A Comparison of Serial and GPU Parallel Processing on the K Means Algorithm

Ming Jeng
mjeng@ucsc.edu

Abstract. K-means clustering is a famous and ubiquitously used Machine Learning algorithm, and a type of unsupervised learning to cluster information based on latent structured information presented within the data. The goal of k-means is to find different clusters in the data with the number of groups represented by K. However, as the number of data points, clusters, and iterations increases, the run time increases exponentially which serves as a bottleneck in the algorithm's performance. This work shows how dramatically time complexity of K means algorithm can be reduced by taking advantage of steps that can be parallelized; while comparing and analyzing different serial and parallel implementations of K means algorithm, the paper will dive into parallel implementations that leverages GPU architecture and parallelization of "Embarrassingly Parallel" tasks for better performance.

1 Introduction

K-means is one of the most commonly used unsupervised learning algorithms and is able to classify unlabeled data with its very intuitive algorithm. The goal when applying k means algorithm is usually either: 1. Get meaningful intuition of the structure of the data 2. If there's a wide variation in the behaviors of different subgroups, then K Means can cluster-then-predict where different models will be built for different clusters. For example, K Means can cluster patients into different subgroups then having seperate models for each subgroup to predict probability of the risk of heart attack. [Dab20] K means is also used in a variety of applications such as market segmentation, document clustering, image segmentation, etc.

Therefore both performance and efficiency improvements of this widely used algorithm will have significant implications for the operations of large-scale data processing workflows. Offloading this K Means algorithm from power hungry CPUs to power efficient GPUs on a single node can save significant amount of energy and eliminating inter-node communications' energy overhead [Far+08]. In this paper, I will compare and analyze between serial and different parallel implementations (Cuda) of K Means algorithm.

1.1 K Means Algorithm

Clustering is a method of grouping n data points into k subgroups or "clusters" where each cluster has maximal similarity as defined by the features/distance metric.

The following is the pseudo code for K means algorithm[Dab20]:
1. Specify number of clusters K.
2. Initialize centroids by first shuffling the dataset and then randomly selecting K data points for the centroids without replacement.
4. Keep iterating until there is no change to the centroids. i.e assignment of data points to clusters isn't changing.
- Compute the distance from each point xi to each cluster centroid uj
- Assign each data point to the closest cluster (centroid).
- Compute the centroids for the clusters by taking the average of the all data points that belong to each cluster.

1.2 Parallelization

The random initialization of the centroids prevent the algorithm from getting same results each run.
The time complexity for K-Means is O(i iterations * k centroids * n data points * d dimensions). As you can see in step 4 (from psuedo code above), it contains highly parallelizable steps of:
1. Compute the distance from each point xi to each cluster centroid uj.
2. Assign each point to the centroid it's closest to.

There's no information that needs to be shared between each point/thread in the above two steps and could be even considered embarrassingly parallel. Each thread can represent a data point and calculate its distance to each centroid. To take advantage of GPU's shared memory, the centroids can be stored in shared memory since all the threads of each block need to access it to compute the distance. For the update step (step 5), where we recompute the new centroids based on the mean of all the points assigned to that cluster, will require information between the different threads. This will be like an average reduction operation done on each individual cluster's subset of the whole dataset (Will be discussed in **Discussion**). However, the tricky part is that we only want to do the average for each cluster and not the whole dataset. Shared memory to store the values for each cluster, then averaging all the values from each shared memory of each block to get the final mean values could be a way to get the mean values for each cluster.

2 Methods

2.1 CPU Implementations (Python and C)

Python Implementation: Here's a very unoptimized implementation of K means to serve as a starting reference point. My Python implementation doesn't utilize the built in K means function from scikit-learn/scipy frameworks nor does it use the numpy library for vectorization optimization benefits. This python implementation only uses nested for loops and list comprehension (probably the

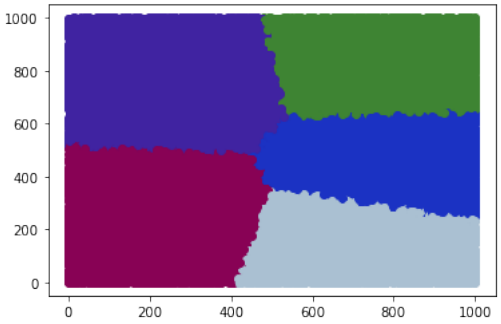
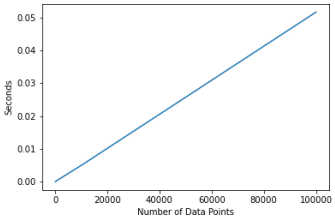


Fig. 1: Illustration of clustering 100,000 2D data points with 5 clusters and 5 iterations using python implementation (25977s)

	50	1000	10000	100000
Serial C (seconds)	0.000032	0.000533	0.004979	0.051663

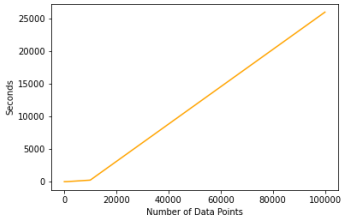
(a) Run times in seconds to data points



(b) Graph representation of table

	50	1000	10000	100000
Serial Python (seconds)	0.283686	5.481091	231.183552	25977.75157

(a) Run times in seconds to data points



(b) Graph representation of table

most unoptimized method possible). [Git20] Both Python and C implementations are fixed in the following parameters $k = 5$ (clusters) and iterations $= 5$ with the only parameter changing is n data points.

As expected, both Python and C serial implementations' run times scale linearly relative to the increase of n data points.[Mar] However, if comparing the run times between Python and C implementations, it becomes drastically clear how much faster C implementation is. This could be attribute to the optimization benefits of vectorization of compiling C code on the CPU[Gol] .

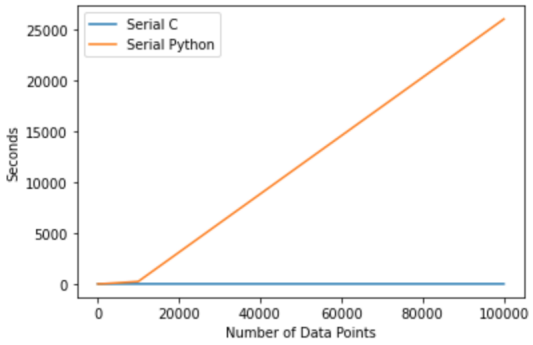


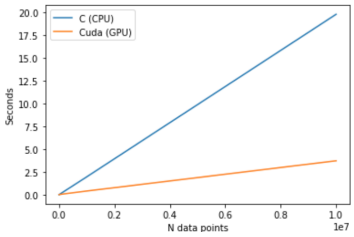
Fig. 4: Comparison between unoptimized python vs C implementation

2.2 Unoptimized GPU Implementation

As mentioned in **Section 1.2**, since the algorithm computes the distance between each data point and each centroid, the biggest parallelization could be done on the algorithm is by dispatching one data point to a thread[Li+10], then each thread calculates the distance from one data point to all the centroids. No information or data is shared across individual data points ("Embarassingly Parallel"), except for immutable cluster centroids [Gol]. However, for this section I will not utilize shared memory, instead, I will use *atomic* counter for calculating the average. All run times results that are presented are averages of 10

	10	50	1000	10000	100000	1000000	10000000
Serial	0.000031	0.000114	0.002056	0.019759	0.199265	1.959600	19.80350
Parallel	0.006552	0.005973	0.006251	0.011533	0.057346	0.404053	3.72182

(a) Run times between CPU and GPU (unoptimized)



(b) Graph representation of table

identical separate runs to mitigate possibility of having an outlier result run time. Interestingly, for number of data points around under 1000, the CPU implementation actually outperforms the GPU implementation, because the cost of transferring data from the host to device for GPU implementation itself takes a lot of time. Only when the number of data points increases, the GPU's runtime can scale and make use of its parallel processing. The span complexity of the GPU doesn't increase but only the work increases, assuming there's enough thread for each data point.[Far+08] As expected, the GPU implementation drastically outperform the CPU implementation. However, there's the bottle neck

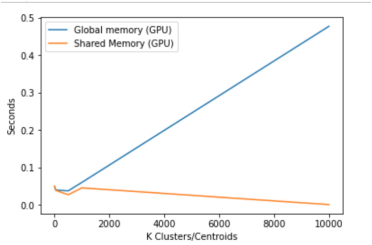
with using atomic operations which may be the cause for the GPU runtime increasing linearly. The next section will show how the GPU code is optimized.

2.3 Shared Memory GPU Implementation

The architectural style of GPUs impose certain limits and coding strategies to gain significant speed ups[Far+08]. In the previous section, the unoptimized GPU implementation requires every thread go to global memory to fetch the cluster means is inefficient. GPU threads are organized into thread blocks, and each block of threads are executed concurrently on one Shared Memory[Li+10]. Threads in thread block can share data through the shared memory and can perform barrier synchronization[Li+10]. A simple adjustment to boost the implementation speed is to place the centroids into shared memory and have threads load them from there (requires minimal code changes). Significant improvements can be shown by increasing the number of clusters: As you can see there's a

	5	50	500	1000	10000
Global Mem GPU	0.049022	0.039605	0.037235	0.059830	0.477184
Shared Memory	0.050313	0.038598	0.026859	0.045248	0.000519

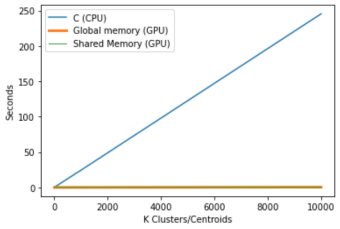
(a) Run times between using Global Memory and Shared Memory (Seconds and K clusters)



(b) Graph representation of table

	5	50	500	1000	10000
CPU	0.195248	1.368100	12.330000	24.390000	245.400000
Global Mem GPU	0.049022	0.039605	0.037235	0.059830	0.477184
Shared Memory	0.050313	0.038598	0.026859	0.045248	0.000519

(a) Comparison of all 3 implementations (Seconds and K clusters)



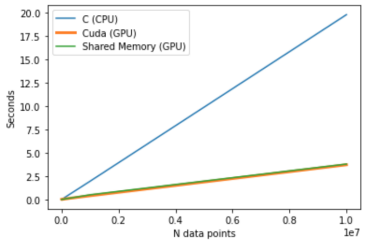
(b) Graph representation of table

dramatic improvement in run time by using shared memory when comes to increasing the number of clusters. The above hyperparameters have the following fixed: n=100,000 and iterations = 20. Like the previous section, all results (run times) are an average of 10 separate identical runs to mitigate possibility of outliers in terms of performance.

However, such drastic improvement wasn't shown if number of clusters remain the same, and only n data points increasing (fixed parameters: k=5,iter=20):

	10	50	1000	10000	100000	1000000	10000000
CPU	0.000031	0.000114	0.002056	0.019759	0.199265	1.959600	19.80350
GPU	0.006552	0.005973	0.006251	0.011533	0.057346	0.404053	3.72182
Shared Memory	0.004570	0.004586	0.005128	0.010240	0.050414	0.510330	3.80274

(a) Comparisons of all 3 implementations



(b) Graph representation of table

This is due to the fact, that the data latency isn't that big of role if the number of centroids stays at only '5', since transferring that small amount of data from global memory to shared memory will be so fast that it won't impact much to overall performance.

3 Discussion and Future work

3.1 Multi GPU

K means can also be utilized in distributed GPU settings. In each GPU, the data assignment and K-centroids recalculations steps of K means algorithm can be offloaded for a massively parallel processing; this solves hardware limitations such as storage capacity and number of threads[Fak+18][Git15].

3.2 Reduction

A reduction algorithm can most definitely be used on calculating the new centroid values, this was the solution to my implementation's bottleneck. While the work complexity is still the same, but the span complexity is only logarithmic. The implementation can be done by keeping a shared memory segment in each thread block and for each cluster and each thread, check if the thread is assigned to the cluster and write the thread's value into the shared memory segment if yes, otherwise write a zero in the place. Then do a simple reduction.This utilizes shared memory very frequently and would be a great way to further optimize my current shared memory implementation[Li+10][Gol].

3.3 Scaling Number of Dimensions

Throughout my implementations, I have only used 2 dimensional data and tuned the algorithm for that assumption. However, a lot of applications require

multiple dimensions and this will create a new kind of bottleneck in performance. In *Speeding up k-Means algorithm by GPUs*[Li+10], they observed that data dimensionality is an important factor that should be taken into consideration when parallelizing k-means on GPUs[Li+10]. For low-dimensional data sets, they exploit GPU on-chip registers to significantly decrease data access latency. For high-dimensional datasets, they design novel algorithm that matrix multiplication and exploits GPU on-chip shared memory [Li+10].

4 Conclusion

I have shown how to parallelize a widely used Machine Learning unsupervised clustering algorithm used by many applications. The results show that there's much to gain by utilizing the parallel processing power of GPUs and their architectural advantages. I have also showed that utilizing shared memory is an optimal way of tackling the problem of scaling the number of clusters specifically. As performance and energy consumption become a prime concern for any data-intensive computations, there will be more deeper ways to tackle k means from different angles for different specific applications[Sal19][Far+08] as we see in **Discussion and Future Work**.

References

[Far+08] Reza Farivar et al. "A Parallel Implementation of K-Means Clustering on GPUs." In: Jan. 2008, pp. 340–345.

[Li+10] Y. Li et al. "Speeding up K-Means Algorithm by GPUs". In: *2010 10th IEEE International Conference on Computer and Information Technology*. 2010, pp. 115–122.

[Git15] Inc. GitHub. *Open Source Survey*. `https://github.com/NVIDIA/kmeans`. 2015.

[Fak+18] Hicham Fakhi et al. "Distributed GPU-Based K-Means Algorithm for Data-Intensive Applications: Large-Sized Image Segmentation Case". In: *International Journal of Advanced Computer Science and Applications* 8 (Jan. 2018), pp. 171–178. DOI: `10.14569/IJACSA.2017.081221`.

[Sal19] Sajjad Salaria. *K Means Clustering for Imagery Analysis*. Aug. 2019. URL: `https://medium.com/datadriveninvestor/k-means-clustering-for-imagery-analysis-56c9976f16b6`.

[Dab20] Imad Dabbura. *K-means Clustering: Algorithm, Applications, Evaluation Methods, and Drawbacks*. Apr. 2020. URL: `https://towardsdatascience.com/k-means-clustering-algorithm-applications-evaluation-methods-and-drawbacks-aa03e644b48a`.

[Git20] Inc. GitHub. *Open Source Survey*. `https://github.com/ming94539/k-means-cluster-CUDA`. 2020.

[Gol] Peter Goldsborough. *Exploring K-Means in Python, C++ and CUDA*.

[Mar] Robert Andrew Martin. *Reasonable Deviations*.