www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 3 Issue 5 May, 2014 Page No. 5500-5605

K-Means on GPU: A Review

Vidya Dhamdhere, Rahul G. Ghudji

Rahul Ghudji is with G. H. Raisoni College Of Engineering and Management, Pune, India

B. Padmavathi is Assistant Professor, GHRCEM, Pune University, Wagholi, Pune, India.

Abstract: ***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.***

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

1. Introduction

Graphics processors (GPUs) have developed very rapidly in recent years. GPUs have moved beyond their originally-targeted graphics applications and increasingly become a viable choice for general purpose computing. Nowadays, most desktop computers are equipped with programmable graphics processing units (GPUs) with plenty 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 [1].

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) [10].

The paper provides a brief literature review of all versions of K-Means implementation on GPUs using CUDA that have published till date. Section II brief describes K-Means with its parallel implementation details on GPU. Section III explains literature review of K-Means on GPUs. Finally, Section IV discusses the findings of this review work and future scope for improvement.

2. RELATED WORK

2.1 K-Mean

K-Means is a commonly used clustering algorithm used for data mining. Clustering is a means of arranging n data points into k clusters where each cluster has maximal similarity as defined by an objective function. Each point may only belong to one cluster, and the union of all clusters contains all n points. The algorithm assigns each point to the cluster whose center is nearest. The center is the average of all the points in the cluster that is, its coordinates are the arithmetic mean for each dimension separately over all the points in the cluster. The algorithm steps are [4]:

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

$\mathbf{c_j} \leftarrow$ random $\mathbf{x_i} \in \mathcal{X}, j = 1, \ldots, k$, s.t. $\mathbf{c_j} \neq \mathbf{c_i} \forall i \neq j$
repeat
  $\mathcal{C}_j \leftarrow \emptyset, j = 1, \ldots, k$
  for all $\mathbf{x_i} \in \mathcal{X}$ **do**
    $j \leftarrow \arg\min \mathcal{D}(c_j, x_i)$
    $\mathcal{C}_j \leftarrow \mathcal{C}_j \cup x_i$
  end for
  for all $\mathbf{c_j} \in \mathcal{C}$ **do**
    $\mathbf{c_j} \leftarrow \frac{1}{|\mathcal{C}_j|} \sum_{\mathbf{x_i} \in \mathcal{C}_j} \mathbf{x_i}$
  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 labeling stage is executed where each data point x_i ε X is assigned to the cluster Cj for which D(xi, cj) is minimal. Each centroid cj is then recalculated by the mean of all data points xi ε Cj. The labelling and centroid update stage are executed repeatedly until C no longer changes [2].

2.2 Parallel K-Mean

2.2.1 Data objects assignment

Data objects assignment and k centroids recalculation are the most intensive arithmetic task load of k-means. There are two strategies in data objects assignment process suited to GPU-based kmeans. The first is the centroids-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 centroids, then data object will be assigned to the cluster represented by the centroid with the shortest distance from it [5].

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 [3].

2.2.2 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 [7].

2.2.3 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 Cj 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 Xi; i = 0; : : : t. This is merely an offset and range calculation each thread executes giving those xi each thread processes in the labeling stage. All threads execute the labeling stage for their partition of X. The label of each data point xi is stored in a component li 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 mj holds the number of data points assigned to cluster Cj . Next another loop over all centroids is performed scaling each centroid cj by 1/mj giving the final centroids. Convergence is also determined by the master thread by checking whether the last labeling stage introduced any changes in the clustering. Slave threads are signaled 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 threadId = 0 **then**
  $\mathbf{c_j} \leftarrow$ random $\mathbf{x_i} \in \mathcal{X}, j = 1, \ldots, k$, s.t. $\mathbf{c_j} \neq \mathbf{c_i} \forall i \neq j$
end if
synchronize threads
repeat
  for all $\mathbf{x_i} \in \mathcal{X}_{threadId}$ **do**
    $l_i \leftarrow \arg\min \mathcal{D}(c_j, x_i)$
  end for
  synchronize threads
  if threadId=0 **then**
    for all $\mathbf{x_i} \in \mathcal{X}$ **do**
      $\mathbf{c_{l_i}} \leftarrow \mathbf{c_{l_i}} + \mathbf{x_i}$
      $m_{l_i} \leftarrow m_{l_i} + 1$
    end for
    for all $\mathbf{c_j} \in \mathcal{C}$ **do**
      $\mathbf{c_j} \leftarrow \frac{1}{m_j}\mathbf{c_i}$
    end for
    if convergence **then**
      signal threads to terminate
    end if
  end if
until convergence

2.3 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. [14] provide a nice description of contemporary NVIDIA GPU architectures.

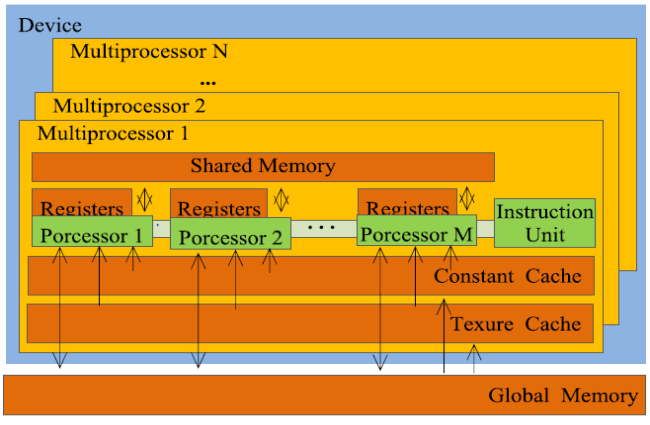

Figure1: GPU Architecture

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 re-converging. 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 [5] or Maspar [2] 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.

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 very 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 request 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.

2.4 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.

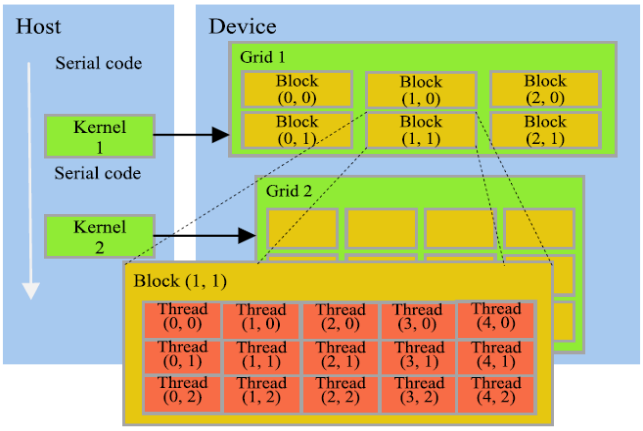

Figure 2: CUDA Architecture

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.

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.

3. LITERATURE REVIEW

According to literature survey we found Several implementations of the popular K-means data clustering algorithm for GPUs exist[6]. One common restriction of the published approaches, however, is the limitation of reported dimensionality of test data to small values of 60 or below. To the best of our knowledge there are two GPU implementations of k-means are available which taken as a benchmark for all comparisons. We have studied both the approaches and found some observations as follows:

3.1 GPUMiner

GPUMiner stores all input data in the global memory, and loads *k* centroids to the shared memory. Each block has 128 threads, and the grid has *n*/128 blocks. The main characteristic of *GPUMiner* is the design of a bitmap. The workflow of *GPUMiner* is as follows. First, each thread calculates the distance from one data point to every centroid, and changes the suitable bit into true in the bit array, which stores the nearest centroid for each data point. Second, each thread is responsible for one centroid, finds all the corresponding data points from the bitmap and takes the mean of those data points as the new centroids.

The main problem of *GPUMiner* is the poor utilization of memory in GPU, since *GPUMiner* accesses most of the data (input data point) from global memory directly. As pointed out in [9], bitmap approach is elegant in expressing the problem, but not a good method for high performance, since bitmap takes more space when *k* is large and requires more shared memory.

3.2 UV_k-Means

In order to avoid the long time latency of global memory access, *UV_k-Means* copies all the data to the texture memory, which uses a cache mechanism. Then, it uses constant memory to store the *k* centroids, which is also more efficient than using global memory. Each thread is responsible for finding the nearest centroid of a data point; each block has 256 threads, and the grid has *n*/256 blocks[7].

The work flow of *UV_k-Means* is straightforward. First, each thread calculates the distance from one corresponding data point to every centroid and finds the minimum distance and corresponding centroid. Second, each block calculates a temporary centroid set based on a subset of data points, and each thread calculates one dimension of the temp centroid. Third, the temporal centroid sets are copied from GPU to CPU, and then the final new centroid set is calculated on CPU.

UV_k-Means has achieved a speedup of twenty to forty over the single-thread CPU-based *k*-Means in our experiment. This speedup is mainly achieved by assigning each data point to one thread and utilizing the cache mechanism to get a high reading efficiency. However, the efficiency could be further improved by other memory access mechanisms such as registers and shared memory.

Some other implementations of k-means are also shows considerable speedup on GPU like Hall and Hart propose two theoretical options for solving the problem of limited instance counts and dimensionality: multi-pass labeling and a different data layout within the texture [8]. None of the approaches have been implemented though. In addition to the naive k-means implementation the data is reordered to minimize the number of distance calculations by only calculating the metrics to the nearest centroids. This is achieved by finding those centroids by traversing a previously constructed kd-tree. The authors could not observe any problems caused by the non standard compliant floating point arithmetic implementations on the GPU, stating that the exact same clustering have been found.

The approach of Cao et. al. in [6] differs in that the centroid indices are stored in an 8-bit stencil buffer instead of the frame buffer limiting the number of total centroids to 256. Limitations in dimensionality and instance counts due to maximum texture sizes are solved via a costly multi-pass approach. No statements concerning precision of the GPU version were made.

4. DISCUSSION

Analyzing the various papers on K-Means on GPU we can deduce:

- 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 presented implementation is only limited in the available memory on the GPU and therefore scales well.
- 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. 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] 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*.

[2] 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*.

[3] 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 (UCHPC-MAW '09), 2009, doi: 10.1145/1531666.1531668.*

[4] 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*)

[5] H. 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*)

[6] 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*

[7] 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*.

[8] 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*

[9] 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.*

[10] You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu,"Speeding up K-Means Algorithm by GPUs".

Author Profile

Mrs. B. Padmavathi – Asst. Professor in Computer Engineering Department **,** G.H.Raisoni College of Engineering & Mgnt, Wagholi, Pune, Pune University. India.

Rahul Ghudji received B.E. degree in Computer Science and Engineering from H. V. P. M. College Of Engineering, Amravati. He is pursuing M.E. at G. H. Raisoni College Of Engineering and Management, Wagholi, Pune.