Kmeans with CUDA

Fukun Lai

EID: fl6944

1 Background

This report discusses different implementation of K-means and compares their performance. K-Means is a machine-learning algorithm most commonly used for unsupervised learning. The algorithm has basically two major steps,

1. Find the nearest centroids for a data point;
2. Assign the data point to the nearest centroids;
3. Calculate the new centroids as the mean of all the data points in this cluster.

This could be described by the following pseude code:

```
kmeans(data, k) {
  // initialize centroids randomly
  numFeatures = dataSet.getNumFeatures();
  centroids = randomCentroids(numFeatures, k);

  iterations = 0;
  oldCentroids = null;

  // core algorithm
  while(iterations < MAX_ITERS && !converged(centroids, oldCentroids)) {

    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);
}
```

2 Hardware

All the test is running in Codio in this report.

3 Results and Discussion

The four implementations, sequential, thrust GPU, basic CUDA GPU, shared memory CUDA GPU have been implemented in this report. Since GPU process the date parallel, the speed of the four implementation should be increased by the order of sequential, Thrust, basic CUDA, shared memory CUDA. Thrust basically vectorizes the operations but with over-heads, while basic CUDA potentially removes those over-heads, and shared memory CUDA reduces access time to the global memory. Each of implementation has somehow improvement.

The actual test data proves the above speculation as show in Table 1, which shows the time measure for different implementation with 65536 points, 16 dimensions and 32 clusters with different implementations. It is observed that shared memory CUDA gives the best performance among all of the four implementation, and sequential gives the worst performance as is expected.

GPU parallel is mainly done on the clustering point step because it could be done parallel without much effort, and thus the performance of this part improved most significantly. However, the GPU parallel brings trade-off on data transfer, where the data has to transferred between GPU memory and CPU memory, this affect performance but it is only one time cost, i.e., we don't need to do it for every iteration.

Table 1. Time measure for different implementation with 65536 points, 16 dimensions and 32 clusters.

Implementation	Threshold	Total Iteration	Total Time per Iteration [ms]	Time per Iteration[ms]		Data transfer [ms]
				Clustering points	Update centroids	
Sequential	1.00E-06	89	164	152	12	0.3
Thrust	1.00E-06	89	15	10	2.8	198
Basic CUDA	1.00E-06	89	7.2	3.6	0.5	283
CUDA Shmem	1.00E-06	89	6.4	3.5	0.5	207

Figure 1 shows the comparison of the four implementation with different

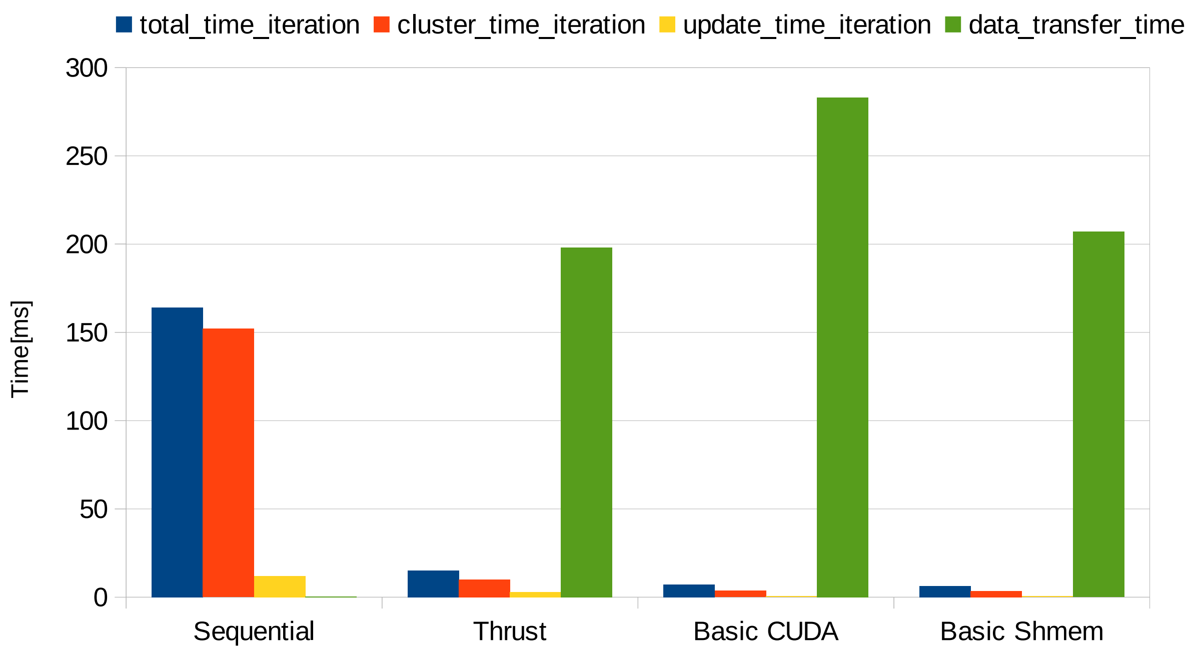

Figure 1: Performance of HS algorithm

4 Conclusion

This lab focuses on GPU programming, four implementations have been programmed and the results have been studied. It is found shared memory CUDA gives the best performance among the four, though it is only slightly better than basic CUDA.

I have enjoyed doing this lab, and learned a lot from it.

I spent in total about 30 hrs in this lab.

Appendix Additional command to run the code with different implementation

-a or –algo <algorithm>

seq: sequential implementation

thrust: thrust implementation

cuda: basic cuda implementation

cuda-shmem: shared memory implementation