# CS380P: Parallel Systems

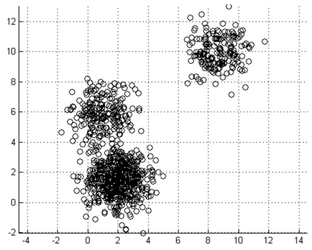
## Lab #2: KMeans with CUDA

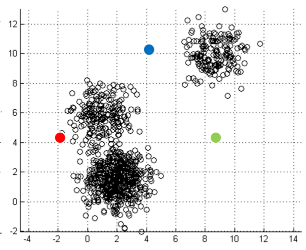
The goal of this assignment is exposure to GPU programming. You will solve k-means, using CUDA and Thrust.

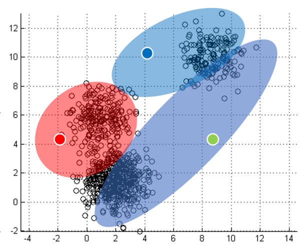
### K-Means

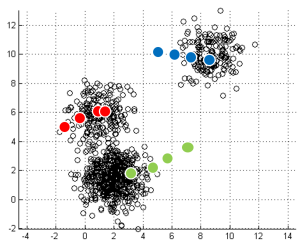
K-Means is a machine-learning algorithm most commonly used for unsupervised learning. Suppose you have a data set where each data point has a set of features, but you don't have labels for them, so training a classifier to bin the data into classes cannot rely on supervised algorithms (e.g. Support Vector Machines, which learn hypothesis functions to predict labels given features).

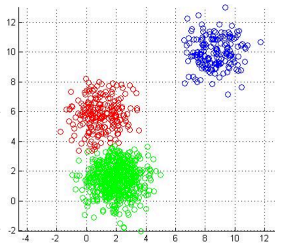
One of the most straightforward things we can do with unlabeled data is to look for groups of data in our dataset which are similar: clusters. K-Means is a "clustering" algorithms. K-Means stores *k* centroids that define clusters. A point is considered to be in a particular cluster if it is closer to that cluster's centroid than to any other centroid. K-Means finds these centroids by alternately assigning data points to clusters based on a current version of the centroids, and then re-computing the centroids based on the current assignment of data points to clusters. The behavior the algorithm can be visualized as follows:


Initial input


Choose three random centers


Map each point to its nearest centroid


New centroid is mean of all points mapping to it. Iterations move the centroids closer to their destination.


Centroids stop moving.
Each point labeled with its nearest centroid.

### The Algorithm

In the clustering problem, we are given a training set `x(1),...,x(m)`, and want to group the data into cohesive "clusters." We are given feature vectors for each data point x(i) encoded as floating-point vectors in D-dimensional space. But we have no labels `y(i)`. Our goal is to predict `k` centroids and a label `c(i)` for each datapoint. Here is some pseudo-code implementing k-means:

```
kmeans(dataSet, k) {

  // initialize centroids randomly
  numFeatures = dataSet.getNumFeatures();
  centroids = randomCentroids(numFeatures, k);

  // book-keeping
  iterations = 0;
  oldCentroids = null;

  // core algorithm
  while(!done) {

    oldCentroids = centroids;
    iterations++;

    // labels is a mapping from each point in the dataset
    // to the nearest (euclidean distance) centroid
    labels = findNearestCentroids(dataSet, centroids);

    // the new centroids are the average
    // of all the points that map to each
    // centroid
    centroids = averageLabeledCentroids(dataSet, labels, k);
    done = iterations > MAX_ITERS || converged(centroids, oldCentroids);
}
```

### For detailed instructions on how to write your code and submit it for evaluation, please see these instructions.

You should include a report that answers the following questions. In cases where we ask you to explain performance behavior, it is fine to speculate, but be clear whether your observations are empirical or speculation.

- Report the GPU hardware details, CPU hardware details, and OS version on the machine where you did your measurements if you measured in any environment other than Codio; if you just used Codio for measurements, it's fine to just report that fact.

- Which of your implementations is fastest? Does it match your expectations of which should be fastest? Estimate the best-case performance speedup your CUDA implementations should have based on the number of threads in your program and the number of processing contexts actually supported by your hardware. How far of that prediction is your best-case performance?
- Which of the parallel implementations is slowest, and does it match your expectations? Why or why not?
- What fraction of the end-to-end runtime in your CUDA versions is spent in data transfer?
- How much time did you spend on the lab?