Final Project: Image Processing Optimization

Course: Advanced Python
Prepared by: Ksenia Saenko (ks4841), Stella Sun (ss8955), James Guo (xg626)
Term: Spring 2018

1. PROBLEM DESCRIPTION

Image processing is painfully slow because we need to access each pixel, therefore, we have to include a lot of "for loops" and "if else" statement in the code. For almost every step in Image Processing and Pattern Recognition, pixel manipulation is an essential part of the entire process.

However, due to the growing dimensionationally of images, the inefficiency of traversing images in Python is becoming the bottleneck of the program. More and more default operations are implemented in machine learning libraries like scikit-learn to avoid the high time penalty of the running time. Meanwhile, except for common utility functions, sometimes we still need to write some customized functions.

To demonstrate how an image processing code can be optimized, we selected a code designed to identify Parallelograms in an Image. The code did not use any built-in functions and relied on many for-loops for processing. It took around 90 seconds to process a single image, causing concern about the scalability and processing multiple images. Our goal was to improve performance of the code by at least 20% in terms of the time it takes to run.

2. METHODOLOGY

All code was written in Python 3 using Jupyter Notebooks and we identified two key parts for improving the code:

Part 1: Improving the code in Python:

- Stage 1: Reduce for-loops and function calls, apply built-in functions, redundancy control,etc.
- Stage 2: Implement with Numpy/vectorizing
- Stage 3: Implement with Itertools
- Stage 4: Implement with Cython
- Stage 5: Implement with Numba

Part 2: Implementing the code with other tools:

- Implement the code in Julia
- Parallelize the code with Pyspark

Each function in the code iterated through the stages listed above. The progress for each function was tracked and recorded. (See Fig.2) The best performing version was included in the final code.

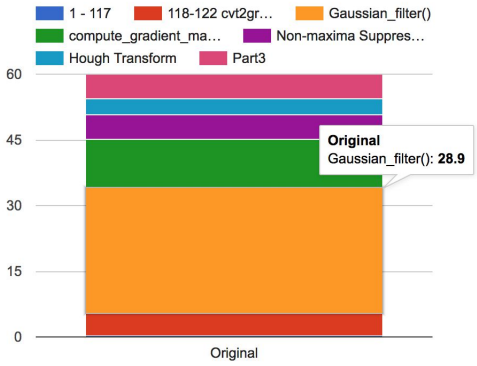
Fig.2 Sample of progress record

		Update on Timing					
Function	**Original Timing**	**Update 1**	**Method 1**	**Update 2**	**Method 2**	**Update 3**	**Method 3**
cvt2grayscale	5.264 s	0.019 s	Numpy vectorizing	0.253 s	Numba	6.793 s	Itertools
smooth_image_with_Gaussian_filter	27.494 s	3.533 s	Numpy vectorizing	0.040 s	Numpy built-in function	0.066 s	Itertools
compute_gradient_magnitude_and_gradient_angle	11.2s	10.1s	itertools	4.06 s	Numba		Cython
quantize_angle_of_the_gradient_to_four_sectors	464ns	460ns	itertools	-	(Numba makes it slower)	9000ns	Julia
Step3 (line 228 - 244)	2min 43s	2min 40s	itertools				
Lines 176-200	6.854 s	5.241 s	Numpy vectorizing	4.871 s	Itertools	6.953 s	Numba

2.0 Profiling original code to identify bottlenecks

In order to get a sense of the running time distribution and identify potential performance bottlenecks, we did a line-by-line profiling with the line-profiler before we get started.

Fig.1 line-profiling of the original code



According to the result of line-profiling, we were able to catch a glimpse of the detailed running time distribution and reveal the top three functions that consumes over 70% of the total running time:

1. Smooth_image_with_Gaussian_filter (41.3%)
2. Compute_gradient_magnitude_and_gradient_angle (21.9%)
3. Cvt2grayscale (6.0%)

Thus the main goal of our work is to speed up the procedure and eliminate the performance bottlenecks.

2.1. Improving the code in Python

Packages and Functions: Pandas, Itertools, Numpy, Cython, Numba, Scipy

- Build-in functions

Gaussian filter is a common preprocessing procedure to smooth the image in Computer Vision. Fortunately, the built-in functions, Scipy "ndimage.gaussian_filter" function optimized the code performance the most.

- Itertools

In particular, Itertools "Product" was used to replace double for-loops.

- Cython

We optimized code to run Cython by defining each variable type. Still, we have not observed any performance improvement with this particular approach.

- Numba

Optimizing code with Numba almost consistently resulted in improvement.

- Customized refactoring

Except for the large functions, some minor code snippets could contribute to the efficiency slowdown. By refactoring the implementation and utilizing efficient mechanism in Python, we were able to tackle it and avoid the expensive operations.

The matrix initialization could serve as a good example. For instance, the original implementation tends to initialize a matrix to zeros and then assign designated values to it through for-loops. After closer scrutiny, we refactor the

code by simply initializing it to 1 and then assign designated values to it through broadcasting, which bring the running time of 244 ms down to 2.38 ms and speed it up to 100 times.

```
edge_map = np.zeros((row, col), dtype='uint8')
# Initialize to edge map to 255
for i in range(0, row):
    for j in range(0, col):
        edge_map[i][j] = 255
```

```
edge_map = np.ones((row, col), dtype='uint8')
# Initialize to edge map to 255
edge_map *= 255
```

As the pattern can be identified several time in the codebase, we believe it is nontrivial to conduct code refactoring like this.

2.2 Implementing the code in Julia

Packages and Functions: Julia, PyCall, @pyimport

We learnt Julia in class, so it was a good chance to try it out and see if Julia can beat Python in some functions. In Julia, there is a PyCall query that could call any python types, like numpy.array, integers, etc., and @pyimport could import python functions such as math.sqrt. I wrote a two-for-loops function in Julia and import the original python code to compare the runtime, and Julia returned a much better result:

```
d = Dict()
d["Julia"] = minimum(julia_bench.times) / 1e6
d["Python"] = minimum(py_bench.times) / 1e6
d
```

```
Dict{Any,Any} with 2 entries:
  "Julia"  => 0.0244
  "Python" => 4.29684
```

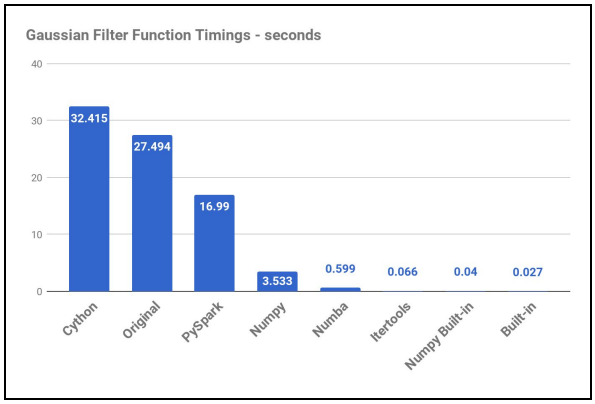
One thing that needs to take consideration was that, the reason why Julia was faster could be that I called python code, and compare two functions in Julia, if I tested the other way around (call Julia in Python framework), what would be the result? We didn't do this test in this project, but it could be a new experiment in the future.

2.3 Parallelizing the code with PySpark

Packages and Functions: PySpark

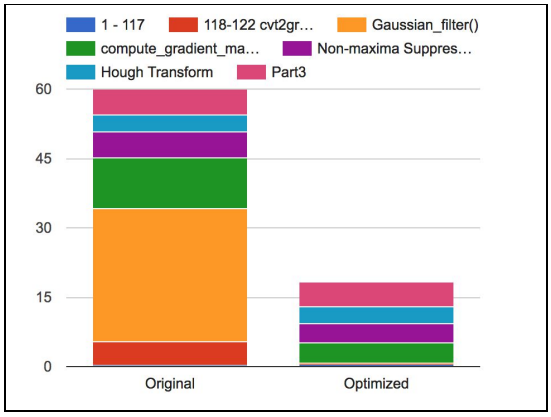
To test code performance in Spark, we implemented two functions - Gaussian Filter and converting image to a grey scale. PySpark resulted in a faster code that the original implementation. However, it was slower than all other methods except Cython (See Fig.3). Considering that we only processed a single image and many of the operations are sequential, we think that PySpark would be useful for processing multiple images as then the performance could be parallelized.

Fig.3 Example of methods applied to Gaussian Filter Function



3. RESULTS

Fig.4 Comparison between the original code and the optimized one



Compared with the original code, the optimized version runs nearly 5 times faster. As shown in Fig.4, The previous overheads, such as functions cvt2grayscale() and Gaussian-filter(), etc. have been almost eliminated. The running time of other parts have also been optimized more or less.

4. CONCLUSION

- Python packages, and built-in functions

We were not surprisingly to find out that, built-in function was the best choice if we can find one, but at the same time, testing the result to make sure that built-in function returned the exact same result was important.

We also found out that, although Cython was faster in some functions, but it didn't always work out well, so it was better to test all possible approaches and chose the optimal solution.

- Other languages

There were so many languages nowadays, and picking the right language was very important. As we all assumed that PySpark would be the solution, however, it didn't make our code faster. The explanation might be, PySpark was ideal for dealing with big data, or parallel-processing, but our code was only based on one input, and all functions were running in order; therefore, PySpark didn't improve the speed, however, if in the future, the code could be used to process hundreds or thousands of pictures at the same time, PySpark could be a good choice.

5. REFERENCES

1) Jupyter notebooks from Advanced Python class
2) Fast, optimized for pixel loops with OpenCV and Python. (2017, August 27). Retrieved from https://www.pyimagesearch.com/2017/08/28/fast-optimized-for-pixel-loops-with-opencv-and-python/