Accelerating Canny Edge Detection with Parallelization

Min-Hsuan Wang
312551054
michael548562@gmail.com

Shiau-Rung Tsai
312551112
mick20001108.cs12@nycu.edu.tw

Yueh-Sheng Lu
313551144
luyuehsheng4768@gmail.com

1 Abstract

This project focuses on accelerating the Canny edge detection algorithm using parallelization techniques, including SIMD, OpenMP, and CUDA. SIMD enhances performance by processing multiple data points simultaneously, while OpenMP leverages multithreading to efficiently utilize CPU cores. CUDA is employed for GPU parallelization, enabling massive parallel execution of edge detection tasks. The implementation parallelizes key steps of the Canny algorithm, including Gaussian Blur, Gradient Calculation (Sobel Operator), Non-Maximum Suppression, Double Thresholding, and Edge Tracking by Hysteresis. By optimizing the algorithm for various parallel computing paradigms, this work demonstrates significant speedups, making Canny edge detection more suitable for real-time applications.

2 Introduction

Edge detection is a cornerstone in the field of computer vision, providing critical information for applications like object recognition, image segmentation, and motion analysis. Among the various edge detection techniques, the Canny algorithm has long been regarded as a gold standard due to its high accuracy, noise resilience, and ability to detect edges with minimal artifacts. The algorithm consists of several sequential steps: Gaussian Blur for noise reduction, Gradient Calculation (Sobel Operator) for detecting intensity changes, Non-Maximum Suppression to refine edge localization, Double Thresholding to classify strong and weak edges, and Edge Tracking by Hysteresis to ensure edge connectivity. While effective, these steps are computationally intensive, especially when applied to high-resolution images or video streams.

With the proliferation of applications requiring real-time edge detection, such as autonomous vehicles, surveillance systems, and smart IoT devices, the computational bottlenecks of the Canny algorithm become a significant limitation. Traditional single-threaded implementations struggle to meet the demands of modern workloads, particularly when scaling to large datasets or operating on hardware-constrained devices. Addressing this challenge requires innovative approaches that can harness the computational power of modern hardware architectures.

This project explores advanced parallelization techniques to accelerate the Canny algorithm. SIMD (Single Instruction, Multiple Data) enables simultaneous processing of multiple pixels, improving data-level parallelism. OpenMP exploits multithreading to distribute workloads across CPU cores, maximizing resource utilization. CUDA leverages GPU parallelism to handle the massive computations required for edge detection at scale. By carefully optimizing each stage of the Canny algorithm for these paradigms, this work achieves significant speedups, demonstrating the potential for real-time performance even on computationally demanding inputs.

The motivation for this work lies in bridging the gap between the algorithm's computational complexity and the performance requirements of modern applications. By making the Canny algorithm faster and more efficient, this project contributes to enabling its widespread adoption in scenarios where edge detection is a critical component of real-time processing pipelines.

3 Related Work

High Performance Canny Edge Detector using Parallel Patterns for Scalability on Modern Multicore Processors
This paper introduces an optimized implementation of the Canny Edge Detector that employs parallel patterns to boost scalability on contemporary multicore processors. The authors present a technique that harnesses parallel patterns to enhance performance and scalability, showcasing notable performance gains on multicore architectures.

Canny Edge Detector Algorithm Optimization using 2D Spatial Separable Convolution
This paper emphasizes the optimization of the Canny Edge Detector algorithm through 2D spatial separable convolution. The primary objective is to lower the computational complexity of 2D discrete convolution by implementing a separable convolution, which accelerates the application of the kernel matrix to a specified image.

GUD-Canny: a Real-Time GPU-Based Unsupervised and Distributed Canny Edge Detector
This paper presents GUD-Canny, a real-time GPU-based unsupervised and distributed Canny edge detector. The authors tackle the shortcomings of conventional Canny edge detectors, including lengthy multistage processes and intricate computational tasks, by utilizing GPU acceleration to achieve real-time performance.

Accelerated Computational Methods in Image Processing using GPU Computing: Variational Saliency Detection and MRI-CT Synthesis
This paper investigates accelerated computational techniques in image processing through GPU computing, concentrating on variational saliency detection and MRI-CT synthesis. The authors seek to deliver efficient algorithms for real-time resolution utilizing multicore and many-core systems.

Efficient Implementation of Canny Edge Detection Filter for ITK using CUDA
This paper introduces an optimized implementation of the Canny Edge Detection filter for the Insight Segmentation and Registration Toolkit (ITK) utilizing CUDA. The authors reveal substantial performance improvements compared to the standard ITK Canny running on CPU models, emphasizing the benefits of CUDA-enabled solutions.

4 Proposed Solution

This project focuses on optimizing the Canny edge detection algorithm by parallelizing its key steps using SIMD, OpenMP, and CUDA. Each step is carefully tailored to leverage specific parallelization techniques for improved performance:

- **Gaussian Blur:** To reduce noise in the input image, a Gaussian kernel is applied. SIMD accelerates this step by performing convolution operations on multiple pixels simultaneously, while OpenMP distributes the workload across multiple threads to efficiently handle large images. CUDA enables further acceleration by parallelizing convolution computations on the GPU.
- **Gradient Calculation (Sobel Operator):** The gradient magnitude and direction are calculated using the Sobel operator to identify areas of rapid intensity change. SIMD speeds up this computation by vectorizing operations, and OpenMP ensures efficient multithreading. CUDA further optimizes gradient calculations by leveraging GPU cores for parallel processing of image regions.
- **Non-Maximum Suppression:** This step refines edge localization by retaining only the strongest edge pixels along the gradient direction, suppressing non-edges. OpenMP divides the image into subregions for concurrent processing, while CUDA employs shared memory to improve access speed and accelerates the pixel-wise comparison needed for suppression.
- **Double Thresholding:** Edge pixels are classified as strong or weak based on two threshold values. OpenMP ensures that the classification is executed in parallel across image regions, and CUDA efficiently processes pixel classifications using GPU threads. This step prepares the edges for connectivity analysis in the next phase.
- **Edge Tracking by Hysteresis:** Weak edges are transformed into strong edges if they are connected to an identified strong edge, ensuring continuity. This step is inherently iterative and is parallelized by dividing the image into independent subregions, where each region is processed concurrently using OpenMP or CUDA. GPU shared memory helps track edge connections efficiently, enabling faster traversal and accurate results.

By applying SIMD for data-level parallelism, OpenMP for multithreading, and CUDA for GPU-based parallelization, this implementation achieves significant speedups for each phase of the Canny algorithm. These optimizations ensure real-time edge detection performance on high-resolution images and video streams, demonstrating the algorithm's suitability for modern applications requiring both speed and accuracy.

5 Experimental Methodology

The process begins by reading the input image and converting it to grayscale. Then, the following steps are performed sequentially: First, Gaussian blur is applied to reduce noise and smooth the image. Next, the gradient magnitude and direction are calculated to detect edges. Non-maximum suppression is used to refine edge localization by keeping only the strongest pixels. In the double thresholding step, pixels are classified as strong or weak edges. Finally, edge tracking by hysteresis is performed, where weak edges connected to strong edges are retained, ensuring continuous edges. The result is an image highlighting the detected edges.

5.1 Serial Implementation

Gaussian blur:

In this step, Gaussian blur is applied to reduce noise and smooth the input image, making it suitable for subsequent edge detection. The process starts by generating a Gaussian kernel, which defines the weights for averaging neighboring pixels. The kernel size determines the region of influence around each pixel, and the standard deviation controls the spread of the weights, with larger values leading to more extensive smoothing. The weights are calculated using the Gaussian function based on their distance from the kernel's center, and the kernel is normalized so that the sum of all weights equals one, preserving the overall intensity of the image. Once the kernel is generated, the convolution process applies the blur. For each pixel in the image (excluding the edges), the algorithm multiplies the pixel values within the kernel's range by the corresponding kernel weights and sums the results. This produces a new pixel value that replaces the original, effectively averaging intensities to reduce noise. The resulting values are clamped to the valid range of [0, 255] to ensure proper image representation. This step efficiently smooths the image while preserving important structural details, providing a noise-free foundation for the gradient calculation in the next phase of the algorithm.

Gradient Calculation (Sobel Operator):This step involves calculating the image gradient, a critical operation for identifying regions of significant intensity change that correspond to edges. The process begins with the application of the Sobel operator to compute gradient components in both the horizontal G_x and vertical G_y directions. Specifically, two 3x3 Sobel kernels are used: one to detect intensity variations along the x-axis and the other along the y-axis. For each pixel (excluding boundary pixels), the Sobel operator performs a convolution by multiplying the values in a 3x3 neighborhood with the respective kernel and summing the results. This process yields the gradient components G_x G_y, representing the rate of intensity change in the horizontal and vertical directions, respectively. After obtaining the gradient components, the gradient magnitude and direction are computed. The magnitude is defined as

$$\sqrt{G_x^2 + G_y^2}$$

Quantifies the strength of the edge at each pixel. Meanwhile, the direction, calculated using atan2(G_y, G_x), indicates the orientation of the edge relative to the horizontal axis. The direction is further converted from radians to degrees to facilitate subsequent processing steps. These computed gradient properties form the foundation for refining and accurately localizing edges in later stages of the algorithm. The magnitude highlights potential edge pixels, while the direction provides essential information for edge tracing and suppression.

Non-Maximum Suppression: Non-Maximum Suppression is a crucial step in edge detection algorithms, aimed at thinning the edges by suppressing non-maximum values in the gradient magnitude map. In this function, for each pixel in the image, the gradient direction and magnitude are used to determine whether the pixel is a local maximum along the gradient direction. First, the angle of the gradient is adjusted to fall within the range [0, 180]. Then, based on the direction, the neighboring pixels are selected for comparison. Depending on the angle, the neighbors are chosen either

horizontally, vertically, or diagonally. For each pixel, the magnitude is compared to its two neighbors in the gradient direction. If the current pixel's magnitude is greater than or equal to both neighbors, it is retained; otherwise, it is suppressed by setting the value to zero. This process helps in preserving only the strongest edges while eliminating spurious ones, resulting in a more precise edge map.

Double Thresholding: The double thresholding step classifies the pixels in the edge map into three categories: strong edges, weak edges, and non-edges, refining the edges detected by the edge detection algorithm. In this process, each pixel value is compared to two threshold values: a low threshold and a high threshold.

- **Strong edges**: If the pixel value is greater than the high threshold, it is classified as a strong edge (set to 255).
- **Weak edges**: If the pixel value lies between the low and high thresholds, it is classified as a weak edge (set to 128), indicating an uncertain edge.
- **Non-edges**: If the pixel value is below the low threshold, it is considered a non-edge (set to 0).

The low and high threshold values are set to fixed values (20 and 50, respectively), but they could be adjusted. After applying the double threshold, the resulting image clearly marks strong edges, while weak edges remain uncertain and can be further processed in the next step (edge tracking). This approach helps retain important edge information while discarding noise.

Edge Tracking by Hysteresis: Edge tracking is the final step in the Canny edge detection algorithm. It helps determine which weak edges, identified during the double thresholding step, should be retained by following strong edges. The goal is to connect weak edges that are likely part of the same object or feature.In this step, for each pixel, if it is a strong edge (i.e., the pixel value is 255), the algorithm checks its 8 neighboring pixels (vertically, horizontally, and diagonally adjacent). If any of these neighbors are weak edges (i.e., the pixel value is 128), the algorithm marks them as strong edges (255), indicating that they should be kept. This process continues, expanding the strong edges to include neighboring weak edges that are connected. After this, the result image only retains true edges, while any disconnected weak edges are discarded. Through edge tracking, weak edges connected to strong edges are preserved, ensuring the accuracy and completeness of the edge detection.

5.2 SIMD Parallelization

When parallelizing image processing tasks using SIMD (Single Instruction, Multiple Data), such as Gaussian blur and Sobel gradient operations, several key considerations must be taken into account to ensure efficiency and correctness.

First, memory alignment plays a crucial role in maximizing performance. SIMD instructions perform best when the data is properly aligned in memory, typically on boundaries that are multiples of the SIMD register size (e.g., 32 bytes for AVX2). Therefore, ensuring that input image data is allocated with the correct memory alignment (e.g., using `aligned_alloc`) is essential for avoiding costly memory access penalties.

Second, when utilizing SIMD for operations like Gaussian blur and Sobel gradient calculation, the data should be processed in parallel, with each SIMD register handling multiple pixels at once. For instance, in AVX2, a 256-bit register can process eight single-precision floating-point values simultaneously, making it possible to apply the filter to eight neighboring pixels in parallel. This greatly accelerates the computation, reducing the number of iterations required compared to scalar implementations.

Another important aspect is the use of vectorized operations for convolution and gradient calculation. For the Gaussian blur, the convolution kernel needs to be applied to each pixel and its neighbors, which can be efficiently done using SIMD instructions that operate on multiple pixel values in parallel. Similarly, when calculating the Sobel gradient, the X and Y gradients are computed for each pixel, and SIMD can be used to process multiple neighboring pixels at once, further reducing computational time.

However, parallelizing image processing with SIMD also requires careful handling of boundary conditions. In image processing, pixels at the edges of the image do not have neighboring pixels in all directions. This necessitates special handling, either by using padding to extend the image or by applying boundary-specific algorithms that do not rely on missing neighbors.

In conclusion, leveraging SIMD for image processing operations like Gaussian blur and Sobel gradient computation can significantly improve performance by processing multiple pixels simultaneously. However, achieving optimal performance requires attention to memory alignment, boundary conditions, and efficient use of vectorized instructions to maximize parallelism.

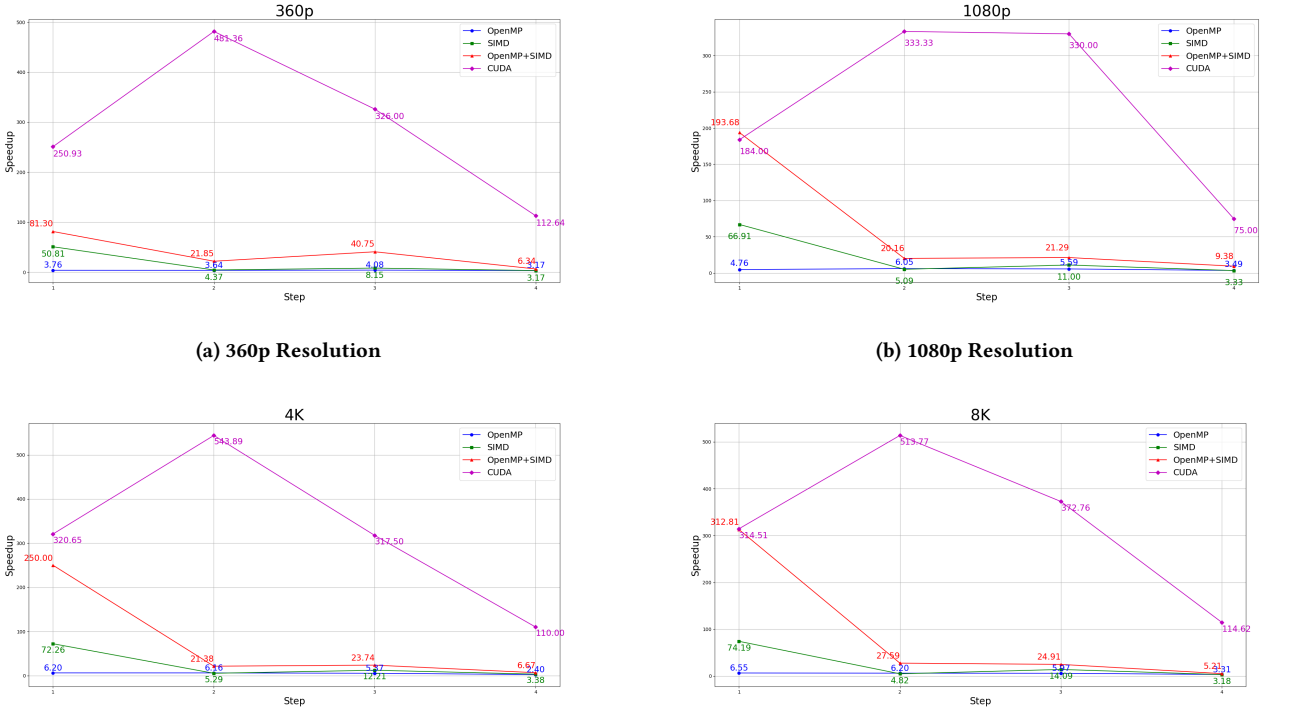
5.3 OpenMP Parallelization

OpenMP was used to parallelize the Canny edge detection algorithm. By adding parallel directives to key sections of the code, such as the gradient calculation and non-maximum suppression stages, OpenMP enabled concurrent execution across multiple threads. Loop-level parallelism was applied to independently process each pixel, while task parallelism allowed the different stages of the algorithm to run simultaneously. This approach significantly reduced processing time, improving the algorithm's efficiency and making it more suitable for real-time applications.

5.4 CUDA Parallelization

In each step of the Canny algorithm, the calculation of each output pixel value is independent, so we assign each output pixel to a GPU thread. Taking the Gaussian blur and Sobel operator as examples, each output pixel's value is calculated by taking the dot product of nearby pixels and the filter, so the convolution operation for each output pixel is clearly independent. Similarly, the computations for other steps such as gradient calculation, Non-Maximum Suppression, Double Thresholding, and Edge Tracking by Hysteresis are also independent for each pixel, meaning that each output pixel is also assigned to a GPU thread.

Currently, the allocation is managed by grouping 16x16 threads (pixels) into a single-thread block. Since image sizes are not necessarily a multiple of 16, we use conditional checks to determine whether the pixel assigned to the thread lies within the image boundaries. If it does not, no computation is needed for that pixel (but due to CUDA's mechanism, some time is still spent executing, though no memory access occurs). Additionally, as mentioned earlier, pixels at the edges of the image do not have neighboring pixels



(a) 360p Resolution

(b) 1080p Resolution

(c) 4K Resolution

(d) 8K Resolution

Figure 1: Comparison of results at different resolutions: 360p, 1080p, 4K, and 8K. These results demonstrate the speedup achieved by our parallelization methods at different resolutions in each step of Canny Edge Detection.

in all directions, requiring special handling, either by padding the image to extend it or by applying boundary-specific algorithms that do not depend on missing neighbors.

For further improvement of the CUDA implementation, we considered that, assuming a kernel size of 3x3 for Gaussian blur and Sobel gradient calculations, most input pixels would be accessed 9 times from off-chip memory. Since access time to off-chip memory is relatively long and can significantly affect performance, we proposed copying the input data to shared memory before computation to reduce the number of global memory accesses. However, this approach did not lead to any noticeable speed improvement. The likely reason for this is that the current bottleneck does not lie in global memory access. Future work can involve investigating potential bottlenecks in the CUDA implementation of the algorithm.

5.5 Experimental Setup

In all our parallelization methods, we combine the Non-Maximum Suppression step and Double Thresholding into one step, resulting in a total of four steps. This approach is taken because, in GPU parallelization, reading from and writing to off-chip memory before and after each step is time-consuming. In our CUDA implementation, we store the results of Non-Maximum Suppression in local registers and proceed directly to the Double Thresholding step without additional memory accesses. To facilitate comparison, we also adopt the four-step approach in other parallelization methods.

(1) Gaussian blur
(2) Gradient Calculation
(3) Non-Maximum Suppression & Double Thresholding
(4) Edge Tracking by Hysteresis

In all experiments, the SIMD width is set to 8, and the number of OpenMP threads is 6. The computing platform used for our implementation is Debian 12.6.0, with an Intel(R) Core(TM) i5-10500 CPU @ 3.10GHz and a GeForce GTX 1060 6GB GPU. The compiler used is g++-12 and CUDA 12.5.1. The test images used in our experiments had resolutions of 360p, 1080p, 4K, and 8K.

6 Experimental Results

This section evaluates the performance of our various parallelization methods across different resolutions. As shown in Figure 2, these are comparative results for 360p, 1080p, 4K, and 8K resolutions.

In our implementation of OpenMP, we used 6 threads. We observed that, except for the 360p resolution, the speed-up was close to 6. The 360p resolution did not achieve a similar speedup, likely due to its low resolution, resulting in insufficient data, thus limiting the benefits of parallelization. For SIMD, we set the width to 8. The speedup exceeded 8 because we aligned the memory, which was not done in our serial version. It can be seen that CUDA performed the best among all methods. Additionally, the larger the image, the higher the speedup achieved by CUDA.

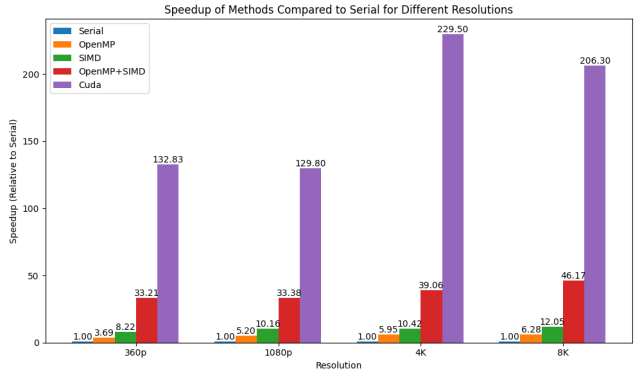
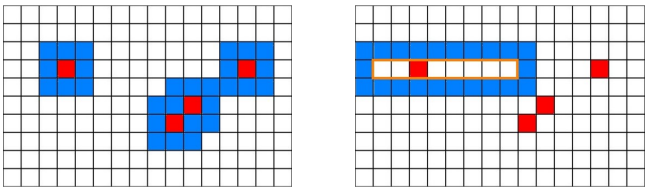


Figure 2: Comparison of results at different resolutions: 360p, 1080p, 4K, and 8K. These results demonstrate the total speedup achieved by our parallelization methods at different resolutions.



(a) Serial **(b) SIMD**

Figure 3: Illustration of Edge Tracking Step

As shown in Figure 1, we also found that in the second step, the Sobel calculation, SIMD was slower than OpenMP. The primary reason is that, when calculating the gradient direction, we need to use the arc tangent function. However, since we are using the GCC compiler, it does not provide an implementation of the arctangent function, so we are still calculating the gradient direction sequentially.

Similarly, in the fourth step, the Edge Tracking, SIMD processes 8 consecutive pixels in a row simultaneously. Therefore, when checking for Weak Edges around Strong Edges, we can only examine the 8 neighbors of every pixel for Weak Edges, then use a mask to determine if the pixel itself is a Strong Edge, as shown in Figure 3b. This is slower compared to the serial implementation, which only checks the neighbors of the Strong Edge.

7 Conclusions

This work demonstrates significant advancements in accelerating the Canny edge detection algorithm through various parallelization techniques. Our implementation successfully leveraged SIMD, OpenMP, and CUDA to achieve substantial performance improvements across different image resolutions, from 360p to 8K. The experimental results revealed several key insights: First, CUDA-based GPU parallelization consistently delivered the highest speedup among all methods, with the performance advantage becoming more pronounced at higher resolutions. This finding underscores the particular suitability of GPU architectures for edge detection tasks in high-resolution images. Second, OpenMP parallelization with 6 threads achieved near-linear speedup for most resolutions, except for 360p where the limited data size restricted the benefits of parallelization. This demonstrates the effectiveness of CPU multithreading for medium to high-resolution images while highlighting the importance of considering data size when choosing parallelization strategies. Third, SIMD parallelization, while generally effective, showed mixed results across different algorithm steps. While memory alignment optimization led to performance gains exceeding the theoretical 8x speedup in some cases, certain operations like gradient direction calculation and edge tracking revealed limitations in the current SIMD implementation. Our contributions extend beyond performance improvements to include practical insights into the implementation challenges and optimization strategies for each parallelization method. The detailed analysis of performance characteristics across different image resolutions provides valuable guidance for selecting appropriate parallelization techniques based on specific use cases.

Future research directions could include:

(1) Investigation of the current bottlenecks in the CUDA implementation.
(2) Development of hybrid approaches that combine multiple parallelization techniques to leverage the strengths of each method
(3) Exploration of adaptive algorithms that can automatically select the optimal parallelization strategy based on input characteristics and available hardware resources
(4) Extension of the current work to handle real-time video processing and streaming applications
(5) Implementation of vectorized versions of complex operations like arctangent calculation to further improve SIMD performance

These findings and future directions contribute to the ongoing evolution of edge detection algorithms in the context of modern computing architectures, particularly as demand grows for real-time processing of high-resolution images and video streams.

References

[1] Eduardo Alcaín Ballesteros et al. 2021. Accelerated computational methods in image processing using gpu computing: variational saliency detection and mri-ct synthesis. (2021).
[2] Antonio Fuentes-Alventosa, Juan Gómez-Luna, and Rafael Medina-Carnicer. 2022. GUD-Canny: a real-time GPU-based unsupervised and distributed Canny edge detector. *Journal of Real-Time Image Processing* 19, 3 (2022), 591–605.
[3] Martin Králik and Libor Ladányi. 2021. Canny edge detector algorithm optimization using 2D spatial separable convolution. *Acta Electrotechnica et Informatica* 21, 4 (2021), 36–43.
[4] Luis HA Lourenco, Daniel Weingaertner, and Eduardo Todt. 2012. Efficient implementation of canny edge detection filter for ITK using CUDA. In *2012 13th Symposium on Computer Systems*. IEEE, 33–40.
[5] Hope Mogale. 2017. High Performance Canny Edge Detector using Parallel Patterns for Scalability on Modern Multicore Processors. *arXiv preprint arXiv:1710.07745* (2017).