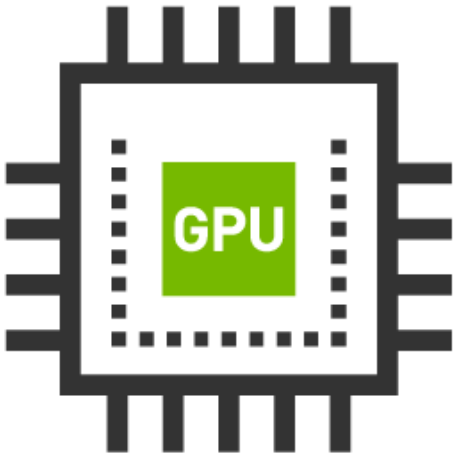
# ECSE 420 - Parallel Computing
# Lab 0 Report




Group 45

Garrett Kinman – 260763260

Ahmed Azhar – 260733580

# Table of Contents

# Hardware Specifications

For reference, the results were obtained locally with an Intel Core i5-7600k CPU and an Nvidia GeForce GTX 1080 GPU. Speedup and relative processing times will vary with different CPUs and GPUs.

# Image Rectification

## Objective

For this part of the lab we had to perform an image rectification on the originally provided image. Image rectification is a transformation routine used to project an image on to a common image plane. For a given input image, the output image is produced by iteratively executing the following operation:

$$Output[\,i\,][\,j\,] = input[\,i\,][\,j\,] \; if \; input[\,i\,][\,j\,] \geq 0, \; 0 \; otherwise$$

Each pixel has four integer values (i.e. RGBA) that range as [0, 255]. Rectification will not change the original image unless it has negative values. Since there are no negative pixel values for PNG images we had to use 127 instead of 0 for the shift.

$$Output[\,i\,][\,j\,] = input[\,i\,][\,j\,] \; if \; input[\,i\,][\,j\,] \geq 127, \; 127 \; otherwise$$

## Example

Before Image Rectification

After Image Rectification

## Observation

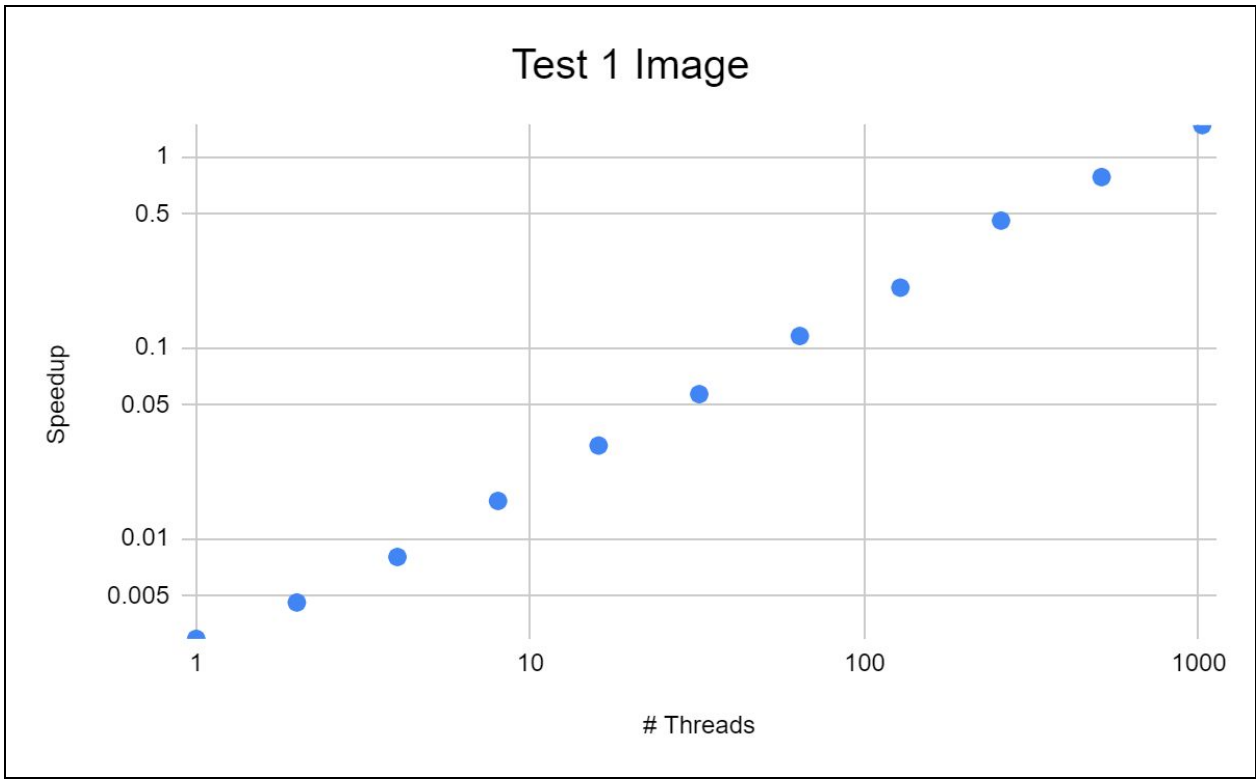



*Figure 1: Speedup vs Threadcount for rectifying Test 1 Image*

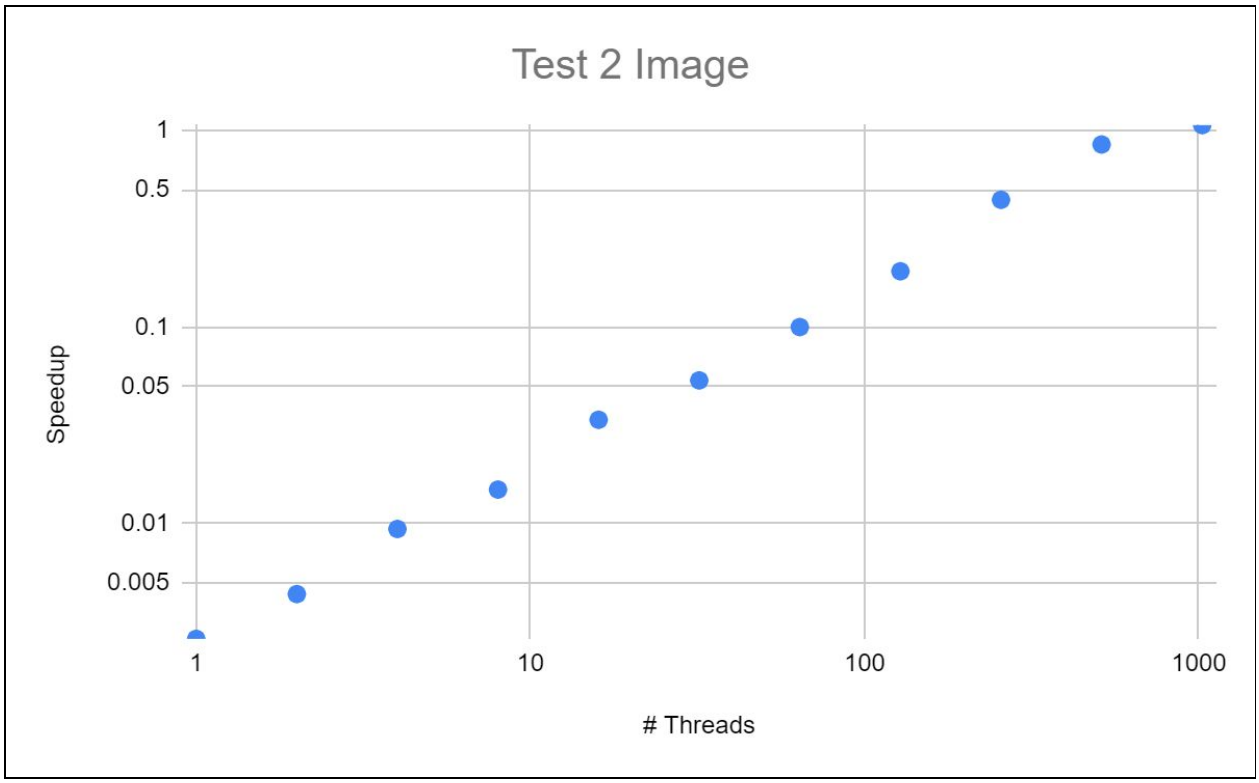



*Figure 2: Speedup vs Threadcount for rectifying Test 2 Image*

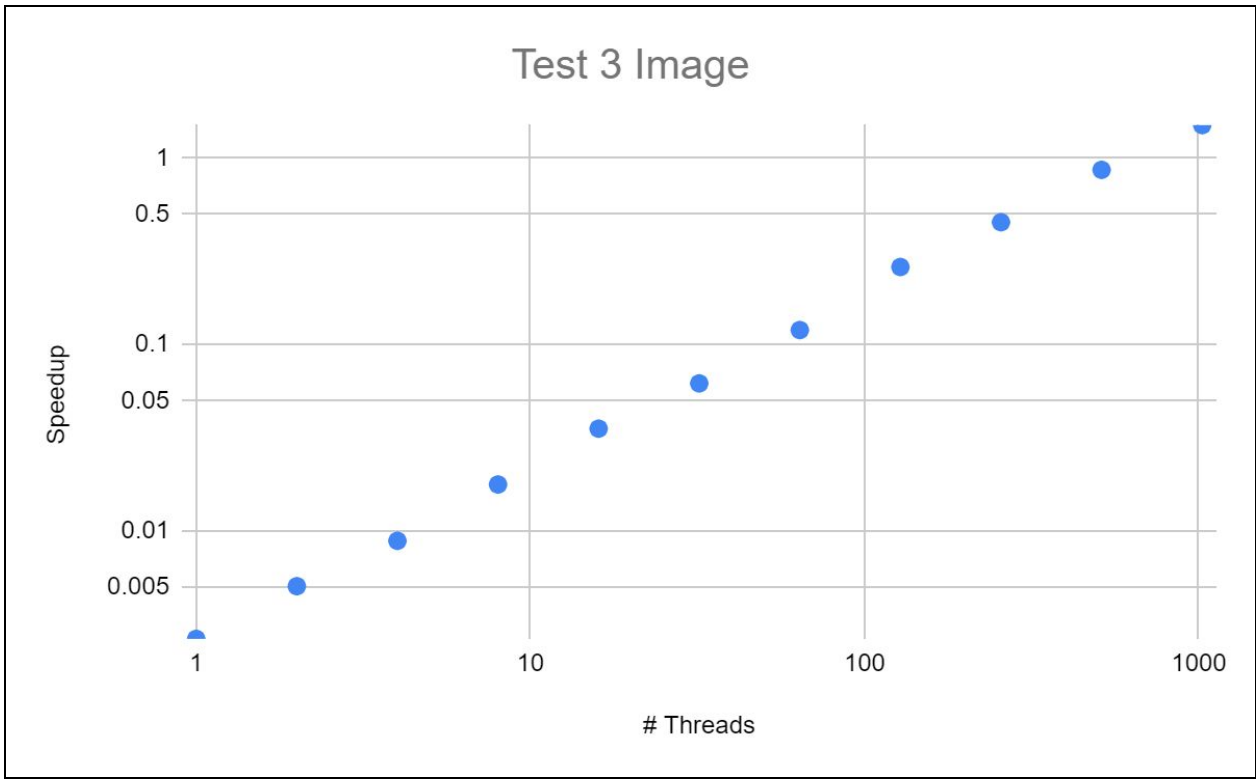



*Figure 3: Speedup vs Threadcount for rectifying Test 3 Image*

## Analysis and Discussion

If we were to use the sequential approach we would simply iterate through each pixel in the image, compare it to 127 and adjust the pixel value accordingly. With the parallel approach we get to use multiple threads that allow us to equally divide the pixels in the image in chunks among the threads so that each thread can perform the operation in parallel. Each thread iterates through every index, performs the operation and then updates the image array accordingly.

As we can see in Figures 1–3 above, there is a roughly linear relation between thread-count and speedup. From that we can infer that doubling the number of threads roughly doubles the speedup.

However, we also observe that, for fewer than 1024 threads, the parallelized operation has a speedup less than 1. This is likely because of the overhead associated with operating the GPU and the lower clock rate of the GPU (as compared to the CPU). This suggests that parallelizing with the GPU is ideal when it is possible to use a large number of threads.

# Max Pooling

## Objective

For this part of the lab, for a given input image, we were required to compare the pixel values in all 2x2 sectors of the image and then use the greatest pixel value from each sector to produce an output image and thereby shrinking the image to half its original size.

## Example

Before Max Pooling

After Max Pooling

## Observation

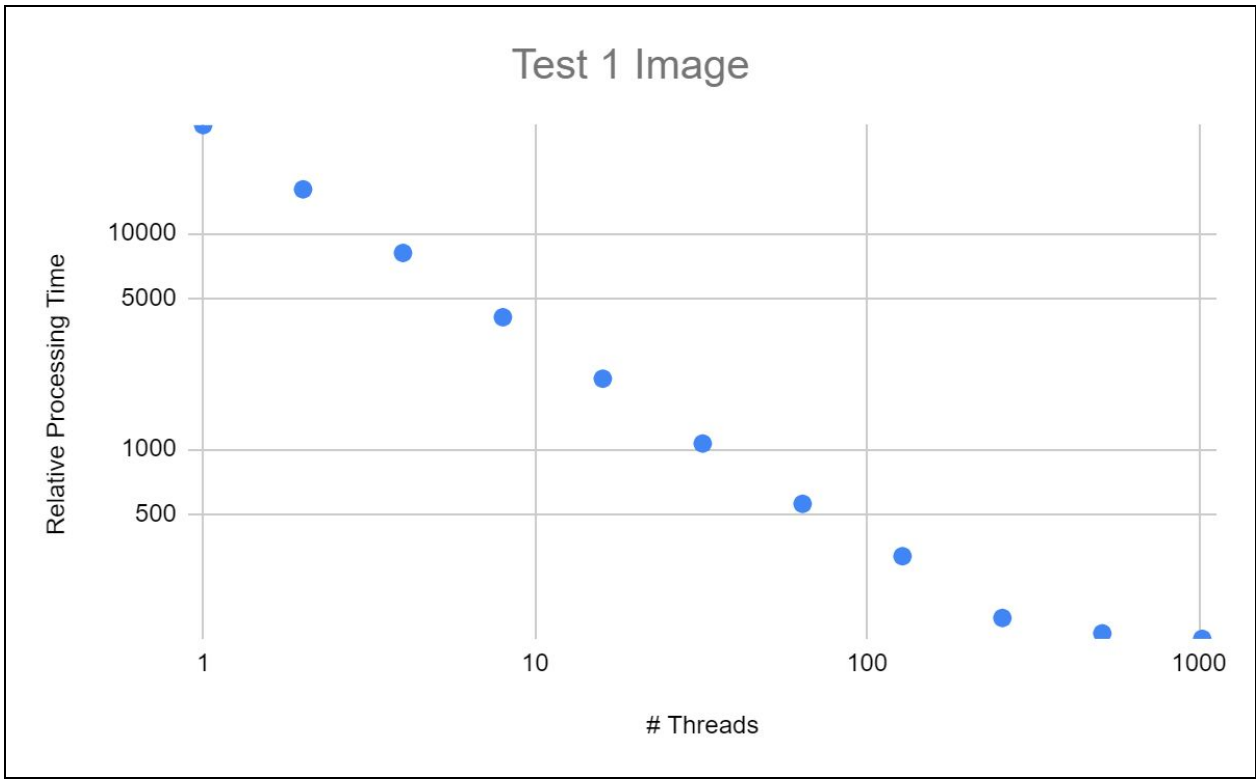



*Figure 4: Relative Processing Time vs Threadcount for max pooling Test 1 Image*

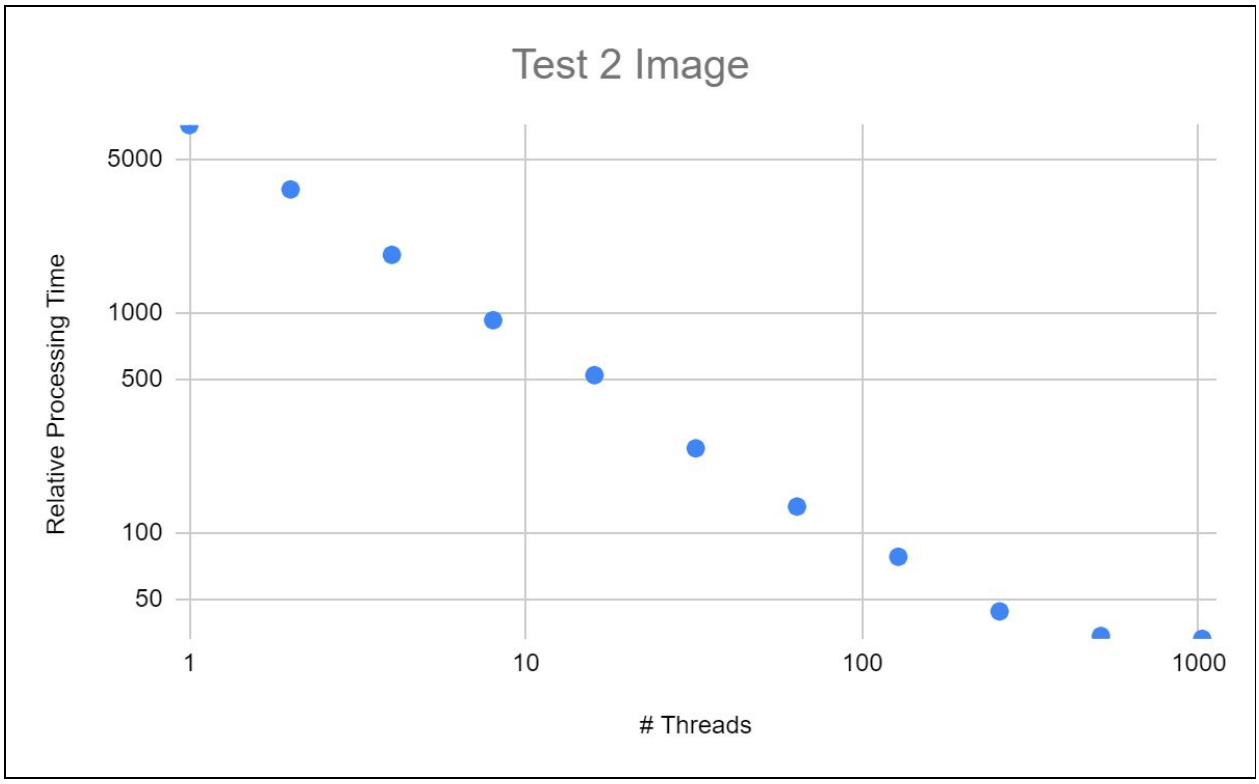



*Figure 5: Relative Processing Time vs Threadcount for max pooling Test 2 Image*

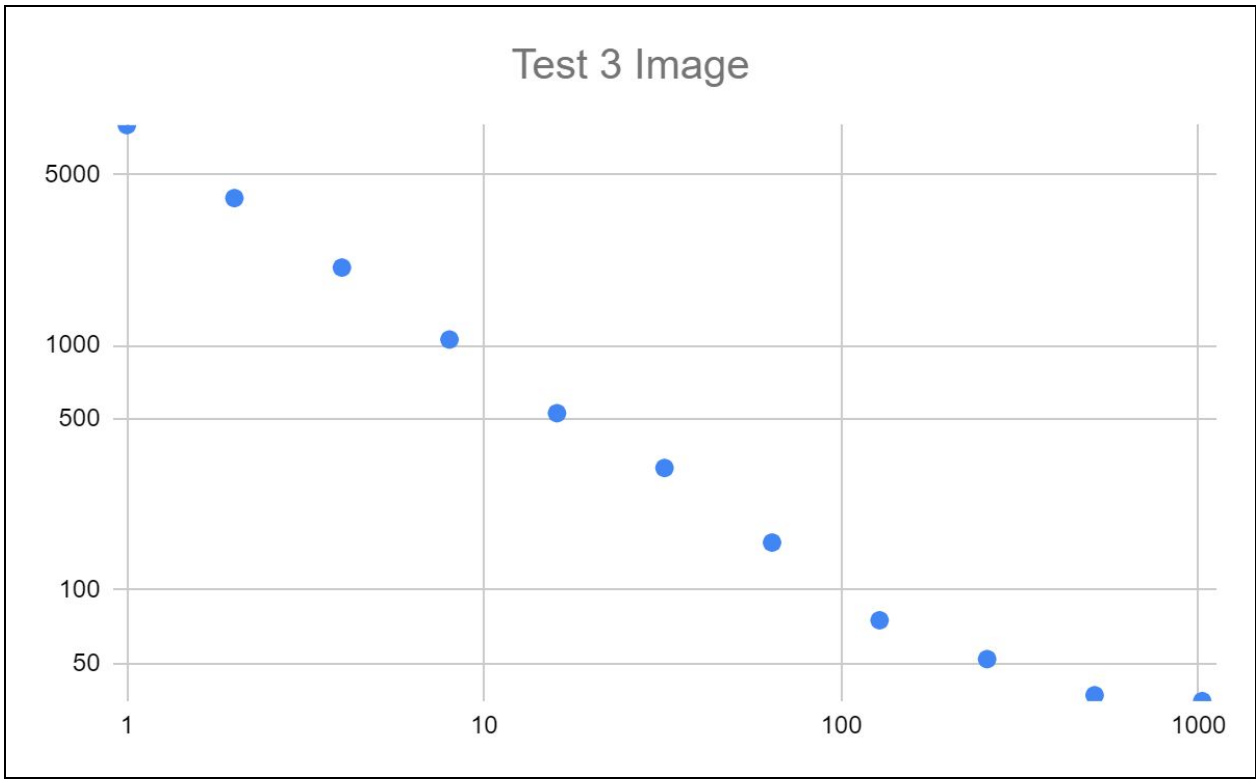



*Figure 6: Relative Processing Time vs Threadcount for max pooling Test 3 Image*

## Analysis and Discussion

If we were to approach maximum pooling sequentially, we would iterate through 2x2 sectors of the image array, comparing the four pixel values in that sub-matrix and picking the highest value from them. To implement the parallel approach the image is divided into sections of equal size and then each of these sections will be executed by a thread each. The main challenge was to parse the 1D image array into a 2D matrix in order to access the 4 corners of the 2x2 sectors which then are used to translate these 2D indices back into the index of the 1D image array.

As we can see in Figures 4–6 above, there is a roughly inverse linear relation between thread-count and speedup. From that we can infer that doubling the number of threads roughly halves the relative processing time.

However, we also see that, beyond 256 threads, that linearity ends, and we observe diminishing returns to greater threadcounts. It appears to be approaching a limit. This would suggest that rectifying would likely have a limit as well, although evidently not reached within 1024 threads.

# Appendix

*Figure 7: Rectified Test 1 Image*

*Figure 8: Pooled Test 1 Image*

*Figure 9: Rectified Test 2 Image*

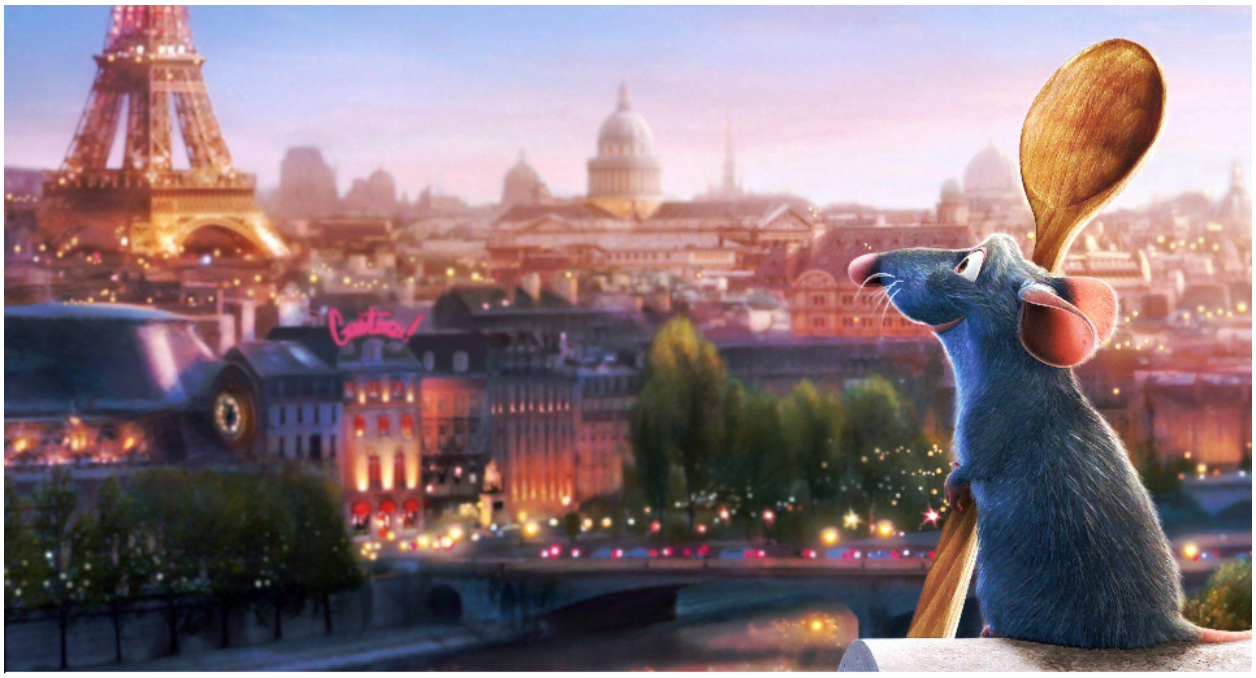


*Figure 10: Pooled Test 2 Image*

*Figure 11: Rectified Test 3 Image*

*Figure 12: Pooled Test 3 Image*