Acceleration of K-Means Clustering

E4750_2021Fall_AMAC_report

Zhejian Jin zj2324 Ruilin Fan rf2756
Columbia University

Abstract

K-means clustering is a classic iterative clustering algorithm. The main objective of the K-Means algorithm is to minimize the sum of distances between the data points and the cluster centroid. In this paper, we explored the acceleration of the K-means clustering algorithm in the programming language CUDA using NVIDIA Graphics Processing Units (GPUs). We proposed 4 ways to accelerate the classic K-means clustering algorithm and compared the speed-up results with the naive python implementation and the optimized algorithm provided by Sklearn. We achieved around 5 times the speed-up on average with our fastest implementation compared to the result of the Sklearn K-means algorithm when the number of data points comes to 1,000,000 with dimension of 8 and the cluster number is set to 5.

Key Words*: K-means, Euclidean Distance, CUDA, GPU acceleration. shared memory*

1. Overview

1.1 Problem in a Nutshell

Data is expanding rapidly in all domains, and the techniques for data analysing are becoming more and more important. By using machine learning, natural language processing, and signal processing, we can learn some patterns from the huge amount of data. For example, clustering algorithms learn the similarities among data points and assign them based on similarities.

A cluster is a collection of data aggregated together due to some similarities. Naive K-means algorithm is one of the most popular unsupervised machine learning algorithms. The idea of the K-means algorithm is very simple. It is a classic clustering algorithm, which uses an iterative refinement technique. It first defines K centroids representing the center of the cluster. Then, every point is assigned to each of the clusters by calculating the distance between each data point and the centroids. As long as each point is allocated, we recompute the centroids of the new clusters. The assignment and recomputation is conducted iteratively until the centroids do not change or the maximum number of iterations has been reached.

The principle of the naive K-means algorithm is relatively simple, it is also easy to achieve. However, the speed of the naive K-means algorithm is slow. There are some optimization methods to the naive K-means algorithm. Some of the previous work focused on the algorithm design and proposed fancy complex K-means algorithms, while other work investigated how to apply a parallel approach to speed up the naive K-means algorithm. In this paper, we also implement multiple parallelization ways to speed up the naive K-means algorithm. The major contributions of this paper are:

- Use shared memory to calculate distance for each data point.
- Use shared memory and parallel scan to sum up each centroid individually.
- Compare the impact on the results of each of the parallel methods.

1.2 Prior Work

From the perspective of the algorithm design itself, we have multiple choices to speed up the K-means algorithm. For example, K-Means++[2] Algorithm has been optimized for the selection of the initial cluster center point, and briefly makes the initial cluster center point as much as possible, which can effectively reduce the number of iterations and speed up the computational speed. Elkan K-Means[3] utilizes both sides equal to the third side, and the difference between the differences between the two sides is less than the triangular nature of the third side, to reduce the calculation of distances. Mini Batch K-Means[4] can preliminarily predict the location of the cluster centers through experiments with small samples. Taking the average of the clustering results of n times of Mini Batch K-Means as the location of the initial cluster centers in a large sample experiment can also reduce the amount of calculations to speed up the clustering speed.

Machine learning algorithms are computationally demanding. Some of the prior work focuses on implementation of K-means on parallel computing platforms. J. Bhimani, M. Leeser and N. Mi speeded up the algorithm by parallel initialization [5]. Shruti Karbhari and Shadi Alawneh used shared memory and constant memory to speed up the K-means clustering algorithm for image segmentation [6].

2. Description

2.1. Goals and Objectives

In this project, the problem setup is simple, we try to use different approaches to implement PyCuda's parallel computing power into the speedup of K-Means algorithm. And since there are multiple time consuming operations in K-Means, more than one approach might be proposed. We aim to explore different methods, compare their performance and analyze the differences.

2.2. Problem Formulation and Design

A K-Means algorithm can be separated into three main parts: (1) calculation of distance between each point and centroids, (2) assignment of points to closest centroids, (3) and lastly the update of centroids means based on the assignment. As the algorithm iterates, the centroids move closer and closer to the actual centers of each cluster, at least in theory. When assignments and centroids converge, the algorithm is done, and we have our cluster centers.
For a traditional serial program, the algorithm loops through all points and centroids to perform all 3 of the above tasks, for a single iteration. To cut back the 'for loop' a little, most of the programs combine (1) and (2) together in a single step so that they keep a running temp to keep track of the closest centroid of each point. Meaning once we loop through the centroids, we naturally obtain the assignment.

Following this idea, the problem we face consists of two parts: (1) calculate distance between points and centers, assign points based on distance. (2) Update centroids means based on the previous assignment. And we do these two parts iteratively. The overview of this structure is shown in algorithm 1:

```
Algorithm 1 kmeans in nutshell
  get input data set of n*m.
  set $numofcluster = k$.
  initialize k centroids of k*m.
  while Assignment changes do
      calculate distance between each points and centroids, assign points to
  clusters
      sum up means for each centroids and update
  end while
```

Algorithm 1: K-means in nutshell

To test out the performance of parallel programs we devised a naive serial python code to perform the above task for comparison.

2.3 System and Software Design

The first naive implementation of GPU computing is to use a kernel function to replace the outer 'for loop' in distance calculation. This is achieved by appointing each thread to manage one point assignment. We use global memory in this kernel function for simplicity. The kernel is followed:

```
Algorithm 2 naive distance kernel
  initialize temp, tid.
  for all centroids do
      calculate distance dist (data[tid],centroids[i])
      if distance ¡ temp then
          temp = distance
          update assignment
      end if
  end for
```

Algorithm 2: naive distance kernel

For the centroids update part, we need to implement a large amount of Atomic operations in Pycuda. The dilemma we face here is this: in order to know the means of given clusters of points, we have to independently sum up all points in the cluster as well as their counts. If we are to take the advantage of parallel computing and distribute workload across the SMs, Atomic operation is required to consistently perform the sum. The naive version of sum kernel is:

```
Algorithm 3 naive sum kernel
  initialize count[], centroids[], tid.
  if check assignment[tid] belongs to centroids n then
      AtomicAdd(centroids[n],data[tid]);
      AtomicAdd(count[n],1);
  end if
```

Algorithm 3: naive sum kernel

Combine the two kernels we have a parallel version of K-Means:

```
Algorithm 4 dist + sum in PyCuda
  get input data set of n*m.
  set $numofcluster = k$.
  initialize k centroids of k*m.
  while Assignment changes do
      device memory setups
      memory initialize
      run naive distance kernel
      update assignment
      device memory setups
      memory initialize
      run naive sum kernel
      update centroids
  end while
```

Algorithm 4: dist + sum in PuCuda

The naive method here can already yield much better performance over our serial implementation, but taking a look at the algorithm 4 we can immediately spot the huge overhead in memory set up between two kernels. We can eliminate this by setting up a unified device memory object that can be used throughout the program.

```
Algorithm 5 dist + sum integrated
  get input data set of n*m.
  set $numofcluster = k$.
  initialize k centroids of k*m.
  unified device memory setups
  memory initialize
  while Assignment changes do
      essential memory initialize
      run naive distance kernel
      essential memory initialize
      run naive sum kernel
  end while
```

Algorithm 5: dist + sum integrated

To further increase performance, we first focus on the distance calculation kernel. In the naive approach, we loop through m dimensions to sum over the distance. Together with the fact that we store dataset in global memory, this means we have over m*k global access. By loading the dataset into shared memory before each operation, we can in theory shrink global access to only m times per block. But in reality this implementation of shared memory is not appreciated in the performance test, which we will discuss later.

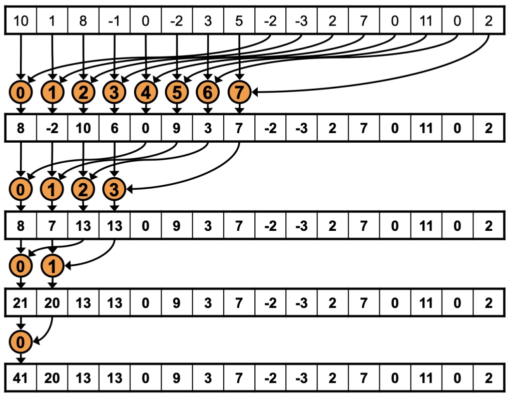


Fig 1: A traditional parallel scan

Now we turn to another potential point for optimization, reducing the Atomic operations used in the sum kernel. We recall that to have a sum over a large dataset, we need to perform add at a specific memory space, which inevitably requires thread and block level synchronization. And that is where overhead comes from. If we draw inspiration from parallel scan application, shown in Fig 4.

We can apply this idea into the sum calculation, with a premise of summing over all input data regardless of their belongings. To combat this we load only zero values into shared memory where data points do not belong to the current cluster. By doing this, we can sum over the clusters with a relatively small amount of Atomic operations. The kernel is:

```
Algorithm 6 Scan + shared sum kernel
initialize count[], centroids[], tid.
if $assignment[id] == cur$ then
    for i to m do
        load shared memory[tid,i];
    end for
else if
    for   dotheni to m
        load shared memory[tid,i] with 0;
    end for
end if
syncthreads;
for stride to 0 do
    Scan through block
end for
syncthreads;
AtomicAdd(centroids[cur],shared memory [0])
```

Algorithm 6: Scan + shared sum kernel

Compared with the previous naive sum kernel, this function moves atomic operations outside of the 'for loop', thus dramatically decreasing the number of atomic operations required. In previous kernel, we have over (n*m) atomic operations for each iteration, now we only have (n*m / block size) for one iteration. But this method needs to be performed (k) times to get sums for all centroids, thus might not be suitable for large number clustering.

3. Results

3.1 Configuration

We use google cloud for our project. The GPU platform is NVIDIA Tesla T4, the operating system is Ubuntu 18.04 LTS and CUDA toolkit version is 10.2. For the experiments, we set dimensions of data to 8 and assigned data points to 5 clusters. The test results are based on ten averages. The data points are generated randomly and follow a normal distribution. The initial centroids are generated first and applied to all of the implementations for fairness.

3.2 Figures & plots

First we implemented the parallel naive distance calculation and replaced the distance calculation with the parallel one. Then we compared the results with the naive K-means one. Figure 2 shows the comparison of speed between naive python K-means implementation and K-means using CUDA distance calculation.

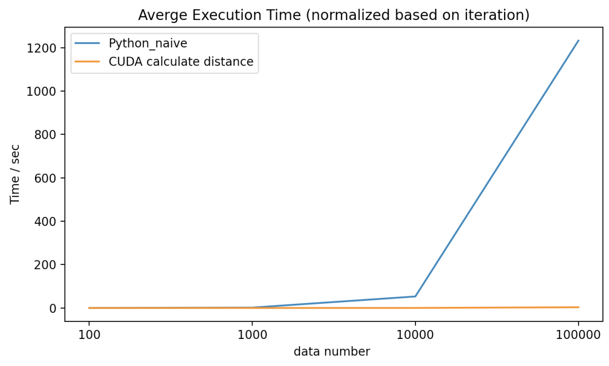


Fig 2: speed comparison between naive python method and K-means using naive parallel distance calculation

From the graph, we can see that only implementing the distance calculation can significantly speed up the K-means algorithm, especially when the data points size is large.

Then, we implemented the parallel naive mean calculation method and compared the results between the naive distance calculation method and the one with both naive distance calculation and naive mean calculation. Figure 3 shows the comparison of speed of the two methods.

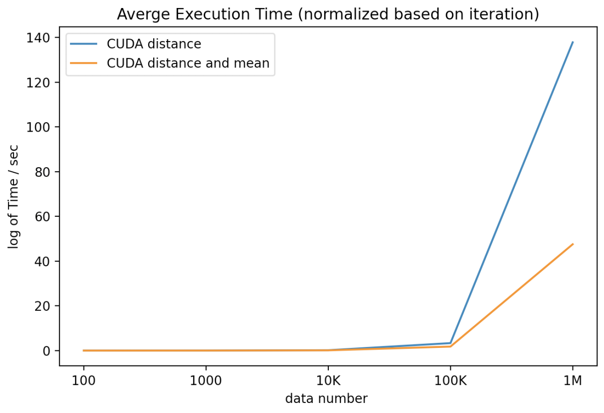


Fig 3: speed comparison between K-means using naive parallel distance calculation and K-means using naive parallel distance calculation and naive parallel mean calculation

We can see from the graph that with the naive parallel mean calculation, the algorithm can be faster. We further implement the integrated method which avoids memory allocation and transfer every time the two kernel functions are called. We rewrite the program so that we are looping between the kernels without memory transfer overhead. Figure 4 shows the comparison of speed of the three methods.

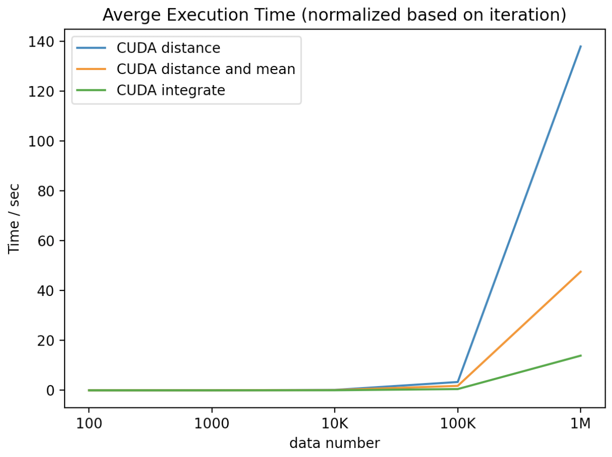


Fig 4: speed comparison between K-means using naive parallel distance calculation, K-means using naive parallel distance calculation and naive parallel mean calculation and K-means using integrated method

We then implemented the parallel distance calculation using shared memory for the integrated method. Figure 7 shows the comparison of speed of the four methods.

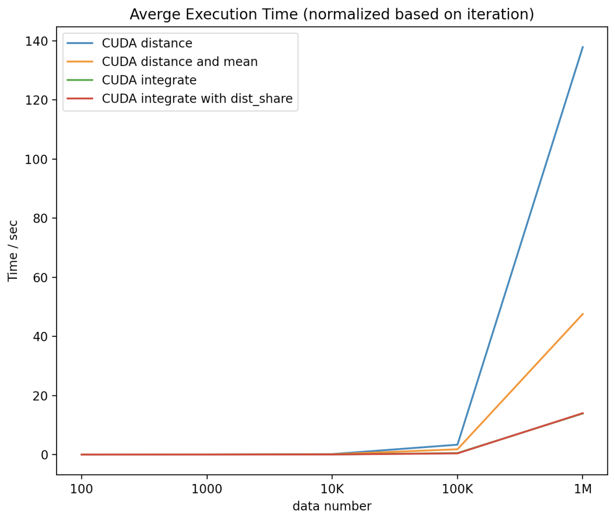


Fig 5: speed comparison between K-means using naive parallel distance calculation, K-means using naive parallel distance calculation and naive parallel mean calculation, K-means integrated method and K-means integrated method using parallel distance calculation

In Figure 5, the speed of K-means integrated method without parallel distance calculation and K-means integrated method using parallel distance calculation is close to each other so that the green line is covered by the red line.

We finally implemented the parallel scan calculation using shared memory on top of the parallel distance calculation using shared memory for the integrated method. Figure 6 shows the comparison of speed of all of the five methods.

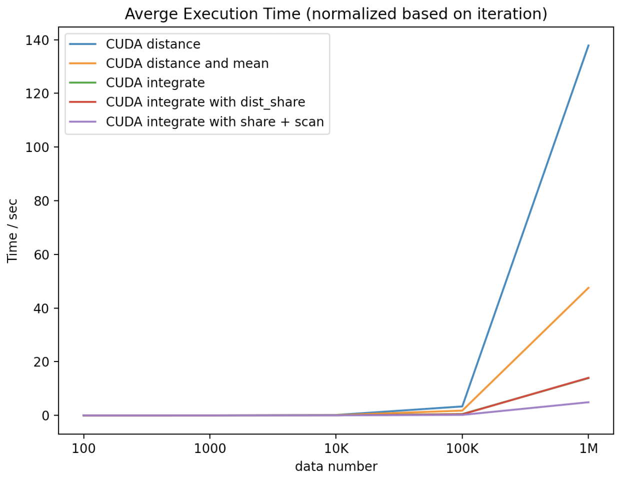
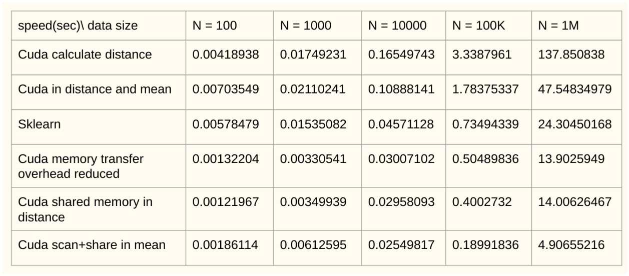


Fig 6: Speed comparison of the 5 implementation of K-means acceleration algorithm

Not surprisingly, the final implementation using parallel scan calculation using shared memory and parallel distance calculation using shared memory achieves the best performance. We also compared the performance of our implementations with the K-means algorithm provided by Sklearn. The following table shows the results.



speed(sec)\ data size	N = 100	N = 1000	N = 10000	N = 100K	N = 1M
Cuda calculate distance	0.00418938	0.01749231	0.16549743	3.3387961	137.850838
Cuda in distance and mean	0.00703549	0.02110241	0.10888141	1.78375337	47.54834979
Sklearn	0.00578479	0.01535082	0.04571128	0.73494339	24.30450168
Cuda memory transfer overhead reduced	0.00132204	0.00330541	0.03007102	0.50489836	13.9025949
Cuda shared memory in distance	0.00121967	0.00349939	0.02958093	0.4002732	14.00626467
Cuda scan+share in mean	0.00186114	0.00612595	0.02549817	0.18991836	4.90655216

Table 1: Speed comparison of 5 K-means acceleration implementations and Sklearn K-means

Our best solution (implementation of the parallel scan calculation using shared memory on top of the parallel distance calculation using shared memory for the integrated method) gives a nearly 5 times faster speed-up compared to the Sklearn method.

3.2 Profiling

We also profiled on our integrated implementation using CUDA Profiling using Nvidia Nsight Compute to see how long the execution of each portion of the code takes and how much memory the process is occupying at runtime. The following 4 figures show the profiling results of streaming multiprocessor (SM) utilization and memory utilization.

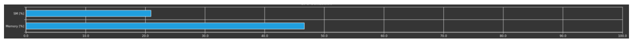


Fig 7: Profiling of parallel distance calculation without using shared memory

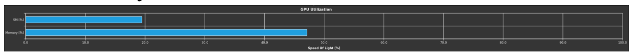


Fig 8: Profiling of parallel distance calculation using shared memory

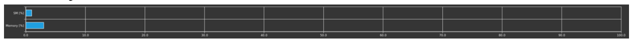


Fig 9: Profiling of parallel scan calculation without using shared memory

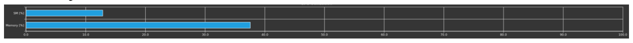


Fig 10: Profiling of parallel scan calculation using shared memory

From Figure 7 and Figure 8, we can see that using or not using shared memory does not impact the overall memory utilization, and using shared memory reduces the SM utilization a little. This also explains why in Figure 7 the speed of K-means integrated method without parallel distance calculation and K-means integrated method using parallel distance calculation is close to each other.

Figure 9 and Figure 10 shows the profiling results using or not using shared memory when doing parallel scan. We can see that using shared memory significantly increases the SM utilization and memory utilization. This proves the speed-up shown in Fig 8.

4. Discussion and Further Work

4.1 Discussion

Our results show a gradual improvement as we implement different optimization to the K-means algorithm. The results seem reasonable. It’s also encouraging that our best optimization beats the Sklearn K-means algorithm by speeding up 5 times when the number of data points comes to 1,000,000 with dimension of 8 and the cluster number is set to 5.

We have come up with other approaches to reduce the atomic operations used in mean calculation, including one method of updating only the data points that have changed assignment in the last iteration. Therefore, we avoid sum over the whole data set, therefore reduce the atomic operations. But due to unclear reasons, we have yet to get it running with consistent performance. We suspect precision of sum plays a part in this result, but we have to test more to be sure.

There are also some interesting facts when doing the experiments. First, the Nvidia GPU may optimize badly on pow(). When calculating the distance, we chose to use the pow() function but the result was not inspiring. We use the profiling tool and find that the distance calculation has a quite high SM utilization. After simply replacing the pow() function with simple multiplication, we achieved 6 times faster speed-up. The reason could be that the Nvidia GPU does not have optimization on the pow() function, while the computing power of each computing unit is poorer than the CPU.

Another interesting fact is that Sklearn.kmeans has overhead in initialization. When we first plotted the Sklearn.kmeans speed v.s. data size, we find that the speed at the first small data size is slow. After we do the experiments multiple times we find that this phenomenon only happens when first calling the Sklearn.kmeans function. So our solution is simply to do it 10 times and take the average as the final result.

4.2 Future Work

In the future, we will use libraries like thrust and CUBLAS to better optimize the distance calculation. Furthermore, we can change the algorithm design, for example, using a better centroids initialization to see whether the convergence speed can be faster.

5. Conclusion

In this work we proposed multiple parallel implementations for the K-means algorithm. Our solution exploits the computational power of the GPU and handles memory transfer efficiently. The result shows a significant speed-up. In the future, we are going to use matrix multiplication libraries and change the algorithm design to see whether we can achieve better performance.

6. References

[1] https://github.com/RayFan123/4750_project
[2] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. In Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2007.
[3] Elkan, Charles, "Using the Triangle Inequality to Accelerate K-Means." 2003 International Conference on Machine Learning ICML 03, 2003, pp. 147–153.
[4] James Newling and François Fleuret. "Nested mini-batch K-means". In Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16), 2016, pp. 1360–1368.
[5] J. Bhimani, M. Leeser and N. Mi, "Accelerating K-Means clustering with parallel implementations and GPU computing," 2015 IEEE High Performance Extreme Computing Conference (HPEC), 2015, pp. 1-6, doi: 10.1109/HPEC.2015.7322467.
[6] S. Karbhari and S. Alawneh, "GPU-Based Parallel Implementation of K-Means Clustering Algorithm for Image Segmentation," 2018 IEEE International Conference on Electro/Information Technology (EIT), 2018, pp. 0052-0057, doi: 10.1109/EIT.2018.8500282.

7. Appendices
Individual Student Contributions (in %)

Task	**% Zhejian Jin**	**% Ruilin Fan**
Overall	**50**	**50**
code	**50**	**50**
presentation	**50**	**50**
report	**50**	**50**