



Oxford Internet Institute, University of Oxford

## Assignment Cover Sheet

Candidate Number	1039636
Assignment	Data Analytics at Scale
Term	Michalemas Term 2019
Title/Question	Data Analytics at Scale Report
Word Count	3477

**By placing a tick in this box ☒ I hereby certify as follows:**

- (a) This thesis or coursework is entirely my own work, except where acknowledgments of other sources are given. I also confirm that this coursework has not been submitted, wholly or substantially, to another examination at this or any other University or educational institution;
- (b) I have read and understood the Education Committee's information and guidance on academic good practice and plagiarism at <https://www.ox.ac.uk/students/academic/guidance/skills?wssl=1>.
- (c) I agree that my work may be checked for plagiarism using Turnitin software and have read the Notice to Candidates which can be seen at: <http://www.admin.ox.ac.uk/proctors/turnitin2w.shtml>, and that I agree to my work being screened and used as explained in that Notice;
- (d) I have clearly indicated (with appropriate references) the presence of all material I have paraphrased, quoted or used from other sources, including any diagrams, charts, tables or graphs.
- (e) I have acknowledged appropriately any assistance I have received in addition to that provided by my [tutor/supervisor/adviser].
- (f) I have not sought assistance from a professional agency;
- (g) I understand that any false claims for this work will be reported to the Proctors and may be penalized in accordance with the University regulations.

**Please remember:**

- To attach a second relevant cover sheet if you have a disability such as dyslexia or dyspraxia. These are available from the Higher Degrees Office, but the Disability Advisory Service will be able to guide you.

# Data Analytics at Scale Report

1039636

## 1 Introduction and Overview

This report follows the general structure below.

- Plan for analysis of the FINd algorithm:
  - The first profiling will be done using ‘timeit’ in a Jupyter notebook over the whole algorithm, to time it running over a sample of runs of image hashing;
  - The second profiling will be done using ‘prun’ in a Jupyter notebook over the whole algorithm, to determine which portion of the image hashing algorithm is taking the majority of the time;
  - The third profiling involves the use of ‘line profiler’ in the Anaconda prompt, to go line-by-line in the portions of the image hashing algorithm that are slowest, and determine which should be a focus of code optimization;
  - The fourth profiling involves memory profiling, to determine the algorithm’s use of system memory during the image hashing;
- The second part of this report involves the testing of various optimisation strategies, discussion, and comparison.
  - Relevant portions of the code identified in the profiling are optimised using various faster python tools;
  - The algorithm is run using multi-processing techniques to allow multiple CPUs to compute images concurrently
  - The algorithm is tested on alternative hardware such as GPUs using free services provided by Google Colab.
- The final part of this report involves comparisons with other algorithms for speed and accuracy of image hashing.

## 2 Profiling

### 2.1 Plan for Profiling

We set up profiling to be standardised across different files and calls from notebooks. The initial code targeted for optimisation, provided in `FINd.py`, has two key functions added for processing. The first is `read_images_from_file`, which reads all the relevant hashing images from the file into a list of filenames, and calls a random subset of these filenames, with the amount given by a parameter for the number of images requested. The second function is `benchmarking_basic`, which calls `read_images_from_file`, passes the number of images specified by the users, receives the list of filenames, and returns the hashes.

This pair of functions adds some slight overhead with the python OS module, but as the filenames are only read once, this overhead does not scale with the number of images we request to hash, so does not effect or benchmarking results. With these infrastructure set up, we can complete the profiling outlined above.

### 2.2 Profiling Initial Code for RunTime

Note that code and outputs are provided in the attached Jupyter Notebook file.

The original provided set of code was profiled first to determine various parameters about the algorithm. This section briefly discusses the speed of the algorithm, and how this speed it scales with the number of images. We then discuss various levels of detailed code profiling, including at the function level, and then for relevant functions we wish to examine, at the individual line level.

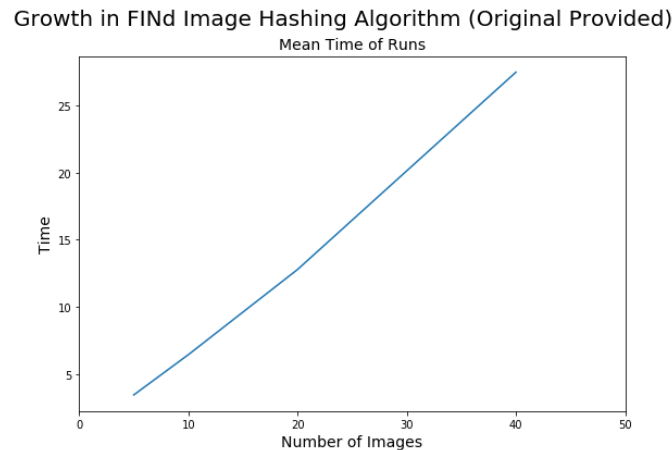
The first profile was to use the in-built `%timeit` magic in Jupyter notebook to run a series of timing profiles. The function was looped through 5, 10, 20 and 40 image calls, to get a sense of time, and to see the form of the function's growth in time in terms of the number of images.

On the hardware used in this testing (a 2016 Surface Pro), the code runs at an average of around 0.7 seconds per image. This is obviously something we would wish to improve, as this implies a long run time for 50,000 images. The variation in these runs is relatively small, around 3.5 per cent per loop. This is consistent with our expectation, as the function is performing relatively basic arithmetic op-

erations that should not vary in complexity meaningfully with different image inputs.

The growth in the code is clearly linear. Doubling the inputs resulted in a doubling of run time, which is apparent in the graph below, which plots the mean time of the 4 runs of the loop above.

Figure 1: First RunTime Profiling



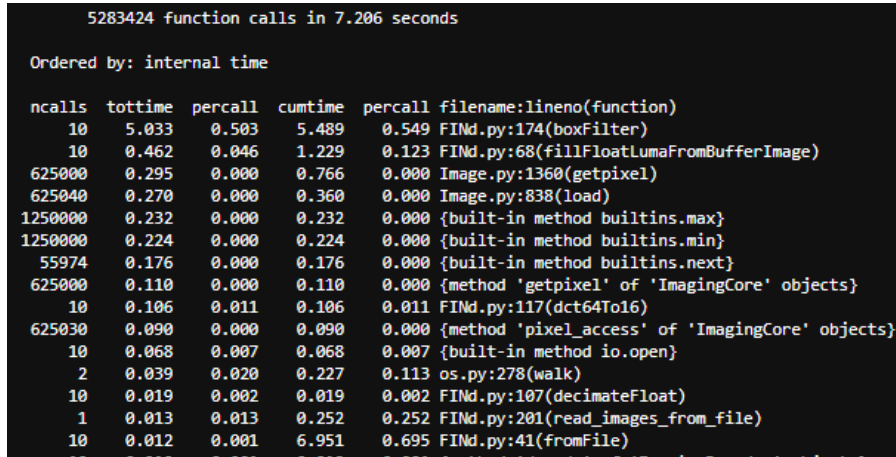
## 2.3 Detailed Function RunTime Profiling

To investigate further, we begin detailed profiling. We used the `%prun` Jupyter notebook magic examine the function calls over a run of 10 images through the `benchmarking_basic` function. The full output is available in the notebook, and a relevant snippet of the output is shown in Figure 2 below. Clearly we can see the function `BoxFilter` is taking the most of amount of time, followed by `FillFloatLumafrom-BufferImage`.

Some back-of-the-envelope maths from these results (and similar runs) suggests that `boxfilter` takes around 60-70% of the time for the execution, and `FillFloatLuma` another 5%. `BoxFilter` is an obvious candidate for optimisation, with `FillFloatLuma` behind that as well.

Helpfully, this snippet also demonstrates that the overhead created by the `benchmarking_basic` function does not appear to be relevant and does not appear to scale with the number of images, giving us confidence that our benchmarking results are not being effected by this implementation.

Figure 2: First Function RunTime Profiling



```

5283424 function calls in 7.206 seconds

Ordered by: internal time

ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
    10    5.033    0.503    5.489    0.549  FINd.py:174(boxFilter)
     10    0.462    0.046    1.229    0.123  FINd.py:68(fillFloatLumaFromBufferImage)
  625000    0.295    0.000    0.766    0.000  Image.py:1360(getpixel)
  625040    0.270    0.000    0.360    0.000  Image.py:838(load)
1250000    0.232    0.000    0.232    0.000  {built-in method builtins.max}
1250000    0.224    0.000    0.224    0.000  {built-in method builtins.min}
   55974    0.176    0.000    0.176    0.000  {built-in method builtins.next}
  625000    0.110    0.000    0.110    0.000  {method 'getpixel' of 'ImagingCore' objects}
     10    0.106    0.011    0.106    0.011  FINd.py:117(dct64To16)
  625030    0.090    0.000    0.090    0.000  {method 'pixel_access' of 'ImagingCore' objects}
     10    0.068    0.007    0.068    0.007  {built-in method io.open}
       2    0.039    0.020    0.227    0.113  os.py:278(walk)
     10    0.019    0.002    0.019    0.002  FINd.py:107(decimateFloat)
       1    0.013    0.013    0.252    0.252  FINd.py:201(read_images_from_file)
     10    0.012    0.001    6.951    0.695  FINd.py:41(fromFile)

```

## 2.4 Detailed Line-Level RunTime Profiling

The functions `BoxFilter` and `FillFloatLumafromBufferImage` are the majority of the time the code requires to run. As such, it is useful to investigate how these functions are performing at the individual lines of code. This allows us to potentially focus any code optimisations we implement to the relevant processes or lines of code, if we choose to go down that route.

We use the line profiler extension, implemented in the Python command line environment to produce a report detailing the line-by-line times for each of these functions. The first, shown in Figure 3 below, is for `BoxFilter`. This output shows that a section of 4 nested loops in the function results in the innermost loop taking around 85% of the function execution time to implement (see section shown in red box).

The same exercise was conducted for `FillFloatLumafromBufferImage` (even though it takes up a much smaller share of the execution time relative to `BoxFilter`). The output of this exercise is shown in Figure 4. Similar to `BoxFilter`, the operations of the innermost loops of the function are taking around 90% of the execution time (lines 73 and 77).

The combined results of the runtime profiling suggests that the `FINd` algorithm is quite slow (around 0.7 seconds per image on the rather dated hardware used in this exercise), but provides the small mercy of being order  $O(n)$  in the number of images; that is, the time the algorithm takes grows linearly as the number of images grows

Figure 3: First Function RunTime Profiling

```
(base) Oxford\Summatives\das2019>python -m line_profiler FInd-Copy2.py.lprof
Timer unit: 1e-07 s

Total time: 3.49828 s
File: FInd-Copy2.py
Function: boxFilter at line 167
```

Line #	Hits	Time	Per Hit	% Time	Line Contents
167					@classmethod
168					@profile
169					def boxFilter(cls, input, output, rows, cols, rowWin, colWin):
170	1	36.0	36.0	0.0	halfColWin = int((colWin + 2) / 2) # 7->4, 8->5
171	1	11.0	11.0	0.0	halfRowWin = int((rowWin + 2) / 2)
172	251	1125.0	4.5	0.0	for i in range(0, rows):
173	62750	287043.0	4.6	0.8	for j in range(0, cols):
174	62500	297495.0	4.8	0.9	s=0
175	62500	506599.0	8.1	1.4	xmin=max(0, i-halfRowWin)
176	62500	483642.0	7.7	1.4	xmax=min(rows, i+halfRowWin)
177	62500	454348.0	7.3	1.3	ymin=max(0, j-halfColWin)
178	62500	463259.0	7.4	1.3	ymax=min(cols, j+halfColWin)
179	435250	2267920.0	5.2	6.5	for k in range(xmin, xmax):
180	2595831	13831928.0	5.3	39.5	for l in range(ymin, ymax):
181	2223081	15781700.0	7.1	45.1	s+=input[k*rows+l]
182	62500	607716.0	9.7	1.7	output[l*rows+j]=s/((xmax-xmin)*(ymax-ymin))

```
(base) Oxford\Summatives\das2019>
```

Figure 4: Second Function RunTime Profiling

```
(base) C:\Users\... \Desktop\Oxford\Summatives\das2019>python -m line_profiler FInd-Copy3.py.lprof
Timer unit: 1e-07 s

Total time: 0.287893 s
File: FInd-Copy3.py
Function: fillFloatLumaFromBufferImage at line 66
```

Line #	Hits	Time	Per Hit	% Time	Line Contents
66					@profile
67					def fillFloatLumaFromBufferImage(self, img, luma):
68	1	64.0	64.0	0.0	numCols, numRows = img.size
69	1	9286.0	9286.0	0.3	rgb_image = img.convert("RGB")
70	1	15.0	15.0	0.0	numCols, numRows = img.size
71	251	1084.0	4.3	0.0	for i in range(numRows):
72	62750	272130.0	4.3	9.5	for j in range(numCols):
73	62500	1964592.0	31.4	68.2	r, g, b = rgb_image.getpixel((j, i))
74					luma[i * numCols + j] = (
75					self.LUMA_FROM_R_COEFF * r
76					+ self.LUMA_FROM_G_COEFF * g
77	62500	631758.0	10.1	21.9	+ self.LUMA_FROM_B_COEFF * b
78					)

```
(base) C:\Users\... \Desktop\Oxford\Summatives\das2019>
```

linearly. This means that any reduction in the execution time of this function will scale directly with the total number of images we wish to hash (i.e. a 50% reduction in the time the algorithm takes to run one image will yield a 50% reduction in the total runtime of all images).

This work suggests that some optimisations of the BoxFilter function, and to a lesser extent the FillFloatLumafromBufferImage function, would yield some significant benefits. I note that there is little evidence that the function is limited by I/O with the data on disk, at least at this point.

## 2.5 Profiling Initial Code for Memory Use

Another potential bottleneck for algorithms is the use of the computers working memory, or RAM. In this case, if the algorithm's execution saturates the available working memory it has to operate, the execution time will degrade. This is because accessing files and objects held in RAM is numerous orders of magnitude faster than accessing files through more static memory types such as hard drives.

Memory profiling was conducted using the Jupyter notebook cell magic operator `%memit`, which showed that the peak amount of RAM used by the run of the algorithm was a little over 70 megabytes, with an increment of between 2 and 3 megabytes. Testing with multiple different numbers of images to hash shows that these numbers do not scale with the number of images the algorithm is hashing.

This is consistent with our expectations as image hashing is a serial process, that is, for a number of images we request to hash, it computes the hash of one image at a time, in sequences. Further, each individual image and the related objects (data arrays for computation) are relatively small in size. As such, a relatively small amount of RAM is used repeatedly for each individual image.

The size of the memory used in this processing is trivial for modern machines. As a result, we did not conduct further memory profiling work and do not consider memory optimisation a high priority. Nonetheless, it is likely that the approximate figure of 70 megabytes could be reduced somewhat if required, although RAM is cheap and plentiful enough that this might be excessive.

## 2.6 Profiling Conclusion

Taken together, these results show that the algorithm is clearly limited by both the speed (or efficiency) of the Python code, and (in some sense by extension) by the CPU that is being used to run it. Memory is clearly not a major issue.

One confirmation from profiling (which fits our expectations) is that this algorithm is repeated individual jobs that run in sequence, and thus would be susceptible to multiprocessing.

### 3 Optimisation

In this section, we consider three broad types of optimisation. The first is code optimisation, where we take the existing FIND algorithm and make minor adjustments to the functions identified above as being slow. The second is multiprocessing, where we take the sequential operations that are being conducted as each image is hashed in turn, and spread them over different CPUs to complete the operation more quickly. Finally, the third is hardware optimisation: using hardware that is more modern and better designed for efficient (and thus rapid) computation of mathematical operations.

### 4 Code optimisation

The first code optimisation we conduct is to use existing libraries for optimising code. In this case, we use the Python library Numba and apply it to BoxFilter. Numba is a compiler for functions that use array and numerical operations in Python, and it takes the existing Python code and converts it into optimised machine code. Since BoxFilter is a relatively simple numerical operation, it is an excellent fit for optimisation via the Numba library.

This (relatively simple) change was implemented, and an adjusted FINDHasher\_2 class and benchmarking\_2 function were executed. This function runs image hashes in just over 0.35 seconds, or around a 45% speed up. We then conducted Similar profiling to the list above. Figure 5 shows how this algorithm is much faster, and still  $O(n)$ .

After this implementation, we ran various profiling to investigate how effective this optimisation was. As expected, these results (like Figure 6) show that the speed up comes from much quicker run times for BoxFilter.

I also made an attempt to do some code line-level improvements to FillFloatLumaFrom-BufferImage. However, this code did execute but consistently failed my unit tests, and I ran out of time to continue investigating. I have attached the code but am not sure why this did not work.



Figure 5: Code Time with Numba Improvement

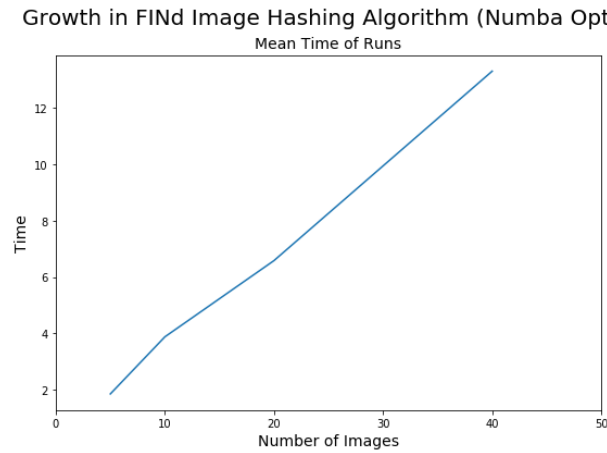


Figure 6: Profiling with Numba Improvement

```
11821595 function calls in 5.090 seconds
```

Ordered by: internal time

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
1153820	0.787	0.000	3.097	0.000	typeof.py:24(typeof)
1153820	0.622	0.000	1.827	0.000	functools.py:822(wrapper)
1153820	0.602	0.000	1.033	0.000	functools.py:768(dispatch)
10	0.391	0.039	1.073	0.107	FINd_numba.py:85(fillFloatLumaFromBufferImage)
1153820	0.340	0.000	0.340	0.000	weakref.py:395(__getitem__)
10	0.340	0.034	3.436	0.344	FINd_numba.py:14(faster_boxFilter)
1153820	0.277	0.000	0.483	0.000	<string>:1(__new__)
576900	0.267	0.000	0.679	0.000	Image.py:1360(getpixel)
576940	0.228	0.000	0.310	0.000	Image.py:838(load)
55974	0.214	0.000	0.214	0.000	{built-in method builtins.next}
1153820	0.205	0.000	0.205	0.000	{built-in method __new__ of type object at 0x00007FFB87F26BA0}
1153820	0.172	0.000	0.172	0.000	typeof.py:109(_typeof_bool)
10	0.110	0.011	0.110	0.011	FINd_numba.py:134(dct64To16)

## 5 Multiprocessing

The image hashing process is highly sequential. That is, the process occurs in a sequence, with one image hashed after the other. As many (most?) modern computers have multiple CPUs, we can split these sequential jobs into batches, and have two CPUs work on the jobs simultaneously.

With multiprocessing, the performance of the baseline algorithm was improved by around 40% when using 2 CPUs (the max available to my computer). We use the `.pool` method from multiprocessing, which creates two separate processes (one for each CPU). This method ensures that the processes do not share the same memory and thus cannot alter each others variables and cause incorrect outputs.

This optimisation resulted in a speed up of around 40% over the base algorithm for just two CPUs, as shown in Figure 5, resulting in an image hashing time of around 0.4 seconds per image.

Figure 7: Multiprocess Speedup

```
[9]: %%timeit -r5
test = benchmarking_basic(40)

28.3 s ± 772 ms per loop (mean ± std. dev. of 5 runs, 1 loop each)

[10]: %%timeit -r5
# split jobs
map_list = [20, 20]

with multiprocessing.Pool(2) as pool:
    test = pool.map(benchmarking_basic, map_list)

16.7 s ± 654 ms per loop (mean ± std. dev. of 5 runs, 1 loop each)
```

We also conducted this exercise in Google Colab (discussed more below), to see if more modern hardware can improve this optimisation from multiprocessing. Figure 6 shows that this speed up of almost 45%, resulting in an image hashing time of around 0.32 seconds.

Figure 8: Multiprocess Speedup on Google Colab

```
[11]: # Google CPU multiprocessing comparison - run on the Google CPU (only 1 process)
# This was run on Google Colab

%%timeit
test = benchmarking_basic_1(40)

1 loop, best of 3: 22.1 s per loop

[10]: # Multiprocessing on the Google CPU
# This was run on Google Colab

%%timeit
# split jobs
map_list = [20, 20]

with multiprocessing.Pool(2) as pool:
    test = pool.map(benchmarking_basic_1, map_list)

1 loop, best of 3: 12.5 s per loop
```

In the case of using Multiprocessing, unit tests are unlikely to be necessary; we are simply running the same benchmarking function for the original algorithm as above on two different CPUs (and the .pool method ensures that these processes do not

share memory, and so cannot influence the output of the other processes like they may do for multi-threading, etc).

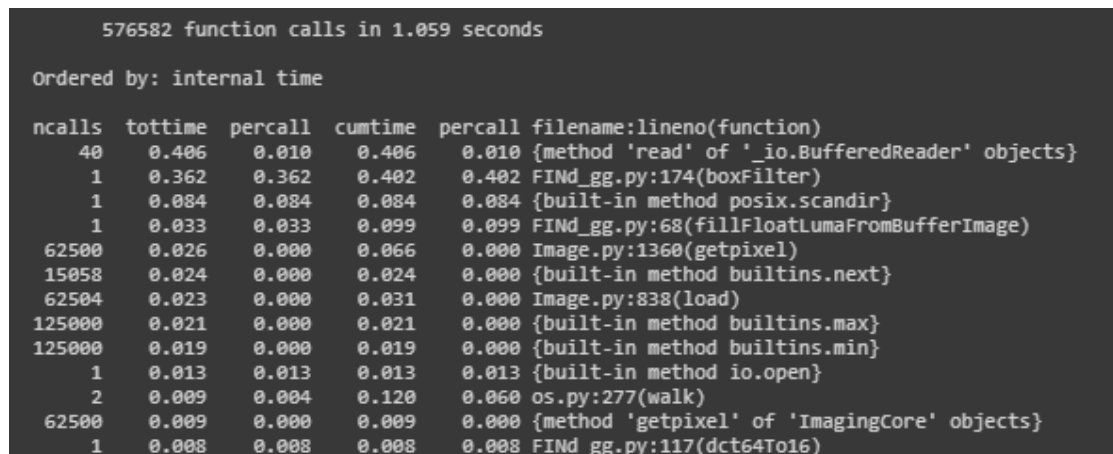
## 6 Hardware

Developments in modern computer science have allowed for the production of hardware that is suited to running specific operations. GPUs are an example of hardware with specific architecture designed to complete numerical operations more quickly than CPUs. As image hashing is mostly numerical computation, GPU acceleration of this code is a worthwhile optimisation to try.

While I do not have access to a personal GPU, Google Colab provides GPU hardware acceleration for free. I loaded a sample (around 15000) images onto Google Drive, and began benchmarking the original code on that platform. These notebook cells were moved to the Jupyter notebook attached with this assignment.

Google's default CPU hardware (without hardware acceleration) is slightly slower than the CPU execution on my rather dated 2016 Surface Pro (around 1 second per image in the benchmarking\_basic function). I suspected this was due to Google-Collab specific factors rather than Google using old hardware, and the output of %lprun on Google's hardware in Figure 7 suggests this is true.

Figure 9: Google Collab - Slow to Read From Disk



```
576582 function calls in 1.059 seconds
Ordered by: internal time
```

ncalls	tottime	percall	cumtime	filename:lineno(function)
40	0.406	0.010	0.406	0.010 {method 'read' of '_io.BufferedReader' objects}
1	0.362	0.362	0.402	0.402 FIND_gg.py:174(boxFilter)
1	0.084	0.084	0.084	0.084 {built-in method posix.scandir}
1	0.033	0.033	0.099	0.099 FIND_gg.py:68(fillFloatLumaFromBufferImage)
62500	0.026	0.000	0.066	0.000 Image.py:1360(getpixel)
15058	0.024	0.000	0.024	0.000 {built-in method builtins.next}
62504	0.023	0.000	0.031	0.000 Image.py:838(load)
125000	0.021	0.000	0.021	0.000 {built-in method builtins.max}
125000	0.019	0.000	0.019	0.000 {built-in method builtins.min}
1	0.013	0.013	0.013	0.013 {built-in method io.open}
2	0.009	0.004	0.120	0.060 os.py:277(walk)
62500	0.009	0.000	0.009	0.000 {method 'getpixel' of 'ImagingCore' objects}
1	0.008	0.008	0.008	0.008 FIND_gg.py:117(dct64To16)

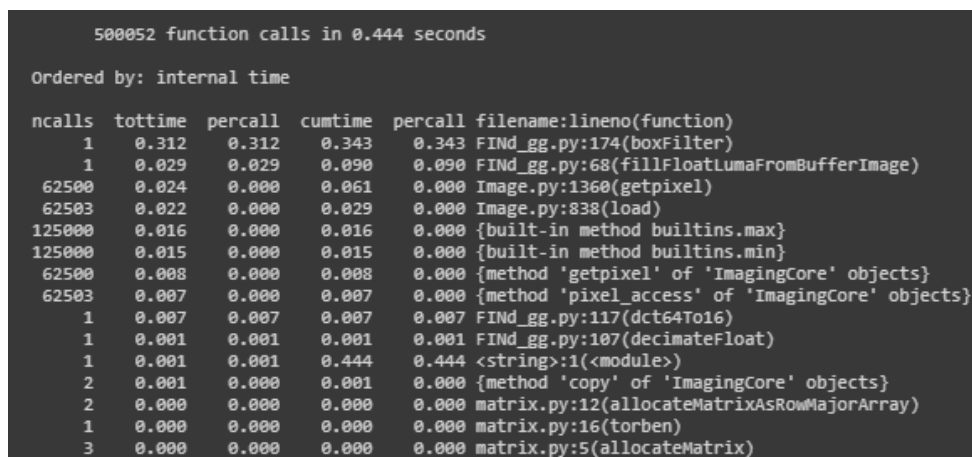
The top line of output here shows that the I/O process to read data is taking significant time, indeed, even more than the boxFilter. This suggests that the Python process is waiting a substantial amount of system time to load information from the Google Drive environment before being able to execute code. This was confirmed

through examining online discussion of this matter.

Notably though, and as seen in the attached Jupyter Notebook, the execution time of the GPU hardware was around 30 % faster than execution time on the CPU only hardware. To try to abstract from any IO issues from Google Drive, I then loaded a sample of images into Google Colab memory.

Running the same profiling as above shows that this solves the input/output problem, shown in Figure 10, where as with local execution of the algorithm, the majority of the execution time is taken up by BoxFilter.

Figure 10: Google Colab - Faster to Read from Memory



```

500052 function calls in 0.444 seconds

Ordered by: internal time

```

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
1	0.312	0.312	0.343	0.343	FINd_gg.py:174(boxFilter)
1	0.029	0.029	0.090	0.090	FINd_gg.py:68(fillFloatLumaFromBufferImage)
62500	0.024	0.000	0.061	0.000	Image.py:1360(getpixel)
62503	0.022	0.000	0.029	0.000	Image.py:838(load)
125000	0.016	0.000	0.016	0.000	{built-in method builtins.max}
125000	0.015	0.000	0.015	0.000	{built-in method builtins.min}
62500	0.008	0.000	0.008	0.000	{method 'getpixel' of 'ImagingCore' objects}
62503	0.007	0.000	0.007	0.000	{method 'pixel_access' of 'ImagingCore' objects}
1	0.007	0.007	0.007	0.007	FINd_gg.py:117(dct64To16)
1	0.001	0.001	0.001	0.001	FINd_gg.py:107(decimateFloat)
1	0.001	0.001	0.444	0.444	<string>:1(<module>)
2	0.001	0.000	0.001	0.000	{method 'copy' of 'ImagingCore' objects}
2	0.000	0.000	0.000	0.000	matrix.py:12(allocateMatrixAsRowMajorArray)
1	0.000	0.000	0.000	0.000	matrix.py:16(torben)
3	0.000	0.000	0.000	0.000	matrix.py:5(allocateMatrix)

In this case, the GPU code is around 15% faster, to around 0.38 seconds per image, as shown in Figure 11. This is likely a more consistent reflection of better performance, as loading the images into memory abstracts from the I/O process times from Google Drive, which in practice were highly variable.

## 6.1 Summary of Optimisations

Overall, each of these three optimisations strategies resulted individually in some success. The Numba optimisation improved the speed of this code by around 45%. The multiprocessing approach cut a given job of image hashes by around 45%, and using GPUs cut it by around 15%.

There are some weaknesses however. For example, Google Collab only allows us access to one GPU, and so we cannot combine the approaches of multiprocessing and our Numba-optimised code together. The Numba-optimised code ran on a GPU

Figure 11: Google Collab - Faster to Read from Memory

```

[12]: # To run on the default CPU hardware - 20 images - around 0.445ms per image
%%timeit
hash_list = []

for i in range(len(imgs_in_mem_list)):
    temphash = find.fromImage(imgs_in_mem_list[i])
    hash_list.append(temphash)
    temphash = []

1 loop, best of 3: 9.03 s per loop

[15]: # Run on the default GPU hardware - 20 images, around 0.382 seconds per image
%%timeit
hash_list = []

for i in range(len(imgs_in_mem_list)):
    temphash = find.fromImage(imgs_in_mem_list[i])
    hash_list.append(temphash)
    temphash = []

1 loop, best of 3: 7.63 s per loop

```

computed image hashes in around 0.2 seconds. Based on improvements from CPU multiprocessing with 2 GPUs the execution time might fall around 40% further.

In any case, further progress could be made by editing individual functions in the algorithm. Although I conducted some early work on this, I had issues with implementing them successfully (see attached a .py file showing an attempt to streamline FillFloatLuma, this executed without errors but failed unit tests for reasons I could not solve in time). This reflects some of the tradeoffs with optimising a foreign codebase. With the relatively cheap availability of hardware, such as through AWS, spinning up numerous CPUs and multiprocessing in the cloud (even spinning up multiple GPUs!) and computing these results relatively quickly is likely to be very cheap. Alternatively, many of the major open computation tasks have been solved through highly efficient open source libraries, such as the image hashing libraries we use below, which I would assess to be the best approach to this problem. If we were forced to use FINDHasher, I would recommend some further code optimisation for efficiency, and then the job be applied in as much scale as affordable on a number of cloud GPUs. This job time could then be minutes.

That said, the value of optimising code at the line-level does mean it could be run on a home computer, and so more progress in that department might be valuable. Nonetheless, I have shown that careful analysis of the FINDHasher algorithm can result in significant optimisations which could be extended further if required (and with more time and support available to me).

## 7 Accuracy Testing

Accuracy testing these images is also an important part of benchmarking this code. The goal of image hashing algorithms is to produce image hashes that represent the image underlying them. That is, similar images should have relatively similar hashes, and different images should have relatively different hashes.

In image hashing this concept is known as the Hamming distance, which represents the distance between two 256-bit hashes as the number of bits that are different at each position. Helpfully, this distance is implemented in the image hashing library used for this codebase.

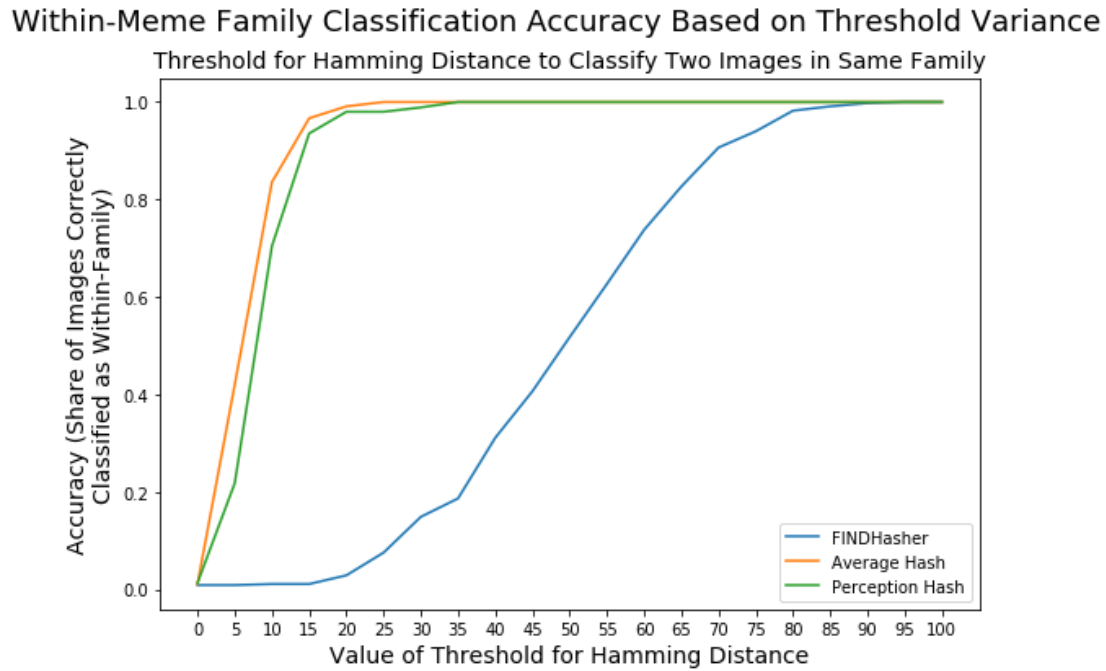
In the image database used for testing this code, images are classified into ‘families’ of different memes. As a result, we can easily access images from the same family, and examine the clustering of the hashes within family using a distribution of pairwise Hamming distances. This gives us a sense of how well the image hashing algorithm groups various families of images.

We do this by hashing a sample of images from a sample of meme families using the base algorithm FINDHasher, and plot these as histograms of the pair-wise hamming distances between each image to examine how close they are on average. As an example, the distributions for a sample of 10 images from 10 meme families for image hashing from FINDHasher are shown in Appendix 1. We then imagine that we do not know the classification of these images, and would wish to collect them into meme families. What is the minimum threshold for the Hamming distance would we require for this algorithm to collect them all accurately?

We loop over the pairwise values and compute accuracy for a large range of different thresholds for the Hamming distance. We then extend this exercise to include (and also evaluate the performance of) average hashing and perception hashing, for comparison. This is shown in Figure 12. This shows that FINDHasher does not do a particularly good job of collecting images into meme families, relative to the other two algorithms.

One weakness of this approach is that it only identifies the share of false negatives (it only identifies images that have been inappropriately classified as *not in* the same meme family, rather than images that have been inappropriately classified as *in* the same meme family. We can tell nonetheless that the FINDHasher algorithm

Figure 12: Hashing Accuracy Comparisons



is far more likely to suffer from false negatives, as it takes a much larger Hamming distance to collect all images in the same meme family (around 95 or so). The other two algorithms collect almost all of their images with thresholds around 35.

This shows that the other hashing algorithms (perception and average) are probably much more suited to this task if it was ‘in the wild’, i.e., if the task was to create collections of similar images.

Computational profiling of these algorithms also showed that they were very quick at their tasks relative to FINDhasher (see Figure 13). FINDhasher (Numba optimised) ran 10 images in 3 seconds, while perceptual hasher took 87 milliseconds and average hasher took 70.

Figure 13: Hashing Accuracy Comparisons

```

findHasher_1

%%timeit
img_sample = sample(img_filename_list, 10)
hash_output_base = []
for i in range(0, len(img_sample)):
    img_string_input = 'das_images/das_images/{}'.format(img_sample[i])
    hash_temp = findHasher_1.fromFile(img_string_input)
    hash_output_base.append(hash_temp)

2.99 s ± 114 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Average Hasher

%%timeit
img_sample = sample(img_filename_list, 10)
hash_output_avg = []
for i in range(0, len(img_sample)):
    img_string_input = 'das_images/das_images/{}'.format(img_sample[i])
    hash_temp = imagehash.average_hash(Image.open(img_string_input))
    hash_output_avg.append(hash_temp)

70.2 ms ± 5.4 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

Perceptual Hasher

%%timeit
img_sample = sample(img_filename_list, 10)
hash_output = []
for i in range(0, len(img_sample)):
    img_string_input = 'das_images/das_images/{}'.format(img_sample[i])
    hash_temp = imagehash.phash(Image.open(img_string_input))
    hash_output.append(hash_temp)

87.1 ms ± 8.84 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

```

## A Appendix

### A.1 Hamming Distance Histograms - Within Meme Families



