

International Journal of Computer Science and Mobile Computing



A Monthly Journal of Computer Science and Information Technology

ISSN 2320-088X

IJCSMC, Vol. 4, Issue. 1, January 2015, pg.555 – 561

REVIEW ARTICLE

Review of K-means Clustering Algorithm on GPU

Harshal A. Bhavsar

PG Student Department of Computer Engineering, MCOERC, Nashik

Savitribai Phule Pune University, India

hb2708@gmail.com

Abstract — *K-Means is the most popular clustering algorithm in data mining. K-Means is the most popular clustering algorithm in data mining. The size of various data sets has increased tremendously day by day. Due to recent development in the shared memory inexpensive architecture like Graphics Processing Units (GPU). 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. Cluster analysis plays a critical role in a wide variety of applications, but it is now facing the computational challenge due to the continuously increasing data volume. Parallel computing is one of the most promising solutions to overcoming the computational challenge.*

Keywords— *Clustering, k-means, Graphics Processing Unit (GPU), Compute Unified Device Architecture (CUDA), Data Mining*

I. INTRODUCTION

Graphics processors (GPUs) have developed very rapidly in recent years. GPUs have moved beyond their originally targeted graphics applications and increasingly become a feasible choice for general purpose computing. Nowadays, most desktop computers are equipped with programmable graphics processing units (GPUs) with many powerful Single Instruction Multiple Data (SIMD) processors that can support parallel data processing and high-precision computation. The rapid advance in GPUs performance, coupled with recent improvements in its programmability, made it possible to parallelize k-means on personal computers. CUDA technology gives computationally intensive applications access to the tremendous processing power of the latest GPUs through a C-like programming interface.

As a general-purpose and high performance parallel hardware, Graphics Processing Units (GPUs) develop continuously and provide another promising platform for parallelizing k-Means. GPUs are dedicated hardware for manipulating computer graphics. Due to the huge computing demand for real-time and high-definition 3D graphics, GPUs have evolved into highly parallel many-core processors. The advances of computing power and memory bandwidth in GPUs have driven the development of general-purpose computing on GPUs (GPGPU)

The paper provides a brief literature review many implementation of K-Means clustering algorithm on GPUs using CUDA that have been introduced till date.

II. RELATED WORK

A. K-Means

K-Means is a commonly used clustering algorithm it's popularly used for data mining. Clustering is a means of arranging n data points into k clusters where each cluster has maximum 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. The algorithm assigns each point to the cluster whose center is closest. 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 [3]:

- 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)

Algorithm 1 Sequential K-Means Algorithm

```

 $c_j \leftarrow \text{random } x_i \in \mathcal{X}, j = 1, \dots, k, \text{ s.t. } c_j \neq c_i \forall i \neq j$ 
repeat
   $C_j \leftarrow \emptyset, j = 1, \dots, k$ 
  for all  $x_i \in \mathcal{X}$  do
     $j \leftarrow \arg \min D(c_j, x_i)$ 
     $C_j \leftarrow C_j \cup x_i$ 
  end for
  for all  $c_j \in \mathcal{C}$  do
     $c_j \leftarrow \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i$ 
  end for
until convergence

```

An initial clustering C is created by choosing k random Centroids from the set of data points X . This is known as the seeding stage. Next a labelling stage is executed where each data point $x_i \in X$ is assigned to the cluster C_j for which $D(x_i, c_j)$ is minimal. Each Centroid c_j is then recalculated by the mean of all data points $x_i \in C_j$. The labelling and Centroid update stage are executed repeatedly until C no longer changes [3].

B. Parallel K-Means

1) Data objects assignment:

Data objects assignment and k Centroid recalculation is the most intensive arithmetic task load of k -means. There are two strategies in data objects assignment process suited to GPU-based k -means. The first is the Centroid-oriented, in which distance from each Centroid to all data objects are calculated and then, each data point will merge itself into the cluster represented by nearest Centroid. This method has advantages when the number of processors of GPU is relatively small so that every processor can deal with data objects in series. Another is the data objects-oriented, namely, each data point calculates the distance from all Centroid, and then data object will be assigned to the cluster represented by the Centroid with the shortest distance from it [6].

In k -means algorithm, every data point must choose the nearest Centroid after calculating all the distances, this selecting process consists a series of comparison which could be carried out through Deep Buffer in early GPUs. In this way, the latency of memory access could be avoided while one thread is waiting for memory access, and other threads will be optimized to use the arithmetic resources

2) K- Centroids recalculation:

The new Centroid is the arithmetic means of all data objects. The positions of the k Centroids are also parallel recalculated by GPU and every thread is responsible for a new Centroid. After data objects assignment, we get the cluster label of every data point. A straightforward idea for recalculating the position of one Centroid is to read all data objects and determine whether the data point belongs to this Centroid or not. Unfortunately, massive condition statements are not suitable to the stream processor model of GPUs. We add another procedure that the cluster labels are downloaded from the device (GPU) to the host (CPU) and the host rearranges all data objects and counts the number of data objects

contained by each cluster. And then, both structures are uploaded to the global memory of the device. In this way, every thread of CUDA kernel can complete its task by reading its own data objects continuously [6].

3) Algorithm:

The labelling 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 . Since the GPU is a shared memory multiprocessor architecture this section briefly outlines a parallel implementation on such a machine. It only slightly diverges from the approach proposed by Dhillon. Processors are now called threads and a master-slave model is employed. Each thread is assigned an identifier between 0 and $t - 1$ where t denotes the number of threads. Thread 0 is considered the master thread, all other threads are slaves. Threads share some memory within which the set of data points X , the set of current centroids C as well as the clusters C_j reside. Each thread additionally owns local memory for miscellaneous data. It is further assumed that locking mechanisms for concurrent memory access are available. Given this setup the sequential algorithm can be mapped to this programming model as follows. The master thread initializes the centroids as it is done in the sequential version of k -means.

Next X is partitioned into subsets X_i ; $i = 0; \dots; t$. This is merely an offset and range calculation each thread executes giving those x_i each thread processes in the labelling stage. All threads execute the labelling stage for their partition of X . The label of each data point x_i is stored in a component l_i of an n -dimensional vector. This eliminates concurrent writes when updating clusters and simplifies bookkeeping. After the labelling stage the threads are synchronized to ensure that all data for the centroid update stage is available. The centroid update stage could then be executed by a reduction operation. However, for the sake of simplicity it is assumed that the master thread executes this stage sequentially. Instead of iterating over all centroids the master thread iterates over all labels partially calculating the new centroids. A k -dimensional vector m is updated in each iteration where each component m_j holds the number of data points assigned to cluster C_j . Next another loop over all centroids is performed scaling each centroid c_j by $1/m_j$ giving the final centroids. Convergence is also determined by the master thread by checking whether the last labelling stage introduced any changes in the clustering. Slave threads are signalled to stop execution by the master thread as soon as convergence is achieved. Algorithm 2 describes the procedure executed by each thread.

Algorithm 2 Parallel K-Means Algorithm

```

if threadIdx = 0 then
     $c_j \leftarrow \text{random } x_i \in \mathcal{X}, j = 1, \dots, k, \text{ s.t. } c_j \neq c_i \forall i \neq j$ 
end if
synchronize threads
repeat
    for all  $x_i \in \mathcal{X}_{\text{threadId}}$  do
         $l_i \leftarrow \arg \min D(c_j, x_i)$ 
    end for
synchronize threads
if threadIdx=0 then
    for all  $x_i \in \mathcal{X}$  do
         $c_{l_i} \leftarrow c_{l_i} + x_i$ 
         $m_{l_i} \leftarrow m_{l_i} + 1$ 
    end for
    for all  $c_j \in C$  do
         $c_j \leftarrow \frac{1}{m_j} c_j$ 
    end for
    if convergence then
        signal threads to terminate
    end if
end if
until convergence

```

C. Graphics Processing Unit

NVIDIA's Tesla unified computing architecture is designed to support both graphics and general purpose computing. The programmable processing elements share a common, very general-purpose instruction set that is used by both graphics and general-purpose computation. Each processing element (PE) supports 128 concurrent thread contexts, allowing a very simple pipeline. Latencies are simply tolerated by switching threads. Current Tesla-architecture products can support up to 30720 concurrent threads. Although it describes the previous generation GeForce 8800 GTX and related products. Lindholm et al. [11] provide a nice description of contemporary NVIDIA GPU architectures.

Each SM consists of 8 processing elements, called Stream Processors or SPs. To maximize the number of processing elements that can be accommodated within the GPU die, these 8 SPs operate in SIMD fashion under the control of a single instruction sequencer. The threads in a thread block (up to 512) are time-sliced onto these 8 SPs in groups of 32 called warps. Each warp of 32 threads operates in lockstep and these 32 threads are quad-pumped on the 8 SPs. Multithreading is then achieved through a hardware thread scheduler in each SM. Every cycle this scheduler selects the next warp to execute. Divergent threads are handled using hardware masking until they re-converge. Different warps in a thread block need not operate in lockstep, but if threads within a warp follow divergent paths, only threads on the same path can be executed simultaneously. In the worst case, if all 32 threads in a warp follow different paths without reconverging, effectively resulting in a sequential execution of the threads across the warp. A 32 * penalty will be incurred. Unlike vector forms of SIMD, Tesla's architecture preserves a scalar programming model, like the Illiac or Maspar architectures; for correctness the programmer need not be aware of the SIMD nature of the hardware, although optimizing to minimize SIMD divergence will certainly benefit performance.

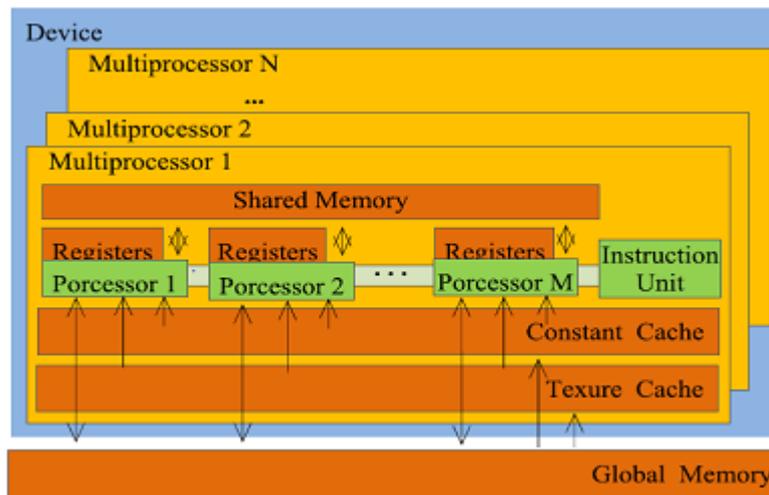


Fig. 1 GPU Architecture

When a kernel is launched, the driver notifies the GPU's work distributor of the kernel's starting PC and its grid configuration. As soon as an SM has sufficient thread and PBSM resources to accommodate a new thread block, a hardware scheduler randomly assigns a new thread block and the SM's hardware controller initializes the state for all threads (up to 512) in that thread block. The Tesla architecture is designed to support workloads with relatively little temporal data locality and only much localized data reuse. As a consequence, it does not provide large hardware caches which are shared among multiple cores, as is the case on modern CPUs. In fact, there is no cache in the conventional sense: variables that do not fit in a thread's register file are spilled to global memory. Instead, in addition to the PBSM, each SM has two small, private data caches, both of which only hold read-only data: the texture cache and the constant cache. (The name texture comes from 3D graphics, where images which are mapped onto polygons are called textures.) Data structures must be explicitly allocated into the PBSM, constant, and texture memory spaces. The texture cache allows arbitrary access patterns at full performance. It is useful for achieving maximum performance on coalesced access patterns with arbitrary offsets. The constant cache is optimized for broadcasting values to all PEs in an SM and performance degrades linearly if PEs requests multiple addresses in a given cycle. This limitation makes it primarily useful for small data structures which are accessed in a uniform manner by many threads in a warp.

D. Compute Unified Device Architecture

We have shown the GPU's potential to support interesting applications with diverse performance characteristics. In the course of developing these applications, we made many observations about the CUDA programming model. Threads in CUDA are scalar, and the kernel is therefore a simple scalar program, without

the need to manage vectorization, packing, etc. as is common in some other programming models. In fact, in CUDA data accesses do not need to be contiguous at all, that is to say each thread can access any memory location and still obtain the benefits of SIMD execution as the instruction sequence stays in lockstep within a warp. Although non-contiguous memory references may reduce effective memory bandwidth, this is only a concern for applications that are memory bound. Even in that case, packing is not a prerequisite for working code, but rather an optimization step, which dramatically reduces the software development burden. The CUDA model is not a purely data-parallel model. For example, programmers can specify task parallelism within a warp, but they must keep in mind that this might cause a severe performance penalty due to thread divergence. Alternatively, task parallelism can be specified between warps within a thread block, but programs are limited to synchronizing all warps via syncthreads () thread blocks can perform different work, but cannot have producer-consumer relationships except across kernel calls.

Barrier synchronization is widely perceived as inefficient, but can actually be more efficient than a large quantity of fine-grained synchronizations. Barriers are mainly detrimental in those cases where the program is forced to synchronize all threads to satisfy the needs of only a few. It is not clear how often this is a concern. Barriers also provide a much simpler abstraction to the programmer. These tradeoffs are poorly understood when the synchronization occurs on chip with hardware synchronization primitives. Currently, programmers must specify the number of working threads explicitly for a kernel, and threads cannot fork new threads. Often some thread resources are wasted, as in our Needleman-Wunsch implementation. Add to these limitations a lack of support for recursion, and the interface is missing a set of powerful, key abstractions that could hinder their uptake as programmers struggle to restructure their old code as CUDA programs.

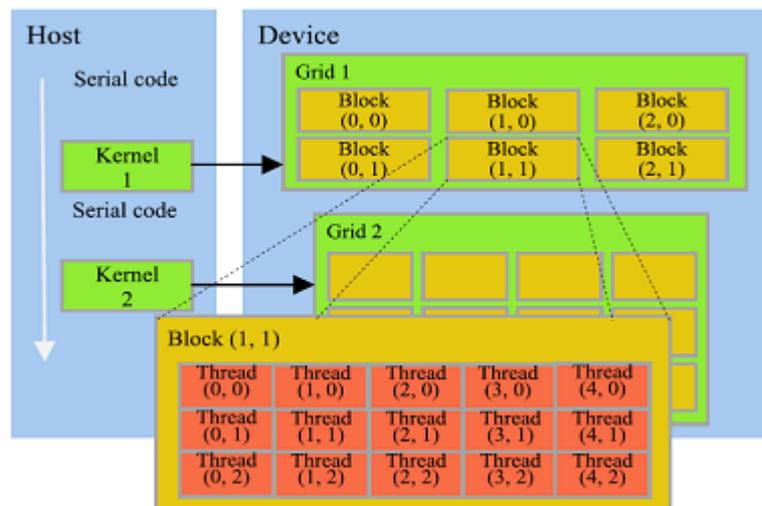


Fig. 2 CUDA Architecture

Lack of persistent state in the per-block shared memory results in less efficient communication among producer and consumer kernels than might be otherwise possible. The producer kernel has to store the shared memory data into device memory; the data is then read back over the bus by the consumer kernel. This also undercuts the efficiency of global synchronization which involves kernel termination and creation; however, a persistent shared memory contradicts the current programming model, in which thread blocks run to completion and by definition leave no state afterwards. Alternatively, a programmer can choose to use a novel algorithm that involves less communication and global synchronization, such as the pyramid algorithm that we use in HotSpot, but this often increases program complexity. CUDA's performance is hurt by its inability to collect data from a set of producer threads and stream them to a set of consumer threads. Intermediate data has to be stored in device memory before it is consumed by another thread in a new kernel.

III. LITERATURE REVIEW

S.A. Arul Shalom et al.[7] present an efficient implementation of the k-means clustering algorithm completely in the GPU. They realize this by using the multipass rendering and multi-shader capabilities of the GPU. This is done by maximizing the use of textures and minimizing the use of shader program constants. In this implementation they have minimized the use of GPU shader constants thus improving the performance as well as reducing the data transactions between the CPU and the GPU. Handling data transfers between the necessary textures within the GPU is much more efficient than using shader constants. This is mainly due to the high memory bandwidth available in the GPU pipeline. Since all the steps of k-means clustering

could be implemented in the GPU, the transferring of data back to the CPU during the iterations is avoided. The programmable capabilities of the GPU have been thus exploited to efficiently implement k-means clustering in the GPU. Implementation is done using OpenGL as the Application Programming Interface (API), and the operational kernels are invoked via shader programs, using the Graphics Library Shading Language (GLSL). The Single Instruction Multiple Data (SIMD) technique is employed to achieve data or vector level parallelism in the fragment processor.

Reza Farivar et al.[3] have used two types of data structures. They have implemented the CUDA-accelerated k-means algorithm in three distinct stages of operation. The 1st stage initializes the CUDA hardware, allocates the appropriate host and device memory storage areas, estimates the initial set of centroids and loads the data set into the on-board memory of the graphics card. The second part, which is the workhorse of the program, is the kernel running on the GPU device. Each thread will process a single data point, and compute the distance between the point and each centroid. The third part of the program relabels points to the nearest centroid, and computes the next centroid estimation. This part is executed serially in the host. The data set consists of a one dimensional array of unsigned long integers, which are 4 bytes in size per point.

BAI Hong-Tao et al.[6] have proposed a novel Single Instruction Multiple Data (SIMD) architecture processors (GPUs) based k-means algorithm. In this algorithm, in order to accelerate compute-intensive portions of traditional k-means, both data objects assignment and k centroids recalculation are off-loaded to the GPU in parallel. They have implemented this GPU-based k-means on the newest generation GPU with Compute Unified Device Architecture (CUDA). The numerical experiments demonstrated that the speed of GPU-based k-means could reach as high as 40 times of the CPU-based k-means.

Mario Zechner et al.[5] have proposed the algorithm which 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. The CPU takes the role of the master thread. As a first step it prepares the data points and uploads them to the GPU. As the data points do not change over the course of the algorithm they are only transferred once. The CPU then enters the iterative process of labelling the data points as well as updating the centroids. Each iteration starts by uploading the current centroids to the GPU. Next the GPU performs the labelling. The results from the labelling stage, namely the membership of each data point to a cluster in form of an index, are transferred back to the CPU. Finally the CPU calculates the new centroid of each cluster based on these labels and performs a convergence check. Convergence is achieved in case no label has changed compared to the last iteration. Optionally a threshold difference check of the overall movement of the centroids can be performed to avoid iterating infinitely for some special cluster configurations.

Ren Wu et al.[4] have proposed parallel k-means clustering using GPUs to accelerate clustering of very large data sets. They investigate if GPUs can be useful accelerators even with very large data sets that cannot fit into GPUs on-board memory. They have used MineBench as their baseline for performance comparison and used randomly generated data sets. They have introduced Multi-tasked Streaming for large datasets.

Kai J. Kohlhoff et al.[1] have proposed how K-means can be fully implemented on massively parallel general purpose computing platforms without putting limits on the number of data points, clusters, or dimensionality of the data other than the available GPU main memory, while avoiding thread divergence and maintaining near-optimal GPU occupancy. This is achieved by introducing a fast parallel sorting algorithm based on parallel-prefix-sum followed by even faster resorting with an updating step in subsequent iterations.

IV. CONCLUSIONS

The version of this template is V2. Most of the formatting instructions in this document have been compiled by Causal Productions from the IEEE LaTeX style files. Causal Productions offers both A4 templates and US Letter templates for LaTeX and Microsoft Word. The LaTeX templates depend on the official IEEEtran.cls and IEEEtran.bst files, whereas the Microsoft Word templates are self-contained. Causal Productions has used its best efforts to ensure that the templates have the same appearance.

This paper is based on k-means clustering algorithm. In this the study of different implementations of k-means clustering on GPU is studied. Nearly all complex and time-cost computation of k-means can be speedup substantially by offloading work to GPU. The CUDA technology used is modern GPGPU architecture, which is adopted by many NVIDIA GPUs. As current trends indicated, future GPU designs, also based on general purpose multiprocessors, will offer even more computational power. Exploiting the GPU for the labelling stage of k-means proved to be beneficial especially for large data sets and high cluster counts. The GPU with CUDA parallel computing architecture will provide compelling benefits for data mining applications. In addition, its superior floating-point computation capability and low cost will definitely appeal to medium sized business and individuals. Applications that used to rely on a cluster or a supercomputer to process will be solved on a desktop.

REFERENCES

- [1] Kai J. Kohlhoff, Vijay S. Pande, and Russ B. Altman, "K-Means for Parallel Architectures Using All-Prefix-Sum Sorting and Updating Steps", *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, VOL. 24, NO. 8, AUGUST 2013.
- [2] Liheng Jian, ChengWang, Ying Liu, Shenshen Liang, Weidong Yi, Yong Shi, "Parallel data mining techniques on Graphics Processing Unit with Compute Unified Device Architecture(CUDA) ", *The Journal of Supercomputing Springer* , Volume 64, Issue 3, June 2013.
- [3] R. Farivar, D. Rebolledo, E. Chan, and R. Campbell, "A Parallel Implementation of K-Means Clustering on GPUs," *Proc. Int'l Conf. Parallel and Distributed Processing Techniques and Applications*, 2008.
- [4] R. Wu, B. Zhang, M. Hsu, and Clustering, "Billions of Data Points Using GPUs," *Proc. Combined Workshops Unconventional High Performance Computing Workshop Plus Memory Access Workshop (UCHPCMAW '09)*, 2009, doi: 10.1145/1531666.1531668.
- [5] M. Zechner and M. Granitzer, "Accelerating K-Means on the Graphics Processor via CUDA," *Proc. First Int'l Conf. Intensive Applications and Services (INTENSIVE '09)*, pp. 7-15 , 2009, doi:10.1109/INTENSIVE.2009.19)
- [6] Bai, L. He, D. Ouyang, Z. Li, and H. Li, "K-Means on Commodity GPUs with CUDA," *Proc. WRI World Congress Computer Science and Information Eng*, vol. 3, pp. 651 -655, 2009, doi:10.1109/CSIE.2009.491)
- [7] S.A.A. Shalom, M. Dash, and M. Tue, "Efficient K-Means Clustering Using Accelerated Graphics Processors," *Proc. 10th Int'l Conf. Data Warehousing and Knowledge Discovery I. Song, J. Eder, and T. Nguyen, eds.*, pp. 166-175, 2008, doi: 10.1007/978-3-540-85836-2_16
- [8] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: An Efficient Data Clustering Method for Very Large Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 103 -114, 1996, doi: 10.1145/235968.233324.
- [9] S. Che, J. Meng, J.W. Sheaffer, and K. Skadron, "A Performance Study of General Purpose Applications on Graphics Processors," *Proc. First Workshop General Purpose Processing on Graphics Processing Units*, 2007
- [10] You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu, "Speeding up K-Means Algorithm by GPUs".
- [11] Lindholm, Erik, John Nickolls, Stuart Oberman, and John Montrym. "NVIDIA Tesla: A unified graphics and computing architecture." *Ieee Micro* 28, no. 2 (2008): 39-55.