Lab 3 - Report

Juan David Gomez, Nicolas Ortiz, Juan David Rengifo Mera & Francisco Suarez
Parallel Programming
Pontificia Universidad Javeriana Cali

June 09, 2021

1 Hardware Features

The experimental environment where the tests were taken are as follows:

Configuration items	Item value
System version	Linux 5.4 (Google Colab)
Python	3.6.5
CPU	Intel Xeon CPU @2.20Ghz
GPU	NVIDIA Tesla T4 (2560 Cuda Cores)
RAM size	13 GB

Table 1: General system information

2 Problem Specification & algorithm description

2.1 Algorithm description

- **Input:** This algortighms has many inputs: obs, korGuess, iter, thresh, checkFinite and seed. obs is a matrix where each row of the M by N array is an observation vector. The columns are the features seen during each observation. KorGuess is the number of centroids to generate, the initial k centroids are chosen randomly or it can be passing an array. iter is the number of times to run k-means. thresh is a float and it terminates the k-means algorithm if the change in distortion since the last k-means iteration is less than or equal to threshold. checkFinite is a boolean and check that the input matrices contain only finite numbers. seed is for initializing the pseudo-random number generator.In the algorithm iter, thresh, checkFinete and seed are optional parameters.

- **Output:** First the algorithm return a k by N matrix of k centroids .The ith centroid codebook[i] is represented with the code i. The centroids and codes generated represent the lowest distortion seen, not necessarily the globally minimal distortion.

  Secondly the algorithm returns a float which represents the distortion and is the mean (non-squared) Euclidean distance between the observations passed and the centroids generated.

- **Description:** The algorithm performs K-means on a set of observation vector forming k clusters. The k-means algorithm adjusts the classification of the observations into clusters and updates the cluster centroids until the position of the centroids is stable over successive iterations. A vector v belongs to cluster i i it is closer to centroid i than any other centroid. If v belongs to i, we say centroid i is the dominating centroid of v. The k-means algorithm tries to minimize distortion, which is defined as the sum of the squared distances between each observation vector and its dominating centroid.

2.2 Computational complexity analysis

The overall complexity of the K-mean algorithm is $O(n^2)$ this due to our main k-mean functio (kmeans) calls ”iter” times the function -kmeans and this function has a $O(n)$ complexity, so in that way the overall complexity of the K-means algorithm is $O(n^2)$.

2.3 Secuential Algorithm

```

def vq(obs, code_book, check_finite=True):
    obs = _asarray_validated(obs, check_finite=check_finite)
    code_book = _asarray_validated(code_book, check_finite=check_finite)
    ct = np.common_type(obs, code_book)

    c_obs = obs.astype(ct, copy=False)
    c_code_book = code_book.astype(ct, copy=False)

    if np.issubdtype(ct, np.float64) or np.issubdtype(ct, np.float32):
        return _vq.vq(c_obs, c_code_book)
    return py_vq(obs, code_book, check_finite=False)


def py_vq(obs, code_book, check_finite=True):
    obs = _asarray_validated(obs, check_finite=check_finite)
    code_book = _asarray_validated(code_book, check_finite=check_finite)

    if obs.ndim != code_book.ndim:
        raise ValueError("Observation and code_book should have the same rank")

    if obs.ndim == 1:
        obs = obs[:, np.newaxis]
        code_book = code_book[:, np.newaxis]

    dist = cdist(obs, code_book)
    code = dist.argmin(axis=1)
    min_dist = dist[np.arange(len(code)), code]
    return code, min_dist

def _kmeans(obs, guess, thresh=1e-5):

```

```python

    code_book = np.asarray(guess)
    diff = np.inf
    prev_avg_dists = deque([diff], maxlen=2)
    while diff > thresh:
        # compute membership and distances between obs and code_book
        obs_code, distort = vq(obs, code_book, check_finite=False)
        prev_avg_dists.append(distort.mean(axis=-1))
        # recalc code_book as centroids of associated obs
        code_book, has_members = _vq.update_cluster_means(obs, obs_code,
                                                          code_book.shape[0])
        code_book = code_book[has_members]
        diff = prev_avg_dists[0] - prev_avg_dists[1]

    return code_book, prev_avg_dists[1]


def kmeans(obs, k_or_guess, iter=20, thresh=1e-5, check_finite=True,
           *, seed=None):

    obs = _asarray_validated(obs, check_finite=check_finite)
    if iter < 1:
        raise ValueError("iter must be at least 1, got %s" % iter)

    # Determine whether a count (scalar) or an initial guess (array) was passed.
    if not np.isscalar(k_or_guess):
        guess = _asarray_validated(k_or_guess, check_finite=check_finite)
        if guess.size < 1:
            raise ValueError("Asked for 0 clusters. Initial book was %s" %
                             guess)
        return _kmeans(obs, guess, thresh=thresh)

    # k_or_guess is a scalar, now verify that it's an integer
    k = int(k_or_guess)
    if k != k_or_guess:
        raise ValueError("If k_or_guess is a scalar, it must be an integer.")
    if k < 1:
        raise ValueError("Asked for %d clusters." % k)

    rng = check_random_state(seed)

    # initialize best distance value to a large value
    best_dist = np.inf
    for i in range(iter):
        # the initial code book is randomly selected from observations
        guess = _kpoints(obs, k, rng)
        book, dist = _kmeans(obs, guess, thresh=thresh)
        if dist < best_dist:
            best_book = book
            best_dist = dist
    return best_book, best_dist
```

3 Proposal for the parallel implementation

Wr consider using Global memory and Constant memory for doing the parallelization. Furthermore, the focus of our work will consist on the initial centroid selection. The clustering part is implemented the same as the K-means algorithm. In the centroid selection process, every time a new centroid is found, it's index is stored in a separate array. We plan to use these indices to find the coordinates of the centroid in the array containing all the points.
Each thread in the centroid selection process will be responsible for calculating the distance of a single point from all the selected centroids; hence, each thread has an equal workload. This workload increases equally with each subsequent centroid selection. If the K is small, the workload of each thread is small and equal at all times.

4 Data dependency analysis

Notice that each iteration of the algorithm proposes more accurate centroids and with this, all the distances must be recalculated. As we paralellize the computing of these distances, observe that nor distance calculation collides with another one, then there are no data dependency between distances calculations.

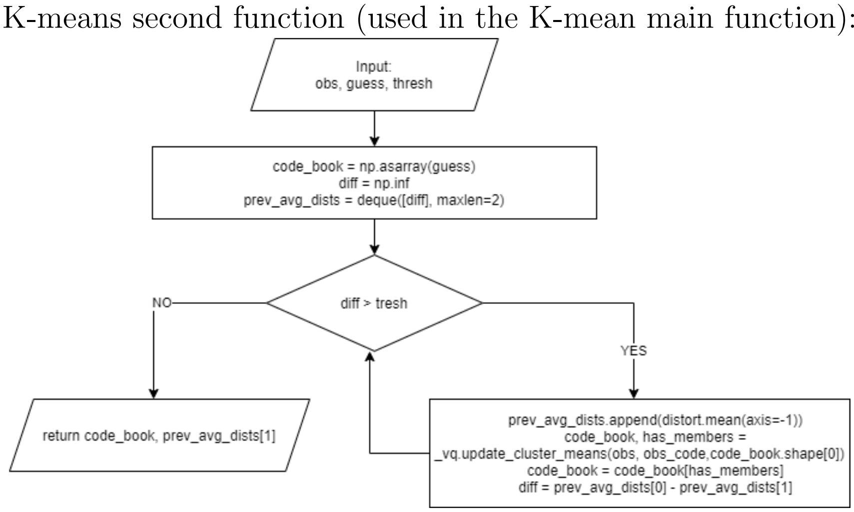
5 Data Limits

For the execution of the k-means algorithm we will use a data limit of up to 200,000 vectors, this due to the limitations imposed by the google colabs platform

6 Flow Diagram

K-means main function:

K-means second function (used in the K-mean main function):



7 Practical uses of the algorithm

The K-means algorithm is an algorithm commonly used in the unsupervised learning of automated systems and in the data analysis.The main applications of unsupervised learning are the segmentation of data sets by shared attributes. Detection of anomalies that do not fit into any group. Simplification of datasets by adding variables with similar attributes. it serve as a previous step to perform a subsequent processing: higher level of abstraction, calculation of prototypes by class.

8 CUDA implementation

The first thing done was assigning datapoints to their nearest centroid . This step is not difficult to parallelize because the distance computations can be performed independently for each datapoint.

Here the thread index validx corresponds to the index of the datapoint. The rest of the code is pretty straight-forward. This distance between the datapoint g_idata[validx] and each centroid g_centroids[c] is computed and the centroid that is closest is then assigned to that datapoint. One drawback of this code is that the centroids and datapoints are read from global memory which is somewhat slow.

The next step is to recompute the centroids given the cluster assignments computed in the previous step. This is much more tricky to parallelize since the centroid computations depend on all of the other datapoints in its cluster. However, operations that rely on distributed datasets to compute a single output value can still be parallelized, and we proceeded partitioning the input array and performing the sum on each partition in parallel then merge the partitions and repeat the process until all partitions have been merged and you are left with the final sum value. This parallelization allows for logarithmic complexity rather than linear as with the serial case.

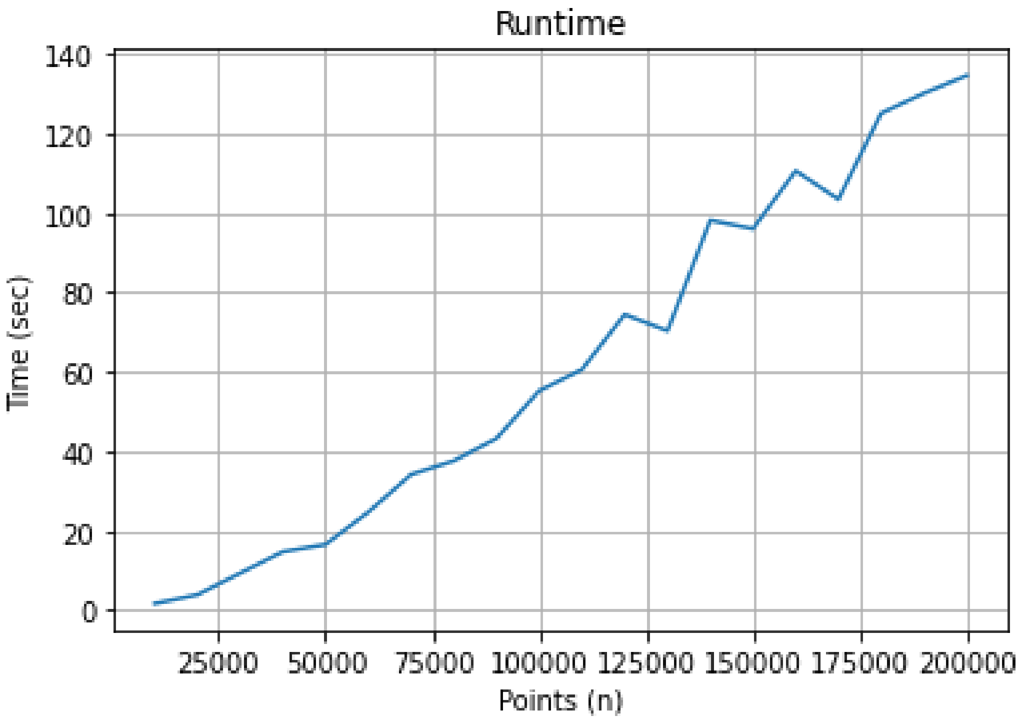
```
def cu_vq(obs, clusters):
    global vals
    kernel_code_template = """
     #include "float.h"
      __device__ void loadVector(float *target, float* source, int dimensions
          ){
          for( int i = 0; i < dimensions; i++ ) target[i] = source[i];
      }
      // the kernel definition
      __global__ void cu_vq(float *g_idata, float *g_centroids, int * cluster,
          float *min_dist, int numClusters, int numDim, int numPoints) {
        int valindex = blockIdx.x * blockDim.x + threadIdx.x ;
        __shared__ float mean[%(DIMENSIONS)s];
        float minDistance = FLT_MAX;
        int myCentroid = 0;
        if(valindex < numPoints){
          for(int k=0;k<numClusters;k++){
```

```
            if(threadIdx.x == 0) loadVector( mean, &g_centroids[k*(numDim)], numDim );
            __syncthreads();
            float distance = 0.0;
            for(int i=0;i<numDim;i++){
              float increased_distance = (g_idata[valindex+i*(numPoints)] - mean[i]);
              distance = distance +(increased_distance * increased_distance);
            }
            if(distance<minDistance) {
              minDistance = distance ;
              myCentroid = k;
            }
          }
          cluster[valindex]=myCentroid;
          min_dist[valindex]=sqrt(minDistance);
        }
      }
    """
    nclusters = clusters.shape[0]
    points = obs.shape[0]
    dimensions = obs.shape[1]
    block_size = 512
    blocks = int(math.ceil(float(points) / block_size))

    kernel_code = kernel_code_template % {'DIMENSIONS': dimensions}
    mod = compiler.SourceModule(kernel_code)

    dataT = obs.T.astype(np.float32).copy()
    clusters = clusters.astype(np.float32)

    cluster = gpuarray.zeros(points, dtype=np.int32)
    min_dist = gpuarray.zeros(points, dtype=np.float32)

    kmeans_kernel = mod.get_function('cu_vq')

    start = drv.Event()
    end=drv.Event()
    #Start Time
    start.record()

    kmeans_kernel(driver.In(dataT), driver.In(clusters), cluster, min_dist, np.int32(nclusters), np.int32(dimensions), np.int32(points), grid=(blocks, 1), block=(block_size, 1, 1),)

    end.record()
    end.synchronize()
    #Measure time difference, give time in milliseconds, which is converted to seconds.
    secs = start.time_till(end)*1e-3

    vals.append(secs)

    return cluster.get(), min_dist.get()
```

9 Implementation Problems

When the program was implemented, no major problems occurred. However, during the tests implementation flaws made evident when looking at the code in detail. We noticed that declaring the variable "mean" as shared, the algorithm could access to it more fast and was possible to use it because it just needed to be acceded by its own block.
Additionally, we was taking times with the timeit library, but we noticed that taking the times by events was better.

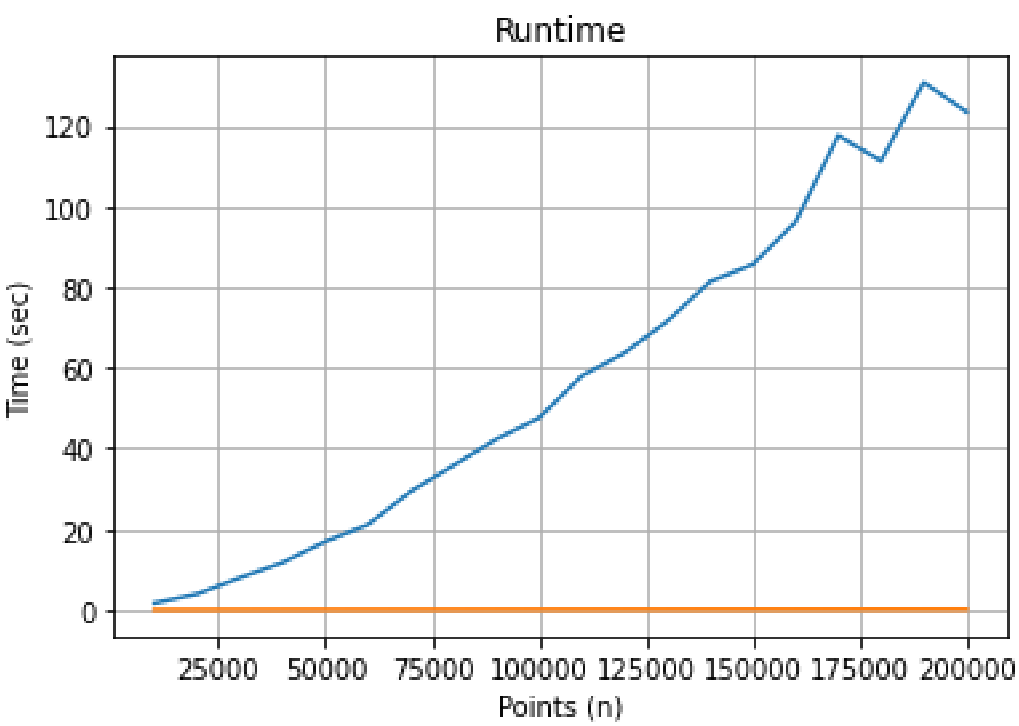
10 Results



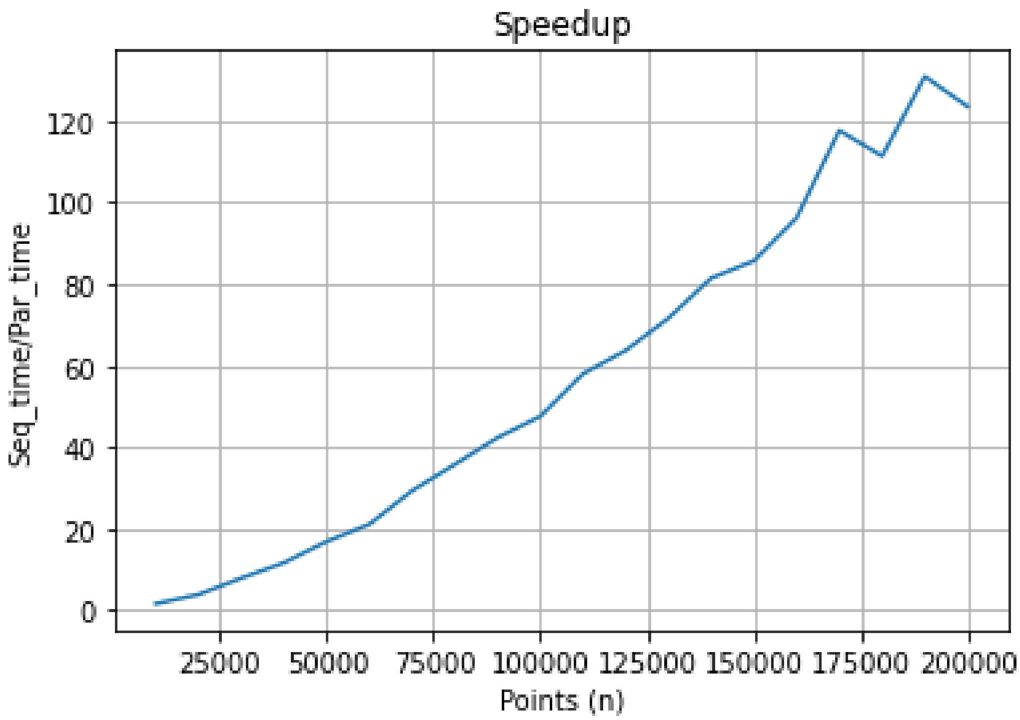
[h]

Serial	Mean	STD
10000	1.67920044899995	0.02642871031616843
20000	3.8688504110000395	0.018929306043201515
30000	9.318562095999937	0.006423371739815899
40000	14.733394031999978	0.03778233550418811
50000	16.540350877999913	0.008819986987224724
60000	24.680594795000047	0.010516554969591188
70000	34.19437143699997	0.008024302891698889
80000	37.623000642999955	0.09573294528073512
90000	43.235951366999984	0.08996239437257884
100000	55.23486680799999	0.14777400677011218
110000	60.59440694299997	0.052545871031616843
120000	74.45860012399987	0.018929306043201515
130000	70.33475058499994	0.006423371739815899
140000	98.15704099999994	0.03778233550418811
150000	96.01786996999999	0.008819986987224724
160000	110.60915491399987	0.010516554969591188
170000	103.43504140699997	0.00804895169188389
180000	125.09680287600008	0.01953732944073512
190000	130.15401496699974	0.02393435567888499
200000	134.56824703799975	0.12523523557011218

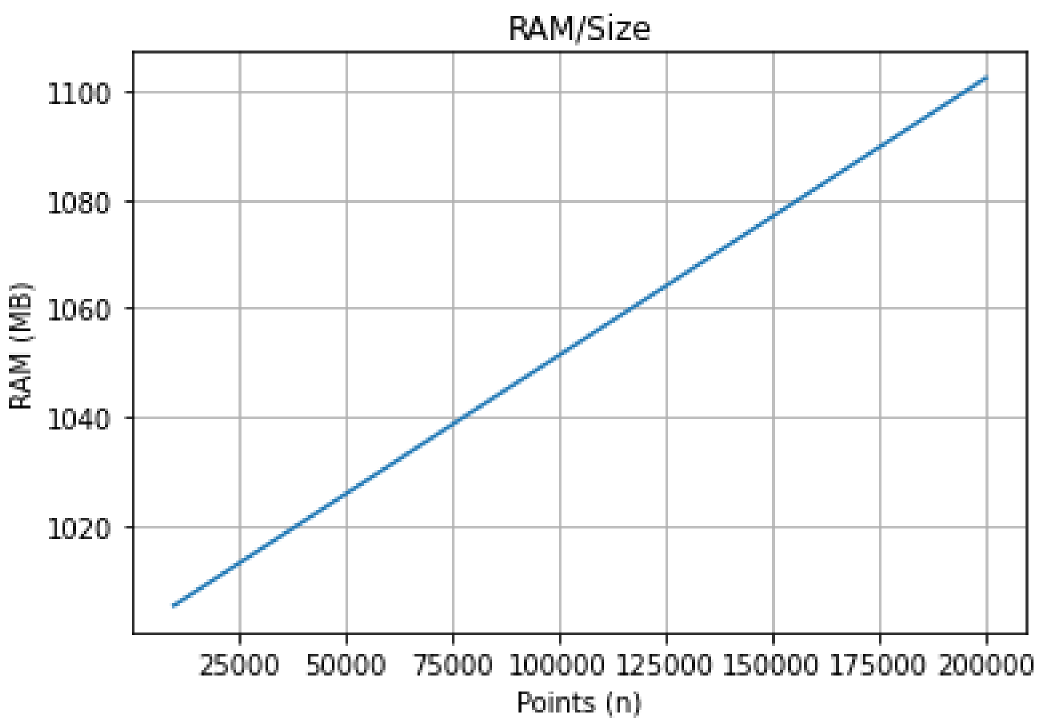
Serial	Mean	STD
10000	0.01624552384738264	0.02749183745632184
20000	0.025172825437603578	0.08547685216549841
30000	0.04510437941551208	0.00658472145875421
40000	0.04747380488259451	0.06895476523145854784
50000	0.06419676920572917	0.062547865896541242
60000	0.0712083884208433	0.045874665862456266
70000	0.09130078722979573	0.00847326685625666
80000	0.09451321072048612	0.087458512353326332
90000	0.10439322909793335	0.089752563265632663
100000	0.09742510505344557	0.02364268365622566
110000	0.11937221916049136	0.025626842656236513
120000	0.11919515930978876	0.0135584858185482884
130000	0.1351646402994792	0.0001848417569484862
140000	0.14075888141832857	0.01848486549188548
150000	0.15756463419596353	0.007458756325668568
160000	0.16266283840603302	0.0326485325625628899
170000	0.17756630194365086	0.003258954258865569
180000	0.17768174435047623	0.0258545685245696314
190000	0.20065733476118602	0.02348946845255588
200000	0.19760932585480928	0.125866687451455167



[h]



[h]



[h]

11 Conclusions

As we can see above, the parallelization of the algorithm makes a huge optimization of the runtime. This is because of the algorithmical properties that we exploded with CUDA. Notice that the CUDA implementation parallelizes the computing procedures for the distances calculations between the data points and the centroids; as the datapoints are arranged in an array, we used divide and conquer for the array division part and processed each branch of the tree in parallel, making that calculation process $O(logn)$, therefore, the overall theorical complexity is $O(nlogn)$. But, notice that in the practice, the running time is always less than 0, then we conclude that the algorithm is extremely efficient, almost constant.

12 References:

- https://medium.com/datos-y-ciencia/aprendizaje-no-supervisado-en-machine-learning-a
C3%B3n-bb8f25813edc
- https://github.com/scipy/scipy/blob/master/scipy/cluster/vq.py
- https://docs.nvidia.com/cuda/
- https://github.com/shackenberg/cukmeans.py/blob/master/cukmeans.py
- https://stackoverflow.com/questions/29187479/kmeans-clustering-acceleration-in-gpuc
- https://iopscience.iop.org/article/10.1088/1757-899X/790/1/012036/pdf