# Harris-Detector General Implementation Using Parallelism

PROJET GPGPU 2023 - 2024

**Ahmad El Acham, El Hassan Hajbi, Fei Meng**

A report presented for
3A MMIS

Supervised by M. Christophe Picard

February 15, 2024

# Contents

# 1 Abstract

The Harris Corner Detection algorithm is a widely used method for identifying corners within an image. Corners are points of interest that exhibit significant changes in intensity in multiple directions. This algorithm operates by observing the changes in intensity resulting from shifting a window in various directions over the image.
We improved the Harris Corner Detection algorithm of Javier Sánchez[1] the slowest step calculation of the autocorrelation matrix. This sped up the method significantly.

# 2 Overview of the harris detector

The Harris Corner Detector is a cornerstone in the field of computer vision, used for detecting corners within images. It is renowned for its ability to identify points of interest that are invariant to rotation, scale, illumination, and viewing angle, making it a powerful tool for feature detection.

## 2.1 Operations

The operations of the Harris Corner Detector can be summarized in the following steps:

1. **Smoothing the Image**:The purpose of this step is to reduce image noise and aliasing artifacts through the convolution with a Gaussian function

2. **Gradient Calculations:** Calculate the image gradients (Ix and Iy) to measure the change in intensity across the image.

3. **Compute the Gradient Products:** For each pixel, calculate the products of gradients ($Ix^2$, $Iy^2$, and IxIy), we also called it autocorrelation matrix. And we can use it to calculate R below.

4. **Gaussian Weighting:** Apply a Gaussian filter to the gradient products to give more importance to pixels near the center of the window.

5. **Compute the Corner Response Function (R):** Calculate the response function using the formula:
$$R = \det(M) - k \cdot (\text{trace}(M))^2$$
where $M$ is the matrix of gradient products, $\det(M)$ is its determinant, $\text{trace}(M)$ is the sum of its diagonal elements, and $k$ is a sensitivity factor.

6. **Non-Maximum Suppression:** Apply non-maximum suppression to filter out all but the most prominent corner responses within a local neighborhood.

7. **Thresholding:** Apply a threshold to the response values to select the final corners. Points with a response above the threshold are considered corners.

# 3 Design methodology

## 3.1 Parallelism introduced to the algorithm

The algorithm consists of seven main steps, and we want to parallelize the longest one. According to the report, the third step, which involves calculating the autocorrelation matrix, is the slowest due to the three Gaussian convolutions. We timed each step and confirmed that it is indeed the slowest part. This step has two sub-steps: computing the coefficients of the autocorrelation matrix in each pixel and convolving its elements with a Gaussian function.
However, this computation for one image doesn't take a significant amount of time (at worst 3s), thus we adapted our code to take as input a list of images, this is meant to increase the sequential time, hence profit from parallel computation.

## 3.2 Calculate the coefficients of the autocorrelation matrix

The first sub-step is relatively simpler. We have the number of pixels of slaves, and each slave will compute the three coefficients using the gradients of this pixel. Once all slaves complete their tasks, the master can obtain three complete matrices.

For calculating the coefficients of the autocorrelation matrix, we discovered that the gradient was being copied and read repeatedly for each slave . To optimize the process, we decided to share the gradient for all slaves and store it in texture memory instead of local memory. And as it is used for all slaves, texture memory is a better choice than shared memory. We wanted to share the gradient for all slaves, so that we tried to use texture memory to store the data of gradient instead of shared memory. Therefore, instead of copying all gradients for each slave, they can just read the texture memory directly to accomplish the calculation. Finally for the result of coefficients, we allocated it in the local memory so that each slave could update its value.

## 3.3 Parallelization of Gaussian Convolution

We had three matrix to apply the Gaussian convolution so that we utilize a 3D grid approach for parallelization, detailed as follows:

### 3.3.1 X Dimension (Pixels)

Each thread applies Gaussian convolution to a specific pixel within the autocorrelation matrices, enabling parallel processing of all pixels in an image.

### 3.3.2 Y Dimension (Multiple Images)

The computation extends across multiple images simultaneously, with each set of threads processing different images to leverage the GPU's high-throughput computation capability.

#### 3.3.3 Z Dimension (Coefficient Matrices)

The z-dimension differentiates between the three matrices (As, Bs, Cs), allowing simultaneous processing and maximizing the GPU's parallel processing capabilities.

### 3.4 Execution Flow

1. **Data Preparation:** Autocorrelation matrix coefficients are organized into vectors (As, Bs, Cs) for efficient memory access.
2. **Kernel Configuration:** Grid and block sizes are optimized based on the image sizes .
3. **Kernel Execution:** we two kernels for the two steps of Gaussian convolution one for that convolves in rows and the other for that convolves in columns , with each thread computing the convolution for its assigned pixel.
4. **Synchronization and Result Handling:** Ensures all convolutions rows finish before the convolutions on columns are done to ensure that the the convolutions on columns is done on the results of the convolution in rows.

### 3.5 Benefits

- **Efficiency:** Significantly reduces computation time compared to sequential approaches.
- **Scalability:** Efficiently scales with the number of images and their resolutions.
- **Flexibility:** Adaptable to various image sizes and convolution kernels by adjusting grid and block sizes.

## 4 Results/Data Analysis

We performed the Harris Corner Detector both sequentially and in parallel on two sets of images with varying sizes. To facilitate accurate comparison of results, we displayed the time taken for each step of the algorithm. Additionally, we printed and compared the number of corners detected to validate the correctness of the parallelization. The implementation of parallelization techniques in the Harris Corner Detection algorithm has shown significant improvements in the performance. We conducted several experiments to analyze the impact of parallel computation using CUDA and compared it with the traditional sequential approach.

### 4.1 Gaussian Computation Times

The first set of experiments measured the time taken for Gaussian computation in both parallel and sequential executions. The results, as shown in Figure 1 and Figure 2, indicate that the parallel (CUDA) implementation is substantially faster than the sequential approach. For example, we reduced the processing time from approximately 4 seconds down to just 0.1 second and with the first set of 10 images and around 40 seconds down to 1 second with the second set of 10 images.

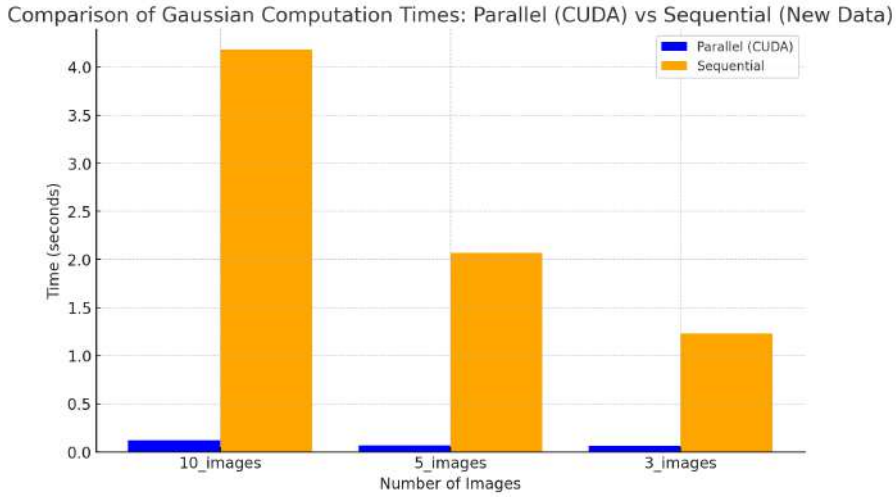


Figure 1: Comparison of Gaussian computation times: Parallel (CUDA) versus Sequential in images of resolution [1600x1200].

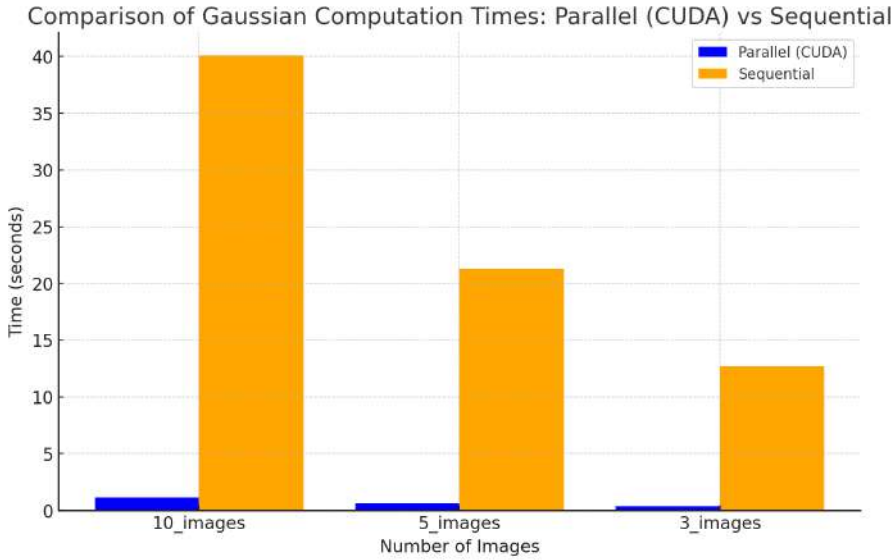


Figure 2: Comparison of Gaussian computation times: Parallel (CUDA) versus Sequential in images of resolution [5000x3000].

## 4.2 Speed Up Analysis

We further analyzed the speed up achieved by parallelizing the Gaussian computation. The speed up factor, illustrated in Figures 3 and 4 , shows how many times the parallel approach is faster compared to the sequential one.

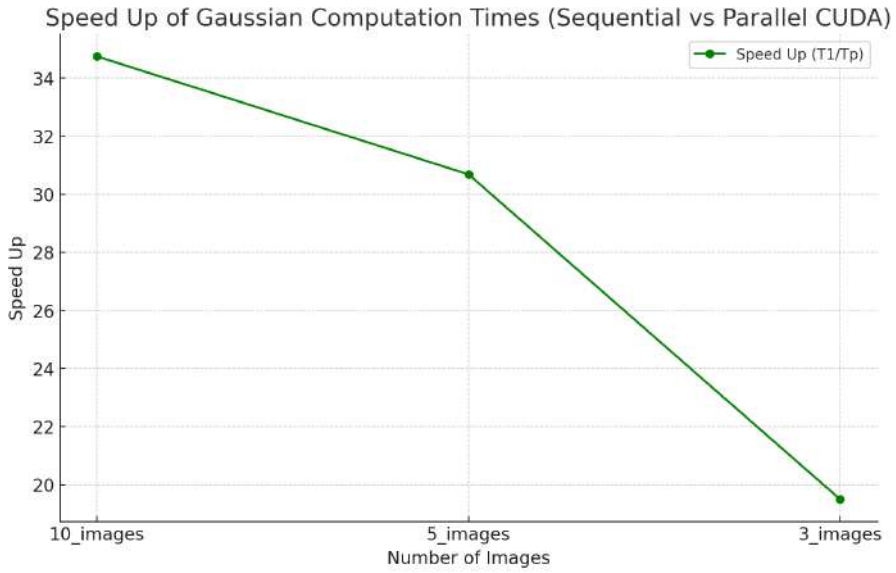


Figure 3: Speed up of Gaussian computation times when comparing Sequential versus Parallel for images of resolution [1600x1200].

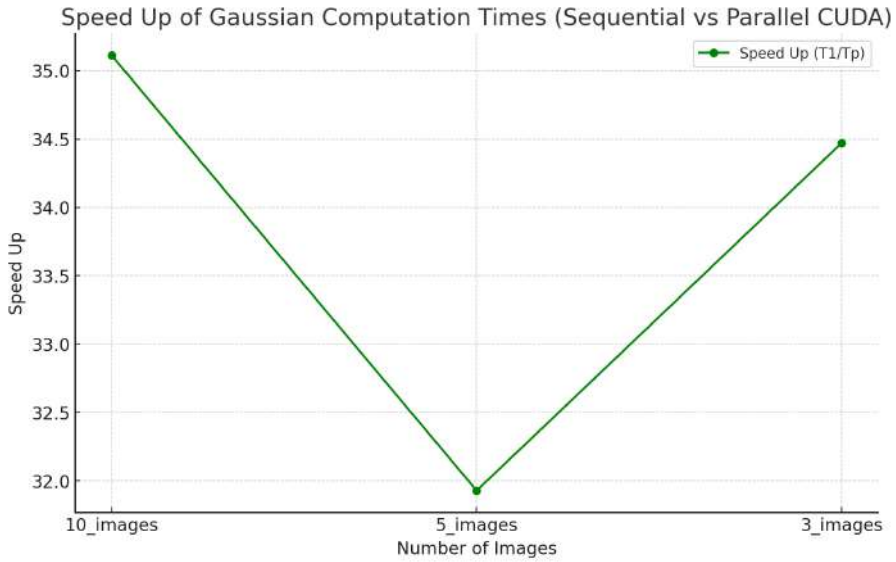


Figure 4: Speed up of Gaussian computation times when comparing Sequential versus Parallel for images of resolution [5000x3000]

We note that the higher the speedup is the better. For the high resolution images, the speed up doesn't vary a lot, however it's high for the three scenarios, ranging from 32 to 35. While for the low resolution images, we have a higher speedup only when we increase the number of images to process and the range of speed up is from 19 to 35.

## 4.3 Effect of Block Size

An additional experiment was conducted to observe the effect of varying the block size in CUDA on the computation time. As depicted in Figure 5, changing the the block size does seem to have a small effect with block size= 64 giving the fastest computation time.

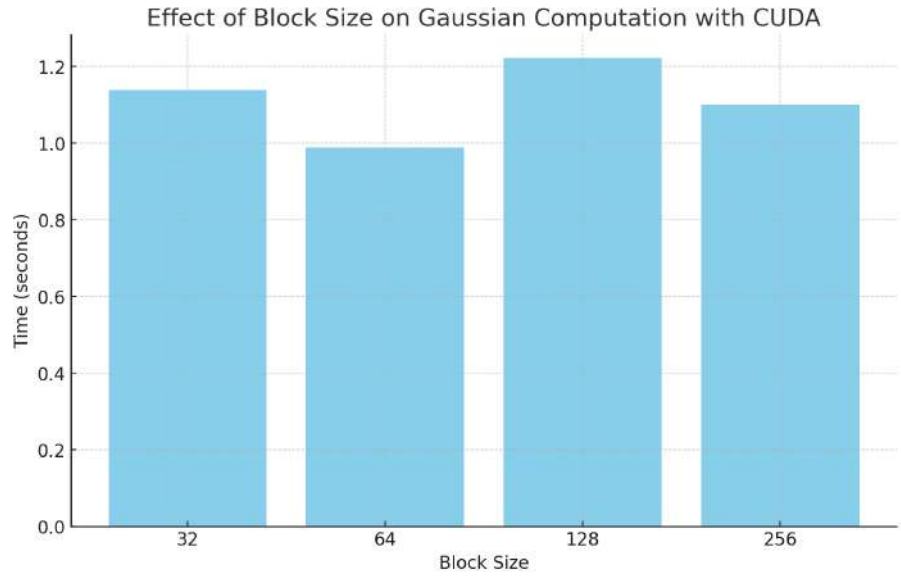


Figure 5: Effect of block size on Gaussian computation with CUDA.

## 4.4 Comparison of computation time for all steps

The comparative analysis of the execution times, as illustrated in Figures 6, indicates a drastic reduction in time for the parallelized steps, especially for the computation of Gaussian convolutions which is the most time-consuming part in the process. The parallel execution using CUDA significantly decreased the processing time, showcasing the effectiveness of parallelism in image processing tasks.

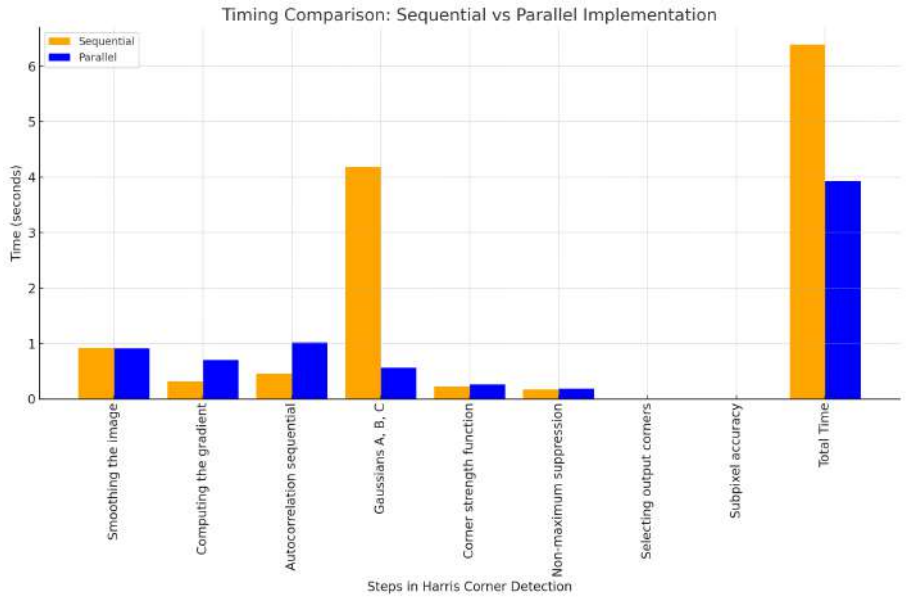


Figure 6: comparison of computation time for all steps in Harris Corner Detector between parallel implementation and sequential.

The consistency of corner detection across different images further validates the accuracy of the parallel implementation. The corner detection algorithm was able to identify

the same number of corners across 10 different images, ensuring that parallelization did not compromise the integrity of the algorithm.

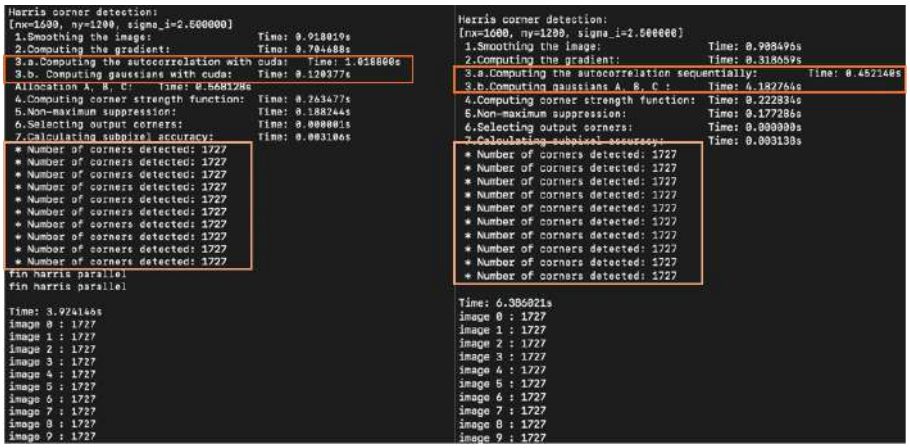


Figure 7: comparison of number of corners detected left is parallel and right is sequential.

Overall, the results highlight the advantages of employing GPU parallelism for computationally intensive tasks in computer vision, providing substantial time savings and efficiency enhancements.

# 5 Conclusion

We attempted to speed up the Harris Corner Detection algorithm by parallelizing its slowest part, which involves calculating the autocorrelation matrix. This approach proved to be effective in significantly accelerating the entire algorithm. Since the calculation for this part is identical for each pixel, we were able to extract and execute it on the GPU instead of the CPU. Based on the results we observed, we were able to significantly reduce the time. The parallelism of the GPU provided better performance for this repeated and computationally intensive task. To write the new code for the GPU, we need to design the architecture and gain a thorough understanding of it to ensure that we provide the correct index in the kernel. To maximize performance, we also need to consider the type of memory, including the speed of memory access, data dependencies, and the life cycle of data. The main difficulty in this project is to rewrite the code CPU on GPU when the code original is complicated. We need to extract the repeated code in the kernel part and adapt it with the index of the kernel. It is simple to introduce the error and it is difficult to debug inside the CUDA kernel.

# References

1 Javier, S., Nelson M., Agustín, S. 2018. An Analysis and Implementation of the Harris Corner Detector. Avalible at:https://www.ipol.im/pub/art/2018/229/article.pdf