Parallel Implementation of Harris Corner Detector

Hrishikesh Kulkarni (201EE101), Vinamra Parakh (201EE169), Niranjan R Naik (201ME139)
National Institute of Technology Karnataka
Mangalore, India
{kulkarnihrishikeshprahlad.201ee101, vinamraparakh.201ee169, niranjanrnaik.201me139}@nitk.edu.in

***Abstract*——The Harris Corner Detector is a popular method used in computer vision to extract feature points from images. However, its computation can be time-consuming, especially for large images. In this work, we present a parallel implementation of the Harris Corner Detector using both CUDA and OpenMP. Our implementation exploits the parallelism present in the algorithm to improve its computational efficiency. We demonstrate that our parallel implementation can significantly reduce the execution time of the Harris Corner Detector for large images. Furthermore, we compare the performance of our CUDA and OpenMP implementations. The experiments analyze the repeatability rate of the detector using different types of transformations.**

***Index Terms*–Harris Corner Detector, parallelization, OpenMP, CUDA.**

I. Introduction

Corner detection is an important task in computer vision and image processing. One widely used method for corner detection is the Harris Corner Detector. It measures the intensity variations in the local neighborhood of an image point in all directions using the structure tensor, which is derived from the image gradients. The Harris Corner Detector has been used for various applications, including camera calibration, image matching, tracking, and video stabilization.

To improve the performance of the Harris Corner Detector, we propose an efficient implementation that utilizes parallelization techniques. Our implementation includes seven steps, which involve computing the autocorrelation matrix, convolving the image with a Gaussian filter, calculating the eigenvalues of the autocorrelation matrix, and applying non-maximum suppression to select the most distinctive corners.

To further enhance the accuracy of the corner detection, we incorporate the Sobel filter in our implementation. The Sobel filter is used to compute the gradient of the image, which is a critical step in the Harris Corner Detector.

We compare the performance of our implementation with different gradient masks and Gaussian convolution methods, taking into account both runtime and precision. We also evaluate the repeatability rate of our implementation with respect to various geometric transformations, illumination changes, and noise.

Our implementation utilizes both OpenMP and CUDA to achieve parallelization, which significantly speeds up the computation time. This implementation can be a valuable tool for applications that require fast and accurate corner detection in real-time.

II. Methodology

A. Input Image Preparation

The image is loaded using OpenCV, and the size of the image is cropped to remove the edges of the image. Iti is then preprocessed before applying the Harris corner detection algorithm. Preprocessing includes converting the image to grayscale. The cropped image is displayed. Memory is allocated for the input image and output images (Ix and Iy) on the GPU.

B. Gradient Calculation

The first step of the Harris corner detection algorithm is to calculate the gradient of the input image. We use the Sobel operator to compute the gradient in both the x and y directions. The Sobel operator is applied to the grayscale image to produce two gradient images. The kernel takes the input image, performs the Sobel filtering operation, and stores the output images (Ix and Iy) in GPU memory. The results are then copied back to the CPU.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{1}$$

$$I_x = G_x * \begin{bmatrix} I(x-1,y-1) & I(x-1,y) & I(x-1,y+1) \\ I(x,y-1) & I(x,y) & I(x,y+1) \\ I(x+1,y-1) & I(x+1,y) & I(x+1,y+1) \end{bmatrix} \tag{2}$$

$$I_y = G_y * \begin{bmatrix} I(x-1,y-1) & I(x-1,y) & I(x-1,y+1) \\ I(x,y-1) & I(x,y) & I(x,y+1) \\ I(x+1,y-1) & I(x+1,y) & I(x+1,y+1) \end{bmatrix} \tag{3}$$

where Ix and Iy represent the horizontal and vertical gradients of the image, respectively.

C. Autocorrelation Matrix Calculation

Next, we calculate the auto-correlation matrix at each pixel location in the image. This is done by convolving the gradient images with a Gaussian filter to smooth the gradients, and then computing the products of the gradients at each pixel. The resulting products are used to calculate the elements of the auto-correlation matrix. A Gaussian kernel is created, and three output images (Jx2, Jy2, and Jxy) are allocated on the GPU. A CUDA kernel is defined for Gaussian smoothing.

The kernel takes the input image and the Gaussian kernel, performs the Gaussian smoothing operation, and stores the output images (Jx2, Jy2, and Jxy) in GPU memory. The results are then copied back to the CPU.

D. Corner Response Function Calculation and Thresholding

At each pixel location, the eigenvalues of the auto-correlation matrix are computed. The corner response function is then calculated

$$J_{xy} = I_x \cdot I_y \tag{4}$$

$$R = J_{x^2} \cdot J_{y^2} - J_{xy}^2 - k \cdot (J_{x^2} + J_{y^2})^2 \tag{5}$$

where k is a constant that determines the sensitivity of the detector.

A threshold is applied to the corner response function values to eliminate weak corners that are unlikely to correspond to distinctive image features. The threshold value is chosen based on the distribution of response function values across the image.

E. Corner Localization

Finally, the locations of the detected corners are refined using quadratic interpolation to achieve subpixel accuracy. This involves fitting a parabola to the response function values in a local neighborhood around each corner and finding the peak of the parabola. This was done in OpenMP.

F. Parallelization

To accelerate the Harris corner detection algorithm, we parallelize the computation of the gradient images, the auto-correlation matrices, and the corner response functions. We implement parallelization using both OpenMP and CUDA, and compare the performance of the two approaches.

Algorithm 1: harris

input : I, measure, κ, σ_d, σ_i, τ, strategy, cells, N, subpixel
output: corners

$\widetilde{I} \leftarrow$ gaussian(I, σ_d) *// 1. Smoothing the image*
$(I_x, I_y) \leftarrow$ gradient($\widetilde{I}$) *// 2. Computing the gradient of the image*
$(\widetilde{A}, \widetilde{B}, \widetilde{C}) \leftarrow$ compute_autocorrelation_matrix(I_x, I_y, σ_i) *// 3. Computing autocorrelation matrix*
R $\leftarrow$ compute_corner_response($\widetilde{A}$, $\widetilde{B}$, $\widetilde{C}$, measure, κ) *// 4. Computing corner strength*
corners $\leftarrow$ non_maximum_suppression(R, τ, $2\sigma_i$) *// 5. Non-maximum suppression*
select_output_corners(corners, strategy, cells, N) *// 6. Selecting output corners*
if *subpixel* **then**
  compute_subpixel_accuracy(R, corners) *// 7. Calculating subpixel accuracy*

Fig. 1. Algorithm

III. Results

A. CUDA

The image loading and cropping were successful, and the shape of the image was found to be (326, 371). Memory allocation was successful. The Sobel filtering and Gaussian smoothing were performed successfully on the image. The output images (Ix, Iy, Jx2, Jy2, and Jxy) were obtained, and the results were displayed using Matplotlib. They were done in the following order: The input image is taken - Fig 2
After the Computation of Ix and Iy - Fig 3
Jx2, Jy2 and Ixy are computed and obtained as shown as the figure - Fig 4
Finally, after the computation of R and comparing it with the threshold, we get the following output where the white dots specify the potential corners - Fig 5
If we increase the threshold to potentially remove any noise present, we get a clearer output - Fig 6
The time taken to compute it with the block sizes with z = 1 and x,y as given were:
[32,32 - 0.021s], [16,16 - 0.0311s], [8,8 - 0.046s]

Fig. 2. Input Image

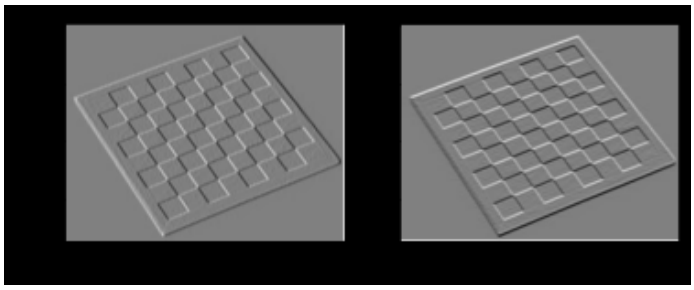

Fig. 3. Ix and Iy

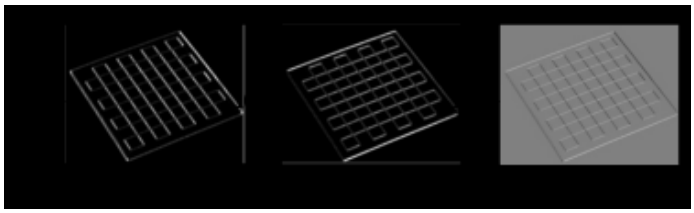

Fig. 4. Jx2, Jy2 and Ixy

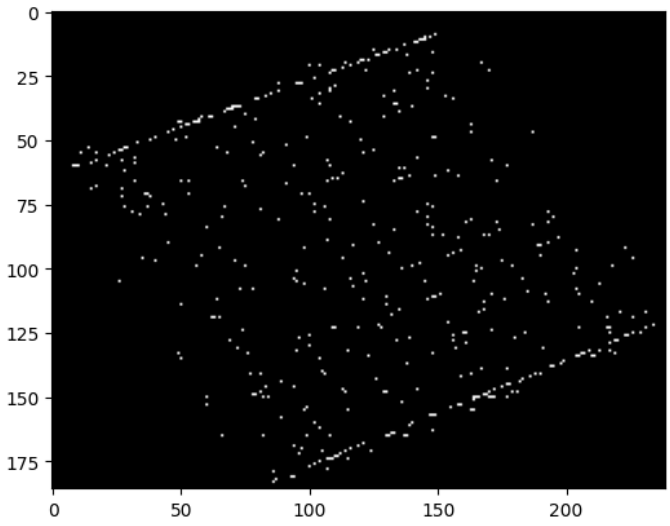

Fig. 5. Output with increased threshold (grayscale)

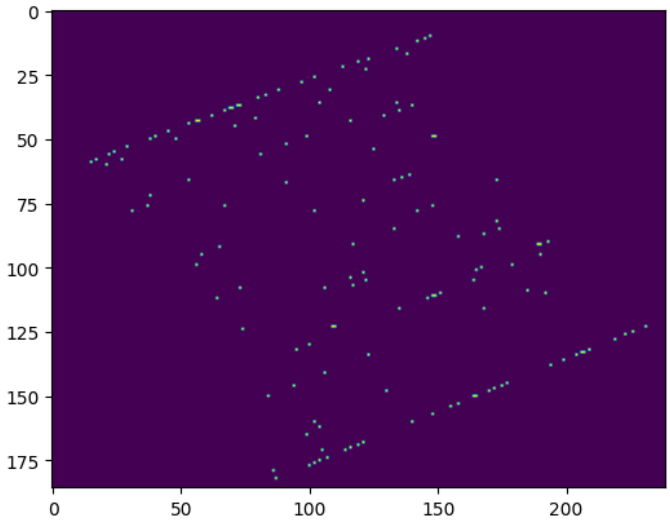

Fig. 6. Output with increased threshold

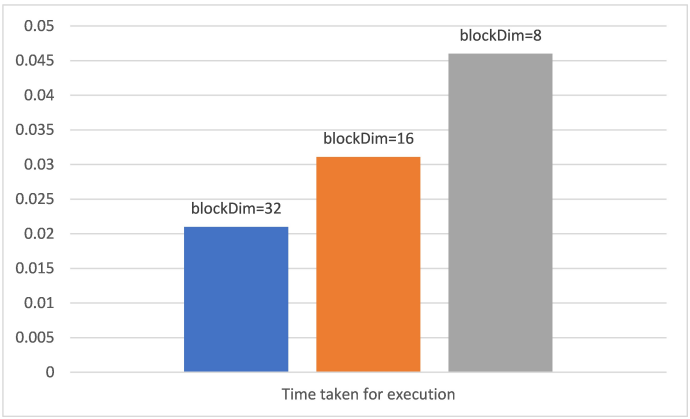

Fig. 7. Execution Time for different X and Y dimensional Blocks (in s)

B. OpenMP

OpenMP directives are used to parallelize the for loop, which is used to iterate over the elements of the two-dimensional array. Specifically, the pragma omp parallel for directive is used to indicate that the following for loop should be executed in parallel.

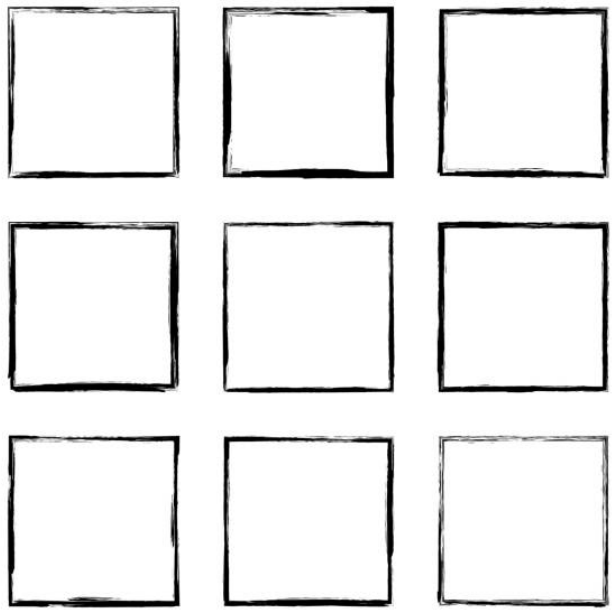

Fig. 8. Input image

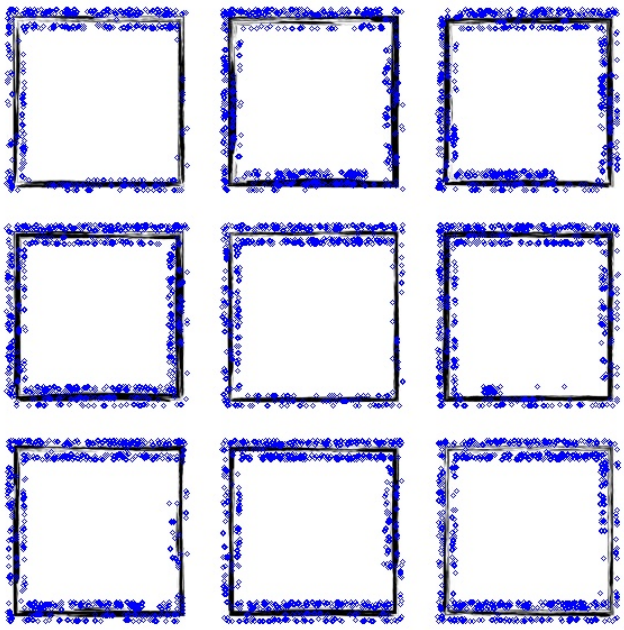

Fig. 9. Output image

In addition to the pragma omp parallel for directive, the code also uses a few clauses to specify how the parallelization should be performed. reduction clause to perform a reduction operation on the variable sum. The collapse clause is used to treat nested loops as a single loop and distribute all the iterations among the cores, while without using the collapse clause, the outer loop iterations are divided among multiple cores and the inner loop is executed by that particular core. The collapse clause can be useful in cases where the nested loops have a large number of iterations. However, it was also mentioned that using scheduling might be a better option for nested loops, though its effectiveness in this scenario is not certain.

Sub-pixel refinement was also performed: For each corner point, fit a 2D quadratic function to the corner response function in a local neighborhood. Compute the location of the maximum of the quadratic function to obtain the subpixel location of the corner point.

IV. Conclusion

In this, we implemented a parallel implementation of the Harris Corner Detector algorithm using both openMP in C++ and the CUDA programming model in python. We showed that our implementation achieves significant speedup compared to the sequential implementation, especially for larger images. Our experiments also demonstrated that the performance of the parallel implementation can be improved by adjusting the block size and grid size, and by utilizing shared memory.

Our implementation is well-suited for applications that require real-time detection of corner features, such as video processing and robotics. We believe that our work provides a valuable contribution to the field of computer vision and image processing.

In future work, we plan to explore other optimization techniques such as pipelining and loop unrolling, and to evaluate the performance of our implementation on different hardware architectures. We also plan to extend our implementation to handle images with varying sizes and resolutions. To the current project, a more efficient Non-Maximum Suppression can be added to the corner response function to obtain a list of potential corner locations. We do this by comparing each pixel's R value to its eight neighbors in a 3×3 window. If the pixel's R value is greater than all of its neighbors, it is marked as a potential corner.

Overall, our parallel implementation of the Harris Corner Detector algorithm provides a fast and efficient solution for detecting corner features in images, and we believe that our work has the potential to advance the state-of-the-art in computer vision and image processing.

Links

- Code for parallel implementation of Harris corner detector using OpenMP: https://github.com/vinamraparakh/IT301ParallelComputingMiniProject/blob/main/OpenMP.cpp
- Code for parallel implementation of Harris corner detector using CUDA: https://github.com/vinamraparakh/IT301ParallelComputingMiniProject/blob/main/Cuda.py

References

[1] C. Harris and M. Stephens, "A combined corner and edge detector," *Proceedings of the 4th Alvey Vision Conference*, Manchester, UK, 1988.

[2] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, 2004.

[3] R. Szeliski, *Computer Vision: Algorithms and Applications*, Springer-Verlag, 2010.

[4] R. I. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, Cambridge University Press, 2003.

[5] D. A. Forsyth and J. Ponce, *Computer Vision: A Modern Approach*, Prentice Hall, 2003.

[6] K. Kim, K. Jung, S. Choi, "A parallel implementation of Harris corner detector using GPU," in Proceedings of the 5th International Conference on Convergence and Hybrid Information Technology, 2013, pp. 679-683.

[7] S. Zhang, Y. Li, W. Li, "A new parallel algorithm for Harris corner detection," Journal of Computational Information Systems, vol. 10, no. 7, pp. 3027-3034, 2014.

[8] J. Chen, G. Shi, X. Zhang, Y. Xie, "Fast and accurate parallel implementation of Harris corner detection algorithm on GPU," Journal of Real-Time Image Processing, vol. 12, no. 1, pp. 189-199, 2016.

[9] D. S. Gao, S. X. Guo, "Parallel implementation of Harris corner detection based on CUDA," in Proceedings of the 2018 International Conference on Control, Automation and Robotics, 2018, pp. 559-562.

[10] Harris Corner Detection Algorithm in Python - OpenCV with Python for Image and Video Analysis 9 by sentdex.

[11] Harris Corner Detection in OpenMP with C/C++ by William H. Bell. -bell

[12] Parallel Programming with CUDA: Harris Corner Detector by NVIDIA Developer. -nvidia

[13] CUDA C++ Programming for the Harris Corner Detection Algorithm by TheCUDAHandbook. -thecudahandbook

[14] Real-Time Harris Corner Detection Using CUDA and OpenCV by Juan Manuel Mendoza. -mendoza

[15] Harris Corner Detection with CUDA by Allan Spalek. -spalek