K-Means Clustering on GPU

Chandini Shetty
Dept. of Computer Engineering,
University of California, Riverside

Reeta Kumari
Dept. of Computer Science
University of California, Riverside

***Abstract*— This project looks at a parallel implementation of the popular k-means clustering algorithm using the NVIDIA's CUDA Programming technology. K-means is a machine learning clustering algorithm that is used widely in solving important scientific research problems and in commercial applications. It is ubiquitous because of its simplicity and being an unsupervised learning algorithm. K-means is typically run on large datasets and hence is very computational intensive. Also the computation is iterative in nature and can be performed on independent data points, and hence it exhibits characteristics suitable for parallel implementation. In this paper we look at a parallelized CUDA-based implementation of the k-means algorithm on an CUDA Supported GPU. Our implementation primarily focuses on efficient use of single thread to perform work on more than one data point and an improved reduction which is a key step in k-means clustering algorithm. We show improvement in the overall CUDA kernel execution with these enhancements.**

***Index Terms*—Parallel computing, Machine learning, K-means,CUDA, GPU Computing**

I. Introduction

Clustering of large datasets is an important use case in data analytics and scientific research to understand hidden data patterns. Many algorithms exist to do this but the simplest one to understand and use is the k-means algorithm. From computational intensity point, it is easy to implement, since it requires to choose a set of random cluster centers, compute the distance of the points to these centers and assign them based on minimum distance to it. Then the cluster centers are recalculated based on the data points assigned to each center. This is repeated until fair amount of convergence is achieved in terms of the points not changing cluster membership. Although the problem is NP-hard in the sense, it is hard to achieve the optimum cluster centers, a good local optimum is usually enough to satisfy the use case. And usually a good choice of initial cluster centers instead of randomly chosen ones, gives good result.

Sequential K-means Algorithm

The sequential implementation of the algorithm is fairly simple. We start with co-ordinates of the data points. The nature of the dataset should be such that these co-ordinates or attributes are numerical in nature, since k-means calculates the average. We cannot use k-means on data that has categorical attributes. We calculate the distance of the 'n' data points to a set of randomly initialized 'K' cluster centers. Here 'K' is an input parameter from the user specifying the number of clusters that needs to be created. The distance is calculated for 'm' dimensions (or co-ordinates) of the dataset and this is typically the Euclidean distance or squared distance. Each data point computes it's distance to each of the 'K' centers and assigns itself to the center to which it has the minimum distance. At the end of the first iteration we would have the 'n' data-points partitioned into 'K' clusters. The next step would be to calculate the new centers based on the average distances to current cluster center. This repeated till we have the number of points changing cluster comes down to a tolerable limit set by the user (usually a specified threshold). The final output of the program is the set of K cluster center co-ordinates and the membership detail of each input points.

From time complexity perspective the algorithm will have exponential running time in worst case i.e. when the number of input data-points is large. Hence we look at a parallelized approach. Note that since the data-points compute the distance to the 'K' centers independently, this makes the algorithm highly parallelizable

Benchmark sequential K-means Implementation

For the purposes of this project, we have borrowed an existing sequential k-means implementation by Prof. Wei-keng Liao, ECE Department, Northwestern University [1]. This is an efficient sequential implementation and we use it to benchmark against all further implementation of our parallel algorithm. We also used this implementation as a golden to compare the accuracy of the results obtained- which is the cluster centers co-ordinates.

II. Our Contributions

We implement our parallelized k-means algorithm using CUDA, which is a programming model created by Nvidia and can run on a CUDA enabled Graphics Processing Unit(GPU).

We build our implementation on two key CUDA kernel functions- *findCluster* and r*educeCluster* with input data points being two dimensional in space. Below is a description of the key ideas of the code implementation.

The first kernel findCluster will be the function that does majority of the work, which is to calculate the cluster centers and assigning the input data points to the cluster. Each thread executing the kernel loads two input data point's co-ordinates into shared memory arrays (thread coarsening concept). The first set of randomly initialized cluster centers are stored in constant

memory, since these never change for the first iteration and all the threads need to access this data continuously which implies high cache hits and hence constant memory is most suitable. We also load the cluster points ID's into shared memory array which is needed to identify the point. All further calculations are now done using the arrays in shared memory. Next we compute the min-distance from the cluster centers and store these in the previously used shared memory arrays. This intermediate result is needed can be reused in further steps and saving it in already allocated shared memory array eliminates additional memory load/store operations. We eventually perform reduction on each shared memory data and the results of each reduction is copied to a temporary array. This array holds all partial reductions from each thread block. Next we invoke a final reduction (reduceCluster) which on the partial accumulated reduction and this is used to obtain the average (new center of the cluster). These two kernels will be used to calculate new center co-ordinates for each cluster and will be repeated till coordinates of centers from the current and previous iteration do not match till a precision of 0.01, as we used float as data type because CUDA doesn't support double. Below is the code snippet of the *findCluster* kernel function,

```
__global__ void findCluster (float* xarray, float* yarray, float* tempx, float* tempy, int* countP, int
numPoints, int numClusters, int clusid){
//Loading input coordinates into two shared memory arrays
  shared   float sx[2*BLOCK_SIZE];.
.....
sx[threadIdx.x] = (index1 < numPoints) ? xarray[index1] : 0.0;
.....
  syncthreads();
//Calculating dist of each point to this cluster center
float distMin = (sx[threadIdx.x] - cx_cm[clid])*(sx[threadIdx.x] - cx_cm[clid]) + (sy[threadIdx.x] -
cy_cm[clid])*(sy[threadIdx.x] - cy_cm[clid]);
...
//Loop that calculates the minimum distances to cluster centers
for(int k = 1; k < numClusters; ++k)
{
 distTemp = (sx[blockDim.x * threadIdx.x] - cx_cm[k])*(sx[blockDim.x * threadIdx.x] - cx_cm[k]) +
(sy[blockDim.x * threadIdx.x] - cy_cm[k])*(sy[blockDim.x * threadIdx.x] - cy_cm[k]);
  if(distTemp < distMin)
  {
   distMin = distTemp;
   ind = k;
  }
}
// Summing the value of points and number of points with the help of reduction
for(int stride = blockDim.x; stride > 0; stride >>= 1)
{
  __syncthreads();
  if(threadIdx.x < stride)
  {
   sx[threadIdx.x] +=  sx[stride + threadIdx.x];
.....
//Copy the output of each block level reduction into tempx ,tempy array that will be given to a final
reduction kernel.
__syncthreads();
tempx[blockIdx.x] =  sx[0];
 .....
}
```

Some of the key design decisions that we have considered before implementing in CUDA is the data management. One of the first observations is that the input dataset consisting on 'n' points of 'm' dimensions is large and this is based on how typical real world use case would be. So this data would be stored in the global memory/device memory. The input data is in a simple text file and is read and copied by host code into the global memory. Each thread of the *findCluster* kernel will load one or more elements from the global memory into smaller sized shared memory arrays. In our implementation we have used thread blocks of size 256 threads (in x dim) since this is most suited for the GPU specifications of our test equipment.

Next we focus on improving the reduction algorithm used, since it is frequently repeated step by every thread block and repeats for every iteration. The reduce steps that calculates the new cluster centers is implemented within the *findCluster* kernel function, since here we are able to reuse the data already available in shared memory arrays. Also in addition to strided index and sequential addressing in reduction kernel, as further improvement we also unroll the last warp which eliminates the need for __syncthreads() since instructions are synchronous within the warp.[2 Reduction article by Mark Harris] As a result of this we see further improvement in the kernel execution time.

In addition to this, we have incorporated improvement suggestions for instruction optimizations based on the CUDA Best practices guide, specifically in the Arithmetic Areas. We have used bit shift operators instead of division, explicit multiplying instead of exponentiation and signed integers for loop counters.

III. Testing and Optimizations

A. Test data size:

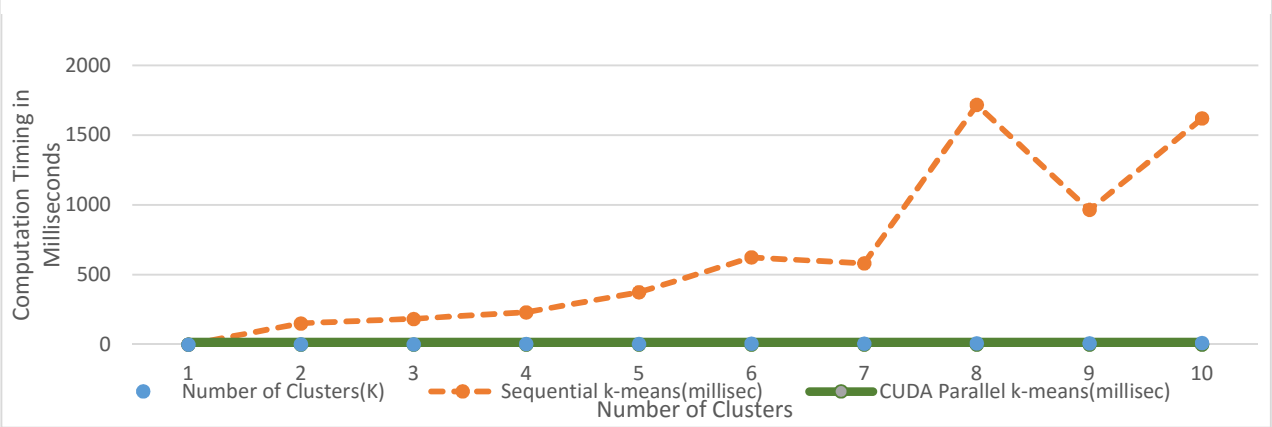
Our primary use case is to profile and analyze the implementation against a variety of input data sizes. To meet this purpose, we used actual input data sets from the Machine learning repository hosted by UC Irvine[3]. Here we used dataset Iris and Image segmentation. The Iris dataset is a smaller dataset and we were not able to distinguish amongst the sequential and parallel implementation significantly in terms of execution time since they were comparable. So we used this only as a measure of accuracy between the sequential and our parallel implementation. The Image_segementation dataset is comparatively a larger data set and we extracted only the histogram category. This particular dataset has about 64080 input point and with 2 dimensions. We base all our future results with this test data as our baseline since we see significant differences in the kernel execution time for this larger dataset.

B. Cluster size as a parameter:

Second parameter of our Profiling test is the number of clusters to be created. Here we observe our kernel execution time- specifically for the findCluster kernel- and observe its execution time variation as we increase the number of clusters to be created linearly from 1 to 11, in unit size. Plots showing this comparison are added in further sections.

Our test setup consists of Dual Intel(R) Xeon server, having a Nvidia Tesla C2075 with Compute capability:2.0.

Figure 1: Graph depicting the comparison of computation timing of Sequential and Parallel CUDA k-means.



Number of Clusters(K)	Sequential k-means(millisec)	CUDA Parallel k-means(millisec)
2	44.9	0.02124
3	152	0.02389
4	184.7	0.02506
5	232.2	0.02749
6	374.9	0.02866
7	625.4	0.03103
8	582.2	0.03217
9	1717.1	0.03455
10	965.9	0.03571
11	1620.3	0.03806

Table 1: Comparison of Sequential Implementation and CUDA parallel k-means

IV.RESULTS

As a first test of the code, when profiled with Visual Profiler there was no indication of any Memory coalescing and branch divergence issues

We go on to compare our implementation with that of the sequential k-means described earlier. Table 1. is comparison of the computation timing of the sequential code and our parallel kernel execution time. Here we see as speedup of 3 orders of magnitude in the parallel implementation. Note this is only computation time and IO/data transfer is not accounted for both sequential and our implementation.

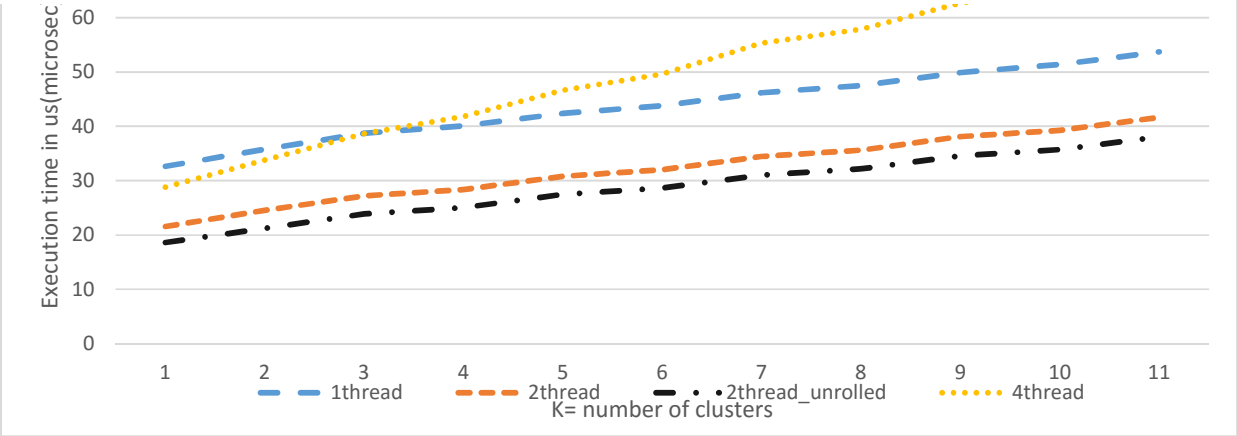
Next we provide a comparison of the kernel execution times, for several variations of our implementation that we have tried. First we would like to describe the 4 different kernel implementations tried out for the findCluster kernel. In implementation 1, which would be further referenced by name 1thread_findCluster, we use 1 thread to work upon only 1 input data point. Each thread within the grid of threads works on calculating only its distance to the K set of current cluster centers. In the second implementation- 2thread_findCluster, we use 1 thread to work upon 2 elements of the input dataset. Everything else remains the same as above. For the third implementation 2thread_unrolled, we unroll the last warp completely in two of the reduction steps. And in the 4th implementation we use 1 thread to work upon 4 input data elements. Based on our test results we find that the 2thread_unrolled (implementation 3) achieves the best timing results.

Below are the tabularized results for each implementation for varying cluster sizes. The Average execution time for findCluster kernel is depicted in Table 2 in us(microseconds)

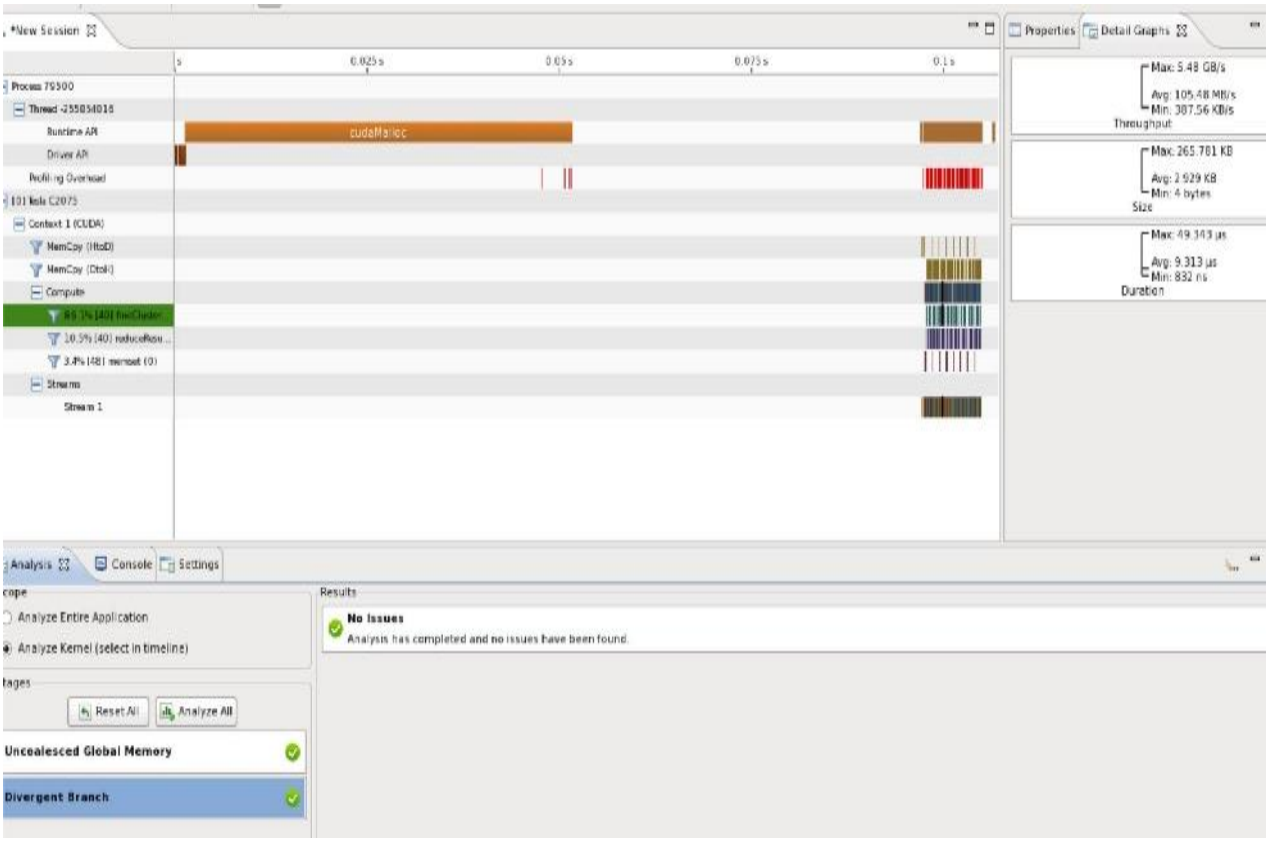
Clusters(K)	1thread	2thread	2thread_unrolled	4thread
1	32.60s	21.54us	18.61us	28.79us
2	35.80us	24.53us	21.24us	33.77us
3	38.71us	27.23us	23.89us	38.68us
4	40.07us	28.37us	25.06us	41.83us
5	42.39us	30.79us	27.49us	46.61us
6	43.79us	32.02us	28.66us	49.59us
7	46.20us	34.46us	31.03us	55.33us
8	47.54us	35.63us	32.17us	57.88us
9	49.90us	38.08us	34.55us	62.70us
10	51.37us	39.27us	35.71us	65.66us
11	53.70us	41.64us	38.06us	70.76us

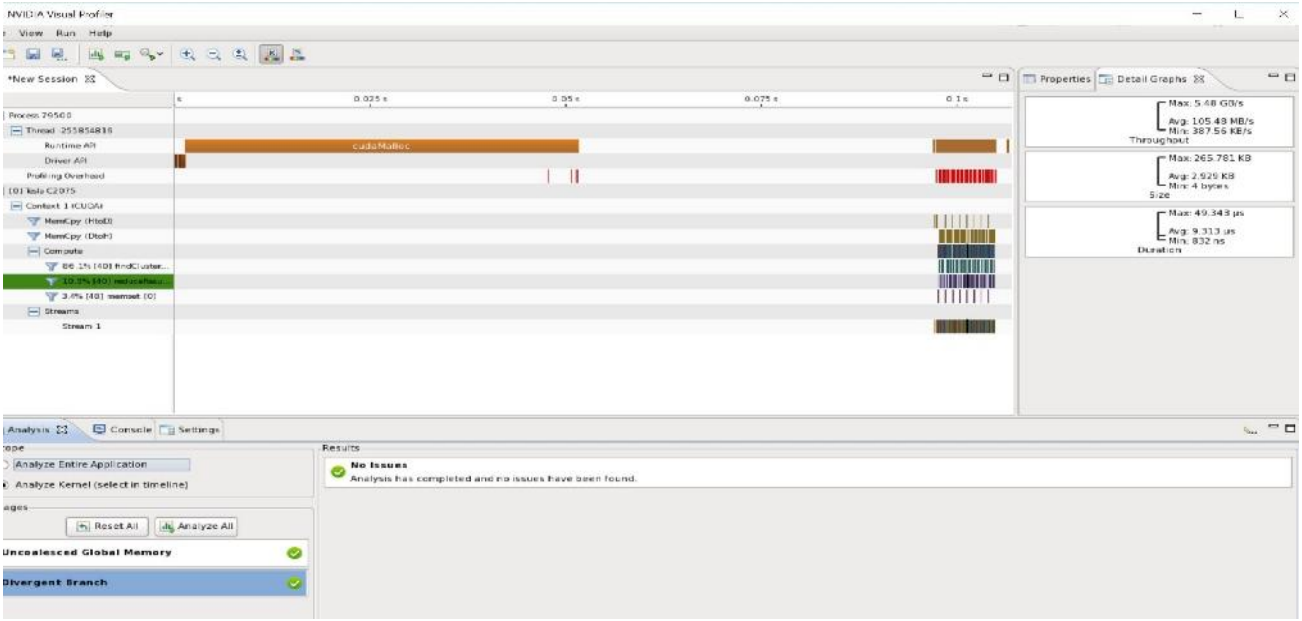
Table 2: findCluster execution time for the four different implementations

Figure 2: Comparison of the findCluster execution time for the four different implementations



Memory Coalescing and Branch Divergence results from Visual Profiler for findCluster and reduceCluster:





V. CONCLUSION

As machine learning algorithms and clustering analysis become more widely used, a fast implementation of clustering algorithms like k-means brings a lot of value to the application requiring it. By effective use of thread coarsening techniques, leveraging CUDA shared memory and improved reduction we are able to significantly improve the sequential algorithm. As a future work, we look at extending this implementation for data points that are represented in the ‘m’ dimensions.

***Acknowledgments**:*

We would like to thank Prof. Nael Abu-Ghazaleh, Computer Science and Engineering Department, University of California Riverside for his valuable guidance and feedback during the course of this project.

REFERENCES

[1] Sequential K-means implementation by Professor Wei-keng Liao:http://users.eecs.northwestern.edu/~wkliao/Kmeans/index.html

[2] MarkHarris.https://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf.

[3] UC Irvine Machine Learning repository (archive.ics.uci.edu/ml/datasets. html).