CSE 590 Final Project Report
Parallel GPU based Implementations of K-means Clustering

Aadarsh Jajodia, Akshit Poddar, Shahzeb Patel

Aim: Implementation of parallel k-means clustering algorithm on CUDA and OpenCL and do their comparison with the sequential approach to analyze the speedup achieved

Summary — The k-means algorithm is widely used for unsupervised clustering. Our project describes an efficient parallel k-means algorithm. Different from existing GPU-based k-means algorithms, and we have used the algorithm implemented in http://www.know-center.tugraz.at/download_extern/papers/latex8.pdf. Emphasis is placed on optimizations directly targeted at this architecture to best exploit the computational capabilities available. The algorithm is realized in a hybrid manner, parallelizing distance calculations on the GPU while sequentially updating cluster centroids on the CPU based on the results from the GPU calculations. We have also made use of other parallel k-means clustering algorithms papers and referenced them and have referenced them below. Section 4 highlights the results of our empirical evaluations of CUDA and OpenCL implementations against sequential implementations.

Note: Most of the content of the literature in this report is credited towards the main paper that we have used and the reference papers mentioned in the end. We have presented the matter here inorder to present a holistic view of the report. Our main aim has been the implementation of the approach in the paper mentioned above in CUDA and OpenCL, and compare them against sequential approach.

Implementation reference: We have used the raw skeleton code, mostly to read input files and create cluster data structure from the project of a professor from the following reference: http://users.eecs.northwestern.edu/~wkliao/Kmeans/index.html

1. Introduction and Background

Clustering is a method of unsupervised learning that partitions a set of data objects into clusters, such that intra-cluster similarity is maximized while inter-cluster similarity is minimized. The k-Means algorithm is one of the most popular clustering algorithms and is widely used in a variety of fields such as statistical data analysis, pattern recognition, image analysis and bioinformatics. It has been elected as one of the Top 10 data mining algorithms. The running time of k-means algorithm grows with the increase of the size and also the dimensionality of the data set. Hence clustering large-scale data sets is usually a time-consuming task. Parallelizing k-Means is a promising approach to overcoming the challenge of the 3 huge computational requirement. In P-CLUSTER has been designed for a cluster of computers with a client-server model in which a server process partitions data into blocks and sends the initial centroid list and blocks to each client.

It has been further enhanced by pruning as much computation as possible while preserving the clustering quality. In the k-means clustering algorithm has been parallelized by exploiting the inherent data-parallelism and utilizing message passing.

With the appearance of programmable graphics hardware in 2001, using the GPU as a low-cost highly parallel streaming co-processor became a valuable option. In the following years scientific interest in this new architecture resulted in numerous publications demonstrating the advantages of GPUs over CPUs when used for data parallel tasks. Much attention was focused on transferring common parallel processing primitives to the GPU and creating frameworks to allow for more general purpose programming. The most problematic aspect of this undertakings was transforming the problems at hand into a graphics pipeline friendly format, a task needing knowledge about graphics programming. The reader shall be referred to where an in-depth discussion on mapping computational concepts to the GPU can be found. This entry barrier was recently lowered by the introduction of NVIDIA's CUDA as well as ATI's Close to Metal Initiative. Both were designed to enable direct exploitation of the hardware's capabilities circumnavigating the invocation of the graphics pipeline via an API such as OpenGL or DirectX. In this work CUDA was chosen due to its more favorable properties, namely the high-level approach employed by its seamless integration with C and the quality of its documentation.

2. Related Work

To the best of the authors' knowledge, three different implementations of k-means on the GPU exist. All three implementations are similar to the parallel k-means implementation outlined in section 3.3 formulated as a graphics programming problem. In Takizawa and Kobayashi try to overcome the limitations imposed by the maximum texture size by splitting the data set and distributing it to several systems each housing a GPU. A solution to this problem via a multi-pass mechanism was not considered. Also the limitation on the maximum number of dimensions was not tackled. It is also not stated whether the GPU implementation produces the same results as the CPU implementation in terms of precision. Hall and Hart propose two theoretical options for solving the problem of limited instance counts and dimensionality: multi-pass labeling and a different data layout within the texture. None of the approaches have been implemented though. In addition to the naive k-means implementation the data is reordered to minimize the number of distance calculations by only calculating the metrics to the nearest centroids. This is achieved by finding those centroids by traversing a previously constructed kd-tree.

The authors could not observe any problems caused by the non standard compliant floating point arithmetic implementations on the GPU, stating that the exact same clusterings have been found. The approach differs in that the centroid indices are stored in an 8-bit stencil buffer instead of the frame buffer limiting the number of total centroids to 256. Limitations in dimensionality and instance counts due to maximum texture sizes are solved via a costly multi-pass approach. No statements concerning precision of the GPU version were made. Summarizing the presented previous work, the following can be observed: All implementations suffer from architectural constraints such as maximum texture size limiting the number of instances, dimensions and clusters. The limitations can only be overcome by employing more costly multi-pass approaches.

3. K-MEANS CLUSTERING

A. Algorithm

K-Means is a commonly used clustering algorithm used for data mining. Clustering is a means of arranging n data points into k clusters where each cluster has maximal similarity as defined by an objective function. Each point may only belong to one cluster, and the union of all clusters contains all n points. We describe Lloyd's fast algorithm for computing an approximate solution to the K-means problem. The algorithm assigns each point to the cluster whose center (also called centroid) is nearest. The center is the average of all the points in the cluster that is, its coordinates are the arithmetic mean for each dimension separately over all the points in the cluster. The algorithm steps are:

1) Choose the number of clusters, k.

2) Randomly generate k clusters and determine the cluster centers, or directly generate k random points as cluster centers.

3) Assign each point to the nearest cluster center.

4) Re-compute the new cluster centers.

5) Repeat the two previous steps until some convergence criterion is met (usually that the assignment hasn't changed).

The main disadvantage of this algorithm is that it does not yield the same result with each run, since the resulting clusters depend on the initial random assignments. It minimizes intra-cluster variance, but does not ensure that the result has a global minimum of variance [5]. If, however, the initial cluster assignments are heuristically chosen to be around the final point, one can expect convergence to the correct values. proposes an optimization by choosing initial centroids close to existing clusters. The first and second phases of the algorithm take $\Theta(k)$ time, while the third phase takes $\Theta(nk)$ to complete. Finally, the fourth phase's execution time is in the order of $\Theta(n + k)$. In a typical application $n>k$, therefore the execution time of the algorithm is bound by the third phase. This observation directed us to concentrate our efforts on parallelizing the third phase of the k-means algorithm.

B. Parallel K-means clustering

Parallel K-Means: In a parallel implementation of kmeans on distributed memory multiprocessors. The labeling stage is identified as being inherently data parallel. The set of data points X is split up equally among p processors, each calculating the labels of all data points of their subset of X . In a reduction step the centroids are then updated accordingly. It has been shown that the relative speedup compared to a sequential implementation of k-means increases nearly linearly with the number of processors. Performance penalties introduced by communication cost between the processors in the reduction step can be neglected for large n.

A brief description of the parallelization implementation is described below:

1. The main idea was to parallelize the labelling of the data objects into its respective clusters, and the computation of delta needed to determine the convergence.
coalescing, easy access and reduce bank conflicts.

4. Experimental Evaluation

We divide the evaluation in two parts. First we compare the implementation with CUDA with the sequential implementation on CPU. The input is varied for dimensions of 2, 20, 200 and of input size 5000, 50000, 100000. For all such input sizes, the number of cluster varies from 2 to 128 with incremental powers of 2.

The input provided to the parallel algorithm is multi dimensional points with dimensions represented as Single-precision floating-point.

eg: Following are 5 points with 10 dimensions

1 1.138306 -1.215842 0.555809 1.026545 -2.430758 4.322971 -1.217299 -2.184030 -1.388817 0.166098

2 -2.876999 -4.873641 1.715104 4.581660 4.803759 0.744806 2.249417 1.776730 1.125599 -3.469210

3 -2.077210 1.499280 -4.915942 -2.140810 3.764414 -0.272780 -4.171116 0.006001 -1.817014 -0.969820

4 -3.276886 4.321291 2.814338 2.278923 0.347836 -4.616421 1.601894 4.130538 -1.800450 -4.786923

5 -0.703364 0.322551 -4.660563 -3.988260 -0.095789 -4.856805 1.756546 -2.846372 1.919924 -2.117855

The source code tarball also contains program to generate input based on given

First we compare the sequential run time with the CUDA parallel implementation. The number of vertices are 5000, 50000, 10000 and dimensions are of 2, 20, 200. The number of cluster range from 2 to 128 with increasing power of 2. The results are as follows.

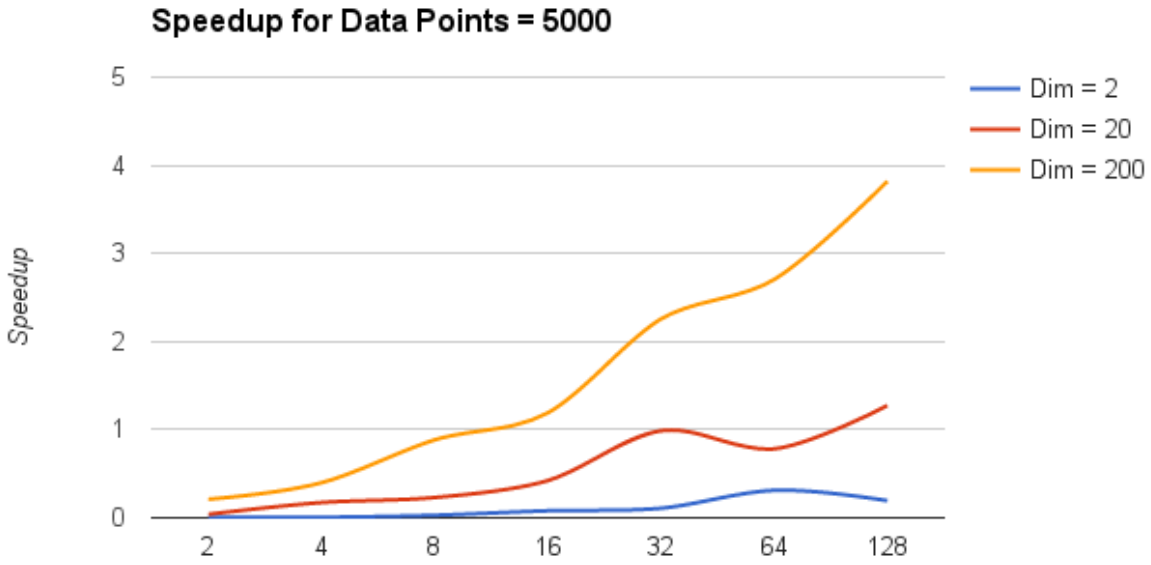
CUDA Results

1. Run time 5000 nodes:

Number of Clusters	CPU(Dim = 2)	GPU(Dim = 2)	CPU(Dim = 20)	GPU(Dim = 20)	CPU(Dim = 200)	GPU(Dim = 200)
2	0.0047	1.6138	0.0656	1.6509	0.451	2.1453
4	0.0077	1.3333	0.2416	1.393	0.6237	1.5667
8	0.0381	1.3371	0.3155	1.3711	1.4002	1.5821
16	0.1069	1.3358	0.5893	1.3953	1.8373	1.5419
32	0.1439	1.332	1.3473	1.3642	3.4217	1.5158
64	0.4177	1.3384	1.0732	1.3716	4.2123	1.5576
128	0.2607	1.3402	1.7357	1.3633	6.1313	1.6059

Speedup:

Number of Clusters	Dim = 2	Dim = 20	Dim = 200
2	0.002912380716	0.03973590163	0.2102270079
4	0.005775144379	0.1734386217	0.3980979128
8	0.02849450303	0.2301072132	0.885026231
16	0.08002695014	0.4223464488	1.191581815
32	0.108033033	0.9876117871	2.257355852
64	0.3120890616	0.7824438612	2.704352851
128	0.1945232055	1.273160713	3.817983685

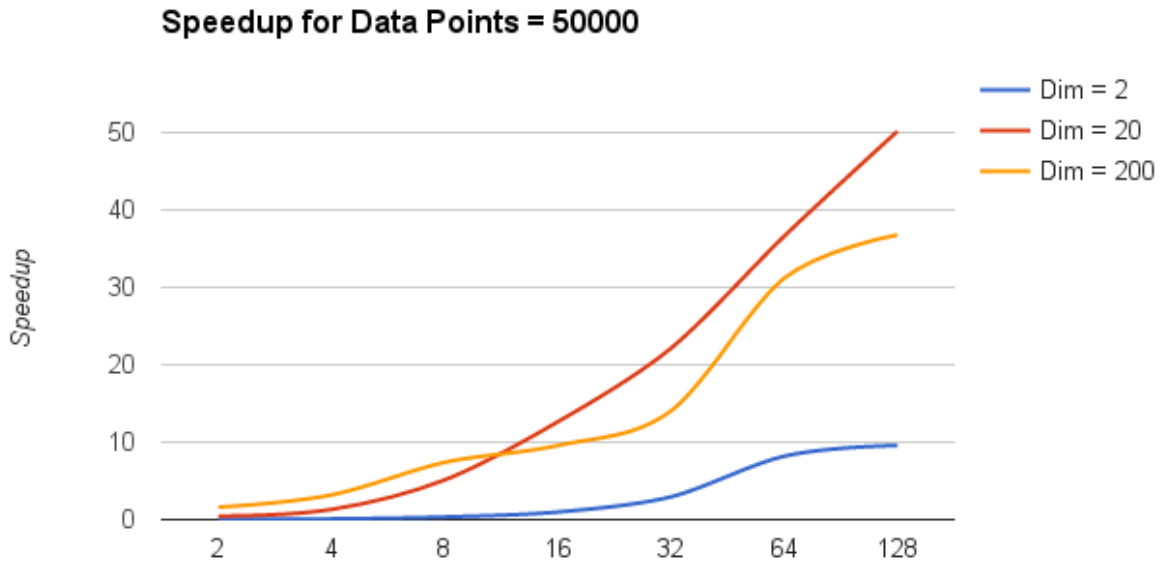


2. Runtime 50000 nodes:

Number of Clusters	CPU(Dim = 2)	GPU(Dim = 2)	CPU(Dim = 20)	GPU(Dim = 20)	CPU(Dim = 200)	GPU(Dim = 200)
2	0.0598	1.6403	0.7285	1.7903	12.9893	8.0939
4	0.1245	1.3288	1.7506	1.9035	22.7088	7.1148
8	0.4512	1.3278	6.8116	2.284	53.0952	7.1728
16	1.302	1.3284	16.8019	2.3741	80.8285	8.4399
32	3.9175	1.33	29.4579	2.5649	106.8482	7.6028
64	10.9041	1.3309	48.7375	2.6366	176.9092	5.6719
128	12.7576	1.3307	66.7613	2.3066	183.6776	4.9939

Speedup:

Number of Clusters	Dim = 2	Dim = 20	Dim = 200
2	0.03645674572	0.4441260745	1.604825857
4	0.0936935581	1.317429259	3.19176927
8	0.3398102124	5.129989456	7.402297569
16	0.9801264679	12.64822343	9.576949964
32	2.945488722	22.14879699	14.05379597
64	8.193027275	36.61995642	31.19046528
128	9.587134591	50.17006087	36.78039208

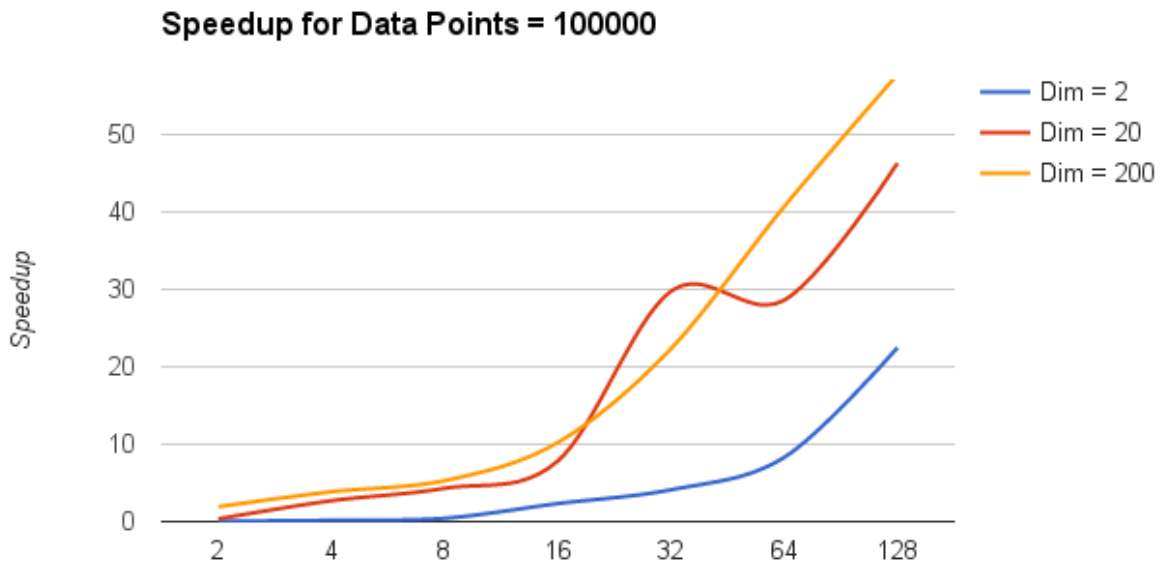


3. Runtime 100000 nodes:

Number of Clusters	CPU(Dim = 2)	GPU(Dim = 2)	CPU(Dim = 20)	GPU(Dim = 20)	CPU(Dim = 200)	GPU(Dim = 200)
2	0.0716	1.6016	0.7287	2.0445	20.2902	10.5202
4	0.2316	1.3514	7.1117	2.6328	73.4858	19.0342
8	0.6	1.4177	17.2575	4.0122	143.5491	27.1399
16	3.4737	1.4875	32.227	4.1089	213.2782	20.8642
32	6.5977	1.6058	112.1626	3.7683	384.6378	17.2082
64	12.8961	1.5549	138.4984	4.8427	527.8646	12.9657
128	36.1652	1.6099	294.1505	6.3547	846.6653	14.6607

Speedup:

Number of Clusters	Dim = 2	Dim = 20	Dim = 200
2	0.04470529471	0.3564196625	1.928689569
4	0.1713778304	2.701192647	3.86072438
8	0.4232207096	4.301256169	5.2892273
16	2.335260504	7.84321838	10.22220838
32	4.108668576	29.76477457	22.3520066
64	8.293845263	28.59941768	40.7123873
128	22.46425244	46.28865249	57.75067357



OpenCL Results

Now we repeat the same evaluations for OpenCL implementation. Vertices range from 5000, 50000, 100000 and dimensions change from 2, 20, 200.

The number of clusters vary from 2 to 128 with increasing power of 2.

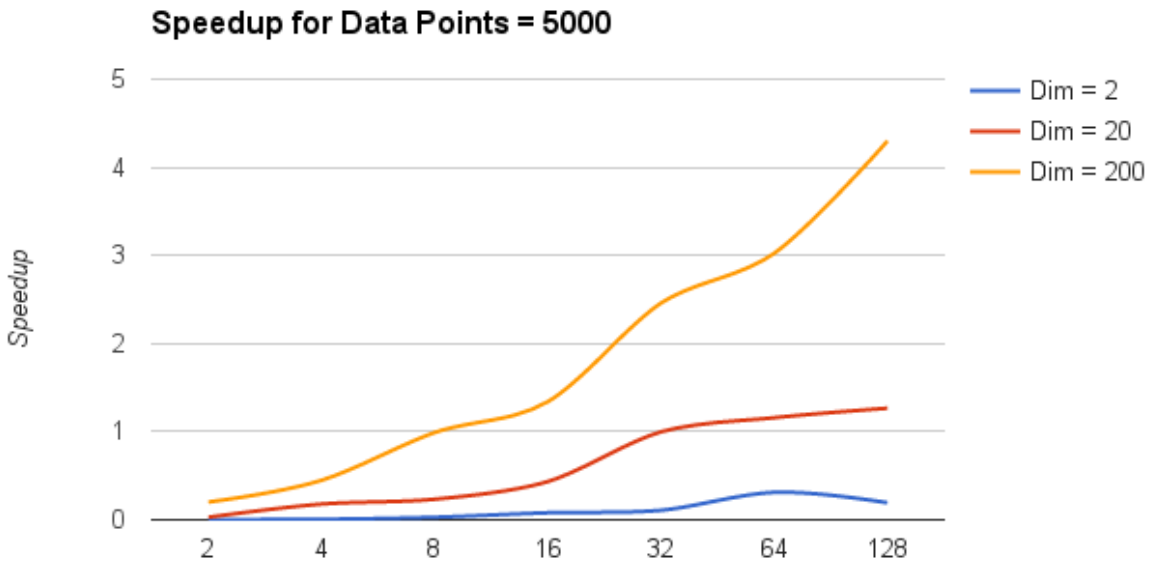
1. Runtime: 5000 nodes

Number of Clusters	CPU(Dim = 2)	GPU(Dim = 2)	CPU(Dim = 20)	GPU(Dim = 20)	CPU(Dim = 200)	GPU(Dim = 200)
2	0.0047	2.27	0.0656	2.26	0.451	2.24
4	0.0077	1.334	0.2416	1.352	0.6237	1.39
8	0.0381	1.333	0.3155	1.351	1.4002	1.413
16	0.1069	1.338	0.5893	1.358	1.8373	1.37
32	0.1439	1.335	1.3473	1.349	3.4217	1.391
64	0.4177	1.34	1.5733	1.358	4.2123	1.393
128	0.2607	1.345	1.7357	1.371	6.1313	1.426

Speedup

Number of Clusters	Dim = 2	Dim = 20	Dim = 200
2	0.002070484581	0.02902654867	0.2013392857
4	0.005772113943	0.1786982249	0.448705036
8	0.02858214554	0.233530718	0.9909412597
16	0.07989536622	0.4339469809	1.341094891
32	0.1077902622	0.9987398073	2.459884975
64	0.3117164179	1.158541973	3.02390524

128	0.1938289963	1.266010212	4.299649369



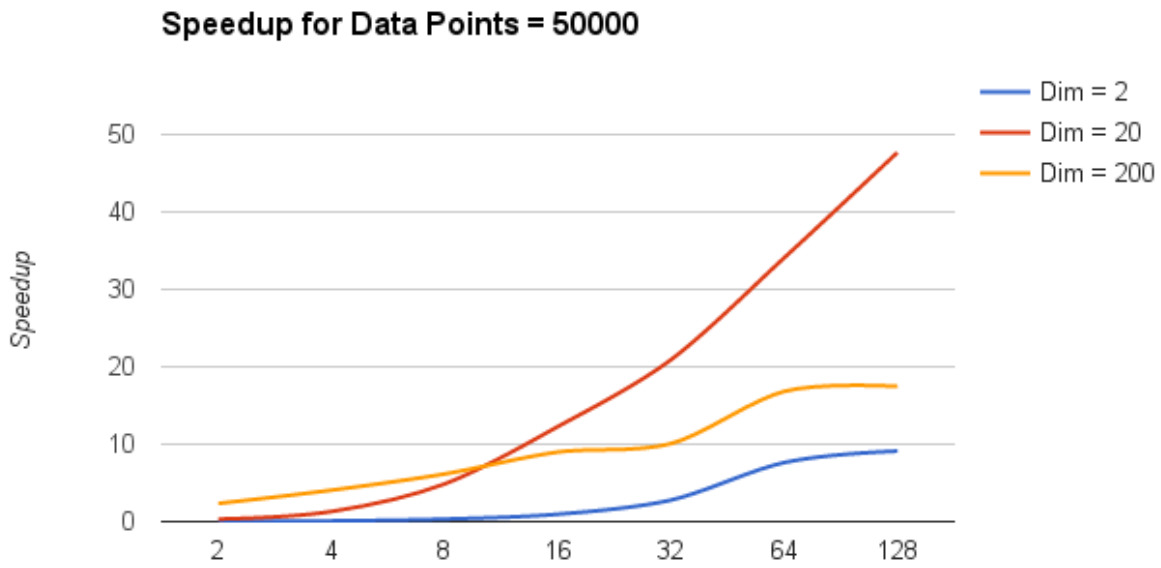
---

2. Runtime: 50000 nodes:

Number of Clusters	CPU(Dim = 2)	GPU(Dim = 2)	CPU(Dim = 20)	GPU(Dim = 20)	CPU(Dim = 200)	GPU(Dim = 200)
2	0.0598	2.254	0.7285	2.329	12.9893	5.53
4	0.1245	1.342	1.7506	1.529	22.7088	5.59
8	0.4512	1.398	6.8116	2.284	53.0952	8.625
16	1.302	1.368	16.8019	2.054	80.8285	8.684
32	3.9175	1.408	29.4579	2.136	106.8482	8.995
64	10.9041	1.43	48.7375	2.288	176.9092	10.604
128	12.7576	1.4	66.7613	2.025	183.6776	10.517

Speedup

Number of Clusters	Dim = 2	Dim = 20	Dim = 200
2	0.02653061224	0.3232031943	2.348878843
4	0.09277198212	1.304470939	4.062397138
8	0.3227467811	4.872389127	6.155965217
16	0.951754386	12.28209064	8.985936631
32	2.782315341	20.92180398	10.07621652
64	7.625244755	34.08216783	16.82126082
128	9.112571429	47.68664286	17.46482837

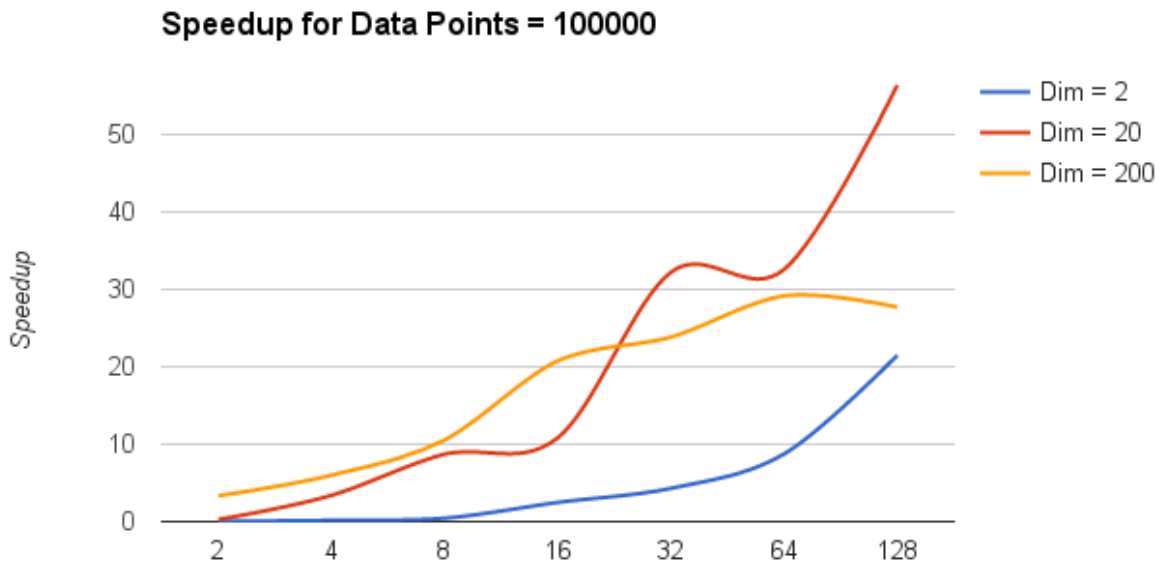


3. Runtime: 100000 nodes:

Number of Clusters	CPU(Dim = 2)	GPU(Dim = 2)	CPU(Dim = 20)	GPU(Dim = 20)	CPU(Dim = 200)	GPU(Dim = 200)
2	0.0716	2.597	0.7287	2.698	20.2902	6.077
4	0.2316	1.359	7.1117	2.095	73.4858	12.261
8	0.6	1.383	17.2575	1.978	143.5491	13.605
16	3.4737	1.399	32.227	2.974	213.2782	10.26
32	6.5977	1.536	112.1626	3.481	384.6378	16.136
64	12.8961	1.474	138.4984	4.25	527.8646	18.093
128	36.1652	1.682	294.1505	5.217	846.6653	30.559

Speedup:

Number of Clusters	Dim = 2	Dim = 20	Dim = 200
2	0.02757027339	0.2700889548	3.338851407
4	0.170419426	3.394606205	5.993458935
8	0.4338394794	8.724721941	10.55120176
16	2.482987848	10.83624748	20.78734893
32	4.295377604	32.22137317	23.83724591
64	8.749050204	32.58785882	29.17507323
128	21.50130797	56.38307456	27.70592297



5. **Conclusion**:

The main aim being implementation of parallel implementation in CUDA and OpenCL, we would like to specify the differentiation on the basis of two criterias.

1. Performance:

For larger data sizes, the performance of OpenCL was slightly bad as compared to CUDA. As per the pointed out in the main paper we used for our implementation, CUDA makes use of texture memory in GPU, and the toolkit allows us to leverage it.

Also we couldn't run a larger data size more than 1 million with 200 dimension on OpenCL but we could do it on CUDA. For the reason to have a fair comparison between them, we limited the data size to 1 million.

2. Ease of implementation:

Implementation in CUDA was easier and also required less lines of code as compared to OpenCL. Trivial things like setting appropriate function arguments are aptly handled in CUDA while they had to separately done in OpenCL. Overall we can say that CUDA is developer friendly.

References:

1. Mario Zechner, Michael Granitzer: Accelerating KMeans on the Graphics Processor via

CUDA. [http://www.knowcenter.tugraz.at/download_extern/papers/latex8.pdf]

2. You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu: Speeding up KMeans Algorithm by

GPUs. [http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5578441]

3. Jiadong Wu and Bo Hong: An Efficient kmeans Algorithm on CUDA

[http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6009040]

4.BAI Hongtaoa,b, HE Lilia,b: KMeans on commodity GPUs with CUDA

[http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5170921]

5. M. E. Hodgson, “Reducing the computational requirements of the minimum-distance

classifier,” [http://www.sciencedirect.com/science/article/pii/0034425788900454]

6. Hiroyuki Takizawa and Hiroaki Kobayashi. Hierarchical parallel processing of large scale data clustering on a pc cluster with gpu co-processing. J. Supercomput.,

36(3):219–234, 2006