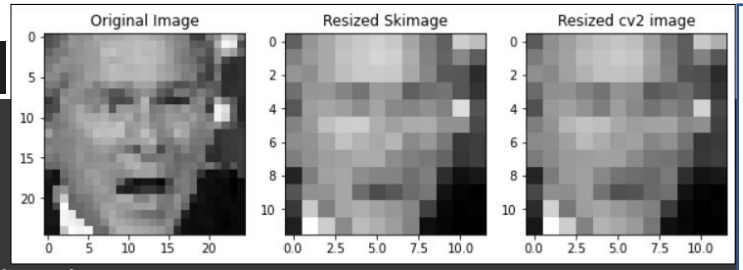
Task 1 :

```
%lprun -f res_cvimage res_cvimage(imgs)
```




```
Timer unit: 1e-09 s

Total time: 0.00209092 s
File: <ipython-input-9-93f758906d52>
Function: res_cvimage at line 5

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     5                                           def res_cvimage(imgs):
     6         1      13325.0  13325.0      0.6    new_size = (imgs[0].shape[0]//2, imgs[1].shape[1]//2)
     7         1        347.0    347.0      0.0    res_im = []
     8       200     101751.0    508.8      4.9    for im in imgs:
     9       200    1731026.0   8655.1     82.8      image_resized = cv2.resize(im, new_size, interpolation =
cv2.INTER_AREA)
    10       200      89674.0    448.4      4.3      res_im.append(image_resized)
    11         1     154794.0 154794.0      7.4    return np.asarray(res_im)
```

In **task 1**, I have used the function cv2.resize() which increases the speed of the code and takes less time that is 82.8%.
As cv2.resize is a function from OpenCV library written in C++, which is much faster than Python.
Also OpenCV utilize multiple CPU cores to process images which improves the performance of cv2.resize() function.

Task 2 :

MULTI PROCESSING

```
%%timeit
def approximate_pi_multiprocessing(nums):
    def worker_process(queue, n):
        pi_2 = 1
        nom, den = 2.0, 1.0
        for i in range(n):
            pi_2 *= nom / den
            if i % 2:nom += 2
            else:den += 2
        queue.put(2*pi_2)
    queue = multiprocessing.Queue()
    processes = []
    for n in nums:
        process = multiprocessing.Process(target=worker_process,
                                          args=(queue, n))
        processes.append(process)
        process.start()

    for process in processes:
      process.join()

    pi_values = []
    while not queue.empty():
      pi_values.append(queue.get())
    return pi_values
```

```
63.1 ns ± 9.28 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

MULTI THREADING

```
%%timeit
def approximate_pi_multithread(nums):
    def worker_thread(queue, n):
        pi_2 = 1
        nom, den = 2.0, 1.0
        for i in range(n):
            pi_2 *= nom / den
            if i % 2:
                nom += 2
            else:
                den += 2
        queue.put(2*pi_2)

    queue = queue.Queue()
    threads = []
    for n in nums:
        thread = threading.Thread(target=worker_thread, args=(queue, n))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

    pi_values = []
    while not queue.empty():
        pi_values.append(queue.get())

    return pi_values
```

```
72.5 ns ± 1.12 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

In **task 2**, the Multi Processing is faster than Multi Threading because it is a CPU bound task which is running in parallel for calculating the approximate pi. Here, multiprocessing has multiple CPU cores when compared to multithreading which is having a single core.

Task 3 :

MULTI THREADING

```
%%timeit
def load_array_threading(filenames):
  def worker_thread(filename, data):
    data.append((filename, np.load(filename)))

  data = []
  threads = []

  for filename in filenames:
    thread = threading.Thread(target=worker_thread, args=(filename, data))
    threads.append(thread)
    thread.start()

  for thread in threads:
    thread.join()

  return data
```

```
74.8 ns ± 1.39 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

MULTI PROCESSING

```
%%timeit
def load_array(filenames):
    with concurrent.futures.ProcessPoolExecutor() as executor:
        futures = {executor.submit(np.load, filename):filename for filename in filenames}
        result = {}

        for future in concurrent.futures.as_completed(futures):
          filename = futures[future]
          if future.exception() is None:
            result[filename] = future.result()
          else:
            print(f'{filename} generated an exception: {future.exception()}')
    return result
```

```
70.5 ns ± 0.599 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

In **task 3**, the Multi Processing is faster than Multi threading because concurrent.futures library provides a higher level interface for creating and managing threads which simplifies the code.

But, generally, for I/O bound tasks, using multiple threads is faster.

NAME: ABHIK SARKAR
MATRIKELNUMMER: 23149662
IDM Id : lo88xide