Scalable Clustering using Multiple GPUs

Wasif Mohiuddin K
Center for Visual Information Technology
International Institute of Information and Technology,
Hyderabad, India
wasif.m@research.iiit.ac.in

P. J. Narayanan
Center for Visual Information Technology
International Institute of Information and Technology,
Hyderabad, India
pjn@iiit.ac.in

Abstract

K-Means is a popular clustering algorithm with wide applications in Computer Vision, Data mining, Data Visualization, etc. Clustering is an important step for indexing and searching of documents, images, video, etc. Clustering large numbers of high-dimensional vectors is very computation intensive. In this paper, we present the design and implementation of the K-Means clustering algorithm on the modern GPU. All steps are performed entirely on the GPU efficiently in our approach. We also present a load balanced multi-node, multi-GPU implementation which can handle up to 6 million, 128-dimensional vectors. We use efficient memory layout for all steps to get high performance. The GPU accelerators are now present on high-end workstations and low-end laptops. Scalability in the number and dimensionality of the vectors, the number of clusters, as well as in the number of cores available for processing are important for usability to different users. Our implementation scales linearly or near-linearly with different problem parameters. We achieve up to 2 times increase in speed compared to the best GPU implementation for K-Means on a single GPU. We obtain a speed up of over 170 on a single Nvidia Fermi GPU compared to a standard sequential implementation. We are able to execute one iteration of K-Means in 136 seconds on off-the-shelf GPUs to cluster 6 million vectors of 128 dimensions into 4K clusters and in 2.5 seconds to cluster 125K vectors of 128 dimensions into 2K clusters.

1. Introduction

Unsupervised clustering is a means to discover the structure inherent in a large volume of data. Applications of this are abound in Computer Vision, Data Mining, Search, etc. The derived clusters help in understanding and visualizing the original data more efficiently and effectively. Many problems use large numbers of data items of high dimensionality and large number of natural groupings or clusters. Clustering such data is computationally very expensive. The users of such processing are no longer limited to large institutions; smaller institutions, research groups, and even individuals may need similar processing. For instance, clustering could be a first step towards organizing one's personal collection of photographs using computer vision based techniques. The state of the art scene descriptors like SIFT [10] used in computer vision are typically 128 dimensional, in case of GIST [12] upto 580. Many times researchers are forced to use low dimensional data due to computational limitations. The computation needed will be heavy even for modest photo collections. A fast, scalable, and available clustering approach is necessary to solve this problem.

We present the design and implementation of a fast and scalable K-Means clustering algorithm in this paper. K-Means is the most commonly used clustering technique [11], with a sequential time complexity of O(n^{kd+1} $\log n$), where n is the input size, d the dimensionality, and k the number of clusters. We attempt to bring its running time to a practical range by exploiting all computing resources. We concentrate on the highly available multi- and many-core accelerator architectures for our implementation to increase the reach of the approach. Accelerators are common today and are likely to remain so in the future. Graphics Processing Units (GPUs) are now part of most PCs and laptops. While the specific hardware available may change over time, their availability is likely to remain. Our K-Means algorithm is implemented completely on the GPU. The closest center finding and the mean evaluation are done efficiently on the GPU. We also extend our approach to a multi-GPU framework. GPUs provide practical parallel programming to a large number of users. Our objective is to provide a solution that scales well with all aspects of the problem size and the number of available cores. We show significant performance improvement even on GPUs found on laptops.

Our K-Means implementation exploits the parallelism within each data item as well as among the different data items. This is essential to utilize the hardware resources under the massively multithreaded model. We perform label assignment and mean evaluation completely on GPU. The data transfer between CPU and GPU is minimized as a result. We use the K-Means++ [1] algorithm for generating initial cluster centers for better clustering. Our approach performs better than prior GPU implementations. Implementation on different generations of GPUs was done to study their respective architectural features. We achieve perfor-

mance nearly 170 times faster than a standard single-core CPU implementation on the GPU. We scale our algorithm for large n, using multi-GPU approach. We achieve linear scaling in the running time in the number of vectors and the dimensionality, and superlinear scaling in the number of clusters. The running time scales linearly with number of data objects, feature space and clusters chosen.

2. Related Work

K-Means has been worked on by many researchers. Pelleg and Moore [16] employed kd-trees to improve the K-Means algorithm. Weber and Zezula [18] found that bounding trees do not scale well with increasing dimensions. Elkan [5] used triangle inequality to reduce unnecessary distance calculations based on distance from the previous centers and maintaining a look up table between old and new centers. Although there was a reduction in distance evaluations by a factor of nearly 10, for high values of k the book keeping turned out to be a dominant expense. There have been attempts to run K-Means on the GPU. Hall and Hart [6] in their implementation on NVIDIA's GeForceFX 5900 Ultra used the fragment shader to fetch input data from texture memory and cluster center from the constant memory for metric evaluation. Their approach was constrained in dimensionality due to texture memory limitation.

Che *et al.* [4] and Zechner *et al.* [21] perform some steps of K-means on the GPU, where every thread is associated with a data object sequentially evaluating its label but the evaluation of new means was done entirely on CPU. Hong-Tao *et al.* [8] in their approach further moved the new center evaluation partially on the GPU, achieving a speedup of 70 on NVIDIA GeForce 8800 GTX for small clusters but the rearrangement of input vectors as per labels was done on the CPU. In GPUMiner Wenbin *et al.* [19] discussed two approaches for small and large input data due to the limited device memory on GPU and streamed data whenever size exceeded and achieved a speed up of 50-88 on GTX 280. Its drawback is poor memory utilization and high space requirements for large k. HPKmeans by Wu *et al.* [20] considered GPU memory hierarchy to utilize the bandwidth efficiently. The used constant and texture memory for their caches and shared memory for frequently accessed data. Li *et al.* [9] moved the mean evaluation on GPU using a divide and rule approach. The input data is divided into n/M chunks where M is a multiple of number of SM's. Partial centroids are evaluated on GPU iteratively and eventually sent to CPU for final processing. Their approach would not be beneficial for large n. Zhang *et al.* [22] used a document clustering technique MSF (Multi Species Flocking) based on neighbour flock mates within a certain range by defining force components (Separation, Alignment and Cohesion) which is not sensitive to initial state. But the complexity of the algorithm is quadratic and also requires message passing amongst nodes.

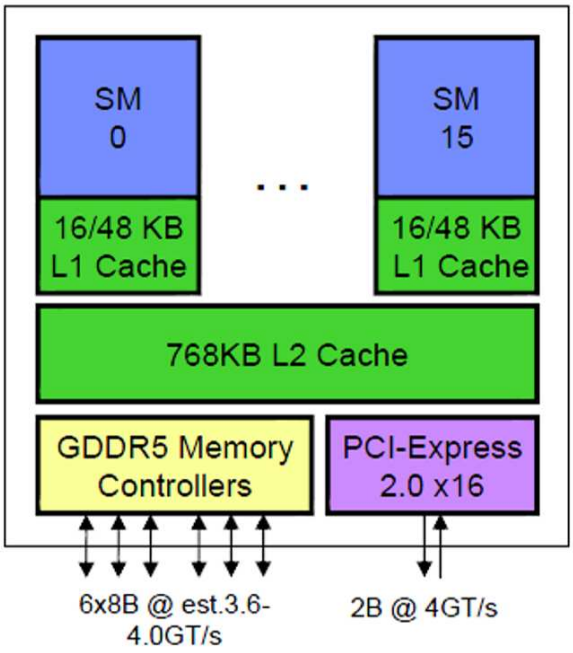

Figure 1. The Fermi Architecture

The prior GPU approaches exploited the parallelism of the multiple data items only. Data objects were independent and were assigned to a single thread. The evaluation on each dimension can also be parallelized, which we exploit. We propose an implementation on the multiple GPUs with multiple threads computing the label for a single data object. The new mean evaluation is also entirely performed on GPU using fast scalable sort [15] and rearranging of input vectors. We use the transpose primitive to increase data coherence and achieve good performance.

3. GPU Architecture

In this section, we describe NVIDIA's GPU architectures briefly. The GPU has a manycore architecture, which is being used on applications that require high computational power. NVIDIA's Fermi architecture [14] has 16 Streaming Multiprocessors (SM) as shown in Figure 1 with each SM having 32 cores, so on the whole it has 512 CUDA [13] cores. Every core in an SM executes the same instruction. The function calls are made in the form of kernels which unleash multiple threads to perform a task in a Single Instruction Multiple Data (SIMD) fashion.

Every SM has registers divided equally among its threads. Each thread has a private local memory. The off-chip global device memory per card can be accessed by every thread in

the grid but consumes hundred of clock cycles for a single fetch. The performance of applications requiring frequent access of some data can exploit the shared memory. The Fermi architecture has a single unified memory request path for loads and stores using the L1 cache per SM multiprocessor and unified L2 cache that services all operations. L1 cache is configurable to support both shared memory and caching of local and global memory operations. The 64 KB memory can be configured as either 48 KB of Shared memory with 16 KB of L1 cache or vice-versa. By configuring 48 KB of shared memory, programs that make extensive use of shared memory performed up to three times faster. The lifetime of this memory is same as that of a block. Fermi features a 768 KB unified L2 cache that services all load, store, and texture requests. The L2 provides efficient, high speed data sharing across the GPU. Apart from the global device memory the GPU has Constant and Texture memory too. The salient features of Fermi architecture are double precision, faster context switching, faster atomic operations, and multiple kernel execution.

4. K-Means Algorithm

K-Means is an unsupervised clustering algorithm for vectors proposed by MacQueen [11]. Given a set $X \subset R^d$ of n points in a d-dimensional space and an integer $k \geq 2$, the problem is to partition X into k disjoint nonempty subsets $(S_1, S_2.., S_k)$ along with a set $C = c_1, ..., c_k$ of corresponding centers such that the vectors in S_i is closest to the center c_i than any other c_j for a pre-defined distance measure. The sequential K-Means algorithm iteratively (Algorithm 1) calculates the distance between every input vector to each of the current centers and assigns it a label i between 1 and k based on the center c_i it is nearest to. In a second step, each center c_i is updated to the mean of all vectors with the label i. This 2-step process is repeated until no vector changes the label or another convergence condition is met. The computations incurred per iteration may be given by $O(nkd)$ as the distance from each input point to each center needs to be calculated.

Algorithm 1 K-Means Algorithm

1: Input $\{x_i | i = 1...n\} \subset R^d$ and a set $C = c_1, ..., c_k$ of initial centers
2: Membership evaluation : Assign each vector x_i to cluster c_j with a label j for which the distance(x_i,c_j) is minimum among the current cluster centers.
3: Mean evaluation : Evaluate new centers c'_j as the mean of all vectors x_i that was assigned the label j.
4: Check condition for convergence, if true then convergence is achieved else go back to step 1 with the set of new centers C'

The first task is to select the initial centers in order to cluster the input data. They could be selected randomly from the set of points. This could result in large number of iterations before convergence. We use the K-Means++ scheme to generate the initial centers which are spread out [1]. In K-Means++, the first center is selected at random from the input vectors. The subsequent ones are selected based on the probability of distance $D(\mathrm{x}_i)^2$ from the previous selected center. K-Means++ is computationally more expensive but reduces the overall computation time in practice by lowering the number of iterations.

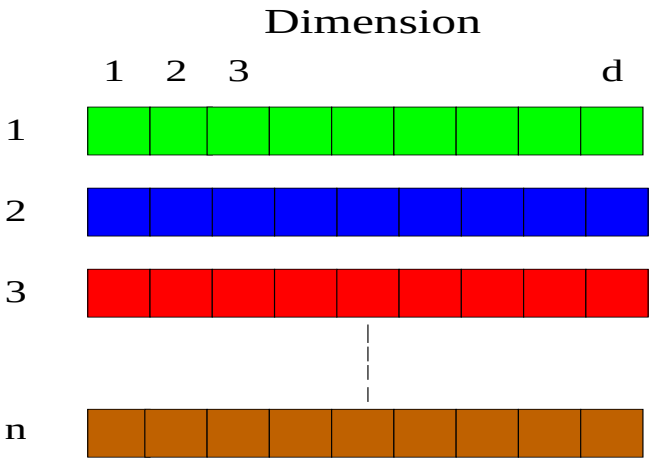

Figure 2. n input vectors of d dimensions are stored in row major format in the GPU global memory.

4.1. Single GPU Implementation

The implementation is divided into two parts: membership evaluation and mean evaluation. We have extended the parallelism to the computation done on the d components of each input and center vector. The input vectors and cluster centers are stored in row major format to enable fast coalesced reading of the whole vector using d threads.

4.1.1. Data Layout. The n input vectors are arranged in a row major format as shown in Figure 2. This provides perfectly coherent memory accesses as consecutive components of each vector is worked on by consecutive CUDA threads.

4.1.2. Membership Evaluation. To generate the membership labels, we need to evaluate the distance of each input vector with all cluster centers. We process q input vectors in each CUDA block as shown in Figure 3. The membership kernel processes these using a $q \times d$ geometry for the block. The total number of blocks is $p = n/q$. The minimum square distance and the corresponding cluster number is stored for each of the q vectors in the shared memory of the block. Component-wise distances are evaluated in parallel

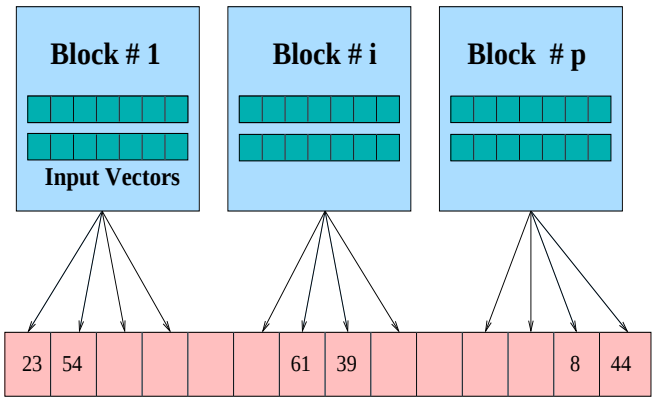

Figure 3. Layout of the CUDA blocks. Each block evaluates new labels for q input vectors and writes to membership array

and stored in the shared memory. The distance for a pair of vector and cluster center is evaluated using a log reduction of these values. This is compared with the current minimum distance for updation if necessary.

The distance and membership evaluation are performed using Algorithm 2. We ignore the square root function for distance comparison to reduce computation. We store the following onto the shared memory: s_data holds the input vectors, s_dist stores square of the differences for every dimension, min_y and $membership_y$ store the global minimum distance and label for vector id y. The calculations are done differently for the current and earlier generation of GPUs. The $syncThreads$ call ensures that threads in a block which have completed the task wait for other threads to finish the task before executing the next instruction. On earlier generation GPUs, shared memory per SM was restricted to 16 KB. The centers were accessed via texture memory and l of them were loaded first onto the shared memory. The ql distance evaluations were done by the kernel then. The current generation GPUs (Fermi) have an L2 cache and thrice as much as shared memory. This helps in dedicating the shared memory for input vectors and distance evaluations. We load the cluster centers directly from the global memory. The L2 cache help us achieve good performance as all blocks access the same set of centers. We observed that the performance was good for 1500 centers of 128 dimensional vectors. The performance deteriorated sharply when the number of centers were increased. This is because L2 cache exceeded its limit and relied on global memory for accessing the additional centers. So in GTX 480 we send centers in batches of approximately 1500 making efficient use of L2 cache. It is, however, important to select the optimum block dimension, number of vectors processed per block, shared memory utilization on a per block basis, etc., so that the GPU is efficiently utilized. We tried varying these above mentioned parameters to find an optimal estimate which gives us high performance.

Algorithm 2 Membership Evaluation

$Input$: *I_Data*, *Centers*
$Output$: *Membership*
$t_i \leftarrow$ Thread Id in a block
$dim \leftarrow$ Vector dimension
$dist_{yz} \leftarrow$ Distance between vector y and center z
$min_y \leftarrow$ Minimum distance of vector y from a center
$s\_dist_{yz} \leftarrow$ Distance components

1: **for** $y = 1$ to n in parallel **do**
2: *membership_y*, *s_data_y*[*dim*]
3: $s\_data_y[] \leftarrow I\_Data_y[]$
4: **for** $z = 1$ to k clusters **do**
5: $s\_dist_{yz}[t_i] = (s\_data_y[t_i] - center_z[t_i])^2$
6: SyncThreads
7: **for** $i = dim/2$ to 0 **do**
8: **if** $Tidx \leq i$ **then**
9: Add $s\_dist_{yz}[2 * t_i + i]$ to $s\_dist_{yz}[t_i]$
10: **end if**
11: SyncThreads
12: $i \leftarrow i/2$
13: **end for**
14: $dist_{yz}$ = $s\_dist_yz[0]$
15: **if** $min_y \leq s\_dist_{yz}[0]$ **then**
16: $min_y \leftarrow s\_dist_{yz}[0]$
17: $Membership_y \leftarrow y$
18: **end if**
19: **end for**
20: **end for**

4.1.3. Center Evaluation. Center evaluation involves finding the sum of all vectors with the same label. For a parallel approach this task involves concurrent writes since data objects having the same membership may add the histogram count at a time. One significant contribution of this paper is implementing the entire process of mean evaluation on the GPU. Algorithm 3 shows the sequence of kernel calls made for center evaluation. In our approach Input data is partitioned into k clusters of different sizes to find the sum. We sort the input vectors based on their labels, using the *split* primitive [15]. The $SplitSort$ kernel brings the records with each label together. This is done by forming a list of 64-bit records combining the new label value and global index of the input vector. We split this using the label value as the key as shown in Figure 4, shuffling the original global index order. The *gather* primitive [15] is used subsequently to rearrange the input vectors in the order of labels using the index after the split. The gather primitive moves data efficiently, using coalesced read/write operations. We mark the cluster boundaries using the function $get\_Boundaries()$

based on the change in sorted labels. These are used to sum the vectors belonging to the same label using segmented scan [17]. Using histogram() function we finalize the count of each cluster. The vectors with similar label are now grouped together using $Rearrange()$ so that the list can be summed component-wise to compute the new centers. The row major storage of the vectors make component-wise addition uncoalesced in memory accesses and hence inefficient on the GPUs. We solve this problem by converting the input vectors to a column major format, i.e., to a $n \times d$ arrangement from the $d \times n$ one. This is done using an efficient transpose operation that uses the shared memory efficiently [2]. The components are now arranged consecutively. The use of transpose is a significant contribution towards the mean evaluation. We use a segmented sum scan of the list of nd elements divided into kd segments. The kernel call $Compact()$ then extracts the new means from the result obtained by segmented scan. Transpose operation is performed once again to revert back the values from row major storage. The final kernel call $Divide()$ gives us the new means for this iteration. All steps are performed using full occupancy and exploit the hardware well. Once we have the new means we check for the convergence condition which if failed we go back with these new centers and evaluate new labels again else we terminate the algorithm. Attaining convergence typically depends on the initial seeding (K-Means++) of data objects. The use of initial seeding depends on the type of clustering application. Our mean evaluation was able to fix the concurrent write problem which most of the previous approaches failed to solve. Also the pure coalesced memory access for data rearrangement proved to be vital step in enhancing mean evaluation on GPU.

Algorithm 3 Kernel sequence for evaluation of new centers

Input: I_data, $Membership$
Output: $new\_centers$

1: $sorted\_Membership \leftarrow$ SplitSort(*Membership*, *Key*)
2: $flag\_hist \leftarrow$ get_Boundaries(*sorted_Membership*)
3: $hist\_scan \leftarrow$ Segscan(*Sorted_Membership*, *Flag_hist*)
4: $Histogram \leftarrow$ Histogram(*Hist_scan*, *Flag_hist*)
5: $data\_sorted \leftarrow$ Rearrange(*I_data*, *Sorted_index*)
6: $transp\_data \leftarrow$ Transpose(*Data_sorted*)
7: $flag\_scan \leftarrow$ get_Boundaries(*sorted_membership*)
8: $seg\_scan \leftarrow$ Segscan(*transp_data*, *flag_scan*)
9: $sum \leftarrow$ Compact(*seg_scan*)
10: $total\_sum \leftarrow$ Transpose(*sum*)
11: $new\_centers \leftarrow$ Divide(*total_sum*, *histogram*)

4.2. Multi-GPU Implementation

A single GPU device with limited global memory makes it hard to process large number of high dimensional input

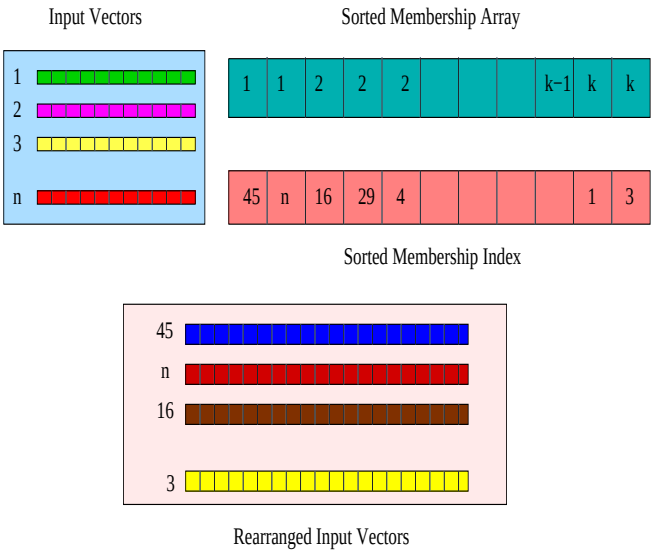

Figure 4. The membership array is sorted on the labels. The input vectors are then rearranged to bring those with the same labels together

vectors. We present a multi-GPU approach which removes some of the memory limitations. Algorithm 4 describes our approach using G_k GPU devices and Z nodes. Figure 5 shows the data partition amongst the G_k GPUs and Z nodes. The input vectors are partitioned uniformly among all available GPUs. When devices have different capabilities, the partitioning should be done on the basis of the capability of the device like number of cores and available device memory. The cluster centres are copied to all GPUs as they occupy much less space. One of the CPUs serves as the master node (Figure 5) and distributes the data to other nodes. The cluster centers are broadcast to all nodes and copied to all GPUs at the start of each iteration. We can truly exploit the computational power of each device and balance the load distribution. This frame work may be adopted in cases where we have large number of GPU devices available on multiple nodes.

Each GPU performs operations similar to the single GPU implementation. It first computes the new membership for each input vector in its partition. Then, the partial sums of its share of vectors for each of the K clusters is computed on the GPU. This data is sent to the CPU of the node along with the size of each new cluster. This data is accumulated at the CPU and send to the master node. The master node computes actual cluster centres from the data, which is broadcast to each GPU via its node CPU for the next iteration. This ensures that all $O(n)$ work is performed on the parallel GPUs while some of the $O(k)$ work is performed on the CPU. In practice, the work on the CPU is small compared to membership evaluation.

The number of clusters k is much smaller than the number

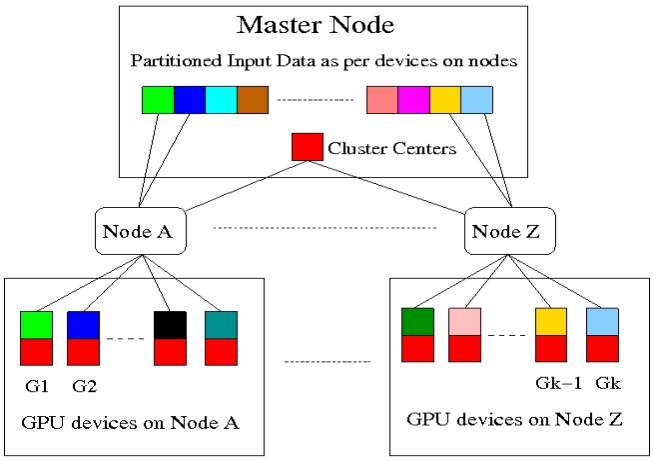

Figure 5. For multinode, multi-GPU configuration, the input vectors are partitioned among the GPUs of all nodes without duplication. The cluster centres are copied to each GPU in each node.

of vectors n. Typically $k \approx \sqrt{n}$ and hence it is fine to have all centres stored in each GPU. In Computer Vision, visual vocabulary is built using high values of n – in tens of millions – depending upon the application. The number of clusters is typically 5000 for such data. In each node, the GPU to CPU bandwidth is reasonably high, to the order of 4GB/s. The cluster centres consist of kd numbers, which comes to 2MB of data for 5K vectors of 128 dimensions. The partial sums can thus be sent in less than 1 millisecond to the CPU. Since the CPU aggregates all data from its GPUs, an equal amount of data needs to be sent from each node to the master node. We used a commodity LAN to connect the network which takes 10-20 milliseconds for such a transfer. A proper HPC cluster will have better bandwidth. At the start of each iteration, the new cluster centres need to be broadcast to all nodes, which involve exactly the same amount of data transfer as partial sums. Transferring input data across the nodes is a one time cost. The communication times are not significant for such large data sets as the membership evaluation takes several seconds even on the GPUs.

5. Results

We evaluate the performance of our approach on NVIDIA's 8600, Tesla S1070 with 4 Tesla T10 devices, and GTX 480 GPUs. For sequential implementation, we use a 32 bit Intel (2.4 GHz, 1GB RAM) Core 2 duo. For the multi GPU implementation we used all four Tesla T10 devices connected over a commodity LAN and also the GTX 480 device. For the sequential results we used Sardi's [7] implementation for K-Means. The input for our K-Means implementation were SIFT vectors. SIFT vectors are high dimensional vectors which represent a key point in an image,

Algorithm 4 Multi-GPU K-Means algorithm

Input: I_data, $Centers$, No of Nodes Z, No of GPUs - G
Output: $Membership$, $new\_centers$

1: Partition the input data into chunks proportional to number of cores, Global memory on each GPU for every node and transfer them to each of the GPUs.
2: Broadcast the k cluster centers to all the GPU nodes.
3: **for** G GPUs in parallel **do**
4: Perform membership evaluation using Algorithm 2 for own partition.
5: Perform mean evaluation using algorithm 3 to calculate partial centers on each device and send them back to respective Nodes.
6: **end for**
7: Every node performs summation of partial means collected from their respective GPUs.
8: Nodes send these partial sum to the Master node.
9: Perform final summation to evaluate new centers for the next iteration on the Master Node.
10: Check for convergence. If true exit else go to step 2 with new centers.

which are clustered to build vocabulary for classification purposes. These are generated using Lowe's [10] implementation. We use one float to store each dimension. We compare with the latest approach on GPU and with a standard CPU implementation as a reference to show how the accelerators can enhance the performance in practice for someone using the CPU. The timings given below are average timing for 20 iterations. The iteration time includes labelling and new mean evaluation. In our experiments, we use our own GPU implementation of K-Means++ for selecting initial clusters. It is an expensive evaluation consuming a time equivalent to 2-4 iterations of K-Means for large n but has been found to drastically reduce the number of iterations. This extra time can be shared by all the iterations which is generally high for large input data. For an input size of 10K we observed it took almost 74 iterations to converge using random centers for initialization and only 15 iterations using K-Means++. Initialization using K-Means++ was found to provide a 2-fold speedup compared to initialization using random initial centers. We have performance timings for inputs upto 6 million vectors of 128 dimension. By varying the cluster centers k, dimension d and input size n, we study the scalability of our approach.

For the Fermi GPU implementation, we process 2 input vectors per block (i.e., $q = 2$). We loop over the cluster centers, taking four centers (i.e., $l = 4$) a time. Each CUDA block uses 256 threads in a row major format contributing two membership values after looping over all centers. The centers were accessed globally via texture memory for the Tesla and the global memory for the Fermi via L2 cache.

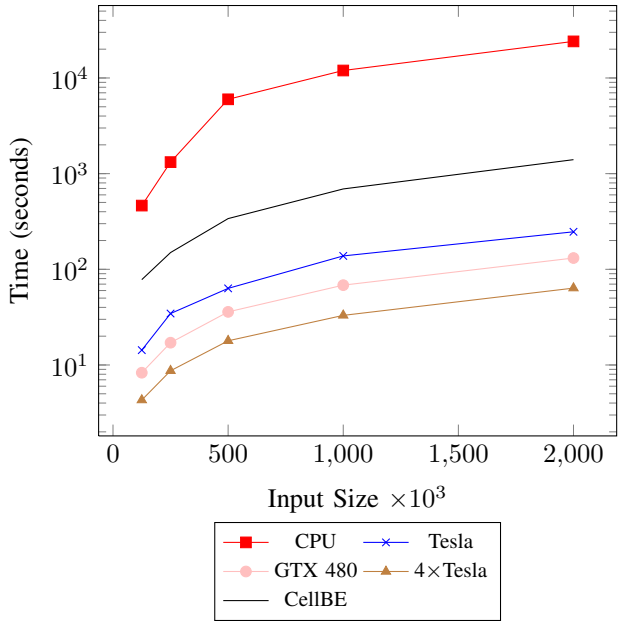

Figure 6. Running time per iteration in seconds for different input sizes for $d = 128$ and $k = 4$K.

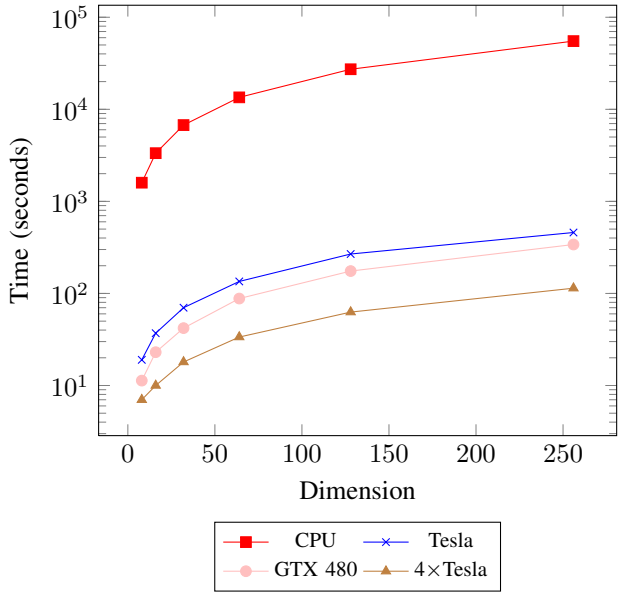

Figure 7. Running time per iteration in seconds for vectors of different dimensionality for $n = 1$M and $k = 8$K

The amount of shared memory used by 2 input vectors and performing 8 distance evaluations was 5136 KB. We achieved an occupancy of 83% for the above parameters on GTX 480. This combination of parameters gave the best performance in practice. The looping over all centers is the time consuming part of membership kernel.

We also show results on the IBM CellBE platform. We perform the membership on the SPUs by streaming the cluster centres in each iteration. Centre evaluation is also performed on the SPU by bringing vectors of selected labels to it. The SIMD intrinsics work very well for the large vectors we use. Buehrer *et al.* [3] made use of single instruction multiple data (SIMD) intrinsics. The architecture is slightly dated, but our results help to compare it with the GPUs. The running times shown in Table 1 provide a comparison between CPU, CellBE and different generations GPUs for different combinations of n and k. The speedup obtained is from 170 on a GTX480, over 400 using 4 Tesla T10 devices and almost 25 on CellBE.

Input Size n	# clusters k	CPU	Tesla T10	GTX 480	4× T10	IBM CellBE
10 K	80	1.3	0.119	0.18	0.097	0.389
50 K	800	71.3	2.73	1.73	0.891	11.2
125 K	2K	463.6	14.18	7.71	2.47	58.4
250 K	4K	1320	38.5	27.7	7.45	149.7
500 K	4K	5985	72.2	54.9	14.9	339
1 M	8K	28936	268.6	170.6	68.5	1356

Table 1. Running time per iteration of K-Means on different GPUs, CellBE and the CPU for 128 dimensional vectors. The timings given are in seconds.

Figure 6 shows a plot of time per iteration for different n with a fixed dimension of 128 and $k = 4000$. A single GPU provided a speed up ranging from 50-170 with increasing n. The multi-GPU results shown use 4× Tesla devices. Figure 7 shows the variation of running time with the dimension d. We can deduce that the performance varies nearly linearly

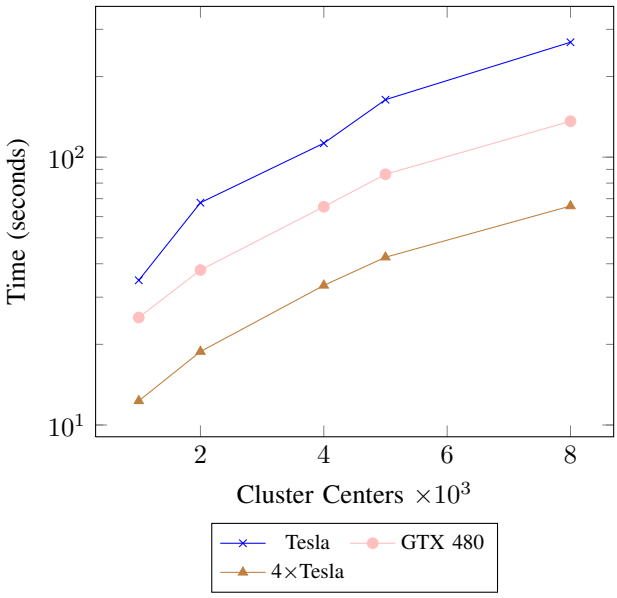

Figure 8. Running time per iteration in seconds for different numbers of clusters for $n = 50$K and $d = 128$

Input Size: n	Number of Centers: k	Approach [8]	[8] with Lazy eval	Our Version of [8]
30 K	1,000	2.955	1.63	0.945
50 K	2,000	10.11	5.2	3.14
125 K	2,000	18.8	8.1	7.41
250 K	4,000	78.3	37.4	26.7

Table 2. Running time in seconds on a GTX480 of Hong-Tao *et al.* [8] approach with and without lazy evaluation and our approach for d=128

input size and the number of dimensions. The distance evaluation time vary linearly with d and the number of distances linearly with n.

Figure 8 shows the dependence of the running time on k, the number of cluster centers. The variation in performance was slightly more than linear. Increasing centers not only add up more work in membership evaluation step but also divergence in the mean evaluation on GPU.

The scalability in the number of cores of the GPU can be seen on low end card like 8600 with 32 cores and limited device memory (256 MB). We limit the input size to 200K vectors and compared the individual performance of 8600 (peak performance 100 GFLOPS) with that of Tesla (peak performance of 1TB) and GTX 480 (peak performance of 1.35TB). For low input size the performance of 8600 compared to other high end cards is shown in Figure 9. GTX 480 boosted the speed on an average by a factor of 1.5-1.8 compared to Tesla. The significant points are that the 8600 also was able to give good performance and the algorithm scales well with increasing number of cores.

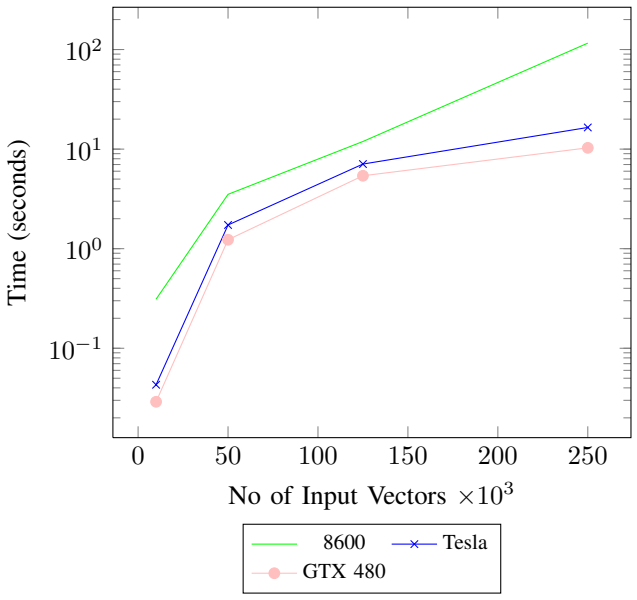

Figure 9. Running time on 8600, T10, and GTX480 GPUs for small input values, d=128, k=80 for n=10k and k=1k for rest.

K-Means involves large amount of distance evaluations. Most of them do not affect current membership of the data object. The approach by Hong-Tao *et al.* [8] evaluate all these distances. We added a lazy evaluations step in which threads whose running sum of the distance exceeds the present minimum exiting early. This can eliminate unnecessary evaluations, but can introduce thread divergence on the GPUs. This is not very serious as the terminated threads only abstain from the computations. Table 2 compares the result of the implementations of the prior method with our modification as well as with of our new method. We have not used lazy evaluation in our implementation technique. Lazy evaluation speeds up computation by a factor of 2 for large d. Our fully parallel implementation further speeds up by roughly 1.5 over the lazy evaluation method for large n and k.

We compare our approach with that of Li *et al.* [9] with the best performance so far and with Wu *et al.* [20]. Table 3 shows the comparison of various approaches on GTX 280 hardware. Unfortunately Wu *et al.* did not provide results for high dimensional data. The computation iterates over the dimensions and cluster centers. Our performance was nearly 4 times better than Wu et al. For low dimensional data Li *et al.* perform marginally better. As the dimension or number of clusters increase, our performance improve by a factor of 2 or better.

Table 4 shows a comparison of timings for increasing input size on single GPU and Multi-GPU devices. The blank values in the table are due to limited global memory on devices we were not able to run experiments on single GPU device for large datasets and had to rely on Multi-GPU approach. We have used Open Message Passing Interface (MPI) for node to node communication in case of multi node multi GPU using 4×Tesla devices on node A and GTX 480 on node B. The scalability of n is dependant on total number of devices across the nodes. We maximized n up till the point where devices can handle data since they have global memory restrictions. So if we have large number of devices available we may scale our performance to a larger n and d.

Input Size: n	dim d	1 Tesla Device	GTX 480 Device	4×Tesla Devices	4×Teslas + GTX 480
1 M	128	120.4	73.3	33.6	22.8
1.5 M	128	181.7	95.6	47.2	34.8
3 M	128	364.2	-	89.671	67.4
6 M	128	-	-	183.8	136.7
16 M	16	220.4	-	57.8	40.9
32 M	16	-	-	116	84.3

Table 4. Running time per iteration in seconds for different input sizes on 1 T10, GTX480, 4× Tesla devices, and 4× T10 + GTX480 for k=4K.

Input Size: n	Dimension d	Number of Centers: k	Our K-Means	Li *et al.* [9] K-Means	Wu *et al.* [20] K-Means
2 Million	8	400	1.27	1.23	4.53
4 Million	8	100	0.734	0.689	4.95
4 Million	8	400	2.4	2.26	9.03
51,200	64	32	0.191	0.403	-
51,200	128	32	0.282	0.475	-

Table 3. Running time in seconds of our approach and those of Li *et al.* and Wu *et al.* on a GTX280.

n, d, k	CPU		1 Tesla		GTX 480	
	New Label	New Mean	New Label	New Mean	New Label	New Mean
50K, 32, 34	33.5	2.28	0.104	0.24	0.073	0.2
0.5M, 32,34	207.78	16.67	0.316	0.4	0.248	0.29
0.5M, 128,2k	2499	548	42.3	1.9	24.5	2.9
1M, 128, 4k	11864	2604	113	7.6	69.2	4.1

Table 5. Times for the labelling and mean evaluation steps per iteration in seconds

Membership evaluation consumes a major percentage of total time of each K-Means iteration. Table 5 shows time division for label evaluation and new mean evaluation. The efficient mean evaluation performed on the GPU results in its time share to about 6% of total time for large input of high dimension. For low d, k the mean evaluation consumes major time but with increasing parameters its share reduces.

5.1. Performance: Discussions

The membership evaluation step is both computation and memory bandwidth intensive. Each input vector needs to compute its distance with each cluster centre. A single distance evaluation needs $3d$ floating point operations (one subtraction, one multiplication, and one addition to find the sum). The max finding uses K comparisons per vector. Thus the total computations are bound by $4NKd$. This comes to $4 \times 2^{20} \times 2^{12} \times 2^7$ or 2 terra floating point operations per iteration for 1M vectors of 128 dimensions with 4K clusters. A single Tesla, with a peak compute power of 1 TFLOPS, should be able to perform these in 2 seconds if fully utilized. Membership evaluation also loads all K cluster centers from the global memory for each input vector. This results in a memory traffic of $4NKd$ bytes per iteration for 4-byte float data. This comes to 2 terrabytes of memory reads, which should take about 20 seconds if the rated peak bandwidth of 100GB per second can be sustained. Table 5 shows the total time to be about 113 seconds on 1 Tesla GPU. This shows that the average effective performance we obtain is off by a factor of 7 from the peak. The achieved absolute performance is better by a factor of 4 using 4 Teslas, though the average utilization is the same. This leaves the possibility for significant future speedup by utilizing the resources more efficiently.

6. Conclusions

We presented the design and implementation of the K-Means clustering algorithm entirely one or more GPUs in this paper. We achieved high performance on different GPU generations using the massive parallelism supported by the CUDA model. Our implementation is scalable in the problem size, the number of dimensions, the number of centers, and the number of available cores on the available GPU. The multi-GPU approach produced nearly linear speedups with large data. Our multi-GPU framework can efficiently solve the clustering problem using multiple nodes each with multiple GPUs. Even low end GPUs found on laptops are shown to provide speedups of 10-20 compared to the CPU version.

The performance we achieve are indicators of the performance that can be obtained on future accelerators, in our view. Such accelerators are becoming more common and are likely to play key roles in different computation steps performed by individuals on their PCs. Scalability to large vectors and problems is important as all aspects of data handled by even individual users is growing very fast.

References

[1] D. Arthur and S. Vassilvitski. K-Means++: The advantages of careful seeding. In *Symposium on Discrete Algorithms*, 2007.

[2] M. Bader, H. Bungartz, D. Mudigere, S. Narsimhaan, and B. Narayanan. Fast GPGPU data rearrangement kernels using CUDA. In *HiPC - High Performance Computing Student Symposium*, 2009.

[3] G. Buehrer and S. Parathasarathy. The potential of cell broadband engine in data mining. In 33^{rd} *Int. Confernce on Very Large Databases(VLDB)*, Austria, 2007.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68:1370–1380, October 2008.

[5] C. Elkan. Using the triangle inequality to accelerate K-Means. In *International Conference on Machine Learning*, pages 147–153, 2003.

[6] J. D. Hall and J. C. Hart. GPU acceleration on iterative clustering. In *SIGGRAPH poster*, 2004.

[7] Har-Peled and B. Sadri. How fast is the K-Means method? In *Proceedings of the* 16^{th} *ACM-SIAM Symposium on Discrete algorithms*, pages 877–885, 2005.

[8] B. Hong-Tao, H. Li li, O. Dant-ong, L. Zhan-shan, and L. He. K-Means on commodity GPU's with CUDA. In *Proceedings of WRI World Congress on Computer Science and Information Engineering.*, volume 3, pages 651–655, 2009.

[9] Y. Li, K. Zhao, X. Chu, and J. Liu. Speeding up K-Means algorithm by GPUs. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 115–122, Washington, DC, USA, 2010. IEEE Computer Society.

[10] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, November 2004.

[11] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium of Mathematical Statistics and Probability*, pages 281–297, Berkely, 1967.

[12] K. P. Murphy, A. Torralba, D. Eaton, and W. T. Freeman. Object detection and localization using local and global features. In *Toward Category-Level Object Recognition*, pages 382–400. Springer-Verlag New York, Inc., 2006.

[13] NVIDIA. CUDA: http://developer.nvidia.com/object/cuda.html.

[14] V. Osipov, N. Leischner, and P. Sanders. NVIDIA fermi Architecture white paper http://www.nvidia.com, 2009.

[15] S. Patidar and P. J. Narayanan. Scalable split and sort primitives using ordered atomic operations on the GPU. In *IIIT, Hyderabad TR/2009/99. Tech. Rep*, 2009.

[16] D. Pelleg and A. Moore. Accelerating exact K-Means algorithm with geometric reasoning. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKKD)*, 1999.

[17] M. H. S. Sengupta, Y. Zhang, and J. Owens. Scan primitives for GPU computing. In *Proceedings of Graphics Hardware*, 2007.

[18] R. Weber and P. Zezula. The theory and practice of searches in high dimensional dataspaces. In *Proceedings of the Fourth DELOS Workshop on ImageIndexing and Retrieval*, 1997.

[19] F. Wenbin, L. Ka, Keung, L. Mian, X. Xiangye, L. Chi, Kit, Y. Philip, H. Bingsheng, L. Qiong, S. Pedro, V, and Y. Ke. Parallel data mining on graphic processors. In *Technical Report HKUSTCS08*, 2008.

[20] R. Wu, B. Zhang, and M. Hsu. Clustering billions of data points using GPUs. In *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, UCHPC-MAW '09, pages 1–6, New York, NY, USA, 2009. ACM.

[21] M. Zechner and M. Granitzer. Accelerating K-Means on the graphics processor via CUDA. In *Proceedings of the 2009 First International Conference on Intensive Applications and Services*, pages 7–15, Washington, DC, USA, 2009. IEEE Computer Society.

[22] Y. Zhang, F. Mueller, X. Cui, and T. Potok. Data-intensive document clustering on graphics processing unit (GPU) clusters. *Journal of Parallel and Distributed Computing.*, 71:211–224, February 2011.