Parallel Implementation of Harris Corner Detector

Viren Varma - 191ME293
Mechanical Engineering
National Institute of Technology Karnataka
Surathkal, India 575025
Email: viren.191me293@nitk.edu.in

Rahul Maheshwari -
191CH039
Chemical Engineering
National Institute of Technology
Karnataka Surathkal, India 575025
Email: rahul.191ch039@nitk.edu.in

Shreyas Rao - 191EE224
Electrical and Electronics Engineering
National Institute of Technology Karnataka
Surathkal, India 575025
Email: mshreyasrao.191ee224@nitk.edu.in

Abstract—**In this work, we present an implementation and thorough study of the Harris corner detector implemented with parallel computing. This feature detector relies on the analysis of the eigenvalues of the autocorrelation matrix. The algorithm comprises seven steps, including several measures for the classification of corners, a generic non-maximum suppression method for selecting interest points, and the possibility to obtain the corners position with subpixel accuracy. We study each step in detail and propose several alternatives for improving the precision and speed. The experiments analyze the repeatability rate of the detector using different types of transformations.**

Index Terms—**Harris Corner Detector, parallelization, OpenMP, MPI**

I. Introduction

The Harris corner detector is a standard technique for locating interest points on an image. Despite the appearance of many feature detectors in the last decade, it continues to be a reference technique, which is typically used for camera calibration, image matching, tracking or video stabilization. The algorithm relies on the autocorrelation function of the image for measuring the intensity differences between a patch and windows shifted in several directions. The success of the Harris detector resides in its simplicity and efficiency. It depends on the information of the autocorrelation matrix, and the analysis of its eigenvalues, in order to locate points with strong intensity variations in a local neighborhood. This matrix, also called structure tensor, is the base for several image processing problems, such as the estimation of the optical flow between two images. In this work, we propose an efficient implementation of the method, which comprises seven steps including parallelising the convolutional loops using CUDA GPU package in python. The autocorrelation matrix depends on the gradient of the image and the convolution with a Gaussian function. We study the influence of several gradient masks and different Gaussian convolution methods. The aim is to understand the performance of each strategy, taking into account the runtime and precision.

From the eigenvalues of the autocorrelation matrix, it is possible to define several corner response functions, or measures, for which we implement the best known approaches. A non-maximum suppression algorithm is necessary for selecting the maxima of these functions, which represent the interest points. In the following step, the algorithm permits to sort the output corners according to their measures, select a subset of the most distinctive ones, or select corners equally distributed on the image. Finally, the location of the interest points can be refined in order to obtain subpixel accuracy, using quadratic interpolation.

In particular, we rely on the repeatability rate measure to study the performance of our implementation with respect to several geometric transformations, such as rotations, scalings, and affinities, as well as illumination changes and noise.

II. The Harris Corner Detector

The idea behind the Harris method is to detect points based on the intensity variation in a local neighborhood: a small region around the feature should show a large intensity change when compared with windows shifted in any direction.

This idea can be expressed through the autocorrelation function as follows: let the image be a scalar function $I : \Omega \rightarrow R$ and h a small increment around any position in the domain, $x \in \Omega$ Corners are defined as the points x that maximize the following functional for small shifts h,

$$E(h) = \sum w(x) \left(I(x+h) - I(x)\right)^2 , \quad ....(1)$$

i.e. the maximum variation in any direction. The function w(x) allows selecting the support region that is typically defined as a rectangular or Gaussian function. Taylor expansions can be used to linearize the expression I(x+h) as $I(x+h) \simeq I(x)+\nabla I(x)Th$, so that the right hand of (1) becomes

$$E(h) \simeq \sum w(x) \left(\nabla I(x) h\right)^2 dx = \sum w(x) \left(h^T \nabla I(x) \nabla I(x)^T h\right) . \quad ....(2)$$

This last expression depends on the gradient of the image through the autocorrelation matrix, or structure tensor, which is given by

$$M = \sum w(x)\left(\nabla I(x)\nabla I(x)^T\right) = \begin{pmatrix} \sum w(x)I_x^2 & \sum w(x)I_xI_y \\ \sum w(x)I_xI_y & \sum w(x)I_y^2 \end{pmatrix}. \quad ....(3)$$

The maxima of (2) are found through the analysis of this matrix. The largest eigenvalue of M corresponds to the direction of largest intensity variation, while the second one corresponds to the intensity variation in its orthogonal direction. Analyzing their values, we may find three possible situations:

• Both eigenvalues are small, $\lambda 1 \approx \lambda 2 \approx 0$, then the region is likely to be a homogeneous region with intensity variations due to the presence of noise.
• One of the eigenvalues is much larger than the other one, $\lambda 1 \gg \lambda 2 \approx 0$, then the region is likely to belong to an edge, with the largest eigenvalue corresponding to the edge orthogonal direction.
• Both eigenvalues are large, $\lambda 1 > \lambda 2 \gg 0$, then the region is likely to contain large intensity variations in the two orthogonal directions, therefore corresponding to a corner-like structure.

Based on these ideas, the Harris method can be implemented through Algorithm 1. In the first step, the image is convolved with a Gaussian function of small standard deviation, in order to reduce image noise and aliasing artifacts.

```
Algorithm 1: harris
  input  : I, measure, $\kappa$, $\sigma_d$, $\sigma_i$, $\tau$, strategy, cells, N, subpixel
  output: corners
  $\widetilde{I} \leftarrow$ gaussian(I, $\sigma_d$)                                   // 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($I_x$, $I_y$, $\sigma_i$)  // 3. Computing autocorrelation matrix
  R $\leftarrow$ compute_corner_response($\widetilde{A}$, $\widetilde{B}$, $\widetilde{C}$, measure, $\kappa$)     // 4. Computing corner strength
  corners $\leftarrow$ non_maximum_suppression(R, $\tau$, $2\sigma_i$)           // 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
```

The autocorrelation matrix is calculated from the gradient of the image, and its eigenvalues are used to identify any of the previous situations. A non-maximum suppression process allows selecting a unique feature in each neighborhood. In this function, it is necessary to specify a threshold to discard regions with small values. This threshold depends on the corner strength function used and is related to the level of noise in the images. In the last two steps, the points can be selected in several ways and the precision of the corners can be improved by using quadratic interpolation. The following sections explain each of these steps in detail and analyze different alternatives

.

In order to improve the stability of the response, we propose a simple scale space approach in Algorithm 2. The stability is measured by checking that the corner is still present after a zoom out. At each scale, we first zoom out the image by a factor of two and compute the Harris' corners. This is done in a recursive way, as many times as specified by the number of scales chosen (NScales). Then, we compute the corners at the current scale and check that they are present in both scales (through the function select corners). The algorithm stops at the coarsest scale (NScales = 1).
Note that the value of σi is also reduced by a factor of two at the coarse scales. This allows the preservation of the same area of integration. Algorithm 3 selects the points at the finer scale for which there exists a corner at a distance less than σi. The corner position is divided by two inside the distance function.

```
Algorithm 2: harris_scale
  input  : I, NScales, measure, $\kappa$, $\sigma_d$, $\sigma_i$, $\tau$, strategy, cells, N, subpixel
  output: corners
  if NScales$\leq$ 1 then
      // Compute Harris' corners at coarsest scale
      corners$\leftarrow$ harris(I, measure, $\kappa$, $\sigma_d$, $\sigma_i$, $\tau$, strategy, cells, N, subpixel)
  else
      // Zoom out the image by a factor of two
      $I_z \leftarrow$ zoom_out(I)
      // Compute Harris' corners at the coarse scale (recursive)
      corners$_z$ $\leftarrow$
          harris_scale($I_z$, NScales-1, measure, $\kappa$, $\sigma_d$, $\sigma_i/2$, $\tau$, strategy, cells, N, subpixel)
      // Compute Harris' corners at the current scale
      corners$\leftarrow$ harris(I, measure, $\kappa$, $\sigma_d$, $\sigma_i$, $\tau$, strategy, cells, N, subpixel)
      // Select stable corners
      corners$\leftarrow$ select_corners(corners, corners$_z$, $\sigma_i$)
```

```
Algorithm 3: select_corners
  input  : corners$_1$, corners$_2$, $\sigma_i$
  output: corners
  foreach corner in corners$_1$ do
      j $\leftarrow$ 0
      // Search the corresponding corner
      while $j < size(corners_2)$ and $distance^2(corner, corners_2(j)) > \sigma_i^2$ do
          j $\leftarrow$ j+1
      if $j < size(corners_2)$ then
          corners.insert(corner)
      return corners
```

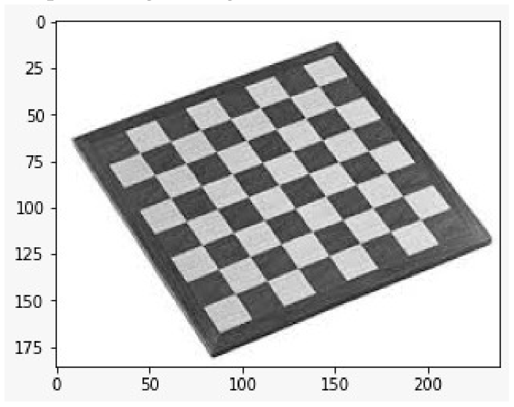
III. NUMBA

Here we use the Numba python library to parallelise our convolution tasks. Numba translates Python functions to optimized machine code at runtime using the industry-standard LLVM compiler library. Numba-compiled numerical algorithms in Python can approach the speeds of C or FORTRAN. Numba is designed to be used with NumPy arrays and functions. Numba generates specialized code for different array data types and layouts to optimize performance. Special decorators can create universal functions that broadcast over NumPy arrays just like NumPy functions do.
Numba supports CUDA GPU programming by directly compiling a restricted subset of Python code into CUDA kernels and device functions following the CUDA execution model. Kernels written in Numba appear to have direct access to NumPy arrays. NumPy arrays are transferred between the CPU and the GPU automatically.

IV. METHODOLOGY

We first code to perform the 2D convolution operation.

Input :We have taken image A of size N x N, Kernel K of size (2M + 1 x 2M + 1).
Output : We got image B of size N x N.



Steps:

1. Zero pad the input image by adding M extra rows of zero to the top and bottom of the image and by adding M columns of zeros to the left and right of the image. For example If 2M+1=3 and N=256 then after zero padding, the new image should have the dimension 258 x 258.
2. We have Implemented the following equation to get the transformation of the input image:

$$B[i,j] = \sum_{m=-M}^{m=M} \sum_{n=-M}^{n=M} K[m,n]A[i-m,j-n]$$

1. Use the following kernel and get the output image (let's call the output image as Ix)

1	0	-1
2	0	-2
1	0	-1

2. Use the following kernel and get the output image (let's call the output image as Iy).

1	2	1
0	0	0
-1	-2	-1

3. Generate three images Ix2,Iy2 and Ixy in the following way. Ix2 and Iy2 are the element wise squares of matrices Ix and Iy respectively. Ixy is the element wise product of matrices Ix and Iy.

During applying convolution to the image using the kernel we use Numba's njit, jit operations using its CUDA gpu facility to parallelize the code.

4. Apply a gaussian kernel to all the three images. The Gaussian kernel is shown below. Let's call the output matrices Jx2,Jy2 and Jxy.

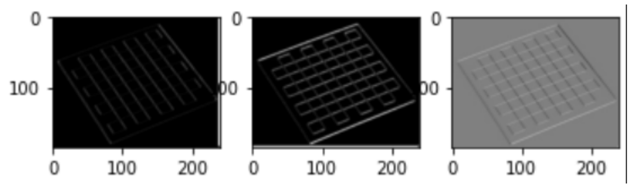
0.0625	0.125	0.0625
0.125	0.25	0.125
0.0625	0.125	0.0625

5. After applying gaussian kernel, for each pixel [i,j] in the image design a 2x2 matrix M whose entries are given as shown

$$M = \begin{bmatrix} Jx2[i,j] & Jxy[i,j] \\ Jxy[i,j] & Jy2[i,j] \end{bmatrix}$$

6. Compute the following parameter,

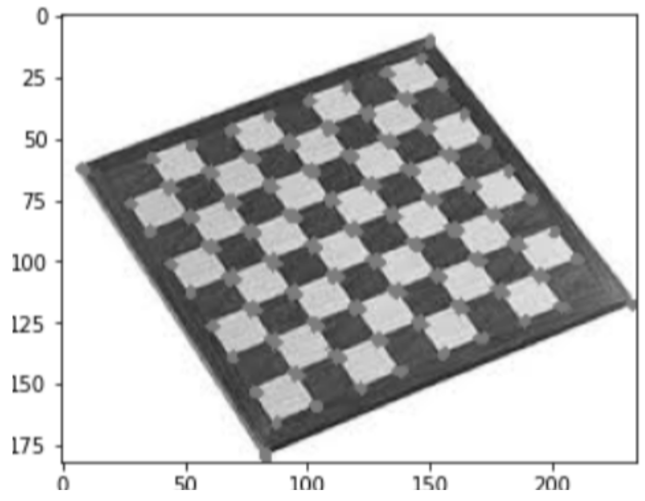
$$R = det(M) - 0.04 * (trace(M))^2$$



7. Generate a binary image by comparing R value for each pixel with a threshold. If R is greater than threshold, assign 1 to the corresponding pixel and 0 otherwise. Try different values of threshold in the range 10^4 to 10^8.

8. Write a function to plot the detected corners (pixels with value 1 in the binary image R) on top of the original image. Pass the generated binary image and original image to this function and observe the output.

Make plots to show the provided input and its output, saved as jpg files. Please submit the plots and the code to us by the time specified in the mail.



V. Computing the Autocorrelation Matrix

Algorithm 4 shows the steps for computing the autocorrelation matrix (3). The products of the derivatives are calculated at each position and the coefficients of the matrix are convolved with a Gaussian function. By default, we use the SII method. The standard deviation, σi, defines the region of integration.

This step is the slowest of the method because of the three Gaussian convolutions. The SII method is very fast and the execution time remains constant independently of the value of σi, except for border initializations. Typically, the runtime of the algorithm increases with the value of σi for other similar filters.

```
Algorithm 4: compute_autocorrelation_matrix
// Gradient of the image and standard deviation
input  : $I_x$, $I_y$, $\sigma_i$
// Coefficients of the autocorrelation matrix
output: $\widetilde{A}$, $\widetilde{B}$, $\widetilde{C}$

// Compute coefficients of the autocorrelation matrix in each pixel
foreach pixel at $(i, j)$ do
    $A(i,j) \leftarrow I_x^2(i,j)$
    $B(i,j) \leftarrow I_x(i,j)\ I_y(i,j)$
    $C(i,j) \leftarrow I_y^2(i,j)$

// Convolve its elements with a Gaussian function
$\widetilde{A} \leftarrow$ gaussian(A, $\sigma_i$)
$\widetilde{B} \leftarrow$ gaussian(B, $\sigma_i$)
$\widetilde{C} \leftarrow$ gaussian(C, $\sigma_i$)
```

VI. Results & Analysis

Linear equations were randomly generated in the form of a matrix of size up to 701 for the performance analysis of the sequential and parallel implementations of the Gaussian elimination algorithm.

VII. Conclusion

In this paper, the various parallel computing approaches which can be leveraged to optimize the Harris Corner algorithm which is popularly used to solve systems of linear equations with a large number of unknowns were analyzed and discussed. All the three parallel computing approaches i.e, Numba library for CUDA gpu were compared with the sequential implementation of Harris Corner algorithm. The current approach performed better than the standard sequential algorithm as theoretically expected by a standard of 4. The relevance of the topic was also very briefly discussed where it is used in numerous real life scenarios such as weather forecasting, scheduling and mainly in image processing.

WORK DISTRIBUTION:

Algo Research:

Shreyas Rao

Initial Algorithm:

Viren Varma, Shreyas Rao

Parallelization:

Shreyas Rao, Rahul Maheshwari

Documentation:

Rahul Maheshwari, Viren Varma

References

[1] An Analysis and Implementation of the Harris Corner Detector by Javier Sa´nchez and Nelson Monzo´n August´ın Salgado.

[2] https://numba.pydata.org/numba-doc/latest/index.html

[3] https://docs.python.org/3/library/threading.html

[4] https://numba.readthedocs.io/en/stable/cuda/index.html