# K-means Clustering using CUDA

# Problem Description

- K-means is the clustering algorithm which assigns each datapoint to nearest cluster among K clusters.

- In this Project, we will parallelize following two steps of K-means algorithm using CUDA:

  1. Assignment of scalar data points to the clusters.
  2. Updating centroids.

  Each of the two steps are iterated until the cluster assignment converges  or upto certain number of iterations which will give the final result.

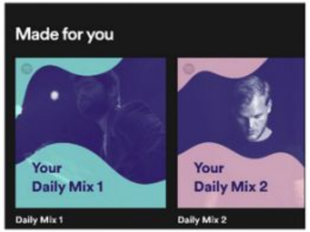
# Applications

- Document Clustering:
  - Clustering helps to group similar douments together

Document Clustering

- Image segmentation:
  - To club similar pixels in the image together we can apply clustering to create clusters having similar pixels in the same group.

- Recommendation Systems:
  - Clustering to find similar songs liked by a friend and finally recommend the most similar songs.



# K-means Algorithm

- K-means algorithm is iterative algorithm and is divided into two distinct, alternating steps: the *assignment* step and the *update* step.

- The **pseudocode** is as follows:
    1. Given cluster centroids **μi** initialized in some way,
    2. For iteration **t=1..T**
        1. Compute the distance from each point **x** to each cluster centroid **μ**,
        2. Assign each point to the centroid it is closest to,
        3. Recompute each centroid **μ** as the mean of all points assigned to it,

  where **T** is the number of iterations we wish to run this algorithm for. In each iteration, (1) and (2) are the assignment step and (3) is the update step.

- The time complexity of this approach is **O(n×k×Txd)** where d is dimension of data

# Expected Input and Output

**Inputs:**

N : Number of datapoints to be clustered

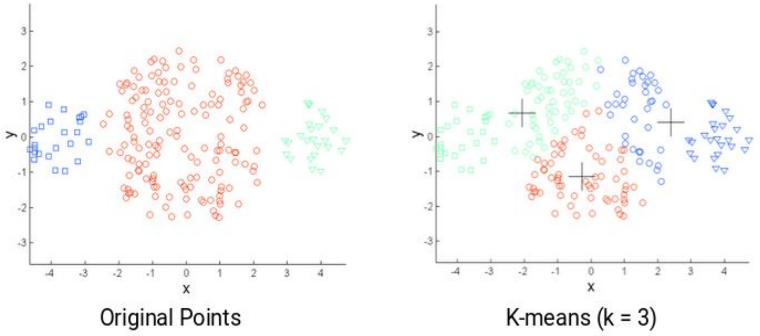
K : Number of clusters

Points : Set of datapoints which are randomly initialized

Iterations : Number of iterations for the algorithm to run for

**Output:**

Final Centroids/ Cluster Means : Final set of cluster centroids



# Output Screenshots - Serial and Parallel

Serial Code Output

```
For 10000 datapoints
For 10 clusters
and for 10 iterations
Serial Time required is 20.000000 ms
Final centroid: 0 :94.953766
Final centroid: 1 :67.764412
Final centroid: 2 :35.032383
Final centroid: 3 :46.871479
Final centroid: 4 :14.169797
Final centroid: 5 :77.689804
Final centroid: 6 :57.664452
Final centroid: 7 :24.251581
Final centroid: 8 :86.165039
Final centroid: 9 :4.526030
```

Parallel Code Output

```
Iteration 9: centroid 0: 19.446245
Iteration 9: centroid 1: 40.706402
Iteration 9: centroid 2: 100.300133
Iteration 9: centroid 3: 65.292175
Iteration 9: centroid 4: 12.755508
Iteration 9: centroid 5: 91.270485
Iteration 9: centroid 6: 20.131657
Iteration 9: centroid 7: 8.170594
Iteration 9: centroid 8: 2.825983
Iteration 9: centroid 9: 0.545912
Block Size: 32
Input Size: 1000
Number of clusters: 10
Iteration: 10
Time taken  is 1.394272 ms
```

# Parallelization strategy

- Assignment of data points to nearest cluster is parallelized by independently computing the distance for each datapoint. So each data point is handled per thread.
- We bring cluster assignments and data points to shared memory for centroid re-computations since global memory access is slow.
- Next Step is partitioning inputs into blocks and one thread in each block i.e thread 0 will add the data points to the corresponding clusters sums stored in local variables.
- Finally, we update the centroids in the global variables by using atomic add operations to avoid race conditions.
- Centroid recomputation step uses reduction operation for calculating sums.
- The Time complexity for the algorithm is **$O(n \times k \times T / p)$** where p is no. of processors.

# Profiling

For n = 10k

```
Time is 1.309344==27349== Profiling application: ./a.out
==27349== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   66.57%  178.11us         9  19.790us  19.167us  22.880us  kMeansCentroidUpdate(fl
                   14.53%  38.880us        18  2.1600us  2.1440us  2.1760us  [CUDA memset]
                   10.81%  28.927us         9  3.2140us  3.0400us  4.0960us  kMeansClusterAssignment
                    4.77%  12.768us         9  1.4180us  1.3440us  1.6960us  [CUDA memcpy DtoH]
                    3.31%  8.8630us         3  2.9540us  1.0560us  6.2710us  [CUDA memcpy HtoD]
      API calls:   97.26%  135.04ms         4  33.759ms  12.764us  134.75ms  cudaMalloc
                    1.35%  1.8790ms       188  9.9940us     390ns  419.98us  cuDeviceGetAttribute
                    0.32%  444.05us         4  111.01us  15.254us  227.59us  cudaFree
                    0.28%  394.38us        18  21.909us  15.212us  70.487us  cudaLaunch
                    0.24%  326.95us        12  27.246us  14.730us  41.620us  cudaMemcpy
                    0.18%  248.30us         2  124.15us  114.75us  133.55us  cuDeviceTotalMem
                    0.16%  215.56us         2  107.78us  93.920us  121.64us  cuDeviceGetName
                    0.14%  189.15us        18  10.508us  5.8780us  26.094us  cudaMemset
                    0.02%  23.892us        63     379ns     202ns  6.2840us  cudaSetupArgument
                    0.01%  20.306us         1  20.306us  20.306us  20.306us  cudaEventSynchronize
                    0.01%  16.364us         2  8.1820us  6.4600us  9.9040us  cudaEventRecord
                    0.01%  15.308us         2  7.6540us  5.3220us  9.9860us  cudaEventCreate
                    0.01%  11.232us        18     624ns     472ns  1.8980us  cudaConfigureCall
                    0.01%  7.4310us         4  1.8570us     548ns  5.1390us  cuDeviceGet
                    0.00%  4.4120us         3  1.4700us     542ns  2.9410us  cuDeviceGetCount
                    0.00%  3.9360us         1  3.9360us  3.9360us  3.9360us  cudaEventElapsedTime
```

For n = 1000k

```
==27677==  Time is 1.318880Profiling application: ./a.out
==27677== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   66.54%  177.47us         9  19.718us  19.200us  22.815us  kMeansCentroidUpdate(f
                   14.58%  38.880us        18  2.1600us  2.1120us  2.1760us  [CUDA memset]
                   10.81%  28.831us         9  3.2030us  3.0720us  4.0640us  kMeansClusterAssignmen
                    4.75%  12.672us         9  1.4080us  1.3440us  1.6640us  [CUDA memcpy DtoH]
                    3.32%  8.8640us         3  2.9540us  1.0560us  6.2720us  [CUDA memcpy HtoD]
      API calls:   97.32%  130.43ms         4  32.607ms  10.604us  130.12ms  cudaMalloc
                    1.27%  1.7029ms       188  9.0570us     310ns  401.06us  cuDeviceGetAttribute
                    0.33%  447.62us         4  111.90us  15.090us  233.04us  cudaFree
                    0.29%  392.31us        18  21.794us  14.838us  68.764us  cudaLaunch
                    0.25%  328.38us        12  27.364us  15.988us  42.824us  cudaMemcpy
                    0.16%  214.14us         2  107.07us  98.363us  115.78us  cuDeviceTotalMem
                    0.15%  200.31us        18  11.128us  5.6680us  38.114us  cudaMemset
                    0.15%  196.24us         2  98.119us  85.648us  110.59us  cuDeviceGetName
                    0.02%  26.980us        63     428ns     206ns  8.4340us  cudaSetupArgument
                    0.02%  25.616us         2  12.808us  10.044us  15.572us  cudaEventCreate
                    0.01%  19.828us         1  19.828us  19.828us  19.828us  cudaEventSynchronize
                    0.01%  17.160us         2  8.5800us  6.7180us  10.442us  cudaEventRecord
                    0.01%  10.912us        18     606ns     464ns  1.9100us  cudaConfigureCall
                    0.00%  6.6630us         4  1.6650us     423ns  4.7670us  cuDeviceGet
                    0.00%  4.0000us         3  1.3330us     515ns  2.6650us  cuDeviceGetCount
                    0.00%  3.3840us         1  3.3840us  3.3840us  3.3840us  cudaEventElapsedTime
```

# Results (Time in milliseconds)

| Input Size | K | Iterations | Serial | Parallel | Speedup |
|---|---|---|---|---|---|
| 10000 | 3 | 10 | 0 | 0.622 | 0 |
| 10000 | 3 | 20 | 10 | 1.2263 | 8.1546 |
| 10000 | 3 | 50 | 30 | 3.144 | 9.542 |
| 10000 | 3 | 100 | 60 | 6.242 | 9.612 |
| 100000 | 5 | 10 | 80 | 2.221 | 36.019 |
| 100000 | 5 | 20 | 150 | 4.17 | 35.971 |
| 100000 | 5 | 50 | 360 | 11.511 | 31.274 |
| 100000 | 5 | 100 | 670 | 23.17 | 28.916 |

# Results (Time in milliseconds)

| Input Size | K | Iterations | Serial | Parallel | Speedup |
|---|---|---|---|---|---|
| 10000 | 10 | 10 | 20 | 0.819 | 24.42 |
| 10000 | 10 | 20 | 30 | 1.498 | 20.026 |
| 10000 | 10 | 50 | 70 | 3.423 | 20.450 |
| 10000 | 10 | 100 | 140 | 6.806 | 20.57 |
| 100000 | 20 | 10 | 230 | 3.934 | 58.464 |
| 100000 | 20 | 20 | 430 | 7.997 | 53.77 |
| 100000 | 20 | 50 | 1100 | 20.068 | 54.813 |
| 100000 | 20 | 100 | 1850 | 40.29 | 45.917 |

# Results (Time in milliseconds)

| Input Size | K | Iterations | Serial | Parallel | Speedup |
|---|---|---|---|---|---|
| 1000000 | 3 | 10 | 450 | 15.769 | 28.537 |
| 1000000 | 3 | 20 | 540 | 33.204 | 16.263 |
| 1000000 | 3 | 50 | 850 | 85.547 | 9.936 |
| 1000000 | 3 | 100 | 1930 | 171.928 | 11.225 |
| 10000000 | 5 | 10 | 6200 | 172.98 | 35.842 |
| 10000000 | 5 | 20 | 12110 | 341.19 | 35.493 |
| 10000000 | 5 | 50 | 30870 | 840.06 | 36.747 |
| 10000000 | 5 | 100 | 59810 | 1673.83 | 35.732 |

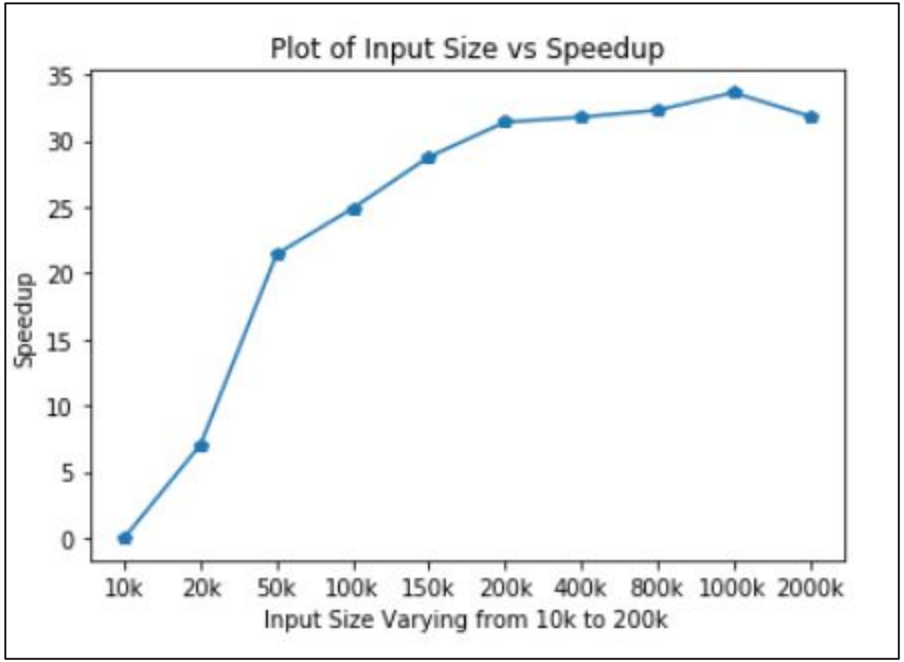
# Results (Time in milliseconds)

| Input Size | K | Iterations | Serial | Parallel | Speedup |
|---|---|---|---|---|---|
| 1000000 | 10 | 10 | 1240 | 22.72 | 54.577 |
| 1000000 | 10 | 20 | 2070 | 46.46 | 44.554 |
| 1000000 | 10 | 50 | 5170 | 117.44 | 44.022 |
| 1000000 | 10 | 100 | 10070 | 229.19 | 43.937 |
| 10000000 | 20 | 10 | 17250 | 327.67 | 52.644 |
| 10000000 | 20 | 20 | 33490 | 650.99 | 51.444 |
| 10000000 | 20 | 50 | 83290 | 1616.49 | 51.525 |
| 10000000 | 20 | 100 | 170150 | 3216.35 | 52.901 |

# Plots - Input Size vs Speedup

K = 3
Iterations = 10
Block Size = 32

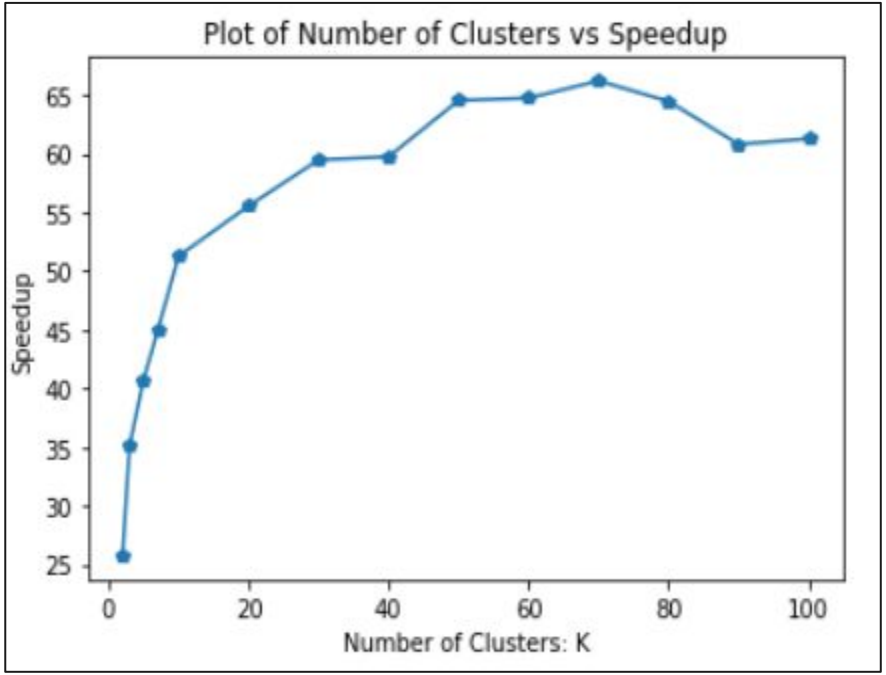
As the memory bottleneck hit in serial case with increasing number of N, and parallel computing doesn't have that memory bottleneck the increase in the speedup in the figure can be observed. After speedup of 34 at 1000k the threshold limit is reached, the decrease in speedup can be due to scheduling cost, atomic operations, and synchronization of threads in parallel computation.



# Plots - Number of Clusters vs Speedup

N = 100000
Iterations = 10
Block Size = 32

In this figure, as shown by the number of clusters increase the numbers of data-points also increase and thus speed up goes up. After 70 clusters the speedup reach threshold limit where it is consistent with the increase in number of data-points.



# Conclusion and Future Work

- We observe significant speedup as number of the clusters increases.
- Using shared memory helps in achieving speedup and also reducing memory access time.
- Use of atomic operations helps to avoid data race conditions but they are slow in nature since the atomic counter increment will be greatly contended and serialize all thread’s accesses.
- The Algorithm can further be optimized using various other strategies like Parallel Reduction can be used during centroid recomputation.
- Further optimization  could include using fully parallelised mean computation
- Generalise k means algorithm for N=(2, 3) dimensional data points

# References


- http://alexminnaar.com/2019/03/05/cuda-kmeans.html
- http://www.goldsborough.me/c++/python/cuda/2017/09/10/20-32-46-exploring_k-means_in_python,_c++_and_cuda/
- https://www.analyticsvidhya.com/blog/2019/08/comprehensive-guide-k-means-clustering/