International Journal of Enhanced Research in Management & Computer Applications, ISSN: 2319-7471
Vol. 3 Issue 9, Sept.-2014, pp: (15-24), Impact Factor: 1.296, Available online at: www.erpublications.com

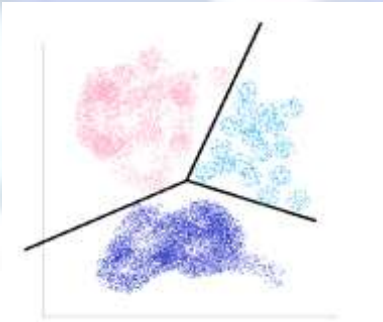
Implementation of k-Means Clustering Algorithm in CUDA

Hanu Hooda[1], Rainu Nandal[2]
[1,2]Dept. of Computer Science Engineering, UIET, MDU Rohtak, Haryana

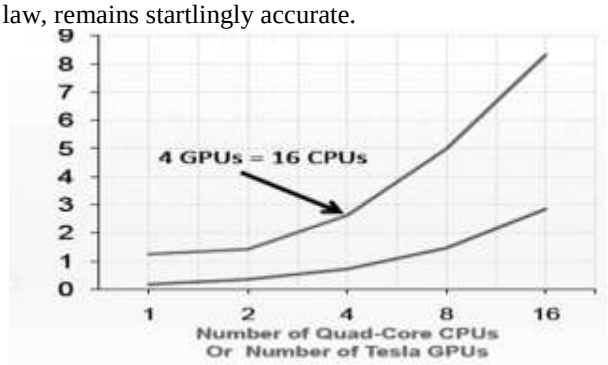
Abstract: Big Data poses a very great computational challenge for programmers as well as machines as a lot of number crunching is to be done.Due to recent development in the shared memory inexpensive architecture like Graphics Processing Units (GPU), an alternative has emerged. In this paper, we target at decreasing runtime for k-Means, which is one of the most popular clustering algorithms, by using the widely available Graphics Processing Units (GPUs). The general – purpose applications are implemented on GPU using Compute Unified Device Architecture (CUDA). Cost effectiveness of the GPU and several features of CUDA like thread Divergence and coalescing memory access. Shared memory architecture is much more efficient than distributed memory architecture. Depending on hardware, data set, and k, dramatic improvements in performance can be seen over CPU implementations.
Keywords: Clustering, k-means, Graphics Processing Unit (GPU), Compute Unified Device Architecture (CUDA), Data Mining.

I. INTRODUCTION

Cluster analysis or **clustering** is the task of grouping a set of objects in such a way that objects in the same group (called a **cluster**) are more similar (in some sense or another) to each other than to those in other groups (clusters). It is a main task of exploratory data mining, and a common technique for statistical data analysis, used in many fields, including machine learning, pattern recognition, image analysis, information retrieval, and bioinformatics.
Cluster analysis itself is not one specific algorithm, but the general task to be solved. It can be achieved by various algorithms [1] [2] [3] that differ significantly in their notion of what constitutes a cluster and how to efficiently find them. Popular notions of clusters include groups with small distances among the cluster members, dense areas of the data space, intervals or particular statistical distributions. Clustering can therefore be formulated as a multi-objective optimization problem. Cluster analysis as such is not an automatic task, but an iterative process of knowledge discovery or interactive multi-objective optimization that involves trial and failure. It will often be necessary to modify data pre-processing and model parameters until the result achieves the desired properties.



Computers get faster and faster every year. In 1965, Intel co-founder Gordon Moore made a prediction that the number of transistors that could inexpensively be fit onto a chip would double every two years. Almost 50 years later, his prediction, now called Moore's law, remains startlingly accurate.



Despite this explosion in speed, computers aren't able to keep up with the scale of data becoming available. By some estimates, advances in gene sequencing technology will make gene-sequence data available more quickly than processors are getting faster. Hardware industry has reached three walls; Power wall, Frequency wall, Memory bandwidth wall. No more significant single core performance improvement in the foreseeable future can be seen. Industry is going multi-core.Two approaches to multi-cores: A few powerful cores or Many simple cores. These two approaches are closing their gap. GPU cores are getting more powerful, while the first group increases the number of cores. Traditional GPUs are hard to program for general purpose algorithms, because they are being designed for SIMD problems.

To circumvent physical and mechanical constraints on individual processor speed, manufacturers are turning to another solution: multiple processors. If two, or three, or more processors are available, then many programs can be executed more quickly. While one processor is doing one aspect of some computation, others can work on another. All of them can share the same data, but the work will proceed in parallel.

In order to be able to work together, multiple processors need to be able to share information with each other. This is accomplished using a shared-memory environment. The variables, objects, and data structures in that environment are accessible to all the processes. The role of a processor in computation is to carry out the evaluation and execution rules of a programming language. In a shared memory model,different processes may execute different statements, but any statement can affect the shared environment.

With the increasing adoption of GPUs in high performance computing (HPC), NVIDIA GPUs are becoming part of some of the world's most powerful supercomputers and clusters. The most recent top 500 list [9] of the world's fastest supercomputers included nearly 50 supercomputers powered by NVIDIA GPUs.Modern GPUs have morphed into programmable cores instead of fixed purpose hardware programmable shader cores instead of old-style fixed hardware vertex and pixel shaders. NVIDIA GPUs can be programmed with CUDA, an extension to the C language.

Similar languages exist from AMD/ATI (Stream SDK) and Apple (OpenCL). It can improve the performance in the orders of magnitude in certain types of applications. Many scientific applications and in particular clustering are inherently suitable for parallel computing. In many cases data can be divided into almost independent chunks which can be acted upon almost independently.

Many algorithms[4][5][6][7][8] have been proposed to improve the performance of clustering operations using multi-level memory hierarchies that include disks, main memories, and several level of processor caches. Over the last few years, database architectures have moved from disk-based systems to main-memory systems and the resulting applications tend to be either computation-bound or memory-bound.

II. K-MEANS CLUSTERING

The term "k-means" was first used by James MacQueen in 1967[10]. The standard algorithm was first proposed by Stuart Lloyd in 1957 as a technique for pulse-code modulation, though it wasn't published outside of Bell Labs until 1982. In 1965, E.W.Forgy published essentially the same method, which is why it is sometimes referred to as Lloyd-Forgy [11]. A more efficient version was proposed and published in Fortran by Hartigan and Wong in 1975/1979[3].

k-meansis an unsupervised method of cluster analysis that aims to partition n observations into k clusters, in which each observation belongs to the cluster with the nearest mean. Given a set of d-dimensional vectors D = {xi|i = 1, . . . ,N}, the algorithm is initialized by picking k "centroids" randomly or by some heuristic, the algorithm proceeds by alternating between two steps till convergence:

1. Data Assignment. In iteration t, each data point is assigned to its closest centroid.

$$S_i^{(t)} = \{x_p : \|x_p - m_i^{(t)}\|^2 \leq \|x_p - m_j^{(t)}\|^2 \ \forall j, 1 \leq j \leq k\},$$

where each $x_p$is assigned to exactly one $S^{(t)}$, even if it could be is assigned to two or more of them.

The default measure of closeness is the Euclidean distance. In some applications, KL-divergence is used to measure the distance between two data points representing two discrete probability distributions.

2. Relocation of "means". Calculate the new centroid of the data points in the cluster.

$$m_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j$$

There is no guarantee that it will converge to the global optimum, and the result may depend on the initial clusters. Therefore, it is common to run the algorithm multiple times with different initial centroids. There has been some works for studying the advantages of the parallelism in the k-means procedure. Parallel k-means has been studied by previously for very large databases. The speedup and scale-up variation with respect to number of documents (vectors), the number of clusters and the dimension of each of the documents has been studied. The parallel k-mean algorithm in Master-slave mode is described as follows:

1. Partition the dataset into P blocks D1, . . . ,DP , each processor handles roughly N/P.
2. Processor P0 builds the initial k centriods (m1, . . . ,mk)global, and broadcasts them to all processors.
3. Processor Pr reads the part of dataset Dr based on its responsibility, and determines the centroid to which its set of xi is closest using global parameters, one xi at a time. Then Pr computes its local centroids (m1, . . . ,mk)local, and then sends it to P0.
4. After P0 collects all local centroids, computes the new global centriods and broadcast it to all processors.
5. Iterate 3,4 until convergence.

CPU-based k-Means

1. **while** flag &&iter <= Max_iter
2. **for** each r **in** R
3. **for** each s **in** S
4. Compute d(r, s);
5. **end** of **for**
6. **end** of **for**
7. **for** each r **in** R
8. **for** each s **in** S
9. **if** d(ri, sj) <min_D
10. r belongs to sj;
11.min_D _ d(ri, sj);
12. **end** of **if**
13. **end** of **for**
14. **end** of **for**
15. **for** each Si in S
16. find new centroid of Si
17. **end** of **for**
18. **if** the change of the centroid is less than threshold
19. flag _ false;
20. iter _ iter + 1;
21. **end** of **if**
22. **end** of **while**

k-Means firstly loads the data on the register, and reads the data from the register each time when calculating the distance from each centroid, resulting in a low global memory access times and latency, since reading from register is by far faster than reading from other memories.
The correct choice of k is often ambiguous, with interpretations depending on the shape and scale of the distribution of points in a data set and the desired clustering resolution of the user. In addition, increasing k without penalty will always reduce the amount of error in the resulting clustering, to the extreme case of zero error if each data point is considered its own cluster (i.e., when k equals the number of data points, n). Intuitively then, the optimal choice of k will strike a balance between maximum compression of the data using a single cluster, and maximum accuracy by assigning each data point to its own cluster. If an appropriate value of k is not apparent from prior knowledge of the properties of the data set, it must be chosen somehow.One popular method looks at the percentage of variance explained as a function of the number of clusters: One should choose a number of clusters so that adding another cluster doesn't give much better modeling of the data. More precisely, if one plots the percentage of variance explained by the clusters against the number of clusters, the first clusters will add much information (explain a lot of variance), but at some point the marginal gain will drop, giving an angle in the graph. The number of clusters is chosen at this point, hence the "elbow criterion". This "elbow" cannot always be unambiguously identified. Percentage of variance explained is the ratio of the between-group variance to the total variance, also known as an F-test. A slight variation of this method plots the curvature of the within group variance.

K-means analysis is a divisive, non-hierarchical method of defining clusters. This is an iterative process, which means that at each step the membership of each individual in a cluster is reevaluated based on the current centers of each existing cluster. This is repeated until the desired number of clusters (or the number of individuals) is reached. Thus, it is non-hierarchical because an individual can be assigned to a cluster, and reassigned at any later stage in the analysis. Clusters are defined based on Euclidean distances so as to reduce the variability of individuals within a cluster, while maximizing the variability between clusters.

For low-dimensional data set, our algorithm utilizes GPU registers and achieves a speedup of approximately 5-8 times over CPU only version. Then it adopts shared memory and registers to avoid multiple accesses of the global memory, and finally achieves a speedup of approximately 4 times as compared with CPU only version.

GPU-based k-Means

1. while a/N > threshold
2. assign a = 0
3. for I = 0 to N – 1
4. for J = 0 to K – 1
5. compute d(object[I], cluster[J])
6. if distance < dmin
7. dmin = distance
8. n = J
9. if cmem[I] != n
10. a = a + 1
11. cmem[I] = n
12. cnew[n] = cnew[n] + object[I]
13. increment new cluster size
14. for J = 0 to K-1
15. cluster[J][*] = cnew[J][*] / cluster size
16. cnew[J][*] = 0

Data objects to be clustered are evenly partitioned among all processes while the cluster centers are replicated. Assign each point to the nearest cluster center. This step can be parallelized. Global-sum reduction for all cluster centers is performed at the end of each iteration in order to generate the new cluster centers.

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 huge computational requirement.

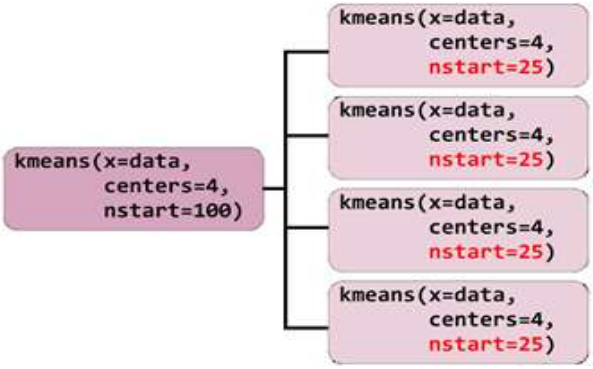
The main disadvantage of this algorithm is that it does not yield the same result with each run, sincethe resulting clusters depend on the initial randomassignments. It minimizes intra-cluster variance, butdoes not ensure that the result has a global minimumof variance. If, however, the initial clusterassignments are heuristically chosen to be aroundthe final point, one can expect convergence to thecorrect values. D. Arthur and S. Vassilvitskii[33]propose an optimization bychoosing initial centroids close to existing clusters.

It is well known that the k-means algorithm is a hard threshold version of the expectation-maximization (EM) algorithm. Meanwhile, EM algorithm has a natural connection to Gibbs sampling methods for Bayesian inference, which is widely used in Mixture model. One can envision that the EM algorithm and Gibbs Sampling can be effectively parallelized using essentially the similar strategy as that used in this parallel k-means.

In this paper, we discuss a parallel implementationof multi-dimensional k-meansalgorithm on the GPU using R scripts. Our approachconsists of parallelizing the steps where we calculate the nearest centroid toeach point. Indeed, since k is generally a feworders of magnitude larger than the amount ofcores currently available, the algorithm is boundby this phase and not by the second. To see this,observe that with p cores we can accelerate thefirst phase to !(nk/p), so the ratio between thetwo is !(nk/p(n + k)) ' !(k/p) (for n (k).The correctness of the algorithm is guaranteed,to the extent that it produces the same output asthe original k-means algorithm. Our implementationperforms the same steps as the original k-meansalgorithm in parallel without changing the semanticsor logic of the original implementation.

III. Parallel K-Mean

Data objects assignment and k centroids recalculation arethe most intensive arithmetic task load of k-means. There aretwo strategies in data objects assignment process suited toGPU-based kmeans. The first is the centroids -oriented, inwhich distance from each centroid to all data objects arecalculated and then, each data point will merge itself into thecluster represented by nearest centroid. This method hasadvantages when the number of processors of GPU isrelatively small so that every processor can deal with dataobjects in series. Another is the data objects -oriented,namely, each data point calculates the distance from allcentroids, then data object will be assigned to the clusterrepresented by the centroid with the shortest distance from it.



In k-means algorithm, every data point must choose thenearest centroid after calculating all the distances, thisselecting process consists a series of comparison whichcould be carried out through Deep Buffer in early GPUs. Inthis way, the latency of memory access could be avoidedwhile one thread is waiting for memory access, and otherthreads will be optimized to use the arithmetic resources.
When the data set is small enough to fit in the GPU memory, we adopted an implementation that transfers the entire data set and an initial set of k-centers to GPU memory upon start; the data set is then transposed in the GPU memory so that the use of GPU memory bandwidth is optimized. Thereafter, for each iteration, GPU computes the "assign center" procedure which assigns a center id to each data point, and transfers the assignment results back to CPU which computes the new set of k-centers. The new set of k-centers is then copied to GPU to start the next iteration. Note that the data set is transferred from CPU to GPU and transposed only once. In the stream-based algorithm, the data set is partitioned into large blocks. At every iteration, we process these blocks in turn, until all of them have been processed. Processing of each block includes transferring the block from CPU to GPU, transposing it to a column-based layout, computing cluster membership for each data point in the data block, and transferring the assignment results back to CPU. At any given time, more than one block is being processed concurrently. CUDA streams have been used to keep track of the progress on each block. Note that all calls are asynchronous, which give maximum possibilities for overlapping computation and memory transfers.
Commonly used initialization methods are Forgy and Random Partition. The Forgy method randomly chooses k observations from the data set and uses these as the initial means. The Random Partition method first randomly assigns a cluster to each observation and then proceeds to the update step, thus computing the initial mean to be the centroid of the cluster's randomly assigned points. The Forgy method tends to spread the initial means out, while Random Partition places all of them close to the center of the data set. According to Hamerly et al. [12], the Random Partition method is generally preferable for algorithms such as the k-harmonic means and fuzzy k-means. For expectation maximization and standard k-means algorithms, the Forgy method of initialization is preferable.

IV. R

R is a system for statistical computation and graphics. It consists of a language plus a run-time environment with graphics, a debugger, access to certain system functions, and the ability to run programs stored in script files.
The design of R has been heavily influenced by two existing languages: S and Scheme. Whereas the resulting language is very similar in appearance to S, the underlying implementation and semantics are derived from Scheme. The core of R is an interpreted computer language which allows branching and looping as well as modular programming using functions. Most of the user-visible functions in R are written in R. It is possible for the user to interface to procedures written in the C, C++, or FORTRAN languages for efficiency. The R distribution contains functionality for a large number of statistical procedures. Among these are: linear and generalized linear models, nonlinear regression models, time series analysis, classical parametric and nonparametric tests, clustering and smoothing. There is also a large set of functions which provide a flexible graphical environment for creating various kinds of data presentations. Additional modules ("add-on packages") are available for a variety of specific purposes.
A large group of individuals has contributed to R by sending code and bug reports.
R has a home page at http://www.R-project.org/. It is free software distributed under a GNU-style copyleft, and an official part of the GNU project ("GNU S").

V. GRAPHIC PROCESSING UNIT

GPU is an acronym for **graphics processing unit**. It is a special-purpose co-processor that helps the traditional central processor (CPU) with graphical tasks. A GPU is designed to process parallel data streams at an extremely high throughput. It does not have the flexibility of a typical CPU,but it can speed up some calculations by over an order of magnitude. Recent architectural decisions have made the GPU more flexible, and new general purpose software stacks allow them to be programmed with far greater ease.

The use of GPUs in HPC is targeted at data-intensive applications which spend nearly all of their time running mathematical kernels that are amenable to SIMD operations. These kernels must exhibit finely-grained parallelism, both in terms of being able to process many independent steams of data as well as pipelining operations on each stream. This emphasis on data-parallelism means that GPUs will not aid programs that are constrained by Amdahl's Law or which require the use of complex memory structures or data access patterns. A GPU is connected to a host through a high speed IO bus slot, typically PCI-Express in current high performance systems. The GPU has its own device memory, up to several gigabytes in current configurations. Data is usually transferred between the GPU and host memories using programmed DMA, which operates concurrently with both the host and GPU compute units, though there is some support for direct access to host memory from the GPU under certain restrictions. As a GPU is designed for stream or throughput computing, it does not depend on a deep cache memory hierarchy for memory performance. The device memory supports very high data bandwidth using a wide data path. On NVIDIA GPUs, its 512-bits wide, allowing sixteen consecutive 32-bit words to be fetched from memory in a single cycle. It also means there is severe

effective bandwidth degradation for strided accesses. A stride-two access, for instance, will fetch those 512 bits, but only use half of them, suffering a 50% bandwidth penalty.

NVIDIA GPUs have a number of multiprocessors, each of which executes in parallel with the others. A Keplar multiprocessor has 12 groups of 16 stream processors. I'll use the more common term core to refer to a stream processor. A high-end Keplar has 15 multiprocessors and 2880 cores. Each core can execute a sequential thread, but the cores execute in what NVIDIA calls SIMT (Single Instruction, Multiple Thread) fashion; all cores in the same group execute the same instruction at the same time, much like classical SIMD processors. SIMT handles conditionals somewhat differently than SIMD, though the effect is much the same, where some cores are disabled for conditional operations.

The code is actually executed in groups of 32 threads, what NVIDIA calls a warp. There is also a small software-managed data cache attached to each multiprocessor, shared among the cores; NVIDIA calls this the shared memory. This is a low-latency, high-bandwidth, indexable memory which runs close to register speeds. When the threads in a warp issue a device memory operation, that instruction will take a very long time, perhaps hundreds of clock cycles, due to the long memory latency. Mainstream architectures include a two-level or three-level cache memory hierarchy to reduce the average memory latency, but mostly GPUs are designed for stream or throughput computing, where cache memories are ineffective. Instead, GPUs tolerate memory latency by using a high degree of multithreading. When one warp stalls on a memory operation, the multiprocessor control unit selects another ready warp and switches to it. By this way, cores can be productive as long as there is enough parallelism to keep them busy.
Most GPU programming is based on the C language and its extensions [16] [17] [18] [19]. In the wider context, having a background in parallel computing techniques (threading, message passing, and vectorization) will help one understand and apply GPU acceleration.

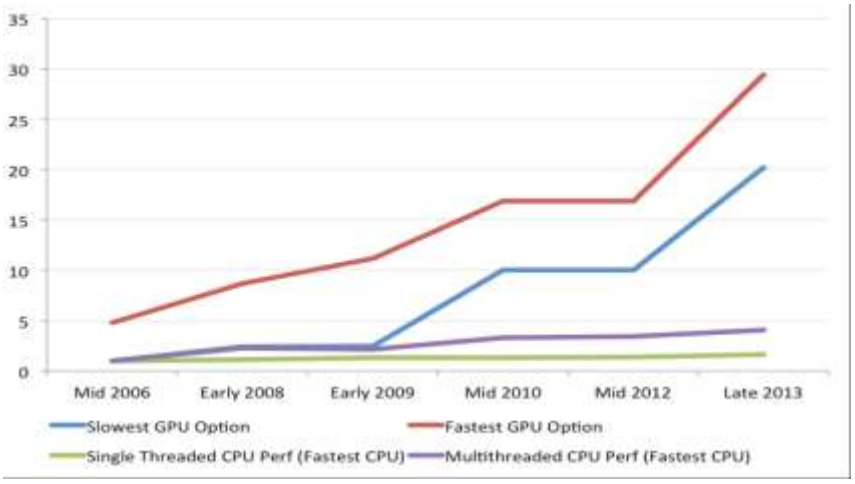
CUDA or "Compute Unified Device Architecture" is an NVIDIA SDK and associated toolkit for programming GPUs to perform general purpose computation, implemented as an extension to standard C. As well as a fully featured API for parallel programming on the GPU, CUDA includes standard numerical libraries, including FFT (Fast Fourier Transform) and BLAS (Basic Linear Algebra Subroutines), a visual profiler, and numerous examples illustrating the use of the CUDA in a wide variety of applications.

NVIDIA GPUs are programmed as a sequence of kernels. Typically, each kernel completes execution before the next kernel begins, with implicit barrier synchronization between kernels. Keplar has support for multiple, independent kernels to execute simultaneously, but many kernels are large enough to fill the entire machine. As mentioned, the multiprocessors execute in parallel, asynchronously. However, GPUs do not support a fully coherent memory model that would allow the multiprocessors to synchronize with each other. Classical parallel programming techniques can't be used here. Threads can spawn more threads on Keplar GPUs, so nested parallelism is supported. However, threads on one multiprocessor can't send results to threads on another multiprocessor; there's no facility for a critical section among all the threads of the whole system.

CUDA offers a data parallel programming model that is supported on NVIDIA GPUs. In this model, the host program launches a sequence of kernels, and those kernels can spawn sub-kernels.

Threads are grouped into blocks, and blocks are grouped into a grid. Each thread has a unique local index in its block, and each block has a unique index in the grid. Kernels can use these indices to compute array subscripts, for instance.
Threads in a single block will be executed on a single multiprocessor, sharing the software data cache, and can synchronize and share data with threads in the same block; a warp will always be a subset of threads from a single block. Threads in different blocks may be assigned to different multiprocessors concurrently, to the same multiprocessor concurrently (using multithreading), or may be assigned to the same or different multiprocessors at different times, depending on how the blocks are scheduled dynamically. There is a hard upper limit on the size of a thread block, 1,024 threads or 32 warps for Keplar. Thread blocks are always created in warp-sized units, so there is little point in trying to create a thread block of a size that is not a multiple of 32 threads; all thread blocks in the whole grid will have the same size and shape. A Keplar multiprocessor can have 2,048 threads simultaneously active, or 64 warps. These can come from 2 thread blocks of 32 warps, or 3 thread blocks of 21 warps, 4 thread blocks of 16 warps, and so on up to 16 blocks of 4 warps; there is another hard upper limit of 16 thread blocks simultaneously active on a single multiprocessor.
Performance tuning on NVIDIA GPUs requires optimizing all these architectural features:

- Finding and exposing enough parallelism to populate all the multiprocessors.
- Finding and exposing enough additional parallelism to allow multithreading to keep the cores busy.
- Optimizing device memory accesses for contiguous data, essentially optimizing for stride-1 memory accesses.
- Utilizing the software data cache to store intermediate results or to reorganize data that would otherwise require non-stride-1 device memory accesses.



VI. LITERATURE REVIEW

According to literature survey we found severalimplementations of the popular K-means data clustering algorithm for GPUs exist. One common restriction of thepublished approaches, however, is the limitation of reporteddimensionality of test data to small values of 60 or below. Tothe best of our knowledge there are two GPU implementationsof k-means are available which are taken as a benchmark for allcomparisons. We have studied both the approaches andfound some observations as follows:

GPUMiner stores all input data in the global memory, andloads k centroids to the shared memory. Each block has 128threads, and the grid has n/128 blocks. The maincharacteristic of GPUMiner is the design of a bitmap. Theworkflow of GPUMiner is as follows. First, each threadcalculates the distance from one data point to every centroid,and changes the suitable bit into true in the bit array, whichstores the nearest centroid for each data point. Second, eachthread is responsible for one centroid, finds all thecorresponding data points from the bitmap and takes themean of those data points as the new centroids.The main problem of GPUMiner is the poor utilization ofmemory in GPU, since GPUMiner accesses most of the data(input data point) from global memory directly. As pointedout in [9], bitmap approach is elegant in expressing theproblem, but not a good method for high performance, sincebitmap takes more space when k is large and requires moreshared memory.

In order to avoid the long-time latency of globalmemory access, UV_k-Means copies all the data to thetexture memory, which uses a cache mechanism. Then, it usesconstant memory to store the k centroids, which is also moreefficient than using global memory. Each thread isresponsible for finding the nearest centroid of a data point;each block has 256 threads, and the grid has n/256 blocks.

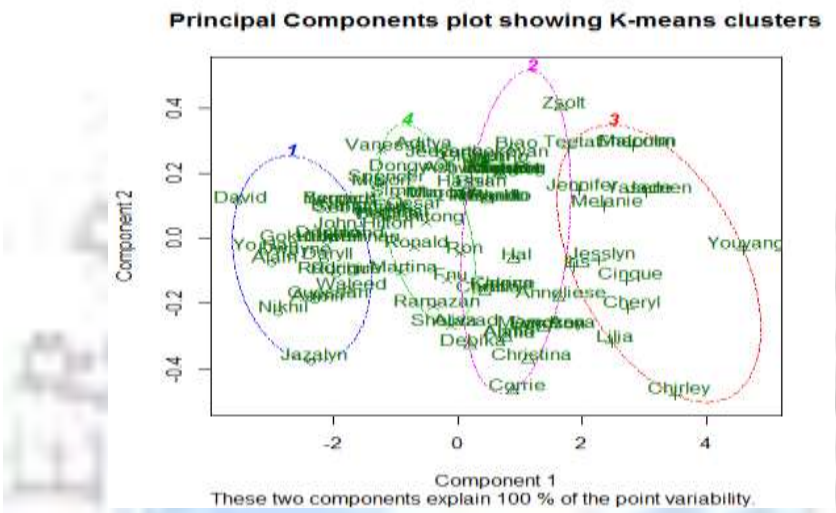
The work flow of UV_k-Means is straightforward.

First, each thread calculates the distance from onecorresponding data point to every centroid and finds theminimum distance and corresponding centroid. Second, eachblock calculates a temporary centroid set based on a subsetof data points, and each thread calculates one dimension ofthe temp centroid. Third, the temporal centroid sets arecopied from GPU to CPU, and then the final new centroid setis calculated on CPU.
UV_k-Means has achieved a speedup of twenty toforty over the single-thread CPU-based k-Means in experiment. This speedup is mainly achieved by assigningeach data point to one thread and utilizing the cachemechanism to get a high reading efficiency. However, theefficiency could be further improved by other memory accessmechanisms such as registers and shared memory.

Some other implementations of k-means are alsoshows considerable speedup on GPU like Hall and Hartpropose two theoretical options for solving the problem oflimited instance counts and dimensionality: multi-passlabelling and a different data layout within the texture.
None of the approaches have been implemented though. Inaddition to the naive k-means implementation the data is reordered to minimize the number of distance calculations byonly calculating the metrics to the nearest centroids. This isachieved by finding those centroids by traversing apreviously constructed kd-tree. The authors could notobserve any problems caused by the non-standard compliantfloating point arithmetic implementations on the GPU, statingthat the exact same clustering has been found.
The approach of Cao et. al. in differs in that thecentroid indices are stored in an 8-bit stencil buffer instead ofthe frame buffer limiting the number of total centroids to 256.

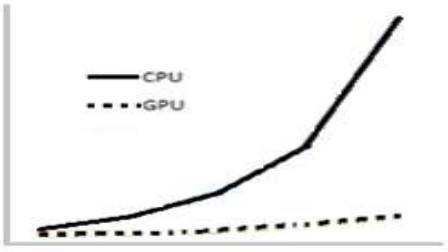
VII. CONCLUSION AND FUTURE WORK

It can be deduced by literature review and experiments that for faster clustering we need to approach the problem with following solutions: maximizing parallelism and load balance, Minimizing thread divergence &Minimizing main memory accesses.To hide latencies, particularly memory latencies, it is important to run a large number of threads and blocks in parallel. By parallelizing, several steps of our algorithm across threads and blocks as well as by using table elements instead of heap objects to represent tree nodes, we were able to assign balanced amounts of work to any number of threads and blocks.Parallelize an elementary data processing operation used by many applications on a highly parallel graphics processing architecture. Computationally bound applications have much to gain by using the idle resources offered by the system through the CUDA architecture. As performance and energy consumption become a prime concern for computer architectures, we are bound to see more applications of off-chip parallel computing in every computing domain from large-scale distributed systems to portable computers.The GPU with CUDA parallel computing architecture will provide compelling benefits for data mining applications.Also, its superior floating-point computationcapability and low cost will definitely appeal to medium sizedbusiness and individuals. Applications that usedto rely on a cluster or a supercomputer to process will besolved on a desktop.



On an x-core PC, it is bestto have x threads each working on n/x points. In CUDA, there is no thread dispatching overhead. Assign each data point to one thread.
to maintain maximum parallelism, it is important for the threads in a warp to follow the same control flow. We found that grouping similar work together can drastically reduce thread divergence, especially in irregular code.

probably the most important optimization for memory-bound code is to reduce main memory accesses as much as possible, for example, by caching data and by throttling warps that are likely to issue memory accesses that do not contribute to the overall progress.



The figure depicts relationship between data points on x-axis and time in seconds on y-axis; number of clusters being constant.

For N=100, d=3, K=4 clusters

Platform	Time	Speedup
Intel Pentium D2.0 ghz	2.999sec	1x
NVidiaGeForce 8800	0.470 sec	6x

Nearly all complex and time-cost computation of k-meanscan be speedup substantially by offloading work to GPU.The CUDA technology used is modern GPGPUarchitecture, which is adopted by many NVIDIA GPUs.As current trends indicated, future GPU designs, alsobased on general purpose multiprocessors, will offereven more computational power.

REFERENCES

1. G Karypis, V Kumar, M Steinbach, "A comparison of document clustering techniques" In KDD Workshop on Text Mining, 2009.
2. AK Jain, MN Murty, PJ Flynn, "Data clustering: a review", in ACM Computing Surveys (CSUR), Pages 264-323, Volume 31 Issue 3, Sept. 1999.
3. 3. JA Hartigan, MA Wong, "Algorithm AS 136: A k-means clustering algorithm", in Journal of the Royal Statistical Society. Series C (Applied Statistics), Pages 100-108, 1979.
4. T Kanungo, DM Mount, NS Netanyahu, "An efficient k-means clustering algorithm: Analysis and implementation", in IEEE Transactions on Pattern Analysis and Machine Intelligence2002.
5. A Vattani, "K-means requires exponentially many iterations even in the plane", in twenty-fifth annual symposium on Computational geometry, pages 324-332,2011.
6. RT Ng, J Han, "Efficient and Effective Clustering Methods for Spatial Data Mining", in 20th International Conference on Very Large Data Bases, pages 144-155, 1994.
7. P Berkhin, "A survey of clustering data mining techniques", in Grouping multidimensional Data, recent advances in Clustering, pages 25-71, 2006.
8. T Zhang, R Ramakrishnan, M Livny, "BIRCH: an efficient data clustering method for very large databases", in ACM SIGMOD international conference on Management of data, Volume 25 Issue 2, Pages 103-114, 1996.
9. www.top500.org
10. JB MacQueen, "Some Methods for classification and Analysis of Multivariate Observations" in 5-th Berkeley Symposium on Mathematical Statistics and Probability,1967.
11. SP Lloyd, "Least square quantization in PCM" in IEEE Transactions on Information Theory, Volume 28 Issue 2, Page 129-137,1982.
12. G Hamerly, C Elkan, "Alternatives to the k-means algorithm that find better clustering", in Proceedings of the eleventh international conference on Information and knowledge management, pages 600-607,2002.
13. Ren Wu, Bin Zhang, Meichun Hsu, "Clustering Billions of Data Points Using GPUs", in UCHPC-MAW '09 Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop pages 1-6,2009.
14. D Judd, PK McKinley, AK Jain, "Large-Scale Parallel Data Clustering" in IEEE Transactions on Pattern Analysis and Machine Intelligence1998.
15. IS Dhillon, DS Modha, "A Data-clustering Algorithm on Distributed Memory Multiprocessors" in Large-Scale Parallel Data Mining, Workshop on Large-Scale Parallel KDD Systems, SIGKDD, pages 245-260, 2000.
16. BrookGPU, "http://graphics.stanford.edu/projects/brookgpu/".
17. Cg, "https://developer.nvidia.com/cg-toolkit".
18. GLSL,"http://www.opengl.org/documentation/oglsl.html".
19. HLSL,"http://msdn.microsoft.com/library/en-us/directx9_"
20. J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters" in Communications of the ACM - 50th anniversary issue: 1958 - 2008 CACM Homepage archiveVolume 51 Issue 1, pages 107-113,2004.
21. N.-L. Tran and S. Skhiri, "AROM: Processing Big Data with Data Flow Graphs and Functional Programming" in Cloud Computing Technology and Science (CloudCom), IEEE 4th International Conference,2012.
22. H Xiao, "Towards Parallel and Distributed Computing in Large-Scale Data Mining: A Survey", 2010.
23. Estivill-Castro, Vladimir, "Why so many clustering algorithms – A Position Paper" in ACM SIGKDD Explorations Newsletter Homepage archive Volume 4 Issue 1, pages 65-75, 2002.
24. R Sibson, "An optimally efficient algorithm for the single-link cluster method – SLINK" in The Computer Journal, pages 30-34, 1973.
25. D Defays, "An efficient algorithm for a complete link method" in The Computer journal, pages 264-266, 1977.
26. Kreigel, Hans-Peter, Kroger, Peer, Sander, Jorg, Zimek, Arthur, "Density-Based Clustering" in Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, Volume 1, Issue 3, pages 231–240, May/June 2011.
27. Microsoft academic search: most cited data mining articles: - DBSCAN is on 24th Position.
28. Ankerst, Mihael, Breunig, M Markus, Kriegel, Hans-Peter, Sander, Jorg, "OPTICS: Ordering Points To Identify the Clustering Structure" in Proceedings of the 1999 ACM SIGMOD international conference on Management of data, pages 49-60 Volume 28 Issue 2, June 1999.
29. S Roy, Bhattacharyya, D. K., "An Approach to find Embedded Clusters Using Density Based Techniques" in Lecture Notes in Computer Science Volume 3816, 2005, pp 523-535, 2005.
30. NVidia CUDA, "http://developer.nvidia.com/object/CUDA.html".

31. AMD CTM, "http://ati.amd.com/companyinfo/researcher/documents.html".
32. David Luebke and Greg Humphreys, "How gpus work" in Computer, Volume 40, Issue 2, pages 96-100, Feb. 2007.
33. D. Arthur and S. Vassilvitskii, "k-means++: the advantages of carefull seeding" in eighteenth annual ACM-SIAM symposium on Discrete algorithms, pages 1027-1035, 2007.