the performance as the number of threads increased. Debugging the parallel code was also challenging because errors appeared inconsistently depending on thread execution. These difficulties highlight how difficult it is to balance efficiency and accuracy when parallelizing shared computations.

6.3 Limitations

Through the testing environment and experimental results, several limitations impacted the performance of this project. The hardware configuration was the biggest limitation; the experiments were conducted using a quad-core processor in a virtual machine environment which limited the available physical cores, making it impossible to run more than eight threads at any one time. Thus, running the code with eight threads resulted in performance degradation because of the context-swapping overhead rather than the performance increase anticipated due to parallelized processing. Another primary limitation is that the hysteresis thresholding step in the general process is a serial bottleneck. This step has a data dependency on previously processed pixels for edge tracking. Thus, this step could not be effectively parallelized through the use of standard OpenMP directives, thereby remaining a serial step in the overall process. Benchmarking of the code showed that as the image resolution increased, the performance gained from parallelization diminished. In fact, for images that were 1024x1024 pixels in size, the execution times were beginning to approach those of the serial implementation, thus indicating that the performance gains from parallelization were diminishing. The current methods used for synchronization have proven inefficient. While a critical section method may provide sufficient synchronization for a small number of threads, as the number of threads increases, so does the overhead of the critical section method, thus resulting in degraded performance. In contrast, sacrificing some execution time in exchange for reducing the number of critical sections is a more efficient way to ensure efficient synchronization.

6.4 Implications of Findings

The overall results of our study indicate that parallelizing the Canny edge detection algorithm reduces the execution time compared to the sequential version especially in small size images. As the image size increases to 1024 x 1024, the gain from parallelization decreases and the execution time approaches the sequential version, with the reduction method providing the best performance among the evaluated approaches.

6.5 Future Work

To overcome the current limitations and further optimize the Canny edge detector, future work will focus on geared towards transitioning the heavy computational aspects of Gaussian Blur and Sobel Filtering, which consume significant CPU resources, to a GPU platform using NVIDIA CUDA. As outlined in the original project proposal, this effort will capitalize on a substantial amount of parallelism available when processing high-resolution images. Furthermore, future work will also conduct research on novel scheduling methods within OpenMP, including a combination of dynamic, guided and static schedule type methods, to alleviate any possible load imbalance inherent in using a single static schedule type for images that exhibit uneven edge distributions. Future work will also investigate additional improvements through memory optimization by eliminating redundant calculations by integrating shared memory techniques into the overall Canny Edge Detector design to reduce latencies associated with data transfer, in addition to investigating hybrid forms of parallelization to help lessen the effects of the sequential nature of the Hysteresis process.

7 Conclusion

This project demonstrated that parallelizing the Canny edge detection algorithm using C++ and OpenMP significantly improves performance compared to a sequential implementation. By parallelizing the most computationally intensive stages (Gaussian blur, Sobel filtering, non-maximum suppression, and double thresholding). The Parallel execution reduced runtime and achieved good speedup and efficiency, especially up to four threads when using the reduction synchronization method. The results also demonstrated that performance does not scale well at higher thread counts because of higher parallel overhead relative to the workload size. Overall, the project confirms the effectiveness of parallel computing in accelerating image processing algorithms while balancing parallel efficiency, synchronization, and scalability in real-world implementations.

8 Acknowledgment

We would like to acknowledge the collaborative efforts of all team members in the successful completion of this project. Fatima and Anfal were primarily responsible for the code implementation and parallelization of the Canny edge detection algorithm using OpenMP. The remaining tasks, including data analysis, benchmarking, result interpretation, and paper writing, were carried out collaboratively, with the rest of the members contributing equally to these tasks. Also, we would like to express our sincere gratitude to our course instructor, Dr. Rabab AlKhalifa, for her guidance, support, and valuable feedback provided throughout the development of this project. We also appreciate the resources that enabled us to explore parallel computing concepts and apply them effectively to image processing applications. Special thanks are extended to prior research and publicly available studies that formed the foundation of this work.

9 Appendix

The sequential and parallel implementations used in this study are available at:
https://drive.google.com/drive/folders/1vFjQwKhYYBlHDwxwMvMyCxtMnAWkDVIB?usp=drive_link

References

[1] Edge Detection in Image Processing: An Introduction, Roboflow Blog, 2024. [Online]. Available: https://blog.roboflow.com/edge-detection/

[2] H. Agrawal and K. Desai, “Canny Edge Detection: A Comprehensive Review,” International Journal of Technical Research & Science, special issue ISET-2024, pp. 126–133, 2024. [Online].
Available:https://www.ijtrs.com/uploaded_paper/CANNY%20EDGE%20DETECTION%20A%20COMPREHENSIVE%20REVIEW.pdf

[3] T. L. Ben Cheikh, G. Nicolescu, J. Trajkovic, Y. Bouchebaba, and P. Paulin, “Fast and accurate implementation of Canny edge detector on embedded many-core platform,” in Proceedings of the IEEE International Conference on Embedded and Ubiquitous Computing, 2014, pp. 401–404.

[4] H. T. Mogale and B. M. Esiefarienrhe, “High performance Canny edge detector using parallel patterns for scalability on modern multicore processors,” in Proc. IEEE International Conference on Computing, Networking and Informatics (CK’16), El Paso, TX, USA, July 2016.

[5] E. Clark, G. Hotchner, and E. Scaria, “Parallelizing the Canny Edge Detection Algorithm,” Final Project Report, Dept. of Computer Science and Engineering, University of Central Florida, Orlando, FL, USA, 2016.

[6] T. L. Ben Cheikh, G. Beltrame, G. Nicolescu, F. Cheriet and S. Tahar, "Parallelization strategies of the canny edge detector for multi-core CPUs and many-core GPUs," 10th IEEE International NEWCAS Conference, Montreal, QC, Canada, 2012, pp. 49-52, doi: 10.1109/NEWCAS.2012.6328953.

[7] J. Cao, L. Chen, M. Wang, and Y. Tian, “Implementing a parallel image edge detection algorithm based on the Otsu-Canny operator on the Hadoop platform,” Computational Intelligence and Neuroscience, vol. 2018, Art. no. 3598284, pp. 1–12, May 2018, doi: 10.1155/2018/3598284.

[8] M. Králík and Libor Ladányi, “Canny Edge Detector Algorithm Optimization Using 2D Spatial Separable Convolution,” Acta electrotechnica et informatica, vol. 21, no. 4, pp. 36–43, Dec. 2021, doi: https://doi.org/10.2478/aei-2021-0006.

[9] J. Jonsson, B. L. Cheeseman, S. Maddu, K. Gonciarz, and I. F. Sbalzarini, “Parallel Discrete Convolutions on Adaptive Particle Representations of Images,” IEEE Trans. Image Processing, vol. 31, pp. 4197-4210, Jun. 2022, doi: https://www.researchgate.net/publication/361314931_Parallel_Discrete_Convolutions_on_Adaptive_Particle_Representations_of_Images.

[10] A. Nasir, “Parallel Optimization of OpenCV Functions: Enhancing Image Processing Efficiency with Multicore CPU Execution,” SSRN, 2025,doi : https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5380056.

[11] H. A. Pacco Zuvieta and F. Torres Cruz, "Efficiency of Image Processing with Parallel Techniques," Preprints.org,2024, doi:https://www.researchgate.net/publication/383135872_Efficiency_of_Image_Processing_with_Parallel_Techniques.

[12] M. L. M. Livingston, S. C. Singh, K. Manojkumar, and S. S. Kumar, "A study on parallel and pipelining simulation techniques for edge detection and their performance analysis," Journal of Computational and Theoretical Nanoscience, vol. 16, no. 2, pp. 568–572, Feb. 2019, doi: 10.1166/jctn.2019.7770.

[13] H. Xiao, B. Guo, H. Zhang, and C. Li, "A Parallel Algorithm of Image Mean Filtering Based on OpenCL," IEEE Access, vol. 9, pp. 65001–65016, Jan. 2021, doi: https://doi.org/10.1109/access.2021.3068772.

[14] Y. Yan and S. S. Mokri, "Performance analysis of robotic arm visual servo system based on BFS-canny image edge detection algorithm," Scientific Reports, vol. 15, no. 1, Oct. 2025, doi: https://doi.org/10.1038/s41598-025-19221-1.

[15] Y. Song, C. Li, Q. Zhou, and H. Xiao, "A parallel Canny edge detection algorithm based on OpenCL acceleration," Apr. 04, 2023. https://www.researchgate.net/publication/369862566_A_parallel_Canny_edge_detection_algorithm_based_on_OpenCL_acceleration

[16] G. Sunil. Kumar, J. Kamal. Vijetha, and K. G. S. Venkatesan, "An Investigation of Heterogeneous Embedded CPU and GPU Architectures," International Journal of Scientific Methods in Engineering and Management, vol. 01, no. 03, pp. 20–27, 2023, doi: https://doi.org/10.58599/ijsmem.2023.1303.

[17] A. Bosakova-Ardenska, H. Andreeva, and I. Halvadzhiev, "Fast Parallel Gaussian Filter Based on Partial Sums," EEPES 2025, p. 1, Aug. 2025, doi: https://doi.org/10.3390/engproc2025104001.

[18] L. Cadena, D. Castillo, A. Zotin, P. Diaz, F. Cadena, G. Cadena, and Y. Jimenez, "Processing MRI Brain Image using OpenMP and Fast Filters for Noise Reduction," in *Proceedings of the 2019 8th World Congress on Electrical and Computer Engineering (WCECS)*, 2019, pp. 499–504.

[19] A. Kamalakannan and G. Rajamanickam, "High Performance Color Image Processing in Multicore CPU using MFC Multithreading," International Journal of Advanced Computer Science and Applications, vol. 4, no. 12, 2013, doi: https://doi.org/10.14569/ijacsa.2013.041207.

[20] B. M. López-Portilla, "ACCELERATING THE CANNY EDGE DETECTION ALGORITHM WITH CUDA / GPU," 2015. https://www.semanticscholar.org/paper/ACCELERATING-THE-CANNY-EDGE-DETECTION-ALGORITHM-GPU-L%C3%B3pez-Portilla/f3c0de340623a15d4c96be9a60b3357d5c7b0b73

[21] Y. Luo and Ramani Duraiswami, "Canny edge detection on NVIDIA CUDA," Computer Vision and Pattern Recognition, Jun. 2008, doi: https://doi.org/10.1109/cvprw.2008.4563088.

[22] R. Yadav and H. Gupta, "Optimizing CUDA Kernels for High-Performance Canny Edge Detection," International Journal of Science Engineering and Technology, vol. 13, no. 4, p. 226, Sep. 2025, doi: https://doi.org/10.5281/zenodo.16948653.

[23] L. I. Mochurad, "Canny Edge Detection Analysis Based on Parallel Algorithm, Constructed Complexity Scale and CUDA," Computing and Informatics, vol. 41, no. 4, pp. 957–980, Jan. 2022, doi: https://doi.org/10.31577/cai_2022_4_957.

[24] A. Bettaieb, N. Filali, T. Filali, and H. B. Aissia, "GPU acceleration of edge detection algorithm based on local variance and integral image: application to air bubbles boundaries extraction," Computer Optics, vol. 43, no. 3, pp. 454-460, 2019,doi: https://www.researchgate.net/publication/334597847_GPU_acceleration_of_edge_detection_algorithm_based_on_local_variance_and_integral_image_application_to_air_bubbles_boundaries_extraction .

[25] X. Liu, "Research on the Application of GPU Parallel Computing in Image Processing," International Journal of Advance in Applied Science Research, vol. 4, no. 2, pp. 1-10, 2025,doi : https://www.h-tsp.com/index.php/ijaasr/article/view/73.

[26] L. H. A. Lourenco, D. Weingaertner, and E. Todt, “Efficient Implementation of Canny Edge Detection Filter for ITK Using CUDA,” 2012 13th Symposium on Computer Systems, Oct. 2012, doi: 10.1109/wscad-ssc.2012.21.

[27] N. Ibrahim, A. ElFarag, and R. Kadry, “Gaussian Blur through Parallel Computing,” Proceedings of the International Conference on Image Processing and Vision Engineering, 2021, doi: https://doi.org/10.5220/0010513301750179.

[28] M. Horvath, M. Bowers, and Shadi Alawneh, “Canny Edge Detection on GPU using CUDA,” 2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC), pp. 0419–0425, Mar. 2023, doi: https://doi.org/10.1109/ccwc57344.2023.10099273.

[29] “View of Canny Edge Detection Analysis Based on Parallel Algorithm, Constructed Complexity Scale and CUDA,” Www.cai.sk, 2022. https://www.cai.sk/ojs/index.php/cai/article/view/2022_4_957/1179 (accessed Dec. 06, 2025).

[30] P. Sriramakrishnan, T. Kalaiselvi, and K. Somasundaram, “Parallel Processing Edge Detection Methods for MR Imagery Volumes using CUDA Enabled GPU Machine,” 2018. Available: https://www.ijcseonline.org/spl_pub_paper/PID035.pdf?utm_source

[31] L. Guo and S. Wu, “FPGA implementation of a Real-Time edge Detection system based on an improved Canny algorithm,” Applied Sciences, vol. 13, no. 2, p. 870, Jan. 2023, doi: 10.3390/app13020870.

[32] S. Rahamneh and L. Sawalha, “Efficient OpenCL Accelerators for Canny Edge Detection Algorithm on a CPU-FPGA Platform,” 2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig), pp. 1–5, Dec. 2019, doi: 10.1109/reconfig48160.2019.8994769.

[33] A. Alhomoud et al., “Model-based design of a High-ThroughPut Canny Edge Detection Accelerator on ZYnQ-7000 FPGA,” Engineering Technology & Applied Science Research, vol. 14, no. 2, pp. 13547–13553, Apr. 2024, doi: 10.48084/etasr.7081.

[34] M. Hashemi, M. A. S. Khalid, and R. Christopher, “SYCL-based Acceleration of Canny Edge Detector Algorithm Using DPC++,” 2024 IEEE 67th International Midwest Symposium on Circuits and Systems (MWSCAS), pp. 258–262, Aug. 2024, doi: https://doi.org/10.1109/mwscas60917.2024.10658696.

[35] Christos Gentsos, Calliope Louisa Sotiropoulou, S. Nikolaidis, and Nikolaos Vassiliadis, “Real-time canny edge detection parallel implementation for FPGAs,” International Conference on Electronics, Circuits, and Systems, Dec. 2010, doi: https://doi.org/10.1109/icecs.2010.5724558.

[36] R. Jayarani, “Edge Detection using Distributed CANNY Algorithm and Implementation in FPGA,” 2019. https://www.semanticscholar.org/paper/Edge-Detection-using-DistributedCANNY-Algorithm-in-Jayarani/991d4ff1aeca13aa4f3e9424a1cc94e68c298f95 (accessed Dec. 06, 2025).

[37] J. Zhou, Y. Xu, Y. Sun, and Z. Tao, “FPGA-Based Acceleration of Canny Edge Detection || Final Project || ECE6775 FA23,” YouTube, p. 20, Dec. 2023, Available: https://www.youtube.com/watch?v=pOuEZOvWyRc

[38] Asvilesov, “GitHub - asvilesov/Parallel-Canny-Edge-Detection: A CPU and GPU based implementation for parallelizing the Canny Edge Detection operator,” GitHub. https://github.com/asvilesov/Parallel-Canny-Edge-Detection/tree/master