Parallel K-means Clustering Using CUDA

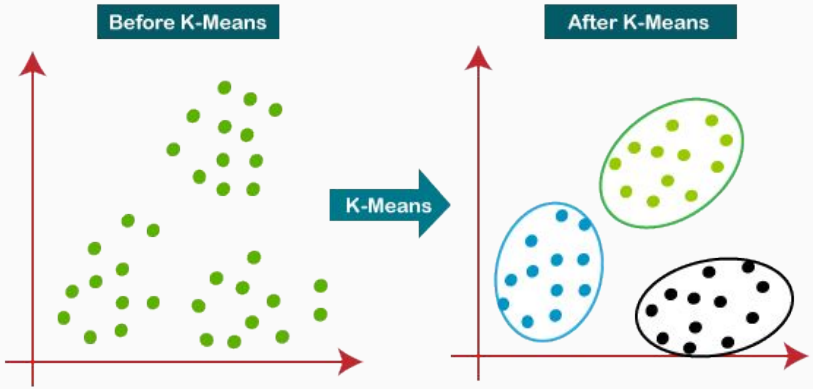
109062171 許育棠　109062113 葛奕宣

Table of contents

01 Problem Description

K-means Clustering Algorithm

An unsupervised machine learning algorithm popularly used in data analysis.
The goal is to group N data into K clusters so that the total within-cluster variation (or error) is minimized.
Data points can be in any dimension. We use 3D data points in our problem and use euclidean distance to calculate the distance.
k<=10, K <<< N

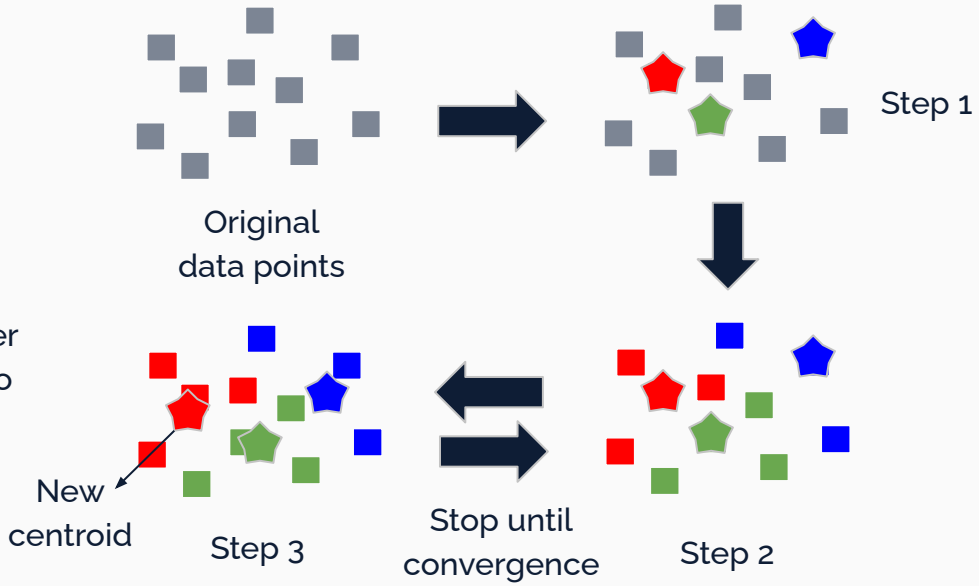


2D k-means clustering

01 Problem Description - Sequential

Algorithm Steps

1. Randomly choose K data points to be initial centroids.
2. Assign each data point to the closest centroid
3. Recalculate new centroids by finding the centroid of data points within each cluster
4. Repeat step 2 and 3 until the centroids no longer move



01 Problem Description - Sequential

Algorithm Steps

1. Randomly choose K data points to be initial centroids.
2. ***assign_cluster():***
   Assign each data point to the closest centroid
3. ***mean_recompute():***
   Recalculate new centroids by finding the centroid of data points within each cluster
4. ***check_modify():***
   Repeat step 2 and 3 until the centroids no longer move

For each point(N), calculate distance from each cluster(K) => O(N*K)

Sum up all points in each cluster and compute cluster mean => O(N+K)

For each point(N), check whether cluster modified => O(N)

Iterate T rounds,
Total time complexity => O(N*K*T)

02 Implementation - Kernel Functions

assign_clusters(step 2)

Threads per block=1024 Number of blocks=(N+1023)/1024
Let each thread manage the assignment of a single data point
Each thread:

1. Calculate Euclidean distance with all cluster => O(K)
2. Find the cluster with the shortest distance and assign to the data point => O(K)
3. Record whether the cluster is modified in device memory (**modified_record[N]**)

From O(N*K) to O(K) per iteration

02 Implementation - Kernel Functions

mean_recompute(step 3)

Sum_up_points():

Threads per block=1024 Number of blocks=(N+1023)/1024

Let each thread manage the summation of a single data point

Each thread:

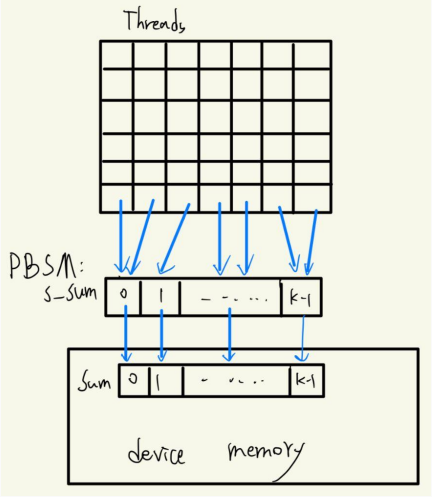
Sum the value to shared memory.

Time complexity => O(1+α), α denotes the overhead of atomic operation.

update_centroid():

Use K threads to calculate Sum[i]/count[i] and update the new centroid.

Time complexity => O(1)



From O(N+K) to O(α) per iteration

02 Implementation - Kernel Functions

Check_modified(step 4):

Threads per block=1024 Number of blocks=(N+1023)/1024

Each thread i:

Check if modified_record[i] == 1, update global var ***not_done*** to 1.

Use **atomic_or()** to set the value

Time complexity => O(1)

02 Implementation - Time Complexity

Algorithm Steps

1. Randomly choose K data points to be initial centroids.
2. ***assign_cluster():***
   Assign each data point to the closest centroid
3. ***mean_recompute():***
   Recalculate new centroids by finding the centroid of data points within each cluster
4. ***check_modify():***
   Repeat step 2 and 3 until the centroids no longer move

O(N*K) => O(k)

O(N+K) => O(1+α)

O(N) => O(1)

Iterate T rounds,
Total time complexity => O(T*(α+K))

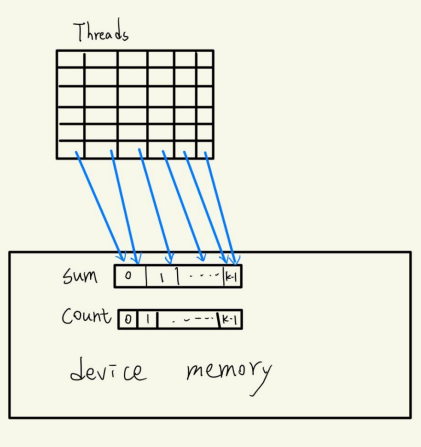
02 Implementation - Optimization

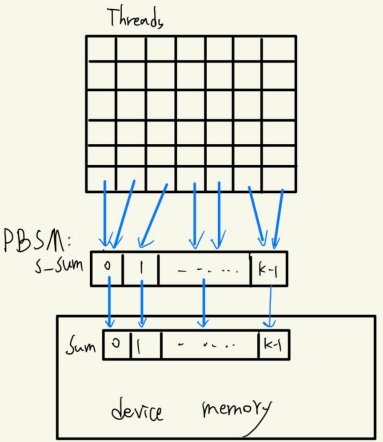
Race Condition in Sum_up_points()

Use **shared memory** and **Atomic Operation** to significantly reduce the overhead
Summation in share memory then write back to device memory

There's no atomicAdd() for double in RTX 1080

Cast double to unsigned long long and use **AtomicCAS()** to set the value





02 Implementation - Optimization

Euclidean Distance in assign_clusters

Sqrt is redundant
Replace C pow() with fast power
=> Time spend in ***assign_cluster()*** improve 4.4x

```
//function to calculate euclidea distance between two points
__device__ double euclid(Point a, Point b){
    double x = a.x- b.x;
    double y = a.y- b.y;
    double z = a.z- b.z;
    double dist = sqrt(pow(x, 2) + pow(y, 2) + pow(z, 2));
    return dist;
}
```

```
__device__ double euclid(Point a, Point b){
    double x = a.x- b.x;
    double y = a.y- b.y;
    double z = a.z- b.z;
    return fastPower(x, 2) + fastPower(y, 2) + fastPower(z, 2);
}
```

02 Implementation - Optimization

```
Time Taken: 1.127657
==105619== Profiling application: ./main 5 ../input/input_2000000.txt output_datapoints_2000000 output_centroid_2000000
==105619== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   80.38%  772.80ms       103  7.5029ms  7.1588ms  11.582ms  Points_Sum_Up(int, int, Point*, Point*, int*)
                   17.87%  171.83ms       104  1.6522ms  1.5908ms  1.7171ms  assign_clusters(Point*, Point*, int, int, double*, shor
t*)
                    0.66%  6.3893ms       106  60.276us     575ns  6.3242ms  [CUDA memcpy HtoD]
                    0.57%  5.4340ms       105  51.752us     640ns  5.3609ms  [CUDA memcpy DtoH]
                    0.47%  4.4777ms       103  43.473us  39.200us  199.23us  check_modify(short*, int*, int)
                    0.03%  265.60us       103  2.5780us  2.4640us  3.3280us  update_centroids(int, Point*, int*, Point*)
                    0.02%  176.42us       103  1.7120us  1.6310us  2.1120us  clear(Point*, int*)
      API calls:   85.79%  950.41ms       516  1.8419ms  1.5830us  11.584ms  cudaDeviceSynchronize
                   12.41%  137.52ms         7  19.646ms  2.4190us  137.22ms  cudaMalloc
                    0.62%  6.9048ms       208  33.196us  2.6560us  5.6711ms  cudaMemcpy
                    0.58%  6.4343ms         3  2.1448ms  5.5200us  6.4164ms  cudaMemcpyAsync
                    0.33%  3.6384ms         7  519.77us  2.6260us  1.8828ms  cudaFree
                    0.25%  2.7667ms       516  5.3610us  2.7320us  933.52us  cudaLaunchKernel
                    0.01%  128.65us       101  1.2730us     111ns  54.421us  cuDeviceGetAttribute
                    0.00%  9.8420us         1  9.8420us  9.8420us  9.8420us  cuDeviceGetName
                    0.00%  6.6700us         1  6.6700us  6.6700us  6.6700us  cuDeviceGetPCIBusId
                    0.00%  1.1080us         3     369ns     203ns     699ns  cuDeviceGetCount
                    0.00%     785ns         2     392ns     132ns     653ns  cuDeviceGet
                    0.00%     773ns         1     773ns     773ns     773ns  cuModuleGetLoadingMode
                    0.00%     376ns         1     376ns     376ns     376ns  cuDeviceTotalMem
                    0.00%     175ns         1     175ns     175ns     175ns  cuDeviceGetUuid
```

02 Implementation-Limitation

Limitation of device memory

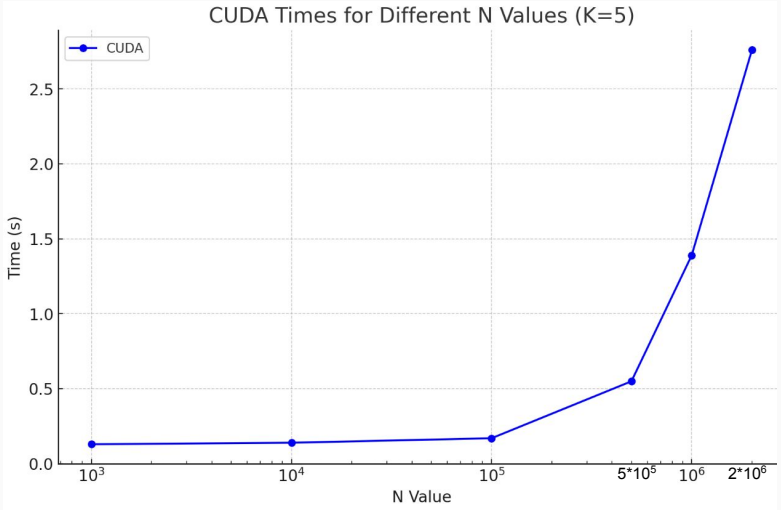
Total amount of global memory is 8507949056 bytes
We use 8NK+30N+60K+4 bytes of global memory
When k=10, N can be supported to 8507948452/110 = 77344985 (77 million)

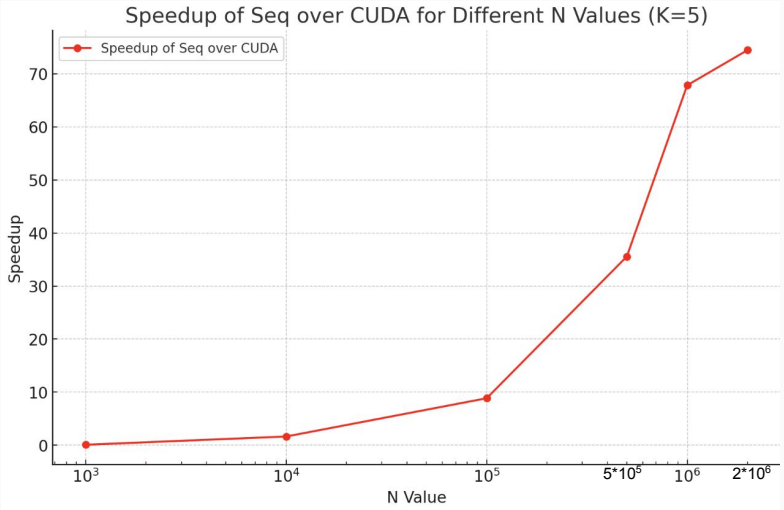
```
Device 0: "NVIDIA GeForce GTX 1080"
  CUDA Driver Version / Runtime Version          12.3 / 11.8
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 8114 MBytes (8507949056 bytes)
  (20) Multiprocessors, (128) CUDA Cores/MP:     2560 CUDA Cores
  GPU Max Clock rate:                            1835 MHz (1.84 GHz)
  Memory Clock rate:                             5005 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
```

03 Experiments-Benchmark

- System Spec: apollo and hades server
- Compare our implementation with sequential code and parallel code using Openmp
- Try different number of data points (N)
- Try different number of clusters (K)
- Test the time distribution of the program

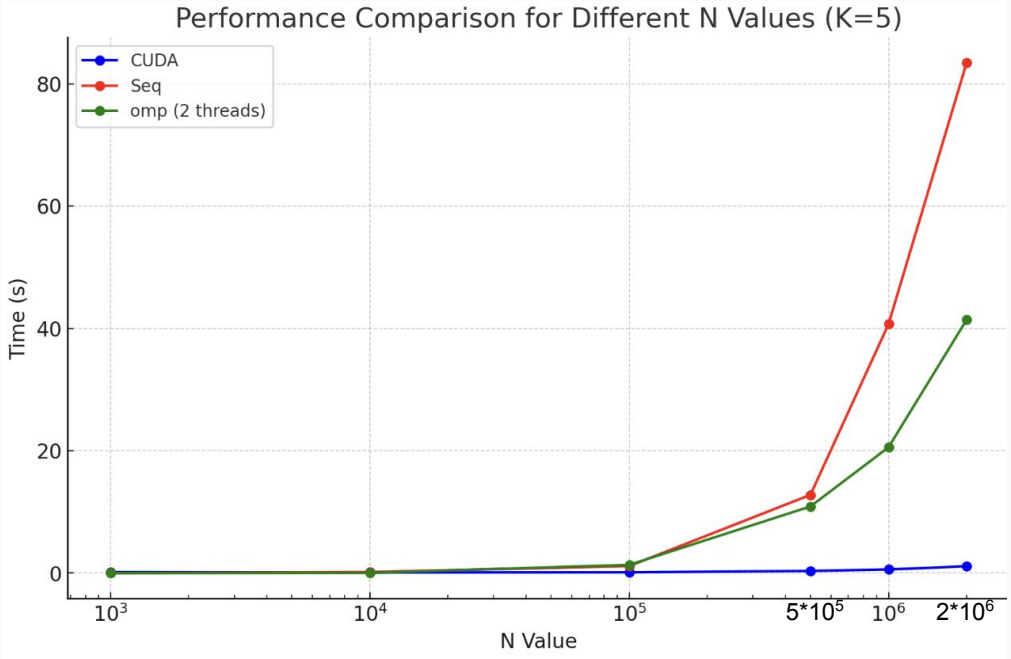
03 Experiments- Scalability





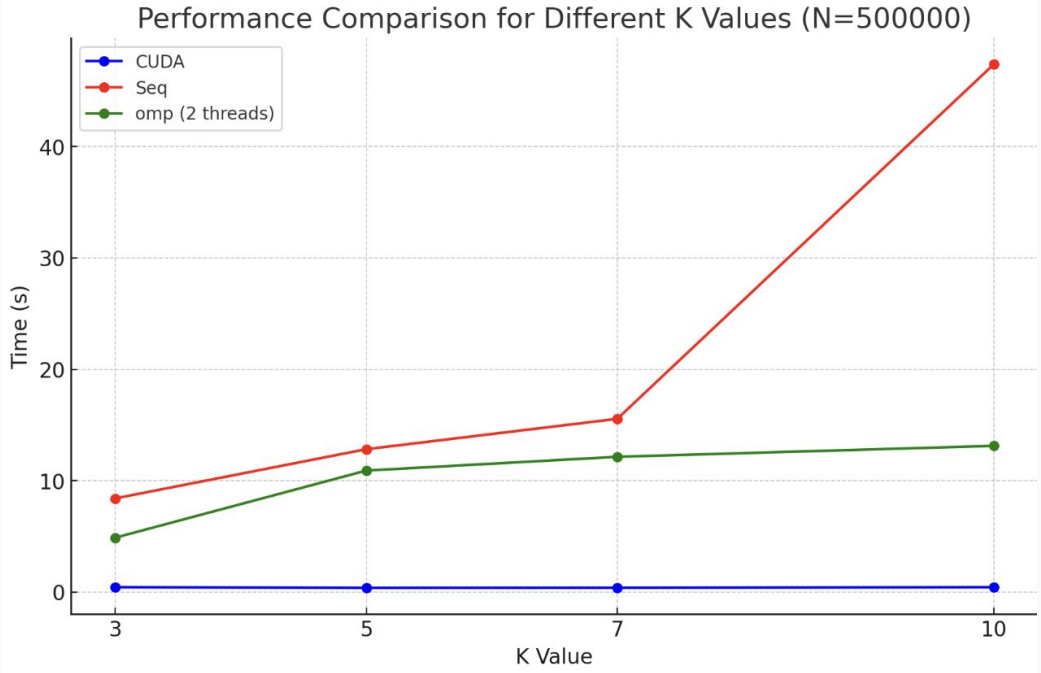
03 Experiments-Comp time Comparison

Same K
Different N

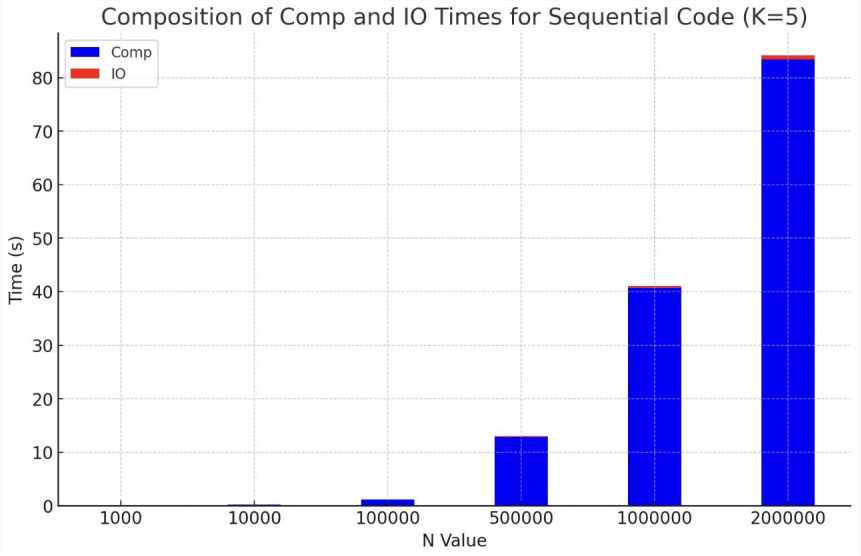


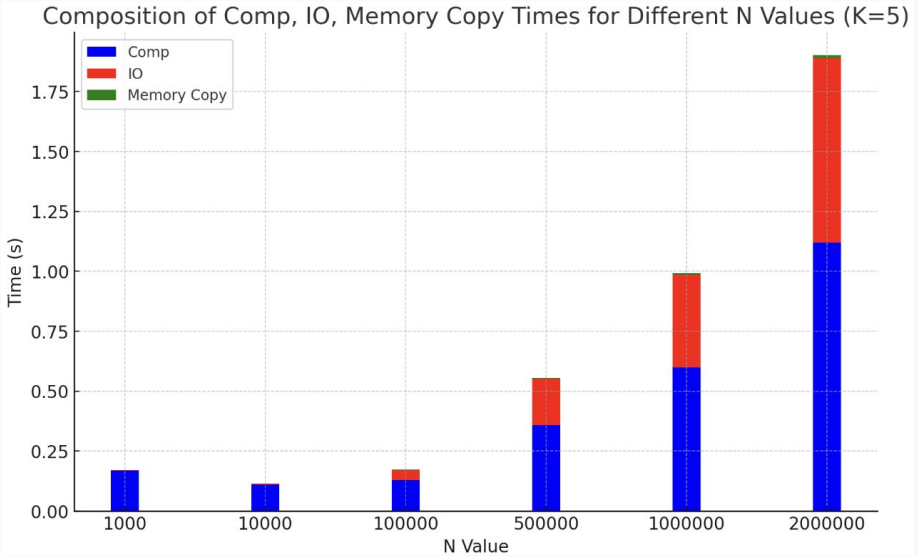
03 Experiments-Comp time Comparison

Same N
Different K



03 Experiments-CUDA Time Distribution





No longer bounded by computation time !

Conclusion

- Greatly speedup k-means clustering computation time for up to 74.46x in our experiment.
- In our implementation, the program is no longer bounded by computation time.
- Successfully parallelizing an algorithm gives us a sense of achievement

Thanks for listening