Comparing Runtimes of GPU and CPU with Image Denoising

•••

What is Image Denoising

Image denoising is a process in which noise (unwanted variations or artifacts) is removed or reduced from an image to improve its visual quality.

Noise can be introduced during image acquisition, transmission, or processing, and it often manifests as random variations in pixel intensity.

The goal of image denoising is to preserve the important features of the image while minimizing the impact of noise.

What is salt and pepper noise

Salt-and-pepper noise is a type of image noise characterized by the appearance of randomly occurring bright and dark pixels, resembling salt and pepper sprinkled on an image.

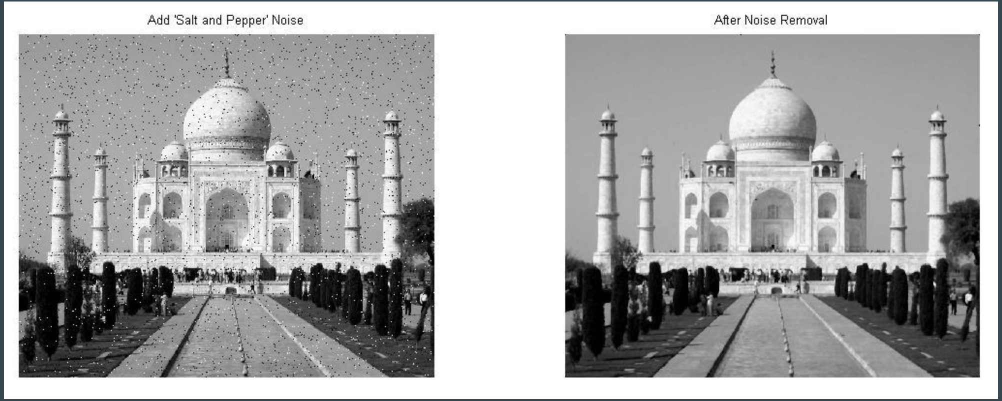
The bright pixels represent "salt," while the dark pixels represent "pepper." This type of noise can significantly degrade the visual quality of an image.

How can we remove salt & pepper noise?

Removing salt-and-pepper noise from images often involves the application of filtering techniques designed to suppress or replace the extreme pixel values caused by the noise.

One commonly used method is **Median Filtering**.

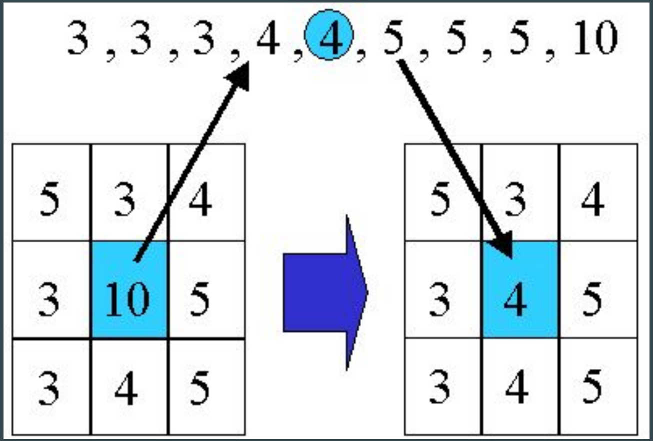
The median filter is a non-linear digital filtering technique, often used to remove noise from an image or signal



How Median Filter Work?

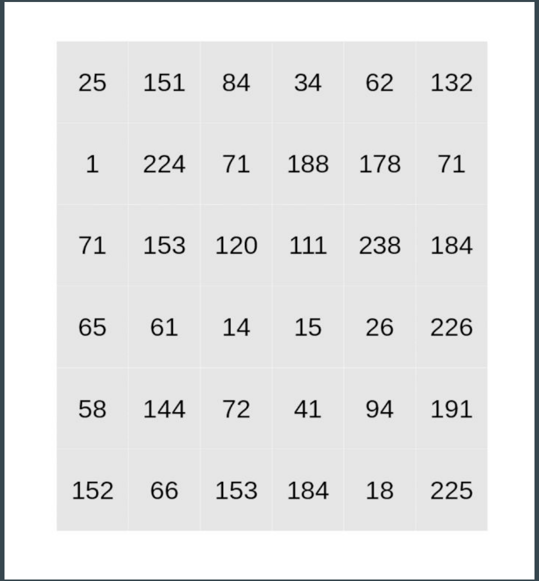
Median filtering involves sorting the pixel values within a defined neighborhood or kernel around each pixel in the image.

The middle value of the sorted list is then chosen as the new value for the central pixel, effectively reducing the impact of outliers or extreme values.



How Median Filter Work?

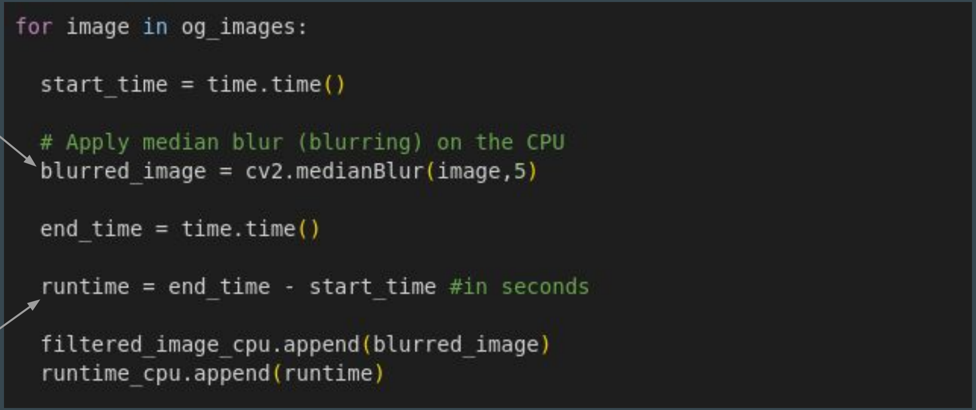
By replacing each pixel with its median value, median filtering is particularly effective at reducing noise in an image, making it a popular technique for image denoising applications.



25	151	84	34	62	132
1	224	71	188	178	71
71	153	120	111	238	184
65	61	14	15	26	226
58	144	72	41	94	191
152	66	153	184	18	225

CPU Implementation

Utilizes OpenCV's medianBlur function to apply a blurring operation with a kernel size of 5 to each original image in the collection

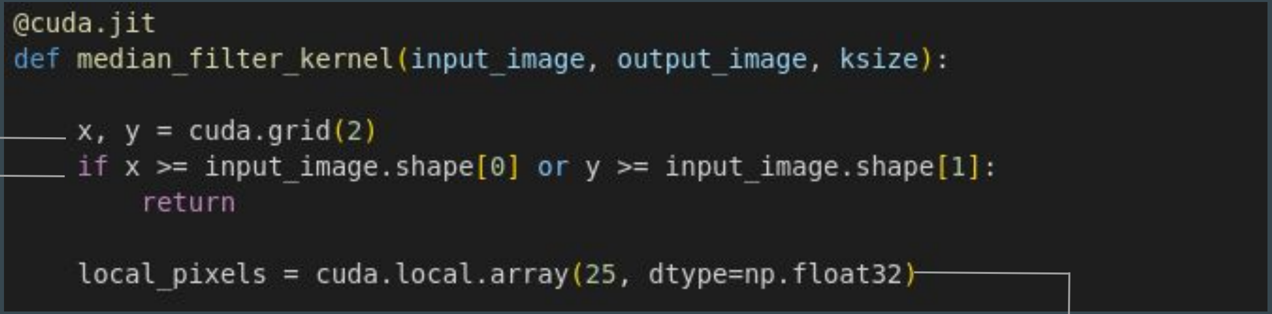


```
for image in og_images:

  start_time = time.time()

  # Apply median blur (blurring) on the CPU
  blurred_image = cv2.medianBlur(image,5)

  end_time = time.time()

  runtime = end_time - start_time #in seconds

  filtered_image_cpu.append(blurred_image)
  runtime_cpu.append(runtime)
```

Measures the time taken for the median blur operation on each image using time.time(), recording the start and end times and calculating the runtime in seconds

GPU Implementation - Cuda Kernel Code

Calculates the thread's absolute position within the 2D grid of threads.
x and y represent the thread's coordinates



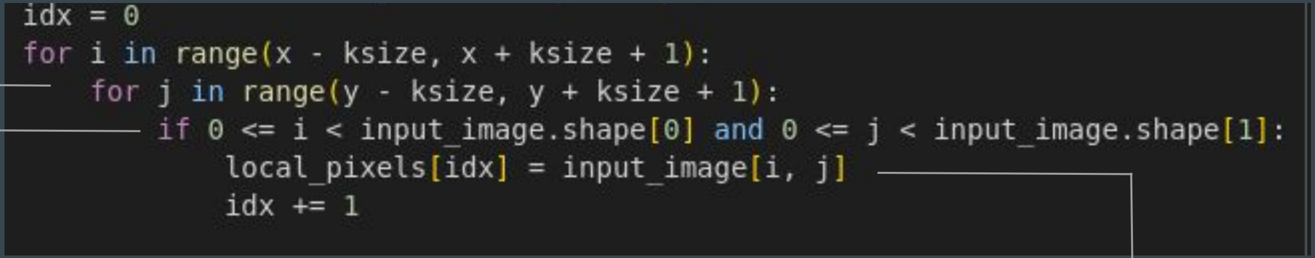
```
@cuda.jit
def median_filter_kernel(input_image, output_image, ksize):

    x, y = cuda.grid(2)
    if x >= input_image.shape[0] or y >= input_image.shape[1]:
        return

    local_pixels = cuda.local.array(25, dtype=np.float32)
```

Ensures that the thread coordinates (x and y) are within the dimensions of the input image. If outside, the thread returns without processing

This is a small array, residing in shared memory on the GPU, used to store pixel values from the local neighborhood.

GPU Implementation - Cuda Kernel Code

Iterate over the local neighborhood of the current thread's position within the image grid.



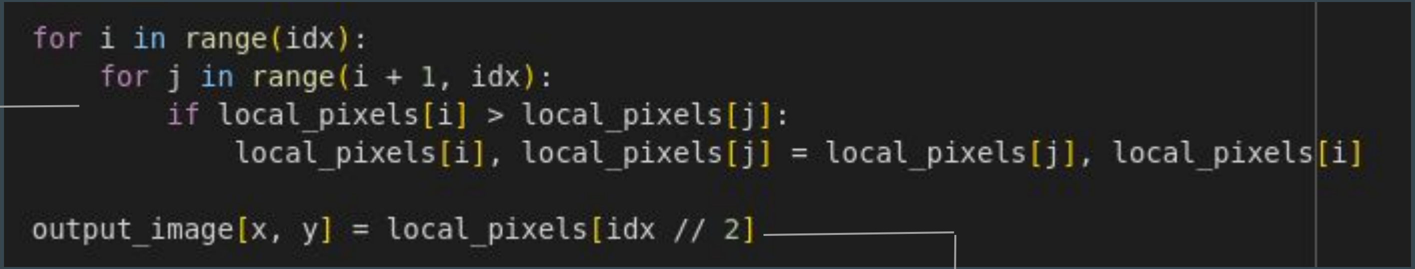
```
idx = 0
for i in range(x - ksize, x + ksize + 1):
    for j in range(y - ksize, y + ksize + 1):
        if 0 <= i < input_image.shape[0] and 0 <= j < input_image.shape[1]:
            local_pixels[idx] = input_image[i, j]
            idx += 1
```

check if the current indices (i and j) are within the bounds of the input image

copies the pixel value at position (i, j) from the input image to the local_pixels array

GPU Implementation - Cuda Kernel Code

Using bubble sort to sort the array in ascending order

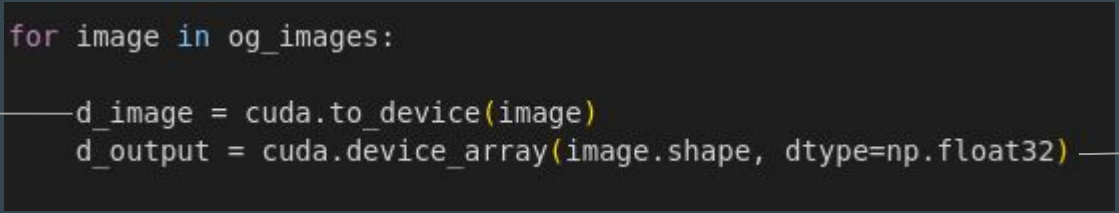


```python
for i in range(idx):
    for j in range(i + 1, idx):
        if local_pixels[i] > local_pixels[j]:
            local_pixels[i], local_pixels[j] = local_pixels[j], local_pixels[i]

output_image[x, y] = local_pixels[idx // 2]
```

Obtain the middle index of the sorted array, which corresponds to the median value

GPU Implementation - Applying Median Filter

Transfers the original image (image) from the host (CPU) to the device (GPU) memory using CUDA

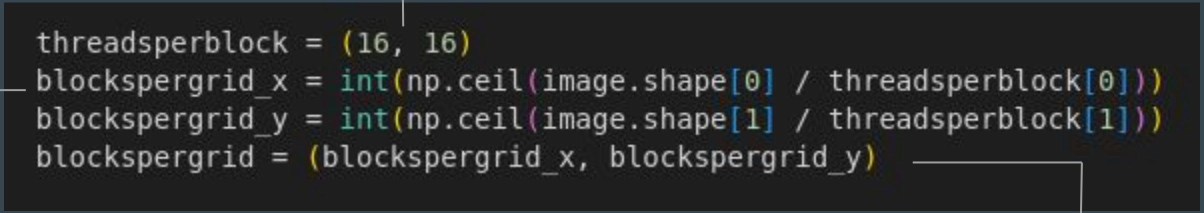


```
for image in og_images:

    d_image = cuda.to_device(image)
    d_output = cuda.device_array(image.shape, dtype=np.float32)
```

Allocates device memory for the output image (d_output) on the GPU. The output image will have the same shape as the input image

GPU Implementation - Applying Median Filter

Specifies the number of threads per block in a 2D configuration.



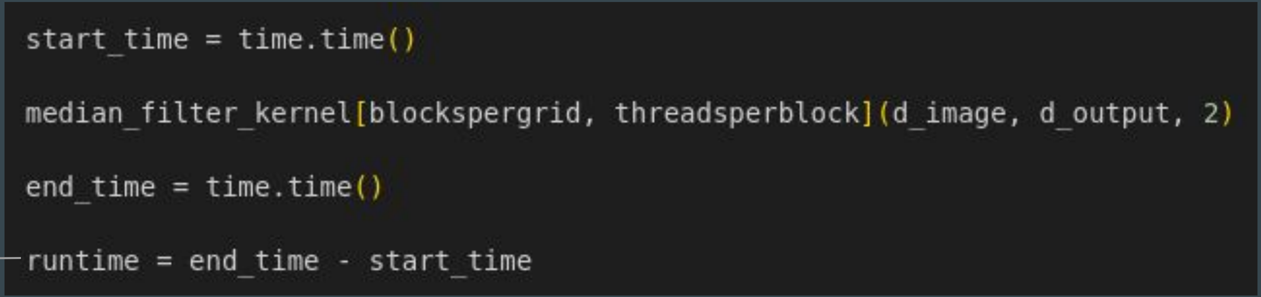
```
threadsperblock = (16, 16)
blockspergrid_x = int(np.ceil(image.shape[0] / threadsperblock[0]))
blockspergrid_y = int(np.ceil(image.shape[1] / threadsperblock[1]))
blockspergrid = (blockspergrid_x, blockspergrid_y)
```

calculate the number of blocks needed along the x-axis to cover the entire height of the image.
the width (number of columns) of the image along the y-axis.

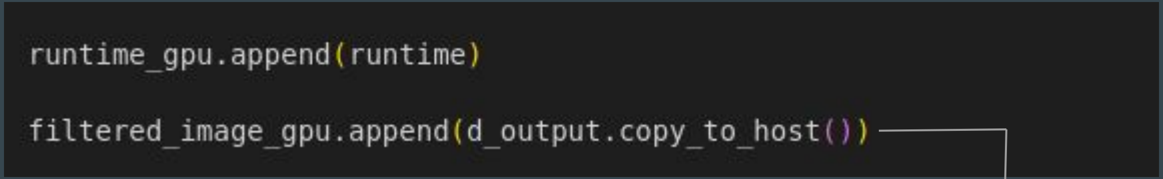
the total number of blocks needed in the 2D grid.

GPU Implementation - Applying Median Filter

Calculates the runtime by subtracting the start time from the end time.



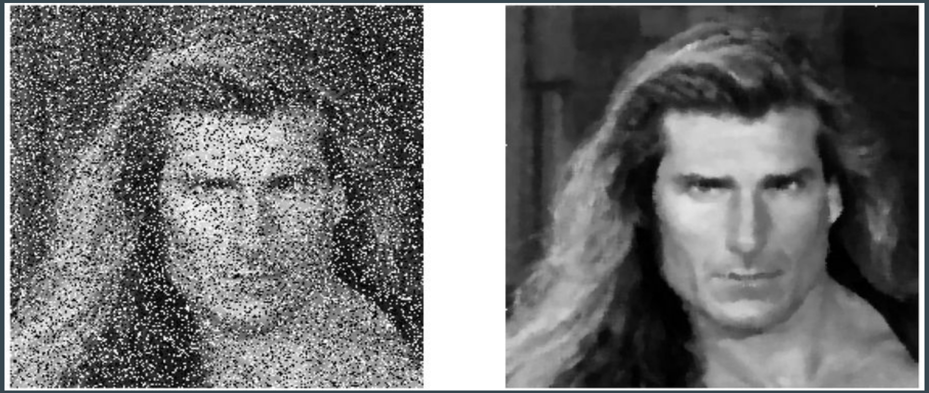
```
start_time = time.time()

median_filter_kernel[blockspergrid, threadsperblock](d_image, d_output, 2)

end_time = time.time()

runtime = end_time - start_time
```

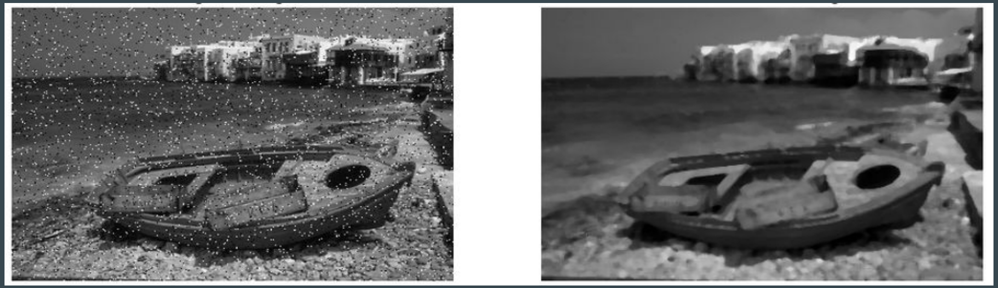


```
runtime_gpu.append(runtime)

filtered_image_gpu.append(d_output.copy_to_host())
```

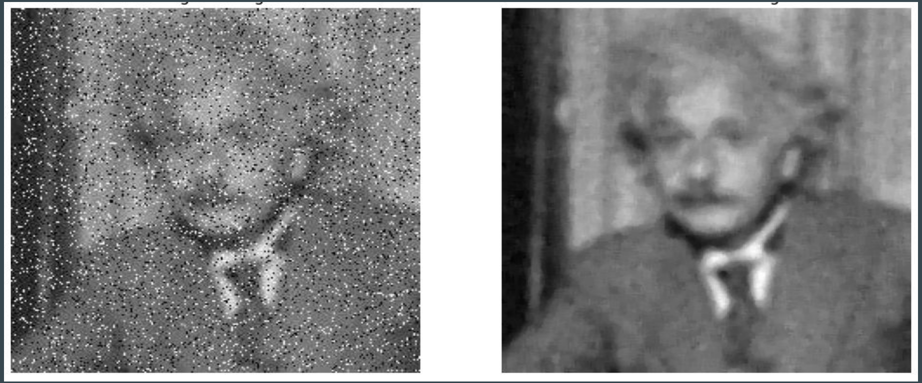
Copies the filtered image back from the device to the host

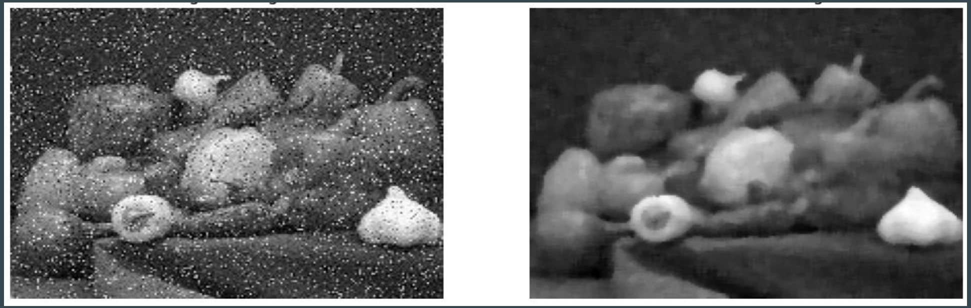
Results after applying the filter



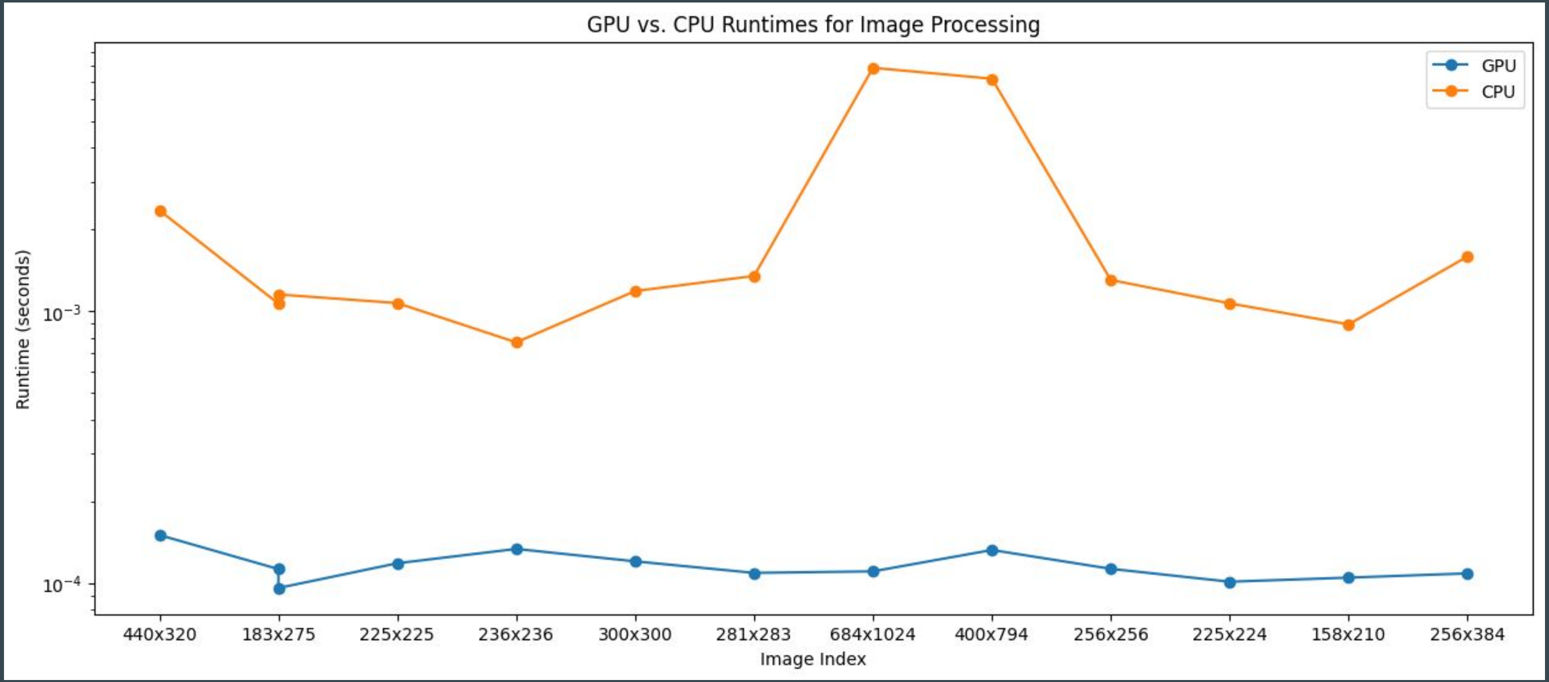


Results after applying the filter

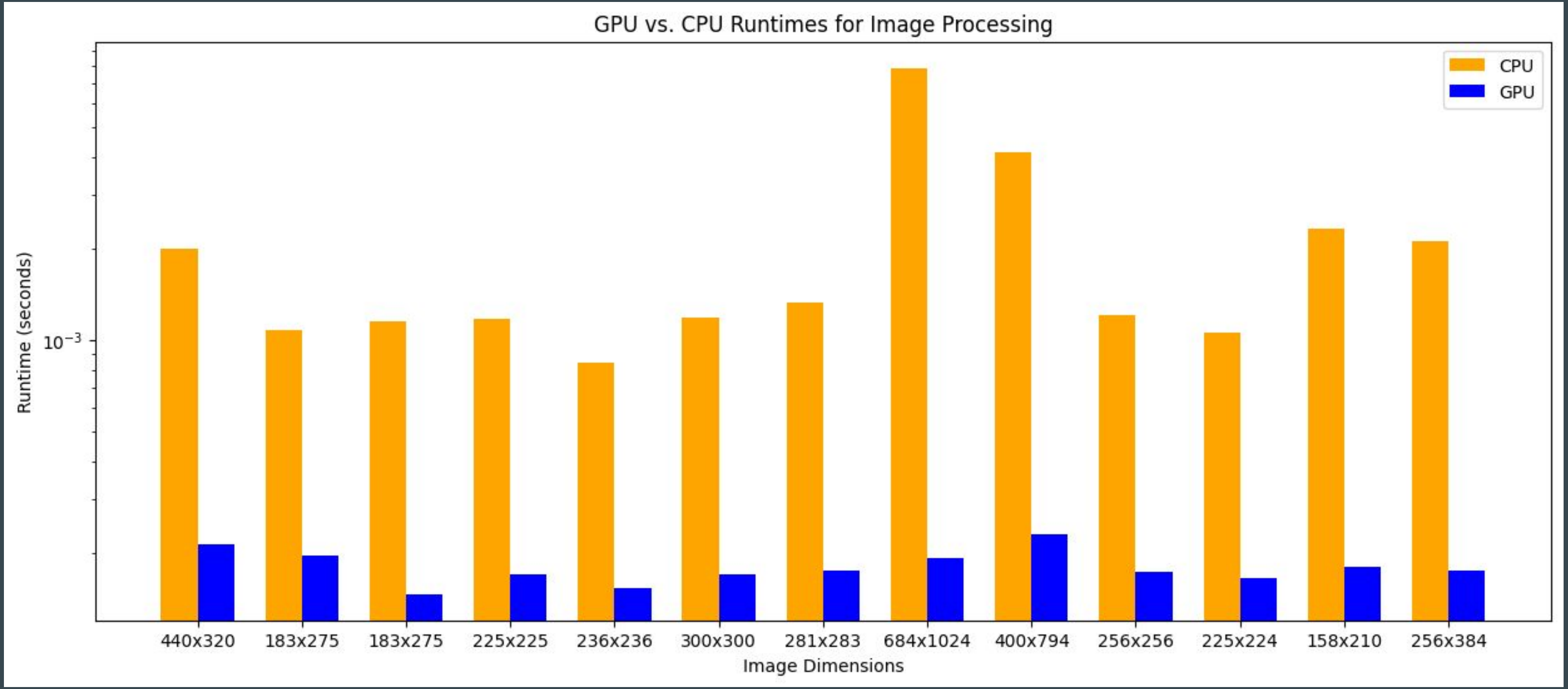




RunTime Analysis - Graphs



RunTime Analysis - Graphs


GPU vs. CPU Runtimes for Image Processing
CPU
GPU
Runtime (seconds)
10^{-3}
440x320
183x275
183x275
225x225
236x236
300x300
281x283
684x1024
400x794
256x256
225x224
158x210
256x384
Image Dimensions