Performance Measurement and Improvement Techniques

Goal

In image processing, since you are dealing with a large number of operations per second, it is mandatory that your code is not only providing the correct solution, but that it is also providing it in the fastest manner. So in this chapter, you will learn:

- To measure the performance of your code.
- Some tips to improve the performance of your code.
- You will see these functions: **cv.getTickCount**, **cv.getTickFrequency**, etc.

Apart from OpenCV, Python also provides a module **time** which is helpful in measuring the time of execution. Another module **profile** helps to get a detailed report on the code, like how much time each function in the code took, how many times the function was called, etc. But, if you are using IPython, all these features are integrated in an user-friendly manner. We will see some important ones, and for more details, check links in the **Additional Resources** section.

Measuring Performance with OpenCV

The **cv.getTickCount** function returns the number of clock-cycles after a reference event (like the moment the machine was switched ON) to the moment this function is called. So if you call it before and after the function execution, you get the number of clock-cycles used to execute a function.

The **cv.getTickFrequency** function returns the frequency of clock-cycles, or the number of clock-cycles per second. So to find the time of execution in seconds, you can do following:

```
e1 = cv.getTickCount()
# your code execution
e2 = cv.getTickCount()
time = (e2 - e1)/ cv.getTickFrequency()
```

We will demonstrate with following example. The following example applies median filtering with kernels of odd sizes ranging from 5 to 49. (Don't worry about what the result will look like - that is not our goal):

```
img1 = cv.imread('messi5.jpg')

e1 = cv.getTickCount()
for i in range(5,49,2):
    img1 = cv.medianBlur(img1,i)
e2 = cv.getTickCount()
t = (e2 - e1)/cv.getTickFrequency()
print( t )

# Result I got is 0.521107655 seconds
```

> **Note**
> You can do the same thing with the time module. Instead of **cv.getTickCount**, use the time.time() function. Then take the difference of the two times.

Default Optimization in OpenCV

Many of the OpenCV functions are optimized using SSE2, AVX, etc. It contains the unoptimized code also. So if our system support these features, we should exploit them (almost all modern day processors support them). It is enabled by default while compiling. So OpenCV runs the optimized code if it is enabled, otherwise it runs the unoptimized code. You can use **cv.useOptimized()** to check if it is enabled/disabled and **cv.setUseOptimized()** to enable/disable it. Let's see a simple example.

```
# check if optimization is enabled
In [5]: cv.useOptimized()
Out[5]: True

In [6]: %timeit res = cv.medianBlur(img,49)
10 loops, best of 3: 34.9 ms per loop

# Disable it
In [7]: cv.setUseOptimized(False)

In [8]: cv.useOptimized()
Out[8]: False

In [9]: %timeit res = cv.medianBlur(img,49)
10 loops, best of 3: 64.1 ms per loop
```

As you can see, optimized median filtering is2x faster than the unoptimized version. If you check its source, you can see that median filtering is SIMD optimized. So you can use this to enable optimization at the top of your code (remember it is enabled by default).

Measuring Performance in IPython

Sometimes you may need to compare the performance of two similar operations. IPython gives you a magic command timeit to perform this. It runs the code several times to get more accurate results. Once again, it is suitable to measuring single lines of code.

For example, do you know which of the following addition operations is better, x = 5; y = x**2, x = 5; y = x*x, x = np.uint8([5]); y = x*x, or y = np.square(x)? We will find out with timeit in the IPython shell.

```
In [10]: x = 5

In [11]: %timeit y=x**2
10000000 loops, best of 3: 73 ns per loop

In [12]: %timeit y=x*x
10000000 loops, best of 3: 58.3 ns per loop

In [15]: z = np.uint8([5])

In [17]: %timeit y=z*z
1000000 loops, best of 3: 1.25 us per loop

In [19]: %timeit y=np.square(z)
1000000 loops, best of 3: 1.16 us per loop
```

You can see that, x = 5 ; y = x*x is fastest and it is around 20x faster compared to Numpy. If you consider the array creation also, it may reach up to 100x faster. Cool, right? *(Numpy devs are working on this issue)*

Note
> Python scalar operations are faster than Numpy scalar operations. So for operations including one or two elements, Python scalar is better than Numpy arrays. Numpy has the advantage when the size of the array is a little bit bigger.

We will try one more example. This time, we will compare the performance of **cv.countNonZero()** and **np.count_nonzero()** for the same image.

```
In [35]: %timeit z = cv.countNonZero(img)
100000 loops, best of 3: 15.8 us per loop

In [36]: %timeit z = np.count_nonzero(img)
1000 loops, best of 3: 370 us per loop
```

See, the OpenCV function is nearly 25x faster than the Numpy function.

Note
> Normally, OpenCV functions are faster than Numpy functions. So for same operation, OpenCV functions are preferred. But, there can be exceptions, especially when Numpy works with views instead of copies.

More IPython magic commands

There are several other magic commands to measure performance, profiling, line profiling, memory measurement, and etc. They all are well documented. So only links to those docs are provided here. Interested readers are recommended to try them out.

Performance Optimization Techniques

There are several techniques and coding methods to exploit maximum performance of Python and Numpy. Only relevant ones are noted here and links are given to important sources. The main thing to be noted here is, first try to implement the algorithm in a simple manner. Once it is working, profile it, find the bottlenecks, and optimize them.

1. Avoid using loops in Python as much as possible, especially double/triple loops etc. They are inherently slow.
2. Vectorize the algorithm/code to the maximum extent possible, because Numpy and OpenCV are optimized for vector operations.
3. Exploit the cache coherence.
4. Never make copies of an array unless it is necessary. Try to use views instead. Array copying is a costly operation.

If your code is still slow after doing all of these operations, or if the use of large loops is inevitable, use additional libraries like Cython to make it faster.

Additional Resources

1. Python Optimization Techniques
2. Scipy Lecture Notes - Advanced Numpy
3. Timing and Profiling in IPython

Exercises