CMSC 33710 Scientific Visualization Project Report: Python HLL

Lynn Zheng

March 15, 2021

1 Introduction and Motivation

I implemented the ray marching procedures for volume rendering in Python, based on the C project we completed in class. I was interested in whether Python (and NumPy) will be able to handle the heavy computation demanded by convolution and whether it is still possible to gain efficiency via multithreading, subject to Python's GIL (Global Interpreter Lock).

2 Implementation Details

2.1 Kernel `eval` vs. `apply` in Convolution

I used NumPy operation broadcasting to implement the `apply` method of the kernel to batch-evaluate the kernel function at multiple indices. The results in this report are obtained using the call to `apply` instead of `eval`, as the latter is too slow for any practical use.

3 Experiments

3.1 Environment Setup

For uniformity, all of the results in this report are obtained running on UChicago's CSIL Slurm Peanut computing cluster. The machines on the `general` partitions have 16 cores and are able to support 16 threads.

3.2 Performance Comparison with Reference C Code

For the command below, the reference C code takes about **0.185 seconds** while the Python script takes about **2 minutes 19 seconds (139 seconds)**. The C program is about **750 times** faster than the Python program. The command generates a 50 by 50 image, using the Cubic B-Spline kernel.

```
[rrendr go | python go.py] -i cube.nrrd -fr 6 12 5 -at 0 0 0 -up 0 0 1 -nc -2.3 -fc
    2.3 -fov 14 -sz 50 50 -us 0.03 -s 0.03 -k bspln3 -p rgbalit -b over -lut lut.nrrd
    -lit rgb.txt -o cube-rgb.nrrd
```

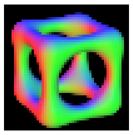

Figure 1: 50 by 50 pixel image generated by the command above

3.3 Performance Comparison using Threads

I tested the performance of threading using the command above. My Python program has a multithreading feature but doesn't experience any speedup due to the presence of the GIL, which I had flagged as a potential roadblock in my project proposal. For the C program, the speedup as a result of multithreading is significant, as shown in the figure below. There does seem to be diminishing returns to the increase in the number of threads, as more threads are contending for a single lock.

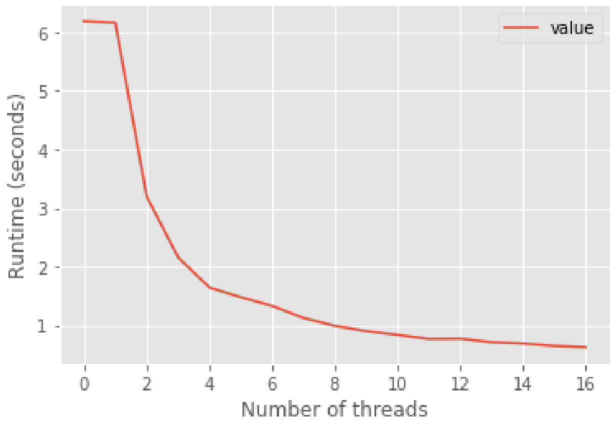

Figure 2: Speedup of C Multithreading

3.4 Performance Comparison on Output Resolution

For a fixed number of threads, I investigated how the number of pixels in the output image affects performance. I took the measurement by varying the side length of the output images (controlled by `-sz`), always keeping the image square. As my Python code is drastically slower than the C code, I had to run my Python code to output smaller images than the C code does. My hypothesis is that the scaling relationship between the output image side length and the runtime should be similar for both the Python and the C program. For instance, an image of side length 20 pixels (400 pixels total) should take roughly 4 times the runtime of an image of side length 10 pixels (100 pixels total). The table and plot (on a normalized scale) below verify my hypothesis.

Table 1: Comparison of runtime using different output image side length

Output length	Source	Runtime (seconds)
10	Python	32.600000
15	Python	71.800000
20	Python	130.100000
25	Python	200.300000
100	C	0.658161
150	C	1.478145
200	C	2.621693
250	C	4.112398

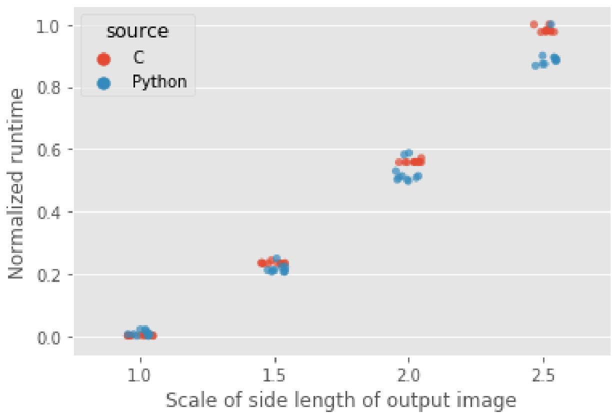

Figure 3: Scale comparison using normalized runtime and output side length

4 Conclusion

I was able to fully replicate the features of the C program in Python. Despite being a fun learning experience, the computation costs and lack of parallelism support of the Python program greatly reduces its practical usability. I have developed a fresh appreciation for fast, low-level languages in scientific computing while working on this project.

Apart from computation time, I also find it interesting to reflect on **developer time**: the time it takes for a developer to implement volume rendering in a low-level language like C versus a high-level one like Python. Python wasn't a time saver for me though I am much more proficient in Python than I am in C. The entirety of the C code (including threading) took me about 20 hours to write. The C macros provided translates easily into NumPy operations in Python. Despite that, translating the C code into Python and debugging took me about 15 additional hours. This is quite significant time considering that I am not implementing any new algorithm (or digesting any new class material and translating math formulas into code) during the translation. One factor that deterred my development speed was that I didn't have comprehensive unit tests for classes like the kernel or the convolution, and had to switch back to debugging these two multiple times during development.