Assignment 03 – Profiling and Numba

Konstantinos Georgiou

General Info

- The code and the results for the assignment 3 is in this GitHub repo: https://github.com/drkostas/DSE512-playground
- The configuration I am using to run this assignment is this: https://github.com/drkostas/DSE512-playground/blob/master/confs/assignment3_local_tcga.yml
- Most of the code is in the *assignment3* folder: https://github.com/drkostas/DSE512-playground/tree/master/assignment3
  - There are 4 different *KMeans* implementations:
    - **simple**: Th non-vectorized *Kmeans* we created in class
    - **vectorized_jacob**: The vectorized *Kmeans* we created in class
    - **vectorized**: My vectorized *Kmeans* version that I created in the previous assignment by improving Jacob's vectorized.
    - **jitted vectorized_jacob**: The jitted **vectorized_jacob** implementation after modifying it a bit to make it able be jitted.
  - I didn't jit my vectorized implementaiton because the use of **scipy.spatial.distance.cdist**, and **np.argmin** were messing with numba's nopython mode.
  - **assignment3.py**: Loads the configuration, and runs the appropriate *KMeans* function for each subconfig (**simple**, **vectorized_jacob**, **vectorized**) by calling either **kmeans.py** or **kmeans_numba.py**
  - **kmeans.py**: It contains the KmeansRunner class which includes all the (non-numba) Kmeans implementations and the load_dataset() function.
  - **kmeans_numba.py**: Contains the **jitted vectorized_jacob** implementation.
- I am also importing some other custom packages I've made, from which the most important ones are:

    - **profileit**: cProfile ContextManager-Decorator for profiling functions or code blocks - https://github.com/drkostas/DSE512-playground/blob/master/playground/profiling_funcs/profileit.py
    - **timeit**: ContextManager-Decorator for timing functions or code blocks - https://github.com/drkostas/DSE512-playground/blob/master/playground/timing_tools/timeit.py
- Profiling raw results & screenshots: https://github.com/drkostas/DSE512-playground/tree/master/outputs/final/assignment3/profiling
- Runtime results & Amdahl Plots: https://github.com/drkostas/DSE512-playground/tree/master/outputs/final/assignment3/results

1. Refactoring kmeans.py

In **kmeans.py**, there is a **run**() function which calls one of: **run_simple()**, **run_vectorized_jacob(),** r**un_vectorized()** inside a **profileit** *with* statement.

I refactored these 3 functions, to call a **_loop()** sub-function which in turn calls the **_compute_distances()**, **_expectation_step()**, **_maximization_step()** functions to run each individual step of the algorithm. Each implementation has different functions for these steps, for example, **run_vectorized_jacob()** calls **_loop_vectorized_jacob()** etc. If this is not clear enough, feel free to ask me and I can elaborate more. Example:

```
@staticmethod
def _compute_distances_simple(num_points: int, num_features: int, num_clusters: int,
                              centroids: np.ndarray, features: np.ndarray):...

@staticmethod
def _expectation_step_simple(num_points: int, num_clusters: int,
                             centroid_distances: np.ndarray, cluster_assignments: np.ndarray):...

@staticmethod
def _maximization_step_simple(num_clusters: int, num_points: int, cluster_assignments: np.ndarray,
                              features: np.ndarray, centroids: np.ndarray):...

@staticmethod
def _loop_simple(num_clusters: int, num_points: int, num_features: int,
                 cluster_assignments: np.ndarray, features: np.ndarray, centroids: np.ndarray):...

@staticmethod
def run_simple(features: np.ndarray, num_clusters: int):...
```

2. Profile kmeans.py

Inside **kmeans.py**, the **run()** functions, calls the appropriate implementation inside a **profileit** context manager. The times are the following:

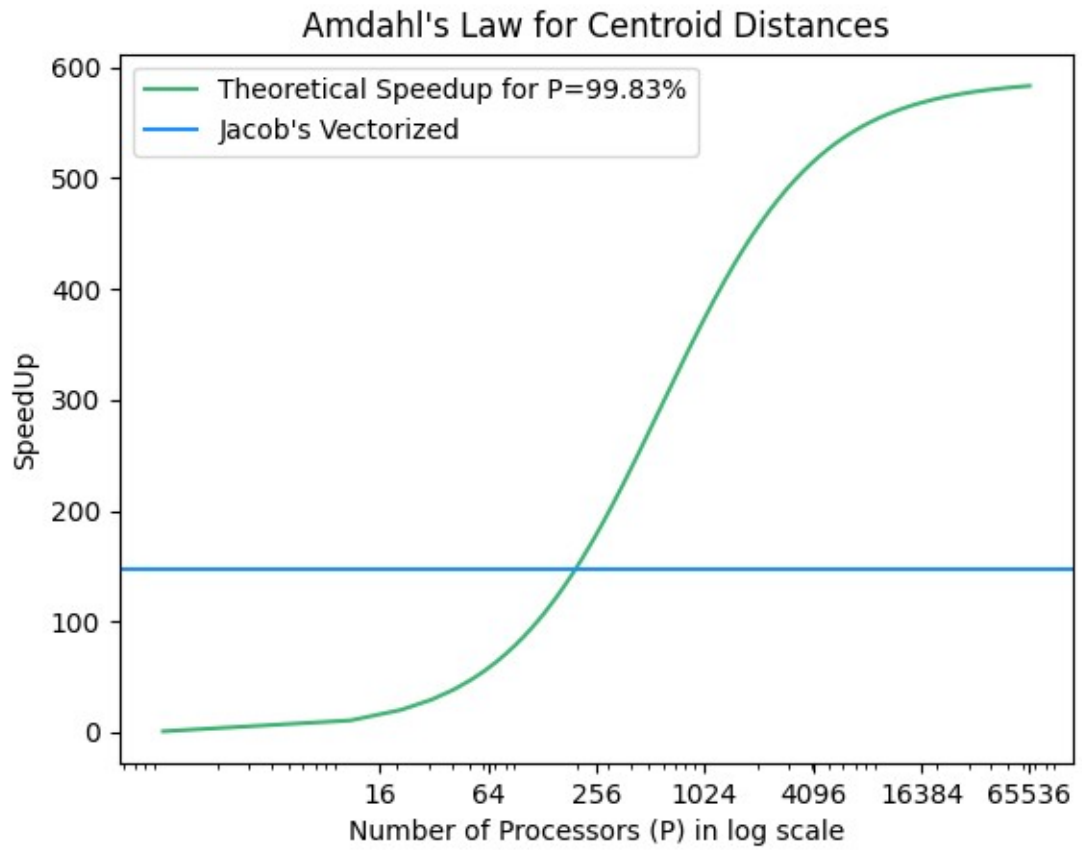
Algorithmic Step	Kmeans Simple	Jacob's Kmeans Vectorized	My Kmeans Vectorized
Compute Distances	627.4(s) – 99.83%	3.27(s) – 76.16%	1.336(s) – 52.22%
Expectation	0.0311(s) – 0.00005%	0.0324(s) – 0.75%	0.0004(s) – 0.02%
Maximaziation	1.02(s) - 0.16%	0.9901(s) – 23.06%	1.221(s) – 47.72%
Total	628.5(s) – 100%	4.294(s) – 100%	2.559(s) – 100%

The speedup from my Kmeans implementation is not relevant because I attempted to improve all three functions.

The total speedup of Jacob's Kmeans Vectorized compared to Kmeans simple is:

Jac_vec_speedup = 628.5(s)/4.294(s) = 146.367

Plotting Amdahl's Law with this speedup yields the following figure:



The maximum theoretical speedup that centroid distances can give if parallelized according to Amdahl's law is **~583 times**. Jacob's vectorized Kmeans achieved **~25.1%** of that maximum.

3. Visualize Icicle Plots

Simple Kmeans

Icicle Plot:

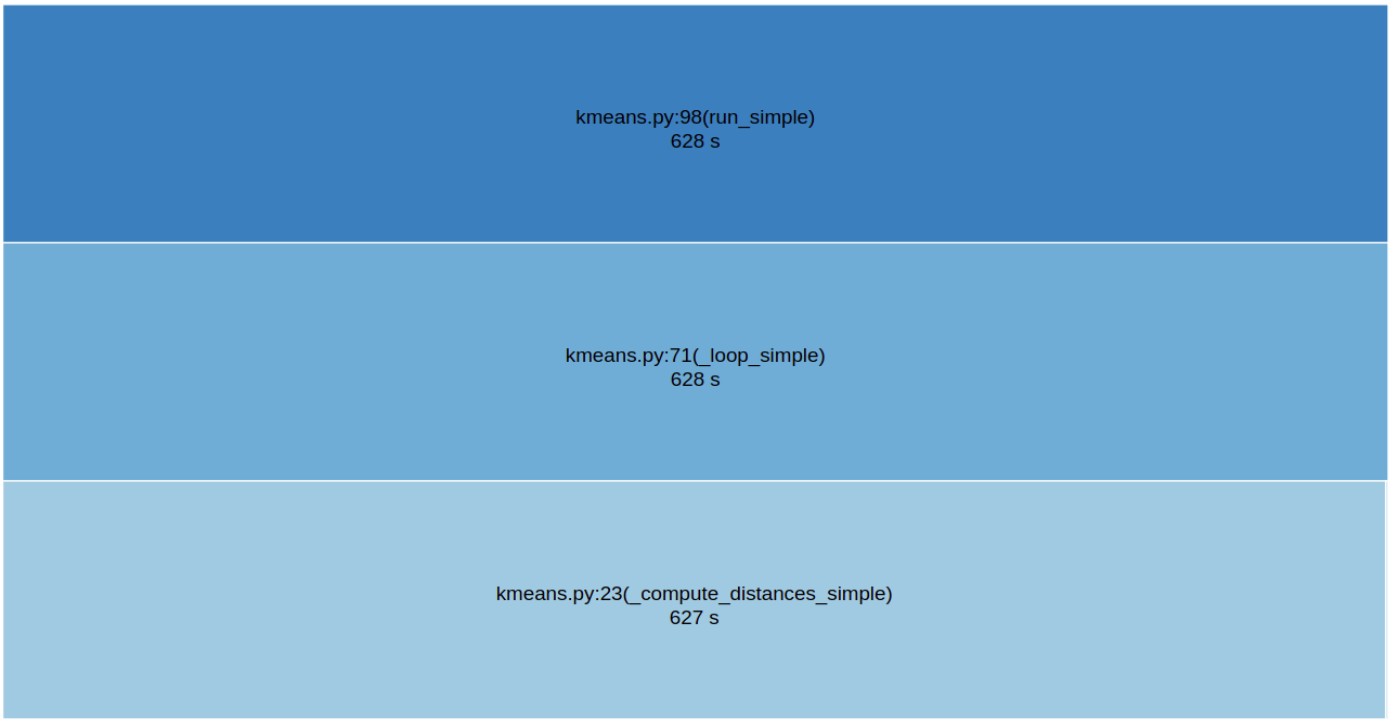


Table with times and calls:

ncalls	tottime	percall	cumtime	percall	file
12	627.4	52.29	627.4	52.29	kmeans.py:23(_compute_distances_simple)
12	1.02	0.08497	1.02	0.08497	kmeans.py:57(_maximization_step_simple)
12	0.03107	0.002589	0.03107	0.002589	kmeans.py:39(_expectation_step_simple)
1	0.001574	0.001574	628.5	628.5	kmeans.py:98(run_simple)
1	0.000139	0.000139	628.5	628.5	kmeans.py:71(_loop_simple)

Showing 1 to 5 of 5 entries (filtered from 19 total entries)

Jacob's Vectorized Kmeans

Icicle Plot:

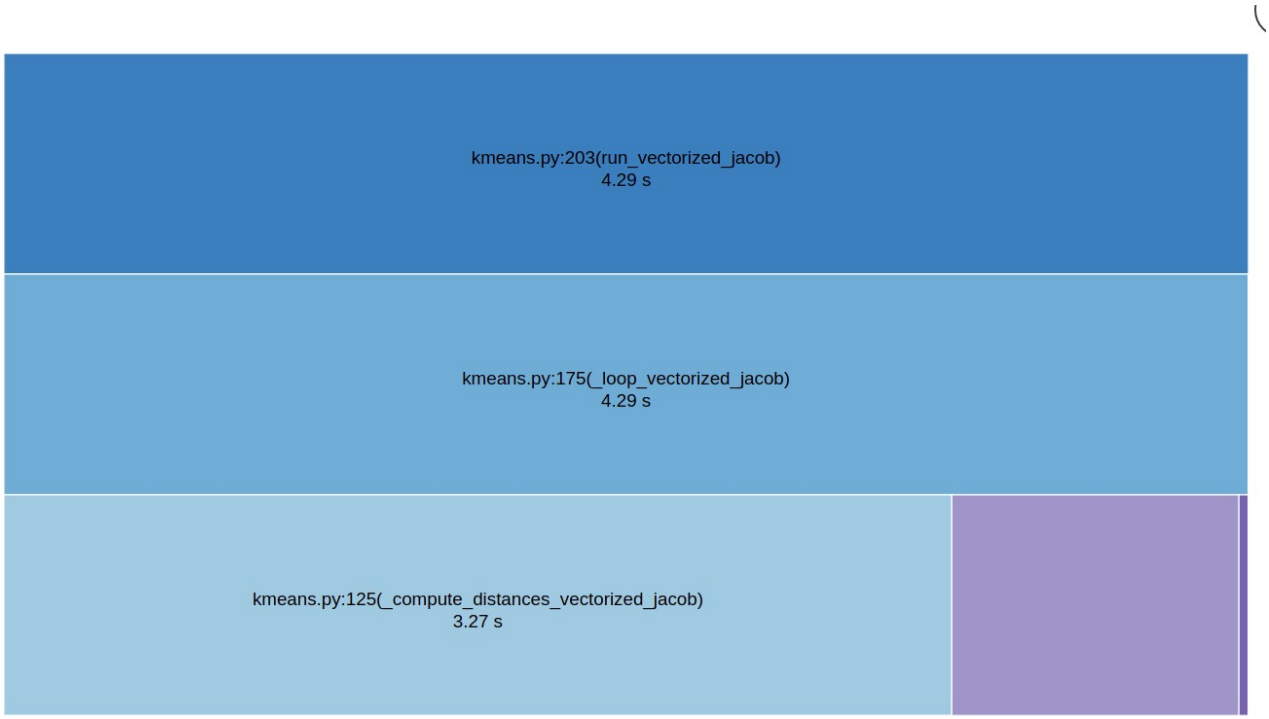


Table with times and calls:

ncalls	tottime	percall	cumtime	percall	filename:line
1	0.000885	0.000885	4.294	4.294	kmeans.py:203(run_vectorized_jacob)
1	0.000119	0.000119	4.293	4.293	kmeans.py:175(_loop_vectorized_jacob)
12	2.642	0.2202	3.27	0.2725	kmeans.py:125(_compute_distances_vectorized_jacob)
12	0.9901	0.08251	0.9901	0.08251	kmeans.py:159(_maximization_step_vectorized_jacob)
12	0.0324	0.0027	0.0324	0.0027	kmeans.py:138(_expectation_step_vectorized_jacob)

Showing 1 to 5 of 5 entries (filtered from 23 total entries)

My Vectorized Kmeans

Icicle Plot:

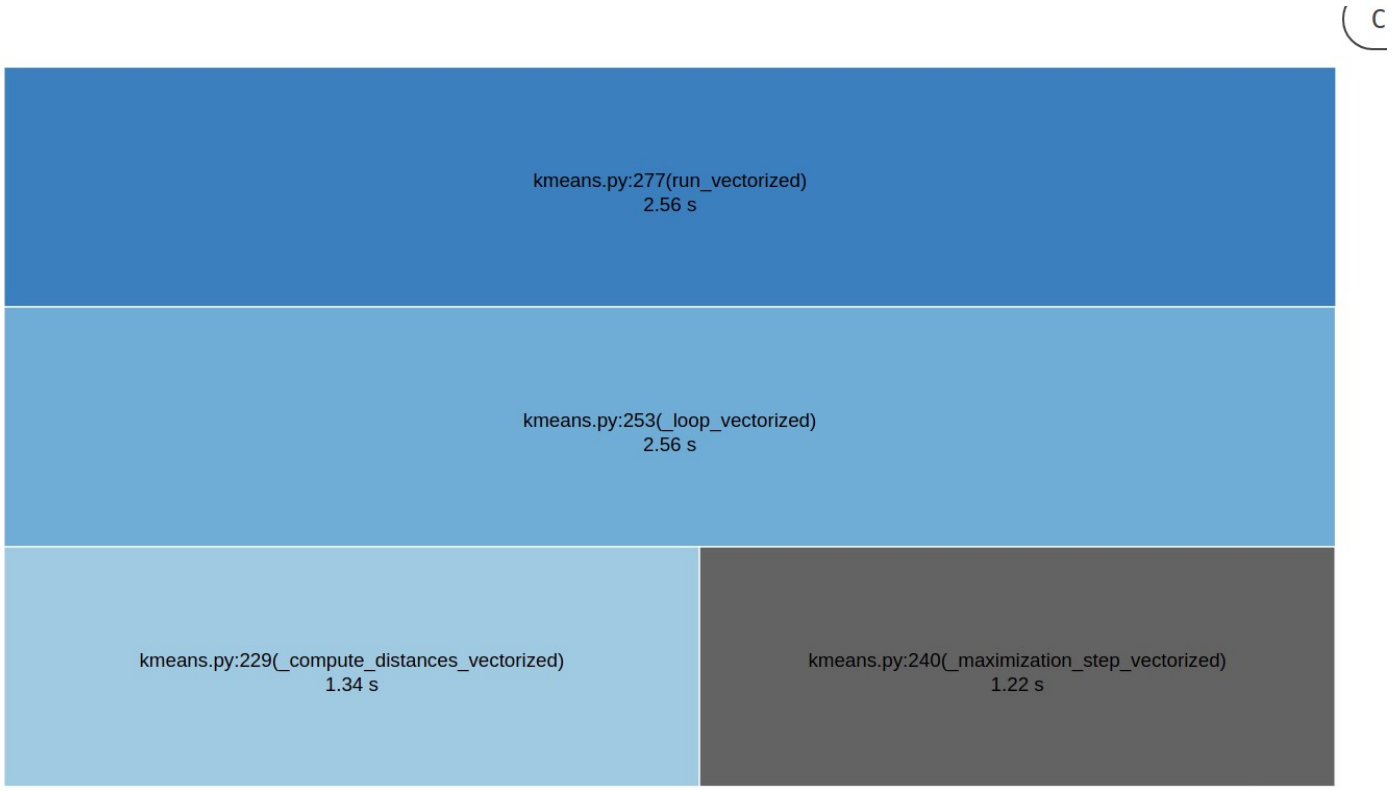


Table with times and calls:

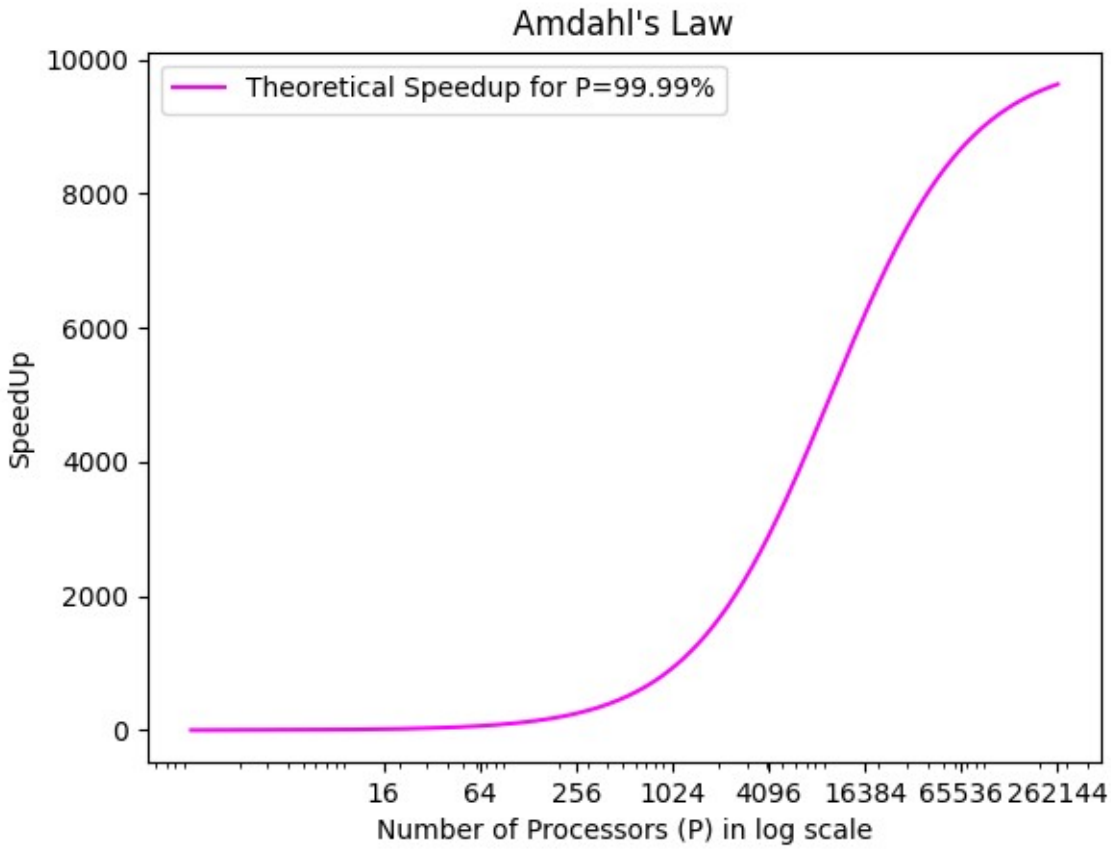
ncalls	tottime	percall	cumtime	percall	filena
1	0.000862	0.000862	2.559	2.559	kmeans.py:277(run_vectorized)
1	0.00015	0.00015	2.558	2.558	kmeans.py:253(_loop_vectorized)
12	0.01222	0.001018	1.336	0.1113	kmeans.py:229(_compute_distances_vectorized)
12	1.072	0.08932	1.221	0.1017	kmeans.py:240(_maximization_step_vectorized)
12	6.1e-05	5.083e-06	0.0004	3.333e-05	kmeans.py:235(_expectation_step_vectorized)
12	8.6e-05	7.167e-06	0.000211	1.758e-05	kmeans.py:249(_break_condition_vectorized)

Showing 1 to 6 of 6 entries (filtered from 333 total entries)

4. Numba

According to the table from Problem 2, the three functions (**compute_distances**, **expectation_step**, and **maximization_step**) take up 99.83%+0.00005%+0.16% = 99.99% of the code.

For the maximum speedup let's plot Amdahl's Law again:



The maximum theoretical speedup that all three can give if parallelized according to Amdahl's law is **~9632.57 times**.

Running **Jacob's Vectorized Kmeans** using numba took this time:

Algorithmic Step	Kmeans Simple	My Kmeans Vectorized	Jacob's Vectorized Kmeans with Numba
Compute Distances	627.4(s) – 99.83%	1.336(s) – 52.22%	3.3935(s) – 68.55%
Expectation	0.0311(s) – 0.00005%	0.0004(s) – 0.02%	0.1570(s) – 3.17%
Maximaziation	1.02(s) - 0.16%	1.221(s) – 47.72%	1.3980(s) – 28.24%
Total	628.5(s) – 100%	2.559(s) – 100%	4.9504(s) – 100%

Jacob's Vectorized Kmeans with Numba speedup: 628.5/4.9504 = 126.95 → 1.3%

My Vectorized Kmeans speedup: 628.5/4.9504 = 245.6 → 2.5%

Plotting Amdahl's Law along with the two speed ups: